1.º Trabalho Laboratorial
Ligação de Dados

Relatório

Mestrado Integrado em Engenharia Informática e Computação

Redes de Computadores

Turma 3

Diogo Sousa
João Varela
Tiago Verdade

Faculdade de Engenharia da Universidade do Porto

31 de Outubro de 2019

Índice

Sumário

O trabalho desenvolvido, no âmbito da cadeira de Redes de Computadores, consistiu na implementação de um protocolo de comunicação de dados assíncrona, e nos testes feitos a esse mesmo protocolo, desenvolvendo uma aplicação simples que permita a transferência de ficheiros através da porta série RS-232.

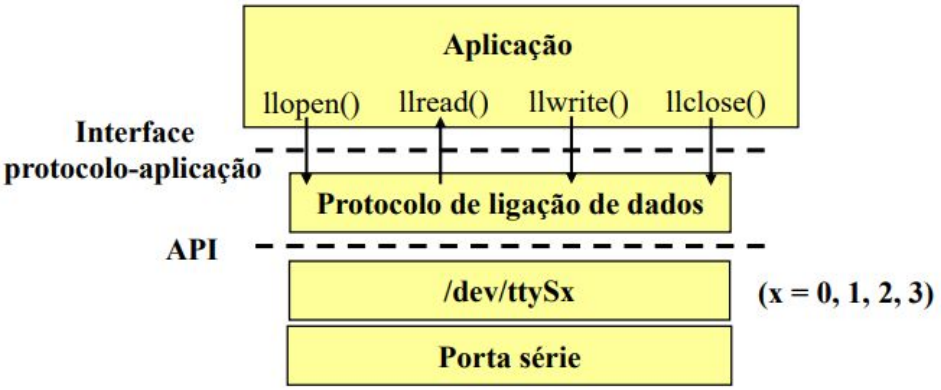
Durante a realização do projeto, concluiu-se que, para que se possa fazer uma comunicação estável e correta entre duas máquinas, é necessário um protocolo que permita a partilha de informação e a correção da mesma quando esta for entregue erroneamente. Para isso, são imprescindíveis vários mecanismos de sincronização e de deteção de erros, bem como uma arquitetura em camadas que confira independência entre as próprias.

Introdução

O objetivo inicial do trabalho laboratorial, passou por desenvolver um protocolo de ligação de dados que forneça um serviço de comunicação de dados fiável entre dois sistemas ligados por um meio de transmissão, sendo neste caso um cabo série. Para isso foi necessário implementar um leque de funções que garantam o sincronismo dos dados (organizando-os em tramas), estabelecimento/terminação da ligação, numeração de tramas, controlo de erros e controlo de fluxo.

Outro propósito, foi o desenvolvimento de um protocolo de aplicação para transferência de ficheiros, utilizando o serviço fornecido pelo protocolo de ligação de dados.

Tornou-se, portanto, também necessária a criação de uma interface entre ambos os protocolos, que contenha as funções necessárias para a comunicação entre as duas camadas. Esta arquitetura pode ser visualizada, de modo resumido, no seguinte esquema:



Este relatório tem como funções principais documentar o desenvolvimento do trabalho desenvolvido e explicitar os métodos e arquiteturas usadas para a execução do mesmo, bem como detalhar os casos de uso e definir os protocolos utilizados. No final do relatório, para além da conclusão, estão também retratados os testes efetuados bem como os cálculos realizados de forma a caracterizar a eficiência do protocolo e compará-la a valores teóricos.

Desta forma, o documento encontra-se dividido nas seguintes secções:

1. **Arquitetura** - descrição dos blocos funcionais e interfaces desenvolvidas;
2. **Estrutura do Código** - descrição da API, estruturas de dados utilizadas e principais funções;
3. **Casos de Uso Principais** - identificação dos principais casos de uso e diagramas de sequência que os representem;
4. **Protocolo de Ligação Lógica** - identificação dos principais aspetos funcionais e descrição da estratégia de implementação (com exemplos de código);
5. **Protocolo de Aplicação** - identificação dos principais aspetos funcionais e descrição da estratégia de implementação (com exemplos de código);
6. **Validação** - descrição dos testes efetuados e apresentação dos resultados;
7. **Eficiência do Protocolo de Ligação de Dados** - caracterização estatística da eficiência do protocolo implementado e comparação com valores teóricos;
8. **Conclusões** - síntese da informação apresentada e reflexão sobre os objetivos de aprendizagem;
9. **Anexos** - código fonte desenvolvido.

Arquitetura e Estrutura do Código

O programa desenvolvido encontra-se, essencialmente, dividido em duas camadas - a camada do Protocolo de Ligação de Dados e a camada de Aplicação; contendo para isso um conjunto de estruturas de dados que as representem e uma interface de comunicação entre as duas. Esta arquitetura baseia-se no princípio de independência entre camadas.

Camada de Ligação de Dados

Funções

A camada de mais baixo nível (ligação lógica de dados), é responsável pela troca de dados entre dois sistemas, contendo, por isso, funções capazes de:

- Estabelecer a ligação entre emissor e recetor - **openPort**;
- Terminar a ligação entre emissor e recetor - **closeTransmitter, closeReceiver**;
- Enviar uma mensagem (*frame*) - **writeDataFrame, writeControlFrame**;
- Receber uma mensagem (*frame*) - **readDataFrame, readControlFrame**.

Estas funções são responsáveis por ler e escrever dados diretamente na porta série, garantindo uma comunicação robusta e eficiente, através de mecanismos de numeração de tramas, stuffing, e deteção de erros. A camada de Ligação de Dados fornece uma interface que irá ser usada pela camada de Aplicação, com as seguintes funções:

- **llopen**;
- **llclose**;
- **llwrite**;
- **llread**.

Estruturas de Dados

Esta camada contém uma estrutura simples, que guarda os valores necessários para uma comunicação segundo o Protocolo de Ligação Lógica que está detalhado na sua própria secção.

```
typedef struct {
    char port[20]; /*Dispositivo /dev/ttySx, x = 0, 1, 2*/
    int baudRate; /*Velocidade de transmissão*/
    unsigned int sequenceNumber; /*Número de sequência da trama: 0, 1*/
    unsigned int timeout; /*Valor do temporizador*/
    unsigned int numTransmissions; /*Número de tentativas em caso de falha*/
    int status; /*TRANSMITTER | RECEIVER*/
    int T_prop; /*Tempo de atraso simulado*/
    int fer; /*Frame error ratio simulado*/
    int maxFrameSize; /*Tamanho da Trama*/
} DataLinkLayer;
```

Camada da Aplicação

Funções

A camada da Aplicação é responsável pelo envio e receção de ficheiros, dividindo-os em pacotes. Esta camada recorre à interface da camada de ligação de dados, sendo que os pacotes criados nesta camada, passam a servir como dados para as tramas (*frames*) na camada inferior. Desta forma, as funções principais podem ser resumidas nas seguintes:

- Inicializar a camada e os seus componentes - **initApplicationLayer**;
- Enviar um ficheiro - **SendFile**;
- Receber um ficheiro - **ReceiveFile**.

Sem esquecer que esta camada não conhece os detalhes do protocolo de ligação, mas apenas a forma como acede ao serviço

Estruturas de Dados

Nesta camada existem 3 estruturas de dados diferentes, duas para representar os dois tipos distintos de pacotes:

```
typedef struct {
    unsigned int sequenceNumber; /* Número de sequência do pacote */
    unsigned int size; /* Tamanho da informação a ser transportada */
    char * data; /* Informação a ser transportada */
} DataPacket;
```

```
typedef struct {
    int type; /* START_PACKET | END_PACKET */
    char  * fileName; /* Nome do ficheiro a transmitir */
    long fileSize; /* Tamanho do ficheiro a transmitir */
} ControlPacket;
```

E uma estrutura que contém dados importantes para o bom funcionamento da camada:

```
typedef struct {
    int fileDescriptor; /*Descritor do ficheiro correspondente ao serial port*/
    int serialPort; /*Numero do serial port a ser usado*/
    int status; /*TRANSMITTER | RECEIVER*/
    int dataSize; /*Tamanho maximo da informação num pacote de dados */
    char * fileName; /*Nome do ficheiro a ser transmitido*/
} ApplicationLayer;
```

