U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Data Link Protocol

Redes de Computadores (RC)

2024/2025

Class 13

Miguel Lopes Guerrinha - up202205038

Tomás da Silva Custódio Teixeira - up202208041

Summary

This project was developed for the Computer Networks ("Redes de Computadores") class of 2024/2025 and focuses on the development and evaluation of a custom data link application layer protocol for reliable file transmission over an RS-232 serial connection. In this project we implemented a data link protocol with some functionalities such as framing, error detection and retransmission using a Stop & Wait approach.
Through this project, we developed a deeper understanding of the complexities involved in designing and implementing communication protocols. We learned how different layers interact to ensure reliable data transfer, and how design choices, such as packet size and error handling strategies, can significantly impact performance. Overall, this project not only reinforced my theoretical knowledge of networking but also enhanced my ability to apply these concepts in a practical setting.

Introduction

The objective of this project is to develop a robust data link and application layer protocol for reliable data transmission over an RS-232 serial connection. The data link layer ensures error detection, retransmission, and flow control using a Stop & Wait protocol, while the application layer manages data segmentation, control messaging, and session handling. Through this project, the goal is to gain a deeper understanding of communication protocols, their implementation challenges, and the trade-offs between efficiency and reliability.
This report is structured to provide a comprehensive overview of the project.
We begin with the **Architecture** section that outlines the main components of the system, including the data link layer, application layer, and auxiliary modules. Then we examine the **Code Structure** that investigates the design of the main functions, APIs, and data structures used to implement the protocol. The **Main Use Cases** section details the typical operations of the system, including connection setup, data transmission, and session termination. After that, in the **Logical Link Protocol** and **Application Protocol** we discuss the implementation strategies for each layer.
In the **Validation** section, we present the tests conducted to evaluate the system's reliability and the results obtained. In the **Data Link Protocol Efficiency** section, the protocol's performance is analyzed when we change some parameters of the protocol like baud rate, packet size and frame error rate.
Finally, the **Conclusions** synthesize the key findings and reflect on the learning outcomes of the project.

Architecture

Functional Blocks

The system architecture is divided into two layers, the application layer and the data link layer:

- **Application Layer**: Implemented in *application_layer.c*, if in transmitter mode, it fragments the file into packets and invokes the data link layer functions for transmission. If in receiver mode, it receives the data packets parsing the data to construct the file we are receiving.

- **Data Link Layer**: Implemented in *link_layer.c*, it manages packet framing, error control, and the Stop & Wait protocol for retransmissions. It offers an API with functions like llopen, llclose, llwrite, and llread.

This two layers interact with each other and with some auxiliary modules:

- **Serial Port**: Implemented in *serial_port.c*, provides functions to open and close the serial port and to send and receive bytes.
- **Alarm**: Implemented in *alarm.c*, manages timing and retransmission control, essential for maintaining protocol timing.
- **Packets Management**: Implemented in *manage_packet.c*, provides functions to assemble, parse, and verify packets, simplifying data handling.
- **State Machines**: Implemented in *state_machine.c*, implements protocol state transitions, ensuring that the protocol behaves consistently with state machine specifications for data link communication.

Interfaces

In order to transfer the desired data through this serial connection, two terminals are required, one for the receiver and another one for the transmitter.
Having this two terminals we just need to run this commands:

```
Unset
// for the transmitter
$ make run_tx
// for the receiver
$ make run_rx
```

Code Structure

APIs

The API functions are implemented in the data link layer where we use this functions:

```c
C/C++
// Establishes the connection
int llopen(LinkLayer connectionParameters)
// Sends data packets with error handling
int llwrite(const unsigned char *buf, int bufSize)
// Receives packets, performing error checks and acknowledgments
int llread(unsigned char *packet)
// Closes the connection
int llclose(int showStatistics)
```

This functions use some auxiliary functions from the auxiliary modules:

- **Serial Port** functions to manage the function of the serial port

```c
C/C++
// Open and configure the serial port
int openSerialPort(const char *serialPort, int baudRate);
```

```
// Restore original port settings and close the serial port.
int closeSerialPort();
// Wait up to 0.1 second (VTIME) for a byte received from the serial port
int readByteSerialPort(unsigned char *byte);
// Write up to numBytes to the serial port
int writeBytesSerialPort(const unsigned char *bytes, int numBytes);
```

- **Alarm** functions to support the Stop & Wait protocol:

```
C/C++
// Alarm function handler
void alarmHandler(int signal)
// Set up a signal handler for the SIGALRM signal
void setSignal()
// Set an alarm with a specific duration
void setAlarm(int t)
```

- **State Machine** functions that implements protocol state transitions:

```
C/C++
// State machine to establish connection
int state_machine_connection(unsigned char byte, LinkLayerRole role)
// State machine to end connection
int state_machine_end_connection(unsigned char byte, volatile int receiver)
// Transmitter state machine to confirm supervision frames
int state_machine_transmitter(unsigned char byte)
// Receiver state machine to confirm information frames
int state_machine_receiver(unsigned char byte, unsigned char *packet)
```

Data Structures

In this project we used some data structures to help us with efficiency and code organization:

```
C/C++
// Identifies whether it is a Receiver or Transmitter
typedef enum
{
    LlTx,
    LlRx,
} LinkLayerRole;
// Saves the connection parameters
typedef struct
{
    char serialPort[50];
    LinkLayerRole role;
    int baudRate;
    int nRetransmissions;
    int timeout;
} LinkLayer;
```

