Ligação de Dados

1º Trabalho Laboratorial

Redes de Computadores

Mestrado Integrado em Engenharia Informática e Computação

João Pedro Pinheiro de Lacerda Campos up201704982@fe.up.pt

Nuno Miguel Teixeira Cardoso up201706162@fe.up.pt

Turma 1

2019/2020

Conteúdo

1 Sumário

Este projeto surgiu como trabalho prático da unidade curricular de Redes de Computadores (RCOM) do primeiro semestre do terceiro ano. Neste projeto foram desenvolvidos protocolos de transferência de dados, tendo por base mecanismos lecionados na unidade curricular referida.

O trabalho foi terminado com sucesso, sendo possível a transferência segura de dados entre computadores, por uma porta de série, sem perdas de informação.

2 Introdução

O principal objetivo do desenvolvimento deste projeto foi a implementação de um protocolo de transferência de dados recorrendo à porta de série, que possibilitasse a comunicação entre dois computadores. A porta de série é um dos mecanismos mais básicos de ligação, permitindo-nos compreender como é feita a ligação a níveis mais altos. No entanto, esta comunicação está sujeita a vários erros e falhas de ligação, o que levou à implementação de protocolos específicos de controlo de erros.

Neste relatório será apresentada a estrutura do protocolo e do código, como também todas as decisões feitas ao longo do trabalho. Deste modo, o relatório serve de documentação ao projeto apresentado e, por isso, segue a seguinte estrutura:

- **Introdução:** identificação dos objetivos e descrição da lógica do relatório;
- **Arquitetura:** descrição da interface do utilizador e divisão por camadas;
- **Estrutura do código:** indicação das principais estruturas de dados utilizadas, principais funções e sua relação com a arquitetura;
- **Casos de uso principais:** casos de aplicação do projeto e sequências de chamadas a funções;
- **Protocolo:** descrição dos protocolos de ligação lógica e de aplicação, tendo em vista a divisão por camadas;
- **Validação:** explicação dos testes realizados com apresentação quantificada de resultados;
- **Eficiência do protocolo:** caracterização estatística e possíveis aspetos a melhorar;
- **Conclusões:** reflexões finais.

3 Arquitetura

O nosso projeto foi construído com uma estrutura de camadas em mente. Existem duas camadas independentes, a da aplicação e a da ligação de dados. A independência destas camadas permite facilidade no tratamento de dados. A camada da aplicação apenas necessita de se preocupar com a utilização e gestão dos dados, enquanto que a camada de ligação tem de se preocupar com a transferência destes.

Na camada da aplicação, na parte do transmissor, é feita a leitura do ficheiro, em fragmentos, que depois são colocados em pacotes. Estes pacotes são depois passados à camada de ligação. Na parte do recetor, este recebe os dados em pacotes e escreve a informação necessária para o ficheiro a ser copiado.

Na camada de ligação de dados, encontra-se uma série de funções que estabelecem, executam e terminam a ligação através da porta de série. Nestas funções, existe controlo de erros utilizando *block check character* (BCC) e *stuffing*.

O programa usa uma interface de texto no terminal. O primeiro argumento passado à função *main()*, por parte do utilizador, corresponde ao nome do programa. O segundo argumento define se o programa a correr é o transmissor ou o recetor e o terceiro argumento estabelece qual a porta utilizada na comunicação (exemplo de transmissor a correr para uma comunicação estabelecida para a porta de série COM1: "./app transmitter /dev/ttyS0"). Além dos argumentos passados, o transmissor recebe um pedido para introduzir o nome do ficheiro a ser transferido para o recetor.

4 Estrutura do código

4.1 Ligação Lógica

As principais funções desta camada são:

```
int llopen(int port, int status);
int llwrite(int port, unsigned char* packet, int length);
int llread(int port, unsigned char* buffer);
int llclose(int port, int status);
```

Figura 1 - Funções da camada de ligação lógica

A função *llopen()* começa por abrir a porta *port* e estabelecer as *flags* necessárias à comunicação. A seguir estabelece a ligação entre os dois computadores, tendo um argumento, *status*, que assegura a divisão entre transmissor e recetor e define o modo de proceder de cada um.

A função *llwrite()* escreve para a porta de série, *port*, o *array packet*, com tamanho *length*. Esta função é chamada ciclicamente pelo transmissor para enviar dados ao recetor e permanecer à espera de respostas de aceitação. Caso a mensagem de aceitação não seja recebida, a função *llwrite()* tem ativado um temporizador que faz com que o ciclo de envio e receção se repita por um número máximo de vezes. Por outro lado, caso a mensagem recebida seja de rejeição, a função encarrega-se de reenviar a informação ao recetor, antecipando assim o temporizador e não incrementado o número máximo de tentativas de emissão. É nesta função que, através de uma função auxiliar (*write_i*), se executa *stuffing*.

A função *llread()* lê bytes da porta de série *port* e coloca a trama no array *buffer*. Esta função é chamada ciclicamente pelo recetor para receber os dados enviados pelo transmissor e responder com a respetiva mensagem de aceitação ou de rejeição, conforme os dados tenham boa informação ou informação corrompida. É nesta função que, através de uma função auxiliar (*read_i*), se executa *destuffing*.

Na função *llclose(),* o transmissor avisa o recetor que terminou a transferência de dados. Ambos restabelecem as definições padrão da porta e fecham-na antes de terminarem.

Para armazenar e executar a gestão dos dados produzidos foram criadas as seguintes estruturas de dados:

```
//Datalinker struct
struct linkLayer {
    char port[11];
    int baudRate;
    unsigned int sequenceNumber;
    unsigned int timeout;
    unsigned int numTransmissions;
};
```

Struct linkLayer, onde foram armazenados os seguintes valores: *string* que representa a porta de série (ex.: /dev/ttyS0), a taxa de transmissão, o número de sequência (0 ou 1), tempo do temporizador em segundos e número máximo de transmissões, por esta ordem.

Figura 2 - Struct linkLayer

```
enum state {
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    BCC_OK,
    FINISH
};
```

```
enum dataState {
    START_I,
    FLAG_RCV_I,
    A_RCV_I,
    C_RCV_I,
    BCC1_OK_I,
    DATA_RCV_I,
    FLAG2_RCV_I,
    FINISH_I,
    ESCAPE_RCV_I
};
```

Enum state e *enum dataState*, usados para a construção de máquinas de estados de receção, respetivamente para tramas de supervisão e não numeradas, e tramas de informação.

Figura 3 - Enums state and dataState

4.2 Aplicação

As principais funções que figuram na camada da Aplicação são:

```
void setup(int argc, char** argv, appLayer *application, int *port);
int transmitter(appLayer *application);
int receiver(appLayer *application);
```

Figura 4 - Funções da camada de aplicação: application.h

```
void transmitter_packets(int fd_file, ctrl_packet* start_packet, ctrl_packet* end_packet,
                         data_packet* data_packet, char *file_to_send);
void receiver_packets(ctrl_packet* start_packet, ctrl_packet* end_packet,
                      data_packet* data_packet);
void packet_to_array(void* packet_void_ptr, char* buffer);
void array_to_packet(void *packet_void_ptr, char *buffer);
```

Figura 4 - Funções da camada de aplicação: packet.h

A função *setup()* verifica os argumentos passados à função *main()* por parte do utilizador e prepara a *struct appLayer*, apresentada a seguir, com os devidos valores.

```
//Application struct
typedef struct appLayer {
    int fd_port;
    int status;
    int file_size;
} appLayer;
```

fd_port: Descritor da porta de série;
status: Valor que indica se é o transmissor ou o rector.

Figura 5 - Struct appLayer

A função *trasmitter()* executa todas as funções a serem efetuadas exclusivamente pelo computador transmissor, enquanto que a função *receiver()* executa todas as funções a serem efetuadas exclusivamente pelo computador recetor. Estas funções por sua vez, de modo a conseguirem cumprir com os seus objetivos, chamam as funções do ficheiro auxiliar *packet.c*, que assegura o tratamento da informação em pacotes de dados, e as funções do protocolo de ligação lógica.

As funções *transmitter_packets()* e *receiver_packets()* preparam os pacotes de dados a serem enviados e recebidos respetivamente, através da alocação de memória para o efeito. Para melhor gestão dos dados dos pacotes criamos três *structs*:

```
typedef struct data_packet {
    unsigned char control;          // = 1 for data
    unsigned char sequence_number;  // number of data_packet
    unsigned char nr_bytes2;
    unsigned char nr_bytes1;  // K = 256 * nr_bytes2 + nr_bytes1
    unsigned char data[MAX_DATA_SIZE];      // data with K bytes
} data_packet;

typedef struct tlv_packet {
    unsigned char type;    /* type 0 - size of file
                              type 1 - name of file */
    unsigned char length;  // size of value
    char* value;  // value
} tlv_packet;

typedef struct ctrl_packet {
    unsigned char control;     /* 2 - start
                                  3 - end   */
    tlv_packet size;  // size of file
    tlv_packet name;  // name of file
} ctrl_packet;
```

data_packet para informação relativa a pacotes de dados;

tlv_packet para cada uma das sequências tipo, comprimento, valor de um pacote de controlo;

cntrl_packet para informação relativa a pacotes de controlo.

Figura 6 - Structs relativas a pacotes de dados

As funções *packet_to_array()* e *array_to_packet()* fazem, respetivamente, a conversão dos dados, de pacotes (*structs* mencionadas anteriormente) para *char arrays* a serem enviados pela função *llwrite()*, e de *char arrays* lidos pela função *llread()* para pacotes.

