1º Trabalho laboratorial

Protocolo de Ligação de Dados

Licenciatura em Engenharia Informática e Computação

Redes de Computadores

Turma 9 | Grupo 1:

Inês Oliveira - up202003343

Pedro Macedo - up202007531

Sandra Miranda - up202007675

2º Semestre

Ano Letivo 2022/2023

Sumário

No âmbito da unidade curricular Redes de Computadores, da licenciatura em engenharia informática e computação, foi nos proposto a resolução do 1º trabalho laboratorial com o intuito de implementar um protocolo de ligação de dados, na linguagem C, através de uma porta de série.

O trabalho foi realizado com sucesso, já que a aplicação é capaz de transmitir corretamente de um computador para o outro, sem erros.

Introdução

O principal objetivo deste trabalho consiste na implementação de um protocolo de transferência de dados que permita o envio de informação entre dois computadores através de uma porta de série RS-232.

O relatório encontra-se estruturado da seguinte forma:

- **Arquitetura** - blocos funcionais e interfaces.
- **Estrutura do Código** – APIs, principais estruturas de dados, principais funções e a sua relação com a arquitetura.
- **Casos de uso principais** - identificação; sequências de chamada de funções.
- **Protocolo de ligação lógica** - identificação dos principais aspetos funcionais; descrição da estratégia de implementação destes aspetos com apresentação de extratos de código.
- **Protocolo de aplicação** - identificação dos principais aspetos funcionais; descrição da estratégia de implementação destes aspetos com apresentação de extratos de código.
- **Validação** - descrição dos testes efetuados com apresentação quantificada dos resultados, se possível.
- **Eficiência do protocolo de ligação de dados** - caraterização estatística da eficiência do protocolo, efetuada recorrendo a medidas sobre o código desenvolvido.
- **Conclusões** - síntese da informação apresentada nas secções anteriores; reflexão sobre os objetivos de aprendizagem alcançados.

Arquitetura

O trabalho desenvolvido consiste em duas partes, o application_layer e o link_layer.

A funções principais são criadas e realizadas no link_layer, sendo que a finalidade do application_layer limita-se a chamar essas funções e a ler o ficheiro. As funções do link_layer fazem recurso às funções da state machine, para processamento da informação lida na porta de serie.

Estrutura do código

No início do programa, começamos por analisar os dados recebidos e guardá-los num objeto da struct LinkLayer. De seguida, chamamos o llopen, enviando o objeto criado e guardando os dados recebidos em variáveis globais (role, baudRate, nRetransmissions e timeout).

Application_layer.c:

Nesta camada encontra-se a função principal, main. Fazendo deste o ponto principal do programa.

Link_layer.c:

Funções principais da camada de ligação:

- llopen() – Abre a porta de série e envia a trama SET e recebe a trama UA.
- llwrite() – efetua byte stuffing das frames de informação e envia para o recetor;
- llread() – Lê as tramas de informação recebidas pelo emissor e envia a trama de supervisão (RR ou REJ);
- llclose() – Efetua a troca da trama DISC e o emissor envia a trama UA. Por fim, fecha a ligação entre as portas de série.

O programa contempla mais um modulo adicional, State Machine, seno este usado para o processamento de dados recebidos na porta de série.

Casos de uso principais

Para que o utilizador possa usar a nossa aplicação, este deve compilar a mesma, executando o comando “make”, dentro da pasta src. De seguida, deve abrir dois terminais distintos, executando, num deles, o comando “make run_tx” (para executar o transmissor) e no outro executando o comando “make run_rx” (para executar o recetor).

O primeiro comando a ser executado deve ser o do recetor, de modo que este espere que o transmissor lhe envie a mensagem de início de ligação. Caso o transmissor seja executado em primeiro, este tentará enviar a mensagem de início de ligação até receber a confirmação do recetor (UA) ou o número de tentativas de retransmissão for atingido.

Numa segunda fase (depois da conexão ser estabelecida), vai começar a analise do ficheiro recebido, sendo disponibilizadas mensagens de erro ou mensagens de aviso, em caso de falha.

Protocolo de ligação lógica

State Machine:

À medida que o código foi sendo criado, reparamos que havia partes do código que eram repetitivas (mais especificamente, na análise dos dados recebidos na porta de série). De modo a resolver este problema, criamos um ficheiro, state.c, que consiste em funções com a finalidade de analisar os dados recebidos.

Na primeira função, changeState, esta recebe o estado atual da State Machine, o caracter lido na porta de série, o campo de endereço A, campo de controlo C. Assim, para o estado atual, este verifica se o caracter recebido foi o pretendido para o estado atual. Caso seja o caracter pretendido, passa para o estado seguinte.

As outras funções são similares à função descrita em cima, mas nestes casos mudam o estado ao processar o pacote de informação e ao processar as tramas de supervisão.

llopen:

No início do programa, começamos por configurar a porta série e guardar as configurações da porta em variáveis globais, recorrendo à função *saveConnectionParameters*.

No decorrer da escrita do código do programa, verificamos que a função *read* ficava bloqueada à espera de conseguir ler uma resposta. Para resolver este problema, adicionamos uma flag à configuração da porta O_NONBLOCK.

De seguida, fazemos um switch case de modo a distinguir o código do transmissor do código do recetor. Na parte do transmissor, este chama a função sendReadyToTransmitMsg que cria o SET (prepareSet) e envia-o para a porta série (sendSet – utiliza a função write).

De seguida, fazemos um switch case de modo a distinguir o código do transmissor do código do recetor. Na parte do transmissor, este trata de criar a trama SET, iniciar um ciclo de escrita e verificar se o recetor recebeu a trama corretamente (quando este recebe a trama UA recetor). Já no recetor, este inicia um clico de leitura, lê a trama da porta série e verifica se está correta, utilizando uma máquina de estados que só acaba quando receber todos os dados da trama corretamente. Caso a trama tenha sido recebida como pretendido, o recetor prepara a trama UA e envia-a para a porta série.

Finalmente, assim que ambas as partes do programa conseguirem estabelecer uma ligação, a função llopen foi finalizada com sucesso imprimindo uma mensagem de sucesso. Caso ocorra algum tipo de erro, esta imprime uma mensagem no terminal explícita para o erro em questão e retorna -1, sendo que no *Application_layer*, caso esta situação ocorra, a porta série é fechada.