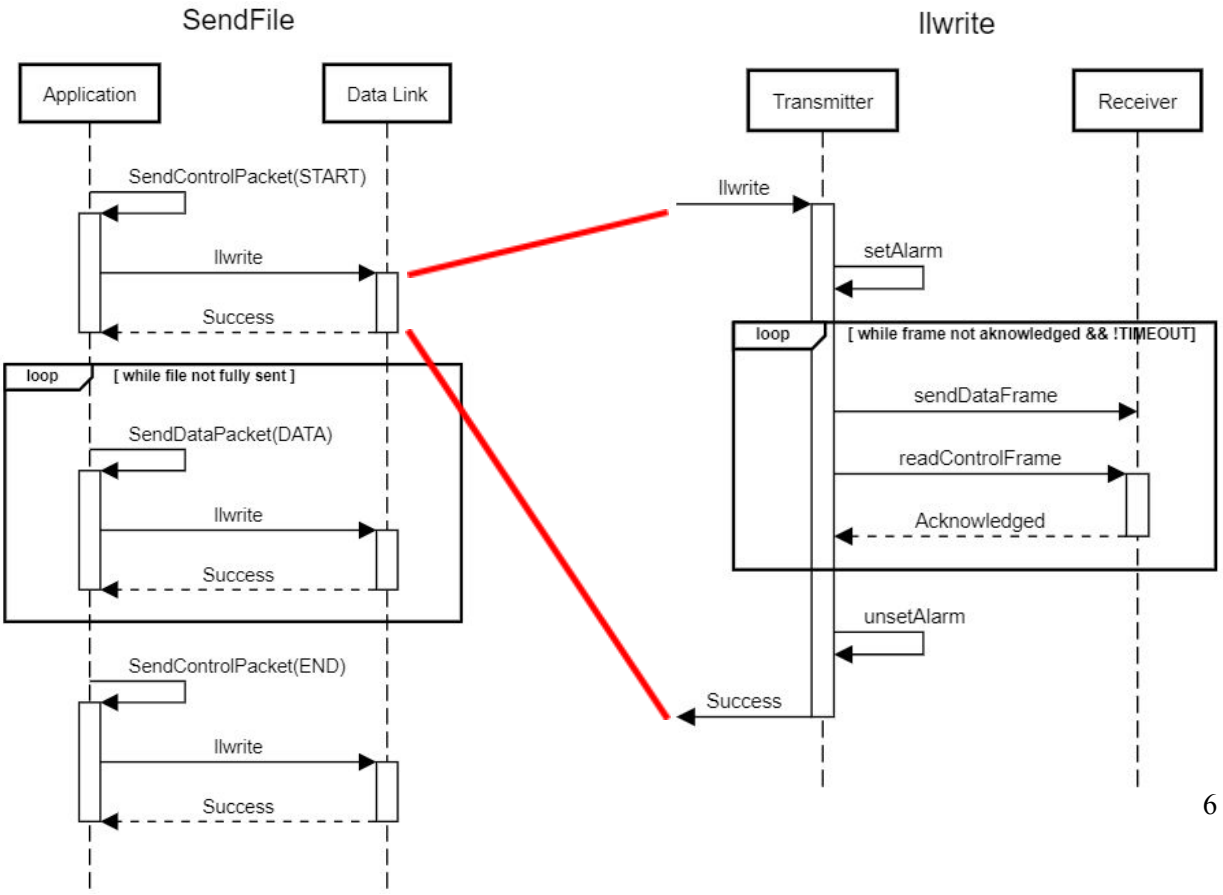
Interface

Existe também uma interface da linha de comandos que permite ao utilizador especificar uma série de parâmetros necessários ao funcionamento do programa e para que possam comparar resultados tendo em conta esses mesmos parâmetros (emissor/recetor, *baudrate*, tempo entre reenvio de frames no caso de falha (timeout), número máximo de tentativas, etc).

Casos de Uso Principais

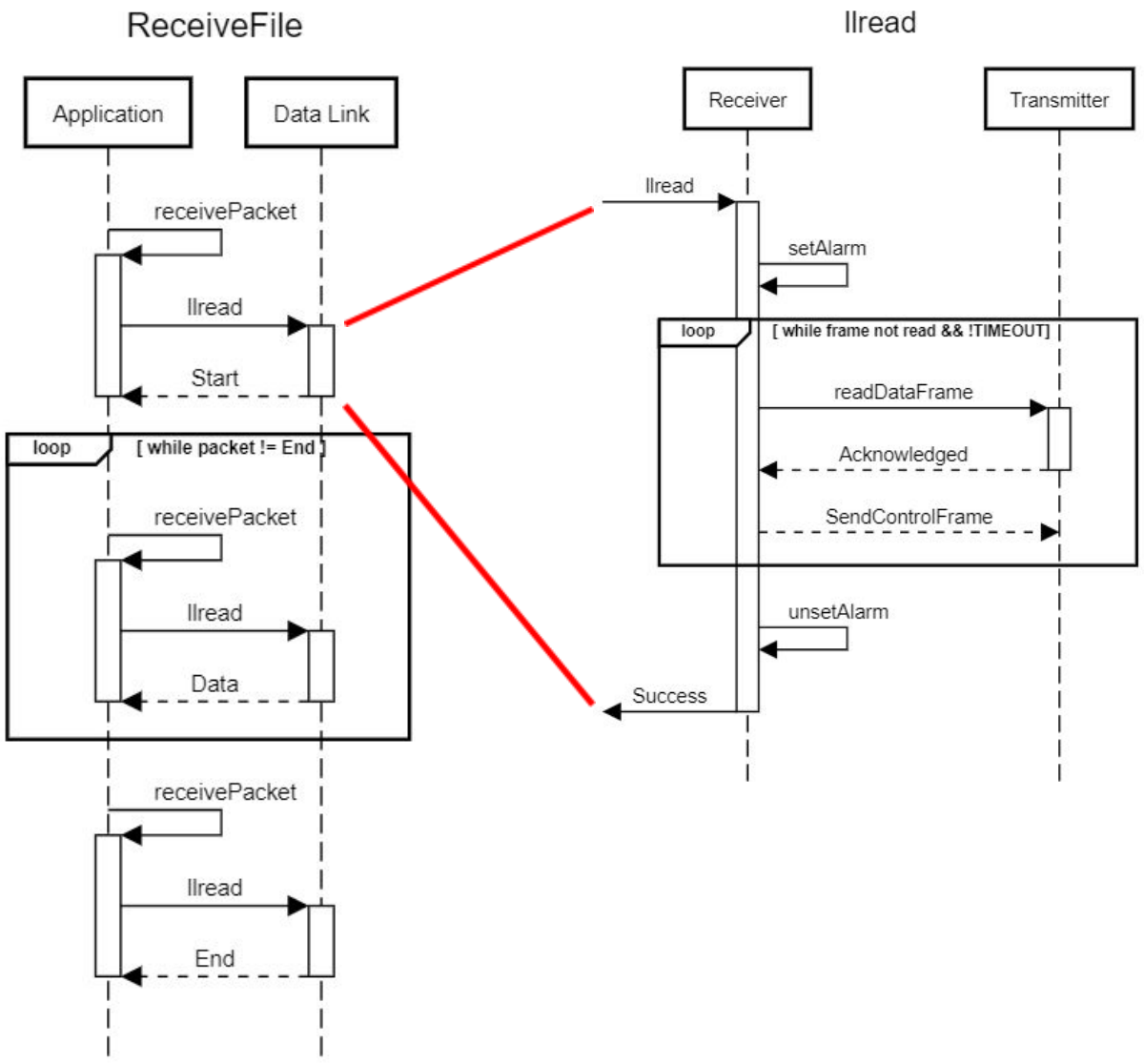
Essencialmente, existem dois casos de uso possíveis correspondentes ao emissor e ao recetor. Correndo o programa como **emissor** (gcc applicationLayer.c -o write && ./write <port nr> 1 <file_name>), após a chamada à função **llopen**, que estabelece a conexão com o recetor, é invocada a função **sendFile** que trata de abrir o ficheiro a transmitir, medir o seu tamanho e enviar um pacote de controlo (**sendControlPacket**) com a informação necessária para que seja possível enviar o ficheiro desejado. Após o reconhecimento do pacote de controlo, o emissor divide o ficheiro em segmentos e coloca-os em pacotes (**sendDataPacket**) que os envia para o recetor através da função **llwrite**. Quando acaba de enviar o ficheiro, é enviado um pacote de controlo que indica o fim da transferência do ficheiro, que será fechado.

Fig. 2 - Diagramas de Sequência (Emissor)



Correndo o programa como **recetor (**gcc applicationLayer.c -o read && ./read <port nr> 0 <file_name>), após invocação do **llopen**, é chamada a função **receiveFile** que após receber o pacote de controlo que indica o começo do envio do ficheiro, guarda a informação enviada pelo emissor (**receivePacket**) através do uso da função **llwrite** até que chegue um pacote de controlo que sinalize o final da transmissão de informação. A informação recebida vai ser escrita num ficheiro criado antes de começar a receção para que se possa obter uma cópia exata do ficheiro que foi transmitido.

