Faculdade de Engenharia da Universidade do Porto

Protocolo de Ligação de Dados

1º Trabalho Laboratorial

Redes de Computadores
Turma 5 - Grupo 5

- Fábio Araújo de Sá (up202007658@fe.up.pt)
- Lourenço Alexandre Correia Gonçalves (up202004816@fe.up.pt)

Porto, 30 de Outubro de 2022

Sumário

Este trabalho realizado no âmbito da Unidade Curricular de Redes de Computadores visa a implementação de um protocolo de comunicação de dados para a transmissão de ficheiros utilizando a Porta Série RS-232.

Através deste projeto conseguimos fazer uso da matéria lecionada nas aulas teóricas para implementar o protocolo em questão, consolidando assim o funcionamento da estratégia *Stop-and-Wait.*

Introdução

O objetivo do trabalho foi o desenvolvimento e teste de um protocolo de ligação de dados, de acordo com as especificações presentes no guião, para transferir um ficheiro através da porta série. O presente relatório está dividido em oito secções:

- **Arquitetura:** Blocos funcionais e interfaces utilizadas.
- **Estrutura do código:** Apresentação das principais APIs, estruturas e funções.
- **Casos de uso principais:** Identificação dos funcionamento do projeto, bem como a sequência de chamadas das funções.
- **Protocolo de ligação lógica:** Funcionamento da ligação lógica e estratégias de implementação
- **Protocolo de aplicação:** Funcionamento da camada de aplicação e estratégias de implementação
- **Validação:** Testes efetuados para avaliar a correção da implementação
- **Eficiência do protocolo de ligação de dados:** Quantização da eficiência do protocolo Stop&Wait implementado na camada de ligação de dados.
- **Conclusões:** Síntese da informação apresentada nas secções anteriores e reflexão sobre os objetivos de aprendizagem alcançados.

Arquitetura

Blocos Funcionais

O projeto foi desenvolvido com base em duas camadas principais: a *LinkLayer* e a *ApplicationLayer.*

A *LinkLayer*, ou camada de ligação de dados, consiste na implementação do protocolo referido anteriormente, presente nos ficheiros *link_layer.h* e *link_layer.c*. Esta camada é responsável pelo estabelecimento e terminação da ligação, criação e envio de tramas de dados através da porta série e por fim a validação das tramas recebidas e o envio de mensagens de erro caso tenha ocorrido algum problema na transmissão.

A *ApplicationLayer,* ou camada de aplicação, implementada nos ficheiros *application_layer.h* e *application_layer.c*, utiliza a API da *LinkLayer* para transferência e receção de pacotes de dados constituintes de um ficheiro. É por isso a camada mais próxima ao utilizador e nela é possível definir o tamanho das tramas de informação, a velocidade da transferência e o número máximo de retransmissões.

Interfaces

A execução do programa é realizada através de dois terminais, um em cada computador, em que um deles executa o binário em modo transmissor e o outro em modo recetor.

```
$ <PROGRAM> <SERIAL_PORT> <ROLE> <FILE>
  - PROGRAM: binário a executar
  - SERIAL_PORT: localização da porta série a utilizar
  - ROLE: 'tx' para o transmissor, 'rx' para o recetor
  - FILE: nome do ficheiro a transferir
```

Estrutura do código

ApplicationLayer

Na implementação desta camada não houve necessidade de criar estruturas de dados auxiliares. As funções implementadas foram:

```
// Função principal deste bloco
void applicationLayer(const char *serialPort, const char *role, int baudRate,
int nTries, int timeout, const char *filename);
// Interpretação de um pacote de controlo
unsigned char* parseControlPacket(unsigned char* packet, int size, unsigned
long int *fileSize);
// Interpretação de um pacote de dados
void parseDataPacket(const unsigned char* packet, const unsigned int
packetSize, unsigned char* buffer);
// Criação de um pacote de controlo
unsigned char * getControlPacket(const unsigned int c, const char* filename,
long int length, unsigned int* size);
// Criação de um pacote de dados
unsigned char * getDataPacket(unsigned char sequence, unsigned char *data, int
dataSize, int *packetSize);
// Leitura de dados do ficheiro a transferir
unsigned char * getData(FILE* fd, long int fileLength);
```

