
Redes de Computadores

# Protocolo de Ligação de Dados

(1º Trabalho Laboratorial)

Mestrado Integrado em Engenharia Informática e Computação

8 de novembro de 2020

Autores:

Ana Teresa Feliciano da Cruz | up201806460@fe.up.pt

André Filipe Meireles do Nascimento | up201806461@fe.up.pt

# Índice

# Sumário

Este relatório foi elaborado no âmbito da cadeira de Redes de Computadores, com o objetivo de complementar o primeiro trabalho prático. Este trabalho consiste no desenvolvimento de uma aplicação capaz de transferir ficheiros de um computador para o outro, através de uma porta série.

O trabalho foi concluído com sucesso, sendo que a transferência de dados foi concretizada sem perdas ou erros.

# Introdução

O objetivo do trabalho é a implementação de um protocolo de ligação de dados e testá-lo com uma aplicação de transferência de ficheiros de um computador para outro utilizando a porta série. Quanto ao relatório, o seu objetivo é explorar a parte teórica inerente ao trabalho, cumprindo a seguinte estrutura:

- **Arquitetura**
  Apresentação dos blocos funcionais e interfaces presentes.
- **Estrutura do código**
  Demonstração das APIs, principais estruturas de dados, principais funções e sua relação com a arquitetura.
- **Casos de uso principais**
  Identificação dos mesmos e demonstração das sequências de chamada de funções.
- **Protocolo de ligação lógica**
  Identificação dos principais aspetos funcionais e descrição da estratégia de implementação dos mesmos com apresentação de extratos de código.
- **Protocolo de aplicação**
  Identificação dos principais aspetos funcionais e descrição da estratégia de implementação dos mesmos com apresentação de extratos de código.
- **Validação**
  Descrição dos testes efetuados com apresentação quantificada dos resultados.
- **Eficiência do protocolo de ligação de dados**
  Caraterização estatística da eficiência do protocolo, feita com recurso a medidas sobre o código desenvolvido.
- **Conclusão**
  Síntese da informação apresentada nas secções anteriores e reflexão sobre os objetivos de aprendizagem alcançados.

# Arquitetura e Estrutura de código

O trabalho está dividido em dois blocos funcionais a camada de ligação de dados e a de aplicação, independentes entre si.

## *Camada de ligação de dados*

A camada de ligação de dados é a camada lógica de mais baixo nível da aplicação, funcionando como ponte entre a porta de série e a camada de aplicação. É responsável pela abertura e fecho da porta série, bem como a leitura e escrita de dados nesta. Para além disso, é também responsável pelo sincronismo, envio e receção de tramas e o controlo de erros.

Para facilitar o uso de variáveis necessárias a esta camada utilizou-se a *struct* ***linkLayer***, e a *struct* ***statistics*** para guardar valores conclusivos do programa, ambas implementadas em *ll.h*. sendo a *linklayer* uma variável global de *ll.c*.

### Principais Funções (*ll.c*)

- *llopen*: responsável pela inicialização das estruturas de dados e estabelecimento da ligação
- *llclose*: responsável pela terminação da ligação
- *llread*: responsável pela receção e leitura da trama de informação, *destuffing* desta, verificação de erros e enviar uma mensagem de confirmação concordante
- *llwrite*: responsável pelo *stuffing* da trama de informação, enviá-la e a leitura da mensagem de confirmação do recetor

### Funções Auxiliares (*messages.c*)

- *readCommand*: responsável pela leitura de tramas SET e DISC
- *readResponse*: responsável pela leitura de tramas UA
- *readAck*: responsável pela leitura de tramas RR e REJ
- *readFrameI*: responsável pela leitura de tramas I
- *processFrameSU*: auxiliar às funções *readCommand*, *readResponse* e *readAck*, é responsável pela verificação dos *bytes* das tramas de supervisão/não numeradas recebidas através de uma máquina de estados
- *processFrameI*: auxiliar à função *readFrameI*, é responsável pela verificação dos *bytes* das tramas de informação recebidas através de uma máquina de estados
- *writeStuffedFrame*: responsável pela formação de uma trama I, incluindo o seu *stuffing* e o seu envio
- *destuffFrame*: responsável pelo *destuffing* de uma trama I recebida

### Macros pertinentes (*ll.h e msg_macros.h*)

- BAUDRATE: velocidade da transmissão
- MAX_TRANSMISSIONS: número máximo de retransmissões possíveis em caso de falha
- TIMEOUT: temporizador
- C_SET: campo de controlo de uma trama SET (*set up*)
- C_DISC: campo de controlo de uma trama DISC (*disconnect*)
- C_UA: campo de controlo de uma trama UA (*unnumbered acknowledgment*)
- C_RR0 e C_RR1: campo de controlo de uma trama RR (*receiver ready/positive ACK*)
- C_REJ0 e C_REJ1: campo de controlo de uma trama REJ (*reject /negative ACK*)
- C_I0 e C_I1: campo de controlo de uma trama I

## *Camada de aplicação*

A camada da aplicação é a camada lógica situada diretamente acima da camada de ligação de dados. É responsável, recorrendo à interface da camada de ligação de dados, pelo envio e receção de ficheiros. Ademais é também responsável pelo tratamento de cabeçalhos, distinção entre pacotes de controlo e de dados e a numeração destes.

Para facilitar o uso de variáveis necessárias a esta camada utilizou-se a *struct* ***applicationLayer***, implementada em *application.h*, que se trata de uma variável global em *application.c*.

### Principais funções (*application.c*)

- *sendFile*: responsável pela abertura e leitura do ficheiro de envio, e envio dos pacotes de controlo e de dados
- *receiveFile*: responsável leitura dos pacotes de controlo e de dados recebidos, e escrita no ficheiro de destino

### Macros pertinentes (*application.h*)

- C_DATA: campo de controlo do pacote de dados
- C_START: campo do controlo do pacote de controlo START
- C_END: campo de controlo do pacote de controlo END
- T_FILE_SIZE: parâmetro correspondente ao tamanho do ficheiro
- T_FILE_NAME: parâmetro correspondente ao nome do ficheiro
- DATA_PACKET_SIZE: tamanho do pacote de dados
- CONTROL_PACKET_SIZE: tamanho do pacote de controlo

## Casos de usos principais

Os casos de uso principais são a interface que permite ao utilizador a introdução dos argumentos do programa e a transferência do ficheiro através da porta série entre dois computadores.

Para inicializar a transferência o utilizador tem de, recorrendo ao terminal, fazer *make* na *root* da pasta do projeto e introduzir alguns argumentos. Sendo o recetor tem de introduzir a *flag 'receiver'*, o local de destino do ficheiro (ex: ./) e o número da porta série (ex: 11), o transmissor tem de introduzir a *flag 'transmitter'*, o nome do ficheiro a enviar (ex: pinguim.gif) e o número da porta série (ex: 10).

Exemplo recetor: ./application receiver ./ 11

Exemplo transmissor: ./application transmitter pinguim.gif 10

| **Transmissor** | **Recetor** |
|---|---|
| Introdução dos argumentos no terminal | Introdução dos argumentos no terminal |
| Abertura da ligação entre os computadores | Abertura da ligação entre os computadores |
| Envio do pacote de controlo START | Receção do pacote de controlo START |
| Envio dos pacotes de dados | Receção dos pacotes de dados |
| Envio do pacote de controlo END | Receção do pacote de controlo END |
| Impressão das estatísticas | Impressão das estatísticas |
| Fecho da ligação entre os computadores | Fecho da ligação entre os computadores |

## Protocolo de ligação lógica

O protocolo de ligação lógica desenvolvido tem como principais aspetos funcionais: configuração da porta série; estabelecimento da ligação com a porta série; transferência de dados através da porta série, fazendo o *stuffing* e *destuffing* dos mesmos; recuperação de erros durante a transferência de dados.

## *llopen*

Esta função é necessária para inicializar a ligação com a porta série. Para isso, começa por alterar as configurações da porta série para as pretendidas, recorrendo à função *openSerial*.