Fig. 2 - Diagramas de Sequência (Recetor)



É de notar que em ambos os casos de uso, podem ser especificados vários argumentos (que tomam valores por defeito quando não o são) e que têm influência no decorrer do programa (**Usage: ./<Program Name> <Port Number> <0|1> <FileName> <BaudRate> <FrameSize> <MaxRetries> <Timeout> <Delay> <FER >**).

Protocolo de Ligação Lógica

O protocolo de ligação de dados fornece um serviço de comunicação entre dois sistemas, sendo responsável pela interação direta com a porta série. Para que a comunicação possa ser realizada, são necessárias funções responsáveis pelo estabelecimento e terminação da ligação (llopen e llclose) que realizam essas ações através do envio e receção de tramas de controlo.

Segundo o protocolo implementado, os dados devem estar organizados em tramas (*framing*) e essas devem ser delimitadas por um *header* e um *footer*, sendo usadas *flags* para delimitar o início e fim de trama. Podemos verificar essa técnica de *framing* na função responsável por escrever uma trama de dados no serial port.

```
int sendDataFrame(int fd, int Ns, char *buffer, int len) {
    unsigned char set[maxPerPacket];
    set[0] = FLAG;
    set[1] = A1;
    set[2] = Ns ? 0x40 : 0x00;
    set[3] = set[1] ^ set[2];
    . . .
    /* preenchimendo da informação e cálculo do bcc2 */
    . . .
    set[pos++] = bcc2;
    set[pos++] = FLAG;
    return write(fd, set, pos);
}
```

No caso de aparecer uma *flag* entre a informação a ser enviada, é necessário recorrer a uma técnica de *byte stuffing* para que esses dados não sejam mal interpretados. Esta técnica consiste em substituir o *byte* em questão por um *byte* do tipo *escape* e adicionar um *byte* à sua frente que corresponda à operação "XOR" entre o *byte* de informação original e o *byte* 0x20. O mesmo se fará para um *byte* de dados que corresponda ao *escape*.

```
if (buffer[i] == FLAG || buffer[i] == ESC)
    { //byte stuffing
         set[pos++] = ESC;
         set[pos++] = buffer[i] ^ STUFFING;
    }
 else
     set[pos++] = buffer[i];
```

Sem esquecer que, seguindo o mesmo raciocínio, ao ler-se um *frame* será também necessário proceder-se à técnica de *destuffing* para que se possa processar a informação pretendida. A leitura de uma trama é feita de modo canónico, lendo-se *bit* a *bit* alterando o estado de um *state machine* que auxilia o reconhecimento e análise dos *frames* recebidos.

Para que ambos, transmissor e recetor, consigam enviar e receber a informação que pretendem, é usada uma variante *Stop and Wait* (com janela unitária e numeração módulo 2) o que permitirá controlar erros que são detetados através de campos de controlo no *header* e no *footer* das tramas.

Se a leitura de uma trama for correta, será enviado um pacote de controlo ao escritor que sinaliza que a trama foi recebida corretamente, senão será enviado um pacote de controlo sinalizando a deteção de erro na trama e, por isso, essa será reenviada (este ciclo pode apenas ocorrer um número limitado de vezes, definido pelo utilizador). Antes da escrita de um pacote de dados é definido um *time-out* ,para o caso do transmissor não receber um pacote de controlo, obrigado à retransmissão do

pacote de dados. É também necessário ter em conta que estas retransmissões podem originar duplicados e, por isso, estes devem ser detetados e eliminados. Podemos observar, de seguida, um segmento de código que representa o processo descrito:

```
    int N = nSentSuccesfull % 2 ? 0x40 : 0x00, numtries = 0, bytesWritten = 0;
    unsigned char response;
    setAlarm();
    do {
        bytesWritten = sendDataFrame(fd, N, buffer, length);
        alarm(TIMEOUT);
        numtries++;
        response = readControlFrame(fd);

        if (response == RR_ONE) {
           if (!N){
                nSentSuccesfull++;
               break;
            } else {
                nSentSuccesfull++;
                readControlFrame(fd);  // when repeated there are multiples
controlframes, so we are getting the response of previous sent packages, this fixes
that error
              break;
            }
        }
        else if (response == RR_ZERO){
           if (N){
                nSentSuccesfull++;
                break;
            } else {
                nSentSuccesfull++;
                readControlFrame(fd);
                break;
            }
       }
    } while (numtries < MAX_NUM_TRIES);
    unsetAlarm();
```

Protocolo de Aplicação

O protocolo de aplicação desenvolvido tem como objetivo principal permitir a transferência de ficheiros, usando o serviço oferecido pelo protocolo de ligação de dados. A aplicação suporta dois tipos de pacotes enviados pelo emissor: pacotes de controlo e pacotes de dados.

Os pacotes de controlo guardam, para além do *byte* que indica o início ou terminação da transferência do ficheiro, o nome e tamanho do ficheiro no formato *TLV (Type-Length-Value)*.

```
    int sendControlPacket(long fileSize, int nameLength, int type) {
        int packetSize = 5 + sizeof(fileSize) + nameLength, i = 0;
        unsigned char controlPacket[packetSize];
        controlPacket[i++] = type;
        controlPacket[i++] = FILE_SIZE;
        controlPacket[i++] = sizeof(long);
        memcpy(&controlPacket[i], &fileSize, sizeof(fileSize));

        i += sizeof(fileSize);

        controlPacket[i++] = FILE_NAME;
        controlPacket[i++] = nameLength;
        memcpy(&controlPacket[i], app->fileName, nameLength);
        return llwrite(app->fileDescriptor, controlPacket, packetSize);
```

```
}
```

Já os pacotes de dados, contêm informação sobre o número de sequência do pacote, informação a transferir e tamanho dessa mesma informação.

```
int sendDataPacket(int sequenceNumber, unsigned char * data, int dataSize){
    int packetSize = 4 + dataSize;
    unsigned char dataPacket[packetSize];

    dataPacket[0] = DATA;
    dataPacket[1] = sequenceNumber % 256;
    dataPacket[2] = dataSize / 256;
    dataPacket[3] = dataSize % 256;
    memcpy(&dataPacket[4], data, dataSize);

    return llwrite(app->fileDescriptor, dataPacket, packetSize);
}
```

Na camada de aplicação, estão implementadas as funções *sendFile:* invocada no caso do trasmissor e que segmenta o ficheiro em pacotes de dados, escrevendo-os no *serial port* recorrendo à função llwrite; e a função *receiveFile*: invocada no caso do recetor, que lê pacotes de dados (recorrendo à função llread) e os coloca num novo ficheiro até que receba um pacote de controlo com a indicação de terminar a transferência.

Validação

Para confirmar a robustez do programa, foram realizados vários testes, de entre os quais se destacam:

- Geração aleatória de erros em tramas de informação (no cabeçalho e no campo de dados);
- Geração de atraso de propagação simulado;
- Interrupção da ligação (através do *serial port*) repetidamente, durante a transferência do ficheiro;
- Introdução de "ruído" no *serial port*, gerado fisicamente.

Gradualmente, foi possível tornar o software resistente a este tipo de erros, confirmando assim a robustez do programa desenvolvido.

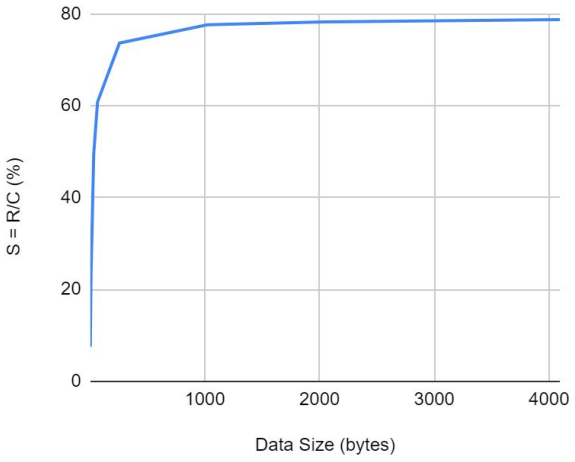
Eficiência do Protocolo de Ligação de Dados

Para que se possa ter uma noção exata da eficiência do programa implementado, calculou-se uma série de estatísticas que permitiram caracterizar o protocolo de ligação de dados. Para isso fizeram-se variar alguns fatores que influenciam a eficiência do protocolo.

