U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Protocolo de Ligação de Dados

(1º Trabalho Laboratorial)

Redes de Computadores · L.EIC025

2023/2024 · 1º Semestre

Turma: 3LEIC08

João Correia up202005015

Miguel Charchalis up201506074

1- Sumário

No âmbito da unidade curricular Redes de Computadores, realizou-se o 1º projeto que nos foi proposto. O projeto “Protocolo de Ligação de Dados” consiste em implementar a funcionalidade de transmissão e receção para permitir transferir um ficheiro armazenado no disco de um computador para um outro computador, estando ligados através de um cabo série.

O nosso principal objetivo no projeto foi alcançado com êxito, uma vez que conseguimos desenvolver com sucesso um protocolo de conexão de dados.

2- Introdução

A elaboração deste relatório, tem o objetivo de explicar de uma forma mais detalhada o funcionamento do protocolo de ligação de dados para a transmissão e receção de ficheiros através de uma porta Série RS-232. A estrutura da documentação é a seguinte:

- **Arquitetura**
  Demonstração dos blocos funcionais e interfaces.
- **Estrutura do código**
  Representação das APIs, as estruturas de dados usadas, as funções e a sua interação com a arquitetura.
- **Casos de uso principais**
  Identificação dos principais casos de uso e sequências da chamada de funções.
- **Protocolo de ligação lógica**
  Identificação dos principais aspetos funcionais e descrição da estratégia de implementação destes aspetos.
- **Protocolo de aplicação**
  Identificação dos principais aspetos funcionais e descrição da estratégia de implementação destes aspetos.
- **Validação**
  Descrição dos testes efetuados com apresentação quantificada dos resultados, se possível.
- **Eficiência do protocolo de ligação de dados**
  Caracterização estatística da eficiência do protocolo.
- **Conclusões**
  Síntese da informação apresentada nas secções anteriores e reflexão sobre os objetivos de aprendizagem alcançados.

3- Arquitetura

Este projeto está dividido em duas camadas: camada da ligação de dados (***link_layer***) e a camada de aplicação (***application_layer***). A camada de ligação de dados é responsável pela delimitação e numeração das tramas, pelo estabelecimento e terminação da ligação às portas série e por controlo de erros e fluxo. A camada de aplicação depende da camada de ligação de dados e é responsável por enviar e receber tramas e processar e transmitir pacotes de dados ou de controlo.

A camada de aplicação está acima da camada de ligação de dados, então não conhece a estrutura interna do protocolo de ligação de dados, apenas conhece o serviço que desempenha. Esse serviço pode ser acedido a partir da interface do utilizador.

A interface do utilizador permite que o utilizador escolha a porta série, a função desempenhada pelo sistema (emissor ou recetor) e o ficheiro que pretende transferir entre os dois computadores. Para tal, a execução do programa é realizada em 2 terminais, um em cada computador, sendo um o transmissor e o outro o recetor.

4- Estrutura do código

4.1- Camada de aplicação (application_layer.c e application_layer.h)

A camada de aplicação é constituída pela função '***application_layer'*** que recebe alguns dados predefinidos e outros introduzidos pelo utilizador. Esses dados são usados para transferir e processar pacotes de dados e de controlo. A camada de ligação dos dados é usada nesta função para comunicar e transferir os ficheiros de acordo com os parâmetros passados. A função 'tlv' é uma função adicional à ***application_layer.***

```
// Application layer main function.
// Arguments:
//   serialPort: Serial port name (e.g., /dev/ttyS0).
//   role: Application role {"tx", "rx"}.
//   baudrate: Baudrate of the serial port.
//   nTries: Maximum number of frame retries.
//   timeout: Frame timeout.
//   filename: Name of the file to send / receive.

void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename);

int tlv(unsigned char *address, int* type, int* length, int** value);
```

4.2- Camada de ligação de dados (link_layer.c e link_layer.h)

Na camada de ligação de dados utilizaram-se três estruturas de dados auxiliares: ***LinkLayer***, onde são caracterizados os parâmetros associados à transferência dos dados, ***LinkLayerRole***, que identifica se o computador está a exercer o papel de recetor ou emissor, ***LinkLayerState***, que identifica o estado da leitura e receção das tramas de informação.

```
typedef struct
{
    char serialPort[50];
    LinkLayerRole role;
    int baudRate;
    int nRetransmissions;
    int timeout;
} LinkLayer;

typedef enum
{
   transmitter,
   receiver,
} LinkLayerRole;
```

```
typedef enum
{
   START,              //0
   FLAG_RCV,           //1
   A_RCV,              //2
   C_RCV,              //3
   BCC1_OK,            //4
   STOP_STATE,         //5
   DATA_FOUND_ESC,     //6
   DATA,               //7
   DISCONNECTED,       //8
   BCC2_OK             //9
} LinkLayerState;
```