LinkLayer

Na implementação desta camada foram utilizadas três estruturas de dados auxiliares: *LinkLayer*, onde são caracterizados os parâmetros associados à transferência de dados, *LinkLayerRole*, que identifica se o computador é um transmissor ou recetor, *LinkLayerState*, que identifica o estado da leitura e receção das tramas de informação.

```
typedef enum {
   LlTx,
   LlRx,
} LinkLayerRole;
```

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

```
typedef enum {
   START, FLAG_RCV, A_RCV, C_RCV, BCC1_OK, STOP_R, DATA_FOUND_ESC,
   READING_DATA, DISCONNECTED, BCC2_OK
} LinkLayerState;
```

As funções implementadas foram:

```
// Estabelecimento da ligação entre o transmissor e recetor
int llopen(LinkLayer connectionParameters);
// Envio de tramas
int llwrite(int fd, const unsigned char *buf, int bufSize);
// Leitura de tramas
int llread(int fd, unsigned char *packet);
// Terminação da ligação
int llclose(int fd);
// Estabelecimento da ligação com a serial port
int connection(const char *serialPort);
// Criação um alarme
void alarmHandler(int signal);
// Leitura de uma trama de supervisão
unsigned char readControlFrame (int fd);
// Envio de uma trama de supervisão
int sendSupervisionFrame(int fd, unsigned char A, unsigned char C);
```

Casos de uso principais

Como já foi referido, o programa pode ser executado nos modos transmissor e recetor. Dependendo de qual for escolhido, as funções utilizadas e a sequência de chamadas serão diferentes.

Transmissor

1. **llopen()**, usada para o *handshake* entre o transmissor e o recetor, através da troca de pacotes de controlo e conexão com a porta série pela chamada *connection()*.
2. **getData(),** retorna o conteúdo do ficheiro a ser transferido.
3. **getControlPacket(),** criação um pacote de controlo.

4. **llwrite()**, cria e envia pela porta série uma trama de informação com base no pacote recebido como argumento.
5. **readControlFrame()**, máquina de estados que lê e valida a resposta do receptor após qualquer escrita na porta série.
6. **llclose(),** usada para terminar a ligação entre o transmissor e o recetor, através da troca de pacotes de controlo.

Recetor

1. **llread()**, máquina de estados que gere e valida a receção de tramas de controlo e tramas de dados.
2. **sendSupervisionFrame(),** cria e envia pela porta série uma trama de supervisão com base na trama lida por *llread().*
3. **parseControlPacket(),** retorna as caraterísticas do ficheiro a ser transferido contidas no pacote de controlo em formato TLV passado como argumento.
4. **getDataPacket(),** retorna um segmento do ficheiro através do pacote de dados passado como argumento.

Protocolo de ligação lógica

A camada de ligação de dados é a camada que interage diretamente com a Porta Série, sendo responsável pela comunicação entre o emissor e o recetor. Para esse fim, utilizamos o protocolo *Stop-and-wait* para o estabelecimento e terminação da ligação e para o envio de tramas de supervisão e informação.

O estabelecimento da ligação é realizado pela função **llopen.** Inicialmente, após abrir e configurar a porta série, o emissor envia uma trama de supervisão SET e espera que o recetor responda com uma trama de supervisão UA. Quando o recetor recebe o SET, responde com UA. Se o emissor receber a trama UA, a ligação foi bem estabelecida. Após estabelecer a ligação, o emissor começa a enviar informação que será lida pelo recetor.

O envio de informação é feito pela função **llwrite.** Esta função recebe um pacote de controlo ou de dados e aplica-lhe a estratégia de *byte stuffing,* de modo a evitar conflitos com os bytes de dados que sejam iguais às *flags* da trama. Posteriormente transforma esse pacote numa trama de informação (*framing*), envia-se para o recetor e espera-se pela sua resposta. Se a trama for rejeitada, o envio é realizado novamente até ser aceite ou exceder o número máximo de tentativas. Cada tentativa de envio tem um limite de tempo após o qual ocorre *time-out.*

A leitura de informação é feita pela função **llread.** Esta função lê a informação recebida pela porta série e verifica a sua validez. Inicialmente faz *destuff* do campo de dados da trama e valida o BCC1 e BCC2, que verifica se ocorreu algum erro durante a transmissão.