5 Casos de uso principais

O presente trabalho tem como principal utilização a transmissão de ficheiros de um computador para outro, recorrendo à porta de série. Outros casos mais específicos incluem:

- utilização de uma interface própria para a escolha do ficheiro a enviar por parte do utilizador;
- configuração de uma ligação entre dois computadores;
  - *llopen() -> open_port() -> set_flags();*
- estabelecimento dessa ligação, com mensagens SET e UA;
  - transmitter: *llopen() -> sendStablishTramas() -> write_set()* e *read_ua();*
  - receiver: *llopen() -> sendStablishTramas() -> read_set()* e *write_ua().*
- abertura e leitura de um ficheiro de *input*, para posterior submissão do mesmo, por parte do transmissor;
  - *llwrite() -> write_i()* e *read_rr().*
- receção de pacotes de dados e consequente exportação para um ficheiro, por parte do recetor;
  - *llread() -> read_i()* e *write_rr() / write_rej().*
- abertura e fecho de todos os ficheiros utilizados na comunicação;
- terminação segura da ligação, com mensagens DISC e UA.
  - transmitter: *llclose() -> write_disc()*, *read_disc()* e *write_ua();*
  - receiver: *llclose() -> read_disc()*, *write_disc()* e *read_ua().*

6 Protocolo

6.1 Ligação Lógica

O protocolo de ligação lógica funciona como um serviço seguro, eficaz e fiável de transferência de dados entre dois sistemas, conectados por uma porta de série. Nele encontram-se todos os mecanismos de correção e verificação de erros, bem como o sistema responsável pela passagem correta da informação. Tendo em vista a divisão por camadas, o protocolo de ligação lógica pertence à *Data Link Layer*.

A nível funcional, o protocolo de ligação lógica possui as seguintes funções com as suas respetivas utilidades:

- A função *llopen()*, responsável pela abertura da porta de série e alteração das suas configurações, guardando as configurações antigas, de modo a serem repostas no fim da execução do programa. Além disso, esta função permite, através dos seus argumentos, a distinção entre código a ser executado pelo transmissor e a ser executado pelo recetor. Assim, destaca as funções *write_set()* e *read_ua()* para o transmissor e as funções *read_set()* e *write_ua()* para o recetor, que se encarregam do estabelecimento

da ligação através de mensagens de Supervisão e Não Numeradas. O transmissor envia uma trama SET e fica à espera da receção de uma trama UA, enquanto que o recetor executa o processo inverso. Para garantir o bom funcionamento de todos estes mecanismos, no início da função *llopen()* é instalado um *timeout_handler()* que se ocupa do tratamento de situações em que o tempo máximo de espera é excedido;

- A função ***llwrite()***, responsável pela escrita de dados por parte do transmissor. Através de um ciclo *while()*, quebrado apenas quando é atingido o número máximo de transmissões ou quando a trama de confirmação é recebida, a função *llwrite()* escreve para o *input* da porta de série *buffers* de dados que lhe são passados como argumento. Ocorrem portanto chamadas cíclicas à função auxiliar *write_i()*, responsável por criar a trama I, executar *stuffing* e escrever as tramas. Dentro do mesmo ciclo, é ainda chamada a função *read_rr()*, que espera pela receção de uma trama RR ou REJ. Caso não haja qualquer tipo de receção, o temporizador encarrega-se de desbloquer o *read()* e força o reinício do ciclo. Caso seja recebida uma trama RR, o ciclo é quebrado. Caso seja recebida uma trama REJ, a função *read_rr()* encarrega-se de retornar um valor que indica rejeição dos dados, forçando o reinício do ciclo e antecipando o timeout;
- A função ***llread()***, responsável pela leitura de dados por parte do recetor. Esta função chama a auxiliar *read_i()* que através de um ciclo *while()* lê *byte* a *byte* as tramas I que lhe são enviadas pelo transmissor. Estes *bytes* passam por uma máquina de estados que assegura a correta verificação da trama e seguinte resposta. Caso o cabeçalho da trama I recebida contenha erros, a trama é inteiramente descartada. Por outro lado, se o cabeçalho não contiver erros, a análise dos dados e a verificação da possível duplicação da trama é que definem se os dados são descartados ou não. Se a trama for um duplicado, os dados são descartados, mas a função *read_i()* indica que será enviada uma trama RR pela função *write_rr()*. Caso seja uma trama nova e os dados estejam certos, por confirmação do BCC2, os dados serão aceites e será chamada a função *write_rr()*. Contudo se for uma trama nova e os dados estejam errados, será chamada a função *write_rej()* que enviará uma trama REJ;
- A função ***llclose()***, responsável pela terminação da ligação e reposicionamento das configurações iniciais da porta de série. O argumento *status* define qual dos computadores está a executar a função: o transmissor ou o recetor. O transmissor executa um ciclo *while()* com *timeout*, onde envia uma trama DISC pela função *write_disc()* e espera a receção de uma trama DISC com a função *read_disc()*. De seguida, após a receção de DISC, envia uma trama UA pela função *write_ua()*. O recetor recebe a trama DISC pela função *read_disc()* e implementa outro ciclo *while()* com *timeout* onde escreve uma trama DISC pela função *write_disc()* e espera a receção de uma trama UA com a função *read_ua()*. Por fim, ambos chamam a função *cleanup()* que restaura as configurações iniciais e fecha a porta de série.

Em qualquer uma das funções mencionadas, cujo protótipo se inicia com *write* ou *read*, são executadas chamadas ao sistema, *write(int fd, const void *buf, size_t nbytes)* e *read(int fd, void *buf, size_t nbytes)*, para escrever e ler da porta de série, respetivamente. Além disso, o argumento *status* serve para fazer a distinção interna no código entre transmissor e recetor.

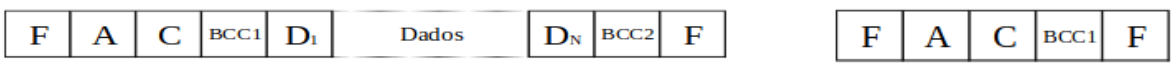


Figura 7 - Estrutura de tramas de Informação e Supervisão, respetivamente

6.2 Aplicação

O protocolo de aplicação serve-se do protocolo de ligação lógica e pertence à *Application Layer*. A função *main()*, presente nesta camada, encarrega-se de chamar sucessivamente as funções constituintes do protocolo de ligação, à medida que é necessário. Inicialmente, é chamada a função *setup()* que se encarrega da verificação dos argumentos passados à *main()* pelo utilizador, incluindo o argumento *status* que faz a distinção entre transmissor e recetor. De seguida, é chamada a função *llopen()*, a ser executada pelo protocolo de ligação lógica. Após a execução dessa função, o programa divide-se em transmissor e recetor através de um *if… else…* É então requerido ao utilizador que introduza o nome do ficheiro que pretende submeter. A Aplicação, do lado do transmissor, encarrega-se de abrir o ficheiro em modo de leitura e, posteriormente, chama a função *transmitter_packets()* que prepara três *structs*: *cntrl_packet start_packet, cntrl_packet end_packet* e *data_packet data_packet*, correspondentes aos pacotes de controlo de *START* e *END* e ao pacote de dados base. Nos pacotes de controlo são introduzidos os valores do nome e do tamanho do ficheiro.

Concluído este processo, a Aplicação chama a função *llwrite()* para enviar o pacote de controlo de *START*, indicando ao recetor que irá iniciar a transmissão do ficheiro. Ciclicamente, de seguida, faz-se a leitura de parte do ficheiro, com um tamanho *FRAG_SIZE* = 512 bits, passa-se a informação lida para o bloco de dados de um pacote através da chamada à função *memcpy()*, converte-se um pacote num *array buffer* com a função *packet_to_array()* e chama-se a função *llwrite()* do protocolo de ligação que envia o *buffer*. O ciclo repete-se enquanto o ficheiro não chegar ao fim. Após o envio do ficheiro, este é fechado. Segue-se a submissão do pacote de controlo de *END.*

Simultaneamente, do lado do recetor é chamada a função *receiver_packets()* com objetivo semelhante ao da função *transmitter_packets()* do lado do transmissor. De seguida é chamada a função *llread()* para obter um *buffer* a ser convertido num pacote de controlo pela função *array_to_packet()*. Abre-se então o ficheiro de *output* e inicia-se um ciclo sobre *llread()* onde se lêem os fragmentos enviados, que são passados para as *structs* de pacotes. Depois de recebido o pacote de controlo de *END*, o ciclo encerra e o ficheiro de *output* e fechado.

O programa encerra com uma chamada a *llclose()*, deixando a terminação da ligação à responsabilidade do protocolo de ligação lógica.

