

**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*](mailto:up201708804@fe.up.pt)

Nuno Miguel Teixeira Cardoso [*up201706162@fe.up.pt*](mailto:up201706907@fe.up.pt)

Turma 1

2019/2020

**Conteúdo**

[**1** **Sumário** 2](#_Toc23981146)

[**2** **Introdução** 2](#_Toc23981147)

[**3** **Arquitetura** 3](#_Toc23981148)

[**4** **Estrutura do código** 3](#_Toc23981149)

[4.1 Ligação Lógica 3](#_Toc23981150)

[4.2 Aplicação 4](#_Toc23981151)

[**5** **Casos de uso principais** 6](#_Toc23981152)

[**6** **Protocolo** 6](#_Toc23981153)

[6.1 Ligação Lógica 6](#_Toc23981154)

[6.2 Aplicação 8](#_Toc23981155)

[**7** **Validação** 9](#_Toc23981156)

[**8** **Eficiência** 10](#_Toc23981157)

[**9** **Conclusões** 12](#_Toc23981158)

[**Anexo I** 13](#_Toc23981159)

[application.h: 13](#_Toc23981160)

[application.c: 14](#_Toc23981161)

[packet.h: 19](#_Toc23981162)

[packet.c: 20](#_Toc23981163)

[datalink.h: 24](#_Toc23981164)

[datalink.c: 27](#_Toc23981165)

[**Anexo II** 51](#_Toc23981166)

# **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*

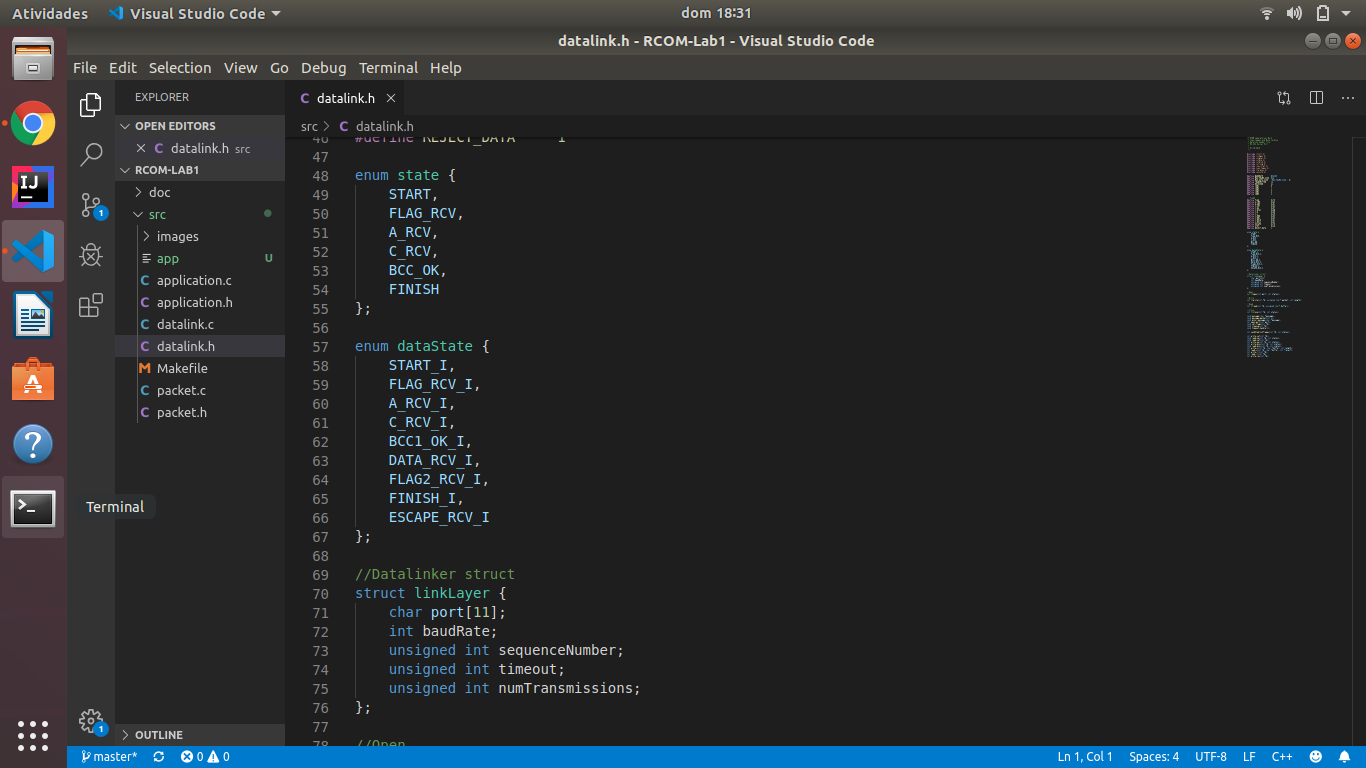
A função *llopen()* começa por abrir a porta *port* e estabelecer as *flags* necessárias à comunicação. A seguir estabelece a ligação entre os dois computadores, tendo um argumento, *status*, que assegura a divisão entre transmissor e recetor e define o modo de proceder de cada um.

