Faculdade de Engenharia da Universidade do Porto

Redes de Computadores

1º Trabalho Laboratorial

Turma 12 - Grupo 1

Afonso Neves (up202108884@up.pt)
João Miranda (up202003518@up.pt)

Porto, 8 de novembro de 2023

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.

Introdução

O objetivo deste trabalho foi o desenvolvimento de um protocolo de transmissão de dados usando uma Porta Série RS-232, de acordo com as indicações presentes no guião fornecido. O presente relatório é composto por sete secções:

- **Arquitetura:** Blocos funcionais e interfaces utilizadas.
- **Estrutura do código:** Apresentação das principais APIs, estruturas e funções.
- **Casos de uso principais:** Identificação do 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 aplicação e estratégias de implementação.
- **Validação:** Testes efetuados para avaliar a correção da implementação.
- **Conclusões:** Síntese da informação apresentada nas secções anteriores e reflexão sobre os objetivos de aprendizagem alcançados.

Arquitetura

Blocos Funcionais

Como dito anteriormente, o projeto foi dividido entre duas camadas principais: a *LinkLayer* e a *ApplicationLayer*.

A *LinkLayer*, ou camada de ligação de dados, é a camada que comunica diretamente com a porta-série. É responsável por abrir e fechar a comunicação, criar e transmitir as tramas, garantir que estas são recebidas sem erros e transmitir mensagens de erro em caso do mesmo.

A *ApplicationLayer*, ou camada de aplicação, é a camada mais próxima do utilizador que interage com a API da *LinkLayer* para transmissão e receção de pacotes de dados de um ficheiro. É nesta camada onde o utilizador pode especificar os aspetos da execução do programa como a velocidade de transmissão e o tamanho dos pacotes.

Interfaces

A execução do programa é realizada através de dois terminais, um em cada computador, sendo um deles executado em modo transmissor e o outro em modo recetor. O programa é executado através dos seguintes comandos.

Modo Transmissor:

```
$ make run_tx
```

Modo Recetor:

```
$ make run_rx
```

O recetor também utiliza o seguinte comando para verificar a validade do ficheiro recebido.

Verificar ficheiro recebido:

```
$ make check_files
```

Estrutura do código

Link Layer

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

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

typedef struct
{
    const char *serialPort;
    LinkLayerRole role;
    int baudRate;
    int nRetransmissions;
    int timeout;
} LinkLayer;
```

```
typedef enum {
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    READING_DATA,
    FOUND_ESC,
    AFTER_ESC,
    BCC1_OK,
    BCC2_OK,
    STOP
} LinkLayerState;
```

As *macros* utilizadas nesta camada foram:

```
// MACRO do número máximo de bytes por pacote
#define MAX_PAYLOAD_SIZE 1000

// MACROS dos valores booleanos
#define FALSE 0
#define TRUE 1
```

```c
// MACRO da FLAG
#define FLAG 0x7E

// MACROS relativas ao procedimento de Byte Stuffing
#define ESC 0x7D
#define ESC_1 0x5D
#define ESC_2 0x5E

// MACROS do byte de endereço
#define A_ER 0x03
#define A_RE 0x01

// MACROS do byte de controlo
#define C_SET 0x03
#define C_UA 0x07
#define C_DISC 0x0B
#define C_I(N) (N << 6)
#define C_RR(N) ((N << 7) | 0x05)
#define C_REJ(N) ((N << 7) | 0x01)
```

As funções implementadas foram:

```c
// Estabelecimento da ligação entre transmissor e recetor
int llopen(LinkLayer connectionParameters);

// Envio de tramas
int llwrite(int fd, const unsigned char *buf, int bufSize);

// Receção de tramas
int llread(int fd, unsigned char *packet);

// Termino da ligação
int llclose(int fd, LinkLayerRole role);

// Configuração do alarme
void alarmHandler(int signal);

// Envio de tramas de supervisão
int sendControlFrame(int serialPort, unsigned char A, unsigned char C);

// Receção e interpretação de tramas de supervisão
unsigned char checkControlFrame(int serialPort, unsigned char A);

// Configuração da porta série
int openSerialPort(const char *serialPort, int baudRate);
```

Application Layer

Nesta camada não houve necessidade de implementar estrutura de dados auxiliares. As funções implementadas foram:

```
// Principal função da camada
void applicationLayer(const char *serialPort, const char *role, int
baudRate, int nTries, int timeout, const char *filename);

// Leitura dos dados do ficheiro a transferir
unsigned char *getFileData(FILE* file, long int fileSize);

// Gera o nome do ficheiro a receber.
char *getNewFilename(const char *filename, const char *appendix, int
fileNameSize);

// Interpretação de um pacote de controlo
char *parseControlPacket(unsigned char *packet, long int *fileSize, const
char *appendix);

// Criação de um pacote de dados
unsigned char *getDataPacket(unsigned char *data, int dataSize, int
*packetSize);

// Criação de um pacote de controlo
unsigned char *getControlPacket(unsigned char C, const char *filename,
long int fileSize, int *packetSize);
```

Casos de uso principais

Como já foi referido, o programa comporta-se de maneira diferente consoante o modo em que está a ser executado. Dependendo de qual modo for escolhido, as funções utilizadas e a sequência de chamadas serão diferentes.

Modo Transmissor

1) **llopen()**, usada para estabelecer ligação entre o transmissor e o recetor, através da troca de pacotes de controlo e conexão com a porta através de **openSerialPort().**
2) **getFileData()**, usada para obter o conteúdo do ficheiro a ser transferido.
3) **getControlPacket()**, usada para criar pacotes de controlo.
4) **getDataPacket()**, usada para criar pacotes de dados do ficheiro.
5) **llwrite(),** usada para enviar pela porta série uma trama de informação com base no pacote recebido como argumento.
6) **checkControlFrame()**, usada como máquina de estados que lê e valida tramas de supervisão.
7) **llclose()**, usada para terminar a ligação entre o transmissor e o recetor, através da troca de tramas de supervisão.

Modo Recetor

1) **llread()**, usada como uma máquina de estados que gere e valida a receção de tramas de informação e/ou supervisão.
2) **sendSupervisionFrame()**, usada para criar e enviar tramas de supervisão.
3) **parseControlPacket()**, usada para obter as informações do ficheiro a ser transferido, através da interpretação do pacote de controlo passado como argumento.
4) **getNewFilename()**, usada para gerar o nome do ficheiro recebido.

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 transmissor e o recetor. Para isso, foi implementado a estratégia de Stop-and-Wait para um melhor controlo de erros.

A função **llopen()** tem como principal objetivo configurar a ligação entre as duas máquinas. Inicialmente, após abrir e configurar a porta série através da função *`openSerialPort()`*, o transmissor envia uma trama de supervisão SET e espera que o recetor responda com uma trama de supervisão UA. Se este receber a trama UA, a ligação foi bem estabelecida. O recetor também abre e configura a porta série através da função *`openSerialPort()`*, mas, ao contrário do emissor, fica a espera de uma trama SET, e na sequência da receção desta envia uma trama de supervisão UA.

A função **llwrite()** tem como principal objetivo enviar tramas de informação. Esta função recebe um pacote e aplica-lhe a estratégia de *Byte Stuffing*, de modo a prevenir erros de bytes com valor igual à flag e ao caracter de ESC. Posteriormente transforma esse pacote numa trama de informação, recorrendo à estratégia de framing, envia-o para o recetor e espera por uma 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 função **llread()** tem como principal objetivo ler tramas da porta série e verificar a sua integridade. Para isso, a função tem implementada uma máquina de estados que permite percorrer o array de bytes recebidos individualmente de forma a conseguir fazer o processo inverso de *Byte Stuffing* e por fim validar os valores dos bytes BCC1 e BCC2 para verificar a existência de erros na transmissão. Caso a trama tenha sido recebida corretamente, será enviada uma trama de supervisão RR, e no caso de erro/falha, uma tram de supervisão REJ.

Por fim, a função **llclose()** tem como principal objetivo terminar a ligação. Para atingir esse objetivo, a função realiza troca de tramas de supervisão. Inicialmente o transmissor envia uma trama de supervisão DISC, a que o recetor também responde com outra trama de supervisão DISC. A partir daí, o transmissor envia uma trama de supervisão UA e fecha a porta. O recetor ao interpretar a trama UA fecha a porta e dá-se por concluído o programa.

Protocolo de aplicação

A camada da aplicação é a camada que interage diretamente com o utilizador. Nela, o utilizador pode definir o ficheiro a transferir, a porta série a usar, a velocidade da transferência, o número de bytes do ficheiro inseridos cada pacote, o número máximo de retransmissões e o tempo máximo de espera da resposta por parte do recetor. Com estes dados, a camada utiliza a API da *LinkLayer* para transferir o ficheiro através de tramas de informação.

A execução desta camada começa com o estabelecimento da ligação entre os dois computadores. A seguir, a função *`getFileData()`* carrega os bytes do ficheiro a transferir para um array. O primeiro pacote a ser enviado pelo transmissor é um pacote de controlo START e contém os dados do ficheiro em formato TLV (Type, Length, Value), onde é especificado o tamanho do ficheiro e o nome do mesmo. Do lado do recetor, esse pacote é interpretado pela função *`parseControlPacket()`* de forma a criar e alocar o espaço necessário para receber o ficheiro.

Na fase seguinte, a camada fragmenta o array com os dados do ficheiro em pacotes de tamanho predefinido pelo utilizador. Cada um destes fragmentos é inserido num pacote de dados através da função *`getDataPacket()`* e enviado pela porta série usando a função *`llwrite()`* presente na API da *LinkLayer*. Já o recetor apenas tem de receber o pacote, através da função *`llread()`*, também da API da *LinkLayer,* retirar os bytes de controlo e colocar o pacote no novo ficheiro.

Por fim, o transmissor envia o último pacote de dados seguindo os passos descritos anteriormente e envia um pacote de controlo END a sinalizar o fim da transmissão do ficheiro. A ligação entre as duas máquinas termina quando é invocada a função da API *`llclose()`*.

Validação

Ao longo do desenvolvimento, o programa foi testado de várias formas para garantir um protocolo correto e coerente. Os testes realizados foram os seguintes:

- ✓ Transferência de ficheiros com nomes distintos.
- ✓ Transferência de ficheiros com tamanhos distintos.
- ✓ Transferência de ficheiros com diferentes velocidades de transferência.
- ✓ Transferência de ficheiros com pacotes de dados de tamanhos distintos.
- ✓ Transferência de ficheiros com interrupção parcial e/ou total da porta série.

Ao longo do desenvolvimento da aplicação, foram usados/implementados testes unitários de forma a assegurar o funcionamento de cada aspeto da aplicação. Como esperado, os testes foram reproduzidos com sucesso durante a apresentação do trabalho.

Conclusões

O protocolo foi desenvolvido de acordo com as especificações indicadas, sendo o protocolo de ligação lógica responsável pela preparação da ligação porta-série e, também, pelo tratamento de erros e o protocolo de aplicação responsável pela interpretação de informação necessária à execução do programa, bem como pela leitura e escrita do conteúdo do ficheiro. Com este projeto consolidamos os conceitos de *Byte Stuffing*, *Framing* e o funcionamento do protocolo *Stop-and-Wait* e como este deteta e lida com erros.

Anexo I – link_layer.h

