U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

# Relatório do 1º trabalho laboratorial

## Redes de Computadores

3º ano do Mestrado integrado em engenharia Informática e Computação

Novembro de 2019


Carlos Jorge Albuquerque up201706735@fe.up.pt

Joaquim Manuel Rodrigues up201704844@fe.up.pt

Maria Helena Ferreira up201704508@fe.up.pt

# Índice

## Sumário

Este trabalho tinha como objetivo implementar um protocolo de ligação de dados que forneça um serviço de comunicação de dados fiável entre dois sistemas ligados por um meio (canal) de transmissão, um cabo série e testar o protocolo com uma aplicação simples de transferência de ficheiros.

Depois da realização deste trabalho podemos observar na prática muitos dos conteúdos que aprendemos na teoria da disciplina de redes de computadores, completando assim a nossa aprendizagem para a disciplina. Dado que todos os objetivos propostos foram concluídos, o trabalho foi um claro sucesso.

## Introdução

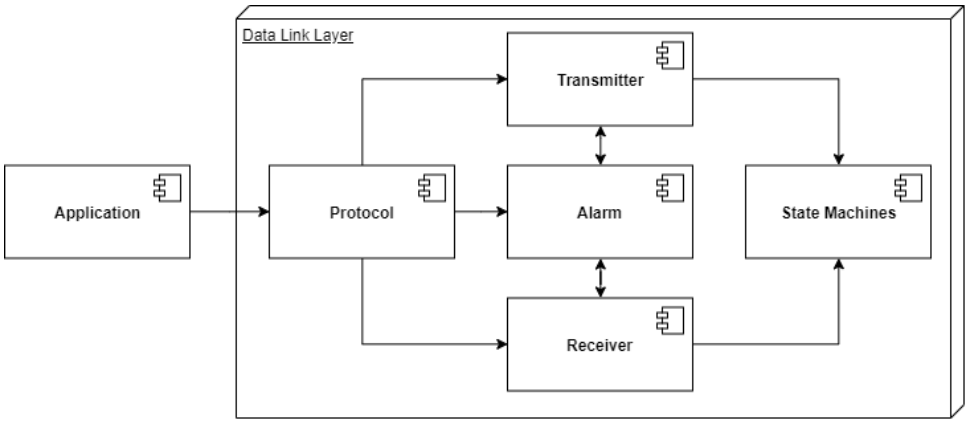
O trabalho laboratorial tem dois objetivos claros:

- Implementar um protocolo de ligação de dados, colocando na prática os conceitos aprendidos em aulas teóricas;
- Desenvolver uma aplicação simples de transferência de ficheiros para testar o protocolo implementado, permitindo também analisar a sua eficiência.

Por sua vez, este relatório tem como funcionalidade evidenciar a forma como decidimos cumprir os dois objetivos do trabalho e também as considerações que tomámos para tornar a nossa implementação o mais bem estruturada, modular e organizada possível. Por essa razão, o relatório começa por analisar o nosso trabalho na generalidade, passando depois a abordar mais concretamente a nossa aproximação a cada um dos objetivos. O código fonte do nosso trabalho segue em anexo e deve ser consultado à medida que é referenciado no relatório.

## Arquitetura

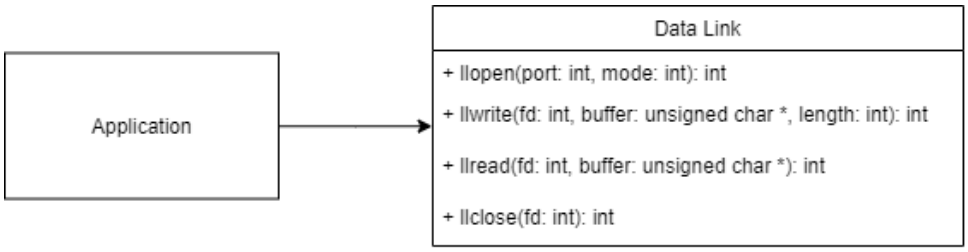
Para uma melhor organização e separação de tarefas no nosso programa, adotámos uma arquitetura por camadas (tal como no modelo TCP/IP). A figura seguinte ilustra os vários módulos do nosso software:




Como se pode ver, a camada superior (Application) faz uso da camada inferior (Data Link) para conseguir transferir ficheiros entre duas máquinas diferentes. É de notar que dentro do bloco Data Link Layer também existe uma hierarquia em camadas, sendo o módulo Protocol a camada superior, os módulos Transmitter, Alarm e Receiver a camada intermédia e o módulo State Machines a camada inferior.

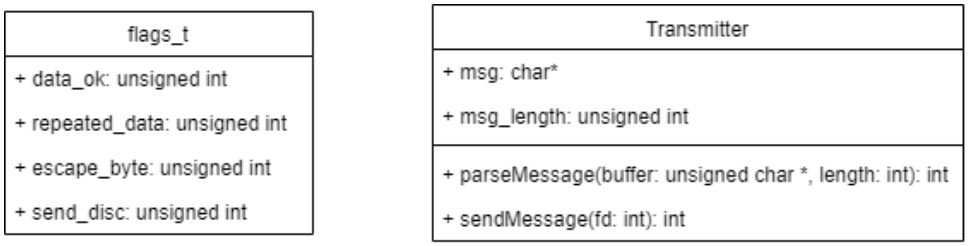
## Estrutura do código

A camada Application usa a API fornecida pela camada Data Link:




Estas quatro funções são o foco do nosso trabalho e estão todas concentradas no módulo Protocol.

Como forma de reutilizar funções, os módulos Transmitter e Receiver foram criados. Estes contêm as funções de baixo nível (i.e., que interagem diretamente com a porta série) utilizadas pelo emissor e pelo recetor, respetivamente. Do módulo Receiver destaca-se a estrutura de dados por nós criada *flags_t*, que tem como objetivo guardar informação sobre a qualidade e o tipo de dados recebidos numa trama (por exemplo, se os dados são repetidos ou se o caracter de controlo é um DISC). Do lado do Transmitter sobressai uma estrutura tipo classe, isto é, existem variáveis (*msg* e *msg_length*) que servem de membros da classe e funções (*parseMessage* e *sendMessage*) que operam sobre essas variáveis como métodos de uma classe.




As funcionalidades de timeout (tanto no emissor como no recetor) são implementadas à custa de alarmes do sistema operativo. O módulo Alarm é o responsável pela gestão dos timeouts e, recorrendo aos módulos Transmitter e Receiver, pelo reenvio das tramas, disponibilizando para isso a função *timeoutHandler*. Por forma a determinar qual a trama a enviar, foi criada uma estrutura de dados auxiliar *phase_t* que permite coordenar qual a atual fase da comunicação (abertura, dados e fecho para o transmissor; abertura e dados para o recetor). O número de timeouts e o intervalo de tempo são definidos no ficheiro *macros.h*, que funciona como um módulo auxiliar dentro do bloco Data Link que guarda todas as constantes utilizadas pelos restantes módulos.

O módulo State Machines é responsável pelas máquinas de estados usados para a interpretação das tramas, disponibilizando três funções: *suFrameSM*, *readSM* e *writeSM;* responsáveis por tramas de supervisão e controlo, tramas esperadas pelo recetor e tramas esperadas pelo emissor, respetivamente.

Por fim, o módulo Application proporciona uma aplicação simples de transferência de ficheiros através da API acima descrita. Este módulo faz uso da estrutura de dados *applicationLayer* que contém o modo de operação e o descritor de ficheiro da porta série. As funções *sendFile* e *receiveFile* coordenam a transferência de um ficheiro e é nelas que reside o foco da aplicação.

## Casos de uso

Existem cinco casos de uso principais:

1. Enviar um ficheiro - Quando a aplicação é aberta em modo de transmissor, a função *sendFile* é chamada, desencadeando o carregamento do ficheiro para um buffer (*getCharBuffer*) . É enviado o pacote de controlo start (*sendControlPacket*), e inicia-se o envio de dados (*sendDataPacket*), recorrendo a *llwrite* para os escrever na porta série. Quando terminado o envio, é transmitido o pacote de controlo end (*sendControlPacket*).
2. Receber um ficheiro – Por sua vez, quando a aplicação é aberta em modo de emissor, a função *receiveFile* é invocada. Esta recebe o pacote de controlo start (*receiveControlPacket*), e bloqueia a espera de dados (*receiveDataPacket*), na chamada de *llread.* O ciclo da receção termina quando todos os bytes especificados no pacote de controlo forem recebidos. No fim é ainda recebido o pacote de controlo end (*sendControlPacket*).
3. Lidar com erros – Caso uma trama de dados seja recebida com erros no BCC, os dados são descartados e é ativada uma flag de erro (*read_dataFrame*). Essa flag é interpretada pela função *llread,* que invoca *write_suFrame* para enviar REJ ao emissor e desencadear uma retransmissão dos mesmos dados.
4. Lidar com dados repetidos – Caso o emissor envia a mesma trama várias vezes seguidas, o recetor ignora os dados em *readDataFrame* e reenvia a confirmação (RR) ao emissor invocando *write_suFrame*.
5. Lidar com desconexão - Caso qualquer um dos intervenientes na comunicação deixe de responder, o mecanismo de timeout (*timeoutHandler*), que é ativado sempre que uma função bloqueia à espera de ler da porta série (*read_suFrame, read_dataFrame*), garante que após um número determinado de timeouts a aplicação encerra com um código de erro.

## Protocolo de ligação lógica

Desde o início do projeto que tivemos uma visão clara do que queríamos fazer: não bastava apenas fazer um protocolo que funcionasse, mas também que fosse bem estruturado e implementado, de tal forma que pudesse ser usado como qualquer biblioteca de C. Assim sendo, o resultado final é uma biblioteca *protocol.h* que fornece as quatro funções fulcrais deste trabalho: *llread, llwrite, llopen* e *llclose* (ver: Anexo 1 – protocol.h). É de notar que todas as funções por nós implementadas estão documentadas nos respetivos header files.