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*.

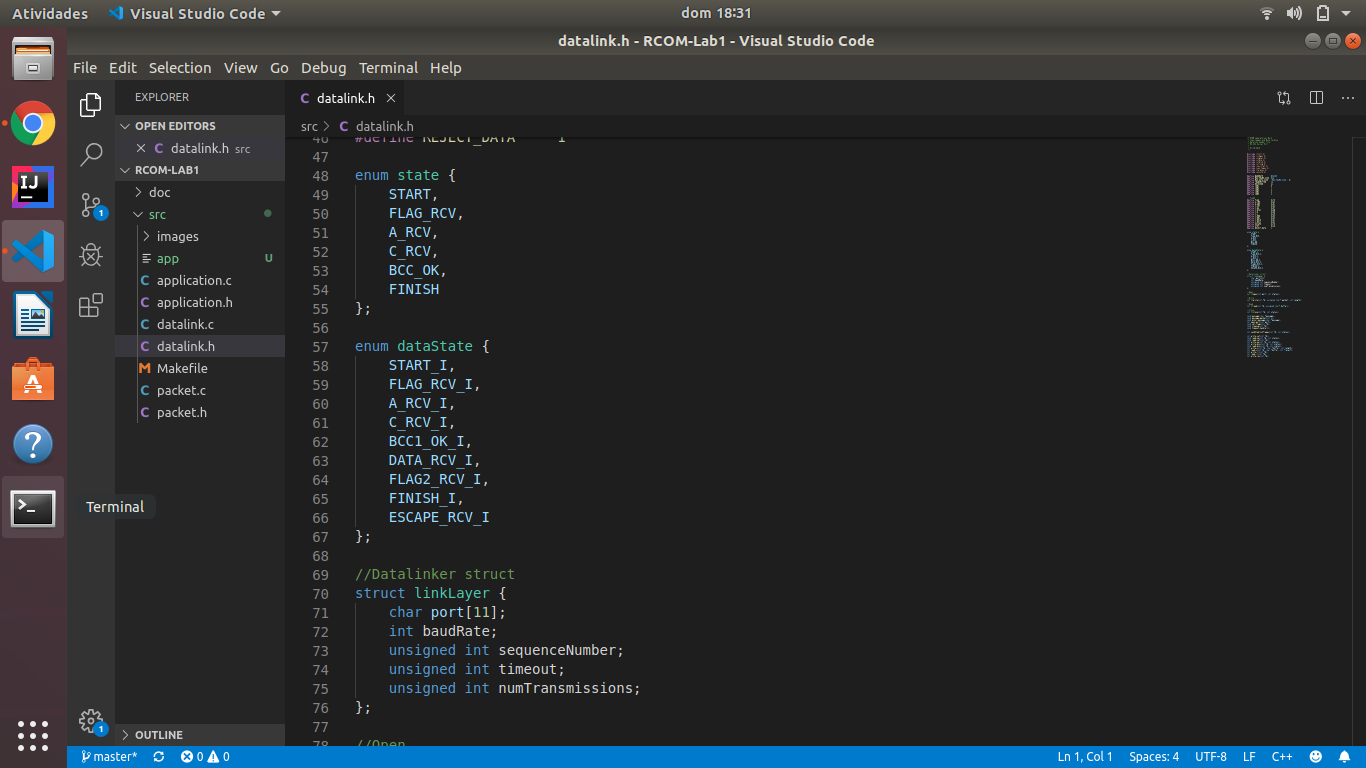
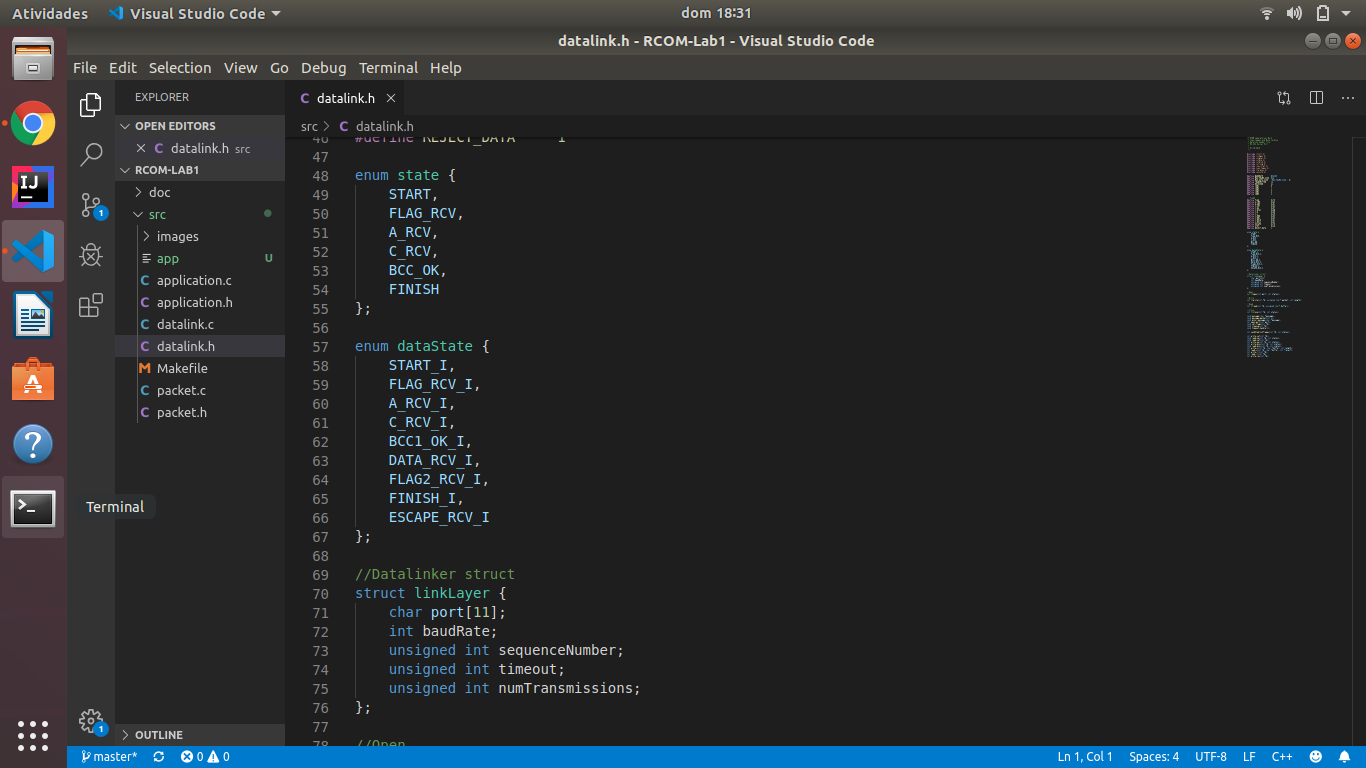
Na função *llclose(),* o transmissor avisa o recetor que terminou a transferência de dados. Ambos restabelecem as definições padrão da porta e fecham-na antes de terminarem.