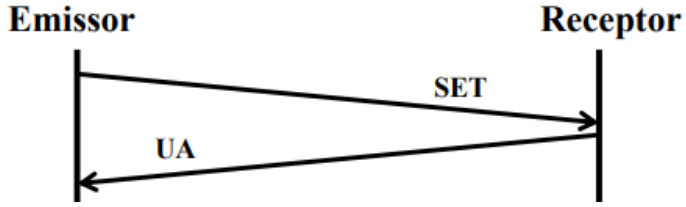

Figura 1 Fase 1 - Estabelecimento da conexão

llwrite:

De acordo com o guião do trabalho, esta função recebe como parâmetros os dados/pacotos de controlo a transmitir (buf) e o seu comprimento (bufSize), sendo o máximo a transmitir de 100 bytes (sendo que este valor pode ser mudado, quando estamos a ler o ficheiro). Caso os bytes que restam de ler do ficheiro seja menor do que o valor pedido, este cria um pacote com tamanho correspondente ao valor retornado pela função fread.

De seguida, este cria a trama de informação, adicionando os cabeçalhos ao parâmetro recebido. O primeiro cabeçalho é composto por 4 bytes: Flag (0x7E), campo de endereço (0x03), campo de endereço (0x03), campo de controlo que difere com o número da sequência do pacote (0x00 ou 0x40), campo de informação (dados recebidos como parâmetro) e o BCC1 (XOR entre o campo de endereço e o campo de controlo). De seguida, o BCC2 é calculando fazendo o XOR dos dados recebidos no array. E, por fim, faz o byte stuffing dos dados e do BCC2.

Finalmente, cria a trama de informação (composta pelo primeiro cabeçalho, os dados recebidos e, por último, o BCC2 e a FLAG). Antes de a função retornar, o número da sequência do transmissor é atualizado.

llread:

Esta função recebe como parâmetros um array de caracteres. Para analisar os dados que recebeu este utiliza uma *state machine*. Durante esta análise, esta função só guarda num array final o parâmetro de dados da trama fazendo *destuffing* ao mesmo tempo.

O processo de *destuffing* consiste no processo contrário ao stuffing, isto é, quando encontrar os bytes 0x7D e 0x5E seguidos, significa que, antes do *byte stuffing*, estávamos perante o byte 0x7E. Caso encontre os bytes 0x7D e 0x5D, significa que o byte, antes do *byte stuffing*, é 0x7D.

Este só guarda da trama recebida o parâmetro dos dados (depois de ler o BCC1) e o BCC2.

De seguida, calcula o BCC2 dos dados lidos e o BCC2 recebido. Caso sejam iguais, significa que não ocorreu nenhum erro no processo de transferência mandando a resposta RR para o transmissor, alterando também o número do recetor (*receiverNumber*). Caso estes dois parâmetros sejam diferentes, o recetor manda a resposta REJ ao transmissor.

llclose:

De forma semelhante ao llopen, o llclose começa por separar o código do transmissor do código do recetor.

O transmissor envia a trama DISC (constituída pela FLAG, campo de endereço, o campo de controlo - 0x0B e o BCC1), começando o alarme enquanto espera pela resposta do recetor. Na parte do recetor, caso o DISC recebido esteja correto, este envia a resposta DISC (semelhante ao enviado pelo transmissor, mas o campo de endereço é 0x01) ao transmissor, e começa o alarme à espera da resposta UA do transmissor. Caso o transmissor receba o DISC do recetor corretamente, envia o UA (consiste na FLAG, campo de endereço, campo de controlo - 0x07 e o BCC1), fecha a porta série e retorna da função de seguida. Caso o recetor receba corretamente o UA, este fecha a porta série e retorna da função, terminado assim o programa.

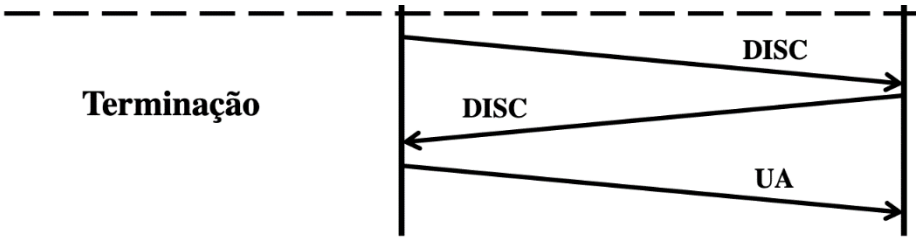

Figura 2 Fase 3 – Termino da conexão da ligação

Protocolo de aplicação

O programa começa pela análise dos parâmetros recebidos como parâmetros. De seguida, o programa procede para a execução da função *llopen* para abrir a ligação entre o transmissor e o recetor. Após estes passos serem concluídos, o programa executará o código dependendo do *role* (transmissor ou recetor). No final, a função *llclose* é chamada no *application_layer*, para que ambas as aplicações em execução terminem o programa.

Por um lado, na parte do transmissor, é criado o pacote de controlo (contendo as informações do ficheiro, com uma flag de *start* - **C**), sendo que este é enviado pata a porta série através da função *llread*. Por outro lado, o recetor ao receber este pacote de controlo, manda uma mensagem ao transmissor (RR, em caso de sucesso, ou REJ, em caso contrário). De seguida, é iniciado um ciclo *while*, onde o transmissor cria um pacote de dados (**C** = 1, número de sequência, número de octetos do campo de dados $\boldsymbol{L_2 L_1}$ e campo de dados) lido do ficheiro, sendo este enviado com o uso da função *llwrite* para o Recetor, até não haver mais ficheiros por ler do ficheiro. No final da transferência do ficheiro é enviado o mesmo pacote de controlo, inicialmente mandado, para o Recetor, com a única diferença em que a flag passa a ser a flag de *end*, para indicar o final da transferência do ficheiro.

Por outro lado, na parte do recetor, é construído um ciclo *while* com a finalidade de ler os dados da porta série e de os processar, tendo como condição de paragem o pacote de controlo *end*. No final da função, é chamado o *fclose* do ficheiro criado de modo a não haver problemas de criação do ficheiro. Dentro deste ciclo, é feita a verificação dos dados recebidos, e caso o byte lido seja 0x03, então o ciclo é terminado e o ficheiro criado é fechado. Caso o byte lido seja 0x01 (pacoto de dados), então adicionasse os dados recebidos (sem o *header* da trama recebida) para o novo ficheiro.