O Transmissor após inicializar a *linkLayer (initDataLinkLayer)*, envia uma trama SET (*sendSet*) e ativa o alarme, que é desativado pela receção de uma trama UA (recebida pela função *readResponse*), enviada pelo recetor. Se não receber resposta dentro do tempo TIMEOUT estipulado, o SET é reenviado até um número máximo de MAX_TRANSMISSIONS, terminando o programa se este número for excedido.

O Recetor espera pela chegada de uma trama SET (recebida pela função *readCommand*), respondendo com uma trama UA (*sendUA*), estabelecendo-se assim a ligação com sucesso.

Estas tramas de supervisão (S) e não numeradas (U) são constituídas por uma *flag* (F) no início e no fim da trama, campo de endereço (A), campo de controlo (C) e campo de proteção (BCC1).

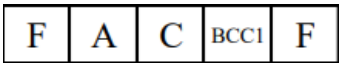


*Figura 1: formato de uma trama S/U*

## *llclose*

Esta função é necessária para terminar a ligação com a porta série.

O Transmissor tenta terminar a ligação enviando uma trama DISC (*sendDISC*), ficando à espera de uma resposta, trama DISC (recebida pela função *readResponse*), enviada pelo Recetor. Ao receber a trama, responde com uma trama UA (sendUA_last), informando o Recetor que sabe que este quer finalizar a ligação.

O Recetor espera até receber uma trama DISC (recebida pela função *readCommand*), respondendo com uma trama do mesmo tipo (*sendDISC*). Após, fica à espera da resposta, trama UA (recebida pela função *readCommand*).

No final, as configurações da porta séria são repostas, terminado a ligação com esta, recorrendo à função *closeSerial*.

## *llwrite*

Esta função é necessária para a escrita dos dados no decorrer da aplicação.

A função recebe um pacote de dados e envia-o pela porta série depois de o encapsular numa trama I. Para tal, recorre à função *writeStuffedFrame* que constrói o cabeçalho da trama, faz o cálculo do BCC2, o *stuffing* dos dados e envia a trama pela porta série, ativando o alarme.


*Figura 2: formato de uma trama I*

Após, fica à espera de uma resposta do Recetor, desativando o alarme quando a resposta recebida for uma trama RR processada pela função *readAck*. Se a resposta não for recebida ou for recebido uma trama REJ, a trama I é reenviada até um máximo de MAX_TRANSMISSIONS, terminando o programa se este número for excedido.

## *llread*

Esta função é necessária para a leitura dos dados no decorrer da aplicação.

A função fica à espera até receber uma trama I, e a sua leitura é feita recorrendo a *readFrameI*. Posteriormente, é feito o *destuffing* recorrendo a *destuffFrame*, e é calculado e verificado o BCC2. No caso de não corresponder ao BCC2 recebido, é mandado uma trama REJ de resposta com o número de sequência correspondente. Caso contrário, é mandado uma trama RR como o número de sequência alternado, demonstrando que está pronto para receber uma nova trama.

# Protocolo de aplicação

O protocolo de aplicação desenvolvido tem como principais aspetos funcionais: criação e transferência dos pacotes de controlo e de dados; leitura e escrita do ficheiro a transferir.

## *sendFile*

Esta função é a responsável pelo comportamento principal do Transmissor, enviando os dados para a camada inferior para que esta os envie pela porta série.

A função começa por abrir o ficheiro em modo de leitura, para posteriormente ser lida a sua informação. De seguida, é criado e enviado o pacote de controlo START recorrendo à função *sendControlPacket*, servindo-se da *llwrite* para codificar e enviar este pacote. Os pacotes de controlo têm um campo de controlo (C), START ou END, e contêm a informação codificada em TLVs (*Type*, *Length*, *Value*), ou seja, para cada para parâmetro do pacote, é necessário passar o tipo (T), o tamanho em octetos (L) e só depois o seu valor (V). Na aplicação desenvolvida, os pacotes de controlo contêm o tamanho e o nome do ficheiro.

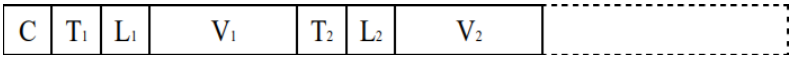


*Figura 3: formato de um pacote de controlo*

Após isto, e através da função *sendDataPacket*, é lido o ficheiro que se quer transferir e, no ciclo de leitura são construídos os pacotes de dados. Dos *bytes* lidos, é construído o pacote, atualizando-se o campo de controlo (C), número de sequência (N), número de octetos do campo de dados (L2 e L1) e o campo de dados do pacote (P1 a Pk). Por fim, são enviados através de *llwrite*.

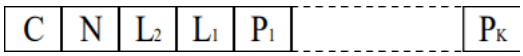


*Figura 4: formato de um pacote de dados*

Finalmente, é enviado o pacote de controlo END, novamente recorrendo à função *sendControlPacket*, terminado assim o envio de dados por parte do Transmissor.

## *receiveFile*

Esta função é a responsável pelo comportamento principal do Recetor, recebendo os dados da camada inferior para construir o ficheiro lido.

No ciclo de leitura, através da função *llread*, são lidos os pacotes. No caso de ser o pacote de controlo START é chamada a função *readControlPacket* que vai permitir obter o tamanho e o nome do ficheiro, abrindo este para escrita. No caso de ser um pacote de dados é chamada a função *readDataPacket* que vai escrever para o ficheiro os dados. Por fim, no caso de ser o pacote de controlo END o ciclo de leitura é terminado, fechando o descritor do ficheiro.

# Validação

Para avaliar a aplicação desenvolvida foram feitos os seguintes testes, todos concluídos com sucesso:

- Enviar ficheiros de diferentes tamanhos;
- Enviar um ficheiro, interromper a ligação durante a transmissão por alguns segundos, restabelecendo a ligação depois;
- Variação do tamanho das tramas;
- Variação da capacidade de ligação;
- Variação do tempo de propagação;
- Variação do FER (*frame error ratio*).

# Eficiência do protocolo de ligação de dados

De forma a avaliar a eficiência do protocolo desenvolvido, foram efetuados os quatro testes apresentados a seguir e elaborados uma tabela e um gráfico, que podem ser encontrados no Anexo II. Para a realização dos testes foi utilizada a imagem pinguins.jpg (com 71kB), que permitiu obter melhores conclusões que o pinguim.gif. Os testes foram feitos através de acesso remoto aos computadores do laboratório.

## *Variação do tamanho das tramas I*

Através dos testes pode-se observar que quanto maior o tamanho da trama I, maior é a eficiência. Tal acontece porque é mandada mais informação de cada vez, sendo o número de tramas enviadas menor o que, consequentemente, faz com que o programa execute de forma mais rápida e eficaz.

## *Variação da capacidade de ligação*

Através dos testes conclui-se que o aumento da capacidade de ligação não traz diferenças notórias à eficiência do programa. No entanto, quanto menor a capacidade de ligação maior é o tempo total de transferência de dados.

## *Variação do tempo de propagação*

Através dos testes conclui-se que com um maior atraso de propagação, a eficiência do programa diminui muito. Isto deve-se ao facto de ser introduzido um atraso no processamento de cada trama recebida e, consequentemente, na resposta enviada.

## *Variação do FER*

Através dos testes sobre a variação da probabilidade de erros no BCC1 conclui-se que, quanto maior a probabilidade, menos eficaz é o programa. Isto deve-se ao facto de, quando gerado um erro, o recetor não manda uma resposta, ficando o transmissor à espera um número previamente escolhido de segundos, atrasando a execução do programa.

Através dos testes sobre a variação da probabilidade de erros no BCC2 conclui-se que não são notórias as diferenças de eficiência, uma vez que tais erros só causam o reenvio da trama, o que é imediato.

# Conclusão

O tema deste trabalho laboratorial é um protocolo de ligação de dados que permite a comunicação entre dois computadores através da porta série.

Neste relatório foi abordado o envio e receção de tramas, o seu *stuffing* e *destuffing* e tratamento de erros, por parte de camada de ligação de dados. Na camada de aplicação foi abordado o envio e receção de pacotes de controlo e de dados, permitindo assim o envio de um ficheiro através da porta série.