As principais funções desta camada são a seguintes:

- ***llopen*** **-** Abre uma conexão usando os parâmetros da *port* definida na estrutura.
- ***llwrite*** **-** Envia informação num *buffer*, com um determinado tamanho, e retorna o número de caracteres escritos.
- ***llread*** **-** Recebe a informação num pacote e retorna o número de caracteres lidos.
- ***llclose*** **-** Fecha a conexão previamente aberta.

```
// Open a connection using the "port" parameters defined in struct linkLayer.
// Return "1" on success or "-1" on error.
int llopen(LinkLayer link);

// Send data in buf with size bufSize.
// Return number of chars written, or "-1" on error.
int llwrite(int fd, const unsigned char *buf, int bufSize);

// Receive data in packet.
// Return number of chars read, or "-1" on error.
int llread(int fd, unsigned char *packet);

// Close previously opened connection.
// Return "1" on success or "-1" on error.
int llclose(int fd, LinkLayerRole);
```

5- Casos de uso principais

Este trabalho explora diversos cenários de aplicação, englobando situações como a transferência de ficheiros entre dois computadores e a implementação de uma interface que permite ao transmissor selecionar o ficheiro a ser enviado. Para executar o programa, é necessário que o recetor utilize o comando make run_rx, enquanto o transmissor deve empregar make run_tx.

No contexto do transmissor, a conexão entre os computadores é iniciada através do llopen. Em seguida, o pacote de controlo start é enviado, dando início à execução da função llwrite onde acontece o processo de payload seguido pelo stuffing. Posteriormente, o pacote de controle end é transmitido, culminando no encerramento da conexão entre os computadores por meio da função do llclose.

No que diz respeito ao recetor, o estabelecimento da conexão entre os computadores ocorre por meio do llopen. A recepção do pacote de controlo start é seguida pela obtenção dos pacotes de dados de tamanho de 512 bytes através da função llread. Em seguida, o pacote de controlo end é recebido, finalizando a conexão entre os computadores pela função llclose.

6- Protocolo de ligação lógica

6.1- llopen

```
int llopen(LinkLayer link);
```

Esta função é responsável pelo estabelecimento da ligação entre os computadores. A função começa por abrir e configurar a porta série. De seguida, o emissor manda uma trama de supervisão SET (comando) e fica à espera que o recetor lhe responda com outra trama de supervisão. O recetor, recebendo o SET, responde com a trama de supervisão UA (resposta ao comando). Caso o emissor receba a resposta UA, a ligação foi bem estabelecida. A leitura dos pacotes, tal como nas restantes funções, é controlada com uma *state machine* para a verificação da sua integridade e controlo.

6.2- llwrite

```
int llwrite(int fd, const unsigned char *buf, int bufSize);
```

Uma vez que a ligação está estabelecida, o emissor começa a enviar informação que será lida pelo recetor. Esse envio de informação é feito pela função ***llwrite*** e apenas o emissor é que a utiliza. A função recebe um pacote de controlo/dados e executa o processo de *byte stuffing* para evitar conflitos entre bytes. O pacote é transformado para uma trama de informação e envia-se para o recetor, esperando pela sua resposta. No caso de a tama ser rejeitada pelo recetor, é enviada novamente até ser aceite ou exceder o número de tentativas. O emissor aguarda até receber a resposta do recetor, sendo que pode ser rejeitada (REJ0/REJ1) ou aceite (RR0/RR1).

6.3- llread

```
int llread(int fd, unsigned char *packet);
```

O recetor procede então à leitura das tramas de informação a partir da função ***llread***. A função faz o *destuffing* do campo de dados e procede à leitura. Se os campos BCC1 e

BCC2 forem válidos, não ocorreram erros na transmissão. No caso de haver erros, a função vai responder ao emissor com uma trama de supervisão REJ. No caso da trama ser aceite, a resposta ao emissor será com um RR.

6.4- llclose

```
int llclose(int fd, LinkLayerRole);
```

A função ***llclose*** é responsável por terminar a ligação através da porta série. O comportamento desta função varia caso se trate de um recetor ou transmissor. Assim, no caso do recetor, espera pela trama de supervisão DISC e quando a recebe, envia um DISC e espera por uma resposta UA final. No caso do emissor, é enviada uma trama de supervisão DISC, recebe outro DISC do recetor e, por fim, envia um UA. A transmissão é depois fechada através da porta série.