Main functions

The main function of this project is implemented in the application layer:

```
C/C++
void  applicationLayer(const  char  *serialPort,  const  char  *role,  int
baudRate, int nTries, int timeout, const char *filename)
```

This is the function that is called in the interface and does all the steps necessary to implement the serial communication to send and receive data using this protocol. Here we communicate with the **Data Link Layer** to check the connection between both terminals, send data, receive data and close the connection.
To construct the respective packets to send we create some auxiliary functions in the auxiliary module **Packets Management**:

```
C/C++
// manage_packet.c functions

// Construction of a control Packet
unsigned  char  *getControlPacket(const  unsigned  int  control,  const  char
*filename, unsigned int filesize, unsigned int *bufSize);

// Get all the data from the file to be transfered
unsigned char *getData(FILE *file, size_t fileSize);

// Construction of the data Packet
unsigned  char  *getPacketData(unsigned  int  sequence,  unsigned  char  *data,
unsigned int dataSize, unsigned int *packetSize);

// Parse the control Packet and take all the necessary information
size_t parseControlPacket(unsigned char *packet);
```

Main Use Cases

As we said before, this program has two modes, the transmitter mode that will send the data through the serial port, and the receiver mode that will receive and parse the data to construct the file sent.

Transmitter

1. **llopen()**, used to establish the connection between the transmitter and the receiver, this connection is established by first opening the serial port calling, **openSerialPort()**, and then sending and receiving some control packets.
2. **getControlPacket()**, create a control packet that will send information about the file we will send, like name and size.

3. **llwrite()**, creates an information frame with the control packet and sends it through the serial port.
4. **getData()**, returns all the data of the file to be transmitted.
5. **llwrite()**, creates an information frame with the packet received in the argument and sends it through the serial port.
6. **getControlPacket()**, create a control packet that will send the information that all data has been sent.
7. **llwrite()**, creates an information packet with the control packet and sends it through the serial port.
8. **llcose()**, used to end the connection between the transmitter and the receiver by exchanging some control packets.

Receiver

1. **llopen()**, used to establish the connection between the transmitter and the receiver, this connection is established by first opening the serial port calling, **openSerialPort()**, and then sending and receiving some control packets.
2. **llread()**, used to read the start packet and send the supervision frame.
3. **parseControlPacket()**, returns the size of the packet to be received.
4. **llread()**, called multiple times to read all the information frames received in the serial port.
5. **llclose()**, used to end the connection between the transmitter and the receiver.

Logical Link Protocol

The logical link protocol is the layer that interacts directly with the serial port. It begins with the establishment of connection between the transmitter and the receiver by both calling the **llopen()** function, in this function the transmitter sends a **SET** supervision frame to the receiver. The receiver will receive this frame and if the frame was successfully received without errors, it will respond, sending a **UA** supervision frame. This exchange of information frames ensures both devices are synchronized and ready for data transfer.
To send information we use the **llwrite()** function. This function will create an information frame with a specific format and apply stuffing to the data that we will transmit. This stuffing process consists in preventing the receiver from reading a false flag inside the frame. Specifically, any byte equal to **0x7E** is converted to **0x7D 0x5E**, and any byte equal to **0x7D** is converted to **0x7D 0x5D**. This transformation ensures that the receiver correctly interprets the frame and doesn't read false flags.
On the receiver side we use the **llread()** function to read all data sent through the serial port by the transmitter. In this function, the receiver reads the received information frame, distuffs the data, and checks for any errors in the frame received by checking the **BCC1** and the **BCC2** fields. Based on the frame's integrity the receiver sends a supervision frame back to the transmitter, indicating a successful or unsuccessful data reception by adjusting the control field accordingly. If the frame is rejected, the transmitter resends it until it is accepted or the maximum number of retransmissions is reached. if the transmitter doesn't receive any confirmation frame before an alarm timeout occurs it retransmittes the frame.
When all the data has been successfully transmitted or the maximum number of retransmissions is reached, the connection must be closed. To do this we call the function **llcose()**. In this function, the transmitter sends a **DISC** supervision frame and waits for the

receiver response with the same **DISC** supervision frame. After this frame exchange, the transmitter sends a final **UA** frame, effectively closing the connection.

Application Protocol

The application protocol layer serves as the interface closest to the user, facilitating direct interaction with the application and managing the file to be transmitted. Here the user can select the file to be transferred, the serial port to use, the baud rate of the transmission, the size of the data packets, the maximum number of retransmissions and the time that the transmitter will wait until receive a response from the receiver. The application layer utilizes the link layer API, where the information frames are created to send reliable information and guarantee a safe and successful transmission.
After the establishment of the connection, a packet is sended by the transmitter with a TLV (Type, Length, Value) format, this frame is created by calling the **getControlPacket()** function where we pass the name of the original file and the size of it. On the receiver side this packet is parsed calling the function **parseControlPacket()** and with that initial information we can confirm if the file was totally received or not.
Knowing this information we are prepared to send the actual file. First, all the data of the file was copied to a local buffer by calling the **getData()** function, then all the data is fragmented into small fragments of data with a specific number of bytes. Each fragment is encapsulated in a data packet using the **getDataPacket()** function and then transmitted over the serial port via the **llwrite()** API function. On the receiver side, each data packet is processed individually by the **llread()** API function and if the packet is valid the information is extracted and written on the new file.
The connection ends when the transmitter sends a packet with the information that the file was totally transferred and both, the receiver and the transmitter, call the **llclose()** API function.