O desenvolvimento deste trabalho contribuiu para um aprofundamento dos conhecimentos teóricos e práticos deste tema. No entanto, as atuais circunstâncias trouxeram um acréscimo na dificuldade da evolução do trabalho e da sua testagem.

Em suma, o trabalho foi realizado com sucesso, tendo-se cumprido os requisitos pedidos.

# Anexo I – Código fonte

- **application.h**

```
#pragma once
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "ll.h"

#define C_DATA  0x01
#define C_START 0x02
#define C_END   0x03

#define T_FILE_SIZE 0x00
#define T_FILE_NAME 0x01

#define DATA_PACKET_SIZE 4
#define CONTROL_PACKET_SIZE 5

typedef struct {
   int serial_fd;
   int sent_file_fd;
   int rec_file_fd;
   char sentFileName[256];
            char recFileName[256];
            off_t sentFileSize;
            off_t recFileSize;
} applicationLayer;

int sendFile(int fd, char *file_name);
int sendControlPacket(unsigned char control_field);
int sendDataPacket();
int receiveFile(int fd, char *dest);
int readControlPacket(unsigned char *packet);
int readDataPacket(unsigned char *packet);
```

- **application.c**

```
#include "application.h"

applicationLayer applayer;
struct timespec start, end;

int sendFile(int fd, char *file_name){
	applayer.sent_file_fd = open(file_name, O_RDONLY);
	if(applayer.sent_file_fd < 0){
		printf("Error opening file\n");
		return -1;
	}

	struct stat info;
	if(fstat(applayer.sent_file_fd, &info) == -1){
		printf("Error in fstat\n");
		return -1;
	}

	applayer.serial_fd = fd;
	applayer.sentFileSize = info.st_size;
	strcpy(applayer.sentFileName, file_name);

	// Send Start Control Packet
	if(sendControlPacket(C_START) == -1){
		printf("Error sending Start Control Packet\n");
		return -1;
	}

	if(clock_gettime(CLOCK_MONOTONIC, &start) < 0) {
		printf("Error in start clock_gettime\n");
		return -1;
	}

	// Send Data Packets
	if (sendDataPacket() == -1) {
		printf("Error sending Data Packets\n");
		return -1;
	}

	if(clock_gettime(CLOCK_MONOTONIC, &end) < 0) {
		printf("Error in end clock_gettime\n");
		exit(-1);
	}
	printf("Time sending data packets: %f seconds\n\n", (double)(end.tv_sec -
start.tv_sec) + ((double)(end.tv_nsec - start.tv_nsec)/1000000000.0));

	// Send End Control Packet
	if(sendControlPacket(C_END) == -1){
		printf("Error sending End Control Packet\n");
```

```
                return -1;
        }

        return 0;
}

int sendControlPacket(unsigned char control_field){
  int file_length = sizeof(applayer.sentFileSize);
        int packet_size = CONTROL_PACKET_SIZE + file_length +
strlen(applayer.sentFileName);
  unsigned char packet[packet_size];

  packet[0] = control_field;

  //Insert file size
  packet[1] = T_FILE_SIZE; //type
  packet[2] = file_length; //length
        memcpy(&packet[3], &applayer.sentFileSize, file_length); //file size

  //Insert file name
  packet[file_length+3] = T_FILE_NAME; //type
  packet[file_length+4] = strlen(applayer.sentFileName); //length
        memcpy(&packet[file_length+5], applayer.sentFileName,
strlen(applayer.sentFileName)); //file name

  if(llwrite(applayer.serial_fd, packet, packet_size) == -1){
    printf("Error llwrite control packet\n");
    return -1;
  }
  else{
                printf("\n***Sent Control Packet successfully***\n\n");
        }

  return 0;
}

int sendDataPacket(){
        char buf[MAX_SIZE];
        int numbytes = 0, sequenceNumber = 0;

        while((numbytes = read(applayer.sent_file_fd, &buf, MAX_SIZE)) != 0){
                unsigned char packet[DATA_PACKET_SIZE + numbytes];

                //build data packet
                packet[0] = C_DATA;
                packet[1] = sequenceNumber % 255;
                packet[2] = numbytes / 256;
                packet[3] = numbytes % 256;
                memcpy(&packet[4], &buf, numbytes);

                if(llwrite(applayer.serial_fd, packet, numbytes + DATA_PACKET_SIZE) == -
1){
```

```
                printf("Error sending the data packet\n");
                return -1;
                }
                else{
                                printf("\n***Sent DATA Packet successfully***\n\n");
                        }

                        sequenceNumber++;

                }

                return 0;
}


int receiveFile(int fd, char *dest){
                unsigned char packet[MAX_SIZE + DATA_PACKET_SIZE];

                applayer.serial_fd = fd;
                strcpy(applayer.recFileName, dest);

                while(1){
                        if(llread(fd,packet) <= 0){
                                printf("Error reading the packet\n");
                continue;
                        }
                        if(packet[0] == C_END){
                                printf("\n***Received END Control Packet
successfully***\n\n");
                                break;
                        }
                        else if(packet[0] == C_START){
                                if (readControlPacket(packet) > 0) printf("\n***Received START
Control Packet successfully***\n\n");
                                else return -1;
                        }
                        else if(packet[0] == C_DATA){
                                if (readDataPacket(packet) != -1) printf("\n***Received DATA
Packet successfully***\n\n");
                        }
                }

                return close(applayer.rec_file_fd);;
}

int readControlPacket(unsigned char *packet){
                int index = 0;
                int size_length;
                off_t file_size = 0;
                char* file_name;

                //FILE SIZE
```

```
	if(packet[1] == T_FILE_SIZE){
		size_length = packet[2];

		for (int i=3, j=0; i < size_length+3; i++, j++){
			file_size += packet[i] << 8 * j;
		}

		printf("File size: %ld bytes\n", file_size);
	}

	if (file_size <= 0) {
		perror("File size error\n");
		return -1;
	}

	applayer.recFileSize = file_size;

	index = size_length+3;

	//FILE NAME
	if (packet[index] == T_FILE_NAME) {
		index++;
		int name_length = packet[index];
		index++;

		file_name = malloc(sizeof(char) * (name_length + 1));
		for (int i = 0; i < name_length; i++) {
			file_name[i] = packet[index];
			index++;
		}
		file_name[name_length] = '\0';

		printf("File Name: %s\n", file_name);
	}

	strcat(applayer.recFileName, file_name);

	applayer.rec_file_fd = open(applayer.recFileName, O_RDWR | O_CREAT, 0777);
	if(applayer.rec_file_fd < 0){
		printf("Error opening file\n");
		return -1;
	}

	return applayer.rec_file_fd;
}

int readDataPacket(unsigned char *packet){
	int dataSize = 256 * packet[2] + packet[3];

	if(write(applayer.rec_file_fd, &packet[4], dataSize) == -1){
		printf("Error writting data packet to file\n");
		return -1;
```

```
        }
        else
                printf("Wrote data packet to file\n");

        return 0;
}

int main(int argc, char** argv) {
        // Parse Args
  if(argc != 4){
     printf("Usage: ./application <receiver/transmitter> <destination/filename>
<port={0,1,10,11}>\n");
        exit(1);
  }

  int port = atoi(argv[3]);
        if (port!=0 && port!=1 && port!=10 && port!=11){
                printf("Port number must be one of {0, 1, 10, 11}\n");
                exit(1);
        }

        int flag;
        if (strcmp("receiver",argv[1])==0){
                flag = RECEIVER;
        }
        else if(strcmp("transmitter",argv[1])==0){
                flag = TRANSMITTER;
        }
        else  {
                printf("Flag must be either 'receiver' or 'transmitter'\n");
                exit(1);
        }

  int fd = llopen(port, flag);
        if(fd == -1){
                printf("llopen error\n");
                return -1;
        }

        // Init application
        if(flag==RECEIVER){
                if(receiveFile(fd, argv[2]) == -1){
                        printf("Error receiving file\n");
                        return -1;
                }
        }
        else if(flag==TRANSMITTER){
                if(sendFile(fd, argv[2]) == -1){
                        printf("Error sending file\n");
                        return -1;
                }
        }
```

```
        else {
                printf("Flag error\n");
                exit(1);
        }

        if(llclose(fd) == -1){
                printf("llclose error\n");
                return -1;
        }

        displayStats();

        return 0;
}
```