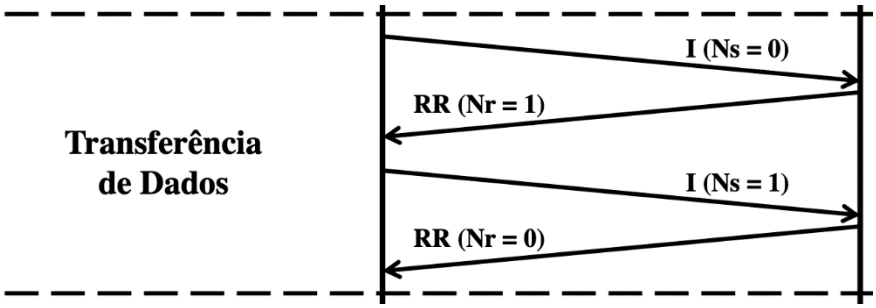

Figura 3 Fase 2 - Transferência de dados

Validação

Após a transferência dos dados ser bem-sucedida, foi necessário tentar encontrar erros do nosso programa. Assim, começámos por interromper a ligação via porta série no *llopen*, fazendo com que ocorra *timeout* do lado do transmissor, pois este não recebe a trama de UA do emissor, tentando enviar até o máximo de tentativas definidas no início do programa (*nRetransmissons*). Consequentemente, o transmissor imprime o número de alarmes/tentativas que já fez e a mensagem de aviso até receber a trama UA ou o número de tentativas máximas ser atingida.

Num segundo teste simularam-se erros nas tramas de informação, mudando os bytes BCC1 e BCC2 da trama. Estes têm como função encontrar erros nas tramas (um dos erros poderia ser no processo de *byte stuffing*). Quando ocorre um *reject (REJ)*, este envia de novo a mesma trama de informação para o recetor.

Eficiência do protocolo de ligação de dados

Para obter a eficiência do nosso programa, começámos por medir o tempo que o programa demora a ler o ficheiro, a enviar os dados e, finalmente, a criar o novo ficheiro *duration*). Também disponibilizamos o número de tramas escritas no ficheiro, o número de tramas que tiveram de ser retransmitidas, o número de bytes total e o número de *timeouts* (quando o alarme é acionado).

No caso do ficheiro do penguin.gif, este apresenta um tamanho de 10968 bytes e 110 tramas de informação. Neste caso, o tempo de execução foi, em média, de 0.437376 segundos.

Sendo este um mecanismo de *Stop&Wait* é necessário um *acknoledgment* por cada trama enviada, o que afeta a eficiência do programa.

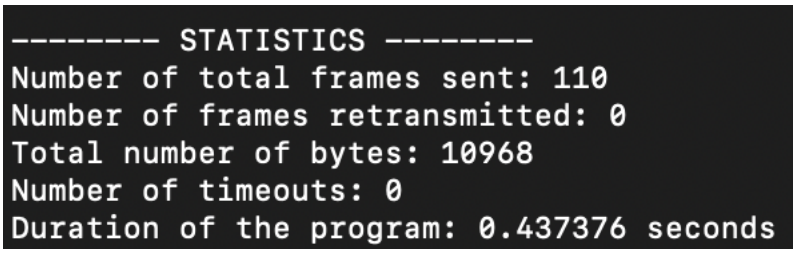

Figura 4 Estatísticas

Conclusões

Neste relatório descrevemos como pode ser feita a transferência de dados pela porta série, descrevendo a estratégia de implementação do protocolo, as funções implementadas, os testes realizados e os resultados obtidos. Foi possível verificar que o código consegue lidar com os erros ao longo da execução do mesmo.

Com este projeto, tivemos a oportunidade de aprender e compreender a implementação de protocolos de comunicação (porta série).

Consideramos que implementamos o protocolo de comunicação com sucesso, testando sempre o nosso programa com erros precisos.