Tamanho da Trama Variável

Com um *baudrate* constante de 38400 bytes e fazendo variar o tamanho dos dados nos pacotes, que por sua vez influenciam o tamanho dos *frames*, obtiveram-se os seguintes resultados:

Data Size (bytes)	Time Spent (s)	R (bit / s)	S = R/C (%)	Tf = L / R
2	30.57	2870	7.5	0.000696
8	9.80	8948	23.30	0.000894
32	4.62	19008	49.50	0.001683
64	3.76	23330	60.80	0.002743
256	3.10	28286	73.70	0.009050
1024	2.94	29848	77.70	0.034307
2048	2.92	30085	78.30	0.068073
4096	2.90	30242	78.80	0.135440



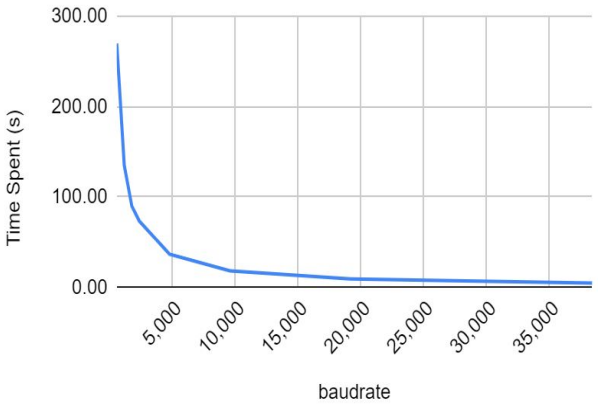
Como seria de esperar, o aumento do tamanho dos dados provoca uma diminuição do tempo de transferência, visto que o número de operações *write* e *read* são muito menores, bem como a quantidade de tramas de controlo que têm de ser enviadas e processadas.

Analisando o gráfico, podemos perceber que com o aumento do tamanho da trama obtém-se um aumento da eficiência, mais acentuado para tramas de dimensões reduzidas, atingindo um limite para valores maiores, aproximando-se do seu valor teórico que pode ser calculado através da seguinte fórmula: $S = \frac{Tf}{Tprop + Tf + Tprop}$. Sendo que uma trama maior implica um Tf maior ($Tf = \frac{L}{R}$) e o $Tprop$ se mantém, já era expectável que a eficiência aumentasse.

Baudrate Variável

Variando o *baudrate* e utilizando pacotes com 32 *bytes* de dados, foram obtidos os seguintes resultados.

baudrate	Time Spent (s)	R (bit / s)	S = R/C (%)
600	270.14	325	54.1
1200	135.10	650	54.1
1800	90.07	974	54.1
2400	73.30	1197	49.9
4800	36.67	2393	49.8
9600	18.35	4781	49.8
19200	9.20	9543	49.7
38400	4.62	19012	49.5

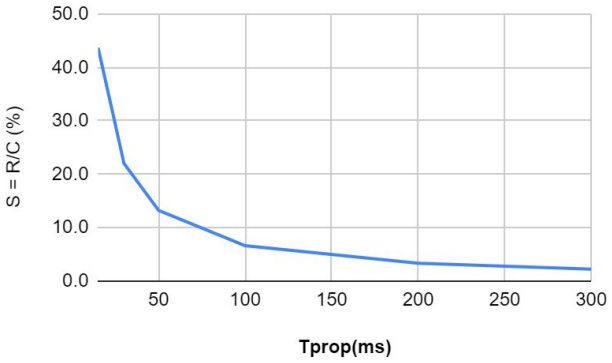


Podemos observar que o tempo gasto na transferência do ficheiro e o *baudrate* são inversamente proporcionais. No entanto, com o aumento do *baudrate* a eficiência tende a diminuir visto que o aumento de débito recebido é menor que o aumento de *baudrate*.

Tempo de Propagação (*Tprop*) Variável

Variando o tempo de propagação das tramas, utilizando um baudrate de 38400 e pacotes com 32 *bytes* de dados, obtiveram-se os seguintes resultados.

Tprop(ms)	Time Spent (s)	R (bit / s)	S = R/C (%)
15	5.23	16788	43.7
30	10.5	8437	22.0
50	17.30	5072	13.2
100	34.55	2539	6.6
200	69.05	1271	3.3
300	103.55	847	2.2

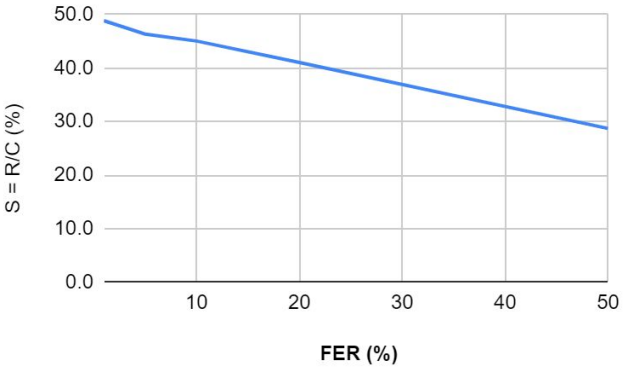


Simulando um tempo de propagação variável, comprovou-se que é inversamente proporcional à eficiência da transferência o que corresponde com a fórmula $S = \frac{Tf}{Tprop + Tf + Tprop}$.

Frame Error Ratio Variável

Variando o *frame error ratio*, utilizando um baudrate de 38400 e pacotes com 32 bytes de dados, obtiveram-se os seguintes resultados.

FER (%)	Time Spent (s)	R (bit / s)	S = R/C (%)
1	4.68	18740	48.8
5	4.94	17775	46.3
10	5.07	17291	45.0
20	5.57	15759	41.0
50	7.96	11018	28.7



Este experimento foi importante, não só para testar a robustez do programa desenvolvido, mas também para confirmar o efeito dos erros na eficiência da comunicação. A última decresce muito rapidamente com o aumento do *FER* (para um *Tprop* constante), o que também é demonstrável através da fórmula da eficiência tendo em conta a probabilidade de erros em tramas: $S = \frac{Tf}{E[A] * (Tprop + Tf + Tprop)}$ em que $E[A] = \frac{1}{1 - FER}$.

Conclusões

Através do projeto desenvolvido, foi possível compreender como funciona a comunicação entre máquinas através de uma porta série. Para que possa existir uma comunicação robusta e fiável é necessário que o protocolo seja bastante minucioso no que diz respeito à sincronização de mensagens e deteção de erros nas mesmas. A arquitetura em camadas independentes entre si permite uma maior flexibilidade no desenvolvimento do programa, visto cada uma desempenhar um papel específico e o facto de possibilitar alterar uma camada sem afetar a outra, porque estão ligadas apenas através de uma interface.

De modo a garantir a robustez do programa, realizaram-se vários testes descritos na secção de validação, o que nos permitiu verificar a qualidade do *software* desenvolvido e perceber que partes melhorar. Para além disso, mediram-se várias estatísticas que nos permitiram analisar a eficiência do protocolo implementado e confirmar factos discutidos nas aulas teóricas.

Todos os desafios propostos foram cumpridos, e o desafio de implementar um protocolo de ligação de dados possibilitou a consolidação e domínio de vários conceitos teóricos implícitos no trabalho.