Para além disso, fizemos questão de garantir que o funcionamento do protocolo é completamente transparente às camadas superiores, ou seja, essas camadas não precisam de saber como nada está implementado para usarem o protocolo de ligação lógica. Dito isto, passemos à análise dos principais aspetos funcionais.

Em primeiro lugar, a função *llopen* trata da abertura da comunicação. Esta função tem dois modos de operação - um para o recetor e outro para o emissor – fazendo cada um a sua parte da abertura:

```
case TRANSMITTER:
  setPhase(open_phase);
  if (write_suFrame(fd, C_SET) == -1)
  {
    return -1;
  }
  printf("Sent SET\n");

  if (read_suFrame(fd, C_UA) == -1)
  {
    return -1;
  }
  resetTimeouts();
  printf("Received UA\n");
  break;
```

```
case RECEIVER:
  setPhase(receiver_phase);
  if (read_suFrame(fd, C_SET) == -1)
  {
    return -1;
  }
  resetTimeouts();
  printf("Received SET\n");

  if (write_suFrame(fd, C_UA) == -1)
  {
    return -1;
  }
  printf("Sent UA\n");
  break;
```

As funções *write_suFrame* (ver: Anexo 1 – transmitter.c - linhas 15 a 29) e read_*suFrame* (ver: Anexo 1 – receiver.c - linhas 31 a 54) são funções genéricas que escrevem e leem, respetivamente, tramas de supervisão e não numeradas na porta série.

Passando agora à função *llwrite* (ver: Anexo 1 – protocol.c - linhas 248 a 286), esta escreve o que lhe for passado como argumento na porta série. Para a tornar o mais eficiente possível decidimos guardar a trama a enviar como uma variável global no módulo Transmitter. Ao invocar *parseMessage* (ver: Anexo 1 – transmitter.c - linhas 31 a 110), essa variável é inicializada com a trama pronta a enviar e é também guardado o tamanho da trama. As variáveis são:

```
static unsigned char *msg;
static unsigned int msg_length;
```

Desta forma a retransmissão da mensagem não obriga a uma reconstrução da trama. Após enviar a trama, a função *llwrite* fica à espera de resposta por parte do transmissor. Caso a resposta não seja uma confirmação, a função reenvia a mensagem até receber uma confirmação.

Por sua vez, a função *llread* (ver: Anexo 1 – protocol.c - linhas 153 a 246) faz uso da uma estrutura de dados especial – *flags_t*, já referida na secção Estutura do código - que é manipulada na função *read_dataFrame* (ver: Anexo 1 – receiver.c - linhas 56 a 124) que lê as tramas de dados, faz o destuffing, verifica se os dados são repetidos ou se contêm erros. Este último ponto é feito de forma ligeiramente complexa para ser o mais eficiente possível. Em vez de carregarmos os dados todos para o buffer e só ao fim verificarmos se o último valor carregado (o BCC2) seria o resultado do XOR de todos os outros valores, o que teria uma complexidade temporal de 2N, optámos por ir verificando a validade dos dados há medida que são lidos da porta série da seguinte forma:

```
// If current byte is bcc2 would data be ok?
if (current_bcc2 == byte)
  flags->data_ok = TRUE;
else
  flags->data_ok = FALSE;
```

Tal como o comentário explica, esta aproximação considera que cada byte lido pode ser o BCC2. Nesse caso, os dados só estariam bem se o valor dos XOR de todos os valores lidos até ao momento (variável *current_bcc2*) fosse igual ao byte lido. Após a leitura de toda a trama da porta séria cabe à função *llread* interpretar as flags e os dados para responder de forma adequada ao transmissor. Esta função só termina quando for recebida a trama de desconexão e for tratada a desconexão.

Por fim, a função *llclose* (ver: Anexo 1 – protocol.c - linhas 288 a 308), que só é invocada pelo transmissor, trata de enviar a trama de desconexão e espera pela respetiva resposta.

Sempre que uma função fica bloqueada à espera de ler da porta série é ativado um alarme que desencadeia uma chamada a *timeoutHandler* (ver: Anexo 1 – alarm.c - linhas 38 a 76). Esta função trata de todos os timeouts e, graças á modularidade do nosso código, invoca as funções que transmitem os dados novamente. Depois de a trama ser reescrita é novamente ativado um alarme. O número máximo de timeouts está definido em *macros.h* na variável TIMEOUT_MAX_ATTEMPTS.

## Protocolo de aplicação

O protocolo de aplicação, sendo a camada de mais alto nível, utiliza os serviços fornecidos pela camada inferior. Visto tratar-se de um protocolo simples cujo principal objetivo é testar a API da ligação lógica, não houve grande preocupação da nossa parte em dividi-la em módulos.

Este protocolo é responsável por:

1. Verificar se os argumentos enviados pela linha de comandos são os corretos;
2. Em modo emissor, ler os caracteres do ficheiro e colocá-los num buffer;
3. Em modo recetor, criar o ficheiro onde serão guardados os dados recebidos;
4. Criar e enviar ou receber e guardar os pacotes de controlo e de dados;
5. Gerir o número sequencial dos pacotes, detetando repetições ou perdas dos mesmos.

No início da execução do programa, os argumentos passados pelo utilizador são interpretados por forma a preencher a seguinte estrutura de dados:

```
3   struct applicationLayer
4   {
5       int fileDescriptor; /* Serial port descriptor */
6       int status;         /* TRANSMITTER | RECEIVER */
7   } applicationLayer;
```

Esta estrutura controla a operação da aplicação, na medida em que guarda o descritor de ficheiro da porta série (retornado após a chamada a *llopen*) a usar e o modo de funcionamento: transmissor ou emissor. A partir deste momento a aplicação divide-se em duas partes distintas, uma para cada modo.

### Modo TRANSMITTER

Neste modo a aplicação foca-se apenas em enviar o ficheiro para o recetor. Toda essa lógica é controlada pela função *sendFile* (ver: Anexo 1 – application.c - linhas 98 a 141):

```
50   int sendFile(char *filename);
```

Esta função começa por invocar *getCharBuffer* (ver: Anexo 1 – application.c - linhas 14 a 35):

```
19   unsigned char *getCharBuffer(char *filename, int *fileSize);
```

Esta função é responsável por abrir o ficheiro com nome *filename* e ler todos os caracteres nele contidos para um buffer (cujo apontador é retornado pela função). O parâmetro *fileSize* é passado como apontador para que no final contenha o tamanho do ficheiro (em bytes), isto é, o tamanho do buffer retornado.

```
42  int sendControlPacket(unsigned int control, int fileSize, char *filename);
```

A função *sendControlPacket* (ver: Anexo 1 – application.c - linhas 61 a 96) cria um pacote de controlo, utilizando como campo de controlo o primeiro argumento (*control*). Esta função é invocada antes do envio dos dados com *control* = 2 e após o envio dos dados com *control* = 3. No pacote são também enviados o tamanho e nome do ficheiro (recorrendo a uma estrtura tipo TLV).

```
31  int sendDataPacket(int sendSize, int sequenceNumber, unsigned char *data, unsigned char *packet);
```

A função *sendDataPacket* (ver: Anexo 1 – application.c - linhas 37 a 59) cria um pacote de dados colocando o campo de controlo a 1. O número de sequência é passado como argumento em *sequenceNumber*. O tamanho do pacote (passado nos campos L2 e L1) é calculado com base no tamanho a enviar - *sendSize.* Os dados a colocar no pacote são passados pelo argumento *data* e depois de o pacote estar completamente criado é retornado através do argumento *packet*.

A função *sendFile* vai enviando os dados através da função *llwrite* até que a quantidade de bytes enviados (variável *sendSize*) seja igual ao tamanho do ficheiro (variável *fileSize*). Depois disso e após o envio do pacote de controlo final, *sendFile* retorna e procede-se ao fecho da ligação através da função *llclose* (ver: Anexo 1 – application.c - linhas 316 a 320).

## Modo RECEIVER

Neste modo a aplicação trata apenas de receber e guardar o ficheiro. Tudo isto é tratado na função *receiveFile* (ver: Anexo 1 – application.c - linhas 265 a 295).

```
67  int receiveFile();
```

Esta função chamada começa por receber um pacote de controlo de onde extrai logo à partida informação sobre o tamanho e o nome do ficheiro.

```
60  int receiveControlPacket(int control, char *filename);
```

A função *receiveControlPacket* (ver: Anexo 1 – application.c - linhas 143 a 204) é responsável por receber um pacote de controlo, lendo-o da porta série usando a função *llread*. O argumento *control* é o número de controlo esperado. Caso este seja diferente do número lido, é retornado um valor negativo. O campo de controlo deverá ser 2 se for o início da receção e 3 se for o fim da receção. O nome do ficheiro será retornado pelo argumento *filename.* Este valor é utilizado para criar o ficheiro no computador do recetor. Em caso de sucesso, a função retorna o tamanho do ficheiro.

Depois de ser recebido o pacote de controlo 2, *receiveFile* passa a receber pacotes de dados, entrando num ciclo que só termina quando o número de bytes lidos for igual ao tamanho do ficheiro. A função que recebe os pacotes de dados é *receiveDataPacket* (ver: Anexo 1 – application.c - linhas 207 a 263).

```
74  int receiveDataPacket(FILE *sendFile, int *fileWritten);
```

O argumento *sendFile* é um apontador para o ficheiro onde serão colocados os dados. O argumento *fileWritten* é usado para retornar o número de bytes já lidos, dado que a função usa o retorno para indicar se houve sucesso ou erro.