- **ll.h**

```
#pragma once
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "messages.h"
#include "alarm.h"

#define BAUDRATE B38400
#define MODEMDEVICE "/dev/ttyS1"
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define TRANSMITTER 0
#define RECEIVER 1

#define TIMEOUT 3
#define MAX_TRANSMISSIONS 3

typedef struct {
   unsigned int numSentFramesI;
   unsigned int numReceivedFramesI;
   unsigned int numTimeouts;
   unsigned int numSentRR;
   unsigned int numReceivedRR;
   unsigned int numSentREJ;
   unsigned int numReceivedREJ;
   struct timespec start, end;
} statistics;

typedef struct {
   char port[20]; // /dev/ttySx
   int flag; //TRANSMITTER/RECEIVER
   unsigned int sequenceNumber; //trama sequence
   unsigned int timeout;
   unsigned int numTransmissions; //atempt number in case of failure
   unsigned int alarm;
   statistics stats;
} linkLayer;

extern linkLayer linklayer;

statistics initStatistics();
void initDataLinkLayer(int port, int flag);

//ll functions
int llopen(int port, int flag);
```

```
int llclose(int fd);
int llwrite(int fd, unsigned char* buffer, int length);
int llread(int fd, unsigned char *buffer);

int openSerial();
int closeSerial(int fd);

void displayStats();
```

- **ll.c**

```
#include "ll.h"

struct termios oldtio, newtio;
linkLayer linklayer;

statistics initStatistics() {
   statistics stats;
   stats.numSentFramesI = 0;
   stats.numReceivedFramesI = 0;
   stats.numSentRR = 0;
   stats.numReceivedRR = 0;
   stats.numSentREJ = 0;
   stats.numReceivedREJ = 0;
   if(clock_gettime(CLOCK_MONOTONIC, &stats.start) < 0) {
                        printf("Error in start clock_gettime\n");
                        exit(-1);
   }

   return stats;
}

void initDataLinkLayer(int port, int flag){
   char porta[12];
   snprintf(porta, 12, "/dev/ttyS%d", port);
   strcpy(linklayer.port, porta);
   linklayer.flag = flag;
   linklayer.numTransmissions = 0;
   linklayer.alarm = 0;
   linklayer.timeout = TIMEOUT;
   linklayer.sequenceNumber = 0;
   linklayer.stats = initStatistics();
}

int openSerial(){
   /*
            Open serial port device for reading and writing and not as controlling tty
            because we don't want to get killed if linenoise sends CTRL-C.
   */
   int fd = open(linklayer.port, O_RDWR | O_NOCTTY );
            if (fd <0) {perror(linklayer.port); exit(-1); }

            if ( tcgetattr(fd,&oldtio) == -1) { /* save current port settings */
                        perror("tcgetattr");
                        exit(-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] = 0;   /* inter-character timer unused */
            newtio.c_cc[VMIN] = 1;   /* blocking read until 1 chars received */

 /*
            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) {
                        perror("tcsetattr");
                        exit(-1);
            }

            printf("New termios structure set\n");
   printf("\n***Started Connection***\n\n");
   return fd;
}

int closeSerial(int fd){
   tcflush(fd, TCIOFLUSH);

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

   printf("\n***Closed Connection***\n\n");

   return close(fd);
}

int llopen(int port, int flag){
   int fd;

   initDataLinkLayer(port, flag);
   fd = openSerial();

   setAlarm();

   if(linklayer.flag == TRANSMITTER){
      do {
         if (sendSET(fd) == -1){
            printf("Error sending SET\n");
         }
         else {
            printf("Sent SET\n");
```

```
        }

        startAlarm();

        if(readResponse(fd) == -1){
            printf("Error receiving UA\n");
        }
        else {
            printf("Received UA\n");
        }

    } while (linklayer.numTransmissions < MAX_TRANSMISSIONS && linklayer.alarm);


    stopAlarm();
    if(linklayer.numTransmissions >= MAX_TRANSMISSIONS){
        printf("Reached max retries\n");
        return -1;
    }
    linklayer.numTransmissions=0;
  }

  else if(linklayer.flag == RECEIVER){
              if (readCommand(fd) == -1 ){
        printf("Error receiving SET\n");
        return -1;
    }
    else {
        printf("Received SET\n");
    }


    if (sendUA(fd) == -1){
        printf("Error sending UA\n");
    }
    else {
        printf("Sent UA\n");
    }

  }

  printf("\n***Established Connection***\n\n");

  return fd;
}

int llclose(int fd){
  if(linklayer.flag == RECEIVER){
                        //read DISC frame
    if(readCommand(fd) == -1){
        printf("Error reading DISC frame\n");
        return -1;
```

```
    }
    else
    {
       printf("Received DISC\n");
    }
    //send DISC frame
    if (sendDISC(fd) == -1){
       printf("Error sending DISC\n");
    }
    else {
       printf("Sent DISC\n");
    }
    //read UA
    if(readCommand(fd) == -1){
       printf("Error reading UA\n");
       return -1;
    }
                        else
                        {
                                printf("Received UA\n");
                        }

  }

  else if(linklayer.flag == TRANSMITTER){
    do{
                                //send DISC
                                if (sendDISC(fd) == -1){
          printf("Error sending DISC\n");
           }
           else {
          printf("Sent DISC\n");
           }

                                startAlarm();

                                //read DISC
                                if(readResponse(fd) == -1){
                        printf("Error reading DISC frame\n");
                        }
                        else{
                        printf("Received DISC\n");
                        }
                        }while (linklayer.numTransmissions < MAX_TRANSMISSIONS &&
linklayer.alarm);

                        stopAlarm();
    if(linklayer.numTransmissions >= MAX_TRANSMISSIONS){
       printf("Reached max retries\n");
       return -1;
    }
    linklayer.numTransmissions=0;
```

```
        if (sendUA_last(fd) == -1){
            printf("Error sending UA\n");
        }
        else {
            printf("Sent UA\n");
        }
    }

    printf("\n***Terminated Connection***\n\n");

    if(clock_gettime(CLOCK_MONOTONIC, &linklayer.stats.end) < 0) {
                            printf("Error in end clock_gettime\n");
                            exit(-1);
    }

    unsetAlarm();
    sleep(1);
                return closeSerial(fd);
}

int llwrite(int fd, unsigned char* buffer, int length){
    do{
        // Send frame
        if (writeStuffedFrame(fd, buffer, length)) printf("Sent Frame\n");
        startAlarm();
        // Read receiver ACK
        if(readAck(fd) == -1){
            stopAlarm();
            linklayer.alarm = 1;
            continue;
        }
    }while(linklayer.numTransmissions < MAX_TRANSMISSIONS && linklayer.alarm);

    stoptAlarm();
    if(linklayer.numTransmissions >= MAX_TRANSMISSIONS){
        printf("Reached max retries\n");
        return -1;
    }
    linklayer.numTransmissions=0;

    return length;
}

int llread(int fd, unsigned char *buffer) {
    int received = 0;
    int frame_length = 0;
    int dframe_length = 0;
    int packet_length = 0;
    unsigned char frame[2*MAX_SIZE+6];
    unsigned char dframe[MAX_SIZE];
    unsigned char control;
```

```
while(!received){
  if((frame_length = readFrameI(fd, frame))){
    printf("Received Frame\n");
    control = frame[2];

    // Destuff received frame
    dframe_length = destuffFrame(frame, frame_length, dframe);

    // Calculate BCC2 after destuffing
    unsigned char bcc2 = dframe[4];
    for(int i=5; i<dframe_length-7+5; i++){
      bcc2 ^= dframe[i];
    }

    // Verify if calculted BCC2 matches with received BCC2
    if(bcc2!=dframe[dframe_length-2]){
      printf("BCC2 Error\n");

      // If not, send REJ
      if(control == C_I0){
        if(sendREJ0(fd) == -1){
          printf("Error sending REJ0\n");
        }
        else {
          linklayer.stats.numSentREJ++;
          printf("Sent REJ0\n");
        }

      }
      else if(control == C_I1){
        if(sendREJ1(fd) == -1){
          printf("Error sending REJ1\n");
        }
        else {
          linklayer.stats.numSentREJ++;
          printf("Sent REJ1\n");
        }
      }

      return -1;
    }

    else {
      // Fills arg buffer with Data from destuffed frame
      for(int i=4; i< dframe_length - 2; i++){
        buffer[packet_length++] = dframe[i];
      }

      // Send correct RR
      if(control == C_I0) {
        if(sendRR1(fd) == -1){
```

```
                printf("Error sending RR1\n");
            }
            else {
                linklayer.stats.numSentRR++;
                printf("Sent RR1\n");
            }
        }
        else if(control == C_I1) {
            if(sendRR0(fd) == -1){
                printf("Error sending RR0\n");
            }
            else {
                linklayer.stats.numSentRR++;
                printf("Sent RR0\n");
            }
        }

        received = 1;
      }

    }

  }

  // Change sequence number (0 / 1)
  linklayer.sequenceNumber^=0x01;

  return packet_length;
}

void displayStats() {
            printf("\n\n***Statistics:***\n\n");
            printf("Total execution time: %f seconds\n",  (double)(linklayer.stats.end.tv_sec -
linklayer.stats.start.tv_sec) + ((double)(linklayer.stats.end.tv_nsec -
linklayer.stats.start.tv_nsec)/1000000000.0));
            printf("Number of sent Frames I: %d\n", linklayer.stats.numSentFramesI);
            printf("Number of received Frames I: %d\n", linklayer.stats.numReceivedFramesI);
            printf("Number of timeouts: %d\n", linklayer.stats.numTimeouts);
            printf("Number of sent Frames RR: %d\n", linklayer.stats.numSentRR);
            printf("Number of received Frames RR: %d\n", linklayer.stats.numReceivedRR);
            printf("Number of sent Frames REJ: %d\n", linklayer.stats.numSentREJ);
            printf("Number of received Frames REJ: %d\n", linklayer.stats.numReceivedREJ);
            printf("\n");
}
```