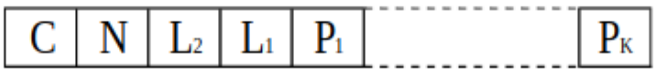


Figura 8 - Estrutura de pacotes de dados

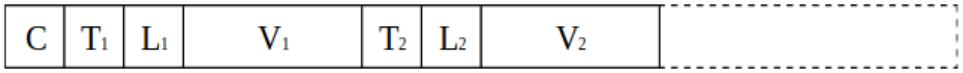
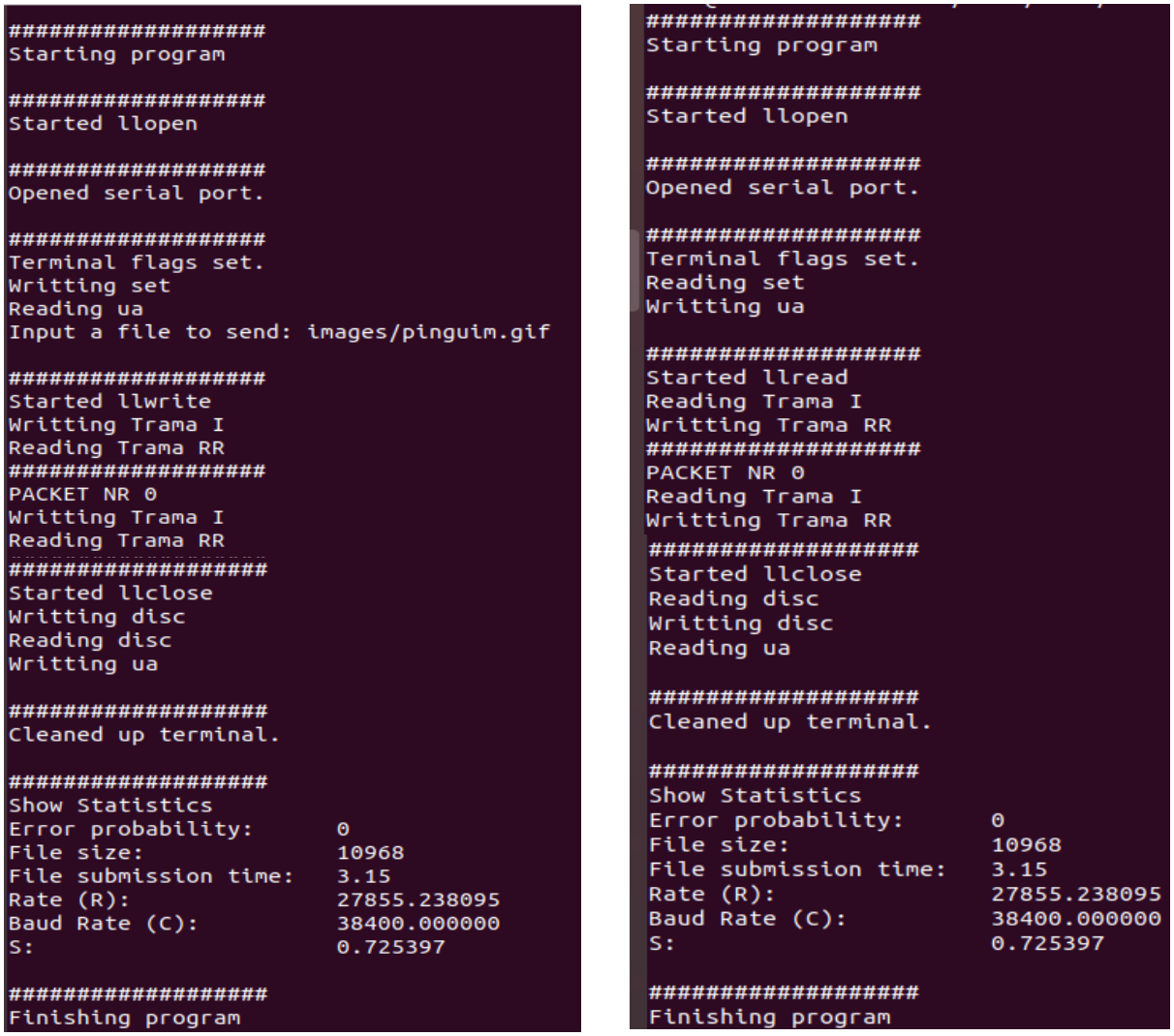


Figura 9 - Estrutura de pacotes de controlo

7 Validação

Foram realizados vários testes à execução do programa, nomeadamente transmissão normal, com introdução de ruído e com interrupção da conexão.

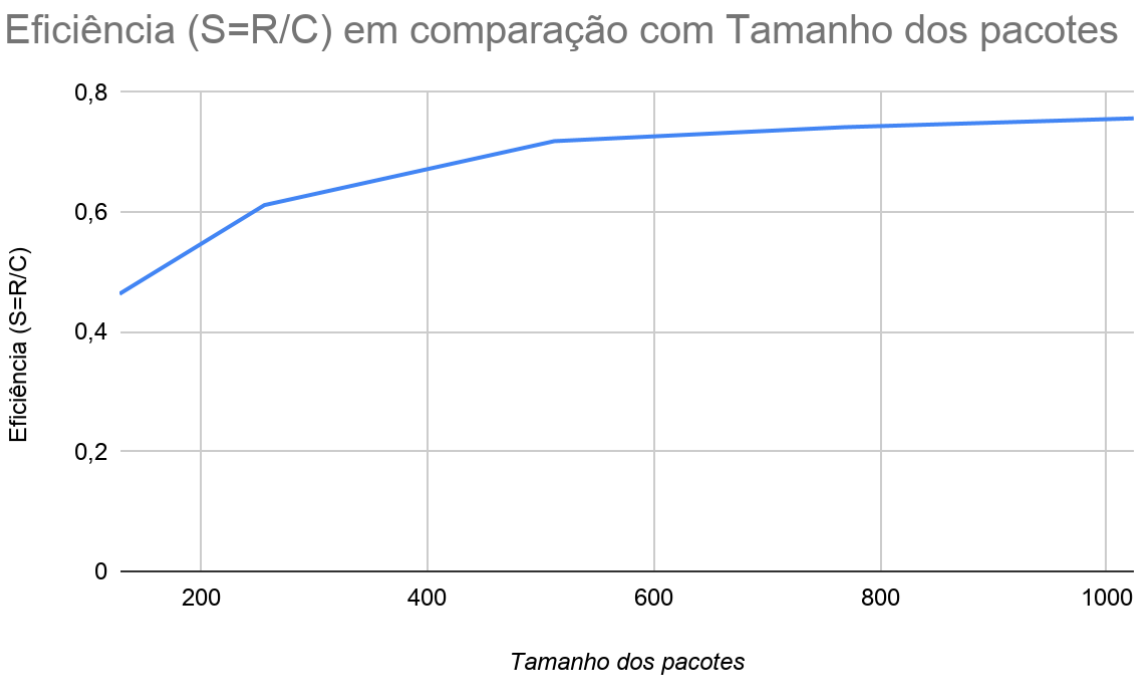


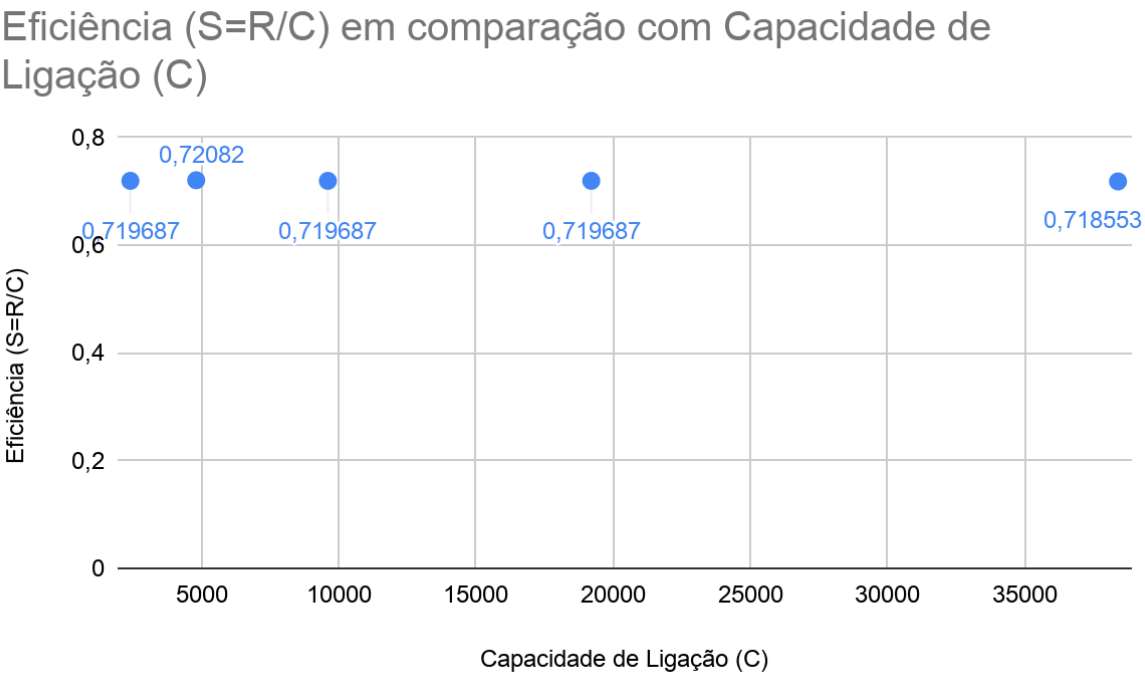
```
####################
Starting program

####################
Started llopen

####################
Opened serial port.

####################
Terminal flags set.
Writting set
Reading ua
Input a file to send: images/pinguim.gif

####################
Started llwrite
Writting Trama I
Reading Trama RR
####################
PACKET NR 0
Writting Trama I
Reading Trama RR
####################
Started llclose
Writting disc
Reading disc
Writting ua

####################
Cleaned up terminal.

####################
Show Statistics
Error probability:      0
File size:              10968
File submission time:   3.15
Rate (R):               27855.238095
Baud Rate (C):          38400.000000
S:                      0.725397

####################
Finishing program
```