Ao ler todos os dados a função *receiveFile* recebe o pacote de controlo 3 e invoca *llread* uma última vez esperando a terminação da ligação e finalizando a receção do ficheiro.

# Eficiência do protocolo de ligação de dados

## Protocolo Stop & Wait

O nosso protocolo de ligação lógica usa o sistema de ARQ (Automatic Repeat ReQuest) Stop & Wait. Este método normalmente é eficiente para baixos débitos de transferência, mas quando os débitos são maiores ou a probabilidade de uma trama ter um erro (Frame Error Ratio ou FER) aumenta, a eficiência diminui bastante.

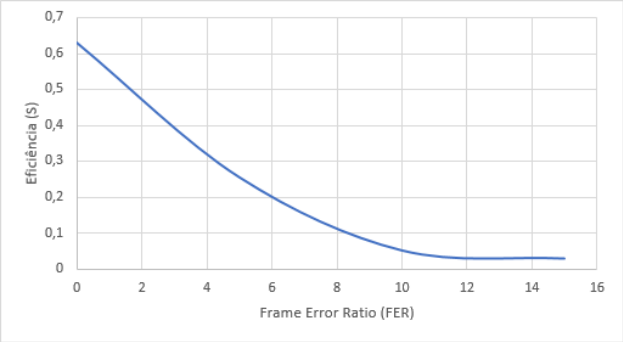
A eficiência teórica do Stop & Wait é dada pela expressão $S = (1 – p_e) / (1 – 2a)$, em que "$p_e$" é o FER e "a" é o quociente entre os tempos de propagação e de transmissão da trama.

## Caraterização estatística da eficiência

A caraterização estatística da eficiência do protocolo foi feita com recurso a medidas sobre o código desenvolvido, fazendo-se variar a Frame Ratio Error (ver anexo 2 – Variação de FER), o tempo de propagação (ver anexo 2 – Variação do Tempo de Propagação), a capacidade da ligação (ver anexo 2 – Variação da Capacidade da Ligação) e o tamanho da trama (ver anexo 2 – Variação do Tamanho da Trama). O ficheiro utilizado foi o pinguim.gif disponibilizado no moodle.

## Caraterização estatística da eficiência alterando a variável FER

Para variar o FER, geramos aleatoriamente erros com a mesma percentagem no BCC1 e no BCC2. Os erros eram introduzidos quando um número aleatório entre 1 e 100 se encontrava abaixo do valor de FER que definimos. O resultado está demostrado no gráfico:




Como esperado, o aumento do FER diminui desmedidamente a eficiência do protocolo. Esta diminuição é fruto de tempo perdido a reenviar tramas por causa de BCC's errados e também tempo perdido em timeouts. A partir dos 20% de FER a transmissão não completou em nenhuma das tentativas, visto que ocorriam 3 erros seguidos no BCC1, levando o transmissor a dar timeout e sair.

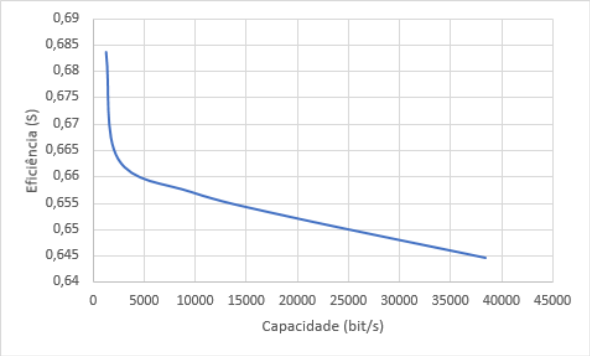
## Caraterização estatística da eficiência alterando o tempo de propagação

Fizemos variar o tempo de propagação de 0 a 0.8. A simulação do tempo de propagação consistiu simplesmente em colocar o recetor a dormir durante o tempo desejado para só receber os dados depois disso. O resultado é o seguinte:


Mesmo sendo uma pequena diferença, esta variação baixou imensamente a eficiência do protocolo. Um maior tempo de propagação aumenta o tempo de envio de cada trama. Este resultado deve-se também ao facto de usarmos tramas de 100 bytes, o que aumentou o número de tramas enviadas e, consequentemente, agravou o impacto do tempo de propagação.

### Caraterização estatística da eficiência alterando a capacidade da ligação

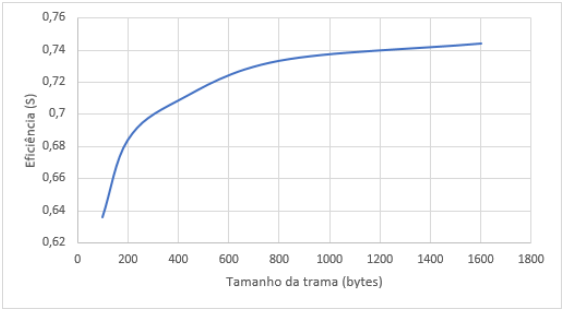
A variação da capacidade de ligação, em bits/s, foi feita alterando o valor de baudrate a ser usado. Este tomou valores entre 1200 e 238400, resultando no gráfico:




Apesar uma variação elevada entre os valores da capacidade da ligação, a eficiência quase não foi afetada (manteve-se sempre na ordem dos 60%). Ainda assim, podemos concluir que a eficiência é melhor para capacidades mais baixas, o que vai ao encontro da análise teórica.

### Caraterização estatística da eficiência alterando o tamanho de trama

O tamanho da trama foi variado através do valor da macro TRANSMIT_SIZE (ver anexo 1 - macros.h), tendo sido testados valores entre 100 e 1600 bytes. O resultado foi:




Dado que com o aumento do tamanho da trama são enviados mais dados de cada vez, criando-se menos tramas, seria de esperar um aumento da eficiência que se comprovou na prática.

## Validação

Para testar o nosso código foram realizadas quatro situações distintas:

1. Foram enviados ficheiros de vários tamanhos.
2. Foi desligada a conexão física (porta série) e verificou-se o encerramento dos dois lados quando se deu o terceiro time-out com retransmissão não sucedida.
3. Foi desligada a conexão física durante um ou dois timeouts, após os quais voltaram-se a unir os cabos e deu-se a esperada retransmissão dos dados, ficando o ficheiro recebido igual ao transmitido.
4. Foi inserido ruído na ligação para simular erros nas tramas, dos quais resultaram duas situações diferentes: afetando a receção dos dados no recetor, este enviava uma confirmação negativa REJ para o emissor que, por sua vez, procedia à retransmissão dos dados; afetando a emissão dos dados no emissor, a trama de confirmação RR era perdida e os dados eram reenviados ao emissor que procedia a tratamento de dados repetidos.

Todas esta situações tiveram os resultados esperados.

## Conclusões

Ao longo da realização do projeto foi-nos possível aprofundar conhecimentos acerca das funções que compõem um protocolo de ligação de dados e que são responsáveis pelo seu bom e normal funcionamento: sincronismo, controlo, estabelecimento e término da ligação, entre outras. Um outro conceito fundamental para a realização do trabalho é a arquitetura em camadas, que dita a separação em termos de funções e privilégios do nível da ligação de dados e do nível da aplicação. Através da comparação da eficiência prática com a eficiência teórica do protocolo foi também possível observar a discrepância entre a eficácia e rapidez teóricas do mesmo ao alterar valores como o tempo de propagação e a taxa de probabilidade de erro na trama.

Assim, consideramos este trabalho bastante produtivo para um melhor aproveitamento da Unidade Curricular uma vez que alia a sua componente prática e teórica, funcionando como uma base dos sistemas de Redes de Computadores.

## Anexo 1 – macros.h

```
#pragma once

#define BAUDRATE B38400
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FALSE 0
#define TRUE 1

// Supervision and Unumbered (SU) frames structure
#define F1_INDEX 0
#define A_INDEX 1
#define C_INDEX 2
#define BCC_INDEX 3
#define F2_INDEX 4
#define SU_FRAME_SIZE 5

// Flag and A characters values
#define FLAG 0x7e
#define A 0x03

// Control character values for SU frames
#define C_SET 0x03
#define C_UA 0x07
#define C_DISC 0x0B
#define RR_0 0x05
#define RR_1 0x85
#define REJ_0 0x01
#define REJ_1 0x81

// Control character values for Data frames
#define CONTROL_0 0x00
#define CONTROL_1 0x40

// Stuffing bytes
#define ESC 0x7d
#define ESC_SOL 0x5d
#define FLAG_SOL 0x5e

// Timeout related values
#define TIMEOUT_INTERVAL 3
#define TIMEOUT_MAX_ATTEMPTS 3

// State Machine states
#define START 0
#define FLAG_RCV 1
#define A_RCV 2
#define C_RCV 3
#define BCC_OK 4
#define DATA_LOOP 5
```

```
#define END 6

//transmit size
#define TRANSMIT_SIZE 1024
```

## Anexo 1 – alarm.h

```h
#pragma once

typedef enum {
    open_phase,
    data_phase,
    close_phase,
    receiver_phase
} phase_t;
      Joaquim Rodrigues, 13 days ago • Uploaded Project1
/**
 *  Sets phase variable value.
 *
 *  @param new_phase New phase value
 */
void setPhase(phase_t new_phase);

/**
 *  Resets the number of timeouts to 0.
 */
void resetTimeouts();

/**
 *  Installs the timeoutHandler for SIGALRM.
 *  Also sets the serial port file descriptor to be used on retransmissions.
 *
 *  @param fd Serial port file descriptor
 *  @return 0 in case of success; -1 on error.
 */
int initializeHandler(int fd);

/**
 *  Handles SIGALRM. Calls serial port transmission methods to retransmit frames.
 *
 *  @param signo Signal number of the received signal
 */
void timeoutHandler(int signo);
```