7- Protocolo de aplicação

A camada de mais alto nível é a de aplicação que é responsável pela leitura e escrita do ficheiro a enviar ou receber e pelo envio e receção dos pacotes de dados e controle. A ***application_layer*** está dividida em dois blocos, e escolhe qual deles executa, dependendo se se trata do emissor ou do recetor. Ainda antes de executar um desses blocos, é chamada a função ***llopen***, para estabelecer a conexão.

8- Validação

Foram efetuados os seguintes testes para garantir a integridade das transmissões e confiabilidade do programa:

- Envio de um ficheiro;
- Interrupção da ligação da porta série enquanto se envia o ficheiro;
-Introduzir ruído na porta série enquanto se envia o ficheiro;

Os testes foram efetuados na presença do docente quando foi efetuada a apresentação.

9- Eficiência do protocolo da ligação de dados

Neste projeto, não foram efetuadas estatísticas relativamente à eficiência, com as diferentes variáveis.

10- Conclusão

O objetivo principal deste projeto, que consiste no envio de um ficheiro entre dois computadores através do uso da porta série RS-232, foi concluído com sucesso.
Este trabalho contribuiu principalmente para uma melhor compreensão acerca da independência de camadas em sistemas e a forma como estes estão organizados.
A realização do projeto também fez com que aprendêssemos conceitos novos tais como o stuffing, framing e deu-nos novas formas de implementação de código.

Anexo I – link_layer.h

```c
#ifndef _LINK_LAYER_H_
#define _LINK_LAYER_H_

#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 <signal.h>
#include <time.h>

#define _POSIX_SOURCE 1
#define BAUDRATE 38400
#define MAX_PAYLOAD_SIZE 1000

#define BUF_SIZE 256
#define FALSE 0
#define TRUE 1

#define FLAG 0x7E
#define ESC 0x7D
#define A_ER 0x03
#define A_RE 0x01
#define C_SET 0x03
#define C_DISC 0x0B
#define C_UA 0x07
#define C_RR0 0x05
#define C_RR1 0x85
#define C_REJ0 0x01
#define C_REJ1 0x81
#define C_I0 0x00
#define C_I1 0x40

typedef enum
{
    transmitter,
    receiver,
} LinkLayerRole;

typedef enum
{
    START,              //0
    FLAG_RCV,           //1
    A_RCV,              //2
    C_RCV,              //3
    BCC1_OK,            //4
    STOP_STATE,         //5
    DATA_FOUND_ESC,     //6
    DATA,               //7
    DISCONNECTED,       //8
    BCC2_OK             //9
} LinkLayerState;

typedef struct
{
    char serialPort[50];
    LinkLayerRole role;
    int baudRate;
    int nRetransmissions;
    int timeout;
} LinkLayer;
```

```c
// Open a connection using the "port" parameters defined in struct linkLayer.
// Return "1" on success or "-1" on error.
int llopen(LinkLayer link);

// Send data in buf with size bufSize.
// Return number of chars written, or "-1" on error.
int llwrite(int fd, const unsigned char *buf, int bufSize);

// Receive data in packet.
// Return number of chars read, or "-1" on error.
int llread(int fd, unsigned char *packet);

// Close previously opened connection.
// Return "1" on success or "-1" on error.
int llclose(int fd, LinkLayerRole);

// timeout
void alarmHandler(int signal);

unsigned char readControlFrame (int fd);

int sendFrame(int fd, unsigned char A, unsigned char C);

#endif // _LINK_LAYER_H_
```

Anexo II – application_layer.h

```c
#ifndef _APPLICATION_LAYER_H_
#define _APPLICATION_LAYER_H_

#include <stdio.h>

#define PCK_SIZE 512

#define CTRLDATA 1
#define CTRLSTART 2
#define CTRLEND 3


// Application layer main function.
// Arguments:
//   serialPort: Serial port name (e.g., /dev/ttyS0).
//   role: Application role {"tx", "rx"}.
//   baudrate: Baudrate of the serial port.
//   nTries: Maximum number of frame retries.
//   timeout: Frame timeout.
//   filename: Name of the file to send / receive.

void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename);

int tlv(unsigned char *address, int* type, int* length, int** value);


#endif // _APPLICATION_LAYER_H_
```