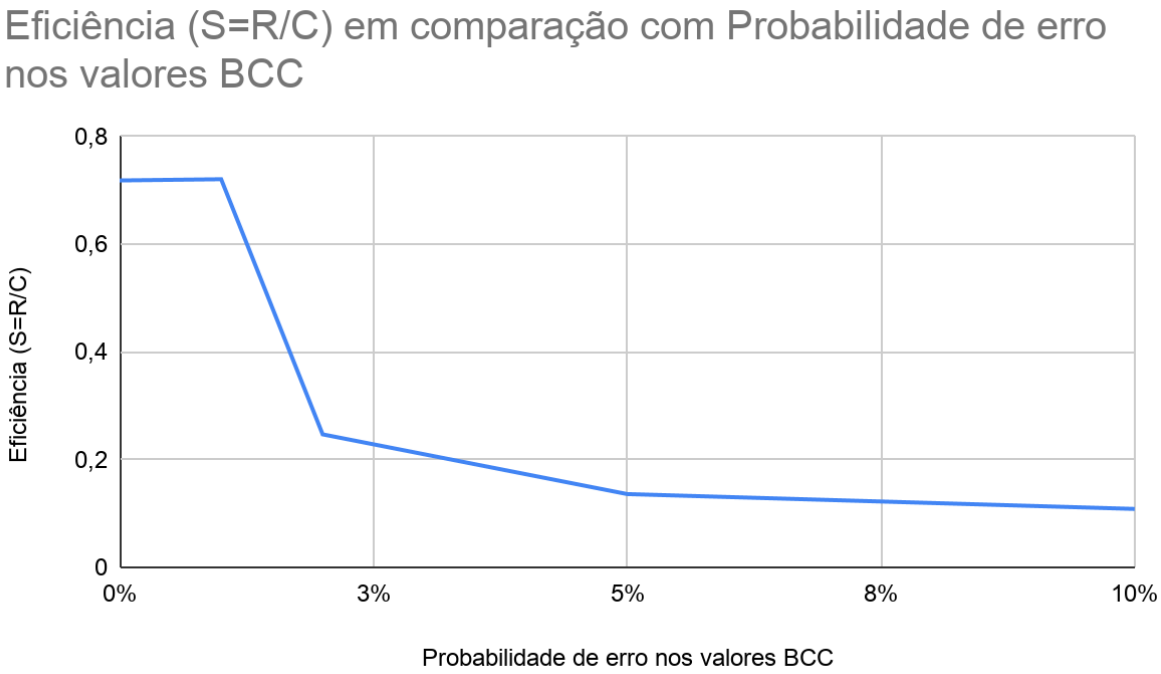
```
####################
Starting program

####################
Started llopen

####################
Opened serial port.

####################
Terminal flags set.
Reading set
Writting ua

####################
Started llread
Reading Trama I
Writting Trama RR
####################
PACKET NR 0
Reading Trama I
Writting Trama RR
####################
Started llclose
Reading disc
Writting disc
Reading ua

####################
Cleaned up terminal.

####################
Show Statistics
Error probability:      0
File size:              10968
File submission time:   3.15
Rate (R):               27855.238095
Baud Rate (C):          38400.000000
S:                      0.725397

####################
Finishing program
```

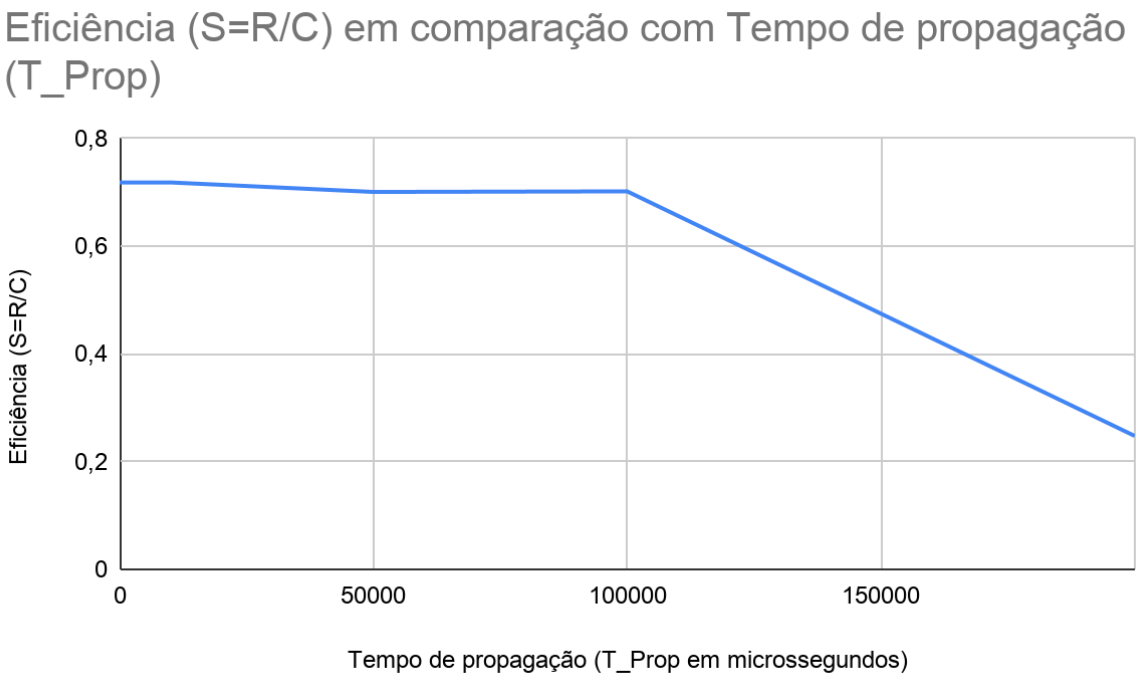
Figura 10 - Execução do programa transmissor (esquerda) e recetor (direita).

8 Eficiência









Ver anexo II para melhor detalhe.

9 Conclusões

O trabalho proposto tem como tema o protocolo de ligação de dados, através da criação de um serviço eficiente e fiável de transmissão entre dois computadores através de um cabo de série.

Surge ainda como elemento educacional, o termo independência entre camadas, que cria a distinção entre camada de Aplicação e camada de Ligação Lógica. A camada de Aplicação é responsável pela criação e gestão de dados em pacotes, não tendo qualquer tipo de conhecimento relativamente ao modo de transmissão destes. A camada de Ligação Lógica não necessita do conhecimento de existência de pacotes, apenas terá de fornecer uma transmissão de dados segura e sem erros.

Em conclusão, todos os objetivos foram cumpridos, o trabalho foi concluído com sucesso e o nosso conhecimento do tema aumentou significativamente.

Anexo I

application.h:

```
/* RCOM Laboratorial Work
 * Joao Campos and Nuno Cardoso
 * Application header file
 * RS-232 Serial Port
 *
 * 07/10/2019
 */

#include "packet.h"

#define START_SIZE      (5 + start_packet.size.length +
start_packet.name.length)
#define END_SIZE        (5 + end_packet.size.length +
end_packet.name.length)
#define DATA_SIZE       (4 + data_packet.nr_bytes2*256 +
data_packet.nr_bytes1)
#define LTZ_RET(n)      if((n) < 0){ return -1;}

//Application struct
typedef struct appLayer {
    int fd_port;
    int status;
    int file_size;
} appLayer;


void setup(int argc, char** argv, appLayer *application, int *port);
int transmitter(appLayer *application);
int receiver(appLayer *application);
```

application.c:

```c
/* RCOM Laboratorial Work
 * Joao Campos and Nuno Cardoso
 * Application main file
 * RS-232 Serial Port
 *
 * 07/10/2019
 */

#include "application.h"

//Counting time
clock_t start, end;
struct tms t;
long ticks;

int main(int argc, char **argv) {
  message("Starting program");
  // Validate arguments
  int port;
  appLayer application;
  setup(argc, argv, &application, &port);

  // Stablish communication
  message("Started llopen");
  application.fd_port = llopen(port, application.status);
  LTZ_RET(application.fd_port)

  //Efficiency stuff
  srand(time(NULL));
  ticks = sysconf(_SC_CLK_TCK);

  // Main Communication
  if (application.status == TRANSMITTER) {
    LTZ_RET(transmitter(&application))
  } else {
    LTZ_RET(receiver(&application))
  }

  // Finish counting time
  end = times(&t);

  // Finish communication
```

```
  message("Started llclose");
  if (llclose(application.fd_port, application.status) < 0) {
    perror("llclose");
    return -1;
  }

  // Show statistics
  message("Show Statistics");
  printf("Error probability: \t1/%d\n", ERROR_PROB);
  printf("File size: \t\t%d\n", application.file_size);
  printf("File submission time:  \t%4.3f\n",(double)(end-start)/ticks);
  printf("Rate (R): \t\t%f\n", (application.file_size*8)/((double)(end-
start)/ticks));
  printf("Baud Rate (C):  \t%f\n", BAUD_VALUE);
  printf("S: \t\t\t%f\n", ((application.file_size*8)/((double)(end-
start)/ticks))/38400.0);

  message("Finishing program");
  return 0;
}

void setup(int argc, char **argv, appLayer *application, int *port) {
  if ((argc != 3) || ((strcmp("/dev/ttyS0", argv[2]) != 0) &&
                      (strcmp("/dev/ttyS1", argv[2]) != 0) &&
                      (strcmp("/dev/ttyS2", argv[2]) != 0) &&
                      (strcmp("/dev/ttyS3", argv[2]) != 0) &&
                      (strcmp("/dev/ttyS4", argv[2]) != 0))) {
    printf("Usage:\tnserial transmitter|receiver SerialPort\n\tex:
nserial "
           "transmitter /dev/ttyS0\n");
    exit(1);
  }

  // Application struct
  if (strcmp(argv[1], "transmitter") == 0)
    application->status = TRANSMITTER;
  else
    application->status = RECEIVER;

  // Port
  if ((strcmp("/dev/ttyS0", argv[2]) == 0))
    *port = COM1;
  else if ((strcmp("/dev/ttyS1", argv[2]) == 0))
```

```
    *port = COM2;
  else if ((strcmp("/dev/ttyS2", argv[2]) == 0))
    *port = COM3;
  else if ((strcmp("/dev/ttyS3", argv[2]) == 0))
    *port = COM4;
  else
    *port = COM5;
}

int transmitter(appLayer *application) {
  char file_to_send[255];
  printf("Input a file to send: ");
  scanf("%s", file_to_send);

  // Open file
  int fd_file = open(file_to_send, O_RDONLY | O_NONBLOCK);
  if (fd_file < 0) {
    perror("Opening File");
    return -1;
  }

  // Create packets
  ctrl_packet start_packet, end_packet;
  data_packet data_packet;
  int packet_nr = 0;
  transmitter_packets(fd_file, &start_packet, &end_packet,
&data_packet, file_to_send);
  application->file_size = atoi(start_packet.size.value);

  // Fragments of file to send
  unsigned char fragment[FRAG_SIZE];
  unsigned char buffer[MAX_DATA_SIZE];
  int numbytes, size_packet, n_chars_written;

  // Write information
  message("Started llwrite");

  //Start counting time
  start = times(&t);

  // Send START packet
  memset(buffer, '\0', MAX_DATA_SIZE);
  packet_to_array(&start_packet, buffer);
```

```
  n_chars_written = llwrite(application->fd_port, buffer, START_SIZE);
  LTZ_RET(n_chars_written)

  // Read fragments and send them one by one
  memset(buffer, '\0', MAX_DATA_SIZE);

  while ((numbytes = read(fd_file, fragment, FRAG_SIZE)) != 0) {
    LTZ_RET(numbytes)

    // Send DATA packets
    data_packet.sequence_number = (data_packet.sequence_number + 1) %
256;
    data_packet.nr_bytes2 = numbytes / 256;
    data_packet.nr_bytes1 = numbytes % 256;
    memcpy(data_packet.data, fragment, numbytes);
    packet_to_array(&data_packet, buffer);

    message_packet(packet_nr);
    packet_nr++;
    n_chars_written = llwrite(application->fd_port, buffer, DATA_SIZE);
    LTZ_RET(n_chars_written)
    memset(buffer, '\0', MAX_DATA_SIZE);
  }
  close(fd_file);

  // Send END packet
  packet_to_array(&end_packet, buffer);
  n_chars_written = llwrite(application->fd_port, buffer, END_SIZE);
  LTZ_RET(n_chars_written)
}

int receiver(appLayer *application) {
  // RECEIVER

  // Fragments of file to read
  ctrl_packet start_packet, end_packet;
  data_packet data_packet;
  unsigned char read_buffer[MAX_DATA_SIZE];
  int n_chars_read, size, packet_nr = 0, fd_file;

  receiver_packets(&start_packet, &end_packet, &data_packet);
```

```
  // Receive information
  message("Started llread");

  // Read START packet
  memset(read_buffer, '\0', MAX_DATA_SIZE);
  n_chars_read = llread(application->fd_port, read_buffer);
  LTZ_RET(n_chars_read)
  array_to_packet(&start_packet, read_buffer);

  // Create file
  fd_file = open(start_packet.name.value,
                 O_WRONLY | O_CREAT | O_TRUNC | O_APPEND, 0664);
  LTZ_RET(fd_file)

  // Read fragments
  memset(read_buffer, '\0', MAX_DATA_SIZE);
  message_packet(packet_nr);

  //Start counting time
  start = times(&t);

  while ((n_chars_read = llread(application->fd_port, read_buffer)) !=
0) {
    LTZ_RET(n_chars_read);
    if (read_buffer[0] != 3) {
      if (n_chars_read != REJECT_DATA)
        packet_nr++;

      array_to_packet(&data_packet, read_buffer);
      write(fd_file, data_packet.data, data_packet.nr_bytes2 * 256 +
data_packet.nr_bytes1);
    } else {
      array_to_packet(&end_packet, read_buffer);
      break;
    }
    memset(read_buffer, '\0', MAX_DATA_SIZE);

    application->file_size = atoi(start_packet.size.value);

    message_packet(packet_nr);
  }

  close(fd_file);
```

```
  return 0;
}
```