O término da ligação é realizado pela função **llclose**. Esta é invocada pelo emissor quando o número de tentativas fracassadas é excedido ou quando a transferência dos pacotes de dados está concluída. O emissor envia uma trama de supervisão DISC e espera que o recetor responda com o mesmo, terminando o seu funcionamento. Quando o emissor volta a receber DISC, responde com UA e interrompe a ligação.

Protocolo de aplicação

A camada da aplicação é a que interage diretamente com o ficheiro a ser transferido e com o utilizador. Nela é possível definir qual o ficheiro a transferir, em que porta série, a velocidade da transferência, o número de bytes de dados do ficheiro inseridos cada pacote, o número máximo de retransmissões e o tempo máximo de espera da resposta por parte do recetor. A transferência do ficheiro ocorre utilizando a API da *LinkLayer,* que traduz os pacotes de dados em tramas de informação.

Após o *handshake* entre o transmissor e o recetor, todo o conteúdo do ficheiro é copiado para um buffer local através de **getData** e fragmentado por **applicationLayer** segundo o número de bytes explicitado no argumento. O primeiro pacote enviado pelo transmissor tem dados em formato TLV (*Type, Length, Value*), criado por **getControlPacket** onde são expressos o tamanho do ficheiro em bytes e o nome do mesmo. No lado do recetor esse pacote é descompactado pela função **parseControlPacket** de forma a criar e alocar o espaço necessário para receber o ficheiro.

Cada fragmento do ficheiro a transferir é inserido num pacote de dados através da função **getDataPacket** e enviado pela porta série usando **llwrite** presente na API. Cada envio é acompanhado por uma resposta por parte do recetor, indicando se aceita ou rejeita o pacote enviado. No primeiro caso o transmissor envia uma com o fragmento seguinte, no segundo caso reenvia o mesmo fragmento. Cada pacote é avaliado individualmente pelo recetor através da função **llread** e **parseDataPacket** extraí do pacote o fragmento original do ficheiro quando recebido corretamente.

A ligação entre as duas máquinas termina quando é invocada a função da API **llclose**, após término da transferência de pacotes de dados ou por ter ultrapassado o número de tentativas fracassadas.

Validação

Ao longo da elaboração do projeto e de forma a garantir continuamente a correta implementação do protocolo desenvolvido, foram efetuados os seguintes testes:

- transferência de ficheiros de diferentes tamanhos
- transferência de ficheiros com nomes distintos
- transferência de ficheiros com diferentes *baudrates*
- transferência de pacotes de dados de diferentes tamanhos
- interrupção parcial e/ou total da Porta Série
- introdução de ruído na Porta Série através de curto-circuito
- rejeição de tramas de dados através da introdução de aleatoriedade

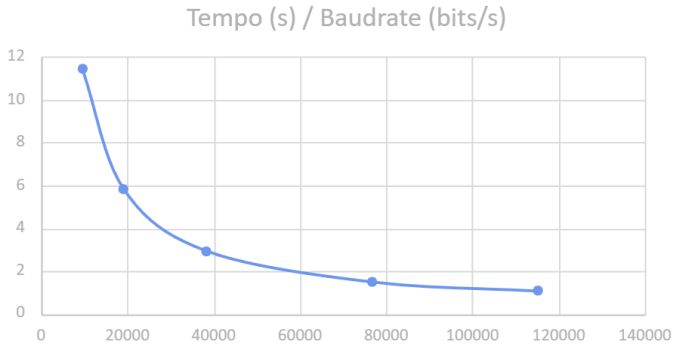
Em todos os cenários apresentados o protocolo *Stop-And-Wait* implementado garantiu a consistência do ficheiro transferido. Os testes foram também reproduzidos sob presença do docente aquando da apresentação do projeto na aula laboratorial.