Anexo III – link_layer.c

```c
#include "link_layer.h"

volatile int STOP = FALSE;
int alarmEnabled = FALSE;
int alarmCount = 0;
int timeout = 0;
int retransmitions = 0;
unsigned char tramaNr = 0;



int sendFrame(int fd, unsigned char A, unsigned char C){
    unsigned char FRAME[5] = {FLAG, A, C, A ^ C, FLAG};

    return write(fd, FRAME, 5);
}

unsigned char readControlFrame(int fd){

    unsigned char byte, cField = 0;
    LinkLayerState state = START;

    while (state != STOP_STATE && alarmEnabled == FALSE) {
        if (read(fd, &byte, 1) > 0) {
            switch (state) {
                case START:
                    if (byte == FLAG) state = FLAG_RCV;
                    break;
                case FLAG_RCV:
                    if (byte == A_RE) state = A_RCV;
                    else if (byte != FLAG) state = START;
                    break;
                case A_RCV:
                    if (byte == C_RR0 || byte == C_RR1 || byte == C_REJ0 || byte == C_REJ1 || byte == C_DISC){
                        state = C_RCV;
                        cField = byte;
                    }
                    else if (byte == FLAG) state = FLAG_RCV;
                    else state = START;
                    break;
                case C_RCV:
                    if (byte == (A_RE ^ cField)) state = BCC1_OK;
                    else if (byte == FLAG) state = FLAG_RCV;
                    else state = START;
                    break;
                case BCC1_OK:
                    if (byte == FLAG){
                        state = STOP_STATE;
                    }
                    else state = START;
                    break;
                default:
                    break;
            }
        }
    }
    return cField;
}

void alarmHandler(int signal) {
    alarmEnabled = TRUE;
    alarmCount++;
}
```

```c
int llopen(LinkLayer link) {


//-----------------------------------CONFIG PORT-----------------//

    const char *serialPortName = link.serialPort;

    // Open serial port device for reading and writing and not as controlling tty
    // because we don't want to get killed if linenoise sends CTRL-C.
    int fd = open(serialPortName, O_RDWR | O_NOCTTY);
    if (fd < 0)
    {
        perror(serialPortName);
        exit(-1);
    }

    struct termios oldtio;
    struct termios newtio;


    // Save current port settings
    if (tcgetattr(fd, &oldtio) == -1)
    {
        perror("tcgetattr");
        exit(-1);
    }

    // Clear struct for new port settings
    memset(&newtio, 0, sizeof(newtio));

    newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;

    // Set input mode (non-canonical, no echo,...)
    newtio.c_lflag = 0;
    newtio.c_cc[VTIME] = 0; // Inter-character timer unused
    newtio.c_cc[VMIN] = 0;  // Blocking read until 5 chars received

    // VTIME e VMIN should be changed in order to protect with a
    // timeout the reception of the following character(s)

    // Now clean the line and activate the settings for the port
    // tcflush() discards data written to the object referred to
    // by fd but not transmitted, or data received but not read,
    // depending on the value of queue_selector:
    //   TCIFLUSH - flushes data received but not read.
    tcflush(fd, TCIOFLUSH);

    // Set new port settings
    if (tcsetattr(fd, TCSANOW, &newtio) == -1)
    {
        perror("tcsetattr");
        exit(-1);
    }

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

```
//----------------------------------ROLES-----------------//


    LinkLayerState state = START;
    unsigned char byte;
    retransmitions = link.nRetransmissions;
    timeout = link.timeout;

    // Handle different roles
    switch (link.role) {
        case transmitter: {

            //alarm setup
            (void) signal(SIGALRM, alarmHandler);

            printf("retransmissions: %d\n", retransmitions);

            // alarm loop
            while (retransmitions > 0 && state != STOP_STATE) {
                // Send frame (C_SET)
                //printf("sending frame\n");
                sendFrame(fd, A_ER, C_SET);
                alarm(link.timeout);
                alarmEnabled = FALSE;

                // Receive and process frames
                while (alarmEnabled == FALSE && state != STOP_STATE) {

