U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Protocolo de ligação de dados

Redes de Computadores

Mestrado Integrado de Engenharia Informática e Computação

10 de novembro de 2020

Daniel Garcia Silva, up201806524@fe.up.pt

Mariana Truta, up201806543@fe.up.pt

3ºano, Turma 3, Grupo 5

Índice

Sumário

No âmbito da unidade curricular de *Redes de Computadores*, foi elaborado um projeto que consistia no desenvolvimento de um *software* que permitisse a **transferência** de ficheiros de um computador para o outro, estando estes ligados por um **cabo série**.

Ao longo deste relatório, será explicado como foram **cumpridos** todos os objetivos do projeto, tendo sido concluída uma aplicação **funcional** e sem **perdas de dados**.

Introdução

Este primeiro projeto tinha dois grandes objetivos: implementar um **protocolo de ligação de dados**, especificado no guião fornecido pelos docentes, e testá-lo com uma **aplicação** simples de transferência de ficheiros. Relativamente ao ambiente de desenvolvimento, o trabalho foi realizado em LINUX, utilizando a linguagem de programação C e portas série RS-232 cuja comunicação é assíncrona.

Neste relatório, pretende-se tornar claro como foi possível a elaboração de um **serviço de comunicação fiável** entre dois computadores por via de uma porta de série assíncrona, apresentando detalhes de toda a **teoria** utilizada. Este relatório está estruturado da seguinte forma:

- **Arquitetura e estrutura de código:** descrição dos blocos funcionais e interfaces implementadas e apresentação das *APIs*, principais estruturas de dados e funções;
- **Casos de uso principais:** identificação dos casos de uso e sequências de chamadas de funções;
- **Protocolo de ligação de dados:** identificação dos principais aspetos funcionais bem como a descrição da estratégia de implementação dos mesmos, sendo complementada com a apresentação de extratos de código;
- **Protocolo de aplicação:** identificação dos principais aspetos funcionais bem como a descrição da estratégia de implementação dos mesmos, sendo complementada com a apresentação de extratos de código;
- **Validação:** descrição dos testes efetuados com apresentação quantificada dos resultados;
- **Eficiência de protocolo de ligação de dados:** caraterização estatística da eficiência do protocolo, recorrendo a medidas sobre o código desenvolvido;
- **Conclusão:** síntese da informação apresentada nas secções anteriores; reflexão sobre os objetivos de aprendizagem alcançados.

Arquitetura e Estrutura do Código

O projeto está dividido em duas camadas muito bem definidas: a camada de **protocolo de ligação de dados** e a camada da **aplicação**.

O objetivo do **protocolo de ligação de dados** é fornecer um serviço de comunicação de dados **fiável** entre dois sistemas ligados por um **cabo série**. Esta camada contem, assim, todas as funções necessárias para a **abertura**, **fecho**, **escrita** e **leitura** da porta de série. Para além disso, tem também a seu encargo o controlo de **erros** e **fluxo** e o *stuffing*/*destuffing* de pacotes. Todas estas funcionalidades são asseguradas pelas seguintes funções:

protocol.c

- ***alarmSenderHandler*/*alarmReceiverHandler*** – são chamadas quando o tempo do alarme do emissor ou do recetor, respetivamente, se esgota;
- ***SandWOpenClose*** - envia uma mensagem com o comando *send* e o endereço *sendAddress* e espera por uma resposta com o comando *receive* e o endereço *recAddress*, usando um mecanismo de *Stop&Wait;*
- ***sendOpenCloseFrame*** – cria e envia uma trama de acordo com o comando (*SET, DISC* ou *UA*) e o endereço passados como argumentos;
- ***sendAckFrame*** - cria e envia uma trama de rejeição ou de aceitação;
- ***receiveOpenCloseFrame*** - lê a mensagem recebida (*SET*, *DISC* ou *UA*) e chama a função *changeStateS* que irá interpretar o byte e mudar de estado de acordo com o mesmo;
- ***receiveAckFrame*** - lê uma trama ACK, chama a função que verifica se é válida ou não, *changeStateAck*, e retorna 1, caso tenha sido rejeitada, ou 0, se for válida;
- ***sendInfoFrame*** - cria e envia uma trama *I* com os dados em *info*, acrescentando a quantidade necessária de *0x00* até ser atingido o valor *IFRAME_SIZE – 2* e garantindo assim que a *frame* tem sempre um tamanho fixo;
- ***receiveInfoFrame*** - é responsável por ler uma trama *I*, guardando o pacote de controlo ou de dados em *info*, após o *byte destuffing.* O seu valor de retorno depende se a trama foi ignorada, rejeitada ou aceite pela máquina de estados, *changeStateInfo,* ou se o tempo do alarme esgotou.

stateMachines.c

- ***changeStateS*** - verifica se o byte recebido é o esperado e age de acordo com o mesmo, atualizando o estado atual da trama *SET*, *DISC* ou *UA*;

- ***changeStateInfo*** - verifica se o byte recebido é o esperado e age de acordo com o mesmo, atualizando o estado atual da trama *I*;
- ***changeStateAck*** - verifica se o byte recebido é o esperado e age de acordo com o mesmo, atualizando o estado atual da trama *REJ ou RR*.

Em relação à camada da **aplicação**, esta está situada acima da camada de ligação de dados, sendo responsável pelo **envio** e **receção** de ficheiros. Tem de permitir enviar e identificar os **pacotes de controlo** bem como **dividir** o ficheiro em vários pacotes, no caso do emissor, ou **concatenar** e **interpretar** toda a informação recebida, no caso do recetor. Nesta camada, as funções principais são:

appSender.c

- ***main*** - é responsável pela escrita de um ficheiro, sendo o *path* fornecido pelo utilizador.
- ***makeControlPacket*** – cria um pacote de controlo com o campo de controlo *START* ou *END* e com o tamanho e nome do ficheiro passados como argumentos;
- ***makeDataPacket*** - cria um pacote de dados com o número de sequência *N* e os dados *info*.

appReceiver.c

- ***main*** – é responsável pela leitura de um ficheiro, sendo guardado em *"./images/ToReceive"*;
- ***parseInfo*** - encaminha o pacote de dados para função respetiva, tendo em conta o seu primeiro byte (*START_BYTE*, *DATA_BYTE* e *END_BYTE*);
- ***parseControlPacket*** - interpreta a informação guardada na *info*, armazenando o nome e o tamanho do ficheiro que irá ser recebido na *struct File*;
- ***checkControlPacket*** - verifica se o pacote de controlo de finalização contém a mesma informação que o pacote de controlo que sinalizou o inicio da transmissão, diferenciando apenas no campo de controlo;
- ***parseDataPacket*** - guarda os dados do ficheiro no campo *data* da *struct File*.

datalink.c

Neste ficheiro, está contida a **interface Protocolo-Aplicação** que se resume essencialmente a quatro funções que serão descritas pormenorizadamente mais à frente: *llopen*, *llwrite*, *llread* e *llclose*. Estas funções facilitam a **comunicação** entre a camada da aplicação e a camada de ligação de dados, permitindo assim que a

arquitetura cumpra o **principio de independência entre camadas**, isto é, nenhuma camada conhece os detalhes da outra.

Para facilitar o armazenamento de informação, foram utilizadas algumas **estruturas de dados definidas** em *dataStructures*.h: *enum State* e *enum AckState* que permitem guardar o estado atual da respetiva trama, *enum Command* que possibilita definir o tipo de comando e a *struct File* que armazena a informação necessária para a escrita do novo ficheiro. Para além disso, em

macros.h, encontram-se todas as **macros** necessárias ao longo do projeto.