# Anexo 1 – alarm.c

```c
#include "alarm.h"      root, 13 days ago • change includes
#include "../macros.h"
#include "../transmitter/transmitter.h"
#include <signal.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

// This variables are static so no one can access them from other files.
static int serial_fd;    /**< Serial port file descriptor. Used for frame retransmissions. */
static int timeouts = 0; /**< Number of timeouts ocurred. */
static phase_t phase;    /**< Connection's phase. Can be open, data or close. */

void setPhase(phase_t new_phase)
{
  phase = new_phase;
}

void resetTimeouts() { timeouts = 0; }

int initializeHandler(int fd)
{
  struct sigaction new_action;
  serial_fd = fd;

  memset(&new_action, 0, sizeof(new_action));
  new_action.sa_handler = timeoutHandler;

  if (sigaction(SIGALRM, &new_action, NULL) < 0)
  {
```

```
32      return -1;
33    }
34
35    return 0;
36  }
37
38  void timeoutHandler(int signo)
39  {
40    if (signo != SIGALRM)
41      return;
42
43    timeouts++;
44    if (timeouts <= TIMEOUT_MAX_ATTEMPTS)
45    {
46      printf("Timeout\n");
47      switch (phase)
48      {
49      case open_phase:
50        write_suFrame(serial_fd, C_SET);
51        printf("Resent SET\n");
52        break;
53      case data_phase:
54        sendMessage(serial_fd);
55        printf("Resent Data\n");
56        break;
57      case close_phase:
58        write_suFrame(serial_fd, C_DISC);
59        printf("Resent DISC\n");
60        break;
61
62      default:
63        break;
64      }
65
66      // Alarm is set again
67      alarm(TIMEOUT_INTERVAL);
68    }
69    else
70    {
71      fprintf(stderr, "Timeout: other end took too long to respond.\n");
72      exit(-1);
73    }
74
75    return;
76  }
```

# Anexo 1 – application.h

```c
#include <stdio.h>

CajoAlbuquerque, 11 days ago | 3 authors (root and others)
struct applicationLayer
{
    int fileDescriptor; /* Serial port descriptor */
    int status;         /* TRANSMITTER | RECEIVER */
} applicationLayer;

struct applicationLayer application;

/**
 *  gets all the chars of the file given to transmit and sends the pointer to them
 *
 *  @param filename name of the file to transmit given by the user
 *  @param fileSize size of the file to transmit know after get the chars
 *
 *  @return all the chars in the file
 */
unsigned char *getCharBuffer(char *filename, int *fileSize);

/**
 * creates the data package to send to llwrite
 *
 * @param sendSize size of the part of the file we are going to send
 * @param sequenceNumber number of files already send
 * @param data read of the file      MHelena45, 10 days ago • add application.h
 * @param packet buffer with the complete packet(4 bytes + data)
 *
 *  @return data package
 */
int sendDataPacket(int sendSize, int sequenceNumber, unsigned char *data, unsigned char *packet);

/**
 * creates the control package
 *
 * @param control 2 if start and 3 if end package control
 * @param fileSize of the all file
 * @param filename name of the file to transmit
 *
 *  @return 0 if success , -1 otherwise
 */
int sendControlPacket(unsigned int control, int fileSize, char *filename);

/**
 * sends a file
 *
 * @param filename name of the file to transmit given by the TRANSMITTER
 * @return 0 if success
 */
int sendFile(char *filename);

/**
 * receives the control package
 *
 * @param control expected control, 2 if start and 3 if end
 * @param filename gives the name of the file receiving (control 2) ou checks if it was well send(control 3)
 *
 *  @return success
 */
int receiveControlPacket(int control, char *filename);
```

```
61
62  /**
63   *  receives a file
64   *
65   *  @return 0 if success
66   */
67  int receiveFile();
68
69  /**
70   * receives the packets with data
71   * @param sendFile file where the data is going to be, the same name of the original data
72   * @param fileWritten number of written chars, used to know when to stop
73   */
74  int receiveDataPacket(FILE *sendFile, int *fileWritten);
```

## Anexo 1 – application.c

```c
#include "application.h"
#include "../macros.h"
#include "../protocol/protocol.h"

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

// Functions of the transmitter
unsigned char *getCharBuffer(char *filename, int *fileSize)
{
    FILE *f;
    unsigned char *fileData;
    struct stat fData;
    if ((f = fopen(filename, "rb")) == NULL)
    {
        perror("Error opening the file :");
        exit(-1);
    }
    stat(filename, &fData);
    (*fileSize) = fData.st_size;

    printf("Get %d bytes form file %s\n", *fileSize, filename);

    fileData = (unsigned char *)malloc(*fileSize);

    fread(fileData, sizeof(unsigned char), *fileSize, f);
```

```
32
33      fclose(f);
34      return fileData;
35  }
36
37  int sendDataPacket(int sendSize, int sequenceNumber, unsigned char *data, unsigned char *packet)
38  {
39      static int sequenceCounter = 0;
40      int count = 0;
41
42      packet[0] = 1 + '0'; //converting in char
43      packet[1] = sequenceNumber;
44
45      int L2 = sendSize / 256;
46      int L1 = sendSize - (256 * L2);
47
48      packet[2] = L2;
49      packet[3] = L1;
50
51      while (count < sendSize) //getting the real data
52      {
53          packet[4 + count] = data[count + (sequenceCounter * TRANSMIT_SIZE)];
54          count++;
55      }
56
57      sequenceCounter++;
58      return 0;
59  }
60
61  int sendControlPacket(unsigned int control, int fileSize, char *filename)
62  {
63      char sizeString[16];
64      sprintf(sizeString, "%d", fileSize);
65
66      int size = 5 + strlen(sizeString) + strlen(filename);
67      unsigned char ctrlPac[size];
68
69      ctrlPac[0] = control + '0';
70      ctrlPac[1] = 0 + '0';
71      ctrlPac[2] = strlen(sizeString) + '0';
72
73      size_t i, j = 3;
74      for(i = 0; i < strlen(sizeString); i++) {
75          ctrlPac[j] = sizeString[i];
76          j++;;
```

```
77      }
78
79      ctrlPac[j] = 1 + '0';
80      j++;
81      ctrlPac[j] = strlen(filename) + '0';
82      j++;
83
84      for(i = 0; i < strlen(filename); i++) {
85          ctrlPac[j] = filename[i];
86          j++;
87      }
88
89      if (llwrite(application.fileDescriptor, ctrlPac, size) < 0) {
90          printf("ERROR in sendCtrlPkt(): llwrite() function error!\n");
91          return -1;
92      }
93
94      return 0;
95
96  }
97
98  int sendFile(char *filename)
99  {
100     int fileSize;
101     int sendSize = 0, sequenceNumber = 0;
102     unsigned char *fileData, *dataSend;
103
104     dataSend = (unsigned char *)malloc((TRANSMIT_SIZE + 4) * sizeof(char));
105
106     fileData = getCharBuffer(filename, &fileSize);
107
108     printf("Sending control packet with control 2\n");
109     sendControlPacket(2, fileSize, filename);
110
111     while ((fileSize - sendSize) >= TRANSMIT_SIZE)
112     { //if possible sends TRANSMIT_SIZE bytes of data
113
114         sendSize += TRANSMIT_SIZE; //each time only TRANSMIT_SIZE are really data
115         sendDataPacket(TRANSMIT_SIZE, sequenceNumber, fileData, dataSend );
116         sequenceNumber = (sequenceNumber + 1) % 255; //sequencial number in modules of 255
117
118         if(llwrite(application.fileDescriptor, dataSend, TRANSMIT_SIZE + 4) < 0){
119             printf("Comunication failed: failed to write to the serial port\n");
120             return -1;
121         }
122     }
123     if ((fileSize - sendSize) > 0)
```

```
124     {
125         sendDataPacket((fileSize - sendSize) , sequenceNumber, fileData, dataSend);
126         if(llwrite(application.fileDescriptor, dataSend, ((fileSize - sendSize)+4)) < 0){
127             printf("Comunication failed: failed to write to the serial port\n");
128             return -1;      CajoAlbuquerque, 8 days ago • Added error checking on llwrite
129         }
130         sendSize += (fileSize - sendSize);
131 
132     }
133     printf("Total bytes sent: %d\n", sendSize);
134 
135     printf("Sending control packet with control 3\n");
136     sendControlPacket(3, fileSize, filename);
137 
138     free(dataSend);
139     free(fileData);
140     return 0;
141 }
142 
143 int receiveControlPacket(int control, char *filename)
144 {
145     unsigned char controlPac[70];
146     int fileSize = 0;
147 
148     if (llread(application.fileDescriptor, controlPac) < 0) {
149         printf("ERROR in rcvCtrlPkt(): \n");
150         return -1;
151     }
152 
153     if ((controlPac[0] - '0') != control) {
154         printf("ERROR in rcvCtrlPkt(): unexpected control field!\n");
155         return -2;
156     }
157 
158     if ((controlPac[1] - '0') != 0) {
159         printf("ERROR in rcvCtrlPkt(): unexpected size param!\n");
160         return -3;
161     }
162 
163     int i, fileSizeLength = (controlPac[2] - '0'), j = 3;
164 
165     char fileSizeStr[25];
166 
167     for(i = 0; i < fileSizeLength; i++) {
168         fileSizeStr[i] = controlPac[j];
169         j++;
170     }
```

```
171
172     fileSizeStr[j - 3] = '\0';
173
174     fileSize = atoi(fileSizeStr);
175
176     if((controlPac[j] - '0') != 1) {
177         printf("ERROR in rcvCtrlPkt(): unexpected name param!\n");
178         return -4;
179     }
180
181     j++;
182     int pathLength = (controlPac[j] - '0');
183     j++;
184
185     char pathStr[30];
186
187     for(i = 0; i < pathLength; i++) {
188         pathStr[i] = controlPac[j];
189         j++;
190     }
191
192     pathStr[i] = '\0';
193
194     if (3 == control && (strcmp(filename,pathStr) != 0)) {
195         //compare with the other filename receive in the start
196         printf("Name of received file can be wrong!\n");
197     }
198     else{
199         strcpy(filename, pathStr);
200     }
201
202     return fileSize;
203
204 }
205
206 //Functions of the receiver
207 int receiveDataPacket(FILE *sendFile, int *fileWritten)
208 {
209     int control;
210     int L1, L2;
211     static int sequenceNumber = 0;
212     int readSize, receiveSequenceNumber;
213     unsigned char fileData[TRANSMIT_SIZE +4];
214
215     readSize = llread(application.fileDescriptor, fileData);
216
217     control = fileData[0] - '0'; //converting to char
218
219     if (control == 3)
220     {
221         printf("Receive control Packet 3 to soon \n");
222         return -1;
223     }
224     else if (control == 2)
225     {
226         //receive control 2 twice, ignoring
227         return -1;
228     }
229     else if (control != 1)
230     {
231         printf("Wrong control receive, expected data control 1\n");
232         return -1;
233     }
234
235     receiveSequenceNumber = fileData[1];
236
237     if (receiveSequenceNumber != sequenceNumber)
238     {
239         printf("Expected sequencial number %d, received %d", sequenceNumber, receiveSequenceNumber);
240         return -1;
241     }
242
243     L2 = fileData[2];
244     L1 = fileData[3];
245
246     if ((readSize-4)  != ((256 * L2) + L1))
247     {
248         printf("Wrong Reception, L1 and L2 don't match readSize \n");
249         return -1;
250     }
251
252     //if everything went well, and only then, change values
253     sequenceNumber = (sequenceNumber + 1) % 255;
254     *fileWritten += (readSize - 4); //control, sequence number and L1 and L2 aren't data
255
256     //puts char by char
257     for(int i=0 ; i < (readSize-4); i++)
258     {
259         putc(fileData[4 + i], sendFile);
260     }
261
262     return 0;
263 }
264
265 int receiveFile()
266 {
```

```
    FILE *sendFile;
    int fileSize, fileReceived = 0;
    char *filename;
    filename = (char *)malloc(30* sizeof(char));

    fileSize = receiveControlPacket(2, filename);

    sendFile = fopen(filename, "w");
    if(sendFile == NULL){
        printf("Couldn't open file\n");
        exit(-1);
    }
    while (fileReceived < fileSize)
    {
        receiveDataPacket(sendFile, &fileReceived);
    }
    printf("Total bytes received: %d\n", fileReceived);
    printf("Receiving control packet with control 3\n");
    receiveControlPacket(3, filename);

    unsigned char buf[1];
    if(llread(application.fileDescriptor, buf) != -1){
        printf("Connection did not close when expected. Aborting...");
        exit(-1);
    }
    fclose(sendFile);
    free(filename);

    return 0;
}

int main(int argc, char **argv)
{
    if (argc < 3 || (strcmp("TRANSMITTER", argv[2]) != 0 && strcmp("RECEIVER", argv[2]) != 0))
    {
        printf("Usage: %s <serial port number> <TRANSMITTER || RECEIVER>\n", argv[0]);
        exit(1);
    }
    if(strcmp("TRANSMITTER", argv[2]) != 0 && !(argc < 4))
    {
        printf("Usage: %s <serial port number> <TRANSMITTER || RECEIVER> <file name to send>\n", argv[0]);
        exit(1);
    }
    if(strcmp("TRANSMITTER", argv[2]) == 0)
        application.status = TRANSMITTER;
    else
        application.status = RECEIVER;

    application.fileDescriptor = llopen(atoi(argv[1]), application.status);

    if (strcmp("TRANSMITTER", argv[2]) == 0)
    {
        sendFile(argv[3]);
        llclose(application.fileDescriptor);
    }
    else
    {
        receiveFile();
    }

    printf("Successfull transmition\n");
    return 0;
}
```