```
// Link layer protocol header.

#ifndef _LINK_LAYER_H_
#define _LINK_LAYER_H_

#include <math.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <termios.h>
#include <sys/stat.h>
#include <sys/types.h>

typedef enum
{
    LlTx,
    LlRx,
} LinkLayerRole;

typedef struct
{
    const char *serialPort;
    LinkLayerRole role;
    int baudRate;
    int nRetransmissions;
    int timeout;
} LinkLayer;

typedef enum {
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    READING_DATA,
    FOUND_ESC,
    AFTER_ESC,
    BCC1_OK,
    BCC2_OK,
    STOP
} LinkLayerState;

// ----- MACROS -----

#define MAX_PAYLOAD_SIZE 1000

#define FALSE 0
#define TRUE 1
```

```c
// FRAMES

#define FLAG 0x7E
#define ESC 0x7D
#define ESC_1 0x5D
#define ESC_2 0x5E

#define A_ER 0x03
#define A_RE 0x01

#define C_SET 0x03
#define C_UA 0x07
#define C_DISC 0x0B
#define C_I(N) (N << 6)
#define C_RR(N) ((N << 7) | 0x05)
#define C_REJ(N) ((N << 7) | 0x01)

// ----- MAIN FUNCTIONS -----

// Sets up the connection with the serial port.
int llopen(LinkLayer connectionParameters);