Casos de uso principais

Inicialmente, é necessário **compilar** o programa, executando o comando *make* na pasta *src.* De seguida, é necessário executar *"./sender <porta> <path do ficheiro a enviar>"*, no caso do **emissor**, e *"./receiver <porta>",* no caso do **recetor**. A porta de série tem de estar no formato *"/dev/ttySX"* sendo X o número da mesma.

A **transmissão**/**receção** dos dados é feita pela seguinte ordem:

no caso do **emissor**,

1. **Estabelecimento** da ligação entre o emissor e recetor na função *llopen*;
2. **Criação** do pacote de controlo *START* e o seu **envio** com *llwrite*;
3. **Criação** de pacotes de dados com os dados do ficheiro e o seu **envio** em *llwrite*;
4. **Criação** do pacote de controlo *END* e o seu **envio** em *llwrite*;
5. **Terminação** da ligação entre o emissor e o recetor na função *llclose.*

no caso do **recetor**,

1. **Estabelecimento** da ligação entre o emissor e recetor na função *llopen*;
2. **Receção** e **análise** do pacote de controlo *START*, enviando uma **resposta**, com auxilio da função *llread*;
3. **Receção** e **análise** dos pacotes de dados, enviando uma **resposta**, com auxilio da função *llread*;
4. **Receção** e **análise** do pacote de controlo *END*, com auxilio da função *llread*;
5. **Escrita** dos dados recebidos num ficheiro com o mesmo nome do ficheiro enviado pelo emissor*;*
6. **Terminação** da ligação entre o emissor e o recetor na função *llclose.*

Protocolo de Ligação Lógica

O **protocolo de ligação lógica** é responsável por várias funcionalidades. Como a *API* da porta de série tem ao seu encargo a **receção** dos dados da camada da aplicação e **estabelecimento** da ligação entre esta e a camada de ligação de dados, é conveniente explicar detalhadamente como é que as funções *llopen*, *llwrite*, *llread* e *llclose* se

relacionam com o **protocolo da ligação lógica**. Estas utilizam o mecanismo de ***Stop&Wait***, isto é, envia-se um pacote de cada vez, esperando por uma resposta até se poder enviar outro. Se esta resposta não chegar, demorar a chegar (máximo 3 segundos) ou for uma resposta de erro, é iniciada uma nova **retransmissão** da mensagem enviada até ao limite de 3 retransmissões.

int llopen(int port, int status)

Estabelece a ligação entre o emissor e o recetor e **abre** a porta de série.

É responsável por **enviar** uma mensagem *SET* e **esperar** por uma mensagem *UA,* no caso do **emissor**. Esta ação é realizada pela função *SandWOpenClose* da camada de ligação de dados. Por sua vez, no caso do **recetor**, é chamada a função *receiveOpenCloseFrame* que irá **receber** a mensagem *SET* e posteriormente **envia** uma mensagem *UA* com o auxilio da função *sendOpenCloseFrame*.

*int llwrite(int fd, char *buffer, int length)*

Esta função **recebe** um buffer que será enviado para a função *sendInfoFrame*, onde será **criada** uma **trama de informação** com este *buffer*, após o *byte stuffing* do mesmo, e **enviada** para o recetor. De seguida, **espera** por uma mensagem *ACK*. Se o pacote tiver sido **rejeitado**, é **retransmitido**. Se tiver sido **aceite**, a transmissão do ficheiro **continuará**.

*int llread(int fd, char *buffer)*

Esta função chama *receiveInfoFrame* que irá **esperar** por uma trama I, analisando o conteúdo do seu cabeçalho à procura de qualquer erro. Posteriormente, é **enviado** um comando *REJ* **/** *RR* na função *sendAckFrame* caso a trama tenha sido **rejeitada** ou **aceite**, respetivamente. A trama só é rejeitada caso não seja um duplicado e tenha sido detetado um erro no campo de dados pelo respetivo *BCC*.

int llclose(int fd, int status)

Termina a ligação entre o emissor e o recetor.

O emissor chama a função *SandWOpenClose* que **envia** o comando *DISC* e **espera** pela receção do comando *DISC* enviado pelo recetor. Termina com o **envio** do comando *UA* com o auxílio da função *sendOpenCloseFrame*. Por outro lado, o recetor, após **receber** o comando *DISC*, **envia** o comando *DISC* para o emissor na função *receiveOpenCloseFrame* e **aguarda** pela receção do comando *UA* com o auxilio da função *SandWOpenClose.* Em ambos os casos, é **fechada** a porta de série.

Protocolo da Aplicação

O protocolo da aplicação tem a seu encargo os seguintes aspetos funcionais:

- **Envio de pacotes de controlo que sinalizam o inicio e o fim da transmissão pelo emissor**

  Na *main* do ficheiro *appSender*, é chamada a função *makeControlPacket* que **constrói** um pacote de controlo com o tamanho e o nome do ficheiro a enviar e é posteriormente enviado com o auxilio da função *llwrite*.
- **Envio de pacotes de dados contendo fragmentos dos dados do ficheiro a enviar pelo emissor**

  Na *main* do ficheiro *appSender*, são lidos **fragmentos** com tamanho *MAX_K* do ficheiro até se chegar ao fim do mesmo. A cada fragmento é acrescentado um cabeçalho em *makeDataPacket* com o seu **tamanho** e o número de **sequencia** *sequenceN*, sendo posteriormente **enviado** com o auxilio de *llwrite*.
- **Leitura de pacotes de controlo e de dados pelo recetor**

  Na *main* do ficheiro *appReceiver*, quando se **recebe** um pacote, é analisado o seu primeiro byte na função *parseInfo* e a informação recebida é passada à função respetiva de acordo com esse mesmo byte: no caso de ser o *START_BYTE*, é chamada a função *parseControlPacket* que é responsável por preencher os campos do **tamanho** e **nome** do ficheiro na *struct File* bem como guardar todo o pacote recebido no campo *controlPacket* para futura comparação com o pacote de terminação; no caso de ser o *DATA_BYTE*, o pacote segue para a função *parseDataPacket*, onde serão guardados os **dados** no campo *data* da *struct File*; no caso de ser o END_BYTE, é chamada a função *checkControlPacket* tem a seu cargo a **comparação** deste pacote com o pacote que sinalizou o inicio da transmissão. O número de sequência do pacote de dados é armazenado na variável estática *N*.
- **Criação do ficheiro recebido no recetor**

  Na *main* do ficheiro *appReceiver*, após a receção de todos os pacotes, é realizada a **criação** e **escrita** do ficheiro recebido com o tamanho e nome enviados, sendo este guardado na pasta *"./imagesToReceive/"*.

Validação

O programa foi capaz de transmitir uma imagem de 10 968 bytes e outra de 3 309 702 bytes, com e sem interrupção da porta de série, variando o BAUDRATE (entre 2400 e 38400) e o tamanho da trama I (entre 128 e 8192 bytes).

Eficiência

Usando uma trama de tamanho 512 bytes, o ficheiro pinguim.gif precisou de 47 packets para ser transferido, ou seja, 192 512 bits foram transferidos. Variando o BAUDRATE, e com uma FER de 5,91% (a probabilidade de erro no cabeçalho e no campo de dados é de 3% em ambos os casos), obtiveram-se os seguintes valores:

BAUDRATE	Tempo(s)	R(Bits/s)	S REAL (R/C)
2400	104,103	1849,246	0,770519163
4800	60,85397	3163,508	0,659064095
9600	35,89102	5363,793	0,558728432
19200	15,64255	12306,94	0,64098671
38400	6,72748	28615,77	0,745202265
57600	13,39685	14369,95	0,249478215
115200	20,38437	9444,099	0,081980022

Como se pode ver pelo gráfico do Anexo 2, para um tamanho de trama I de 512 bytes, a maior eficiência é observada com o BAUDRATE de 2400 bits/s, decrescendo a partir daí até ao BAUDRATE de 9600, para voltar crescer até a sensivelmente o mesmo valor (cerca de 75%) com um BAUDRATE de 38400, para voltar a decrescer, agora mais acentuadamente, com o aumento do mesmo.