Validation

To ensure the reliability of the protocol we have done some tests with different conditions:

1. **Normal File Transfer**
    - **Goal**: Test if the system works properly in normal conditions
    - **Results**: The file was sent successfully and without any errors, demonstrating that the communication protocol works well in normal conditions
2. **Transmitter maximum number of timeouts**
    - **Goal**: Verify the system behavior when the transmitter operates alone, reaching the maximum number of retransmissions.
    - **Results**: The transmitter effectively manages repeated timeouts, demonstrating robust error handling by terminating the transmission after reaching the maximum number of retransmissions.
3. **Interrupted connection**
    - **Goal**: Simulate an occasional loss of connection during transmission. This was simulated by opening the connection between transmitter and receiver, waiting for one timeout and reconnecting the connection.
    - **Results**: Almost all cases of interruption were successfully resolved and the transmission ended without any errors. Only one case wasn't handled, that

was when a frame was received correctly and the confirmation frame was sent by the receiver but the transmitter doesn't receive this frame so it ressent the same packet causing an infinite loop in the transmission.

4. **Error injection**
    - **Goal**: Introduce some noise in the connection that causes errors in the frame to test if all the verifications are working correctly and the rejection of frames are handled.
    - **Results**: The receiver successfully recognizes an error in the frame and rejects the frame causing a retransmission of the correct frame, ensuring that the file was sent properly.

Data Link Protocol Efficiency

To test the efficiency of the protocol's we analyze the time taken to send a file of **10968** bytes and then we calculate the received data rate and the efficiency of the transmission.

Baud Rate Variation

For this experiment the packet size was fixed at **256 bytes** to analyze the impact of the baud rate on transfer time, received data rate and efficiency.

Baud Rate (bits/s)	Average Time (s)	Received Data Rate (bits/s)	Efficiency (%)
1200	97.988	895.456	74.62
2400	48.995	1790.876	74.61
4800	24.496	3581.972	74.62
9600	12.249	7163.360	74.62
19200	6.124	14327.890	74.62
38400	3.062	28655.780	74.62
57600	2.042	42969.638	74.60
115200	1.021	85939.275	74.60

The data presented in the table demonstrates that when we increase the baud rate, the average time of the transmission reduces significantly and improves the received data rate. However the efficiency of the protocol remains almost the same for all baud rates, this indicates that the Stop & Wait protocol effectively utilizes the available bandwidth, but its efficiency is inherently limited by its design which involves waiting for acknowledgments before sending the next frame.

Packet Size Variation

For this experiment the baud rate utilized was fixed at **9600 (bits/s)** to analyze the impact of the packet size variation on transfer time, received data rate and efficiency.

Packet Size (bytes)	Average Time (s)	Received Data Rate (bits/s)	Efficiency (%)
256	12.249	7163.360	74.62
512	11.921	7360.456	76.67
1024	11.749	7468.210	77.79
2048	11.670	7518.766	78.32

As we can see from the table above, the average transmission time decreases slightly as we increase the size of the packets, and both the received data rate and efficiency have a small improvement. The results indicate that using larger packet sizes enhances overall protocol efficiency and data throughput, demonstrating the benefits of optimizing packet size in Stop & Wait protocols.

Frame Error Rate Variation

For this experiment the baud rate utilized was fixed at **9600 bits/s** and the packet size at **256 bytes** to analyze the impact of different framer error rates on transfer time, received data rate and efficiency.

Frame Error Rate (%)	Average Time (s)	Received Data Rate (bits/s)	Efficiency (%)
18.5	13.66	6423.426	66.91
33.6	23.044	3807.672	39.66
55.9	28.831	3043.391	31.70

The table illustrates the impact of varying frame error rates (FER) on the performance of the protocol. As FER increases, the transmission time rises significantly due to the need for more retransmissions. As a result the receive data rate and efficiency decrease drastically. This demonstrates that the protocol is very sensitive to frame errors and higher frame error rates lead to a slower and less efficient data transmission. At higher frame rates that demonstrate above we are not available to test the protocol because it always leads to the only error case we don’t handle, and it's described in the validation section.

Conclusions

This project provided valuable insights into the design and implementation of communication protocols, particularly at the data link and application layers. Through the development of a reliable file transmission protocol, we gained a deeper understanding of essential concepts such as error detection, retransmission mechanisms, and flow control. Overall, this project reinforced both theoretical knowledge and practical application of our understanding about protocol design, providing a solid foundation for exploring more advanced networking techniques in the future.