packet.h:

```
#include "datalink.h"

#define CNTRL_START 2
#define CNTRL_END 3
#define FRAG_SIZE       (MAX_DATA_SIZE / 2 - 4)             // C N L1
L2 FRAG

typedef struct data_packet {
    unsigned char control;          // = 1 for data
    unsigned char sequence_number;  // number of data_packet
    unsigned char nr_bytes2;
    unsigned char nr_bytes1;  // K = 256 * nr_bytes2 + nr_bytes1
    unsigned char data[MAX_DATA_SIZE];      // data with K bytes
} data_packet;

typedef struct tlv_packet {
    unsigned char type;    /* type 0 - size of file
                    type 1 - name of file */
    unsigned char length;  // size of value
    char* value;  // value
} tlv_packet;

typedef struct ctrl_packet {
    unsigned char control;     /* 2 - start
                          3 - end   */
    tlv_packet size;  // size of file
    tlv_packet name;  // name of file
} ctrl_packet;

void transmitter_packets(int fd_file, ctrl_packet* start_packet,
ctrl_packet* end_packet, data_packet* data_packet, char *file_to_send);
void receiver_packets(ctrl_packet* start_packet, ctrl_packet*
end_packet, data_packet* data_packet);
void packet_to_array(void* packet_void_ptr, char* buffer);
void array_to_packet(void *packet_void_ptr, char *buffer);
```

packet.c:

```
#include "packet.h"

void transmitter_packets(int fd_file, ctrl_packet *start_packet,
ctrl_packet *end_packet, data_packet *data_packet, char *file_to_send)
{
  struct stat file_stat;
  if (fstat(fd_file, &file_stat) < 0) {
    message("Error reading file. Exitting.");
    exit(2);
  }

  data_packet->control = 1;
  data_packet->sequence_number = (unsigned char)255;
  data_packet->nr_bytes2 = 0;
  data_packet->nr_bytes1 = 0;

  start_packet->control = 2;
  start_packet->size.type = 0;
  start_packet->size.length = sizeof(int);
  start_packet->size.value = malloc(start_packet->size.length);
  sprintf(start_packet->size.value, "%ld", file_stat.st_size);

  start_packet->name.type = 1;
  start_packet->name.length = strlen(basename(file_to_send));
  start_packet->name.value = malloc(start_packet->name.length);
  sprintf(start_packet->name.value, "%s", basename(file_to_send));

  end_packet->control = 3;
  end_packet->size.type = 0;
  end_packet->size.length = sizeof(int);
  end_packet->size.value = malloc(end_packet->size.length);
  sprintf(end_packet->size.value, "%ld", file_stat.st_size);

  end_packet->name.type = 1;
  end_packet->name.length = strlen(basename(file_to_send));
  end_packet->name.value = malloc(end_packet->name.length);
  sprintf(end_packet->name.value, "%s", basename(file_to_send));
}

void receiver_packets(ctrl_packet *start_packet, ctrl_packet
*end_packet, data_packet *data_packet) {
```

```
  data_packet->control = 1;
  data_packet->sequence_number = (unsigned char)255;
  data_packet->nr_bytes2 = 0;
  data_packet->nr_bytes1 = 0;

  start_packet->control = 2;
  start_packet->size.type = 0;
  start_packet->size.length = 0;

  start_packet->name.type = 1;
  start_packet->name.length = 0;

  end_packet->control = 3;
  end_packet->size.type = 0;
  end_packet->size.length = 0;

  end_packet->name.type = 1;
  end_packet->name.length = 0;
}

void packet_to_array(void *packet_void_ptr, char *buffer) {
  data_packet *data_packet_ptr = (data_packet *)packet_void_ptr;
  ctrl_packet *ctrl_packet_ptr = (ctrl_packet *)packet_void_ptr;

  switch (data_packet_ptr->control) {
  // DATA PACKET
  case 1:
    buffer[0] = data_packet_ptr->control;
    buffer[1] = data_packet_ptr->sequence_number;
    buffer[2] = data_packet_ptr->nr_bytes2;
    buffer[3] = data_packet_ptr->nr_bytes1;
    memcpy((buffer + 4), data_packet_ptr->data, data_packet_ptr-
>nr_bytes2*256+data_packet_ptr->nr_bytes1);
    break;
  // START END PACKET
  case 2:
  case 3:
    buffer[0] = ctrl_packet_ptr->control;
    buffer[1] = ctrl_packet_ptr->size.type;
    buffer[2] = ctrl_packet_ptr->size.length;
    memcpy((buffer + 3), ctrl_packet_ptr->size.value, sizeof(int)+1);
```

```
    buffer[3 + ctrl_packet_ptr->size.length] = ctrl_packet_ptr-
>name.type;
    buffer[4 + ctrl_packet_ptr->size.length] = ctrl_packet_ptr-
>name.length;
    memcpy((buffer + 5 + ctrl_packet_ptr->size.length),
ctrl_packet_ptr->name.value, ctrl_packet_ptr->name.length);
    break;

  default:
    break;
  }
}

void array_to_packet(void *packet_void_ptr, char *buffer) {
  data_packet *data_packet_ptr = (data_packet *)packet_void_ptr;
  ctrl_packet *ctrl_packet_ptr = (ctrl_packet *)packet_void_ptr;

  switch (buffer[0]) {
  // DATA
  case 1:
    data_packet_ptr->control = buffer[0];
    data_packet_ptr->sequence_number = buffer[1];
    data_packet_ptr->nr_bytes2 = buffer[2];
    data_packet_ptr->nr_bytes1 = buffer[3];
    memcpy(data_packet_ptr->data, (buffer + 4), FRAG_SIZE);
    break;
  // START
  case 2:
    ctrl_packet_ptr->control = buffer[0];
    ctrl_packet_ptr->size.type = buffer[1];
    ctrl_packet_ptr->size.length = buffer[2];
    ctrl_packet_ptr->size.value = malloc(ctrl_packet_ptr->size.length);
    memcpy(ctrl_packet_ptr->size.value, (buffer + 3), buffer[2]);

    ctrl_packet_ptr->name.type = buffer[3 + buffer[2]];
    ctrl_packet_ptr->name.length = buffer[4 + buffer[2]];
    ctrl_packet_ptr->name.value = malloc(ctrl_packet_ptr->name.length);
    memcpy(ctrl_packet_ptr->name.value, buffer + 5 + buffer[2],
buffer[4 + buffer[2]]);
    break;
  // END
  case 3:
    ctrl_packet_ptr->control = buffer[0];
```

```
    ctrl_packet_ptr->size.type = buffer[1];
    ctrl_packet_ptr->size.length = buffer[2];
    ctrl_packet_ptr->size.value = malloc(ctrl_packet_ptr->size.length);
    memcpy(ctrl_packet_ptr->size.value, (buffer + 3), buffer[2]);

    ctrl_packet_ptr->name.type = buffer[3 + buffer[2]];
    ctrl_packet_ptr->name.length = buffer[4 + buffer[2]];
    ctrl_packet_ptr->name.value = malloc(ctrl_packet_ptr->name.length);
    memcpy(ctrl_packet_ptr->name.value, buffer + 5 + buffer[2],
buffer[4 + buffer[2]]);
    break;
  default:
    break;
  }
}
```