Conclusão

A implementação deste programa permitiu uma melhor aprendizagem dos conceitos que aplicámos, nomeadamente:

- O protocolo de ligação de dados, neste caso, *Stop&Wait*, bem como a noção de transparência, através do *byte stuffing*;
- A independência entre camadas, de ligação e de aplicação, e a sua interface, através da separação de processamento de pacotes e tramas.

A equipa gostaria ainda de referir que este trabalho prático apresentou algumas dificuldades, causadas principalmente pela dificuldade em testar o programa nos laboratórios.

Anexo 1 – Código Fonte

appReceiver.c

```
#include "appReceiver.h"

static File *file;
static int N = 0;

int main(int argc, char **argv) {

    if ((argc != 2) || ((strcmp("/dev/ttyS10", argv[1]) != 0) && (strcmp("/dev
/ttyS11", argv[1]) != 0) && (strcmp("/dev/ttyS0", argv[1]) != 0) && (strcmp("/
dev/ttyS1", argv[1]) != 0)))
    {
        printf("Usage:\t./sender <SerialPort>\n");
        exit(1);
    }

    int port;
    if (strcmp("/dev/ttyS10", argv[1]) == 0)
        port = COM10;
    if (strcmp("/dev/ttyS11", argv[1]) == 0)
        port = COM11;
    if (strcmp("/dev/ttyS0", argv[1]) == 0)
        port = COM0;
    if (strcmp("/dev/ttyS1", argv[1]) == 0)
        port = COM1;

    int fd = llopen(port, RECEIVER);
    if (fd < 0)
    {
        printf("llopen failed\n");
        exit(1);
    }
    else
        printf("\nConnection established with success!\n");

    printConnectionInfo();

    int size;
    int packets = 0;
    int finished = FALSE;

    if (initFile() < 0){
        printf("Could not allocate memory for file!\n");
        if (closePort(fd, RECEIVER) < 0) printf("closePort failed\n");
        return -1;
    }


    time_t t;
    srand((unsigned) time(&t));

    struct timeval beginTime, endTime;
    gettimeofday(&beginTime, NULL);

    printf("\nReceiving...\n");

    while (!finished)
    {
        unsigned char *info = (unsigned char *)malloc(MAX_PACKET_SIZE);
```

```
        if(info == NULL){
            free(file->name);
            free(file);
            printf("Could not allocate memory for info!\n");
            if (closePort(fd, RECEIVER) < 0) printf("closePort failed\n");
            return -1;
        }

        size = llread(fd, info);

        if (size < 0) {
            free(info);
            continue;
        } else if(size == 0) {
            printf("Timeout, closing.\n");
            free(info);
            freeFile();
            if (closePort(fd, RECEIVER) < 0) printf("closePort failed\n");
            return -1;
        }

        int result = parseInfo(info, size);

        if (result == 1)
            finished = TRUE;
        if (result == -1)
            break;

        packets++;

        free(info);
    }

    double elapsed = 0;

    if (!finished){
        printf("Data reception interrupted!\n");
        if (closePort(fd, RECEIVER) < 0) printf("closePort failed\n");
        return -1;
    }
    else
    {
        gettimeofday(&endTime, NULL);

        elapsed = (endTime.tv_sec - beginTime.tv_sec) * 1e6;
        elapsed = (elapsed + (endTime.tv_usec - beginTime.tv_usec)) * 1e-6;

        unsigned char* filename = (unsigned char*)malloc(MAX_VALUE_SIZE);
        sprintf(filename, "./imagesToReceive%s", file->name);

        int fileDescriptor = open(filename, O_RDWR | O_CREAT, 0777);
        if (fileDescriptor < 0)
        {
            free(filename);
            freeFile();
            printf("Error opening file!\n");
            if (closePort(fd, RECEIVER) < 0) printf("closePort failed\n");
            return -1;
        }

        if (write(fileDescriptor, file->data, file->size) < 0)
        {
            free(filename);
            freeFile();
            printf("Error writing to file!\n");
            if (closePort(fd, RECEIVER) < 0) printf("closePort failed\n");
            return -1;
```

```
        }

        if (close(fileDescriptor) < 0)
        {
            free(filename);
            freeFile();
            printf("Error closing file!\n");
            if (closePort(fd, RECEIVER) < 0) printf("closePort failed\n");
            return -1;
        }

        free(filename);
    }

    printFileInformation(file->size, file->name);

    freeFile();

    if (llclose(fd, RECEIVER) < 0)
    {
        printf("llclose failed\n");
        exit(1);
    }

    printf("\nElapsed: %.5lf seconds\n", elapsed);

    printf("Received %d packets.\n", packets);

    return 0;
}

void printConnectionInfo(){
    printf("\n======= Connection Information ======= \n");
    printf("I Frame size: %d\n", IFRAME_SIZE);
    printf("Retries: %d\n", ATTEMPTS);
    printf("Timeout: %d\n", 3);
}

void printFileInformation(long int filesize, unsigned char* filename){

    printf("\n======= File Information ======= \n");
    printf("Name: %s\n", filename);
    printf("Size: %ld\n\n", filesize);

    fflush(stdout);
}

int initFile(){

    file = malloc(sizeof(File));

    if(file == NULL) return -1;

    file->size = 0;
    file->lastIndex = 0;

    file->name = (unsigned char*)malloc(MAX_VALUE_SIZE);
    if(file->name == NULL){
        free(file);
        return -1;
    }

    file->controlPacket = (unsigned char*)malloc(MAX_PACKET_SIZE);
    if(file->controlPacket == NULL){
        free(file->name);
        free(file->data);
        free(file);
        return -1;
```

```
    }

    return 0;
}

void freeFile(){
    free(file->data);
    free(file->name);
    free(file->controlPacket);
    free(file);
}

int parseInfo(unsigned char *info, int size) {

    unsigned char byte = info[0];

    switch (byte)
    {
    case START_BYTE:
        if (parseControlPacket(info, size) < 0) return -1;
        break;

    case DATA_BYTE:
        if (parseDataPacket(info, size) < 0){
            freeFile();
            return -1;
        }
        N++;
        N %= 255;
        break;

    case END_BYTE:
        if (checkControlPacket(info, size) < 0)
        {
            freeFile();
            printf("End Control Packet is not correct!\n");
            return -1;
        }
        return 1;
        break;

    default:
        break;
    }

    return 0;
}

int parseControlPacket(unsigned char *info, int size) {
    int i = 1;
    int index = 0;
    int l, j;

    while(i < size)
    {
        if (info[i] == FILESIZE) {
            file->controlPacket[index++] = info[i++];

            file->controlPacket[index++] = info[i];
            l = info[i++];
            j = 0;

            unsigned char* sizeString = (unsigned char*)malloc(l+1);
            if(sizeString == NULL){
                printf("Could not allocate memory for size!\n");
                free(file->name);
                free(file);
                return -1;
```

```
            }

            while (j != l)
            {
                file->controlPacket[index++] = info[i];
                sizeString[j] = info[i++];
                j++;
            }
            sizeString[j] = 0;
            file->size = atoi(sizeString);
            free(sizeString);

        } else if (info[i] == FILENAME) {
            file->controlPacket[index++] = info[i++];

            file->controlPacket[index++] = info[i];
            l = info[i++];
            j = 0;

            while (j != l)
            {
                file->controlPacket[index++] = info[i];
                file->name[j++] = info[i++];
            }
            file->name[j] = 0;
            break;
        }
    }

    long int dataSize = file->size;
    file->data = (unsigned char*)malloc(file->size);
    if(file->data == NULL){
        printf("Could not allocate memory for file->data!\n");
        free(file->name);
        free(file);
        return -1;
    }

    return 0;
}

int parseDataPacket(unsigned char *info, int size)
{
    int index = 1;
    unsigned char byte = info[index++];

    if (byte != N)
    {
        printf("Sequence number of data packet is not correct!\n");
        return -1;
    }

    int L2 = info[index++];
    int L1 = info[index++];

    int K = 256 * L2 + L1;

    for (int i = 0; i < K; i++)
    {
        file->data[file->lastIndex++] = info[index++];
    }
    return 0;
}

int checkControlPacket(unsigned char *info, int size) {
    int i = 1;
    int index = 0;
    int l, j, k = 0;
```

```
    unsigned char* temp = (unsigned char*)malloc(MAX_PACKET_SIZE);
    if (temp == NULL) {
        printf("Could not allocate memory for temp!\n");
        return -1;
    }

    while(i < size)
    {
        if (info[i] == FILESIZE)
        {
            temp[k++] = info[i];
            i++;

            temp[k++] = info[i];
            l = info[i++];
            j = 0;

            while (j != l)
            {
                temp[k++] = info[i++];

                j++;
            }
        }
        else if (info[i] == FILENAME)
        {
            temp[k++] = info[i];
            i++;

            temp[k++] = info[i];
            l = info[i++];
            j = 0;

            while (j != l)
            {
                temp[k++] = info[i++];
                j++;
            }
            break;
        }
    }

    while (index < k) {
        if (file->controlPacket[index] != temp[index++]) {
            free(temp);
            return -1;
        }
    }

    free(temp);
    return 0;
}
```