Appendices

Appendix 1 - **application_layer.c**

```
C/C++
// Application layer protocol implementation

#include "application_layer.h"
#include "link_layer.h"

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/stat.h>

#define DATA_SIZE 256

#include "manage_packet.h"

// Application layer main function.
void applicationLayer(const char *serialPort, const char *role, int
baudRate,
                      int nTries, int timeout, const char *filename)
{

    LinkLayerRole linkLayerRole;

    if (strcmp(role, "tx") == 0) {
        linkLayerRole = LlTx;
    }
    else if (strcmp(role, "rx") == 0) {
        linkLayerRole = LlRx;
    }
    else {
        printf("Wrong role\n");
    }

    LinkLayer connectionParam;

    strcpy(connectionParam.serialPort, serialPort);
    connectionParam.role = linkLayerRole;
    connectionParam.baudRate = baudRate;
    connectionParam.nRetransmissions = nTries;
    connectionParam.timeout = timeout;

    printf("LLOPEN\n");

    if (llopen(connectionParam) == -1) {
        perror("Erro: Connection Failed\n");
        exit(-1);
    }

    printf("Connection Established\n");
    FILE* file;
    FILE* newFile;
    unsigned char *packet;
    int STOP;

    switch (connectionParam.role)
    {
    case LlTx:
        file = fopen(filename, "rb");

        if (file == NULL) {
            perror("File not found\n");
            exit(-1);
        }

        struct stat fileStat;

        if (stat(filename, &fileStat) == -1) {
            perror("Error: Could not retrieve file information\n");
            exit(-1);
        }

        printf("File size: %ld bytes\n", fileStat.st_size);

        unsigned int bufSize;

        // Get the control packet to establish connection
        unsigned char *controlStart = getControlPacket(1, filename,
fileStat.st_size, &bufSize);

        // Call llwrite to send the control packet
        if (llwrite(controlStart, bufSize) == -1) {
            perror("Error: Start packet error\n");
            exit(-1);
        }

        unsigned int sequence = 0;

        // Get all content of the file we want to send
        unsigned char *content = getData(file, fileStat.st_size);
        size_t bytesLeft = fileStat.st_size;

        // Loop to send all the content of the file
        while (bytesLeft > 0)
        {
            int dataSize = bytesLeft > (size_t) DATA_SIZE ? DATA_SIZE :
bytesLeft;
            printf("--------------------\n");
            printf("Bytes Left %ld de %ld\n", bytesLeft, fileStat.st_size);
            printf("--------------------\n");
            unsigned char *data = (unsigned char *) malloc(dataSize);
            memcpy(data, content, dataSize);
            unsigned int packetSize;

            // Construct the packet
            unsigned char *packet = getPacketData(sequence, data, dataSize,
&packetSize);

            if (llwrite(packet, packetSize) == -1) {
                perror("Error: error in data packets\n");
                exit(-1);
            }

            bytesLeft -= dataSize;
            content += dataSize;
            sequence = (sequence + 1) % 100;
        }

        // Get the control packet to close connection
        unsigned char *controlEnd = getControlPacket(3, filename,
fileStat.st_size, &bufSize);
        printf("END PACKET\n");

        // Call llwrite to send control packet
        if (llwrite(controlEnd, bufSize) == -1) {
            perror("Error: End packet error\n");
            exit(-1);
        }

        int close = llclose(1);

        if (close == -1) {
            perror ("Error closing serial port\n");
            exit(-1);
        }

        if (close == 1) {
            perror("End connection failed\n");
            exit(-1);
        }
        break;
```

```
case LlRx:

    packet = (unsigned char *) malloc(DATA_SIZE+4);
    int packetSize = -1;
    STOP = FALSE;

    // Loop to receive start packet
    while (!STOP)
    {
        packetSize = -1;
        packetSize = llread(packet);
        if (packetSize > 0 && packet[0] == 1) {
            printf ("START PACKET RCV\n");
            STOP = TRUE;
        }
    }

    // Get the size of the file we will receive
    size_t fileSize = parseControlPacket(packet);
    size_t totalReceived = 0;

    newFile = fopen((char *) filename, "wb+");
    STOP = FALSE;
    int sequenceConfirm = -1;

    // Loop to receive all data
    while (!STOP)
    {
        packetSize = -1;
        packetSize = llread(packet);

        // End packet received
        if (packet[0] == 3) {
            printf("END PACKET RCV\n");
            if (llclose(1) == -1) {
                perror ("Error closing serial port\n");
                exit(-1);
            }
            STOP = TRUE;
        }
        else if (packetSize > 0 && packet[0] == 2) {
            totalReceived += packetSize-4;
            printf("----------------\n");
            printf("Progression %ld/%ld\n", totalReceived, fileSize);
            printf("----------------\n");
            if (packet[1] == sequenceConfirm) {
                perror("Error duplicated frame received");
                exit(-1);
                }
                sequenceConfirm = packet[1];
                fwrite(packet+4, sizeof(unsigned char), packetSize-4,
newFile);
            }
            else continue;
        }

        fclose(newFile);
        break;

    default:
        exit(-1);
        break;
    }

}
```