datalink.h:

```
/* RCOM Laboratorial Work
 * Joao Campos and Nuno Cardoso
 * Datalink header file
 * RS-232 Serial Port
 *
 * 07/10/2019
 */

#include <fcntl.h>
#include <libgen.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <termios.h>
#include <unistd.h>
#include <sys/times.h>
#include <time.h>

//Efficiency
#define ERROR_PROB 10
#define BAUD_VALUE 38400.0

#define BAUDRATE        B38400
#define MAX_FRAME_SIZE  512
#define MAX_DATA_SIZE   (MAX_FRAME_SIZE - 6)
#define TRANSMITTER     12
#define RECEIVER        21
#define COM1            0
#define COM2            1
#define COM3            2
#define COM4            3
#define COM5            4

// FLAGS
#define FLAG            0x7E
#define A_CMD           0x03
#define A_ANS           0x01
#define C_SET           0x03
#define C_UA            0x07
```

```
#define C_DISC          0x0B
#define C_0             0x00
#define C_1             0x40
#define C_RR0           0x05
#define C_RR1           0x85
#define C_REJ0          0x01
#define C_REJ1          0x81
#define ESCAPE          0x7D
#define STUF            0x20
#define REJECT_DATA     1

enum state {
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    BCC_OK,
    FINISH
};

enum dataState {
    START_I,
    FLAG_RCV_I,
    A_RCV_I,
    C_RCV_I,
    BCC1_OK_I,
    DATA_RCV_I,
    FLAG2_RCV_I,
    FINISH_I,
    ESCAPE_RCV_I
};

//Datalinker struct
struct linkLayer {
    char port[11];
    int baudRate;
    unsigned int sequenceNumber;
    unsigned int timeout;
    unsigned int numTransmissions;
};

//Open
int llopen(int port, int status);
```

```
//Write
int llwrite(int fd, unsigned char* packet, int length);

//Read
int llread(int fd, unsigned char* buffer);

//Close
int llclose(int fd, int status);

void message(char *message);
void message_packet(int i);
void minor_message(char *message);
int open_port(int port);
void set_flags(int fd);
void cleanup(int fd);
void timeout_handler();

int sendStablishTramas(int fd, int status);

int write_set(int fd);
void read_ua(int fd, int status);
void read_set(int fd);
int write_ua(int fd, int status);
int write_disc(int fd, int status);
void read_disc(int fd, int status);
int write_i(int fd, char *buffer, int length);
int read_i(int fd, char *buffer, int *reject);
int write_rr(int fd);
int read_rr(int fd);
int write_rej(int fd);

// Eficiency
void generate_errors(unsigned char* buffer, int i);
```