                    //reading response frame

                    if (read(fd, &byte, 1) > 0) {
                        //printf("state: %d\n", state);
                        //printf("reading byte: %d\n", byte);
                        switch (state) {
                            case START:
                                if (byte == FLAG) state = FLAG_RCV;
                                break;
                            case FLAG_RCV:
                                if (byte == A_RE) state = A_RCV;
                                else if (byte != FLAG) state = START;
                                break;
                            case A_RCV:
                                if (byte == C_UA) state = C_RCV;
                                else if (byte == FLAG) state = FLAG_RCV;
                                else state = START;
                                break;
                            case C_RCV:
                                if (byte == (A_RE ^ C_UA)) state = BCC1_OK;
                                else if (byte == FLAG) state = FLAG_RCV;
                                else state = START;
                                break;
                            case BCC1_OK:
                                if (byte == FLAG) state = STOP_STATE;
                                else state = START;
                                printf("state: %d", STOP_STATE);
                                break;
                            default:
                                break;
                        }
                    }
                }

                retransmitions--;
            }
```

```
            if (state != STOP_STATE) {
                printf("ERROR: did not reach STOP state\n");
                return -1;
            }
            printf("\nllopen: Connection successfull\n");
            break;
        }
        case receiver: {
            // Main loop for the receiver role
            while (state != STOP_STATE) {
                if (read(fd, &byte, 1) > 0) {
                    printf("reading byte: %d\n", byte);
                    switch (state) {
                        case START:
                            if (byte == FLAG) state = FLAG_RCV;
                            break;
                        case FLAG_RCV:
                            if (byte == A_ER) state = A_RCV;
                            else if (byte != FLAG) state = START;
                            break;
                        case A_RCV:
                            if (byte == C_SET) state = C_RCV;
                            else if (byte == FLAG) state = FLAG_RCV;
                            else state = START;
                            break;
                        case C_RCV:
                            if (byte == (A_ER ^ C_SET)) state = BCC1_OK;
                            else if (byte == FLAG) state = FLAG_RCV;
                            else state = START;
                            break;
                        case BCC1_OK:
                            if (byte == FLAG) state = STOP_STATE;
                            else state = START;
                            break;
                        default:
                            break;
                    }
                }
            }

            printf("sending ua frame\n");
            //printf("state: %d\n", state);

            // Send response frame (C_UA)
            sendFrame(fd, A_RE, C_UA);
            break;
        }
        default:
            return -1; // Unsupported role
    }

    // Close the serial port and return it
    // close(fd);
    return fd;

}
```

```c
int llread(int fd, unsigned char *buffer){

    unsigned char byte, c_type;
    int i = 0;
    LinkLayerState state = START;

    while (state != STOP_STATE) {
        if (read(fd, &byte, 1) > 0) {
            //printf("state: %d\n", state);
            //printf("byte: %d\n", byte);
            switch (state) {
                case START:
                    if (byte == FLAG) state = FLAG_RCV;
                    break;
                case FLAG_RCV:
                    if (byte == A_ER) state = A_RCV;
                    else if (byte != FLAG) state = START;
                    break;
                case A_RCV:
                    if (byte == C_I0 || byte == C_I1){
                        state = C_RCV;
                        c_type = byte;
                    }
                    else if (byte == FLAG) state = FLAG_RCV;
                    else if (byte == C_DISC) {
                        sendFrame(fd, A_RE, C_DISC);
                        return 0;
                    }
                    else state = START;
                    break;
                case C_RCV:
                    if (byte == (A_ER ^ c_type)) state = DATA;
                    else if (byte == FLAG) state = FLAG_RCV;
                    else state = START;
                    break;
```

```
                case DATA:
                    if (byte == ESC) state = DATA_FOUND_ESC;
                    else if (byte == FLAG){
                        unsigned char bcc2 = buffer[i-1];
                        i--;
                        buffer[i] = '\0';
                        unsigned char acc = buffer[0];

                        for (unsigned int j = 1; j < i; j++)  //isto faz com que de vez em quando
                                                              //o programa nao funcione direito na presença de ruído
                            acc ^= buffer[j];

                        if (bcc2 == acc){
                            state = STOP_STATE;
                            sendFrame(fd, A_RE, tramaNr == 0 ? C_RR1: C_RR0);
                            tramaNr = tramaNr == 0 ? 1 : 0;
                            return i;
                        }
                        else{
                            printf("Error: retransmition\n");
                            sendFrame(fd, A_RE, tramaNr == 0 ? C_REJ0 : C_REJ1);
                            return -1;
                        };

                    }
                    else{
                        buffer[i++] = byte;
                    }
                    break;
                case DATA_FOUND_ESC:
                    state = DATA;
                    if (byte == ESC || byte == FLAG) buffer[i++] = byte;
                    else{
                        buffer[i++] = ESC;
                        buffer[i++] = byte;
                    }
                    break;
                default:
                    break;
            }
        }
    }
    return -1;
}
```