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



*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* 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.

## 

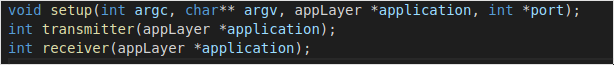
## 

*Figura 3 - Enums state and dataState*

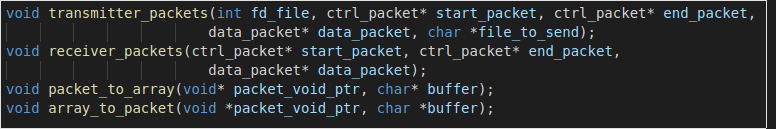
## 

## 4.2 Aplicação

As principais funções que figuram na camada da Aplicação são:

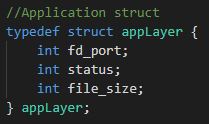


*Figura 4 - Funções da camada de aplicação: application.h*



*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.

**

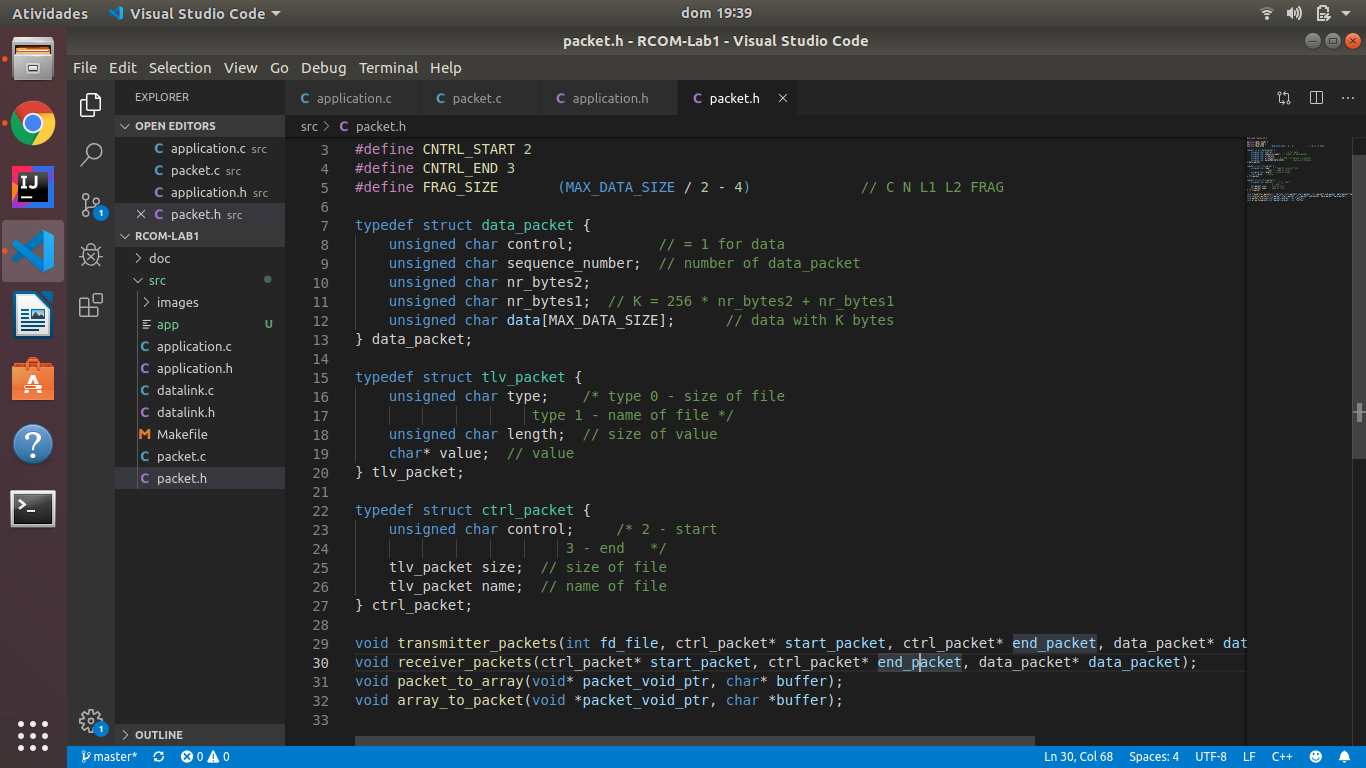
fd\_port: Descritor da porta de série;