Anexo I - Código Fonte

Application_layer.c

```
// Application layer protocol implementation

#include "../include/application_layer.h"
#define BUF_SIZE 256
#define BUF_SIZE2 400

// Statistics variables
int totalFrames = 0;


int prepareControlPacket(unsigned char *controlPacket, int bufSize, unsigned char C, int fileSize, const char
*filename) {
    controlPacket[0] = C;
    controlPacket[1] = 0;

    unsigned char len; // this will be used to hold strlens
    unsigned char sizeStr[10];
    sprintf(sizeStr, "%d", fileSize); // stores the size of the file in a string
    unsigned char length = strlen(sizeStr); // Stores the length of the sizeStr variable

    len = strlen(sizeStr); // Size of the sizeStr variable

    controlPacket[2] = len;

    memcpy(controlPacket + 3, sizeStr, len); // (controlPacket + 3) is positioned on the first byte of the V1 field

    unsigned int currentPos=3+len; // Position the head of the array in the T2 field

    controlPacket[currentPos] = 1; // Indicates the V2 field contains the name of the file
    currentPos++; // Position the head of the array in the L2 field

    len = strlen(filename); // Size of the variable filename
    controlPacket[currentPos] = len; // Indicates the size of the variable to be read in the V2 field
    currentPos++; // Position the head of the array in the first byte of the V2 field
    memcpy(controlPacket+currentPos, filename, len);

    currentPos += len; // Corresponds to the size of the controlPacket array

    return currentPos; // Goes to llwrite!
}

int prepareDataPacket(unsigned char *dataBytes, unsigned char *dataControlPacket, int numSequence, int
numBytes) {
    dataControlPacket[0] = 0x01;
    dataControlPacket[1] = numSequence;
    dataControlPacket[2] = numBytes / 255;
    dataControlPacket[3] = numBytes % 255;
```

```
    for(int i=0;i<numBytes;i++) {
        dataControlPacket[i+4] = dataBytes[i];
    }

    return (numBytes + 4);
}


void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename) {
    // Emissor
    int resTx = strcmp(role,"tx");
    int resRx = strcmp(role,"rx");

    LinkLayer defs;
    strcpy(defs.serialPort, serialPort);
    if(resTx == 0){
        defs.role = LITx; // TRANSMITTER
    } else if(resRx == 0) {
        defs.role = LIRx; // RECEIVER
    } else {
        printf("ERROR: Invalid role!\n");
        return;
    }
    defs.baudRate = baudRate;
    defs.nRetransmissions = nTries;
    defs.timeout = timeout;

    printf("\n\n----------  Phase 1 : Establish connection ----------\n\n");

    // Open the connection between the 2 devices and prepare to send and receive
    if (llopen(defs) < 0) {
        printf("ERROR: Couldn't receive UA from receiver!\n");
        return;
    }

    printf("\n(Connection established successfully)\n");
    printf("\n\n----------  Phase 2 : Start reading the penguim file ----------\n\n");

    unsigned char controlPacket[600];
    struct stat st;
    stat(filename, &st);
    int fileSize = st.st_size;

    clock_t start, end;
    int statistics = 1;
    int totalBytes = 0;

    start = clock();
    if (resTx == 0) {
```

```
    // 1. Create controlPacket
    // 2. Call the llwrite to create the info frame
    // 2. Make byte stuffing on the array
    // 3. Send the information frame to the receiver
    int controlPacketSize = prepareControlPacket(controlPacket, BUF_SIZE, 2, fileSize, filename);

    if (llwrite(controlPacket, controlPacketSize) != 0) {
        printf("ERROR: llwrite() failed!\n");
        return;
    }

    unsigned char dataPacket[BUF_SIZE], dataBytes[BUF_SIZE];
    int dataPacketSize = 0;
    FILE* filePtr;
    int numSequence = 0;

    filePtr = fopen(filename, "rb");
    if (filePtr == NULL) {
        printf("ERROR: Failed to read from file with name '%s'\n", filename);
        return;
    }

    // Start reading the file
    int numBytesRead = 0;
    while ((numBytesRead = fread(dataBytes, (size_t) 1, (size_t) 100, filePtr)) > 0) {
        totalFrames++;
        printf("[LOG] Reading from file\n");

        // Create data packet
        int dataPacketSize = prepareDataPacket(dataBytes, dataPacket, numSequence++, numBytesRead);

        if (llwrite(dataPacket, dataPacketSize) < 0) {
            printf("ERROR: Failed to write data packet to llwrite!\n");
            return;
        }

        totalBytes += numBytesRead;
    }

    // Create control packet end
    int controlPacketSizeEnd = prepareControlPacket(controlPacket, BUF_SIZE, 3, fileSize, filename);

    if (llwrite(controlPacket, controlPacketSizeEnd) < 0) {
        printf("ERROR: Failed to write control packet end to llwrite!\n");
        return;
    }

    fclose(filePtr);
}
else if (resRx == 0) {
```

```
        // 1. Read the information frame
        // 2. Send the response (RR or REJ) to the transmitter

        unsigned char readInformation[BUF_SIZE2];
        int bytesReadCTRLPacket = 0;

        FILE* fileCreating = fopen(filename, "wb");

        if((bytesReadCTRLPacket = llread(controlPacket)) > 0) {
            if(controlPacket[0] == 0x02)
                printf("Received control packet START on application_layer!\n");
            else
                printf("ERROR: Couldn't read control packet!\n");
        }


        int bytesread = 0, totalBytesRead = 0;

        while((bytesread = llread(readInformation)) > 0) { // data + BCC2
            // Choose between writting or closing on the file based on frame received
            if (readInformation[0] == 0x01){
                totalBytesRead += (bytesread - 5);
                unsigned char fileData[bytesread - 5];

                for (int i = 4; i < bytesread - 1; i++) {
                    fileData[i - 4] = readInformation[i]; // Just the penguin
                }

                for (int i = 0; i < (bytesread - 5); i++) {
                    fputc(fileData[i], fileCreating);
                }
            }
            else if(readInformation[0] == 0x03){
                printf("Received control packet END on application_layer!\n");
                break;
            }
        }

        fclose(fileCreating);
    }
    else {
        printf("ERROR: Invalid role!\n");
        exit(1);
    }
    end = clock();
    float duration = ((float)end - start) / CLOCKS_PER_SEC;

    printf("\n\n------------  Phase 3 : Close connection ----------\n\n");

    llclose(&statistics, totalFrames, totalBytes, duration);
}
```