Appendix 2 - **link_layer.c**

C/C++
```
// Link layer protocol implementation

#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <termios.h>
#include <unistd.h>
#include "alarm.h"
#include "link_layer.h"
#include "serial_port.h"
#include "state_machine.h"

// Baudrate settings are defined in <asm/termbits.h>, which is
// included by <termios.h>
#define BAUDRATE B38400
#define _POSIX_SOURCE 1 // POSIX compliant source

#define FALSE 0
```

```c
#define TRUE 1

#define BUF_SIZE 5

volatile int STOP = FALSE;
extern volatile int alarmEnabled;
extern int alarmCount;
extern state s;

int nRetransmissions;
extern int nRetransmissionsTotal;
int timeout;
volatile int receiver = -1;

unsigned int frameNumber;
int i = 0;

////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters)
{
    nRetransmissions = connectionParameters.nRetransmissions;
    nRetransmissionsTotal = 0;
    timeout = connectionParameters.timeout;

    int fd = openSerialPort(connectionParameters.serialPort,
connectionParameters.baudRate);
    if (fd < 0) {
        return -1;
    }


    printf("New termios structure set\n");

    STOP = FALSE;

    if (connectionParameters.role == LlTx) {
        receiver = FALSE;

        unsigned char buf_set[BUF_SIZE] = {FLAG, A_SENDER, C_SET,
A_SENDER^C_SET, FLAG};
        unsigned char buf_ua[BUF_SIZE];
        setSignal();

        // Loop for write the Transmitter Supervision Frame every 3 seconds
until receive the right Receiver Supervision Frame
        while (STOP == FALSE && alarmCount <= nRetransmissions)
        {
            if (alarmEnabled == FALSE) {
                int bytes = writeBytesSerialPort(buf_set, BUF_SIZE);
                printf("%d bytes written\n", bytes);
                setAlarm(timeout);
            }

            s = START;

            while (s != STOP_RCV && alarmEnabled == TRUE)
            {
                int bytes = readByteSerialPort(buf_ua);
                if (bytes > 0)
                    if (state_machine_connection(buf_ua[0],
connectionParameters.role)) {
                        alarmEnabled = FALSE;
                        break;
                    }
            }

            if (s == STOP_RCV) {
                STOP = TRUE;
            }
        }
    }
    else if (connectionParameters.role == LlRx) {
        receiver = TRUE;
        unsigned char buf_set[BUF_SIZE];

        s = START;

        // Loop for read the right Transmitter Supervision Frame
        while (s != STOP_RCV)
        {
            int bytes = readByteSerialPort(buf_set);
            if (bytes > 0) {
                if (state_machine_connection(buf_set[0],
connectionParameters.role)) {
                    continue;
                }
            }
        }

        // Write the confirmation Supervision Frame to confirm the
connection establishment
        unsigned char buf_ua[BUF_SIZE] = {FLAG, A_RECEIVER, C_UA,
A_RECEIVER^C_UA, FLAG};
        int bytes = writeBytesSerialPort(buf_ua, BUF_SIZE);
        printf("%d bytes written\n", bytes);
    }
    if (alarmCount > nRetransmissions) {
        return -1;
    }
    frameNumber = 0;
    return fd;
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(const unsigned char *buf, int bufSize)
{
    int frameSize = 6 + bufSize;
    unsigned char *frame = (unsigned char*)malloc(frameSize);
    if (frame == NULL) {
        perror("Memory allocation failed\n");
        return -1;
    }

    // Config the transmitter frame
    frame[0] = FLAG;
    frame[1] = A_SENDER;
    if (frameNumber % 2 == 0) {
        frame[2] = C_0;
    }
    else if (frameNumber % 2 == 1) {
        frame[2] = C_1;
    }
    frame[3] = A_SENDER ^ frame[2];
    memcpy(frame+4, buf, bufSize);

    // Construction of the BCC2 with original data
    unsigned char BCC2 = buf[0];
    for (unsigned int i = 1; i < bufSize; i++) {
        BCC2 ^= buf[i];
    }

    // Aplying stuffing in data
    int j = 4;
    for (unsigned int i = 0; i < bufSize; i++) {
        if (buf[i] == FLAG) {
            frame = realloc(frame, frameSize+1);
            frame[j++] = ESC;
            frame[j++] = FLAG ^ STUFF;
            frameSize++;
        }
        else if (buf[i] == ESC) {
            frame = realloc(frame, frameSize+1);
            frame[j++] = ESC;
            frame[j++] = ESC ^ STUFF;
            frameSize++;
        }
        else {
            frame[j++] = buf[i];
        }
    }

    if (BCC2 == FLAG) {
        frame = realloc(frame, frameSize+1);
        frame[j++] = ESC;
        frame[j++] = FLAG ^ STUFF;
        frameSize++;
    }
    else if (BCC2 == ESC) {
        frame = realloc(frame, frameSize+1);
        frame[j++] = ESC;
        frame[j++] = ESC ^ STUFF;
        frameSize++;
    }
    else {
        frame[j++] = BCC2;
    }

    frame[j] = FLAG;

    alarmEnabled = FALSE;
    alarmCount = 0;

    unsigned char buf_rc[BUF_SIZE];

    STOP = FALSE;

    int bytesWritten;

    // Write a frame and wait for the response of the receiver
    // If the Transmitter receive a Supervision Frame with a RR0 or RR1
control byte, conitue to the next frame, otherwise send again the same frame
    // If the Transmitter doens't receive nothing in 3 seconds send the
frame again