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*:



*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().*

# 

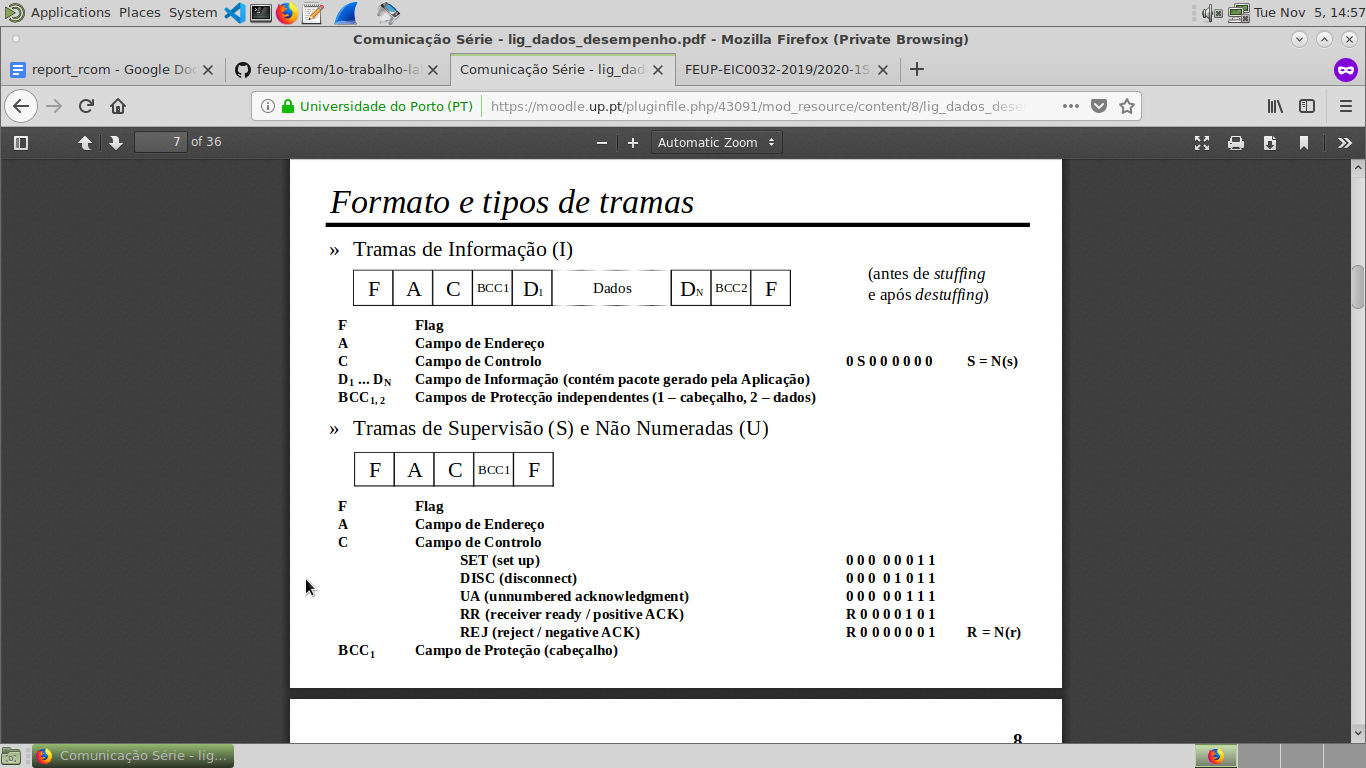
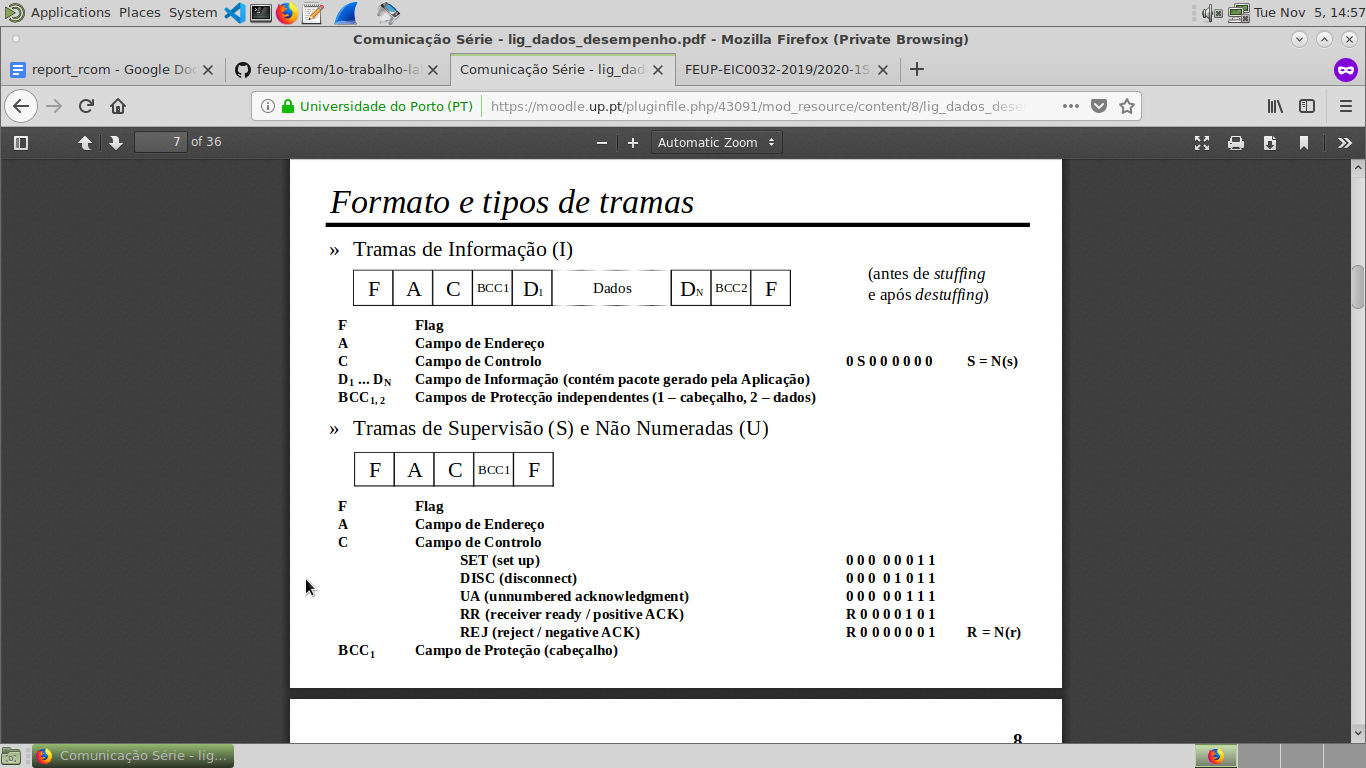
# **6 Protocolo**