Anexos

ApplicationLayer.h

```
#ifndef _APPLICATIONLAYER_H_
#define _APPLICATIONLAYER_H_

typedef struct {
   int fileDescriptor; /*File Descriptor correspondent to the serial port*/
   int serialPort; /*Number of the serial port being used*/
   int status; /*TRANSMITTER | RECEIVER*/
   int dataSize; /*Data bytes per data packet*/
   char * fileName; /*File to be transmitted*/
} ApplicationLayer;

typedef struct {
   int type; /* START_PACKET | END_PACKET */
   char  * fileName; /* Name of the file to be transmitted */
   long fileSize; /* Size of the file to be transmitted */
} ControlPacket;

typedef struct {
   unsigned int sequenceNumber; /* Sequence number of the data packet */
   unsigned int size; /* Size of the data on the packet */
   char * data; /* Data being transported on the packet */
} DataPacket;

/**
 * @brief Initializes the application layer and its components
 *
 * @param porta Serial port Number
 * @param type Connection Type
 * @param fileName File to be transmitted
 * @param baudRate Transmission speed
 * @param dataSize Bytes to be sent as data in each data packet
 * @param retries Number of retransmission attempts should an error occur
 * @param timeout Number of seconds before a timeout is declared
 * @param delay Simulated delay between receiving packets
 * @param random Simulated chance to generate a random processing error
 * @return int 0 on success, less than 0 otherwise
 */
int initApplicationLayer(int porta, int type, char * fileName, int baudRate, int dataSize, int retries, int
timeout, int delay, int random);

/**
 * @brief Frees the space allocated in the application layer
 */
void destroyApplicationLayer();

/**
 * @brief Sends a Control Packet
 *
```

```
 * @param fileSize Size of the file to be transmitted
 * @param nameLength Length of the name of the file
 * @param type  START_PACKET | END_PACKET
 * @return int 0 on success, less than 0 otherwise
 */
int sendControlPacket(long fileSize, int nameLength, int type);

/**
 * @brief Sends a Data Packet
 *
 * @param sequenceNumber Sequence Number of the Data packet
 * @param data Data to be transported
 * @param dataSize Size of the data
 * @return int 0 on success, less than 0 otherwise
 */
int sendDataPacket(int sequenceNumber, unsigned char * data, int dataSize);

/**
 * @brief Reads and stores a Data packet
 *
 * @param dataPacket Data packet to be read
 * @param storedPacket Stores the Data packet
 * @return int type of data packet read
 */
int receiveDataPacket(unsigned char * dataPacket, DataPacket * storedPacket);

/**
 * @brief Reads and stores a Control packet
 *
 * @param controlPacket Control packet to be read
 * @param storedPacket Stores the Control packet
 * @return int type of data packet read
 */
int receiveControlPacket(unsigned char * controlPacket, ControlPacket * storedPacket);

/**
 * @brief Calls the functions that receive packets, according to its type
 *
 * @param packet Packet to be read
 * @param storedPacket Stores the packet
 * @return int type of data packet read
 */
int receivePacket(unsigned char * packet, void * storedPacket);

/**
 * @brief Sends a file, calling all the necessary functions to write it
 *
 * @return int 0 on success, less than 0 otherwise
 */
int sendFile();

/**
 * @brief Receives a file, calling all the necessary functions to read it
 *
```

```
 * @return int Number of bits written, used for the efficiency calculation
 */
int receiveFile();

#endif // _APPLICATIONLAYER_H_
```

ApplicationLayer.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "dataLinkLayer.c"
#include "applicationLayer.h"

#define CLOCK_PRECISION 1E9
#define DATA 1
#define START_PACKET 2
#define END_PACKET 3
#define NOISE -1
#define FILE_SIZE 0
#define FILE_NAME 1
#define MAX_FILE_NAME 32
#define DEFAULT_DELAY 0
#define DEFAULT_FER 0
#define DEFAULT_BAUDRATE 7
#define DEFAULT_DATASIZE 32
#define DEFAULT_NUM_RETRIES 5
#define DEFAULT_TIMEOUT 3

static ApplicationLayer *app;

int initApplicationLayer(int porta, int type, char *fileName, int baudRate, int dataSize, int retries, int
timeout, int delay, int random)
{

   app = malloc(sizeof(ApplicationLayer));
   app->fileName = malloc(sizeof(char) * strlen(fileName));
   app->serialPort = porta;
   app->status = type;
   app->dataSize = dataSize;
   strcpy(app->fileName, fileName);

   if ((app->fileDescriptor = llopen(app->serialPort, app->status)) < 0)
   {
      perror("llopen");
      exit(-1);
   }

   if (type == TRANSMITTER)
      sendFile();
   else if (type == RECEIVER)
   {
```

```
        struct timespec start, end;
        clock_gettime(CLOCK_MONOTONIC_RAW, &start);
        int finalFileSize = receiveFile();
        clock_gettime(CLOCK_MONOTONIC_RAW, &end);
        double timeElapsed = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / CLOCK_PRECISION;
        printf("\nTime elapsed: %f s\n", timeElapsed);

        float transmissionSpeed = getBaudRate(baudRate);
        float bitsPerSecond = (float)finalFileSize / timeElapsed;

        printf("Bits received/second (R): %0.2f\n", bitsPerSecond);
        printf("Baudrate (C) : %d\n", printBaudRate(baudRate));
        printf("Efficiency (R/C) = %0.2f%%\n", bitsPerSecond / (float)printBaudRate(baudRate) * 100);
    }

    return llclose(app->fileDescriptor);
}

void destroyApplicationLayer()
{
    free(app->fileName);
    free(app);
}

int sendControlPacket(long fileSize, int nameLength, int type)
{
    int packetSize = 5 + sizeof(fileSize) + nameLength;
    unsigned char controlPacket[packetSize];

    int i = 0;

    controlPacket[i++] = type;
    controlPacket[i++] = FILE_SIZE;
    controlPacket[i++] = sizeof(long);
    memcpy(&controlPacket[i], &fileSize, sizeof(fileSize));

    i += sizeof(fileSize);

    controlPacket[i++] = FILE_NAME;
    controlPacket[i++] = nameLength;
    memcpy(&controlPacket[i], app->fileName, nameLength);

    return llwrite(app->fileDescriptor, controlPacket, packetSize);
}

int sendDataPacket(int sequenceNumber, unsigned char *data, int dataSize)
{
    int packetSize = 4 + dataSize;
    unsigned char dataPacket[packetSize];

    dataPacket[0] = DATA;
    dataPacket[1] = sequenceNumber % 256;
    dataPacket[2] = dataSize / 256;
    dataPacket[3] = dataSize % 256;
```

```
    memcpy(&dataPacket[4], data, dataSize);

    return llwrite(app->fileDescriptor, dataPacket, packetSize);
}

int receivePacket(unsigned char *packet, void *storedPacket)
{
    llread(app->fileDescriptor, packet);

    if (packet[0] == DATA)
    {
        return receiveDataPacket(packet, storedPacket);
    }
    else if (packet[0] == START_PACKET || packet[0] == END_PACKET)
    {
        return receiveControlPacket(packet, storedPacket);
    }
    else
    { //noise
        return NOISE;
    }
}

int receiveDataPacket(unsigned char *dataPacket, DataPacket *storedPacket)
{ //  returns type of packet -> DATA
    storedPacket->size = 256 * dataPacket[2] + dataPacket[3];
    storedPacket->sequenceNumber = dataPacket[1];
    storedPacket->data = malloc(storedPacket->size);
    memcpy(storedPacket->data, &dataPacket[4 + storedPacket->size], storedPacket->size);

    return dataPacket[0];
}

int receiveControlPacket(unsigned char *controlPacket, ControlPacket *storedPacket)
{ //  returns type of packet -> START_PACKET || END_PACKET
    storedPacket->type = controlPacket[0];
    storedPacket->fileName = malloc(controlPacket[4 + controlPacket[2]]);
    memcpy(&storedPacket->fileSize, &controlPacket[3], controlPacket[2]);
    memcpy(storedPacket->fileName, &controlPacket[5 + controlPacket[2]], controlPacket[4 +
controlPacket[2]]);

    return controlPacket[0];
}

int sendFile()
{
    FILE *file = fopen(app->fileName, "r");
    long fileSize = getFileSize(file);
    int nameLength = strlen(app->fileName);

    if (sendControlPacket(fileSize, nameLength, START_PACKET) <= 0)
    {
        perror("Transmitting Control Packet");
        exit(-1);
```

```
    }

    int sequenceNumber = 0, progress = 0;
    unsigned char *fileBuffer = (unsigned char *)malloc(app->dataSize * sizeof(char));
    int size;
    while ((size = fread(fileBuffer, sizeof(char), app->dataSize, file)))
    {
        if (sendDataPacket(sequenceNumber++, fileBuffer, size) <= 0)
        {
            perror("Transmitting Data Packet\n");
            exit(-1);
        }
        progress += app->dataSize;
        printProgressBar(progress, fileSize);
    }

    if (sendControlPacket(fileSize, nameLength, END_PACKET) <= 0)
    {
        perror("Transmitting Control Packet\n");
        exit(-1);
    }

    printf("\nFile sent\n");

    free(fileBuffer);

    if (fclose(file))
    {
        perror("Error while closing file\n");
        return -1;
    }
}

int receiveFile()
{
    ControlPacket storedControlPacket;
    int packetSize = 4 + app->dataSize;
    unsigned char controlPacket[packetSize];
    if (receivePacket(controlPacket, &storedControlPacket) != START_PACKET)
    {
        perror("Receiving Control\n");
        exit(-1);
    }
    long fileSize = storedControlPacket.fileSize;

    FILE *receivedFile = fopen(app->fileName, "wb");
    unsigned char packet[packetSize];
    DataPacket storedDataPacket;
    int lastPacket = 0, progress = 0, sequenceNumber = 0, noiseInput = 0, duplicatePackets = 0;
    int lastStoredDataPacket = -1;
    while (((lastPacket = receivePacket(packet, &storedDataPacket)) == DATA) || (lastPacket == NOISE))
    {
        if (lastPacket == NOISE)
        { //protection against noise
```