```c
int llwrite(int fd, const unsigned char *buffer, int bufferLength) {

    //--------------------PAYLOAD/PACKET-----------------//

    int frameSize = 6+bufferLength;
    unsigned char *frame = (unsigned char *) malloc(frameSize); //frame fica com a length de frameSize
    frame[0] = FLAG;
    frame[1] = A_ER;
    frame[2] = tramaNr == 0 ? C_I0 : C_I1;
    frame[3] = frame[1] ^ frame[2];
    memcpy(frame+4,buffer, bufferLength);
    unsigned char BCC2 = buffer[0];


    for (unsigned int i = 1 ; i < bufferLength ; i++) BCC2 ^= buffer[i];


    //--------------------STUFFING-----------------//

    int j = 4;
    for (unsigned int i = 0 ; i < bufferLength ; i++) {
        if(buffer[i] == FLAG || buffer[i] == ESC) {
            frame = realloc(frame,++frameSize);
            frame[j++] = ESC;
        }
        frame[j++] = buffer[i];
    }
    frame[j++] = BCC2;
    frame[j++] = FLAG;

    int currentTransmition = 0;
    int rejected = 0, accepted = 0;
```

```
//trying to send to rx

while (currentTransmition < retransmitions) {
    alarmEnabled = FALSE;
    alarm(timeout);
    rejected = 0;
    accepted = 0;
    while (alarmEnabled == FALSE && !rejected && !accepted) {

        write(fd, frame, j);
        printf("write successful\n");
        unsigned char cField = readControlFrame(fd);

        printf("reading Control Frame cField: %d\n", cField);

        if(!cField){
            continue;
        }
        else if(cField == C_REJ0 || cField == C_REJ1) {
            rejected = 1;
        }
        else if(cField == C_RR0 || cField == C_RR1) {
            accepted = 1;
            tramaNr = tramaNr == 0 ? 1 : 0;
        }
        else continue;

    }
    if (accepted){
        printf("receiver accepted the frame sent\n");
        break;
    }
    currentTransmition++;
}

//printf("about to free frame\n");

free(frame); //free memory
if(accepted) return frameSize;
else{
    printf("receiver did not accept the frame sent\n");
    //llclose(fd, transmitter);
    return -1;
}

return 1;
```

```c
int llclose(int fd, LinkLayerRole role){

    LinkLayerState state = START;
    unsigned char byte;
    (void) signal(SIGALRM, alarmHandler);


    if(role == receiver){
        printf("llclose receiver\n");

        ///printf("llclose: analysing disc\n");
        while (1) {
            int c_type;
            if (read(fd, &byte, 1) > 0) {
                //printf("llclose state: %d\n", state);
                //printf("byte: %d\n", byte);
                switch (state) {
                    case START:
                        if (byte == FLAG) state = FLAG_RCV;
                        break;
                    case FLAG_RCV:
                        if (byte == A_ER) state = A_RCV;
                        else if (byte != FLAG) state = START;
                        break;
                    case A_RCV:
                        if (byte == C_DISC){ state = C_RCV; c_type = C_DISC; }
                        else if (byte == C_UA) { state = C_RCV; c_type = C_UA; }
                        else if (byte == FLAG) state = FLAG_RCV;
                        else state = START;
                        break;
                    case C_RCV:
                        if (byte == (A_ER ^ c_type)) state = BCC1_OK;
                        else if (byte == FLAG) state = FLAG_RCV;
                        else state = START;
                        break;
                    case BCC1_OK:
                        if (byte == FLAG) state = STOP_STATE;
                        else state = START;
                        break;
                    default:
                        break;
                }
            }
            if(state == STOP_STATE){
                if(c_type == C_UA){
                    printf("closing llclose\n");
                    return close(fd);
                }

                printf("llclose: sending disc response frame\n");
                sendFrame(fd, A_RE, C_DISC);
                state = START;
                //break;
            }
        }
    }
```