## 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*

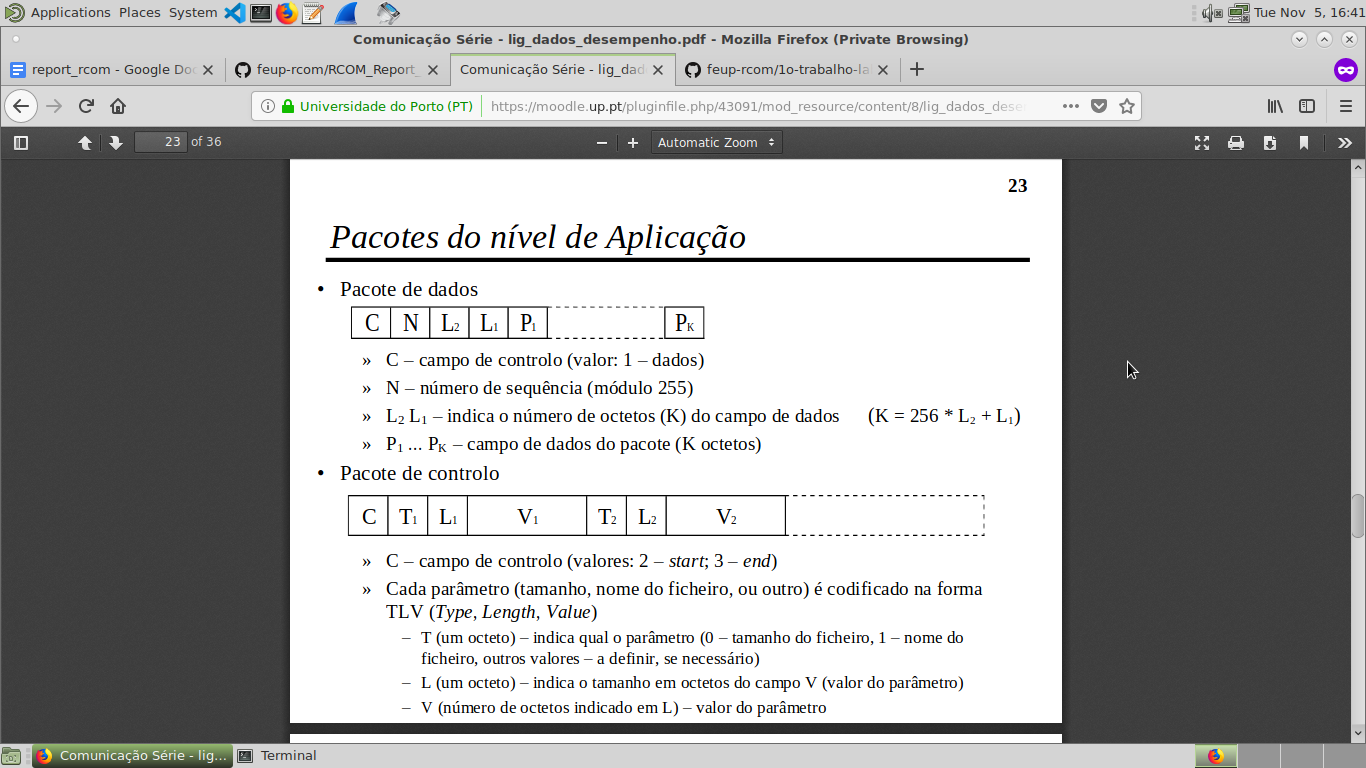
## 6.2 Aplicação

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.

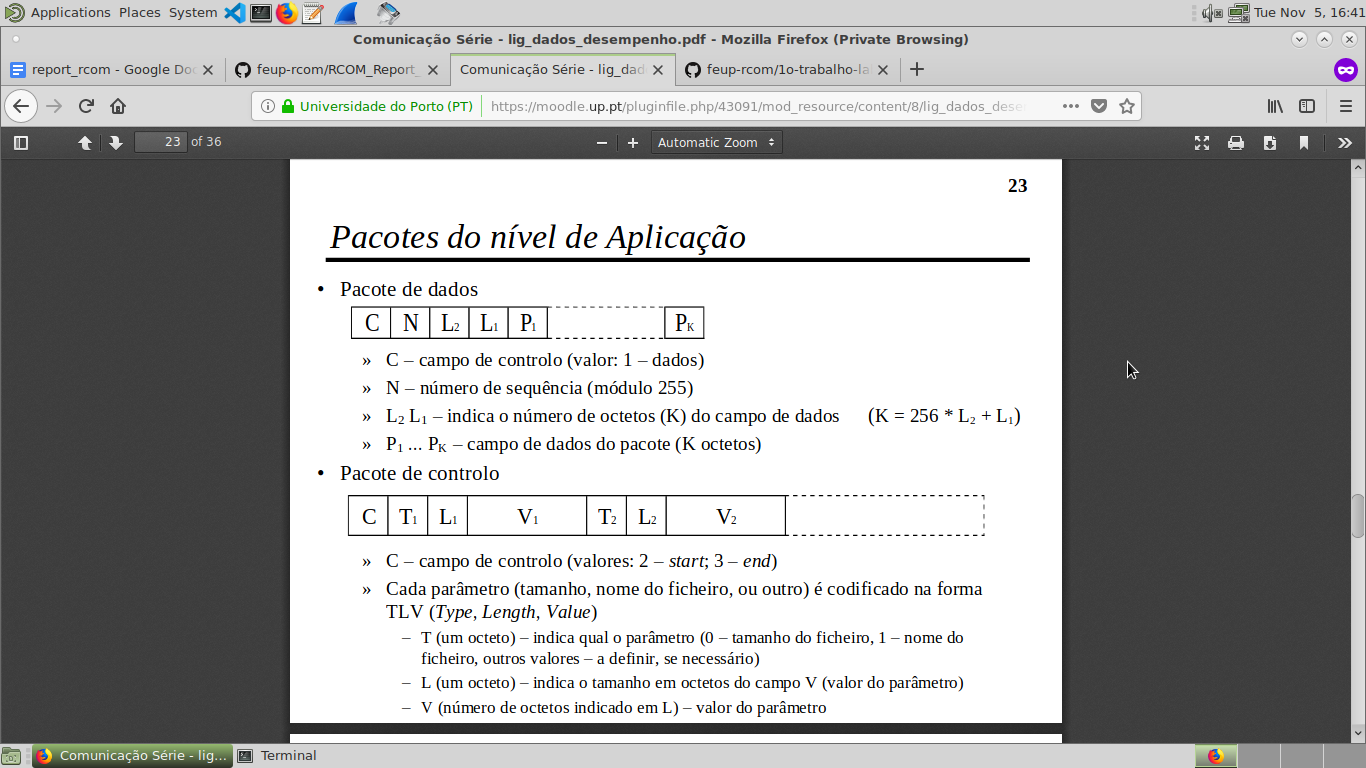
Concluído este processo, a Aplicação chama a função *llwrite()* para enviar o pacote de controlo de *START*, indicando ao recetor que irá iniciar a transmissão do ficheiro. Ciclicamente, de seguida, faz-se a leitura de parte do ficheiro, com um tamanho *FRAG\_SIZE*  = 512 bits, passa-se a informação 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.