// Sends the data in buf.
int llwrite(int fd, const unsigned char *buf, int bufSize);

// Receives the data and writes it to the packet.
int llread(int fd, unsigned char *packet);

// Closes the connection with the serial port.
int llclose(int fd, LinkLayerRole role);

// ----- AUX FUNCTIONS -----

// Default alarm handler.
void alarmHandler(int signal);

// Send a control frame with the parameters provided.
int sendControlFrame(int serialPort, unsigned char A, unsigned char C);

// Checks the received control frame and returns its control byte.
unsigned char checkControlFrame(int serialPort, unsigned char A);

// Open serial port and set its parameters.
int openSerialPort(const char *serialPort, int baudRate);

#endif // _LINK_LAYER_H_
```

Anexo II – link_layer.c

```
// Link layer protocol implementation

#include "link_layer.h"

int alarmCount = 0;
int alarmEnabled = FALSE;
int timeout = 0;
int nRetransmissions = 0;

unsigned char tramaT = 0;
unsigned char tramaR = 0;

struct termios oldtio;
struct termios newtio;

void alarmHandler(int signal)
{
    alarmEnabled = FALSE;
    alarmCount--;

    printf("--- Alarm Alert! ---\n");
}

int sendControlFrame(int serialPort, unsigned char A, unsigned char C)
{
    unsigned char frame[5] = {FLAG, A, C, A ^ C, FLAG};
    return write(serialPort, &frame, 5);
}

unsigned char checkControlFrame(int serialPort, unsigned char A)
{
    unsigned char prevAlarmStatus = alarmEnabled;
    alarmEnabled = TRUE;

    unsigned char byte, C = 0xFF;
    LinkLayerState state = START;

    while (state != STOP && alarmEnabled == TRUE)
    {
        if (read(serialPort, &byte, 1) > 0)
        {
            switch (state)
            {

            case START:
                if (byte == FLAG)
                    state = FLAG_RCV;
                break;

            case FLAG_RCV:
                if (byte == A)
```

```
                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 || byte == C_UA ||
                byte == C_SET)
            {
                state = C_RCV;
                C = byte;
            }
            else if (byte == FLAG)
                state = FLAG_RCV;
            else
                state = START;
            break;

        case C_RCV:
            if (byte == (A ^ C))
                state = BCC1_OK;
            else if (byte == FLAG)
                state = FLAG_RCV;
            else
                state = START;
            break;

        case BCC1_OK:
            if (byte == FLAG)
            {
                // Received Valid Control Frame!
                if (prevAlarmStatus == TRUE) alarm(0);
                alarmEnabled = FALSE;
                state = STOP;
            }
            else
                state = START;
            break;

        default:
            break;
        }
    }
}

    return C;
}
```

```
int openSerialPort(const char *serialPort, int baudRate)
{
    int fd = open(serialPort, O_RDWR | O_NOCTTY);

    if (fd < 0) return -1;

    if (tcgetattr(fd, &oldtio) == -1) return -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] = 5; // Inter-character timer unused
    newtio.c_cc[VMIN] = 0;  // Read without blocking
    tcflush(fd, TCIOFLUSH);

    if (tcsetattr(fd, TCSANOW, &newtio) == -1) return -1;

    return fd;
}

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

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

    switch (connectionParameters.role)
    {
    case LlTx:
    {
        (void)signal(SIGALRM, alarmHandler);
        while (1)
        {
            if (alarmCount == 0) return -1;

            if (alarmEnabled == FALSE)
            {
                sendControlFrame(fd, A_ER, C_SET);
                alarm(timeout);
                alarmEnabled = TRUE;
            }

            if (checkControlFrame(fd, A_RE) == C_UA) break;
        }
        break;
    }
```