    while (STOP == FALSE && alarmCount <= nRetransmissions)
    {
        if (alarmEnabled == FALSE) {
            bytesWritten = writeBytesSerialPort(frame, frameSize);
            setAlarm(timeout);
        }

        s = START;

        while (alarmEnabled == TRUE && s != STOP_RCV)
        {
            int bytes = readByteSerialPort(buf_rc);
            if (bytes > 0) {
                alarmCount = 0;
                if (state_machine_transmitter(buf_rc[0])) {
                    alarmEnabled = FALSE;
                    break;
                }
            }
        }

        if (s == STOP_RCV) {
            STOP = TRUE;
        }
    }
    if (alarmCount >= nRetransmissions) {
        return -1;
    }
    frameNumber++;
    return bytesWritten;
}

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
int llread(unsigned char *packet)
{
    s = START;
    unsigned char buf_rc[BUF_SIZE];
    unsigned char buf[BUF_SIZE];
    buf[0] = FLAG;
    buf[1] = A_RECEIVER;
    i = 0;

    // Wait until a frame is received
    // If the frame is received with correct information send a Supervision
frame with the RR1 or RR0 control byte, otherwise send the frame with REJ0
or REJ1 control byte
    while (s != STOP_RCV)
    {
        int bytes = readByteSerialPort(buf_rc);

        if (bytes > 0) {
            if (state_machine_receiver(buf_rc[0], packet)) {
                buf[2] = (frameNumber % 2 == 0) ? C_REJ0 : C_REJ1;
                break;
            }
        }
        if (s == STOP_RCV) {
            buf[2] = (frameNumber % 2 == 0) ? C_RR1 : C_RR0;
            frameNumber++;
        }
    }

    buf[3] = (A_RECEIVER ^ buf[2]);
    buf[4] = FLAG;

    // Write the Supervision Frame
    writeBytesSerialPort(buf, BUF_SIZE);

    if (s != STOP_RCV) {
        return -1;
    }
    return i-1;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics)
{
    STOP = FALSE;
    alarmCount = 0;
    alarmEnabled = FALSE;

    if (!receiver) {
        unsigned char buf_disc[BUF_SIZE] = {FLAG, A_SENDER, C_DISC, A_SENDER
^ C_DISC, FLAG};
        unsigned char buf[BUF_SIZE];

        // Loop for write the Transmitter Supervision Frame every 3 seconds
until receive the right Receiver Supervision Frame
        while (STOP == FALSE && alarmCount <= nRetransmissions)
        {
            if (alarmEnabled == FALSE) {
                writeBytesSerialPort(buf_disc, BUF_SIZE);
                setAlarm(timeout);
            }

            s = START;
            while (s != STOP_RCV && alarmEnabled == TRUE)
            {
                int bytes = readByteSerialPort(buf);
                if (bytes > 0) {
                    if (state_machine_end_connection(buf[0], receiver)) {
                        alarmEnabled = FALSE;
                        break;
                    }
                }
            }
            if (s == STOP_RCV) {
                STOP = TRUE;
            }
        }
        unsigned char buf_ua[BUF_SIZE] = {FLAG, A_SENDER, C_UA, A_SENDER ^
C_UA, FLAG};
        writeBytesSerialPort(buf_ua, BUF_SIZE);
        setAlarm(0);

        if (alarmCount > nRetransmissions) {
            return 1;
        }

        setAlarm(0);

        if (showStatistics) {
            printf("STATISICS\n");
            printf("----------------------------\n");
            printf("NUMBER FRAMES SENT --> %d\n", frameNumber);
            printf("----------------------------\n");
            printf("RETRANSMISSIONS    --> %d\n", nRetransmissionsTotal);
            printf("----------------------------\n");
        }
    }
    else if (receiver) {
        unsigned char buf_disc[BUF_SIZE];

        s = START;

        // Loop for read the right Transmitter Supervision Frame
        while (s != STOP_RCV)
        {
            int bytes = readByteSerialPort(buf_disc);
            if (bytes > 0) {
                if (state_machine_end_connection(buf_disc[0], receiver)) {
                    continue;
                }
            }
        }

        STOP = FALSE;
        alarmCount = 0;

        unsigned char buf[BUF_SIZE] = {FLAG, A_RECEIVER, C_DISC, A_RECEIVER
^ C_DISC, FLAG};


        while (STOP == FALSE && alarmCount <= nRetransmissions)
        {
            if (alarmEnabled == FALSE) {
                writeBytesSerialPort(buf, BUF_SIZE);
                setAlarm(timeout);
            }

            s = START;

            while (s != STOP_RCV && alarmEnabled == TRUE)
            {
                int bytes = readByteSerialPort(buf_disc);
                if (bytes > 0) {
                    if (state_machine_end_connection(buf_disc[0], receiver))
{
                        continue;
                    }
                }
            }
            if (s == STOP_RCV) {
                STOP = TRUE;
            }
        }
        if (alarmCount > nRetransmissions) {
            return 1;
        }

        setAlarm(0);

        if (showStatistics) {
            printf("STATISTICS\n");
            printf("----------------------------\n");
            printf("NUMBER FRAMES RECEIVED --> %d\n", frameNumber);
            printf("----------------------------\n");
        }
    }

    int clstat = closeSerialPort();
    return clstat;
```

```
}
```