link_layer.c

```
// Link layer protocol implementation

#include "../include/link_layer.h"
#include "fcntl.h"
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>
#include <stdlib.h>
#include <signal.h>
#include <errno.h>

int temp = 0;

#define FALSE 0
#define TRUE 1

#define FLAG 0x7E
#define A_SET 0x03
#define A_UA 0x03
#define C_SET 0x03
#define C_UA 0x07

#define C_RR0 0x05  // receiver ready (c = 0)
#define C_RR1 0x85  // receiver ready (c = 1)
#define C_REJ0 0x01 // receiver rejected (c = 0)
#define C_REJ1 0x81 // receiver rejected (c = 1)
#define C_S0 0x00   // transmitter -> receiver
#define C_S1 0x40   // receiver -> transmitter

#define FLAG1 0x7D
#define FLAG2 0x5E
#define FLAG3 0x5D

#define BUF_SIZE 256
#define BAUDRATE B38400

// Safe connectionParameters variable info
char serialPort[50];
LinkLayerRole role;
int baudRate;
int nRetransmissions;
int timeout;

unsigned char SET[5];
unsigned char UA[5];
unsigned char buf[BUF_SIZE];
char currentC = C_S1;
```

```c
int receiverNumber = 1, senderNumber = 0;
int controlReceiver;

int fd; // serial file descriptor
LinkLayer connectionParametersGlobal;

// statitistics variables
int numFramesRetransmitted = 0;
int numTimeOuts = 0;


// MISC
#define _POSIX_SOURCE 1 // POSIX compliant source

int alarmEnabled = FALSE;
int alarmCount = 0;

// Alarm function handler
void alarmHandler(int signal) {
    if (alarmCount == 0) numFramesRetransmitted++;
    numTimeOuts++;
    alarmEnabled = FALSE;
    alarmCount++;

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

// Starts the alarm
int startAlarm(int timeout) {
    // Set alarm function handler
    (void)signal(SIGALRM, alarmHandler);
    alarmEnabled = FALSE;
    alarm(timeout);
    alarmEnabled = TRUE;

    return 0;
}

void prepareSet(){
    SET[0] = FLAG;
    SET[1] = A_UA;
    SET[2] = C_SET;
    SET[3] = A_UA ^ C_SET;
    SET[4] = FLAG;
}

int sendSet(int fd)
{
    int sentBytes = 0;
    sentBytes = write(fd, SET, 5);
```

```
    printf("Sent SET to receiver!\n");

    return sentBytes;
}

void prepareUA()
{
    UA[0] = FLAG;
    UA[1] = A_UA;
    UA[2] = C_UA;
    UA[3] = A_UA ^ C_UA;
    UA[4] = FLAG;
}

int sendUA(int fd)
{
    int sentBytes = 0;
    sentBytes = write(fd, UA, 5);

    printf("Sent UA for transmitter!\n");

    return sentBytes;
}

enum state receiveUA(int fd, unsigned char A, unsigned char C)
{
    enum state STATE = START;
    char buf;

    while (STATE != STOP && alarmEnabled == TRUE) {
        int bytes = read(fd, &buf, 1);

        if (bytes > 0) {
            STATE = changeState(STATE, buf, A, C);
        }
    }

    return STATE;
}

void receiveSET(int fd, unsigned char A, unsigned char C) {
    enum state STATE = START;
    unsigned char buf;

    while (STATE != STOP) {
        int bytes = read(fd, &buf, 1);

        if (bytes > 0) {
            STATE = changeState(STATE, buf, A, C);
        }
    }
```

```
}

int sendReadyToReceiveMsg(int fd) { // Send UA
    prepareUA();
    if (sendUA(fd) < 0) {
        printf("ERROR: sendReadyToReceiveMsg failed!\n");
        return -1;
    }
    return 0;
}

int sendReadyToTransmitMsg(int fd) { // send SET
    prepareSet();
    if (sendSet(fd) < 0) {
        printf("ERROR: sendReadyToReceiveMsg failed!\n");
        return -1;
    }
    return 0;
}

void saveConnectionParameters(LinkLayer connectionParameters) {
    for (int i = 0; i < 50; i++) {
        serialPort[i] = connectionParameters.serialPort[i];
    }

    role = connectionParameters.role;
    baudRate = connectionParameters.baudRate;
    nRetransmissions = connectionParameters.nRetransmissions;
    timeout = connectionParameters.timeout;
}

////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters) {
    // Save the connectionParameters variable info into global variables
    saveConnectionParameters(connectionParameters);

    (void)signal(SIGALRM, alarmHandler);

    connectionParametersGlobal = connectionParameters;
    fd = open(connectionParameters.serialPort, O_RDWR | O_NOCTTY | O_NONBLOCK);

    if (fd < 0) {
        perror(connectionParameters.serialPort);
        return -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] = 1;  // Blocking read until 1 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_sele/* code */
// Set new port settings
if (tcsetattr(fd, TCSANOW, &newtio) == -1) {
    perror("tcsetattr");
    exit(-1);
}

switch (connectionParameters.role) {
case LITx:
    alarmCount = 0;
    enum state STATE;

    do {
        if (alarmEnabled == FALSE) {
            if (sendReadyToTransmitMsg(fd) < 0) {
                printf("ERROR: Failed to send SET!\n");
                continue;
            }
            startAlarm(connectionParameters.timeout);
        }
        // Read UA
        prepareUA();
        STATE = receiveUA(fd, A_UA, C_UA);

        if(STATE != STOP){
            //reset alarm
            alarm(0);
```

```
            alarmEnabled = FALSE;
        }
    } while (alarmCount < connectionParameters.nRetransmissions && STATE != STOP);

    if (alarmCount < connectionParameters.nRetransmissions) {
        printf("Received UA from receiver successfully!\n");
        //reset alarm
        alarm(0);
        alarmEnabled = FALSE;
        alarmCount = 0;
    }
    else {
        printf("ERROR: Failed to receive UA from receiver!\n");
        return -1;
    }

    break;
case LlRx:
    receiveSET(fd, A_SET, C_SET); // prepare UA
    printf("Received SET from transmitter successfully!\n");

    if (sendReadyToReceiveMsg(fd) < 0) {
        printf("ERROR: Failed to send UA!\n");
    }
    break;
default:
    printf("ERROR: Unknown role!\n");
    llclose(0, 0, 0, 0);
}

return fd;
}

int stuffing(unsigned char *frame, char byte, int i, int countStuffings) {
if (byte == FLAG) {
    frame[4 + i + countStuffings] = FLAG1;
    frame[4 + i + countStuffings + 1] = FLAG2;
    countStuffings++;
}
else if (byte == FLAG1) { // byte is equal to the octeto
    frame[4 + i + countStuffings + 1] = FLAG3;
}
else {
    frame[4 + i + countStuffings] = byte;
}

return countStuffings;
}

/**
 * @brief
```

```
*
* @param buf (control Packet) C | T1 | L1 | V1 | T2 | L2 | V2 | ...
* @param bufSize
* @param C
* @param infoFrame
* @return int
*/

int prepareInfoFrame(const unsigned char *buf, int bufSize, unsigned char *infoFrame) {
    infoFrame[0] = FLAG;
    infoFrame[1] = A_SET;
    infoFrame[2] = (senderNumber << 6); // Changes between C_S0 AND C_S1
    infoFrame[3] = A_SET ^ (senderNumber << 6);

    // Store data
    char bcc2 = 0x00; // Vaiable to store the XOR while going through the data
    for (int i = 0; i < bufSize; i++) {
        bcc2 ^= buf[i];
    }

    // Byte stuffing of the data (buf)
    int index = 4;

    for (int i = 0; i < bufSize; i++) {
        if (buf[i] == 0x7E) {
            infoFrame[index++] = 0x7D;
            infoFrame[index++] = 0x5E;
        }
        else if (buf[i] == 0x7D) {
            infoFrame[index++] = 0x7D;
            infoFrame[index++] = 0x5D;
        }
        else {
            infoFrame[index++] = buf[i];
        }
    }

    // Byte stuffing of the BCC2 before storing it in the array
    if (bcc2 == 0x7E) {
        infoFrame[index++] = 0x7D;
        infoFrame[index++] = 0x5E;
    }
    else if (bcc2 == 0x7D) {
        infoFrame[index++] = 0x7D;
        infoFrame[index++] = 0x5D;
    }
    else {
        infoFrame[index++] = bcc2;
    }

    infoFrame[index++] = FLAG; // Determines the end of the array
```

```
    return index;
}

int readReceiverResponse() {
    unsigned char buf[5] = {0};

    while(alarmEnabled == TRUE){
        int readedBytes = read(fd, buf, 5);

        int verifyReceiverRR, verifyReceiverREJ;

        if (senderNumber == 0) {
            verifyReceiverRR = 0x05 | 0x80; // RR (receiver ready)
            verifyReceiverREJ = 0x01; // REJ (receiver reject)
        }
        else {
            verifyReceiverRR = 0x05; // RR (receiver ready)
            verifyReceiverREJ = 0x81; // REJ (receiver reject)
        }

        if (readedBytes != -1 && buf[0] == FLAG) {
            if ((buf[2] == verifyReceiverRR) && (buf[3] == (buf[1] ^ buf[2]))) {
                alarmEnabled = FALSE;

                if (senderNumber == 1)
                    senderNumber = 0;
                else
                {
                    senderNumber = 1;
                }

                return 1; // SUCCESS
            }
            else if ((buf[2] == verifyReceiverREJ) && (buf[3] == (buf[1] ^ buf[2]))) {
                printf("Received REJ!\n");
                return -2;
            }
            else {
                printf("\nERROR: Received message from llread() incorrectly!\n");
                return -1; // INSUCCESS
            }
        }
    }

    return -1; // INSUCCESS
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
```

```c
/**
 * @brief
 *
 * @param buf (controlPacket created in the applicationLayer)
 * @param bufSize
 * @return int
 */
int llwrite(const unsigned char *buf, int bufSize) {
    // 1. Create the info frame -> DONE
    // 2. Make byte stuffing of the data received (controlPacket - buf) -> DONE

    unsigned char *infoFrame = malloc(sizeof(unsigned char) * (4 + (bufSize * 2) + 2));

    int totalBytes = prepareInfoFrame(buf, bufSize, infoFrame);

    int STOP = FALSE;
    int timerReplaced = 0;

    do {
        printf("[LOG] Writing Information Frame.\n");
        if (alarmEnabled == FALSE) {
            int res = write(fd, infoFrame, totalBytes);
            if (res < 0) {
                printf("ERROR: Failed to send infoFrame!\n");
                continue;
            }
            printf("Sent information frame successfully!\n");

            startAlarm(timeout);
        }

        // Read receiverResponse
        int res = readReceiverResponse();
        if(res == 1) {
            STOP = TRUE;
            alarm(0);
            printf("Received response from llread() successfully!\n");
        } else if (res == -1) {
            printf("ERROR: Couldn't resend info frame.\n");
            alarm(0);
            alarmEnabled = FALSE;
        } else if(res == -2) {
            printf("Received REJ response.\n");
            alarm(0);
            alarmEnabled = FALSE;
            alarmCount++;
        }
    } while (alarmCount < nRetransmissions && STOP == FALSE);

    alarmCount = 0;
```

```
    if(STOP == FALSE){
        return -1;
    }

    return 0;
}

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////

int receiveInfoFrame(unsigned char *packet, unsigned char *buf) {
    int STATE = packSTART;
    unsigned char ch;
    int packetidx = 0;
    int currentPos = 0;
    int foundBCC1 = 0;
    int readBytes = TRUE;
    int transparencyElse = FALSE;

    while (STATE != packSTOP) {
        if (read(fd, &ch, 1) < 0) {

            //printf("ERROR: Can't read from infoFrame!\n");
            continue;
        }

        STATE = changeInfoPacketState(STATE, ch, !(receiverNumber), buf, &currentPos, &foundBCC1,
&transparencyElse);

        switch (STATE) {
        case packSTOP:
            printf("Received Information frame from llwrite() successfully!\n");
            break;
        case packBCC1_RCV:
            if (transparencyElse == TRUE) {
                packet[packetidx++] = buf[currentPos-1];
                packet[packetidx++] = buf[currentPos];
                transparencyElse = FALSE;
            }
            packet[packetidx++] = buf[currentPos];
            break;
        case packTRANSPARENCY_RCV:
            break;
        case packERROR:
            break;
        }
    }

    readBytes = FALSE;
```

```
    return currentPos;
}

int createRR(unsigned char *respondRR)
{
    respondRR[0] = FLAG;  // F
    respondRR[1] = A_SET; // A
    respondRR[2] = (receiverNumber << 7) | 0x05;
    respondRR[3] = respondRR[1] ^ respondRR[2]; // BCC1
    respondRR[4] = FLAG;                        // F

    return 0;
}

int sendRR(unsigned char *respondRR) {
    if (write(fd, respondRR, 5) < 0) {
        printf("ERROR: Failed to sendRR() (link_layer.c)!\n");
        return -1;
    }
    printf("Sent RR successfully!\n");

    return 0;
}

void createREJ(unsigned char *respondREJ) {
    respondREJ[0] = FLAG;  // F
    respondREJ[1] = A_SET; // A
    respondREJ[2] = (receiverNumber << 7) | 0x01;
    respondREJ[3] = respondREJ[1] ^ respondREJ[2]; // BCC1
    respondREJ[4] = FLAG;                          // F
}

int sendREJ(unsigned char *respondREJ) {
    if (write(fd, respondREJ, 5) < 0) {
        printf("ERROR: Failed to sendREJ() (link_layer.c)!\n");
        return -1;
    }
    printf("REJ sent successfully!\n");

    return 0;
}

int llread(unsigned char *packet) {
    unsigned char buf[1000];

    int numBytesRead = receiveInfoFrame(packet, buf);

    // CREATE BCC2

    int bcc2Received = buf[numBytesRead - 1];
```

```
    int bcc2 = 0x00;
    for (int i = 0; i < numBytesRead - 1; i++) {
        bcc2 ^= buf[i];
    }

    if (bcc2 == bcc2Received) { // Create RR
        unsigned char respondRR[5];

        if (createRR(respondRR) < 0) {
            printf("ERROR: createRR() failed in llread (link_layer.c)!\n");
            return -1;
        }

        if (sendRR(respondRR) < 0) {
            printf("ERROR: sendRR() failed in llread (link_layer.c)!\n");
            return -1;
        }
        if (receiverNumber == 0) {
            receiverNumber = 1;
        }
        else {
            receiverNumber = 0;
        }
    }
    else { // Create REJ
        unsigned char respondREJ[5];
        createREJ(respondREJ);
        alarmEnabled = FALSE;

        if (sendREJ(respondREJ) < 0) {
            printf("ERROR: sendREJ() failed in llread (link_layer.c)!\n");
            return -1;
        }

        return numBytesRead; // REJ sent
    }

    for(int i=0;i<numBytesRead;i++)
        packet[i] = buf[i];

    return numBytesRead;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int *statistics, int totalFrames, int totalBytes, float duration) {
    alarmCount = 0;

    if(role == LlRx){
```

```
    unsigned char buf[6] = {0}, parcels[6] = {0};
    unsigned char STOP = FALSE, UA = 0;

    buf[0] = 0x7E;
    buf[1] = 0x03;
    buf[2] = 0x0B;
    buf[3] = buf[1] ^ buf[2];
    buf[4] = 0x7E;
    buf[5] = '\0';


    while(STOP == FALSE) {
        int result = read(fd, parcels, 5);

        if (result < 0) {
            continue;
        }

        parcels[5] = '\0';

        if (strcasecmp(buf, parcels) == 0){
            printf("\nDISC message received. Responding now.\n");

            buf[1] = 0x01;
            buf[3] = buf[1] ^ buf[2];

            while(alarmCount < nRetransmissions){

                if(!alarmEnabled){
                    printf("\nDISC message sent, %d bytes written\n", 5);
                    write(fd, buf, 5);
                    startAlarm(timeout);
                }

                int result = read(fd, parcels, 5);
                if(result != -1 && parcels != 0 && parcels[0]==0x7E){
                    //se o UA estiver errado
                    if(parcels[2] != 0x07 || (parcels[3] != (parcels[1]^parcels[2]))){
                        printf("\nUA not correct: 0x%02x%02x%02x%02x%02x\n", parcels[0], parcels[1], parcels[2],
parcels[3], parcels[4]);
                        alarmEnabled = FALSE;
                        continue;
                    }

                    else{
                        printf("\nUA correctly received: 0x%02x%02x%02x%02x%02x\n", parcels[0], parcels[1],
parcels[2], parcels[3], parcels[4]);
                        alarmEnabled = FALSE;
                        close(fd);
                        return 1;
                    }
```

```
            }

        }

        if(alarmCount >= nRetransmissions){
            printf("\nAlarm limit reached, DISC message not sent\n");
            return -1;
        }

        STOP = TRUE;
      }

    }

  }
  else {
    alarmCount = 0;

    unsigned char buf[6] = {0}, parcels[6] = {0};

    buf[0] = 0x7E;
    buf[1] = 0x03;
    buf[2] = 0x0B;
    buf[3] = buf[1]^buf[2];
    buf[4] = 0x7E;
    buf[5] = '\0'; //assim posso usar o strcmp

    while(alarmCount < nRetransmissions){

      if(!alarmEnabled) {

        int bytes = write(fd, buf, 5);
        printf("\nDISC message sent, %d bytes written\n", bytes);
        startAlarm(timeout);
      }

      int result = read(fd, parcels, 5);

      if (result < 0) {
        continue;
      }

      buf[1] = 0x01;
      buf[3] = buf[1] ^ buf[2];
      parcels[5] = '\0';

      if(result != -1 && parcels != 0 && parcels[0]==0x7E){
        //se o DISC estiver errado
        if(strcasecmp(buf, parcels) != 0){
          printf("\nDISC not correct: 0x%02x%02x%02x%02x%02x\n", parcels[0], parcels[1], parcels[2],
parcels[3], parcels[4]);
```

```
                alarmEnabled = FALSE;
                continue;
            }

            else{
                printf("\nDISC correctly received: 0x%02x%02x%02x%02x%02x\n", parcels[0], parcels[1], parcels[2],
parcels[3], parcels[4]);
                alarmEnabled = FALSE;

                buf[1] = 0x01;
                buf[2] = 0x07;
                buf[3] = buf[1]^buf[2];

                int bytes = write(fd, buf, 5);

                close(fd);

                printf("\nUA message sent, %d bytes written.\n\nI'm shutting off now, bye bye!\n", bytes);
                break;
            }
        }

    }

    if(alarmCount >= nRetransmissions){
        printf("\nAlarm limit reached, DISC message not sent\n");
        close(fd);
        return -1;
    }
  }

  if (statistics) {
    printf("\n-------- STATISTICS --------\n");
    printf("Number of total frames sent: %d\n", totalFrames);
    printf("Number of frames retransmitted: %d\n", numFramesRetransmitted);
    printf("Total number of bytes: %d\n", totalBytes);
    printf("Number of timeouts: %d\n", numTimeOuts);
    printf("Duration of the program: %f seconds\n", duration);
  }

  return 1;
}
```