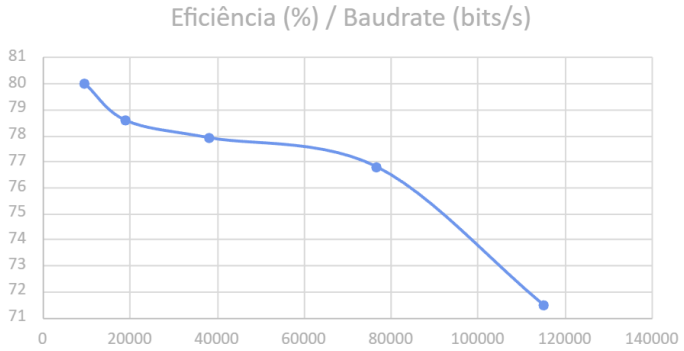
Eficiência do protocolo de ligação de dados

Variação do *baudrate*

Com um ficheiro de 10968 *bytes* e uma trama de informação de tamanho fixo de 1500 *bytes*, a variação do *baudrate* (capacidade da ligação da porta série) originou os seguintes dados:

FER (%)	Média (s)	T frame (bits/s)	Eficiência (%)
10	3,597509	24390	63,52
25	3,864806	22703	59,12
40	4,584223	19140	49,84
50	5,151765	17032	44,35
60	7,064729	12420	32,34



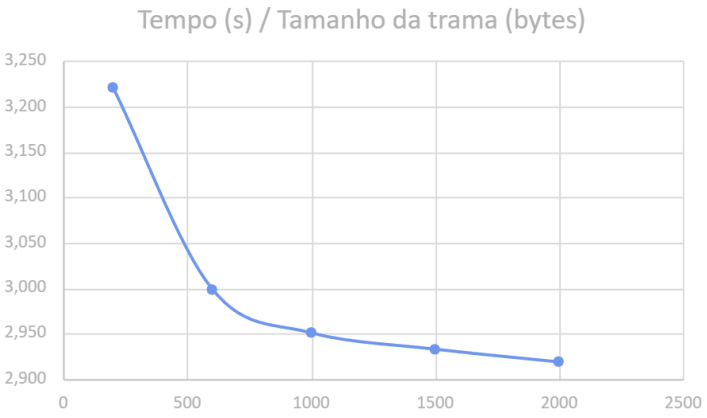


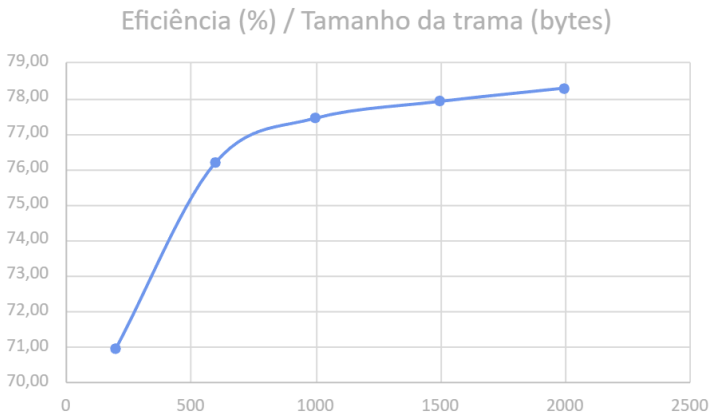
Tal como observado, o tempo total da transferência do ficheiro e a velocidade de propagação no cabo (*baudrate*) são inversamente proporcionais. A eficiência do protocolo diminui gradualmente com o aumento do *baudrate* porque embora a velocidade de propagação das tramas aumente, o tempo de transmissão dessa mesma informação também aumenta.