appReceiver.h

```
#ifndef APPRECEIVER_H
#define APPRECEIVER_H

#pragma once

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <fcntl.h>
#include <termios.h>
```

```c
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "dataLink.h"
#include "macros.h"
#include "dataStructures.h"

/**
 * @brief shows the connection information on the screen
 *
 */
void printConnectionInfo();

/**
 * @brief shows the file information on the screen
 *
 */
void printFileInformation(long int filesize, unsigned char* filename);

/**
 * @brief Initializes and allocates memory for the struct File and its members
 *
 * @return 0 on success; -1 on error
*/
int initFile();

/**
 * @brief Free the memory previously allocated to the struct File
*/
void freeFile();

/**
 * @brief Fowards info and its size to the respective function according to in
fo's first byte
 *
 * @param info the packet to parse
 * @param size the size of the packet to parse
 *
 * @return 1 if it's an end packet; 0 if it's a start or data packet; -
1 on error
*/
int parseInfo(unsigned char *info, int size);

/**
 * @brief Parses control packet (START), storing the file's information in the
 struct File
 *
 * @param info the packet to parse
 * @param size the size of the packet to parse
 *
 * @return 0 on success; -1 on error
*/
int parseControlPacket(unsigned char *info, int size);

/**
 * @brief Parses data packet, storing the file's information in the struct Fil
e
 *
 * @param info the packet to parse
 * @param size the size of the packet to parse
 *
 * @return 0 on success; -1 on error
*/
int parseDataPacket(unsigned char *info, int size);

/**
 * @brief Parses control packet (END) and compares it with the initial packet
stored in controlPacket of the struct File
```

```
 *
 * @param info the packet to parse
 * @param size the size of the packet to parse
 *
 * @return 0 on success; -1 on error
*/
int checkControlPacket(unsigned char *info, int size);

#endif /*APPRECEIVER_H*/
```

appSender.c

```
 #include "appSender.h"

int main(int argc, char **argv) {

    if ((argc != 3) || ((strcmp("/dev/ttyS10", argv[1]) != 0) && (strcmp("/dev
/ttyS11", argv[1]) != 0) && (strcmp("/dev/ttyS0", argv[1]) != 0) && (strcmp("/
dev/ttyS1", argv[1]) != 0)))
    {
        printf("Usage:\t./sender <SerialPort> <path>\n");
        exit(1);
    }

    int port;
    if (strcmp("/dev/ttyS10", argv[1]) == 0)
        port = COM10;
    if (strcmp("/dev/ttyS11", argv[1]) == 0)
        port = COM11;
    if (strcmp("/dev/ttyS0", argv[1]) == 0)
        port = COM0;
    if (strcmp("/dev/ttyS1", argv[1]) == 0)
        port = COM1;

    int fd = llopen(port, SENDER);
    if (fd < 0)
    {
        printf("llopen failed\n");
        exit(1);
    }
    else
    {
        printf("\nConnection established with success!\n\n");
    }

    unsigned char *path = argv[2];

    struct stat fileInfo;
    if(stat(path, &fileInfo) < 0){
        printf("stat failed!\n");
        if (closePort(fd, SENDER) < 0) printf("closePort failed\n");
        return -1;
    }

    struct timeval beginTime, endTime;
    gettimeofday(&beginTime, NULL);

    unsigned char *StartPacket = (unsigned char *)malloc(MAX_PACKET_SIZE);
    if (StartPacket == NULL)
    {
        printf("Could not allocate memory for StartPacket!\n");
        if (closePort(fd, SENDER) < 0) printf("closePort failed\n");
        return -1;
    }

    int packet_size;
```

```
    unsigned char *filename = strrchr(path, '/');
    packet_size = makeControlPacket(START_BYTE, fileInfo.st_size, filename, St
artPacket);

    if (llwrite(fd, StartPacket, packet_size) == -1)
    {
        free(StartPacket);
        printf("Could not send Start Packet!\n");
        if (closePort(fd, SENDER) < 0) printf("closePort failed\n");
        return -1;
    }

    printInformation(fileInfo.st_size, filename);

    int fdFile = open(path, O_RDONLY);
    if (fdFile == -1)
    {
        free(StartPacket);
        printf("Could not open file!\n");
        if (closePort(fd, SENDER) < 0) printf("closePort failed\n");
        return -1;
    }

    int charactersRead;
    long int sequenceN = 0;

    unsigned char *fileBuffer = (unsigned char *)malloc(MAX_K);
    if (fileBuffer == NULL)
    {
        free(StartPacket);
        printf("Could not allocate memory for fileBuffer!\n");
        if (closePort(fd, SENDER) < 0) printf("closePort failed\n");
        return -1;
    }

    unsigned char *dataPacket = (unsigned char *)malloc(MAX_PACKET_SIZE);
    if (dataPacket == NULL)
    {
        free(StartPacket);
        free(fileBuffer);
        printf("Could not allocate memory for dataPacket!\n");
        if (closePort(fd, SENDER) < 0) printf("closePort failed\n");
        return -1;
    }

    int size = 0;
    int packets = 2;

    printf("Sending...\n");

    while (charactersRead = read(fdFile, fileBuffer, MAX_K))
    {
        packet_size = makeDataPacket(fileBuffer, sequenceN, dataPacket, charac
tersRead);

        if (llwrite(fd, dataPacket, packet_size) == -1)
        {
            free(StartPacket);
            free(fileBuffer);
            free(dataPacket);
            printf("Could not send Data Packet number %ld\n", sequenceN);
            if (closePort(fd, SENDER) < 0) printf("closePort failed\n");
            return -1;
        }

        sequenceN++;
        sequenceN %= 255;
        size += charactersRead;
```

```
        packets++;
    }

    close(fdFile);

    unsigned char *EndPacket = (unsigned char *)malloc(MAX_PACKET_SIZE);
    if (EndPacket == NULL)
    {
        free(StartPacket);
        free(fileBuffer);
        free(dataPacket);
        printf("Could not allocate memory for EndPacket!\n");
        if (closePort(fd, SENDER) < 0) printf("closePort failed\n");
        return -1;
    }

    packet_size = makeControlPacket(END_BYTE, fileInfo.st_size, filename, EndP
acket);

    if (llwrite(fd, EndPacket, packet_size) == -1)
    {
        free(StartPacket);
        free(fileBuffer);
        free(dataPacket);
        free(EndPacket);
        printf("Could not send End Packet!\n");
        if (closePort(fd, SENDER) < 0) printf("closePort failed\n");
        return -1;
    }

    gettimeofday(&endTime, NULL);

    double elapsed = (endTime.tv_sec - beginTime.tv_sec) * 1e6;
    elapsed = (elapsed + (endTime.tv_usec - beginTime.tv_usec)) * 1e-6;

    free(StartPacket);
    free(fileBuffer);
    free(dataPacket);
    free(EndPacket);

    if (llclose(fd, SENDER) < 0)
    {
        printf("llclose failed\n");
        exit(1);
    }

    printf("\nElapsed: %.5lf seconds\n", elapsed);
    printf("Send %d packets.\n", packets);

    return 0;
}

void printInformation(long int filesize, unsigned char* filename){

    printf("======= Connection Information ======= \n");
    printf("I Frame size: %d\n", IFRAME_SIZE);
    printf("Retries: %d\n", ATTEMPTS);
    printf("Timeout: %d\n", 3);

    printf("======= File Information ======= \n");
    printf("Name: %s\n", filename);
    printf("Size: %ld\n\n", filesize);

    fflush(stdout);
}

int makeControlPacket(unsigned char control, long int fileSize, unsigned char
*fileName, unsigned char *packet)
```

```
{
    int index = 0;

    packet[index++] = control;

    packet[index++] = FILESIZE;

    unsigned char *n = (unsigned char *)malloc(MAX_VALUE_SIZE);
    sprintf(n, "%ld", fileSize);

    packet[index++] = strlen(n);

    for (int i = 0; i < packet[2]; i++)
    {
        packet[index++] = n[i];
    }

    free(n);

    packet[index++] = FILENAME;
    packet[index++] = strlen(fileName);

    for (int j = 0; j < strlen(fileName); j++)
    {
        packet[index++] = fileName[j];
    }

    return index;
}

int makeDataPacket(unsigned char *info, int N, unsigned char *packet, int leng
th)
{
    int index = 0;

    packet[index++] = DATA_BYTE;
    packet[index++] = N;

    packet[index++] = length / 256;
    packet[index++] = length % 256;

    for (int i = 0; i < length; i++)
    {
        packet[index++] = info[i];
    }

    return index;
}
```