# Anexo 1 – protocol.h

```c
#pragma once
// Data connection protocol services
#define COM0 "/dev/ttyS0"
#define COM1 "/dev/ttyS1"
#define COM2 "/dev/ttyS2"      CajoAlbuquerque, 25 days ago • Added initial project files based on previous
#define COM3 "/dev/ttyS3"
#define TRANSMITTER 0
#define RECEIVER 1

/**
 *  Opens the connection on the serial port.
 *
 *  @param port Serial port to use; use values from 0 to 3
 *  @param mode Flag to switch between TRANSMITTER and RECEIVER modes
 *
 *  @return Connection port file descriptor on success, -1 on error.
 */
int llopen(int port, int mode);

/**
 *  Reads a data frame from the serial port.
 *
 *  @param fd File descriptor of the serial port
 *  @param buffer Array of characters on which the data will be saved
 *
 *  @return Number of characters read (buffer length) on success, -1 on connection closing and -2 on error.
 */
int llread(int fd, unsigned char *buffer);

/**
 *  Writes a data frame to the serial port.
 *
 *  @param fd File descriptor of the serial port
 *  @param buffer Array of characters to be written
 *  @param length Array's length
 *
 *  @return Number of written characters on success, -1 otherwise.
 */
int llwrite(int fd, unsigned char *buffer, int length);

/**
 *  Closes the connection on the serial port.
 *  Resets the serial port configuration and closes its file descriptor
 *
 *  @param fd File descriptor of the serial port
 *
 *  @return 0 on success, -1 otherwise.
 */
int llclose(int fd);
```

## Anexo 1 – protocol.c

```c
#include "protocol.h"
#include "../alarm/alarm.h"
#include "../macros.h"
#include "../receiver/receiver.h"
#include "../state_machine/statemachine.h"
#include "../transmitter/transmitter.h"

#include <fcntl.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <termios.h>
#include <unistd.h>

static struct termios oldtio;

/**
 *  Initializes the serial port connection.
 *  Configures the serial port with non-canonical mode.
 *
 *  @param port Serial port number
 *  @return Serial port file descriptor
 */
int initSerialPort(int port)
{
  struct termios newtio;
  char serialPort[11];
  int fd;

  switch (port)
  {
  case 0:
    strcpy(serialPort, "/dev/ttyS0");
    break;
  case 1:
    strcpy(serialPort, "/dev/ttyS1");
    break;
  case 2:
    strcpy(serialPort, "/dev/ttyS2");
    break;
  case 3:
    strcpy(serialPort, "/dev/ttyS3");
    break;

  default:
    fprintf(stderr, "Invalid serial port number\n");
    return -1;
  }
```

```
50
51      fd = open(serialPort, O_RDWR | O_NOCTTY);
52      if (fd < 0)
53      {
54        perror(serialPort);
55        return -1;
56      }
57
58      if (tcgetattr(fd, &oldtio) == -1)
59      { /* save current port settings */
60        perror("tcgetattr");
61        return -1;
62      }
63
64      bzero(&newtio, sizeof(newtio)); /*memory allocation with zeros*/
65      newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
66      newtio.c_iflag = IGNPAR;
67      newtio.c_oflag = 0;
68      newtio.c_lflag = 0; /* set input mode (non-canonical, no echo,...) */
69
70      newtio.c_cc[VTIME] = 0; /* inter-character timer unused */
71      newtio.c_cc[VMIN] = 1;  /* blocking read until 1 char received */
72
73      tcflush(fd, TCIOFLUSH);
74
75      if (tcsetattr(fd, TCSANOW, &newtio) == -1)
76      {
77        perror("tcsetattr");
78        return -1;
79      }
80
81      printf("New termios structure set\n");
82
83      return fd;
84    }
85
86    /**
87     *  Safely closes the serial port connection.
88     *  Configures the serial port back to its initial state.
89     *
90     *  @param port Serial port number
91     *  @return Serial port file descriptor
92     */
93    int closeSerialPort(int fd)
94    {
95      if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
96      {
97        perror("tcsetattr");
```

```
98      return -1;
99    }
100
101   close(fd);
102   return 0;
103 }
104
105 int llopen(int port, int mode)
106 {
107   int fd = initSerialPort(port);
108
109   if (initializeHandler(fd) == -1)
110     return -1;
111
112   switch (mode)
113   {
114   case TRANSMITTER:
115     setPhase(open_phase);
116     if (write_suFrame(fd, C_SET) == -1)
117     {
118       return -1;
119     }
120     printf("Sent SET\n");
121
122     if (read_suFrame(fd, C_UA) == -1)
123     {
124       return -1;
125     }
126     resetTimeouts();
127     printf("Received UA\n");
128     break;
129
130   case RECEIVER:
131     setPhase(receiver_phase);
132     if (read_suFrame(fd, C_SET) == -1)
133     {
134       return -1;
135     }
136     resetTimeouts();
137     printf("Received SET\n");
138
139     if (write_suFrame(fd, C_UA) == -1)
140     {
141       return -1;
142     }
143     printf("Sent UA\n");
144     break;
145
146   default:
```

```
147     return -1;
148   }
149
150   return fd;
151 }
152
153 int llread(int fd, unsigned char *buffer)
154 {
155   int result;
156   unsigned int NR = getNR();
157   flags_t flags;
158   initFlags(&flags);
159
160   result = read_dataFrame(fd, buffer, &flags);
161   if(result != -1)
162     printf("Received Data: %d bytes\n", result);
163   else
164     return -2;
165
166
167   resetTimeouts();
168
169   // When there is repeated data, buffer will have no content
170   if (flags.repeated_data)
171   {
172     if (NR == 1)
173     {
174       printf("Received Repeated Data 0\n");
175       if (write_suFrame(fd, RR_1) < 0)
176         return -2;
177       printf("Sent RR_1\n");
178     }
179     else
180     {
181       printf("Received Repeated Data 1\n");
182       if (write_suFrame(fd, RR_0) < 0)
183         return -2;
184       printf("Sent RR_0\n");
185     }
186
187     return 0;
188   }
189
190   // When DISC is received, buffer will have no content
191   if (flags.send_disc)
192   {
193     printf("Received DISC\n");
194
195     if (write_suFrame(fd, C_DISC) < 0)
```

```
196         return -2;
197
198       printf("Sent DISC\n");
199
200       setPhase(close_phase);
201       if (read_suFrame(fd, C_UA) < 0)
202         return -2;
203       resetTimeouts();
204       printf("Received UA\n");
205
206       closeSerialPort(fd);
207       return -1;
208     }
209
210     if (flags.data_ok)
211     {
212       if (NR == 1)
213       {
214         setNR(0);
215         if (write_suFrame(fd, RR_0) < 0)
216           return -2;
217         printf("Sent RR_0\n");
218       }
219       else
220       {
221         setNR(1);
222         if (write_suFrame(fd, RR_1) < 0)
223           return -2;
224         printf("Sent RR_1\n");
225       }
226     }
227     else
228     {
229       if (NR == 1)
230       {
231         result = 0;
232         if (write_suFrame(fd, REJ_1) < 0)
233           return -2;
234         printf("Sent REJ_1\n");
235       }
236       else
237       {
238         result = 0;
239         if (write_suFrame(fd, REJ_0) < 0)
240           return -2;
241         printf("Sent REJ_0\n");
242       }
243     }
244
```

```
245     return result;
246   }
247
248   int llwrite(int fd, unsigned char *buffer, int length)
249   {
250     unsigned char control;
251     int result;
252
253     setPhase(data_phase);
254
255     parseMessage(buffer, length);
256
257     do
258     {
259       result = sendMessage(fd);
260       printf("Sent Data: %d bytes\n", result);
261
262       if (result < 0)
263         return -1;
264
265       control = read_responseFrame(fd);
266       resetTimeouts();
267       switch (control)
268       {
269       case RR_0:
270         printf("Received RR_0\n");
271         break;
272       case RR_1:
273         printf("Received RR_1\n");
274         break;
275       case REJ_0:
276         printf("Received REJ_0\n");
277         break;
278       case REJ_1:
279         printf("Received REJ_1\n");
280         break;
281       }
282
283     } while (!parseControl(control));
284
285     return result;
286   }
287
288   int llclose(int fd)
289   {
290     setPhase(close_phase);
291     if (write_suFrame(fd, C_DISC) < 0)
292       return -1;
```

```
293     printf("Sent DISC\n");
294
295     if (read_suFrame(fd, C_DISC) < 0)
296       return -1;
297     resetTimeouts();
298     printf("Received DISC\n");
299
300     if (write_suFrame(fd, C_UA) < 0)
301       return -1;
302     printf("Sent C_UA\n");
303
304     if (closeSerialPort(fd) < 0)
305       return -1;
306
307     return 0;
308   }
309
```