state.c

```
#include "../include/state.h"
#include <string.h>
```

```
stateMachineInfo stateMachine;

#define TRUE 1
#define FALSE 0

#define FLAG 0x7E
#define A_SET 0x03
#define A_UA 0x03
#define C_SET 0x03
#define C_UA 0x07

#define C_S0 0x00 // transmitter -> receiver
#define C_S1 0x40 // receiver -> transmitter
#define RR0 0x05
#define RR1 0x85
#define REJ0 0x01
#define REJ1 0x81

enum state changeState(enum state STATE, unsigned char ch, unsigned char A, unsigned char C) {
    switch (STATE) {
        case START: // Start node (waiting fot the FLAG)
            if (ch == FLAG) {
                STATE = FLAG_RCV; // Go to the next state
            } // else { stay in the same state }
            break;
        case FLAG_RCV: // State Flag RCV
            if (ch == A) {
                STATE = A_RCV; // Go to the next state
            }
            else if (ch == FLAG) {
                STATE = FLAG_RCV; // Stays on the same state
            }
            else {
                STATE = START; // other character received goes to the initial state
            }
            break;
        case A_RCV: // State A RCV
            if (ch == C) {
                STATE = C_RCV; // Go to the next state
            }
            else if (ch == FLAG) {
                STATE = FLAG_RCV;
            }
            else {
                STATE = START;
            }
            break;
        case C_RCV: // State C RCV
            if (ch == (A ^ C)) {
                STATE = BCC_OK; // Go to the next state
            }
```

```
        else if (ch == FLAG) {
            STATE = FLAG_RCV;
        }
        else {
            STATE = START;
        }
        break;
    case BCC_OK: // State BCC_OK
        if (ch == FLAG) {
            STATE = STOP; // Go to the final state
        }
        else {
            STATE = START;
        }
        break;
    default:
        break;
  }

  return STATE;
}


enum stateInfoPacket changeInfoPacketState(enum stateInfoPacket STATE, unsigned char ch, int
senderNumber, unsigned char *buf, int *currentPos, int *foundBCC1, int *transparencyElse) {
  switch (STATE) {
    case packSTART:
        if (ch == FLAG) {
            STATE = packFLAG1_RCV;
        }
        break;
    case packFLAG1_RCV:
        if (ch == A_SET) {
            STATE = packA_RCV;
        }
        else if (ch == FLAG) {
            STATE = packFLAG1_RCV;
        }
        else {
            STATE = packSTART; // other character received goes to the initial state
        }
        break;
    case packA_RCV:
        if (ch == (senderNumber << 6)) {
            STATE = packC_RCV;
        }
        else if (ch == FLAG) {
            STATE = packFLAG1_RCV;
        }
        else {
            STATE = packSTART;
```

```
            }
            break;
        case packC_RCV:;
            char expectedBCC1 = (A_SET ^ (senderNumber << 6));
            if (ch == expectedBCC1) {
                *foundBCC1 = 1;
                STATE = packBCC1_RCV;
                currentPos = 0;
            }
            else if (ch == FLAG) {
                STATE = packFLAG1_RCV;
            }
            else {
                STATE = packSTART;
            }
            break;
        case packBCC1_RCV:
            if (ch == 0x7D) {
                STATE = packTRANSPARENCY_RCV;
            }
            else if (ch == FLAG) {
                STATE = packSTOP;
            }
            else {
                buf[(*currentPos)++] = ch;
                STATE = packBCC1_RCV;
            }
            break;
        case packTRANSPARENCY_RCV:
            if (ch == 0x5E) {
                STATE = packBCC1_RCV;
                buf[(*currentPos)++] = FLAG;
            }
            else if (ch == 0x5D) {
                STATE = packBCC1_RCV;
                buf[(*currentPos)++] = 0x7D; // octeto
            }
            else {
                STATE = packBCC1_RCV;
                buf[(*currentPos)++] = 0x7D;
                buf[(*currentPos)++] = ch;
                *transparencyElse = TRUE;
            }
            break;
        default:
            break;
    }

    return STATE;
}
```

```
unsigned char superviseTrama[2] = {0};

int changeStateSuperviseTrama(unsigned char buf, enum state *STATE) {
    switch ((*STATE)) {
        case START: // Start node (waiting fot the FLAG)
            if (buf == FLAG) {
                STATE = FLAG_RCV; // Go to the next state
            }
            break;
        case FLAG_RCV: // State Flag RCV
            if (buf == A_SET) {
                STATE = A_RCV; // Go to the next state
                superviseTrama[0] = buf;
            }
            else if (buf == FLAG) {
                STATE = FLAG_RCV; // Stays on the same state
            }
            else {
                STATE = START; // other character received goes to the initial state
            }
            break;
        case A_RCV: // State A RCV
            if (buf == RR1) {
                STATE = C_RCV; // Go to the next state
                superviseTrama[1] = buf;
            }
            else if (buf == FLAG) {
                STATE = FLAG_RCV;
            }
            else {
                STATE = START;
            }
            break;
        case C_RCV: // State C RCV
            if (buf == (superviseTrama[0] ^ superviseTrama[1])) {
                STATE = BCC_OK; // Go to the next state
            }
            else if (buf == FLAG) {
                STATE = FLAG_RCV;
            }
            else {
                STATE = START;
            }
            break;
        case BCC_OK: // State BCC_OK
            if (buf == FLAG) {
                STATE = START; // Go to the final state
                switch (superviseTrama[1]) {
                    case RR0:
                        return 1;
                    case RR1:
```

```
                    return 2;
                case REJ0:
                    return 3;
                case REJ1:
                    return 4;
            }
        }
        else {
            STATE = START;
        }
        break;
    default:
        printf("default\n");
        break;
  }

  return 0;
}
```