```
    case LlRx:
    {
        if (checkControlFrame(fd, A_ER) != C_SET) return -1;

        sendControlFrame(fd, A_RE, C_UA);

        break;
    }

    default:
        return -1;
    }

    return fd;
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(int fd, const unsigned char *buf, int bufSize)
{
    int frameSize = bufSize + 6;
    unsigned char *frame = (unsigned char *) malloc(frameSize);

    frame[0] = FLAG;
    frame[1] = A_ER;
    frame[2] = C_I(tramaT);
    frame[3] = frame[1] ^ frame[2];
    memcpy(frame + 4, buf, bufSize);

    unsigned char BCC2 = buf[0];
    for (int i = 1; i < bufSize; i++)
        BCC2 ^= buf[i];

    int i, bytesAdded = 0;
    for (i = 4; i < frameSize - 2; i++)
    {
        if (frame[i] == FLAG)
        {
            frameSize += 2; bytesAdded += 2;
            frame = realloc(frame, frameSize);

            memmove(frame + i + 2, frame + i, frameSize - i - 2);

            frame[i++] = ESC;
            frame[i++] = ESC_1;
            frame[i] = ESC_2;
        }
        else if (frame[i] == ESC)
        {
            frame = realloc(frame, ++frameSize); bytesAdded++;

            memmove(frame + i + 1, frame + i, frameSize - i - 1);
```

```
            frame[i++] = ESC;
            frame[i] = ESC_1;
        }
    }

    frame[i++] = BCC2;
    frame[i] = FLAG;

    unsigned char C;
    LinkLayerState state = START;
    alarmCount = nRetransmissions;

    (void)signal(SIGALRM, alarmHandler);
    while (alarmCount != 0 && state != STOP)
    {
        if (alarmEnabled == FALSE)
        {
            write(fd, frame, frameSize);
            alarm(timeout);
            alarmEnabled = TRUE;
        }

        C = checkControlFrame(fd, A_RE);

        if (C == C_RR(0) || C == C_RR(1))
        {
            state = STOP;
            tramaT = (tramaT + 1) % 2;

            if (frameSize == bufSize + 6 + bytesAdded) printf("OK [%d,%d,%d]\n\n",
bufSize, bytesAdded, frameSize);
            else printf("ERROR\n\n");
        }

        if (C == C_REJ(0) || C == C_REJ(1))
        {
            alarm(0);
            alarmEnabled = FALSE;
        }
    }

    free(frame);
    return (state == STOP) ? frameSize : -1;
}
```

```
////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
int llread(int fd, unsigned char *packet)
{
    int packetPos = 0, bytesRemoved = 0;
    unsigned char byte, C;
    LinkLayerState state = START;

    while (1)
    {
        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_I(0) || byte == C_I(1))
                {
                    state = C_RCV;
                    C = byte;
                }
                else if (byte == FLAG)
                    state = FLAG_RCV;
                else
                    state = START;
                break;

            case C_RCV:
                if (byte == (A_ER ^ C))
                    state = READING_DATA;
                else if (byte == FLAG)
                    state = FLAG_RCV;
                else
                    state = START;
                break;

            case READING_DATA:
                if (byte == ESC)
                    state = FOUND_ESC;
                else if (byte == FLAG)
                {
                    unsigned char BBC2 = packet[--packetPos];
```

```
                    packet[packetPos] = '\0';

                    unsigned char checkBBC2 = packet[0];
                    for (int j = 1; j < packetPos + 1; j++)
                        checkBBC2 ^= packet[j];

                    if (BBC2 == checkBBC2)
                    {
                        printf("OK [%d,%d,%d]\n\n", packetPos, bytesRemoved, packetPos
+ 6 + bytesRemoved);

                        tramaR = (tramaR + 1) % 2;
                        sendControlFrame(fd, A_RE, C_RR(tramaR));
                        return packetPos;
                    }
                    else
                    {
                        printf("ERROR\n\n");

                        sendControlFrame(fd, A_RE, C_REJ(tramaR));
                        return -1;
                    }
                }
                else
                    packet[packetPos++] = byte;
                break;

            case FOUND_ESC:
                if (byte == ESC_1)
                    state = AFTER_ESC;
                else
                {
                    packet[packetPos++] = ESC;
                    packet[packetPos++] = byte;
                    state = READING_DATA;
                }
                break;

            case AFTER_ESC:
                state = READING_DATA;
                if (byte == ESC_2)
                {
                    packet[packetPos++] = FLAG;
                    bytesRemoved += 2;
                }
                else if (byte == ESC)
                {
                    packet[packetPos++] = ESC;
                    state = FOUND_ESC;
                    bytesRemoved++;
                }
                else
                {
                    packet[packetPos++] = ESC;
                    packet[packetPos++] = byte;
```