# Anexo 1 – receiver.h

```h
#pragma once

Joaquim Rodrigues, 14 days ago | 1 author (Joaquim Rodrigues)
typedef struct flags {
    unsigned int data_ok;
    unsigned int repeated_data;
    unsigned int escape_byte;
    unsigned int send_disc;
} flags_t;

/**
 *  Changes NR value
 *
 *  @param new_nr New NR value
 */
void setNR(unsigned int new_nr);

/**
 *  Access NR value
 *
 *  @return NR value
 */
unsigned int getNR();

/**
 * Initializes the flags_t struct with everything false.
 */
void initFlags(flags_t *flags);

/**
 *  Reads a supervision or an unumbered frame from the serial port.
 *
 *  @param fd Serial port file descriptor
 *  @param control Expected control character value
 *  @return 0 in case of success; -1 on error.
 */
int read_suFrame(int fd, unsigned char control);

/**
 *  Reads a data frame from the serial port.
 *
 *  @param fd File descriptor of the serial port
 *  @param buffer Output buffer with the data content
 *  @param flags Output argument with the error and related flags
 *  @return Number of read characters (buffer length) in case of success; -1 in case of error
 */
int read_dataFrame(int fd, unsigned char *buffer, flags_t *flags);
```

# Anexo 1 – receiver.c

```c
#include "receiver.h"       Joaquim Rodrigues, 14 days ago • Uploaded Project1
#include "../macros.h"
#include "../state_machine/statemachine.h"
#include "../alarm/alarm.h"

#include <signal.h>
#include <errno.h>
#include <stdio.h>
#include <unistd.h>

static unsigned int NR = 0;

void initFlags(flags_t *flags)
{
  flags->data_ok = FALSE;
  flags->repeated_data = FALSE;
  flags->escape_byte = FALSE;
  flags->send_disc = FALSE;
}

void setNR(unsigned int new_nr)
{
  NR = new_nr;
}

unsigned int getNR()
{
  return NR;
}

int read_suFrame(int fd, unsigned char control)
{
  int read_res, state = START;
  unsigned char byte;

  alarm(TIMEOUT_INTERVAL);
  while (state != END)
  {
    read_res = read(fd, &byte, 1);
    if (read_res < 0 && errno == EINTR)
    {
      errno = 0;
      continue;
    }
    else if (read_res < 0)
      return -1;

    state = suFrameSM(byte, control, state);
  }
```

```
50
51    alarm(0);
52
53    return 0;
54  }
55
56  int read_dataFrame(int fd, unsigned char *buffer, flags_t *flags)
57  {
58    unsigned char current_bcc2 = 0;
59    unsigned char byte;
60    unsigned int current_index = 0;
61    int read_res;
62    int state = START;
63
64    alarm(TIMEOUT_INTERVAL);
65    while (state != END)
66    {
67      read_res = read(fd, &byte, 1);
68      if (read_res < 0 && errno == EINTR)
69      {
70        errno = 0;
71        continue;
72      }
73      else if (read_res < 0)
74      {
75        return -1;
76      }
77
78      state = readSM(byte, state);
79
80      if (state == C_RCV)
81      { //Checking for repeated data
82        if ((byte == CONTROL_0 && NR == 1) ||
83            (byte == CONTROL_1 && NR == 0))
84          flags->repeated_data = TRUE;
85        else if (byte == C_DISC)
86          flags->send_disc = TRUE;
87      }
88      /*
89        When repeated data is sent by the transmitter,
90        the receiver ignores all data in the frame.
91      */
92      else if (state == DATA_LOOP && !flags->repeated_data)
93      { //Data is being received
94        if (byte == ESC)
95        { //byte used for stuffing
96          flags->escape_byte = TRUE;
97          continue;
98        }
99
100       if (flags->escape_byte == TRUE)
101       { //destuffing the byte
102         if (byte == FLAG_SOL)
103           byte = FLAG; //original byte was 0x7e
104         else if (byte == ESC_SOL)
105           byte = ESC; //original byte was 0x7d
106
107         flags->escape_byte = FALSE;
108       }
109
110       // If current byte is bcc2 would data be ok?
111       if (current_bcc2 == byte)
112         flags->data_ok = TRUE;
113       else
114         flags->data_ok = FALSE;
115
116       current_bcc2 = current_bcc2 ^ byte;
117       buffer[current_index] = byte;
118       current_index++;
119     }
120   }
121   alarm(0);
122
123   return current_index - 1;
124 }
125
```

## Anexo 1 – statemachine.h

```h
#pragma once

/**
 *  State machine to process a supervision or unumbered frame.
 *  The frame's expected control character must be passed as an argument.
 *
 *  @param byte Character to be processed
 *  @param control Expected control character value
 *  @param state State of the machine before reading byte
 *  @return State of the machine after reading byte
 */
int suFrameSM(unsigned char byte, unsigned char control, int state);

/**
 *  Receiver state machine to parse the data frame.
 *
 *  @param byte Character to be processed
 *  @param state State of the machine before reading byte
 *  @return State of the machine after reading byte
 */
int readSM(unsigned char byte, int state);


/**
 *  Transmitter state machine to parse the response frame.
 *
 *  @param byte Character to be processed
 *  @param state State of the machine before reading byte
 *  @return State of the machine after reading byte
 */
int writeSM(unsigned char byte, int state);
```