- **messages.h**

```
#pragma once
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "msg_macros.h"
#include "ll.h"

enum states {START, FLAG_RCV, A_RCV, C_RCV, BCC_OK, DATA, STOP};

int sendSET(int fd);
int sendUA(int fd);
int sendUA_last(int fd);
int sendDISC(int fd);
int sendRR0(int fd);
int sendRR1(int fd);
int sendREJ0(int fd);
int sendREJ1(int fd);

unsigned char processFrameSU(enum states *state,  unsigned char byte);
void processFrameI(enum states *state,  unsigned char byte);

int readCommand(int fd);
int readResponse(int fd);
int readFrameI(int fd, unsigned char *frame);
int readAck(int fd);

int writeStuffedFrame(int fd, unsigned char *buffer, int length);
int destuffFrame(unsigned char* frame, int length, unsigned char* destuffed_frame);

void generateBCC1Error(unsigned char *c, int percentage);
void generateBCC2Error(unsigned char *frame, int percentage);
```

- **messages.c**

```
#include "messages.h"

unsigned char set[5] = {FLAG, A_ER, C_SET, BCC(A_ER, C_SET), FLAG};
unsigned char ua[5] = {FLAG, A_ER, C_UA, BCC(A_ER, C_UA), FLAG};
unsigned char ua_last[5] = {FLAG, A_RE, C_UA, BCC(A_RE, C_UA), FLAG};
unsigned char disc[5] = {FLAG, A_ER, C_DISC, BCC(A_ER, C_DISC), FLAG};
unsigned char rr0[5] = {FLAG, A_ER, C_RR0, BCC(A_ER, C_RR0), FLAG};
unsigned char rr1[5] = {FLAG, A_ER, C_RR1, BCC(A_ER, C_RR1), FLAG};
unsigned char rej0[5] = {FLAG, A_ER, C_REJ0, BCC(A_ER, C_REJ0), FLAG};
unsigned char rej1[5] = {FLAG, A_ER, C_REJ1, BCC(A_ER, C_REJ1), FLAG};

int sendSET(int fd){
   return write(fd, set, 5);
}

int sendUA(int fd){
   return write(fd, ua, 5);
}

int sendUA_last(int fd){
   return write(fd, ua_last, 5);
}

int sendDISC(int fd){
   return write(fd, disc, 5);
}

int sendRR0(int fd){
   return write(fd, rr0, 5);
}

int sendRR1(int fd){
   return write(fd, rr1, 5);
}

int sendREJ0(int fd){
   return write(fd, rej0, 5);
}

int sendREJ1(int fd){
   return write(fd, rej1, 5);
}

unsigned char processFrameSU(enum states *state, unsigned char byte){
   static unsigned char a = 0;
   static unsigned char c = 0;
   switch (*state) {
      case START:
         if (byte == FLAG) {
            *state = FLAG_RCV;
```

```
    }
    break;

case FLAG_RCV:
    if (byte == A_ER || byte == A_RE) {
        *state = A_RCV;
        a = byte;
    }
    else if (byte!= FLAG){
        *state = START;
    }
    break;

case A_RCV:
    if (VERIFY_C(byte)) {
        *state = C_RCV;
        c = byte;
    }
    else if (byte == FLAG){
        *state = FLAG_RCV;
    }
    else{
        *state = START;
    }
    break;

case C_RCV:
    if (byte == BCC(a,c)){
        *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;

case STOP:
    break;

default:
    break;
```

```
    }
    return c;
}

void processFrameI(enum states *state, unsigned char byte){
    static unsigned char c = 0;
    switch (*state) {
        case START:
            if (byte == FLAG) {
                *state = FLAG_RCV;
            }
            break;

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

        case A_RCV:
            if ((byte==C_I0 && linklayer.sequenceNumber==0) || (byte==C_I1 &&
linklayer.sequenceNumber==1)) {
                *state = C_RCV;
                c = byte;
                //generateBCC1Error(&c, 5);
            }
            else if (byte == FLAG){
                *state = FLAG_RCV;
            }
            else{
                *state = START;
            }
            break;

        case C_RCV:
            if (byte == BCC(A_ER,c)){
                *state = BCC_OK;
            }
            else if (byte == FLAG){
                *state = FLAG_RCV;
            }
            else{
                *state = START;
            }
            break;

        case BCC_OK:
            if (byte != FLAG){
                *state = DATA;
```

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

        default:
            break;
    }
}

int readCommand(int fd){
    unsigned char byte;
    enum states state = START;

    while (state != STOP) {      /* loop for input */
        read(fd,&byte,1);
        processFrameSU(&state, byte);
    }

    return 0;
}

int readResponse(int fd){
    unsigned char byte;
    enum states state = START;

    while(state!=STOP && !linklayer.alarm) {
        read(fd, &byte, 1);
        processFrameSU(&state, byte);
    }

    if(linklayer.alarm) return -1;

    return 0;
}

int readFrameI(int fd, unsigned char *frame){
    unsigned char byte;
    int length = 0;
    enum states state = START;

    while (state != STOP) {      /* loop for input */
        read(fd,&byte,1);

        processFrameI(&state, byte);
```

```
        if(state == FLAG_RCV && length !=0) length = 0;

        frame[length++] = byte;
    }

    //generateBCC2Error(frame, 5);

    linklayer.stats.numReceivedFramesI++;
    return length;
}

int readAck(int fd){
    unsigned char byte;
    enum states state = START;
    unsigned char control_field;

    while(state!=STOP && !linklayer.alarm) {
        read(fd, &byte, 1);
        control_field = processFrameSU(&state, byte);
    }

    if(linklayer.sequenceNumber == 0 && control_field == C_RR1){
        printf("Received RR1\n");
        linklayer.sequenceNumber = 1;
        linklayer.stats.numReceivedRR++;
        return 0;
    }

    else if(linklayer.sequenceNumber == 1 && control_field == C_RR0){
        printf("Received RR0\n");
        linklayer.sequenceNumber = 0;
        linklayer.stats.numReceivedRR++;
        return 0;
    }

    else if(control_field == C_REJ0){
        printf("Received REJ0\n");
        linklayer.numTransmissions++;
        linklayer.stats.numReceivedREJ++;
        return -1;
    }

    else if(control_field == C_REJ1){
        printf("Received REJ1\n");
        linklayer.numTransmissions++;
        linklayer.stats.numReceivedREJ++;
        return -1;
    }

    return -1;
}
```

```
int writeStuffedFrame(int fd, unsigned char *buffer, int length) {
    unsigned char frame[2 * length + 6];