Variação do tamanho da trama

Com um ficheiro de 10968 bytes, um *baudrate* fixo de 38400 *bits/s,* a variação do tamanho da trama de informação originou o seguinte conjunto dados:

Tamanho da trama (bytes)	Tempo (s)	T frame (bits/s)	Eficiência (%)
200	3,221709	27235	70,93
600	2,999564	29252	76,18
1000	2,951118	29732	77,43
1500	2,933132	29915	77,90
2000	2,919077	30059	78,28



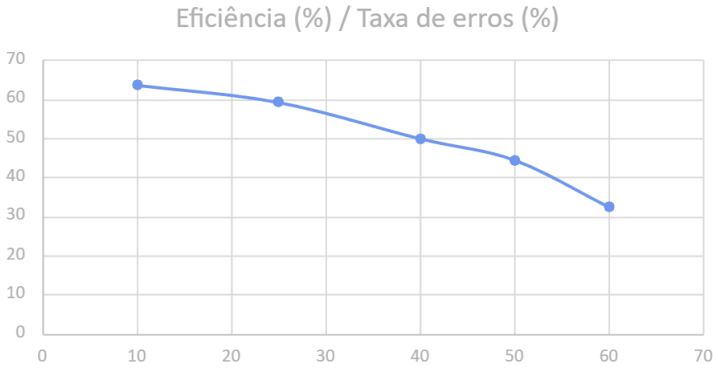


Tal como observado, o tempo total da transferência do ficheiro e o tamanho da trama de dados são inversamente proporcionais, uma vez que quanto menor é o tamanho de cada trama mais transmissões são efetuadas. O aumento da eficiência do protocolo demonstra-se mais acentuado quando o tamanho da trama de informação é menor, estabilizando cada vez mais quando este aumenta.

Variação da taxa de erros

Com um ficheiro de 10968 bytes, um *baudrate* fixo de 38400 *bits/s* e tamanho da trama de dados de 1500 *bytes*, a variação de FER (*Frame Error Ratio*) originou o seguinte conjunto de dados:

Baudrate (bits/s)	Tempo (s)	T frame (bits/s)	Eficiência (%)
9600	11,42377	7681	80,01
19200	5,816547	15085	78,57
38400	2,933132	29915	77,90
76800	1,487831	58974	76,79
115200	1,065769	82329	71,47



Tal como observado, o tempo total da transferência do ficheiro e a percentagem de erros em tramas de informação são diretamente proporcionais. De facto, no protocolo *Stop-And-Wait*, um erro na trama transferida promove uma retransmissão, o que aumenta o tempo de transferência de cada pacote de dados e consequentemente o tempo global gasto no processo. A eficiência é por isso inversamente proporcional à taxa de erros provocada aquando da transmissão.

Conclusões

O protocolo de ligação de dados, constituído pela *LinkLayer* que se encarregou da interação com a porta série e de gerir as tramas de informação, e pela *ApplicationLayer*, uma camada que interagiu diretamente com o ficheiro a ser transferido, foi importante para a consolidação da matéria lecionada nas aulas teóricas. Assim, com este projeto interiorizamos os conceitos de *byte stuffing*, *framing* e o funcionamento do protocolo *Stop-and-Wait* e como este deteta erros e lida com eles.

Anexo I - link_layer.h

```
#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_RR(Nr) ((Nr << 7) | 0x05)
#define C_REJ(Nr) ((Nr << 7) | 0x01)
#define C_N(Ns) (Ns << 6)

typedef enum
{
   LlTx,
   LlRx,
} LinkLayerRole;

typedef enum
{
   START,
   FLAG_RCV,
```

```
    A_RCV,
    C_RCV,
    BCC1_OK,
    STOP_R,
    DATA_FOUND_ESC,
    READING_DATA,
    DISCONNECTED,
    BCC2_OK
} LinkLayerState;

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

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

// 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.
// if showStatistics == TRUE, link layer should print statistics in the console on
close.
// Return "1" on success or "-1" on error.
int llclose(int fd);

// retorna fd ou -1 se der erro
int connection(const char *serialPort);

// timeout
void alarmHandler(int signal);

unsigned char readControlFrame (int fd);

int sendSupervisionFrame(int fd, unsigned char A, unsigned char C);
```

```
#endif // _LINK_LAYER_H_
```