## Anexo 1 – statemachine.c

```c
#include "statemachine.h"      Joaquim Rodrigues, 14 days ago • Uploaded Project1
#include "../macros.h"

int suFrameSM(unsigned char byte, unsigned char control, int state)
{
  switch (state)
  {
  case START:
    if (byte == FLAG)
      state = FLAG_RCV;
    break;
  case FLAG_RCV:
    if (byte == FLAG)
      state = FLAG_RCV;
    else if (byte == A)
      state = A_RCV;
    else
      state = START;
    break;
  case A_RCV:
    if (byte == FLAG)
      state = FLAG_RCV;
    else if (byte == control)
      state = C_RCV;
    else
      state = START;
    break;
  case C_RCV:
    if (byte == (A ^ control))
      state = BCC_OK;
    else if (byte == FLAG)
      state = FLAG_RCV;
    else
      state = START;
    break;
  case BCC_OK:
    if (byte == FLAG)
      state = END;
    else
      state = START;
    break;
  }

  return state;
}

int readSM(unsigned char byte, int state)
{
  static unsigned char control;
  switch (state)
  {
  case START:
    if (byte == FLAG)
      state = FLAG_RCV;
    break;
  case FLAG_RCV:
    if (byte == FLAG)
      state = FLAG_RCV;
    else if (byte == A) {
      state = A_RCV;
    }
    else
      state = START;
```

```
64      break;
65    case A_RCV:
66      if (byte == FLAG) {
67        state = FLAG_RCV;
68      }
69      else if (byte == CONTROL_0 || byte == CONTROL_1 || byte == C_DISC)
70      {
71        control = byte;
72        state = C_RCV;
73      }
74      else {
75        state = START;
76      }
77      break;
78    case C_RCV:
79      if (byte == (A ^ control))
80        state = BCC_OK;
81      else if (byte == FLAG)
82        state = FLAG_RCV;
83      else
84        state = START;
85      break;
86    case BCC_OK:
87      if (byte == FLAG && control == C_DISC)
88        state = END;
89      else if (byte == FLAG)
90        state = FLAG_RCV;
91      else
92        state = DATA_LOOP;
93      break;
94    case DATA_LOOP:
95      if (byte == FLAG)
```

```
96        state = END;
97      break;
98    }
99
100   return state;
101 }
102
103 int writeSM(unsigned char byte, int state)
104 {
105   static unsigned char control;
106   switch (state)
107   {
108   case START:
109     if (byte == FLAG)
110       state = FLAG_RCV;
111     break;
112   case FLAG_RCV:
113     if (byte == FLAG)
114       state = FLAG_RCV;
115     else if (byte == A)
116       state = A_RCV;
117     else
118       state = START;
119     break;
120   case A_RCV:
121     if (byte == FLAG)
122       state = FLAG_RCV;
123     else if (byte == RR_0 || byte == RR_1 || byte == REJ_0 || byte == REJ_1)
124     {
125       control = byte;
126       state = C_RCV;
127     }
```

```
128       else
129         state = START;
130       break;
131     case C_RCV:
132       if (byte == (A ^ control))
133         state = BCC_OK;
134       else if (byte == FLAG)
135         state = FLAG_RCV;
136       else
137         state = START;
138       break;
139     case BCC_OK:
140       if (byte == FLAG)
141         state = END;
142       else
143         state = START;
144       break;
145     }
146
147     return state;
148   }
```

## Anexo 1 – transmitter.h

```c
#pragma once

/**
 *  Writes a supervision or an unumbered frame to the serial port.
 *
 *  @param fd Serial port file descriptor
 *  @param control Expected control character value
 *  @return 0 in case of success; -1 on error.
 */
int write_suFrame(int fd, unsigned char control);

/**
 *  Creates the message to send in the data frame based on the content
 *  of buffer.
 *
 *  @param buffer Array containing the data
 *  @param length Length of the array
 *  @return 0 in case of success; -1 on error
 */
int parseMessage(unsigned char *buffer, int length);

/**
 *  Sends the message through the serial port.
 *
 *  @param fd File descriptor of the serial port
 *  @return number of characters written
 */
int sendMessage(int fd);
```

```c

/**
 *  Reads the response to a data frame.
 *
 *  @param fd File descriptor of the serial port
 *  @return Control character read
 */
int read_responseFrame(int fd);

/**
 *  Parses the control character (RR or REJ) read from the response frame.
 *
 *  @param control Control character received
 *  @return 1 when RR was received, 0 when REJ was received
 */
int parseControl(unsigned char control);
```

# Anexo 1 – transmitter.c

```c
#include "transmitter.h"
#include "../macros.h"
#include "../state_machine/statemachine.h"

#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

static unsigned char *msg;
static unsigned int msg_length;
static unsigned int NS = 0;

int write_suFrame(int fd, unsigned char control)
{
  unsigned char set[SU_FRAME_SIZE];

  set[F1_INDEX] = FLAG;
  set[A_INDEX] = A;
  set[C_INDEX] = control;
  set[BCC_INDEX] = A ^ control;
  set[F2_INDEX] = FLAG;

  if (write(fd, set, SU_FRAME_SIZE) < 0)
    return -1;

  return 0;
}

int parseMessage(unsigned char *buffer, int length)
{
  unsigned char bcc2 = 0;
  unsigned char control;
  unsigned int j = 0;

  msg_length = SU_FRAME_SIZE + 2 * (length) + 1;

  msg = (unsigned char *)malloc(msg_length * sizeof(unsigned char));

  if (msg == NULL)
  {
    return -1;
  }

  if (NS == 1)
  {
    control = CONTROL_1;
  }
  else
  {
    control = CONTROL_0;
  }

  msg[F1_INDEX] = FLAG;
  msg[A_INDEX] = A;
  msg[C_INDEX] = control;
  msg[BCC_INDEX] = (A ^ control);

  for (int i = 0; i < length; i++)
  {
    // Byte stuffing
    if (buffer[i] == ESC)
    {
```

```
65        msg[BCC_INDEX + i + 1 + j] = ESC;
66        j++;
67        msg[BCC_INDEX + i + 1 + j] = ESC_SOL; // ESC_SOL = ESC ^ 0x20
68      }
69      else if (buffer[i] == FLAG)
70      {
71        msg[BCC_INDEX + i + 1 + j] = ESC;
72        j++;
73        msg[BCC_INDEX + i + 1 + j] = FLAG_SOL; // FLAG_SOL = FLAG ^ 0x20
74      }
75      else
76      {
77        msg[BCC_INDEX + i + 1 + j] = buffer[i];
78      }
79
80      /*
81        Since bcc2 starts at 0, bcc2 ^ buffer[0] = buffer[0].
82        This way we avoid an irrelevant condition.
83        Additionally, as buffer remains unchanged, bcc2 wil not be affected by stuffing
84      */
85      bcc2 = bcc2 ^ buffer[i];
86    }
87
88    // BCC2 stuffing
89    if (bcc2 == ESC)
90    {
91      msg[BCC_INDEX + length + 1 + j] = ESC;
92      j++;
93      msg[BCC_INDEX + length + 1 + j] = ESC_SOL;
94    }
95    else if (bcc2 == FLAG)
96    {
```

```
97      msg[BCC_INDEX + length + 1 + j] = ESC;
98      j++;
99      msg[BCC_INDEX + length + 1 + j] = FLAG_SOL;
100   }
101   else{
102     msg[BCC_INDEX + length + 1 + j] = bcc2;
103   }
104
105   msg[BCC_INDEX + length + 2 + j] = FLAG;
106
107   msg_length = BCC_INDEX + length + 2 + j + 1;
108
109   return 0;
110 }
111
112 int sendMessage(int fd)
113 {
114   for(size_t i= 0; i < msg_length; i++){
115   }
116   return write(fd, msg, msg_length);
117 }
118
119 int read_responseFrame(int fd)
120 {
121   int state = START, read_res;
122   unsigned char byte;
123   unsigned char control;
124
125   alarm(TIMEOUT_INTERVAL);
126   while (state != END)
127   {
128     read_res = read(fd, &byte, 1);
```

```
129
130       if (read_res < 0 && errno == EINTR)
131       {
132         errno = 0;
133         continue;
134       }
135       else if (read_res < 0)
136       {
137         return -1;
138       }
139
140       state = writeSM(byte, state);
141       if (state == C_RCV)
142       {
143         control = byte;
144       }
145     }
146
147     alarm(0);
148
149     return control;
150   }
151
152   int parseControl(unsigned char control)
153   {
154     if (control == RR_0 || control == RR_1)
155     {
156       if (control == RR_0)
157         NS = 0;
158       else
159         NS = 1;
160
161       free(msg);
162       return TRUE;
163     }
164
165     return FALSE;
166   }
167
```

# Anexo 2 – Variação de FER

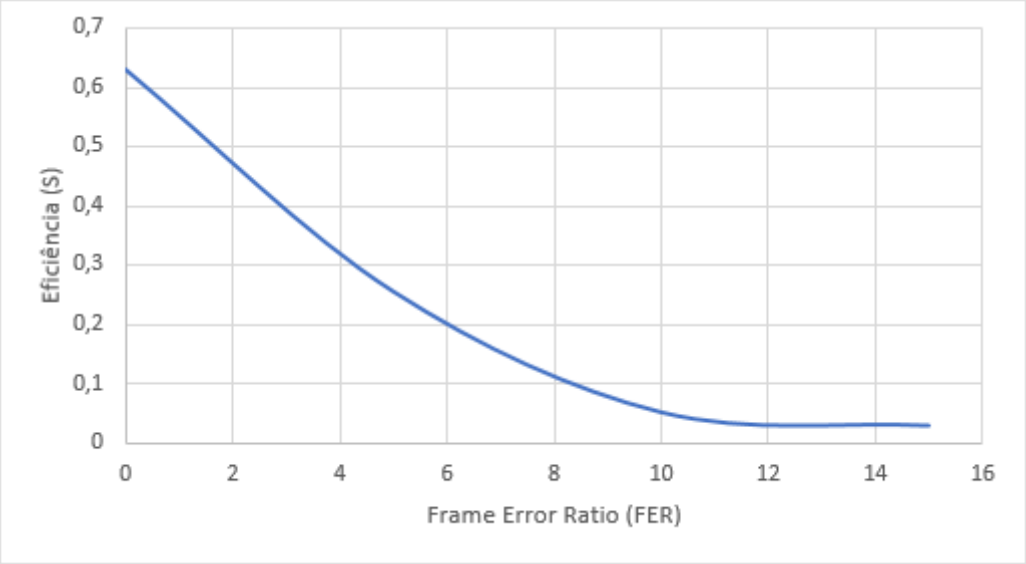



Gráfico 1 – Variação do desempenho com o aumento da FER

| | |
|---|---|
| Nº de bytes do ficheiro | 10968 |
| Nº bits do ficheiro | 87744 |
| C (bit/s) | 38400 |
| Tempo de propagação (s) | 0 |
| Tamanho da trama (bytes) | 100 |

| FER (% erro em BCC1 + % erro em BCC2) | Tempo (s) | R (Nº bits / Tempo) | S (R/C) | S médio |
|---|---|---|---|---|
| 0 + 0 | 3,61 | 24305,81717 | 0,632963989 | 0,630355621 |
| | 3,64 | 24105,49451 | 0,627747253 | |
| 5 + 5 | 7,1 | 12358,30986 | 0,321830986 | 0,256044552 |
| | 12,01 | 7305,91174 | 0,190258118 | |
| 10 + 10 | 43,69 | 2008,331426 | 0,052300298 | 0,052517562 |
| | 43,33 | 2025,017309 | 0,052734826 | |
| 15 + 15 | 76,3 | 1149,986894 | 0,029947575 | 0,029888931 |
| | 76,6 | 1145,483029 | 0,029830287 | |