```
            noiseInput++;
            continue;
        }
        if (lastStoredDataPacket == storedDataPacket.sequenceNumber)
        { //duplicate packet -> ignore packet
            duplicatePackets++;
            fseek(receivedFile, -app->dataSize, SEEK_CUR);
            progress -= app->dataSize;
            sequenceNumber--;
            //continue;
        }
        lastStoredDataPacket = storedDataPacket.sequenceNumber;
        progress += app->dataSize;
        for (int i = 0; i < packet[2] * 256 + packet[3]; i++)
            fputc(packet[i + 4], receivedFile);

        sequenceNumber = (sequenceNumber + 1) % 256;
        printProgressBar(progress, fileSize);
    }
    printf("\nDuplicate Packets: %d", duplicatePackets);
    int finalFileSize = getFileSize(receivedFile) * 8;
    fclose(receivedFile);
    if (lastPacket != END_PACKET)
    {
        exit(-1);
    }

    return finalFileSize;
}

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

    int port, type, delay, random, baudRate, dataSize, maxTries, timeout;
    char *fileName;
    if (argc < 4 || (argc > 4 && argc < 10) || argc > 10)
    {
        printUsage(argv[0]);
        exit(1);
    }
    if ((argc == 4) &&
        ((strcmp("0", argv[2]) == 0) ||
         (strcmp("1", argv[2]) == 0)))
    {
        port = atoi(argv[1]);
        type = atoi(argv[2]);
        fileName = argv[3];
        baudRate = DEFAULT_BAUDRATE;
        dataSize = DEFAULT_DATASIZE;
        maxTries = DEFAULT_NUM_RETRIES;
        timeout = DEFAULT_TIMEOUT;
        delay = DEFAULT_DELAY;
        random = DEFAULT_FER;
    }
```

```
    else {
        port = atoi(argv[1]);
        type = atoi(argv[2]);
        fileName = argv[3];
        baudRate = atoi(argv[4]);
        dataSize = atoi(argv[5]);
        maxTries = atoi(argv[6]);
        timeout = atoi(argv[7]);
        delay = atoi(argv[8]);
        random = atoi(argv[9]);
    }

    //Estatistics

    //variable frame size
    // unsigned int sizes[8] = {2, 8, 32, 64, 256, 1024, 2048, 4096};
    // for (int i = 0; i < 8; i++) {
    //     printf("data size = %d\n", sizes[i]);
    //     initDataLinkLayer(port, type, baudRate, sizes[i], maxTries, timeout, delay, random);
    //     initApplicationLayer(port, type, fileName, baudRate, sizes[i], maxTries, timeout, delay, random);
    //     destroyApplicationLayer();
    //     printf("\n\n");
    // }

    //variable baudrate
    //  unsigned int baudRates[8] = {0, 1, 2, 3, 4, 5, 6, 7};
    //  for (int i = 0; i < 8; i++) {
    //      printf("baudRate test = %d\n", baudRates[i]);
    //      initDataLinkLayer(port, type, baudRates[i], dataSize, maxTries, timeout, delay, random);
    //      initApplicationLayer(port, type, fileName, baudRates[i], dataSize, maxTries, timeout, delay,
random);
    //      destroyApplicationLayer();
    //      printf("\n\n");
    //  }

    //variable frame error ratio
    // unsigned int FER[5] = {100, 20, 10, 5, 2};
    // for (int i = 0; i < 5; i++) {
    //     printf("FER = %d\n", FER[i]);
    //     initDataLinkLayer(port, type, baudRate, dataSize, maxTries, timeout, delay, FER[i]);
    //     initApplicationLayer(port, type, fileName, baudRate, dataSize, maxTries, timeout, delay, FER[i]);
    //     destroyApplicationLayer();
    //     printf("\n\n");
    //  }

    //variable propagation time
    // unsigned int delays[6] = {15000, 30000, 50000, 100000, 200000, 300000};
    //  for (int i = 0; i < 6; i++) {
    //      printf("delay = %d\n", delays[i]);
    //      initDataLinkLayer(port, type, baudRate, dataSize, maxTries, timeout, delays[i], random);
    //      initApplicationLayer(port, type, fileName, baudRate, dataSize, maxTries, timeout, delays[i],
random);
    //      destroyApplicationLayer();
    //      printf("\n\n");
```

```
    //  }


    initDataLinkLayer(port, type, baudRate, dataSize, maxTries, timeout, delay, random);
    initApplicationLayer(port, type, fileName, baudRate, dataSize, maxTries, timeout, delay, random);
    destroyApplicationLayer();

    exit(0);
}
```

DataLinkLayer.h

```
#ifndef _DATALINKLAYER_H_
#define _DATALINKLAYER_H_

typedef struct {
    char port[20]; /*Dispositivo /dev/ttySx, x = 0, 1*/
    int baudRate; /*Velocidade de transmissão*/
    unsigned int sequenceNumber; /*Número de sequência da trama: 0, 1*/
    unsigned int timeout; /*Valor do temporizador*/
    unsigned int numTransmissions; /*Número de tentativas em caso de falha*/
    int status; /*TRANSMITTER | RECEIVER*/
    int T_prop; /*Simulated delay upon recieving packets*/
    int fer; /*Simulated random error chance upon recieving packets*/
    int maxFrameSize;
} DataLinkLayer;

/**
 * @brief Makes so that the a packet is retransmitted
 *
 * @param signal
 */
void alarmHandler(int signal);

/**
 * @brief Set the Alarm object
 *
 */
void setAlarm();

/**
 * @brief Uninstalls the alarm
 *
 */
void unsetAlarm();

/**
 * @brief Sends a Control frame
 *
 * @param fd File descriptor where to write the frame
 * @param controlField Control field of the frame
 * @return int 0 on sucess, less than 0 otherwise
```

```
 */
int sendControlFrame(int fd, char controlField);

/**
 * @brief Reads a Control frame and checks its control field
 *
 * @param fd File descriptor from where to read the frame
 * @param controlField Control field that is expected
 * @return int 0 on sucess, less than 0 otherwise
 */
int checkControlFrame(int fd, char controlField);

/**
 * @brief Reads a Control frame
 *
 * @param fd File descriptor from where to read the frame
 * @return int Control field read on sucess, less than 0 otherwise
 */
int readControlFrame(int fd);

/**
 * @brief Sends a Data frame
 *
 * @param fd File descriptor where to write the frame
 * @param Ns Control field of the frame
 * @param buffer Buffer containing the data to be sent
 * @param len Length of the buffer
 * @return int 0 on sucess, less than 0 otherwise
 */
int sendDataFrame(int fd, int Ns, char *buffer, int len);

/**
 * @brief Reads a Data frame
 *
 * @param fd File descriptor from where to read the frame
 * @param buffer Buffer containing the data to be read
 * @return int 0 on sucess, less than 0 otherwise
 */
int readDataFrame(int fd, char *buffer);

/**
 * @brief Opens serial port and sets its attributes accordingly
 *
 * @return int File descriptor corresponding to the port
 */
int openPort();

/**
 * @brief Opens serial port and establishes the connection
 *
 * @param porta Number of the serial port
 * @param type Connection type
 * @return int File descriptor corresponding to the port
 */
```

```
int llopen(int porta, int type);

/**
 * @brief Communication that allows the transmitter to stop
 *
 * @param fd File descriptor corresponding to the port
 */
void closeTransmitter(int fd);

/**
 * @brief Communication that allows the receiver to stop
 *
 * @param fd File descriptor corresponding to the port
 */
void closeReceiver(int fd);

/**
 * @brief Opens serial port and terminates the connection
 *
 * @param fd File descriptor corresponding to the port
 * @return int 0 on success, less than 0 otherwise
 */
int llclose(int fd);

/**
 * @brief Reads a Data frane
 *
 * @param fd File descriptor from where to read the frame
 * @param buffer Buffer containing the data to be read
 * @return int int 0 on sucess, less than 0 otherwise
 */
int llread(int fd, char *buffer);

/**
 * @brief Writes the data frame, trying at most MAX_NUM_TRIES
 *
 * @param fd File descriptor where to write the frame
 * @param buffer Buffer containing the data to be written
 * @param length Length of the data
 * @return int bytes written on success, -1 otherwise
 */
int llwrite(int fd, char *buffer, int length);

#endif // _DATALINKLAYER_H_
```

DataLinkLayer.c

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

```c
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include "dataLinkLayer.h"
#include "utils.c"