    // Frame Header
    frame[0] = FLAG;
    frame[1] = A_ER;
    frame[2] = (linklayer.sequenceNumber==0 ? C_I0 : C_I1);
    frame[3] = BCC(A_ER, frame[2]);

    // BCC2 before byte stuffing
    unsigned char bcc2 = buffer[0];
    for(int i=1; i<length; i++){
        bcc2 ^= buffer[i];
    }

    // Process data
    int dataIndex=0, frameIndex=4;
    unsigned char bufferAux;

    while(dataIndex < length) {
        bufferAux = buffer[dataIndex++];

        // Byte Stuffing
        if(bufferAux == FLAG || bufferAux == ESCAPE) {
            frame[frameIndex++] = ESCAPE;
            frame[frameIndex++] = bufferAux ^ STUFFING;
        }
        else {
            frame[frameIndex++] = bufferAux;
        }
    }

    // Frame Footer
    if(bcc2 == FLAG || bcc2 == ESCAPE) {
        frame[frameIndex++] = ESCAPE;
        frame[frameIndex++] = bcc2 ^ STUFFING;
    }
    else {
        frame[frameIndex++] = bcc2;
    }

    frame[frameIndex++] = FLAG;
    write(fd, frame, frameIndex);
    linklayer.stats.numSentFramesI++;

    return frameIndex;
}

int destuffFrame(unsigned char* frame, int length, unsigned char* destuffed_frame){
    // Frame Header
    destuffed_frame[0] = frame[0]; // FLAG
```

```
    destuffed_frame[1] = frame[1]; // A
    destuffed_frame[2] = frame[2]; // C
    destuffed_frame[3] = frame[3]; // BCC1

    // Process data
    int dframeIndex=4, frameIndex;

    for(frameIndex=4; frameIndex < length-1; frameIndex++){
        if(frame[frameIndex] == ESCAPE){
            frameIndex++;
            destuffed_frame[dframeIndex] = frame[frameIndex] ^ STUFFING;
        }
        else{
            destuffed_frame[dframeIndex] = frame[frameIndex];
        }
        dframeIndex++;
    }

    // Frame Footer
    destuffed_frame[dframeIndex++] = frame[frameIndex++]; // FLAG

    return dframeIndex;
}

void generateBCC1Error(unsigned char *c, int percentage){
  int prob = (rand() % 100) + 1;

  if (prob <= percentage)
  {
    unsigned char randomByte = (unsigned char)((rand() % 177));
    *c = randomByte;
    printf("Generated BCC1 with errors\n\n");
  }
}

void generateBCC2Error(unsigned char *frame, int percentage){
  int prob = (rand() % 100) + 1;

  if (prob <= percentage){
    unsigned char randomByte = (unsigned char)((rand() % 177));
    frame[4] = randomByte;
    printf("Generated BCC2 with errors\n\n");
  }
}
```

- **msg_macros.h**

```
// Frame Macros

#define FLAG 0x7e  // Flag de inicio e fim
#define ESCAPE 0x7d // Octeto de escape
#define STUFFING 0x20 // Octeto para stuffing

#define A_ER 0x03  // Campo de Endereço (A) de commandos do Emissor, resposta do Receptor
#define A_RE 0x01  // Campo de Endereço (A) de commandos do Receptor, resposta do Emissor

#define C_SET 0x03 // Campo de Controlo - SET (set up)
#define C_DISC 0x0b // Campo de Controlo - DISC (disconnect)
#define C_UA 0x07 // Campo de Controlo - UA (Unnumbered Acknowledgement)
#define C_RR0 0x05 // Campo de Controlo - RR (receiver ready / positive ACK))
#define C_RR1 0x85
#define C_REJ0 0x01 // Campo de Controlo - REJ (reject / negative ACK))
#define C_REJ1 0x81
#define C_I0 0x00 // Campo de Controlo - I
#define C_I1 0x40

#define BCC(a,c) (a ^ c)

#define VERIFY_C(c) (c == C_SET || c == C_DISC || c == C_UA || c == C_RR0 || c == C_RR1 || c
== C_REJ0 || c == C_REJ1) ? 1 : 0

#define MAX_SIZE 1024
```

- **alarm.h**

```
#pragma once
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include "ll.h"

void alarmHandler(int signal);
void setAlarm();
void startAlarm();
void stopAlarm();
void unsetAlarm();
```

- **alarm.c**

```
#include "alarm.h"

void alarmHandler(int signal) {
   if(signal != SIGALRM) {
      return;
   }

   printf("Alarm: %d\n", linklayer.numTransmissions + 1);

   linklayer.alarm = 1;
   linklayer.numTransmissions++;
   linklayer.stats.numTimeouts++;
}

// Set sigaction struct and linkLayer struct
void setAlarm() {
   struct sigaction sa;
   sigemptyset(&sa.sa_mask);
   sa.sa_handler = &alarmHandler;
   sa.sa_flags = 0;

   sigaction(SIGALRM, &sa, NULL);

   linklayer.alarm = 0;

   alarm(linklayer.timeout); // start alarm
}

void startAlarm() {
   linklayer.alarm = 0;
   alarm(linklayer.timeout); // start alarm
}

void stopAlarm() {
   linklayer.alarm = 0;
   alarm(0); // uninstall alarm
}

// Unset sigaction struct
void unsetAlarm() {
   struct sigaction sa;
   sa.sa_handler = NULL;

   sigaction(SIGALRM, &sa, NULL);

   linklayer.alarm = 0;