appSender.h

```
  #ifndef APPSENDER_H
#define APPSENDER_H

#pragma once

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "dataLink.h"

void printInformation(long int filesize, unsigned char* filename);
```

```
/**
 * @brief Makes a control packet with all the information passed as arguments
and stores it in packet
 *
 * @param control START_BYTE/DATA_BYTE/END_BYTE
 * @param fileSize the size of the file
 * @param fileName the name of the file
 * @param packet the control packet
 *
 * @return the number of bytes written in the packet
 *
*/
int makeControlPacket(unsigned char control, long int fileSize, unsigned char
*fileName, unsigned char *packet);

/**
 * @brief Makes a data packet with all the information passed as arguments and
 stores it in packet
 *
 * @param info the info of the file
 * @param N the sequence number of the packet
 * @param length the number of bytes in info
 * @param packet the control packet
 *
 * @return the number of bytes written in the packet
 *
*/
int makeDataPacket(unsigned char *info, int N, unsigned char *packet, int leng
th);

#endif /*APPSENDER_H*/
```

datalink.c

```
 #include "dataLink.h"

static int senderNS = 0;
static int receiverNS = 0;

int llopen(int port, int status)
{
    struct sigaction newAction, oldAction;

    newAction.sa_handler = alarmSenderHandler;
    sigemptyset(&newAction.sa_mask);
    newAction.sa_flags = 0;

    sigaction(SIGALRM, &newAction, &oldAction);

    int fd = initPort(port, 0, 0, status);

    int res;

    if (status == SENDER)
    {
        res = SandWOpenClose(fd, SET, SEND_REC, UA, SEND_REC);
        if (res == 0)
            return fd;
        else if (res == -1)
        {
            tcflush(fd, TCIFLUSH);
            printf("Could not receive UA Frame!\n");
            return -1;
        }
        tcflush(fd, TCOFLUSH);
    }
```

```
    else if (status == RECEIVER)
    {
        int recSet = receiveOpenCloseFrame(fd, SET, SEND_REC);

        if (recSet == 0)
        {
            res = sendOpenCloseFrame(fd, UA, SEND_REC);
            if (res == 0)
            {
                return fd;
            }
            else
            {
                printf("Could not send UA Frame!\n");
            }
        }
        else if (recSet == -1)
        {
            printf("Could not read from port!\n");
        }
    }
    return -1;
}

int llwrite(int fd, char *buffer, int length)
{
    struct sigaction newAction, oldAction;

    newAction.sa_handler = alarmSenderHandler;
    sigemptyset(&newAction.sa_mask);
    newAction.sa_flags = 0;

    sigaction(SIGALRM, &newAction, &oldAction);

    int bytesSent;
    int response;

    for (int i = 0; i < ATTEMPTS; i++)
    {
        bytesSent = sendInfoFrame(fd, senderNS, buffer, length);
        if (bytesSent == -1)
        {
            printf("Could not send I Frame! Attempt number %d\n", i + 1);
            return -1;
        }

        alarmSender = 1;
        alarm(3);

        while (alarmSender)
        {
            response = receiveAckFrame(fd, 1 - senderNS);
            if (response < 0)
            {
                printf("Could not read ACK Frame!\n");
            }
            else
            {
                alarm(0);
                break;
            }
        }

        if (alarmSender && (response == 0))
        {
            senderNS = 1 - senderNS;
            return bytesSent;
        }
```

```
    }

    return -1;
}

int llread(int fd, char *buffer)
{
    int receive = receiveInfoFrame(fd, buffer, receiverNS);

    if (receive == 1) {
        sendAckFrame(fd, REJ, receiverNS);
        return -1;
    }
    else if (receive == 0)
    {
        receiverNS = 1 - receiverNS;
        sendAckFrame(fd, RR, receiverNS);
        return IFRAME_SIZE;
    } else if (receive == -3) {
        sendAckFrame(fd, RR, receiverNS);
        return -1;
    } else if (receive == -2) {
        return 0;
    } else return -1;

}

int llclose(int fd, int status)
{
    struct sigaction newAction, oldAction;

    newAction.sa_handler = alarmSenderHandler;
    sigemptyset(&newAction.sa_mask);
    newAction.sa_flags = 0;

    sigaction(SIGALRM, &newAction, &oldAction);

    int res;

    if (status == SENDER)
    {
        res = SandWOpenClose(fd, DISC, SEND_REC, DISC, REC_SEND);
        if (res == 0)
        {
            res = sendOpenCloseFrame(fd, UA, REC_SEND);
            if (res != 0)
            {
                printf("Could not send UA Frame!\n");
                return -1;
            }
        }
        else if (res == -1)
        {
            printf("Could not receive DISC Frame!\n");
            return -1;
        }
    }
    else if (status == RECEIVER)
    {
        int recDISC = receiveOpenCloseFrame(fd, DISC, SEND_REC);

        if (recDISC == 0)
        {
            res = SandWOpenClose(fd, DISC, REC_SEND, UA, REC_SEND);
            if (res == -1)
            {
                printf("Could not receive UA Frame!\n");
                return -1;
```

```
            }
        }
        else if (recDISC == -1)
        {
            printf("Could not read from port!\n");
            return -1;
        }
    }

    return closePort(fd, status);
}
```