Anexo II - link_layer.c

```
#include "link_layer.h"

volatile int STOP = FALSE;
int alarmTriggered = FALSE;
int alarmCount = 0;
int timeout = 0;
int retransmitions = 0;
unsigned char tramaTx = 0;
unsigned char tramaRx = 1;

int llopen(LinkLayer connectionParameters) {

    LinkLayerState state = START;
    int fd = connection(connectionParameters.serialPort);
    if (fd < 0) return -1;

    unsigned char byte;
    timeout = connectionParameters.timeout;
    retransmitions = connectionParameters.nRetransmissions;
    switch (connectionParameters.role) {

        case LlTx: {

            (void) signal(SIGALRM, alarmHandler);
            while (connectionParameters.nRetransmissions != 0 && state != STOP_R) {

                sendSupervisionFrame(fd, A_ER, C_SET);
                alarm(connectionParameters.timeout);
                alarmTriggered = FALSE;

                while (alarmTriggered == FALSE && state != STOP_R) {
                    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_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_R;
                    else state = START;
                    break;
                default:
                    break;
            }
        }
    }
    connectionParameters.nRetransmissions--;
}
if (state != STOP_R) return -1;
break;
}

case LlRx: {

    while (state != STOP_R) {
        if (read(fd, &byte, 1) > 0) {
            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_R;
                            else state = START;
                            break;
                        default:
                            break;
                    }
                }
            }
            sendSupervisionFrame(fd, A_RE, C_UA);
            break;
        }
        default:
            return -1;
            break;
    }
    return fd;
}

int connection(const char *serialPort) {

    int fd = open(serialPort, O_RDWR | O_NOCTTY);
    if (fd < 0) {
        perror(serialPort);
        return -1;
    }

    struct termios oldtio;
    struct termios newtio;

    if (tcgetattr(fd, &oldtio) == -1)
    {
        perror("tcgetattr");
        exit(-1);
    }

    memset(&newtio, 0, sizeof(newtio));

    newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;
    newtio.c_lflag = 0;
    newtio.c_cc[VTIME] = 0;
    newtio.c_cc[VMIN] = 0;
```

```
    tcflush(fd, TCIOFLUSH);

    if (tcsetattr(fd, TCSANOW, &newtio) == -1) {
        perror("tcsetattr");
        return -1;
    }

    return fd;
}

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

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

    int frameSize = 6+bufSize;
    unsigned char *frame = (unsigned char *) malloc(frameSize);
    frame[0] = FLAG;
    frame[1] = A_ER;
    frame[2] = C_N(tramaTx);
    frame[3] = frame[1] ^frame[2];
    memcpy(frame+4,buf, bufSize);
    unsigned char BCC2 = buf[0];
    for (unsigned int i = 1 ; i < bufSize ; i++) BCC2 ^= buf[i];

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

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

    while (currentTransmition < retransmitions) {
        alarmTriggered = FALSE;
        alarm(timeout);
        rejected = 0;
```

```
        accepted = 0;
        while (alarmTriggered == FALSE && !rejected && !accepted) {

            write(fd, frame, j);
            unsigned char result = readControlFrame(fd);

            if(!result){
                continue;
            }
            else if(result == C_REJ(0) || result == C_REJ(1)) {
                rejected = 1;
            }
            else if(result == C_RR(0) || result == C_RR(1)) {
                accepted = 1;
                tramaTx = (tramaTx+1) % 2;
            }
            else continue;

        }
        if (accepted) break;
        currentTransmition++;
    }

    free(frame);
    if(accepted) return frameSize;
    else{
        llclose(fd);
        return -1;
    }
}

int llread(int fd, unsigned char *packet) {

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

    while (state != STOP_R) {
        if (read(fd, &byte, 1) > 0) {
            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_N(0) || byte == C_N(1)){
            state = C_RCV;
            cField = byte;
        }
        else if (byte == FLAG) state = FLAG_RCV;
        else if (byte == C_DISC) {
            sendSupervisionFrame(fd, A_RE, C_DISC);
            return 0;
        }
        else state = START;
        break;
    case C_RCV:
        if (byte == (A_ER ^ cField)) state = READING_DATA;
        else if (byte == FLAG) state = FLAG_RCV;
        else state = START;
        break;
    case READING_DATA:
        if (byte == ESC) state = DATA_FOUND_ESC;
        else if (byte == FLAG){
            unsigned char bcc2 = packet[i-1];
            i--;
            packet[i] = '\0';
            unsigned char acc = packet[0];

            for (unsigned int j = 1; j < i; j++)
                acc ^= packet[j];

            if (bcc2 == acc){
                state = STOP_R;
                sendSupervisionFrame(fd, A_RE, C_RR(tramaRx));
                tramaRx = (tramaRx + 1)%2;
                return i;
            }
            else{
                printf("Error: retransmition\n");
                sendSupervisionFrame(fd, A_RE, C_REJ(tramaRx));
                return -1;
            };

        }
        else{
            packet[i++] = byte;
        }
        break;
```