   alarm(0); // uninstall alarm
}
```

# Anexo II – Testes de eficiência

- **Variação do tamanho das tramas I**

| | |
|---|---|
| **Nº total de bytes** | 71893 |
| **Nº total de bits** | 575144 |
| **C = Baudrate (bits/s)** | 38400 |

| Tamanho da trama (bytes) | Tempo total (s) | R (bits/s) | S = R/C | S (média) |
|---|---|---|---|---|
| 128 | 21,728023 | 26470,148711 | 0,6893267893 | 0,6893093413 |
| | 21,729123 | 26468,808704 | 0,6892918933 | |
| 256 | 20,274879 | 28367,320959 | 0,7387323166 | 0,7387247564 |
| | 20,275294 | 28366,740329 | 0,7387171961 | |
| 512 | 19,544775 | 29426,995194 | 0,7663279999 | 0,7662956945 |
| | 19,546423 | 29424,514143 | 0,7662633891 | |
| 1024 | 19,180669 | 29985,606863 | 0,7808751787 | 0,780872044 |
| | 19,180823 | 29985,366113 | 0,7808689092 | |
| 2048 | 18,999748 | 30271,138333 | 0,7883108941 | 0,7883063924 |
| | 18,999965 | 30270,792604 | 0,7883018907 | |

*Tabela 1: variação do tamanho das tramas I*

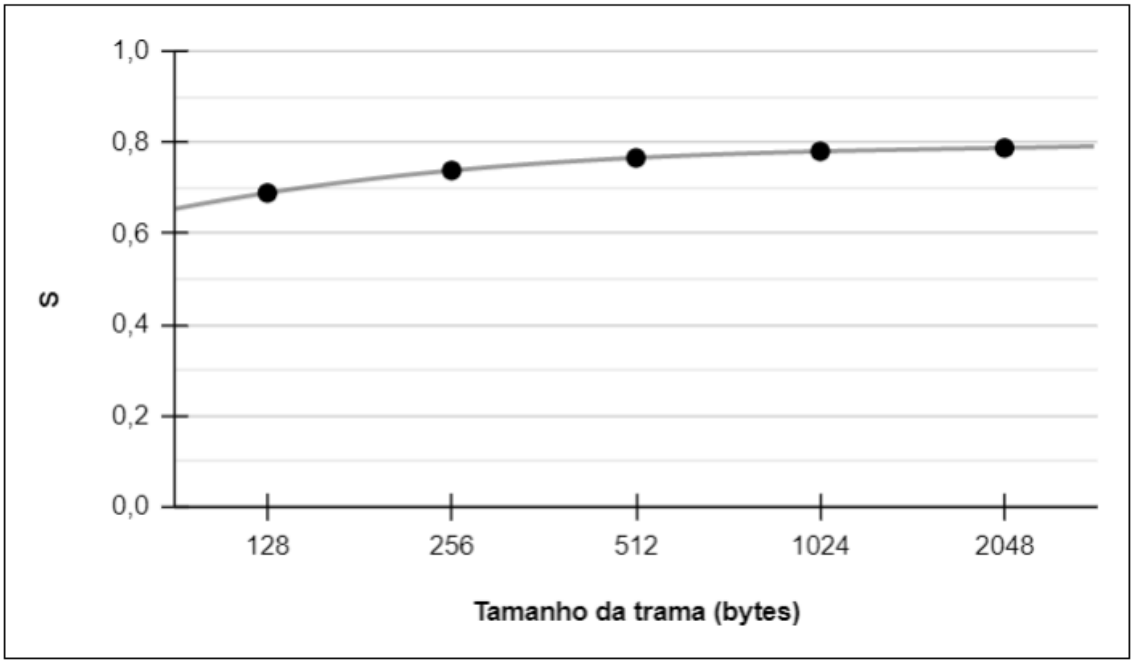


*Gráfico 1: variação do tamanho das tramas I*

- **Variação da capacidade de ligação**

| | |
|---|---|
| **Nº total de bytes** | 71893 |
| **Nº total de bits** | 575144 |
| **Tamanho da trama (bytes)** | 1024 |

| C = Baudrate (bits/s) | Tempo total (s) | R (bits/s) | S = R/C | S (média) |
|---|---|---|---|---|
| 4800 | 153,322009 | 3751,216174 | 0,7815033696 | 0,7815008032 |
| | 153,323016 | 3751,191537 | 0,7814982368 | |
| 9600 | 76,672457 | 7501,311716 | 0,7813919468 | 0,7813919468 |
| | 76,671936 | 7501,362689 | 0,7813919468 | |
| 19200 | 38,344732 | 14999,296383 | 0,7812133533 | 0,7812080359 |
| | 38,345254 | 14999,092195 | 0,7812027185 | |
| 38400 | 19,180918 | 29985,217600 | 0,7808650417 | 0,7808700898 |
| | 19,180670 | 29985,605300 | 0,780875138 | |
| 57600 | 12,793110 | 44957,324685 | 0,7805091091 | 0,7805119461 |
| | 12,793017 | 44957,651506 | 0,7805147831 | |

*Tabela 2: variação da capacidade de ligação*

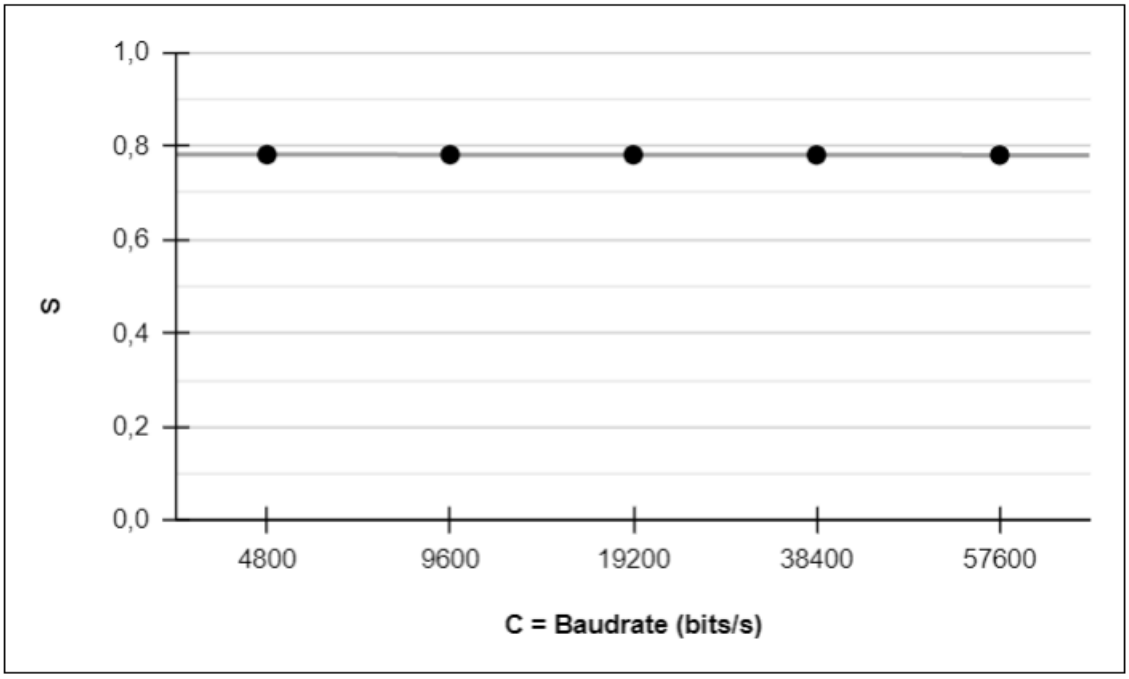


*Gráfico 2: variação da capacidade de ligação*

- **Variação do tempo de propagação**

| | |
|---|---|
| **Nº total de bytes** | 71893 |
| **Nº total de bits** | 575144 |
| **C = Baudrate (bits/s)** | 38400 |
| **Tamanho da trama (bytes)** | 1024 |

| Atraso de propagação (s) | Tempo total (s) | R (bits/s) | S = R/C |