# Anexo 2 – Variação do tempo de Propagação

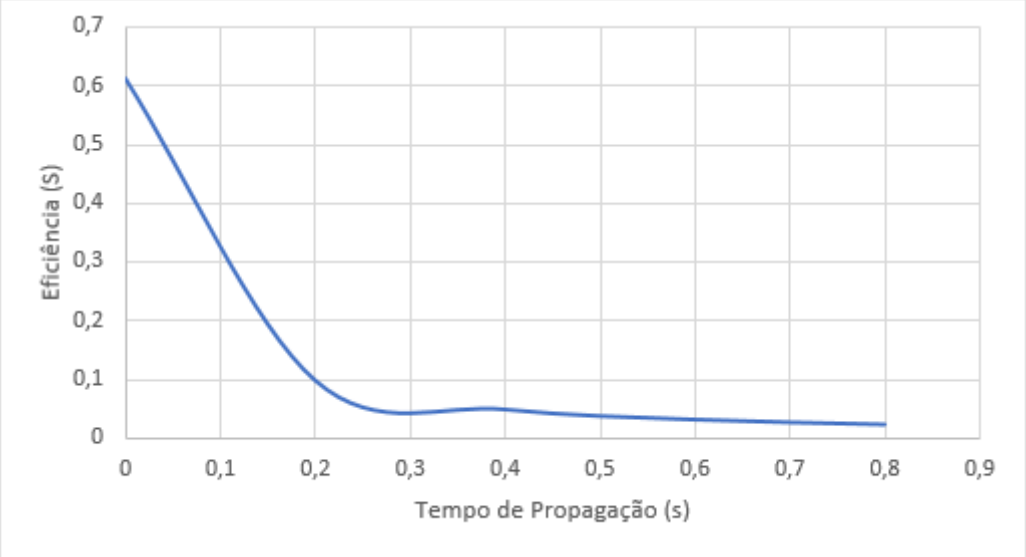



Gráfico 2 – Variação do desempenho com o aumento do tempo de Propagação

| | |
|---|---|
| Nº de bytes do ficheiro | 10968 |
| Nº bits do ficheiro | 87744 |
| C (bit/s) | 38400 |
| FER | 0 |
| Tamanho da trama (bytes) | 100 |

| Tempo de Propagação | Tempo (s) | R (Nº bits / Tempo) | S (R/C) | S médio |
|---|---|---|---|---|
| 0 | 3,77 | 23274,27056 | 0,606100796 | 0,613512354 |
| | 3,68 | 23843,47826 | 0,620923913 | |
| 0.2 | 22,89 | 3833,289646 | 0,099825251 | 0,100110341 |
| | 22,76 | 3855,184534 | 0,100395431 | |
| 0.4 | 45,23 | 1939,95136 | 0,050519567 | 0,050643031 |
| | 45,01 | 1949,433459 | 0,050766496 | |
| 0.6 | 68,03 | 1289,783919 | 0,033588123 | 0,033558552 |
| | 68,15 | 1287,512839 | 0,03352898 | |
| 0.8 | 90,53 | 969,225671 | 0,025240252 | 0,025236071 |
| | 90,56 | 968,9045936 | 0,02523189 | |

# Anexo 2 – Variação da capacidade de ligação

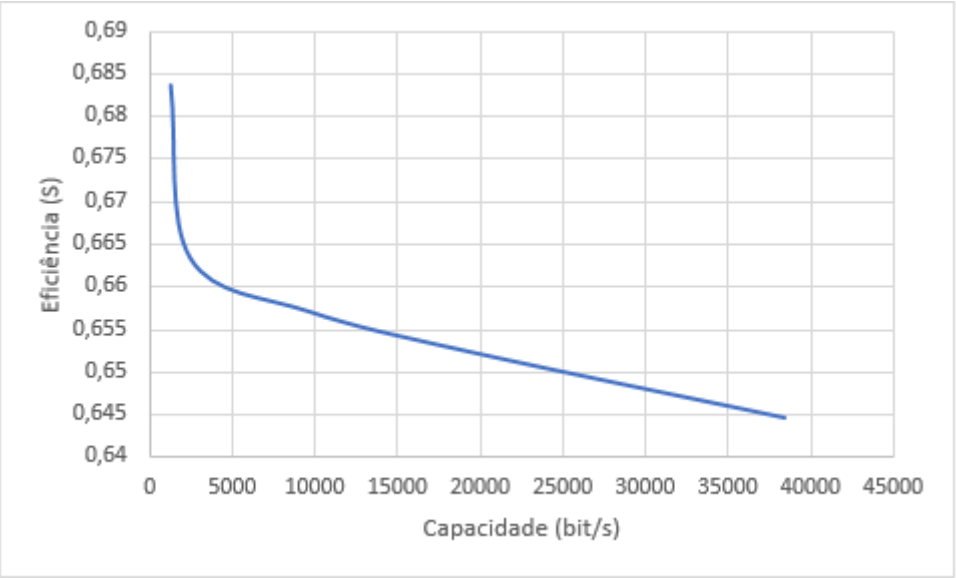



Gráfico 3 – Variação do desempenho com o aumento da capacidade da ligação

| | |
|---|---|
| Nº de bytes do ficheiro | 10968 |
| Nº bits do ficheiro | 87744 |
| Tempo de propagação | 0 |
| FER | 0 |
| Tamanho da trama (bytes) | 100 |

| C (bit/s) | Tempo (s) | R (Nº bits / Tempo) | S (R/C) | S médio |
|---|---|---|---|---|
| 1200 | 107,34 | 817,4399106 | 0,681199925 | 0,683757462 |
| | 106,54 | 823,5779989 | 0,686314999 | |
| 2400 | 55,02 | 1594,76554 | 0,664485642 | 0,663342259 |
| | 55,21 | 1589,277305 | 0,662198877 | |
| 9600 | 13,93 | 6298,923187 | 0,656137832 | 0,657082595 |
| | 13,89 | 6317,062635 | 0,658027358 | |
| 19200 | 6,98 | 12570,77364 | 0,654727794 | 0,652399459 |
| | 7,03 | 12481,36558 | 0,650071124 | |
| 38400 | 3,56 | 24647,19101 | 0,641853933 | 0,644581357 |
| | 3,53 | 24856,65722 | 0,647308782 | |

# Anexo 2 – Variação do tamanho da trama

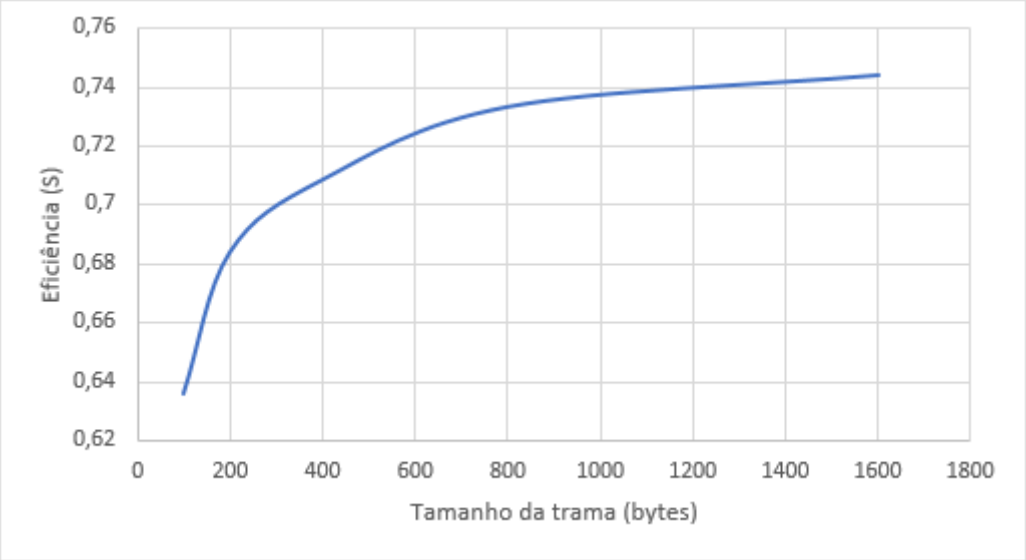



Gráfico 4 – Variação do desempenho com o aumento do tamanho da trama

| | |
|---|---|
| Nº de bytes do ficheiro | 10968 |
| Nº bits do ficheiro | 87744 |
| C (bit/s) | 38400 |
| FER | 0 |
| Tempo de Propagação | 0 |

| Tamanho da trama (bytes) | Tempo (s) | R (Nº bits / Tempo) | S (R/C) | S médio |
|---|---|---|---|---|
| 100 | 3,49 | 25141,54728 | 0,654727794 | 0,636147681 |
| | 3,7 | 23714,59459 | 0,617567568 | |
| 200 | 3,35 | 26192,23881 | 0,682089552 | 0,684137869 |
| | 3,33 | 26349,54955 | 0,686186186 | |
| 400 | 3,29 | 26669,90881 | 0,694528875 | 0,708815071 |
| | 3,16 | 27767,08861 | 0,723101266 | |
| 800 | 3,13 | 28033,22684 | 0,730031949 | 0,733564362 |
| | 3,1 | 28304,51613 | 0,737096774 | |
| 1600 | 3,05 | 28768,52459 | 0,749180328 | 0,744331264 |
| | 3,09 | 28396,1165 | 0,739482201 | |