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

int llclose(int fd){

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

    while (retransmitions != 0 && state != STOP_R) {

        sendSupervisionFrame(fd, A_ER, C_DISC);
        alarm(timeout);
        alarmTriggered = FALSE;

        while (alarmTriggered == FALSE && state != STOP_R) {
            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_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_R;
                    else state = START;
                    break;
                default:
                    break;
            }
        }
    }
    retransmitions--;
}

if (state != STOP_R) return -1;
sendSupervisionFrame(fd, A_ER, C_UA);
return close(fd);
}

unsigned char readControlFrame(int fd){

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

    while (state != STOP_R && alarmTriggered == FALSE) {
        if (read(fd, &byte, 1) > 0 || 1) {
            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_RR(0) || byte == C_RR(1) || byte == C_REJ(0) || byte
== C_REJ(1) || 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_R;
                    }
                    else state = START;
                    break;
                default:
                    break;
            }
        }
    }
    return cField;
}

int sendSupervisionFrame(int fd, unsigned char A, unsigned char C){
    unsigned char FRAME[5] = {FLAG, A, C, A ^ C, FLAG};
    return write(fd, FRAME, 5);
}
```

Anexo III - application_layer.h

```
#ifndef _APPLICATION_LAYER_H_
#define _APPLICATION_LAYER_H_

#include <stdio.h>

// 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);

unsigned char* parseControlPacket(unsigned char* packet, int size, unsigned
long int *fileSize);

void parseDataPacket(const unsigned char* packet, const unsigned int
packetSize, unsigned char* buffer);
```

```
unsigned char * getControlPacket(const unsigned int c, const char* filename,
long int length, unsigned int* size);

unsigned char * getDataPacket(unsigned char sequence, unsigned char *data, int
dataSize, int *packetSize);

unsigned char * getData(FILE* fd, long int fileLength);