datalink.h

```
 #ifndef DATALINK_H
#define DATALINK_H

#pragma once

#include "protocol.h"
#include "port.h"
#include <unistd.h>

/**
 * @brief Establishes a connection between devices, using a Stop and Wait mech
anism.
 *
 * @param port the port to be opened
 * @param status SENDER/RECEIVER
 *
 * @return the file descriptor to be written on / read from on success, -
1 otherwise
 */
int llopen(int port, int status);

/**
 * @brief Sends an array of characters to the other device, using a Stop and W
ait mechanism
 *
 * @param fd the file descriptor
 * @param buffer the array to be sent
 * @param length the array length
 *
 * @return the amount of bytes sent; -1 on error
 */
int llwrite(int fd, char *buffer, int length);

/**
 * @brief Receives an array of characters from the other device, using a Stop
and Wait mechanism.
 *
 * @param fd the file descriptor
 * @param buffer the array to store the data received
 *
 * @return the amount of bytes received; -1 on error
 */
int llread(int fd, char *buffer);

/**
 * @brief Closes the connection previously established, using a Stop and Wait
mechanism.
 *
 * @param fd the file descriptor to be closed
 * @param status SENDER/RECEIVER
 *
 * @return 0 on success, -1 otherwise
 */
```

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

#endif /*DATALINK_H*/
```

dataStructures.h

```
#ifndef DATASTRUCTURES_H
#define DATASTRUCTURES_H

#pragma once

/**
 * @brief Struct to save file's information
*/
typedef struct File
{
    long int size;
    unsigned char *name;
    unsigned char *controlPacket;
    int lastIndex;
    unsigned char *data;
} File;

/**
 * @brief State of the state machine
*/
typedef enum State
{
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    BCC_OK,
    DATA,
    C2_RCV,
    BCC2_OK,
    STOP,
    IGNORE,
    REJECTED
} State;

/**
 * @brief state of the ack state machine
*/
typedef enum AckState
{
    START_ACK,
    FLAG_ACK,
    A_ACK,
    ACK_RCV,
    BCC_ACK,
    STOP_ACK
} AckState;

/**
 * Type of control command
*/
typedef enum ControlCommand
{
    SET,
    DISC,
    UA,
    RR,
    REJ
} ControlCommand;

#endif /*DATASTRUCTURES_H*/
```

macros.h

```
#ifndef MACROS_H
#define MACROS_H

#pragma once

#define BAUDRATE        B38400
#define MODEMDEVICE     "/dev/ttyS1"
#define _POSIX_SOURCE   1 /* POSIX compliant source */
#define _XOPEN_SOURCE   700
#define FALSE           0
#define TRUE            1
#define ATTEMPTS        4

//PORTS
#define COM0            0
#define COM1            1
#define COM10           10
#define COM11           11

//STATUS
#define SENDER          0
#define RECEIVER        1

//S or U Frames
#define S_FRAME_SIZE    5
#define FLAG            0x7E

#define SEND_REC        0X03 //commands from sender, answers from receiver
#define REC_SEND        0X01 //commands from receiver, answers from sender

#define SET_COMMAND     0x03
#define DISC_COMMAND    0x0B
#define UA_ANSWER       0x07
#define RR_ANSWER(R)    ((R == 0) ?  0x05 : 0x85)
#define REJ_ANSWER(R)   ((R == 0) ?  0x01 : 0x81)

//I Frames
#define IFRAME_SIZE     4096
#define DATA_MAX_SIZE   (IFRAME_SIZE - 6)
#define NS(S)           ((S == 0) ?  0x00 : 0x40)
#define ESCAPE          0x7D
#define STUFF_BYTE      0x20

//Packet
#define MAX_PACKET_SIZE (DATA_MAX_SIZE / 2)
#define MAX_K           (MAX_PACKET_SIZE - 4)
#define MAX_VALUE_SIZE  255

#define FILESIZE        0x00
#define FILENAME        0x01

#define DATA_BYTE       0x01
#define START_BYTE      0x02
#define END_BYTE        0x03

#endif /*MACROS_H*/
```

port.c

```
#include "port.h"
```

```
static struct termios oldtioReceiver, oldtioSender;

int initPort(int portInt, int vtime, int vmin, int status)
{

    char *port = (char *)malloc(12);
    if (port == NULL)
    {
        printf("Could not allocate memory for port!\n");
        return -1;
    }
    struct termios newtio;

    sprintf(port, "/dev/ttyS%d", portInt);

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

    /*save current port settings*/
    if ((status == SENDER && (tcgetattr(fd, &oldtioSender) == -
1)) || (status == RECEIVER && (tcgetattr(fd, &oldtioReceiver) == -1)))
    {
        free(port);
        perror("tcgetattr");
        return -1;
    }

    bzero(&newtio, 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] = vtime;
    newtio.c_cc[VMIN] = vmin;

    /*
    VTIME e VMIN devem ser alterados de forma a proteger com um temporizador a

    leitura do(s) pr?ximo(s) caracter(es)
    */

    tcflush(fd, TCIOFLUSH);

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

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

    free(port);
    return fd;
}

int closePort(int fd, int status)
{
    sleep(1);
```

```
    if ((status == SENDER && (tcsetattr(fd, TCSANOW, &oldtioSender) == -
1)) || (status == RECEIVER && (tcsetattr(fd, TCSANOW, &oldtioReceiver) == -
1)))
    {
        perror("tcsetattr");
        return -1;
    }

    close(fd);
    printf("Closed Port with success!\n");
    return 0;
}
```

port.h

```
#ifndef PORT_H
#define PORT_H

#pragma once

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

/**
 * @brief Opens a serial port, saving its attributes in the respective termios
 static struct
 *
 * @param portInt the port to be opened is "/dev/ttySX", where X is portInt
 * @param vtime the character timer
 * @param vmin the minimum number of characters to receive before satisfying t
he read
 * @param status SENDER/RECEIVER
 *
 * @return file descriptor of port on success; -1 on error
 */
int initPort(int portInt, int vtime, int vmin, int status);

/**
 * @brief Closes a serial port, setting its attributes from the respective ter
mios static struct
 *
 * @param fd the file descriptor of the port
 * @param status SENDER/RECEIVER
 *
 * @return 0 on success; -1 on error
 */
int closePort(int fd, int status);

#endif /*PORT_H*/
```

protocol.c

```
#include "protocol.h"

int alarmSender = 1;
int alarmReceiver = 1;

void alarmSenderHandler()
{
```

```
    alarmSender = 0;
    return;
}

void alarmReceiverHandler()
{
    alarmReceiver = 0;
    return;
}

int SandWOpenClose(int fd, ControlCommand send, char sendAddress, ControlComma
nd receive, char recAddress)
{
    int rec;

    for (int i = 0; i < ATTEMPTS; i++)
    {
        if (sendOpenCloseFrame(fd, send, sendAddress) == -1)
        {
            printf("Could not send Frame! Attempt number %d\n", i + 1);
            return -1;
        }

        alarmSender = 1;
        alarm(3);

        while (alarmSender)
        {
            rec = receiveOpenCloseFrame(fd, receive, recAddress);
            if (rec == 0)
            {
                alarm(0);
                break;
            }
            else if (rec < 0)
            {
                printf("Could not read from port! Code: %d\n", rec);
                return -1;
            }
        }

        if (alarmSender)
        {
            return 0;
        }
        else if (i < 3)
        {
            printf("Timeout number %d, trying again...\n", i + 1);
        }
    }
    return -1;
}

int sendOpenCloseFrame(int fd, ControlCommand command, int address)
{
    int res;
    unsigned char frame[S_FRAME_SIZE];

    frame[0] = FLAG;
    frame[1] = address;
    frame[4] = FLAG;

    switch (command)
    {
    case SET:
        frame[2] = SET_COMMAND;
        frame[3] = address ^ SET_COMMAND;
        break;
```