Appendix 3 - **alarm.c**

```c
C/C++
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include "alarm.h"

volatile int alarmEnabled = FALSE;
int alarmCount = 0;
int nRetransmissionsTotal;

// Alarm function handler
void alarmHandler(int signal)
{
    alarmEnabled = FALSE;
    alarmCount++;
    nRetransmissionsTotal++;

    printf("Alarm #%d\n", alarmCount);
}

void setSignal() {
    (void)signal(SIGALRM, alarmHandler);
}

// Set alarm to be triggered in t secons
void setAlarm(int t)
{
    if (alarmEnabled == FALSE)
    {
        alarm(t);
        alarmEnabled = TRUE;
    }
}
```

Appendix 4 - **manage_packets.c**

```
C/C++
#include "manage_packet.h"

// Construction of a control Packet
unsigned char *getControlPacket(const unsigned int control, const char
*filename, unsigned int filesize, unsigned int *bufSize) {
    int L1;
    unsigned int numBits = 0;
    unsigned int auxFileSize = filesize;

    if (auxFileSize == 0) {
        L1 = 1;
    }

    while (auxFileSize > 0)
    {
        numBits++;
        auxFileSize >>= 1;
    }

    L1 = (numBits + 7) / 8;

    int L2 = strlen(filename);
    *bufSize = 5 + L1 + L2;

    unsigned char *packet = (unsigned char*) malloc(*bufSize);

    unsigned int pos = 0;

    packet[pos++] = control;
    packet[pos++] = 0;
    packet[pos++] = L1;

    for (unsigned int i = 0; i < L1; i++) {
        packet[2+L1-i] = filesize & 0xFF;
        filesize >>=8;
    }

    pos += L1;
    packet[pos++] = 1;
    packet[pos++] = L2;
    memcpy(packet+pos, filename, L2);

    return packet;
}

// Get all the data from the file to be transfered
unsigned char *getData(FILE *file, size_t fileSize) {
    unsigned char *content = (unsigned char *) malloc (fileSize);
```

```c
    size_t bytesRead = fread(content, 1, fileSize, file);
    if (bytesRead != fileSize) {
        perror ("Error reading file");
        free(content);
        return NULL;
    }

    return content;
}

// Construction of the data Packet
unsigned char *getPacketData(unsigned int sequence, unsigned char *data,
unsigned int dataSize, unsigned int *packetSize) {
    *packetSize = 4 + dataSize;
    unsigned char *packet = (unsigned char *) malloc(*packetSize);

    packet[0] = 2;
    packet[1] = sequence;
    packet[2] = dataSize >> 8 & 0xFF;
    packet[3] = dataSize & 0xFF;
    memcpy(packet+4, data, dataSize);

    return packet;
}

// Parse the control Packet and take all the necessary information
size_t parseControlPacket(unsigned char *packet) {
    unsigned char nFileSizeBytes = packet[2];
    unsigned char fileSizeAux[nFileSizeBytes];
    memcpy(fileSizeAux, packet+3, nFileSizeBytes);

    size_t fileSize = 0;
    for (unsigned int i = 0; i < nFileSizeBytes; i++) {
        fileSize |= (fileSizeAux[nFileSizeBytes-i-1] << (8*i));
    }
    return fileSize;
}
```