```
    else {
        while (retransmitions != 0 && state != STOP_STATE) {

            printf("llclose: sending disc\n");
            sendFrame(fd, A_ER, C_DISC); //send Disc (Tx)
            alarm(timeout);
            alarmEnabled = FALSE;

            printf("llclose: receiving disc\n");
            while (alarmEnabled == FALSE && state != STOP_STATE) {
                if (read(fd, &byte, 1) > 0) {
                    printf("llclose state: %d\n", state);
                    printf("byte: %d\n", byte);
                    switch (state) {
                        case START:
                            if (byte == FLAG) state = FLAG_RCV;
                            break;
                        case FLAG_RCV:
                            if (byte == A_RE) state = A_RCV;
                            else if (byte != FLAG) state = START;
                            break;
                        case A_RCV:
                            if (byte == C_DISC) state = C_RCV;
                            else if (byte == FLAG) state = FLAG_RCV;
                            else state = START;
                            break;
                        case C_RCV:
                            if (byte == (A_RE ^ C_DISC)) state = BCC1_OK;
                            else if (byte == FLAG) state = FLAG_RCV;
                            else state = START;
                            break;
                        case BCC1_OK:
                            if (byte == FLAG) state = STOP_STATE;
                            else state = START;
                            break;
                        default:
                            break;
                    }
                }
            }
            retransmitions--;
        }


        if (state != STOP_STATE) return -1;

        printf("llclose: valid frame\n");

        printf("llclose: sending ua frame\n");

        sendFrame(fd, A_ER, C_UA);
        printf("closing llclose\n");
        return close(fd);
    }
}
```

Anexo IV – application_layer.c

```c
#include "application_layer.h"
#include "link_layer.h"
#include <string.h>
#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 <math.h>



unsigned char packet[PCK_SIZE];


void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename){


    printf("defining roles\n");

    LinkLayer connectionParameters;
    strcpy(connectionParameters.serialPort, serialPort);

    if(strcmp(role,"rx") == 0) {
        connectionParameters.role = receiver;
    }
    else if(strcmp(role,"tx") == 0) {
        connectionParameters.role = transmitter;
    }
    else {
        perror(role);
        exit(-1);
    }

    connectionParameters.baudRate = baudRate;
    connectionParameters.nRetransmissions = nTries;
    connectionParameters.timeout = timeout;




    printf("\n----------------LLOPEN-----------------\n");

    int fd = llopen(connectionParameters);

    if(fd == -1) {
      perror("Error: connection not successfull\n");
      exit(-1);
    }else{
        printf("Connection successfull\n");
    }
```

```
//--------------------TRANSMITTER----------------------//


if(connectionParameters.role == transmitter) {

    printf("\n------------------TRANSMITTER-------------------\n");

    FILE* file = fopen(filename, "rb"); //open file in binary mode: r == read, b == binary
    printf("reading file: %s\n", filename);

    if(file == NULL) {
        perror("Error openning the file\n");
        exit(-1);
    }

    struct stat st; //to retrieve file information
    int file_size = (stat(filename, &st) == 0) ? st.st_size : 0;
    printf("file_size from stat: %d\n", file_size);

    printf("file_permissions stat: %d\n", st.st_mode & 07777);  //wrong permissions for some reason

    packet[0] = CTRLSTART;
    packet[1] = 0; //control flag
    packet[2] = sizeof(long);           //L2 guide page 25
    *((long*)(packet + 3)) = file_size; //L1 guide page 25


    //printf("\n------------------LLWRITE-------------------\n");

    printf("SENDING CONTROL START PACKET\n");

    int frameSize_llwrite = llwrite(fd, packet, 10);

    if( frameSize_llwrite == -1){
        perror("llwrite returned -1\n");
        exit(-1);
    }
```

```
    //------------------------------FROM FILE TO BYTES-------------------------//

    int bytes_tx = 0;
    unsigned char i = 0;
    do {
        unsigned long total_bytes;
        if(file_size - bytes_tx < PCK_SIZE) {
            total_bytes = fread(packet + 4, 1, file_size - bytes_tx, file);
        }
        else {
            total_bytes = fread(packet + 4, 1, PCK_SIZE, file);
        }

        packet[0] = CTRLDATA;
        packet[1] = i;
        packet[2] = total_bytes >> 8;  //first 8 bytes
        packet[3] = total_bytes % 256; //last 8 bytes

        printf("SENDING CONTROL DATA PACKET\n");

        if(llwrite(fd, packet, total_bytes + 4) == -1){
            printf("failed llwriting CTRLDATA\n");
            exit(-1);
            break;
        }
        printf("\nPacket %i sent\n\n",i);
        bytes_tx += total_bytes;
        i++;
    }while(bytes_tx < file_size);

    printf("SENDING CONTROL END PACKET\n");

    packet[0] = CTRLEND;
    if(llwrite(fd, packet,1) == -1){
        printf("failed llwriting CTRLEND\n");
        perror("llwrite\n");
        exit(-1);
    }


    fclose(file);


    printf("\n------------------LLCLOSE-----------------\n");

    if(llclose(fd, transmitter) == -2) {
        printf("llclose failed\n");
        perror("llclose");
        exit(-2);
    }
}
```