```
    case DISC:
        frame[2] = DISC_COMMAND;
        frame[3] = address ^ DISC_COMMAND;
        break;
    case UA:
        frame[2] = UA_ANSWER;
        frame[3] = address ^ UA_ANSWER;
        break;

    default:
        break;
    }

    res = write(fd, frame, S_FRAME_SIZE);
    if (res == -1)
        return -1;
    return 0;
}

int sendAckFrame(int fd, ControlCommand command, int r)
{
    int res;
    unsigned char frame[S_FRAME_SIZE];

    frame[0] = FLAG;
    frame[1] = SEND_REC;
    frame[4] = FLAG;

    switch (command)
    {
    case RR:
        frame[2] = RR_ANSWER(r);
        frame[3] = RR_ANSWER(r) ^ SEND_REC;
        break;
    case REJ:
        frame[2] = REJ_ANSWER(r);
        frame[3] = REJ_ANSWER(r) ^ SEND_REC;
        break;
    default:
        break;
    }

    res = write(fd, frame, S_FRAME_SIZE);
    if (res == -1)
        return -1;
    return 0;
}

int receiveOpenCloseFrame(int fd, ControlCommand command, int address)
{
    unsigned char buf[255];
    int res;

    State state = START;

    while (state != STOP && alarmSender == 1)
    {
        res = read(fd, buf, 1);
        if (res == 0)
            continue;
        if (res < 0)
            return -1;

        changeStateS(&state, buf[0], command, address);
    }

    return 0;
}
```

```
int receiveAckFrame(int fd, int ns)
{
    unsigned char buf[255];
    int res;
    int nr;
    int acknowledged = -1;

    AckState state = START_ACK;

    while (state != STOP_ACK && alarmSender == 1)
    {
        res = read(fd, buf, 1);

        if (res == 0) continue;
        if (res < 0) return -1;

        nr = changeStateAck(&state, buf[0]);

        if (state == ACK_RCV)
        {
            if (nr == ns || nr == 1 - ns)
            {
                acknowledged = 0;
            }
            else if (nr - 2 == 1 - ns)
            {
                acknowledged = 1;
            }
        }
    }

    return acknowledged;
}

int sendInfoFrame(int fd, int ns, unsigned char *info, int length)
{
    int res, index = 0;
    unsigned char bcc2 = 0x00;

    unsigned char *infoFrame = (unsigned char *)malloc(IFRAME_SIZE);
    if (infoFrame == NULL)
    {
        printf("Could not allocate memory for infoFrame!\n");
        return -1;
    }

    infoFrame[index++] = FLAG;
    infoFrame[index++] = SEND_REC;
    infoFrame[index++] = NS(ns);

    infoFrame[index++] = SEND_REC ^ NS(ns);

    for (int i = 0; i < length; i++)
    {
        bcc2 = bcc2 ^ info[i];
        if (info[i] == FLAG || info[i] == ESCAPE)
        {
            infoFrame[index++] = ESCAPE;
            infoFrame[index++] = info[i] ^ STUFF_BYTE;
        }
        else
        {
            infoFrame[index++] = info[i];
        }
    }

    while (index < (IFRAME_SIZE - 2))
```

```
    {
        infoFrame[index++] = 0x00;
    }

    infoFrame[index++] = bcc2;
    infoFrame[index++] = FLAG;

    res = write(fd, infoFrame, index);

    free(infoFrame);

    if (res < 1)
        return -1;

    return 0;
}

int receiveInfoFrame(int fd, unsigned char *info, int expectedNS)
{
    struct sigaction newAction, oldAction;

    newAction.sa_handler = alarmReceiverHandler;
    sigemptyset(&newAction.sa_mask);
    newAction.sa_flags = 0;

    sigaction(SIGALRM, &newAction, &oldAction);

    unsigned char buf[255];
    int res;
    int i = 0;
    int firstTime = TRUE;
    int escaped = FALSE;
    int duplicated = FALSE;

    int random;

    State state = START;

    alarm(300);

    while (state != STOP && state != IGNORE && state != REJECTED && alarmRecei
ver) {
        res = read(fd, buf, 1);

        if (res == 0) continue;
        if (res < 0) return -1;

        int aux = changeStateInfo(&state, buf[0]);
        if (aux != -1) {
            if (aux != expectedNS) duplicated = TRUE;
        }

        if(state == FLAG_RCV) {
            i = 0;
            firstTime = TRUE;
        }

        if (state == DATA && i < MAX_PACKET_SIZE)
        {

            if (firstTime) {
                if (duplicated) return -3;
                firstTime = FALSE;
                escaped = FALSE;
            }
            else
            {
                if (escaped)
```

```
                {
                    if (buf[0] == (FLAG ^ STUFF_BYTE))
                    {
                        info[i++] = FLAG;
                    }
                    else if (buf[0] == (ESCAPE ^ STUFF_BYTE))
                    {
                        info[i++] = ESCAPE;
                    }
                    escaped = FALSE;
                }
                else if (buf[0] == ESCAPE)
                {
                    escaped = TRUE;
                }
                else if (i < MAX_PACKET_SIZE)
                {
                    info[i++] = buf[0];
                }
            }
        }
    }

    if(alarmReceiver == 0) return -2;

    if (state == IGNORE) return -1;
    else if (state == REJECTED) return 1;

    return 0;
}
```

protocol.h

```
#ifndef PROTOCOL_H
#define PROTOCOL_H

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <unistd.h>
#include "macros.h"
#include "dataStructures.h"
#include "stateMachines.h"

extern int alarmSender;
extern int alarmReceiver;

/**
 * @brief Catches a specific signal and the value of alarmSender becomes 0
*/
void alarmSenderHandler();

/**
 * @brief Catches a specific signal and the value of alarmReceiver becomes 0
*/
void alarmReceiverHandler();

/**
 * @brief Sends a message and waits for response, using a Stop and Wait mechan
ism
 *
 * @param fd the file descriptor of port
 * @param send the command to be sent
 * @param sendAddress the address to be sent
 * @param receive the command to be received
```

```
 * @param recAddress the address to be received
 *
 * @return 0 on success; -1 on error
 */
int SandWOpenClose(int fd, ControlCommand send, char sendAddress, ControlComma
nd receive, char recAddress);

/**
 * @brief Creates and sends a frame (SET, DISC or UA)
 *
 * @param fd the file descriptor of port
 * @param command the command to be sent
 * @param address the address to be sent
 *
 * @return 0 on success; -1 on error
 */
int sendOpenCloseFrame(int fd, ControlCommand command, int address);

/**
 * @brief Creates and sends a frame ACK
 *
 * @param fd the file descriptor of port
 * @param send the command to be sent
 * @param r the required message (0/1)
 *
 * @return 0 on success; -1 on error
 */
int sendAckFrame(int fd, ControlCommand command, int r);

/**
 * @brief Receives a SET/DISC/UA frame and calls the function with the state m
achine
 *
 * @param fd the file descriptor of port
 * @param command the command to be received
 * @param address the address to be received
 *
 * @return 0 on success; -1 on error
 */
int receiveOpenCloseFrame(int fd, ControlCommand command, int address);

/**
 * @brief Receives an ACK frame and calls the function with the state machine
 *
 * @param fd the file descriptor of port
 * @param ns the expected message (0/1)
 *
 * @return 1 if message was rejected; 0 on success; -1 on error
 */
int receiveAckFrame(int fd, int ns);

/**
 * @brief Creates and sends a info frame
 *
 * @param fd the file descriptor of port
 * @param info the info to be sent
 * @param length the size of the info to be sent
 *
 * @return 0 on success; -1 on error
 */
int sendInfoFrame(int fd, int ns, unsigned char *info, int length);