Appendix 5 - **state_machine.c**

```c
C/C++
#include "state_machine.h"
#include <stdio.h>
```

```c
state s;
extern unsigned int frameNumber;
extern int i;
unsigned char bcc2;
unsigned char bcc2_rcv;

// State machine to confirm the bytes received from the connection fase
int state_machine_connection(unsigned char byte, LinkLayerRole role) {
    if (role == LlRx) {
        switch (s)
        {
            case START:
                if (byte == FLAG) {
                    s = FLAG_RCV;
                }
                break;

            case FLAG_RCV:
                if (byte == A_SENDER) {
                    s = A_RCV;
                }
                else if (byte == FLAG) {
                    s = FLAG_RCV;
                }
                else {
                    s = START;
                    printf("ADDRESS FAILED\n");
                    return 1;
                }
                break;

            case A_RCV:
                if (byte == C_SET) {
                    s = C_RCV;
                }
                else if (byte == FLAG) {
                    s = FLAG_RCV;
                }
                else {
                    s = START;
                    printf("CONTROL FAILED\n");
                    return 1;
                }
                break;

            case C_RCV:
                if (byte == (A_SENDER ^ C_SET)) {
                    s = BCC_OK;
                }
                else if (byte == FLAG) {
                    s = FLAG_RCV;
                }
                else {
                    s = START;
                    printf("BCC FAILED\n");
                    return 1;
                }
                break;

            case BCC_OK:
                if (byte == FLAG) {
                    s = STOP_RCV;
                }
                else {
                    s = START;
                    printf("WRONG FINAL FLAG\n");
                    return 1;
                }
                break;
            default:
                s = START;
                return 1;
                break;
        }
    }
    else if (role == LlTx) {
        switch (s)
        {
        case START:
                if (byte == FLAG) {
                    s = FLAG_RCV;
                }
                break;

            case FLAG_RCV:
                if (byte == A_RECEIVER) {
                    s = A_RCV;
                }
                else if (byte == FLAG) {
                    s = FLAG_RCV;
                }
                else {
                    s = START;
                    printf("ADDRESS FAILED\n");
                    return 1;
                }
                break;

            case A_RCV:
                if (byte == C_UA) {
                    s = C_RCV;
                }
                else if (byte == FLAG) {
                    s = FLAG_RCV;
                }
                else {
                    s = START;
                    printf("CONTROL FAILED\n");
                    return 1;
                }
                break;

            case C_RCV:
                if (byte == (A_RECEIVER ^ C_UA)) {
                    s = BCC_OK;
                }
                else if (byte == FLAG) {
                    s = FLAG_RCV;
                }
                else {
                    s = START;
                    printf("BCC FAILED\n");
                    return 1;
                }
                break;

            case BCC_OK:
                if (byte == FLAG) {
                    s = STOP_RCV;
                }
                else {
                    s = START;
                    printf("WRONG FINAL FLAG\n");
                    return 1;
                }
                break;
            default:
                break;
        }
    }
    return 0;
}
```

```
// State machine to confirm the bytes received from the end connection fase
int state_machine_end_connection(unsigned char byte, volatile int receiver)
{
    if (receiver) {
        switch (s)
        {
            case START:
                if (byte == FLAG) {
                    s = FLAG_RCV;
                }
                break;

            case FLAG_RCV:
                if (byte == A_SENDER) {
                    s = A_RCV;
                }
                else if (byte == FLAG) {
                    s = FLAG_RCV;
                }
                else {
                    s = START;
                    printf("ADRESS FAILED\n");
                    return 1;
                }
                break;

            case A_RCV:
                if (byte == C_DISC || byte == C_UA) {
                    s = C_RCV;
                }
                else if (byte == FLAG) {
                    s = FLAG_RCV;
                }
                else {
                    s = START;
                    printf("CONTROL FAILED\n");
                    return 1;
                }
                break;

            case C_RCV:
                if (byte == (A_SENDER ^ C_DISC) || byte == (A_SENDER ^
C_UA)) {
                    s = BCC_OK;
                }
                else if (byte == FLAG) {
                    s = FLAG_RCV;
                }
                else {
                    s = START;
                    printf("BCC FAILED\n");
                    return 1;
                }
                break;

            case BCC_OK:
                if (byte == FLAG) {
                    s = STOP_RCV;
                }
                else {
                    s = START;
                    printf("WRONG FINAL FLAG\n");
                    return 1;
                }
                break;
            default:
                s = START;
                return 1;
                break;
        }
    }
    else if (!receiver) {
        switch (s)
        {
        case START:
                if (byte == FLAG) {
                    s = FLAG_RCV;
                }
                break;

            case FLAG_RCV:
                if (byte == A_RECEIVER) {
                    s = A_RCV;
                }
                else if (byte == FLAG) {
                    s = FLAG_RCV;
                }
                else {
                    s = START;
                    printf("ADDRESS FAILED\n");
                    return 1;
                }
                break;

            case A_RCV:
                if (byte == C_DISC) {
                    s = C_RCV;
                }
                else if (byte == FLAG) {
                    s = FLAG_RCV;
                }
                else {
                    s = START;
                    printf("CONTROL FAILED\n");
                    return 1;
                }
                break;

            case C_RCV:
                if (byte == (A_RECEIVER ^ C_DISC)) {
                    s = BCC_OK;
                }
                else if (byte == FLAG) {
                    s = FLAG_RCV;
                }
                else {
                    s = START;
                    printf("BCC FAILED\n");
                    return 1;
                }
                break;

            case BCC_OK:
                if (byte == FLAG) {
                    s = STOP_RCV;
                }
                else {
                    s = START;
                    printf("WRONG FINAL FLAG\n");
                    return 1;
                }
                break;
            default:
                break;
        }
    }
    return 0;
}

// State machine to confirm the bytes received by the Transmitter
int state_machine_transmitter(unsigned char byte) {
    unsigned char control = (frameNumber % 2 == 0) ? C_RR1 : C_RR0;
    switch (s)
    {
```

```
    case START:
        if (byte == FLAG) {
            s = FLAG_RCV;
        }
        break;

    case FLAG_RCV:
        if (byte == A_RECEIVER) {
            s = A_RCV;
        }
        else if (byte == FLAG) {
            s = FLAG_RCV;
        }
        else {
            s = START;
            printf("ADDRESS FAILED\n");
            return 1;
        }
        break;

    case A_RCV:
        if (byte == control) {
            s = C_RCV;
        }
        else if (byte == FLAG) {
            s = FLAG_RCV;
        }
        else {
            s = START;
            printf("CONTROL FAILED\n");
            return 1;
        }
        break;

    case C_RCV:
        if (byte == (A_RECEIVER ^ control)) {
            s = BCC_OK;
        }
        else if (byte == FLAG) {
            s = FLAG_RCV;
        }
        else {
            s = START;
            printf("BCC FAILED\n");
            return 1;
        }
        break;
    case BCC_OK:
        if (byte == FLAG) {
            s = STOP_RCV;
        }
        else {
            s = START;
            printf("WRONG FINAL FLAG\n");
            return 1;
        }
        break;
    default:
        break;
    }
    return 0;
}

// State machine to confirm the bytes received by the Receiver
int state_machine_receiver(unsigned char byte, unsigned char *packet) {
    unsigned char control = (frameNumber % 2 == 0) ? C_0 : C_1;

    switch (s)
    {
    case START:
        if (byte == FLAG) {
            s = FLAG_RCV;
        }
        break;

    case FLAG_RCV:
        if (byte == A_SENDER) {
            s = A_RCV;
        }
        else if (byte == FLAG) {
            s = FLAG_RCV;
        }
        else {
            s = START;
            printf("ADDRESS FAILED\n");
            return 1;
        }
        break;

    case A_RCV:
        if (byte == control) {
            s = C_RCV;
        }
        else if (byte == FLAG) {
            s = FLAG_RCV;
        }
        else {
            s = START;
            printf("CONTROL FAILED\n");
            return 1;
        }
        break;

    case C_RCV:
        if (byte == (A_SENDER ^ control)) {
            s = DATA_PACKET;
        }
        else if (byte == FLAG) {
            s = FLAG_RCV;
        }
        else {
            s = START;
            printf("BCC1 FAILED\n");
            return 1;
        }
        break;

    case DATA_PACKET:
        if (byte == ESC) {
            s = ESC_RCV;
        }
        else if (byte == FLAG) {
            bcc2_rcv = packet[i-1];
            bcc2 = packet[0];
            for (int j = 1; j < i-1; j++) {
                bcc2 ^= packet[j];
            }
            if (bcc2 == bcc2_rcv) {
                s = STOP_RCV;
                return 0;
            }
            else {
                s = START;
                printf("BCC2 FAILED\n");
                return 1;
            }
        }
        else {
            packet[i] = byte;
            i++;
        }
        break;
    case ESC_RCV:
        s = DATA_PACKET;
        if (byte == (FLAG ^ STUFF)) {
            packet[i] = FLAG;
            i++;
        }
        else if (byte == (ESC ^ STUFF)) {
            packet[i] = ESC;
            i++;
        }
        else {
            s = START;
            printf("ESC ALONE?\n");
            return 1;
        }
        break;

    default:
        s = START;
        break;
    }
    return 0;
}
```