#endif // _APPLICATION_LAYER_H_
```

Anexo IV - application_layer.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>

void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename)
{
    LinkLayer linkLayer;
    strcpy(linkLayer.serialPort,serialPort);
    linkLayer.role = strcmp(role, "tx") ? LlRx : LlTx;
    linkLayer.baudRate = baudRate;
    linkLayer.nRetransmissions = nTries;
    linkLayer.timeout = timeout;

    int fd = llopen(linkLayer);
    if (fd < 0) {
        perror("Connection error\n");
        exit(-1);
    }

    switch (linkLayer.role) {
```

```
        case LlTx: {

            FILE* file = fopen(filename, "rb");
            if (file == NULL) {
                perror("File not found\n");
                exit(-1);
            }

            int prev = ftell(file);
            fseek(file,0L,SEEK_END);
            long int fileSize = ftell(file)-prev;
            fseek(file,prev,SEEK_SET);

            unsigned int cpSize;
            unsigned char *controlPacketStart = getControlPacket(2, filename, fileSize,
&cpSize);
            if(llwrite(fd, controlPacketStart, cpSize) == -1){
                printf("Exit: error in start packet\n");
                exit(-1);
            }

            unsigned char sequence = 0;
            unsigned char* content = getData(file, fileSize);
            long int bytesLeft = fileSize;

            while (bytesLeft >= 0) {

                int dataSize = bytesLeft > (long int) MAX_PAYLOAD_SIZE ?
MAX_PAYLOAD_SIZE : bytesLeft;
                unsigned char* data = (unsigned char*) malloc(dataSize);
                memcpy(data, content, dataSize);
                int packetSize;
                unsigned char* packet = getDataPacket(sequence, data, dataSize,
&packetSize);

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

                bytesLeft -= (long int) MAX_PAYLOAD_SIZE;
                content += dataSize;
                sequence = (sequence + 1) % 255;
            }

            unsigned char *controlPacketEnd = getControlPacket(3, filename, fileSize,
```

```
&cpSize);
            if(llwrite(fd, controlPacketEnd, cpSize) == -1) {
                printf("Exit: error in end packet\n");
                exit(-1);
            }
            llclose(fd);
            break;
        }

        case LlRx: {

            unsigned char *packet = (unsigned char *)malloc(MAX_PAYLOAD_SIZE);
            int packetSize = -1;
            while ((packetSize = llread(fd, packet)) < 0);
            unsigned long int rxFileSize = 0;
            unsigned char* name = parseControlPacket(packet, packetSize, &rxFileSize);

            FILE* newFile = fopen((char *) name, "wb+");
            while (1) {
                while ((packetSize = llread(fd, packet)) < 0);
                if(packetSize == 0) break;
                else if(packet[0] != 3){
                    unsigned char *buffer = (unsigned char*)malloc(packetSize);
                    parseDataPacket(packet, packetSize, buffer);
                    fwrite(buffer, sizeof(unsigned char), packetSize-4, newFile);
                    free(buffer);
                }
                else continue;
            }

            fclose(newFile);
            break;

        default:
            exit(-1);
            break;
    }}
}

unsigned char* parseControlPacket(unsigned char* packet, int size, unsigned long int
*fileSize) {

    // File Size
    unsigned char fileSizeNBytes = packet[2];
    unsigned char fileSizeAux[fileSizeNBytes];
    memcpy(fileSizeAux, packet+3, fileSizeNBytes);
```

```
    for(unsigned int i = 0; i < fileSizeNBytes; i++)
        *fileSize |= (fileSizeAux[fileSizeNBytes-i-1] << (8*i));

    // File Name
    unsigned char fileNameNBytes = packet[3+fileSizeNBytes+1];
    unsigned char *name = (unsigned char*)malloc(fileNameNBytes);
    memcpy(name, packet+3+fileSizeNBytes+2, fileNameNBytes);
    return name;
}

unsigned char * getControlPacket(const unsigned int c, const char* filename, long int
length, unsigned int* size){

    const int L1 = (int) ceil(log2f((float)length)/8.0);
    const int L2 = strlen(filename);
    *size = 1+2+L1+2+L2;
    unsigned char *packet = (unsigned char*)malloc(*size);

    unsigned int pos = 0;
    packet[pos++]=c;
    packet[pos++]=0;
    packet[pos++]=L1;

    for (unsigned char i = 0 ; i < L1 ; i++) {
        packet[2+L1-i] = length & 0xFF;
        length >>= 8;
    }
    pos+=L1;
    packet[pos++]=1;
    packet[pos++]=L2;
    memcpy(packet+pos, filename, L2);
    return packet;
}

unsigned char * getDataPacket(unsigned char sequence, unsigned char *data, int dataSize,
int *packetSize){

    *packetSize = 1 + 1 + 2 + dataSize;
    unsigned char* packet = (unsigned char*)malloc(*packetSize);

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

```
    return packet;
}

unsigned char * getData(FILE* fd, long int fileLength) {
    unsigned char* content = (unsigned char*)malloc(sizeof(unsigned char) * fileLength);
    fread(content, sizeof(unsigned char), fileLength, fd);
    return content;
}

void parseDataPacket(const unsigned char* packet, const unsigned int packetSize,
unsigned char* buffer) {
    memcpy(buffer,packet+4,packetSize-4);
    buffer += packetSize+4;
}
```