```
                    bytesRemoved++;
                }
                break;

            default:
                return -1;
            }
        }
    }

    exit(-1);
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int fd, LinkLayerRole role)
{
    switch (role)
    {

    case LlTx:
    {
        alarmCount = nRetransmissions;

        (void)signal(SIGALRM, alarmHandler);
        while (1)
        {
            if (alarmCount == 0) return -1;

            if (alarmEnabled == FALSE)
            {
                sendControlFrame(fd, A_ER, C_DISC);
                alarm(timeout);
                alarmEnabled = TRUE;
            }

            if (checkControlFrame(fd, A_RE) == C_DISC) break;
        }

        sendControlFrame(fd, A_ER, C_UA);

        sleep(1);

        break;
    }

    case LlRx:
    {
        alarmCount = nRetransmissions;

        if (checkControlFrame(fd, A_ER) != C_DISC) return -1;

        (void)signal(SIGALRM, alarmHandler);
```

```
        while (1)
        {
            if (alarmCount == 0) return -1;

            if (alarmEnabled == FALSE)
            {
                sendControlFrame(fd, A_RE, C_DISC);
                alarm(timeout);
                alarmEnabled = TRUE;
            }

            if (checkControlFrame(fd, A_ER) == C_UA) break;
        }

        break;
    }

    default:
        return -1;
    }

    if (tcsetattr(fd, TCSANOW, &oldtio) == -1) return -1;

    close(fd);

    return 0;
}
```

Anexo III – application_layer.h

```
// Application layer protocol header.

#ifndef _APPLICATION_LAYER_H_
#define _APPLICATION_LAYER_H_

#include <stdio.h>
#include <stdlib.h>

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

// Fetches the file data.
unsigned char *getFileData(FILE* file, long int fileSize);

// Generates the new filename.
char *getNewFilename(const char *filename, const char *appendix, int fileNameSize);

// Parses a control packet.
char *parseControlPacket(unsigned char *packet, long int *fileSize, const char
*appendix);
```

```
// Creates a data packet with the given parameters.
unsigned char *getDataPacket(unsigned char *data, int dataSize, int *packetSize);

// Creates a control packet with the given parameters.
unsigned char *getControlPacket(unsigned char C, const char *filename, long int
fileSize, int *packetSize);

#endif // _APPLICATION_LAYER_H_
```

Anexo IV – application_layer.c

```
// Application layer protocol implementation

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

const char *nameAppendix = "-received";

long int totalBytesSent = 0;
long int totalBytesReceived = 0;

void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename)
{
    LinkLayer linkLayer;
    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) exit(-1);

    printf("Connection established\n\n");

    switch (linkLayer.role)
    {
        case LlTx:
        {
            FILE* file = fopen(filename, "rb");
            if (file == NULL) exit(-1);

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

            printf("File Size: %ld\n\n", fileSize);

            int cpSize;
            unsigned char *controlPacket_Start = getControlPacket(2, filename,
fileSize, &cpSize);
```

```
            printf("Starter Control Packet: ");

            if (llwrite(fd, controlPacket_Start, cpSize) == -1) exit(-1);

            long int bytesLeft = fileSize;
            unsigned char *fileData = getFileData(file, fileSize);

            int payloadNumber = 1;
            while (bytesLeft > 0)
            {
                int dataSize = bytesLeft > (long int) MAX_PAYLOAD_SIZE ?
MAX_PAYLOAD_SIZE : bytesLeft;
                unsigned char *data = (unsigned char *) malloc(dataSize);

                memcpy(data, fileData, dataSize);

                int packetSize;
                unsigned char *packet = getDataPacket(data, dataSize, &packetSize);

                printf("Payload #%d: ", payloadNumber);

                if (llwrite(fd, packet, packetSize) == -1) exit(-1);

                fileData += dataSize; totalBytesSent += dataSize; payloadNumber++;
                bytesLeft -= (long int) MAX_PAYLOAD_SIZE;
            }

            unsigned char *controlPacket_End = getControlPacket(3, filename, fileSize,
&cpSize);

            printf("Ending Control Packet: ");

            if(llwrite(fd, controlPacket_End, cpSize) == -1) exit(-1);

            printf("Total Bytes Sent: %ld\n\n", totalBytesSent);

            llclose(fd, LlTx);

            printf("Connection closed\n\n");

            break;
        }

        case LlRx:
        {
            unsigned char *packet = (unsigned char *) malloc(MAX_PAYLOAD_SIZE + 3);

            printf("Starter Control Packet: ");

            int packetSize = -1;
            while ((packetSize = llread(fd, packet)) < 0);

            long int fileSize = 0;
            char *name = parseControlPacket(packet, &fileSize, nameAppendix);
```