state_constants.h

```
#pragma once

#define FALSE 0
#define TRUE 1

#define TRANSMITTER 0
#define RECEIVER 1

#define MSG_FLAG 0x7E
#define OCT_ESCAPE 0x7d

// A - Campo de Endereço
#define MSG_A_TRANSMITTER_CMD 0x03
#define MSG_A_RECEIVER_RESP 0x03
#define MSG_A_RECEIVER_CMD 0x01
#define MSG_A_TRANSMITTER_RESP 0x01

// C - Campo de Controlo
#define MSG_C_SET 0x03 // Set up
#define MSG_C_DISC 0x0b // Disconnect
#define MSG_UA 0x07 // Unnumbered acknolegment
#define MSG_RR(n) ((n == 0) ? 0x05 : 0x85) // Receiver ready / Positive ACK :
                                           //   - if (n == 0) then (message is going to be sent on the Transmitter)
                                           //   - if (n == 1) then (message is going to be sent on the Receiver)
#define MSG_REJ(n) ((n == 0) ? 0x01 : 0x81) // Rejected / Negative ACK :
                                            //   - if (n == 0) then (message is going to be sent on the Transmitter)
                                            //   - if (n == 1) then (message is going to be sent on the Receiver)
#define MSG_BCC(n) ((n == 0) ? 0x01 : 0x81) // Campo de Proteção (cabeçalho)
```