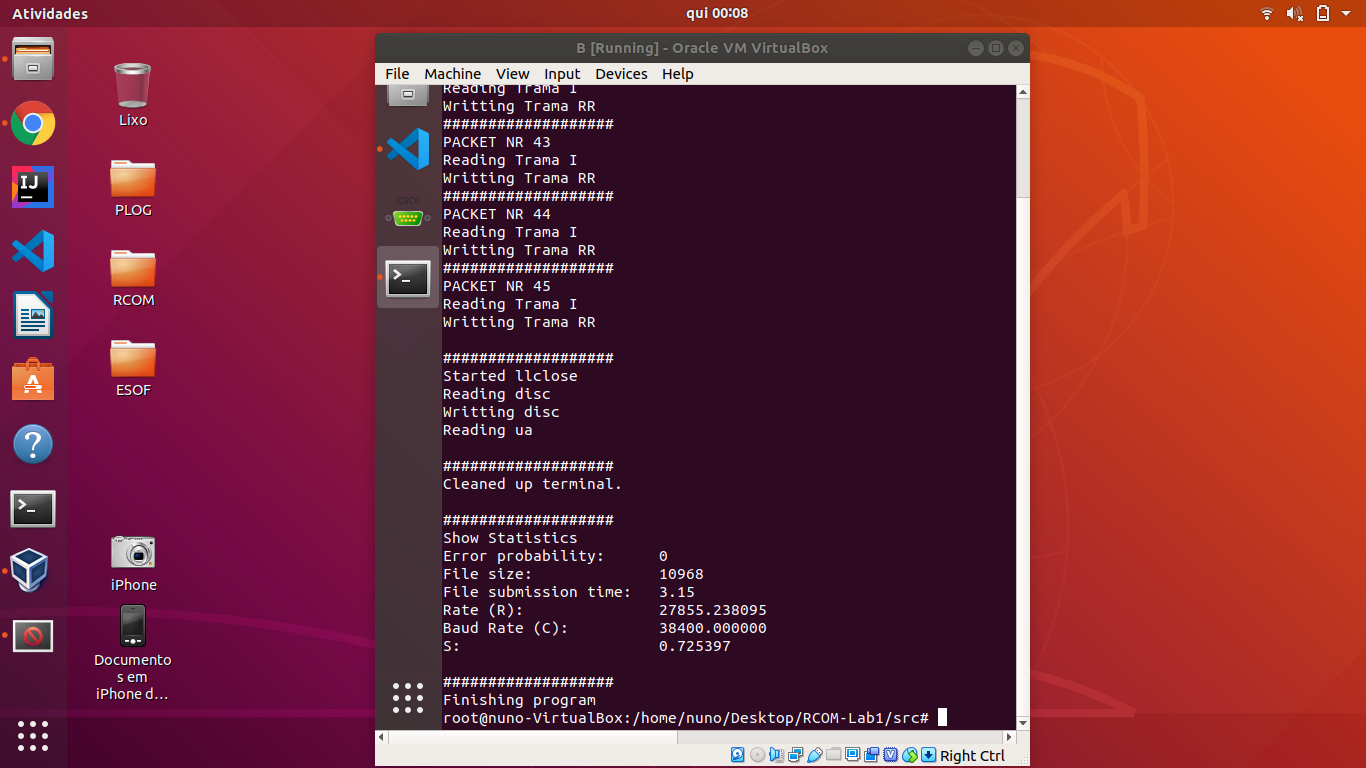
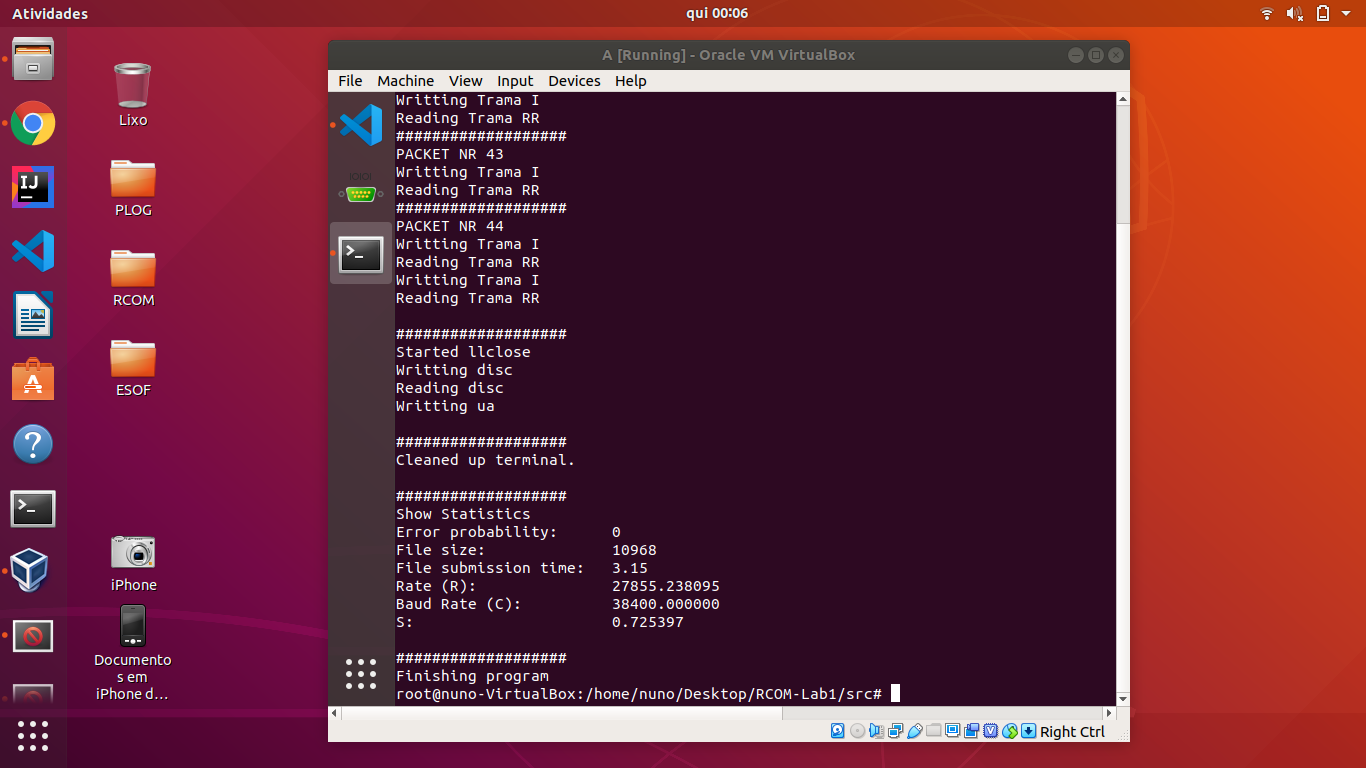