```
            printf("File Size: %ld\n\n", fileSize);
            printf("File Name: %s\n\n", name);

            FILE* newFile = fopen(filename, "wb+");
            if (newFile == NULL) exit(-1);

            int payloadNumber = 1;
            LinkLayerState state = START;
            while (state != STOP)
            {
                printf("Payload #%d: ", payloadNumber);

                while ((packetSize = llread(fd, packet)) < 0);

                if (packet[0] == 1)
                {
                    unsigned char *buffer = (unsigned char *) malloc(packetSize - 3);
                    memcpy(buffer, packet + 3, packetSize - 3);
                    fwrite(buffer, sizeof(unsigned char), packetSize - 3, newFile);
                    free(buffer);
                }
                else if (packet[0] == 3)
                {
                    printf("Total Bytes Received: %ld\n\n", totalBytesReceived);

                    long int fileSize2;
                    name = parseControlPacket(packet, &fileSize2, nameAppendix);

                    if ((fileSize == fileSize2))
                    {
                        fclose(newFile);
                        state = STOP;
                    }
                }

                payloadNumber++;
                totalBytesReceived += packetSize - 3;
            }

            llclose(fd, LlRx);

            printf("Connection closed\n\n");

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

```
unsigned char *getFileData(FILE* file, long int fileSize)
{
    unsigned char *data = (unsigned char *) malloc(sizeof(unsigned char) * fileSize);

    fread(data, sizeof(unsigned char), fileSize, file);

    fclose(file);

    return data;
}

char *getNewFilename(const char *filename, const char *appendix, int fileNameSize)
{
    const char *dotPosition = strrchr(filename, '.');

    char *newFilename = (char *) malloc(fileNameSize);

    strncpy(newFilename, filename, (int)(dotPosition - filename));

    newFilename[(int)(dotPosition - filename)] = '\0';

    strcat(newFilename, appendix);

    strcat(newFilename, dotPosition);

    return newFilename;
}

unsigned char *getControlPacket(unsigned char C, const char *filename, long int
fileSize, int *packetSize)
{
    int fileSizeBytes = (int) ceil(log2f((float) fileSize) / 8.0);
    int fileNameBytes = strlen(filename);

    *packetSize = 5 + fileSizeBytes + fileNameBytes;
    unsigned char *packet = (unsigned char *) malloc(*packetSize);

    int packetPos = 0;
    packet[packetPos++] = C;
    packet[packetPos++] = 0;
    packet[packetPos++] = (unsigned char) fileSizeBytes;

    for (int i = 0 ; i < fileSizeBytes ; i++)
    {
        packet[3 + i] = (unsigned char) ((fileSize >> (8 * i)) & 0xFF);
        packetPos++;
    }

    packet[packetPos++] = 1;
    packet[packetPos++] = (unsigned char) fileNameBytes;

    memcpy(packet + packetPos, filename, fileNameBytes);

    return packet;
}
```

```
unsigned char *getDataPacket(unsigned char *data, int dataSize, int *packetSize)
{
    *packetSize = 3 + dataSize;

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

    packet[0] = 1;
    packet[1] = (unsigned char) dataSize / 256;
    packet[2] = (unsigned char) dataSize % 256;

    memcpy(packet + 3, data, dataSize);

    return packet;
}

char *parseControlPacket(unsigned char *packet, long int *fileSize, const char
*appendix)
{
    int fileSizeBytes = packet[2];
    int fileNameBytes = packet[4 + fileSizeBytes];

    for (int i = fileSizeBytes - 1; i >= 0; i--)
        *fileSize = (*fileSize << 8) | packet[3 + i];

    unsigned char *filename = (unsigned char *) malloc(fileNameBytes + 1);

    memcpy(filename, packet + 5 + fileSizeBytes, fileNameBytes);

    filename[fileNameBytes] = '\0';

    char *newFileName = getNewFilename((const char *) filename, appendix,
fileNameBytes + strlen(appendix) + 1);

    return newFileName;
}
```