datalink.c:

```
/* RCOM Laboratorial Work
 * Joao Campos and Nuno Cardoso
 * Datalink main file
 * RS-232 Serial Port
 *
 * 07/10/2019
 */

#include "datalink.h"

//Global variables
struct linkLayer datalink;
volatile sig_atomic_t received_ua = 0;
volatile sig_atomic_t received_disc = 0;
volatile sig_atomic_t received_i = 0;
volatile sig_atomic_t n_timeouts = 0;
volatile sig_atomic_t break_read_loop = 0;
volatile sig_atomic_t timed_out = 0;
volatile sig_atomic_t nr_tramaI = 0;
volatile sig_atomic_t control_start = 1;
volatile sig_atomic_t fd_port;
struct termios oldtio;
struct termios newtio;
enum state receiving_ua_state;
enum state receiving_set_state;
enum state receiving_disc_state;
enum state receiving_rr_state;
enum dataState receiving_data_state;

int llopen(int port, int status) {
  // Open serial port
  int fd = open_port(port);
  // Check errors
  if (fd < 0)
    return fd;

  // Tramas set and ua
  int res = sendStablishTramas(fd, status);
  // Check errors
  if (res < 0)
    return res;
```

```
  return fd;
}

int llwrite(int fd, unsigned char *buffer, int length) {
  int res_i;
    //Reset flag
    received_i = 0;

    while (n_timeouts < datalink.numTransmissions) {
        if (!received_i) {
            //Write trama I
            minor_message("Writting Trama I");
            res_i = write_i(fd, buffer, length);
            alarm(datalink.timeout);

            //Read trama RR
            minor_message("Reading Trama RR");
            int rej = read_rr(fd);
            alarm(0);
            fcntl(fd, F_SETFL, ~O_NONBLOCK);

            if (rej) {
                timed_out = 0;
                continue;
            }

            //Change sequence number
            if (!timed_out)
               datalink.sequenceNumber = (datalink.sequenceNumber + 1)
% 2;
            timed_out = 0;

        } else
            break;
    }

    n_timeouts = 0;

    //Stop execution if could not send trama I after MAX_TIMEOUTS
    if (!received_i)
        return -1;

    return res_i;
```

```
}

int llread(int fd, unsigned char *buffer) {
  //Testing T_Prop
  //usleep(0);

  //Read trama I
    minor_message("Reading Trama I");
    int reject = 0;
    int data_bytes = read_i(fd, buffer, &reject);
    if (reject) {
        //Write trama REJ
        minor_message("Writting Trama REJ");
        int res_rej = write_rej(fd);
    }
    else {
        if(data_bytes == 1){
            datalink.sequenceNumber = (datalink.sequenceNumber + 1) %
2;
        }

        //Write trama RR
        minor_message("Writting Trama RR");
        int res_rr = write_rr(fd);

        //Change sequence number
        if (!timed_out)
            datalink.sequenceNumber = (datalink.sequenceNumber + 1) %
2;
        timed_out = 0;
    }

    return data_bytes;
}

int llclose(int fd, int status) {
  if (status == TRANSMITTER) {
        //Transmitter

        while (n_timeouts < datalink.numTransmissions) {
            if (!received_disc) {
                //Write disc
                minor_message("Writting disc");
```

```
                int res_disc = write_disc(fd, status);
                if (res_disc < 0)
                    return res_disc;
                alarm(datalink.timeout);

                //Read disc
                minor_message("Reading disc");
                read_disc(fd, status);
                alarm(0);
                fcntl(fd, F_SETFL, ~O_NONBLOCK);
            } else
                break;
        }

        //Stop execution if could not stablish connection after
MAX_TIMEOUTS
        if (!received_disc)
            return -1;

        //Write ua
        minor_message("Writting ua");
        int res_ua = write_ua(fd, status);
        if (res_ua < 0)
            return res_ua;

    } else {
        //Receiver

        //Read disc
        minor_message("Reading disc");
        read_disc(fd, status);

        while (n_timeouts < datalink.numTransmissions) {
            if (!received_ua) {
                //Write disc
                minor_message("Writting disc");
                int res_disc = write_disc(fd, status);
                if (res_disc < 0)
                    return res_disc;
                alarm(datalink.timeout);

                //Read ua
                minor_message("Reading ua");
```

```
                read_ua(fd, status);
                alarm(0);
                fcntl(fd, F_SETFL, ~O_NONBLOCK);
            } else
                break;
        }
    }

    cleanup(fd);

    return 1;
}

void message(char *message) {
    printf("\n####################\n%s\n", message);
}

void minor_message(char *message) {
    printf("%s\n", message);
}

void message_packet(int i) {
    printf("####################\nPACKET NR %d\n", i);
}

int open_port(int port) {
    int fd;

    if (port == 0)
        strcpy(datalink.port, "/dev/ttyS0");
    else if (port == 1)
        strcpy(datalink.port, "/dev/ttyS1");
    else if (port == 2)
        strcpy(datalink.port, "/dev/ttyS2");
    else if (port == 3)
        strcpy(datalink.port, "/dev/ttyS3");
    else strcpy(datalink.port, "/dev/ttyS4");

    fd = open(datalink.port, O_RDWR | O_NOCTTY);

    if (fd < 0)
        return fd;
```

```
    message("Opened serial port.");

    //Set flags
    set_flags(fd);

    fd_port = fd;

    return fd;
}

void set_flags(int fd) {
    if (tcgetattr(fd, &oldtio) == -1) {
        perror("tcgetattr");
        exit(-1);
    }

    bzero(&newtio, sizeof(struct termios));
    newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;
    newtio.c_lflag = 0;
    newtio.c_cc[VTIME] = 0;
    newtio.c_cc[VMIN] = 1;

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

    //Initial sequenceNumber
    datalink.sequenceNumber = 0;

    message("Terminal flags set.");
}

void timeout_handler() {
    if (receiving_ua_state != FINISH) {
        received_ua = 0;
        receiving_ua_state = START;
    } else if (receiving_disc_state != FINISH) {
        received_disc = 0;
        receiving_disc_state = START;
```

```
    } else if (receiving_data_state != FINISH_I) {
        received_i = 0;
        receiving_data_state = START_I;
    }

    fcntl(fd_port, F_SETFL, O_NONBLOCK);

    timed_out = 1;
    break_read_loop = 1;
    n_timeouts++;

    message("Timed out.");
}

int sendStablishTramas(int fd, int status) {
    //Install timeout handler
    (void)signal(SIGALRM, timeout_handler);
    datalink.numTransmissions = 5;
    datalink.timeout = 3;

    if (status == TRANSMITTER) {
        //Transmitter

        while (n_timeouts < datalink.numTransmissions) {
            if (!received_ua) {
                //Write set
                minor_message("Writting set");
                int res_set = write_set(fd);
                if (res_set < 0)
                    return res_set;
                alarm(datalink.timeout);

                //Read ua
                minor_message("Reading ua");
                read_ua(fd, status);
                alarm(0);
                fcntl(fd, F_SETFL, ~O_NONBLOCK);
            } else
                break;
        }

        //Stop execution if could not stablish connection after
MAX_TIMEOUTS
```

```
        if (!received_ua)
            return -1;

    } else {
        //Receiver

        //Read set
        minor_message("Reading set");
        read_set(fd);

        //Write ua
        minor_message("Writting ua");
        int res_ua = write_ua(fd, status);
    }

    //Reset number of timeouts and flags
    n_timeouts = 0;
    received_ua = 0;
    break_read_loop = 0;

    return 0;
}

int write_set(int fd) {
    //Create trama SET
    unsigned char set[5];
    set[0] = FLAG;
    set[1] = A_CMD;
    set[2] = C_SET;
    set[3] = A_CMD ^ C_SET;
    set[4] = FLAG;

    int res = write(fd, set, 5 * sizeof(char));

    return res;
}

void read_ua(int fd, int status) {
    unsigned char ua[MAX_FRAME_SIZE];
    int res;
    int n_bytes = 0;
    unsigned char read_char[1];
    read_char[0] = '\0';
```

```
receiving_ua_state = START;
break_read_loop = 0;

u_int8_t A;
if (status == TRANSMITTER)
    A = A_CMD;
else
    A = A_ANS;

while (!break_read_loop) {
    if (receiving_ua_state != FINISH) {
        res = read(fd, read_char, sizeof(char));
        ua[n_bytes] = read_char[0];
    }

    switch (receiving_ua_state) {
        case START: {
            if (read_char[0] == FLAG) {
                receiving_ua_state = FLAG_RCV;
                n_bytes++;
            }
            break;
        }
        case FLAG_RCV: {
            if (read_char[0] == A) {
                receiving_ua_state = A_RCV;
                n_bytes++;
            } else if (read_char[0] == FLAG) {
                n_bytes = 1;
                break;
            } else {
                receiving_ua_state = START;
                n_bytes = 0;
            }
            break;
        }
        case A_RCV: {
            if (read_char[0] == C_UA) {
                receiving_ua_state = C_RCV;
                n_bytes++;
            } else if (read_char[0] == FLAG) {
                receiving_ua_state = FLAG_RCV;
```

```
                    n_bytes = 1;
                } else {
                    receiving_ua_state = START;
                    n_bytes = 0;
                }
                break;
            }
            case C_RCV: {
                if (read_char[0] == A ^ C_UA) {
                    receiving_ua_state = BCC_OK;
                    n_bytes++;
                } else if (read_char[0] == FLAG) {
                    receiving_ua_state = FLAG_RCV;
                    n_bytes = 1;
                } else {
                    receiving_ua_state = START;
                    n_bytes = 0;
                }
                break;
            }
            case BCC_OK: {
                if (read_char[0] == FLAG) {
                    receiving_ua_state = FINISH;
                    n_bytes++;
                } else {
                    receiving_ua_state = START;
                    n_bytes = 0;
                }
                break;
            }
            case FINISH: {
                break_read_loop = 1;
                received_ua = 1;
                break;
            }
        }
    }
}

void read_set(int fd) {
    unsigned char set[MAX_FRAME_SIZE];
    unsigned char read_char[1];
    int n_bytes = 0;
```

```
int res;
int received_set = 0;
receiving_set_state = START;

// READ
while (!received_set) {
     if (receiving_set_state != FINISH) {
        res = read(fd, read_char, sizeof(char));
        set[n_bytes] = read_char[0];
     }

    switch (receiving_set_state) {
        case START: {
            if (read_char[0] == FLAG) {
                receiving_set_state = FLAG_RCV;
                n_bytes++;
            }
            break;
        }
        case FLAG_RCV: {
            if (read_char[0] == A_CMD) {
                receiving_set_state = A_RCV;
                n_bytes++;
            } else if (read_char[0] == FLAG) {
                n_bytes = 1;
                break;
            } else {
                receiving_set_state = START;
                n_bytes = 0;
            }
            break;
        }
        case A_RCV: {
            if (read_char[0] == C_SET) {
                receiving_set_state = C_RCV;
                n_bytes++;
            } else if (read_char[0] == FLAG) {
                receiving_set_state = FLAG_RCV;
                n_bytes = 1;
            } else {
                receiving_set_state = START;
                n_bytes = 0;
            }
```

```
                break;
            }
            case C_RCV: {
                if (read_char[0] == A_CMD ^ C_SET) {
                    receiving_set_state = BCC_OK;
                    n_bytes++;
                } else if (read_char[0] == FLAG) {
                    receiving_set_state = FLAG_RCV;
                    n_bytes = 1;
                } else {
                    receiving_set_state = START;
                    n_bytes = 0;
                }
                break;
            }
            case BCC_OK: {
                if (read_char[0] == FLAG) {
                    receiving_set_state = FINISH;
                    n_bytes++;
                } else {
                    receiving_set_state = START;
                    n_bytes = 0;
                }
                break;
            }
            case FINISH: {
                received_set = 1;
                break;
            }
        }
    }
}

int write_ua(int fd, int status) {
    //Create trama UA
    unsigned char ua[5];
    ua[0] = FLAG;
    if (status == TRANSMITTER)
        ua[1] = A_ANS;
    else
        ua[1] = A_CMD;
    ua[2] = C_UA;
    if (status == TRANSMITTER)
```

```
        ua[3] = A_ANS ^ C_UA;
    else
        ua[3] = A_CMD ^ C_UA;
    ua[4] = FLAG;

    // WRITE
    int res = write(fd, ua, 5 * sizeof(char));

    return res;
}

int write_disc(int fd, int status) {
    //Create trama DISC
    unsigned char disc[5];
    disc[0] = FLAG;
    if (status == TRANSMITTER)
        disc[1] = A_CMD;
    else
        disc[1] = A_ANS;
    disc[2] = C_DISC;
    if (status == TRANSMITTER)
        disc[3] = A_CMD ^ C_DISC;
    else
        disc[3] = A_ANS ^ C_DISC;
    disc[4] = FLAG;

    int res = write(fd, disc, 5 * sizeof(char));

    return res;
}

void read_disc(int fd, int status) {
    unsigned char disc[MAX_FRAME_SIZE];
    int res;
    int n_bytes = 0;
    unsigned char read_char[1];
    read_char[0] = '\0';
    u_int8_t A;
    if (status == TRANSMITTER)
        A = A_ANS;
    else
        A = A_CMD;
```

```
    receiving_disc_state = START;
    break_read_loop = 0;

    while (!break_read_loop) {
        if (receiving_disc_state != FINISH) {
            res = read(fd, read_char, sizeof(char));
            disc[n_bytes] = read_char[0];
        }

        switch (receiving_disc_state) {
            case START: {
                if (read_char[0] == FLAG) {
                    receiving_disc_state = FLAG_RCV;
                    n_bytes++;
                }
                break;
            }
            case FLAG_RCV: {
                if (read_char[0] == A) {
                    receiving_disc_state = A_RCV;
                    n_bytes++;
                } else if (read_char[0] == FLAG) {
                    n_bytes = 1;
                    break;
                } else {
                    receiving_disc_state = START;
                    n_bytes = 0;
                }
                break;
            }
            case A_RCV: {
                if (read_char[0] == C_DISC) {
                    receiving_disc_state = C_RCV;
                    n_bytes++;
                } else if (read_char[0] == FLAG) {
                    receiving_disc_state = FLAG_RCV;
                    n_bytes = 1;
                } else {
                    receiving_disc_state = START;
                    n_bytes = 0;
                }
                break;
            }
```

```
            case C_RCV: {
                if (read_char[0] == A ^ C_DISC) {
                    receiving_disc_state = BCC_OK;
                    n_bytes++;
                } else if (read_char[0] == FLAG) {
                    receiving_disc_state = FLAG_RCV;
                    n_bytes = 1;
                } else {
                    receiving_disc_state = START;
                    n_bytes = 0;
                }
                break;
            }
            case BCC_OK: {
                if (read_char[0] == FLAG) {
                    receiving_disc_state = FINISH;
                    n_bytes++;
                } else {
                    receiving_disc_state = START;
                    n_bytes = 0;
                }
                break;
            }
            case FINISH: {
                break_read_loop = 1;
                received_disc = 1;
                break;
            }
        }
    }
}

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

    message("Cleaned up terminal.");
}
```

```c
int write_i(int fd, char *buffer, int length) {
    //Create trama
    unsigned char trama[MAX_FRAME_SIZE];
    unsigned char stuff[MAX_DATA_SIZE];
    u_int8_t bcc2 = 0x00;
    trama[0] = FLAG;
    trama[1] = A_CMD;
    if (datalink.sequenceNumber)
        trama[2] = C_1;
    else
        trama[2] = C_0;
    trama[3] = A_CMD ^ trama[2];

    //Efficiency FER
    //generate_errors(trama, 3);

    for (int i = 0; i < length; i++) {
        trama[4 + i] = buffer[i];
        bcc2 = bcc2 ^ buffer[i];
    }
    trama[4 + length] = bcc2;

    //Efficiency FER
    //generate_errors(trama, 4+length);

    trama[5 + length] = FLAG;

    //Stuffing
    int new_bytes = 0;
    int nr_bytes = 6 + length;


    stuff[0] = trama[0];
    stuff[1] = trama[1];
    stuff[2] = trama[2];
    stuff[3] = trama[3];
    for (int j = 4; j < 5 + length; j++) {
        if (trama[j] == FLAG) {
            nr_bytes++;
            new_bytes++;
            stuff[j + new_bytes - 1] = ESCAPE;
            stuff[j + new_bytes] = FLAG ^ STUF;
        } else if (trama[j] == ESCAPE) {
```

```
            nr_bytes++;
            new_bytes++;
            stuff[j + new_bytes - 1] = ESCAPE;
            stuff[j + new_bytes] = ESCAPE ^ STUF;
        } else stuff[j + new_bytes] = trama[j];
    }
    stuff[nr_bytes - 1] = FLAG;

    //Write trama I
    int res = write(fd, stuff, nr_bytes);

    return length;
}

int read_i(int fd, char *buffer, int *reject) {
    unsigned char trama[MAX_FRAME_SIZE] = {};
    unsigned char data[MAX_DATA_SIZE] = {};
    int res;
    int n_bytes = 0;
    int data_bytes = 0;
    unsigned char read_char[1];
    read_char[0] = '\0';

    receiving_data_state = START_I;
    break_read_loop = 0;
    int received_second_flag = 0;

    while (!break_read_loop) {
        if (!received_second_flag) {
            res = read(fd, read_char, sizeof(char));
            if (res < 0)
                continue;
            if (res == 0)
                return res;
            trama[n_bytes] = read_char[0];
        }

        switch (receiving_data_state) {
            case START_I: {
                if (read_char[0] == FLAG) {
                    receiving_data_state = FLAG_RCV_I;
                    n_bytes++;
                }
```

```
                break;
            }
            case FLAG_RCV_I: {
                if (read_char[0] == A_CMD) {
                    receiving_data_state = A_RCV_I;
                    n_bytes++;
                } else if (read_char[0] == FLAG) {
                    n_bytes = 1;
                    break;
                } else {
                    receiving_data_state = START_I;
                    n_bytes = 0;
                }
                break;
            }
            case A_RCV_I: {
                if ((read_char[0] == C_0 && (!datalink.sequenceNumber))
|| (read_char[0] == C_1 && datalink.sequenceNumber)) {
                    receiving_data_state = C_RCV_I;
                    n_bytes++;
                } else if (read_char[0] == FLAG) {
                    receiving_data_state = FLAG_RCV_I;
                    n_bytes = 1;
                    break;
                } else {
                    receiving_data_state = START_I;
                    n_bytes = 0;
                }
                break;
            }
            case C_RCV_I: {
                if (read_char[0] == (A_CMD ^ C_0) || read_char[0] ==
(A_CMD ^ C_1)) {
                    receiving_data_state = BCC1_OK_I;
                    n_bytes++;
                } else if (read_char[0] == FLAG) {
                    receiving_data_state = FLAG_RCV_I;
                    n_bytes = 1;
                } else {
                    receiving_data_state = START_I;
                    n_bytes = 0;
                }
                break;
```

```
            }
            case BCC1_OK_I: {
                if (read_char[0] == ESCAPE) {
                    receiving_data_state = ESCAPE_RCV_I;
                } else {
                    receiving_data_state = DATA_RCV_I;
                    data[data_bytes] = trama[n_bytes];
                    n_bytes++;
                    data_bytes++;
                }
                break;
            }
            case DATA_RCV_I: {
                if (read_char[0] == FLAG) {
                    receiving_data_state = FLAG2_RCV_I;
                    received_second_flag = 1;
                    n_bytes++;
                } else if (read_char[0] == ESCAPE) {
                    receiving_data_state = ESCAPE_RCV_I;
                } else {
                    data[data_bytes] = trama[n_bytes];
                    n_bytes++;
                    data_bytes++;
                }
                break;
            }
            case FLAG2_RCV_I: {
                u_int8_t bcc = 0x00;
                for (int i = n_bytes - data_bytes - 1; i < n_bytes - 2;
i++) {
                    bcc = bcc ^ trama[i];
                }
                if (trama[n_bytes - 2] == bcc) {
                    data_bytes--;
                    receiving_data_state = FINISH_I;
                } else {
                    receiving_data_state = FINISH_I;
                    *reject = 1; //possibility of sending rej message
                    data_bytes = REJECT_DATA;
                }
                break;
            }
            case FINISH_I: {
```

```
                break_read_loop = 1;
                received_i = 1;
                break;
            }
            case ESCAPE_RCV_I: {
                receiving_data_state = DATA_RCV_I;
                trama[n_bytes] = read_char[0] ^ STUF;
                data[data_bytes] = trama[n_bytes];
                data_bytes++;
                n_bytes++;
                break;
            }
        }
    }

    //New trama
    if (nr_tramaI == data[1] || (!control_start && data[1] == '\0')) {
        //bcc2 wrong, then reject
        if (*reject)
            return REJECT_DATA;

        //bcc good, then accept
        memcpy(buffer, data, MAX_DATA_SIZE);
        if (!control_start)
            nr_tramaI = (nr_tramaI + 1) % 256;
        control_start = 0;
    }
    else { //Duplicated trama, then send rr
        *reject = 0;
        data_bytes = REJECT_DATA;
    }

    return data_bytes;
}

int write_rr(int fd) {
    //Create trama rr
    unsigned char rr[5];
    rr[0] = FLAG;
    rr[1] = A_CMD;
    if (datalink.sequenceNumber)
        rr[2] = C_RR0;
    else
```

```
        rr[2] = C_RR1;
    if (datalink.sequenceNumber)
        rr[3] = A_CMD ^ C_RR0;
    else
        rr[3] = A_CMD ^ C_RR1;
    rr[4] = FLAG;

    int res = write(fd, rr, 5 * sizeof(char));

    return res;
}

int read_rr(int fd) {
    unsigned char rr[MAX_FRAME_SIZE];
    memset(rr, '\0', MAX_FRAME_SIZE);
    u_int8_t c_rr, c_rej;
    int res;
    int rej = 0;
    int n_bytes = 0;
    unsigned char read_char[1];

    receiving_rr_state = START;
    break_read_loop = 0;

    while (!break_read_loop) {
        if (receiving_rr_state != FINISH) {
            res = read(fd, read_char, sizeof(char));
            rr[n_bytes] = read_char[0];
        }

        switch (receiving_rr_state) {
            case START: {
                if (read_char[0] == FLAG) {
                    receiving_rr_state = FLAG_RCV;
                    n_bytes++;
                }
                break;
            }
            case FLAG_RCV: {
                if (read_char[0] == A_CMD) {
                    receiving_rr_state = A_RCV;
                    n_bytes++;
                } else if (read_char[0] == FLAG) {
```

```
                    n_bytes = 1;
                    break;
                } else {
                    receiving_rr_state = START;
                    n_bytes = 0;
                }
                break;
            }
            case A_RCV: {
                if (datalink.sequenceNumber) {
                    c_rr = C_RR0;
                    c_rej = C_REJ1;
                }
                else {
                    c_rr = C_RR1;
                    c_rej = C_REJ0;
                }
                if (read_char[0] == c_rr) {
                    receiving_rr_state = C_RCV;
                    n_bytes++;
                } else if (read_char[0] == c_rej) {
                    receiving_rr_state = C_RCV;
                    rej = 1;
                    n_bytes++;
                } else if (read_char[0] == FLAG) {
                    receiving_rr_state = FLAG_RCV;
                    n_bytes = 1;
                } else {
                    receiving_rr_state = START;
                    n_bytes = 0;
                }
                break;
            }
            case C_RCV: {
                if ((read_char[0] == A_CMD ^ c_rr) || (read_char[0] ==
A_CMD ^ c_rej)) {
                    receiving_rr_state = BCC_OK;
                    n_bytes++;
                } else if (read_char[0] == FLAG) {
                    receiving_rr_state = FLAG_RCV;
                    n_bytes = 1;
                } else {
                    receiving_rr_state = START;
```

```
                    n_bytes = 0;
                }
                break;
            }
            case BCC_OK: {
                if (read_char[0] == FLAG) {
                    receiving_rr_state = FINISH;
                    n_bytes++;
                } else {
                    receiving_rr_state = START;
                    n_bytes = 0;
                }
                break;
            }
            case FINISH: {
                break_read_loop = 1;
                if (!rej)
                    received_i = 1;
                break;
            }
        }
    }
    if (n_bytes < 5)
       rej = 1;

    return rej;
}

int write_rej(int fd) {
    //Create trama rej
    unsigned char rej[5];
    rej[0] = FLAG;
    rej[1] = A_CMD;
    if (datalink.sequenceNumber)
        rej[2] = C_REJ1;
    else
        rej[2] = C_REJ0;
    if (datalink.sequenceNumber)
        rej[3] = A_CMD ^ C_REJ1;
    else
        rej[3] = A_CMD ^ C_REJ0;
    rej[4] = FLAG;
```

```
    int res = write(fd, rej, 5 * sizeof(char));

    return res;
}

void generate_errors(unsigned char* buffer, int i){
    if(!(rand()%ERROR_PROB)){
        buffer[i] = 0x00;
    }
}
```