#define BAUDRATE_FAST B38400
#define BAUDRATE_MEDIUM B19200
#define BAUDRATE_SLOW B4800
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FALSE 0
#define TRUE 1

#define START 0
#define FLAG_RCV 1
#define A_RCV 2
#define C_RCV 3
#define BCC_OK 4
#define END 5

#define FLAG 0x7E
#define ESC 0x7D
#define STUFFING 0x20
#define A 0x03
#define A1 0x03
#define A2 0x01
#define SET 0x03
#define UA 0x07
#define DISC 0x0B

#define HEADERSIZE 4
#define FOOTERSIZE 3

#define RR_ONE 0x85
#define RR_ZERO 0x05
#define REJ_ONE 0x81
#define REJ_ZERO 0x01

#define RECEIVER 0
#define TRANSMITTER 1

static struct termios oldtio;
static int destuffEscape = FALSE;

volatile int STOP = FALSE;
volatile int nSentSuccesfull = 0;

static DataLinkLayer *dataLink;

void initDataLinkLayer(int porta, int type, int baudRate, int dataSize, int retries, int timeout, int delay, int
random)
{
   dataLink = malloc(sizeof(DataLinkLayer));
```

```
    snprintf(dataLink->port, 11, "/dev/ttyS%d", porta);
    dataLink->status = type;
    dataLink->baudRate = baudRate;
    dataLink->numTransmissions = retries;
    dataLink->timeout = timeout;
    dataLink->T_prop = delay;
    dataLink->fer = random;
    dataLink->maxFrameSize = HEADERSIZE + 4 + dataSize * 2 + FOOTERSIZE;
}

void alarmHandler(int signal)
{
    if (signal != SIGALRM)
    {
        printf("disconnected\n");
        sleep(3);
        printf("connected\n");
        return;
    }

    STOP = TRUE;
    printf("TIME OUT\n");
}
void setDisconnect()
{
    struct sigaction action;
    memset(&action, 0, sizeof action);
    action.sa_handler = alarmHandler;
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;
    sigaction(SIGINT, &action, NULL);
}

void unsetDisconect()
{
    struct sigaction action;
    action.sa_handler = NULL;
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;
    sigaction(SIGALRM, &action, NULL);
}

void setAlarm()
{
    struct sigaction action;
    memset(&action, 0, sizeof action);
    action.sa_handler = alarmHandler;
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;
    sigaction(SIGALRM, &action, NULL);
}

void unsetAlarm()
{
```

```
    STOP = FALSE;
    struct sigaction action;
    action.sa_handler = NULL;
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;
    sigaction(SIGALRM, &action, NULL);
    alarm(0);
}

int sendControlFrame(int fd, char controlField)
{
    unsigned char set[5];
    set[0] = FLAG;

    // (0x03) em Comandos enviados pelo Emissor e Respostas enviadas pelo Receptor
    // (0x01) em Comandos enviados pelo Receptor e Respostas enviadas pelo Emissor
    if (dataLink->status == RECEIVER)
    {
        if (controlField == SET || controlField == DISC)
            set[1] = A2;
        else
            set[1] = A1;
    }
    else if (dataLink->status == TRANSMITTER)
    {
        if (controlField == SET || controlField == DISC)
            set[1] = A1;
        else
            set[1] = A2;
    }

    set[2] = controlField;
    set[3] = set[1] ^ set[2];
    set[4] = FLAG;
    return write(fd, set, 5);
}

int checkControlFrame(int fd, char controlField)
{
    unsigned char c;
    int state = START;
    unsigned char c_received;
    unsigned char expectedAddress;

    if (dataLink->status == RECEIVER)
    {
        if (controlField == SET || controlField == DISC)
            expectedAddress = A1;
        else
            expectedAddress = A2;
    }
    else if (dataLink->status == TRANSMITTER)
    {
        if (controlField == SET || controlField == DISC)
```

```
            expectedAddress = A2;
        else
            expectedAddress = A1;
    }

    STOP = FALSE;
    simulateDelay(dataLink->T_prop);

    while (STOP == FALSE)
    {
        c = 0;
        read(fd, &c, 1);

        switch (state)
        {
        case START:
            if (c == FLAG)
                state = FLAG_RCV;
            break;
        case FLAG_RCV:
            if (c == expectedAddress)
                state = A_RCV;
            else
                state = START;
            break;
        case A_RCV:
            if ((c_received = c) == controlField)
            {
                state = C_RCV;
            }
            else if (c == FLAG)
                state = FLAG_RCV;
            else
                return -1;
        case C_RCV:
            if (c == FLAG)
                state = FLAG_RCV;
            else if (c == expectedAddress ^ c_received)
                state = BCC_OK;
            else
                state = START;
            break;
        case BCC_OK:
            if (c == FLAG)
            {
                return 0;
                STOP = TRUE;
                break;
            }
        }
    }
    return -1;
}
```

```
int readControlFrame(int fd)
{
  unsigned char c;
  int state = START;
  unsigned char c_received;

  STOP = FALSE;

  simulateDelay(dataLink->T_prop);
  while (STOP == FALSE)
  {
    c = 0;
    read(fd, &c, 1);

    switch (state)
    {
    case START:
      if (c == FLAG)
        state = FLAG_RCV;
      break;
    case FLAG_RCV:
      if (c == A)
        state = A_RCV;
      else
        state = START;
      break;
    case A_RCV:
      if ((c_received = c) == c & SET || c == c & DISC || c == c & UA || c == c & RR_ONE || c == c &
REJ_ONE)
      {
        state = C_RCV;
      }
      else if (c == FLAG)
        state = FLAG_RCV;
      else
        state = START;
      break;
    case C_RCV:
      if (c == FLAG)
        state = FLAG_RCV;
      else if (c == A ^ c_received)
        state = BCC_OK;
      else
        state = START;
      break;
    case BCC_OK:
      if (c == FLAG)
      {
        return c_received;
        STOP = TRUE;
        break;
      }
    }
  }
```

```
    return -1;
}

int openPort()
{
    struct termios newtio;
    int fd = open(dataLink->port, O_RDWR | O_NOCTTY);
    if (fd < 0)
    {
        perror("open port");
        exit(-1);
    }

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

    bzero(&newtio, sizeof(newtio));
    newtio.c_cflag = getBaudRate(dataLink->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 */

    tcflush(fd, TCIOFLUSH);

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

    return fd;
}

int llopen(int porta, int type)
{
    setDisconnect();

    int fd = openPort();

    int tries = 0;

    if (dataLink->status == TRANSMITTER)
    {
        while (tries++ < dataLink->numTransmissions)
        {
            setAlarm();
```

```
            if (sendControlFrame(fd, SET) > 0)
            {
                alarm(dataLink->timeout);
                if (checkControlFrame(fd, UA) == 0)
                    return fd;
            }
            unsetAlarm();
        }
    }
    else if (dataLink->status == RECEIVER)
    {
        if (checkControlFrame(fd, SET) == 0 && sendControlFrame(fd, UA) > 0)
            return fd;
    }
    else
        return -2;

    return -1;
}

void closeTransmitter(int fd)
{
    int numtries = 0;
    setAlarm();
    do
    {
        sendControlFrame(fd, DISC);
        alarm(dataLink->timeout);
        if (checkControlFrame(fd, DISC) == 0)
        {
            sendControlFrame(fd, UA);
            sleep(1); //give time for receiver to read UA, before closing connection
            break;
        }
        numtries++;
    } while (numtries < dataLink->numTransmissions);
    unsetAlarm();
}

void closeReceiver(int fd)
{
    int numtries = 0;
    setAlarm();
    do
    {
        if (checkControlFrame(fd, DISC) == 0)
        {
            sendControlFrame(fd, DISC);
            alarm(dataLink->timeout);
            if (checkControlFrame(fd, UA) == 0)
                break;
        }
        numtries++;
    } while (numtries < dataLink->numTransmissions);
```