Anexo II – Dados

O gráfico que se segue apresenta os resultados obtidos nas estatísticas efetuadas em ficheiros de diferentes dimensões.

Table 1 Tabela de dados

Nº of total frames sent	Nº of total trames retransmitted	Nº of bytes	Timeouts	Duration of program (s)	Frames size (bytes)	Bit rate recebido (*)
110	0	10968	0	0.437376	100	25076.82
160	0	15946	0	0.649569	100	24548.585
2904	0	290358	0	11.671324	100	24877.949
26422	0	2612188	0	104.919846	100	24896.998

(*) Nº of bytes / Duration of program

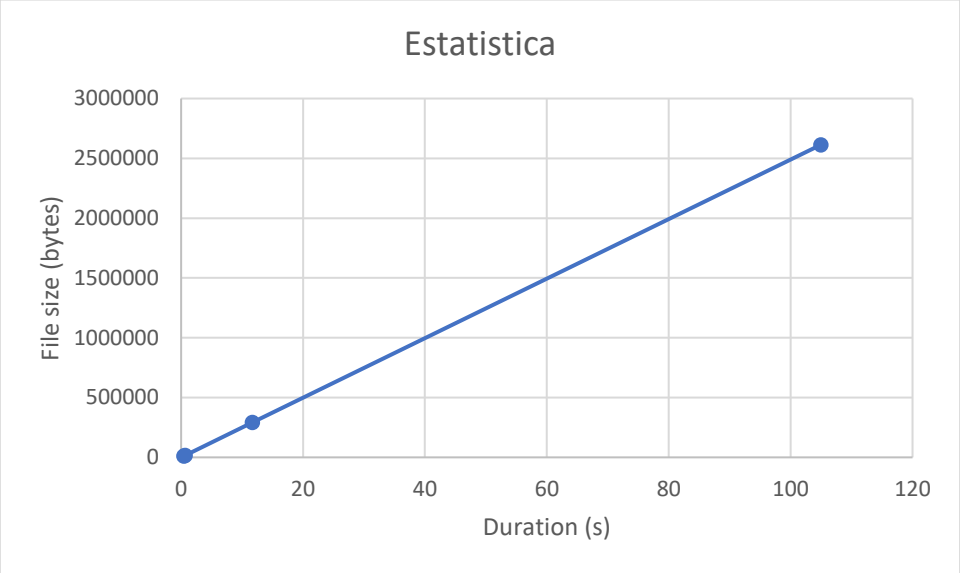

Figure 4 Gráfico de estatísticas