Anexo II

Tamanho do Ficheiro = 10968 bytes

Tempo base de submissão = 3,56s

FER(%)	S(bits/s)	S médio(bits/s)
0	0,718553	0,718553
	0,718553	
1	0,720820	0,720820
	0,720820	
2	0,247027	0,247027
	0,247027	
5	0,148570	0,136176
	0,123781	
10	0,122849	0,108558
	0,094266	

Tamanho dos pacotes (bits)	S(bits/s)	S médio(bits/s)
128	0,463489	0,463489
	0,463489	
256	0,612601	0,611782
	0,610963	
512	0,718553	0,718553
	0,718553	
768	0,741883	0,741883
	0,741883	
1024	0,756623	0,756623
	0,756623	

Capacidade (bits/s)	S(bits/s)	S médio(bits/s)
2400	0,720820	0,719687
	0,718553	
4800	0,720820	0,720820
	0,720820	
9600	0,718553	0,719687
	0,720820	
19200	0,720820	0,719687
	0,718553	
38400	0,718553	0,718553
	0,718553	

Tempo de propagação (microssegundos)	S(bits/s)	S médio(bits/s)
0	0,718553	0,718553
	0,718553	
10000	0,718553	0,718553
	0,718553	
50000	0,700920	0,701999
	0,703077	
100000	0,493521	0,494055
	0,494589	
200000	0,247831	0,247831
	0,247831	