```
    //------------------------------FROM FILE TO BYTES-------------------------//

    int bytes_tx = 0;
    unsigned char i = 0;
    do {
        unsigned long total_bytes;
        if(file_size - bytes_tx < PCK_SIZE) {
            total_bytes = fread(packet + 4, 1, file_size - bytes_tx, file);
        }
        else {
            total_bytes = fread(packet + 4, 1, PCK_SIZE, file);
        }

        packet[0] = CTRLDATA;
        packet[1] = i;
        packet[2] = total_bytes >> 8;  //first 8 bytes
        packet[3] = total_bytes % 256; //last 8 bytes

        printf("SENDING CONTROL DATA PACKET\n");

        if(llwrite(fd, packet, total_bytes + 4) == -1){
            printf("failed llwriting CTRLDATA\n");
            exit(-1);
            break;
        }
        printf("\nPacket %i sent\n\n",i);
        bytes_tx += total_bytes;
        i++;
    }while(bytes_tx < file_size);

    printf("SENDING CONTROL END PACKET\n");

    packet[0] = CTRLEND;
    if(llwrite(fd, packet,1) == -1){
        printf("failed llwriting CTRLEND\n");
        perror("llwrite\n");
        exit(-1);
    }


    fclose(file);


    printf("\n------------------LLCLOSE------------------\n");

    if(llclose(fd, transmitter) == -2) {
        printf("llclose failed\n");
        perror("llclose");
        exit(-2);
    }
}
```

```
//---------------------RECEIVER----------------------//

else if(connectionParameters.role == receiver) {

    int file_size = 0, bytes_rx = 0;


    printf("\nREADING CONTROL START PACKET\n");

    int bytes = llread(fd, packet);

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




    int type,length,*value;

    if(packet[0] != CTRLSTART){
        perror("Failed reading CTRLSTART packet\n");
        exit(-1);
    }

    //get file size (tlv)

    int tlv_size = 1; //tracks the position within the packet
    while(tlv_size < bytes) {
        tlv_size += tlv(packet + tlv_size, &type, &length, &value);
        if(type == 0){
            file_size = *value;
            printf("File size: %d\n",file_size);
        }
    }

    FILE* file = fopen(filename, "wb"); //w = write, b = binary

    if(file == NULL) {
        perror("Cannot open the file\n");
        exit(-1);
    }
    else {
        printf("Control packet received\n");
        printf("ready to write to file\n");
    }
```

```
//------------------------------FROM BYTES TO FILE-------------------------//

printf("\nREADING CONTROL DATA PACKET\n");

int packetNumber = 0;

while(bytes_rx < file_size) {

    printf("bytes_rx: %d\n", bytes_rx);

    int bytes;
    if((bytes = llread(fd, packet)) == -1){
        perror("llread\n");
        exit(-1);
    }

    if(packet[0] == CTRLEND){
        perror("CTRL_END failed\n");
        exit(-1);
    }

    if(packet[0] == CTRLDATA){
        if(bytes < 5) {
            perror("CTRLDATA failed\n");
            exit(-1);
        }
        else if(packet[1] != packetNumber){  //so it doesnt skip packets
            perror("packetNumber failed\n");
            exit(-1);
        }
        else{
            unsigned long size = packet[3] + packet[2]*256; //guide slide 25

            if(bytes != size + 4) { //4 = header
                perror("Wrong header\n");
                exit(-1);
            }

            fwrite(packet + 4, 1, size, file);
            bytes_rx += size;

            printf("Packet %d received\n", packetNumber);

            packetNumber++;
        }
    }
}


printf("\nREADING CONTROL END PACKET\n");
```

```
        int bytes_read = llread(fd, packet);

        if(bytes_read == -1) {
            perror("llread\n");
            exit(-1);
        }
        else if(bytes_read < 1) {
            perror("Short packet\n");
            exit(-1);
        }

        if(packet[0] != CTRLEND){
            printf("CTRLEND failed\n");
        }
        else{
            printf("Received end packet\n");
        }
        fclose(file);


        printf("\nCLOSING CONNECTION\n\n");

        if(llclose(fd, receiver) == -2) {
            printf("llclose failed\n");
            perror("llclose");
            exit(-2);
        }
    }


}


//--------------------AUXILIARY FUNCTIONS----------------------//


int tlv(unsigned char *address, int* type, int* length, int** value){

    *type = address[0];
    *length = address[1];
    *value = (int*)(address + 2);

    return 2 + *length;
}
```