/**
 * @brief Receives an info frame and calls the function with the state machine
 *
 * @param fd the file descriptor of port
 * @param info the info received
 * @param expectedNS the expected sequence number
```

```
 *
 * @return 1 to send a REJ message; 0 to send a RR message; -1 on error; -
2 on timeout; -3 if it's a duplicate
 */
int receiveInfoFrame(int fd, unsigned char *info, int expectedNS);

#endif /*PROTOCOL_H*/
```

stateMachines.c

```
#include "stateMachines.h"

unsigned char bcc2Check = 0x00;
unsigned char answer;
static int escaped = FALSE;
static int s = 0;
static int dataIndex = 0;

void changeStateS(State *state, unsigned char byte, ControlCommand command, un
signed char address)
{
    int isCorrect;

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

    case FLAG_RCV:
        if (byte == address)
        {
            *state = A_RCV;
        }
        else if(byte != FLAG)
        {
            *state = START;
        }

        break;

    case A_RCV:
        isCorrect = FALSE;

        switch (command)
        {
        case SET:
            if (byte == SET_COMMAND)
                isCorrect = TRUE;
            break;
        case DISC:
            if (byte == DISC_COMMAND)
                isCorrect = TRUE;
            break;
        case UA:
            if (byte == UA_ANSWER)
                isCorrect = TRUE;
            break;
        }

        if (isCorrect)
        {
            *state = C_RCV;
        }
```

```
        else if (byte == FLAG)
        {
            *state = FLAG_RCV;
        }
        else
        {
            *state = START;
        }

        break;

    case C_RCV:
        isCorrect = FALSE;

        switch (command) {
        case SET:
            if (byte == (address ^ SET_COMMAND))
                isCorrect = TRUE;
            break;
        case DISC:
            if (byte == (address ^ DISC_COMMAND))
                isCorrect = TRUE;
            break;
        case UA:
            if (byte == (address ^ UA_ANSWER))
                isCorrect = TRUE;
            break;
        }

        if (isCorrect)
        {
            *state = BCC_OK;
        }
        else if (byte == FLAG)
        {
            *state = FLAG_RCV;
        }
        else
        {
            *state = START;
        }
        break;

    case BCC_OK:
        if (byte == FLAG)
        {
            *state = STOP;
        }
        else
        {
            *state = START;
        }
        break;
    }

}

int changeStateInfo(State *state, unsigned char byte)
{
    switch (*state)
    {
    case START:
        if (byte == FLAG)
        {
            *state = FLAG_RCV;
        }
        break;
```

```
    case FLAG_RCV:
        if (byte == SEND_REC)
        {
            *state = A_RCV;
        }
        else if(byte != FLAG)
        {
            *state = START;
        }

        break;

    case A_RCV:
        if (byte == NS(s))
        {
            *state = C_RCV;
            return s;
        }
        else if (byte == NS(1 - s))
        {
            *state = C_RCV;
            s = 1 - s;
            return s;
        }
        else if (byte == FLAG)
        {
            *state = FLAG_RCV;
        }
        else
        {
            *state = START;
        }

        break;

    case C_RCV:
        if (byte == SEND_REC ^ NS(s))
        {
            dataIndex = 0;
            bcc2Check = 0x00;
            *state = DATA;
        }
        else if (byte == FLAG)
        {
            *state = FLAG_RCV;
        }
        else
        {
            *state = IGNORE;
        }
        break;
    case DATA:
        dataIndex++;
        if (dataIndex < DATA_MAX_SIZE)
        {
            if (escaped)
            {
                if (byte == (FLAG ^ STUFF_BYTE))
                {
                    bcc2Check = bcc2Check ^ FLAG;
                }
                else if (byte == (ESCAPE ^ STUFF_BYTE))
                {
                    bcc2Check = bcc2Check ^ ESCAPE;
                }
                escaped = FALSE;
            }
            else if (byte == FLAG)
```

```
            {
                bcc2Check = 0x00;
                dataIndex = 0;
                escaped = FALSE;
                *state = FLAG_RCV;
            }
            else if (byte == ESCAPE)
            {
                escaped = TRUE;
            }
            else
            {
                bcc2Check = bcc2Check ^ byte;
            }
        }
        else if (byte == FLAG)
        {
            bcc2Check = 0x00;
            dataIndex = 0;
            escaped = FALSE;
            *state = FLAG_RCV;
        }
        else
        {
            dataIndex = 0;
            escaped = FALSE;
            *state = C2_RCV;
        }
        break;
    case C2_RCV:

        if (byte == bcc2Check)
        {
            *state = BCC2_OK;
        }
        else if (byte == FLAG)
        {
            *state = FLAG;
        }
        else
        {
            *state = REJECTED;
        }
        break;
    case BCC2_OK:
        if (byte == FLAG)
        {
            *state = STOP;
        }
        else
        {
            *state = START;
        }
        break;
    }

    return -1;
}

int changeStateAck(AckState *state, unsigned char byte)
{
    int isCorrect;
    int nr = -1;

    switch (*state)
    {
    case START_ACK:
        if (byte == FLAG) {
```

```
            *state = FLAG_ACK;
        }
        break;

    case FLAG_ACK:
        if (byte == SEND_REC)
        {
            *state = A_ACK;
        }
        else if(byte != FLAG)
        {
            *state = START_ACK;
        }

        break;

    case A_ACK:
        isCorrect = FALSE;

        switch (byte)
        {
        case RR_ANSWER(0):
            nr = 0;
            isCorrect = TRUE;
            break;
        case RR_ANSWER(1):
            nr = 1;
            isCorrect = TRUE;
            break;
        case REJ_ANSWER(0):
            nr = 2;
            isCorrect = TRUE;
            break;
        case REJ_ANSWER(1):
            nr = 3;
            isCorrect = TRUE;
            break;
        }

        if (isCorrect)
        {
            answer = byte;
            *state = ACK_RCV;
        }
        else if (byte == FLAG)
        {
            *state = FLAG_ACK;
        }
        else
        {
            *state = START_ACK;
        }

        break;

    case ACK_RCV:
        if (byte == (SEND_REC ^ answer))
        {
            *state = BCC_ACK;
        }
        else if (byte == FLAG)
        {
            *state = FLAG_ACK;
        }
        else
        {
            *state = START_ACK;
        }
```

```
        break;

    case BCC_ACK:
        if (byte == FLAG)
        {
            *state = STOP_ACK;
        }
        else
        {
            *state = START_ACK;
        }
        break;
    }
    return nr;
}
```

stateMachines.h

```
#ifndef STATEMACHINES_H
#define STATEMACHINES_H

#include <stdio.h>
#include <string.h>
#include "macros.h"
#include "dataStructures.h"

/**
 * @brief Processes a byte of a SET/DISC/UA frame and updates the state
 *
 * @param state the state of the packet
 * @param byte the byte to process
 * @param command the command of the packet
 * @param address the address of the packet
 *
 */
void changeStateS(State *state, unsigned char byte, ControlCommand command, un
signed char address);

/**
 * @brief Processes a byte of an info frame and updates the state
 *
 * @param state the state of the packet
 * @param byte the byte to process
 *
 * @return the number of the expected message; -1 otherwise
 */
int changeStateInfo(State *state, unsigned char byte);

/**
 * @brief Processes a byte of an ACK frame and updates the state
 *
 * @param state the state of the packet
 * @param byte the byte to process
 *
 * @return the number of the expected message; -1 otherwise
 */
int changeStateAck(AckState *state, unsigned char byte);

#endif /*STATEMACHINES_H*/
```

Anexo 2 – Gráfico de Eficiência