```
    unsetAlarm();
}

int llclose(int fd)
{

    if (TRANSMITTER == dataLink->status)
    {
        closeTransmitter(fd);
    }
    else if (RECEIVER == dataLink->status)
    {
        closeReceiver(fd);
    }
    if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
    {
        perror("tcsetattr");
        return -1;
    }

    if (close(fd) < 0)
    {
        perror("close fd");
        return -1;
    }

    unsetDisconect();

    return 0;
}

int sendDataFrame(int fd, int Ns, char *buffer, int len)
{
    unsigned char set[dataLink->maxFrameSize];

    set[0] = FLAG;
    set[1] = A1;
    set[2] = Ns ? 0x40 : 0x00;
    set[3] = set[1] ^ set[2];

    int pos = 4;
    for (int i = 0; i < len && pos < dataLink->maxFrameSize - 2; i++)
    {
        if (buffer[i] == FLAG || buffer[i] == ESC)
        { //byte stuffing
            set[pos++] = ESC;
            set[pos++] = buffer[i] ^ STUFFING;
        }
        else
            set[pos++] = buffer[i];
    }

    unsigned char bcc2 = 0;
    for (int i = 3; i < pos; i++)
```

```
        bcc2 ^= set[i];

    if (bcc2 == FLAG || bcc2 == ESC)
    {
        set[pos++] = ESC;
        set[pos++] = bcc2 ^ STUFFING;
    }
    else
    {
        set[pos++] = bcc2;
    }

    set[pos++] = FLAG;

    return write(fd, set, pos);
}

int readDataFrame(int fd, char *buffer)
{

    unsigned char c;
    int state = START;
    unsigned char c_received;

    int i = -1;
    unsigned char xor = 0;
    unsigned char last = 0;
    int error = generateErrorChance(dataLink->fer);

    STOP = FALSE;
    simulateDelay(dataLink->T_prop);

    while (STOP == FALSE)
    {
        read(fd, &c, 1);
        switch (state)
        {
        case START:
            if (c == FLAG)
                state = FLAG_RCV;
            else {

            }
            break;
        case FLAG_RCV:
            xor = 0;
            last = 0;
            if (c == A1)
                state = A_RCV;
            else if (c == FLAG)
                state = FLAG_RCV;
            else
                state = START;
            break;
```

```
case A_RCV:
  if ((c_received = c) == 0x40 || c == 0x00)
  {
    state = C_RCV;
  }
  else if (c == FLAG)
    state = FLAG_RCV;
  else if (c == DISC)
  {
    state = C_RCV;
  }
  else
    state = START;
  break;
case C_RCV:
  if (c == FLAG)
    state = FLAG_RCV;
  else if (c == A1 ^ c_received)
  {
    state = BCC_OK;
    xor ^= c;
  }
  else
    state = START;
  break;
case BCC_OK:
  if (c == FLAG)
  {
    if (i == -1 && c_received == DISC)
      return 1;
    if (last == xor || (destuffEscape && (last ^ STUFFING == xor)))
    {
      destuffEscape = FALSE;
      unsigned int token = ((c_received ^ 0x40) << 1) | RR_ZERO;
      if (error == 0)
        sendControlFrame(fd, token);
      else
      {
        printf(" - Random data error simulated.\n");
        if (c_received == 0x40)
          sendControlFrame(fd, REJ_ZERO);
        else if (c_received == 0x00)
          sendControlFrame(fd, REJ_ONE);
      }

      return 0;
    }

    unsigned int token = ((c_received) << 1) | REJ_ZERO;
    if (error == 0)
      if (c_received == 0x40)
        sendControlFrame(fd, REJ_ZERO);
      else if (c_received == 0x00)
        sendControlFrame(fd, REJ_ONE);
```

```
            else
            {
               printf(" - Random data error simulated.\n");
               if (c_received == 0x40)
                  sendControlFrame(fd, REJ_ZERO);
               else if (c_received == 0x00)
                  sendControlFrame(fd, REJ_ONE);
            }
         return 0;
      }
      else
      {

         if (last == ESC)
         {
            if (destuffEscape)
            {
               buffer[i] = last ^ STUFFING;
               xor ^= buffer[i];
               last = c;
               destuffEscape = FALSE;
               i++;
               continue;
            }
            else
            {
               destuffEscape = TRUE;
               xor ^= last;
               last = c;
               continue;
            }
         }

         if (i != -1)
         {
            if (destuffEscape)
            {
               buffer[i] = last ^ STUFFING;
               xor ^= last;
               destuffEscape = FALSE;
            }
            else
            {
               buffer[i] = last;
               xor ^= last;
            }
         }
         last = c;
         i++;
      }
   }
}
sendControlFrame(fd, REJ_ZERO);
return -1;
```

```
}

int llread(int fd, char *buffer)
{
   int numtries = 0;
   int ret;
   setAlarm();
   do
   {
      alarm(dataLink->timeout);
      ret = readDataFrame(fd, buffer);
      numtries++;
   } while ((ret != 0) && (numtries < dataLink->numTransmissions));
   unsetAlarm();

   if (numtries >= dataLink->numTransmissions && ret == -1)
   { //could not read frame
      perror("Connection Timed Out");
      exit(-1);
   }
   return ret;
}

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

   int N = nSentSuccesfull % 2 ? 0x40 : 0x00, numtries = 0, bytesWritten = 0;
   unsigned char response;
   setAlarm();
   do
   {
      bytesWritten = sendDataFrame(fd, N, buffer, length);
      alarm(dataLink->timeout);
      numtries++;
      response = readControlFrame(fd);

      if (response == RR_ONE)
      {
         if (!N)
         {
            nSentSuccesfull++;
            break;
         }
         else
         {
            nSentSuccesfull++;
            readControlFrame(fd); // when repeated there are multiples controlframes, so we are getting
the response of previous sent packages, this fixes that error
            break;
         }
      }
      else if (response == RR_ZERO)
      {
         if (N)
```

```
            {
                nSentSuccesfull++;
                break;
            }
            else
            {
                nSentSuccesfull++;
                readControlFrame(fd);
                break;
            }
        }
    } while (numtries < dataLink->numTransmissions);
    unsetAlarm();

    if (numtries >= dataLink->numTransmissions && response == 255)
    { //could not write frame
        perror("Connection Timed Out");
        exit(-1);
    }

    return bytesWritten;
}
```

Utils.c

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>

long getFileSize(FILE *f)
{
    fseek(f, 0, SEEK_END); // seek to end of file
    long size = ftell(f);  // get current file pointer
    fseek(f, 0, SEEK_SET); // seek back to beginning of file
    return size;
}

void printProgressBar(int progress, long fileSize)
{
    char *progressBar = calloc(32, sizeof(char));
    memcpy(progressBar, "Progress [                ]", 32);
    int progressPercentage = (int)((float)progress / (float)fileSize * 100);
    if (progressPercentage > 100)
    progressPercentage = 100;
    for (int i = 0; i < progressPercentage / 5; i++)
    {
        progressBar[i + 10] = '#';
    }

    printf("\r%s %d%%", progressBar, progressPercentage);
    fflush(stdout);
```

```
}

void printUsage(char * progName) {
   printf("Usage:\t%s <Port Number> <0 (Receiver) || 1 (Sender)> <FileName (to write to/send)>
<BaudRate: 0(B38400) || 1(B19200) || 2(B4800)> <DataSize> <MaxRetries> <Timeout> <Delay (in
microseconds)> <RandomError (1 in RandomError chances)>\n", progName);
   printf("\tex: %s 0 0 pinguim.gif 12000 20 3 3 2000 30\n", progName);
   printf("Can also use the first 3 arguments and remaining ones will take default values\n");
   printf("\tex: %s 0 0 pinguim.gif\n", progName);
}

int generateErrorChance(unsigned int fer)
{
   if (fer == 0)
      return 0;
   static int rngInitialized = 0;
   if (rngInitialized == 0) {
      srandom(time(NULL));
      rngInitialized = 1;
   }
   int randNum = random() % fer;
   if (randNum == 0)
   return 1;
   else
   return 0;
}

void simulateDelay(unsigned int T_prop) {
   usleep(T_prop);
}

int getBaudRate(int baudRateIndex) {
   int baudrate = 0;
   switch (baudRateIndex)
   {
   case 0:
      baudrate = B600;
      break;
   case 1:
      baudrate = B1200;
      break;
   case 2:
      baudrate = B1800;
      break;
   case 3:
      baudrate = B2400;
      break;
   case 4:
      baudrate = B4800;
      break;
   case 5:
      baudrate = B9600;
      break;
   case 6:
```

```
        baudrate = B19200;
        break;
    case 7:
        baudrate = B38400;
        break;
    default:
        break;
    }
    return baudrate;
}

int printBaudRate(int baudRateIndex) {
    int baudrate = 0;
    switch (baudRateIndex)
    {
    case 0:
        baudrate = 600;
        break;
    case 1:
        baudrate = 1200;
        break;
    case 2:
        baudrate = 1800;
        break;
    case 3:
        baudrate = 2400;
        break;
    case 4:
        baudrate = 4800;
        break;
    case 5:
        baudrate = 9600;
        break;
    case 6:
        baudrate = 19200;
        break;
    case 7:
        baudrate = 38400;
        break;
    default:
        break;
    }
    return baudrate;
}
```