|---|---|---|---|
| 0 | 19,181171 | 29984,822095 | 0,780854742 |
| 1 | 90,186959 | 6377,241304 | 0,1660739923 |
| 2 | 161,186896 | 3568,180877 | 0,09292137702 |

*Tabela 3: variação do tempo de propagação*

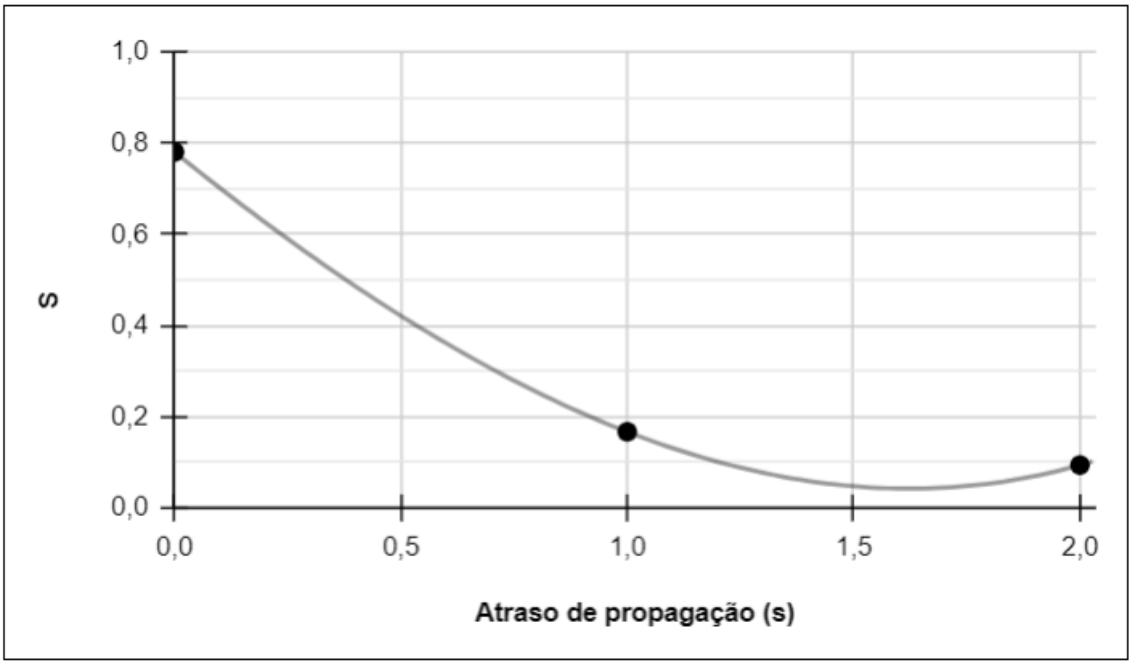


*Gráfico 3: variação do tempo de propagação*

- **Variação do FER – BCC1**

| Nº total de bytes | 71893 |
|---|---|
| Nº total de bits | 575144 |
| C = Baudrate (bits/s) | 38400 |
| Tamanho da trama (bytes) | 1024 |

| Probabilidade de erro BCC1 (%) | Tempo total (s) | R (bits/s) | S = R/C | S (média) |
|---|---|---|---|---|
| 5 | 22,181368 | 25929,149185 | 0,67523826 | 0,6752422936 |
| | 22,181103 | 25929,458963 | 0,6752463272 | |
| 10 | 28,181084 | 20408,867168 | 0,5314809158 | 0,5314770214 |
| | 28,181497 | 20408,568076 | 0,531473127 | |
| 15 | 49,181724 | 11694,262690 | 0,3045380909 | 0,3045388339 |
| | 49,181484 | 11694,319757 | 0,304539577 | |
| 20 | 61,182045 | 9400,535729 | 0,2448056179 | 0,2448060481 |
| | 61,181830 | 9400,568764 | 0,2448064782 | |
| 25 | 76,182263 | 7549,578830 | 0,1966036154 | 0,1966038309 |
| | 76,182096 | 7549,595380 | 0,1966040464 | |

*Tabela 4: variação do FER para BCC1*

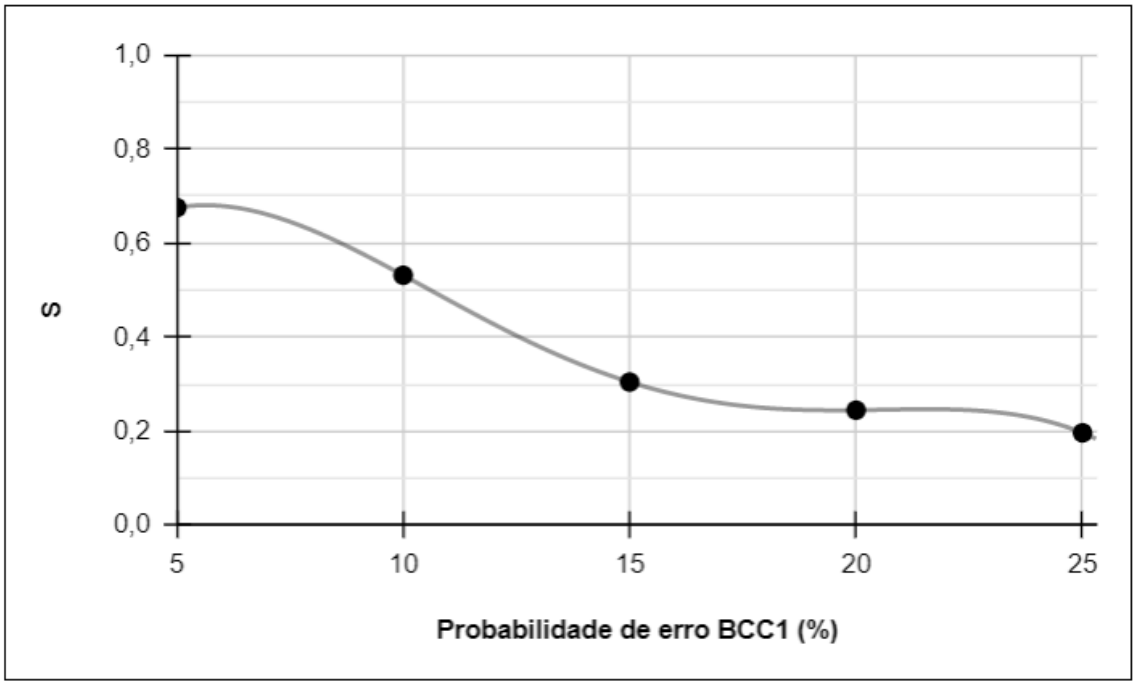


*Gráfico 4: variação do FER para BCC1*

- **Variação do FER – BCC2**

| | |
|---|---|
| **Nº total de bytes** | 71893 |
| **Nº total de bits** | 575144 |
| **C = Baudrate (bits/s)** | 38400 |
| **Tamanho da trama (bytes)** | 1024 |

| Probabilidade de erro BCC2 (%) | Tempo total (s) | R (bits/s) | S = R/C | S (média) |
|---|---|---|---|---|
| 5 | 19,453076 | 29565,709814 | 0,7699403597 | 0,769937134 |
| | 19,453239 | 29565,462081 | 0,7699339083 | |
| 10 | 19,998647 | 28759,145556 | 0,7489360822 | 0,7489430104 |
| | 19,998277 | 28759,677646 | 0,7489499387 | |
| 15 | 21,666633 | 26545,148939 | 0,6912799203 | 0,691430515 |
| | 21,657197 | 26556,714611 | 0,6915811097 | |
| 20 | 22,731534 | 25301,592053 | 0,6588956264 | 0,6589095982 |
| | 22,730570 | 25302,665089 | 0,65892357 | |
| 25 | 23,548639 | 24423,662021 | 0,6360328651 | 0,6360262345 |
| | 23,549130 | 24423,152787 | 0,6360196038 | |

*Tabela 5: variação do FER para BCC2*

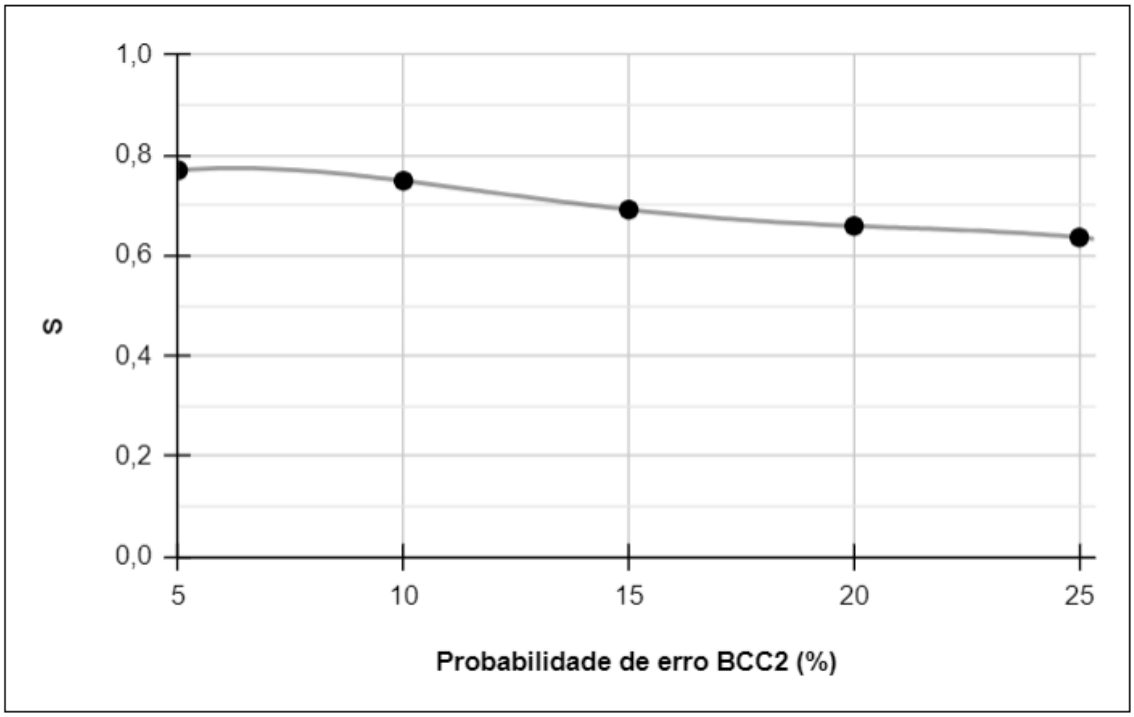


*Gráfico 5: variação do FER para BCC2*