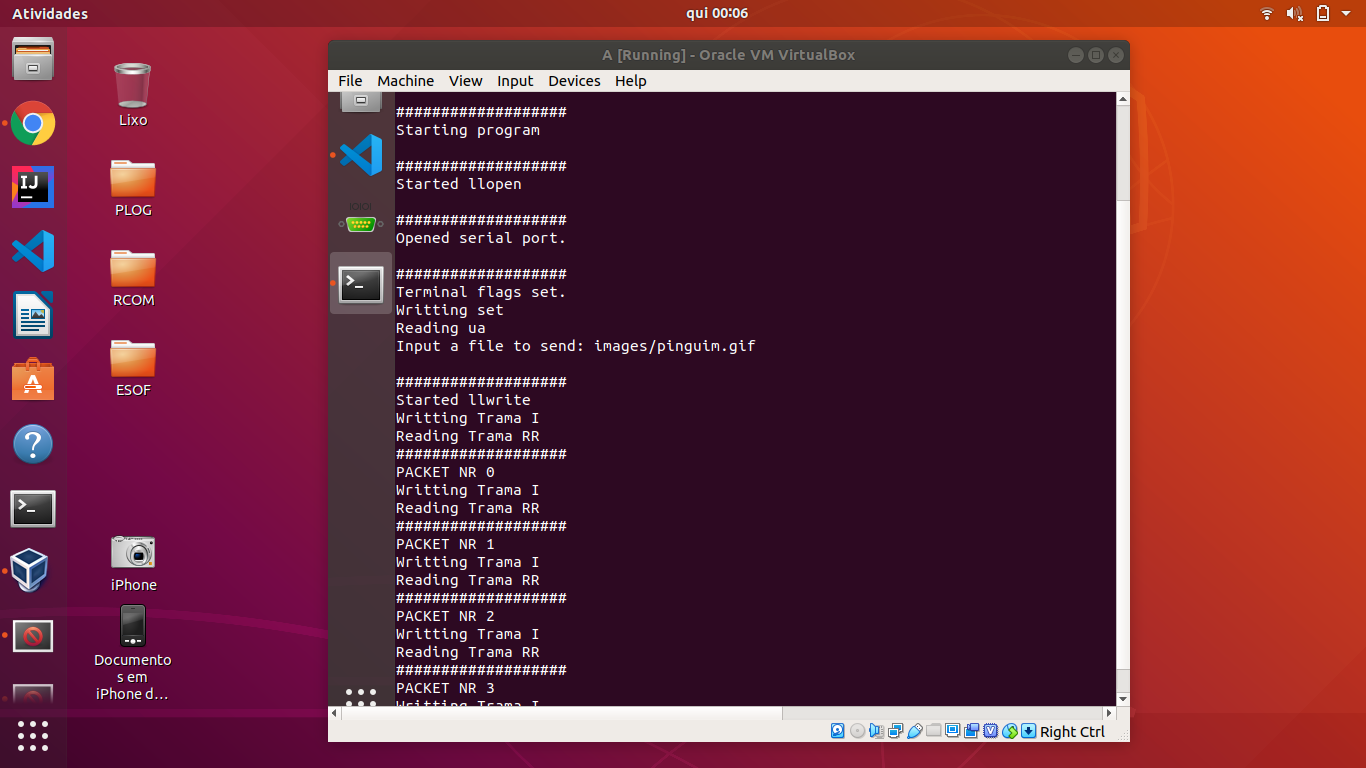
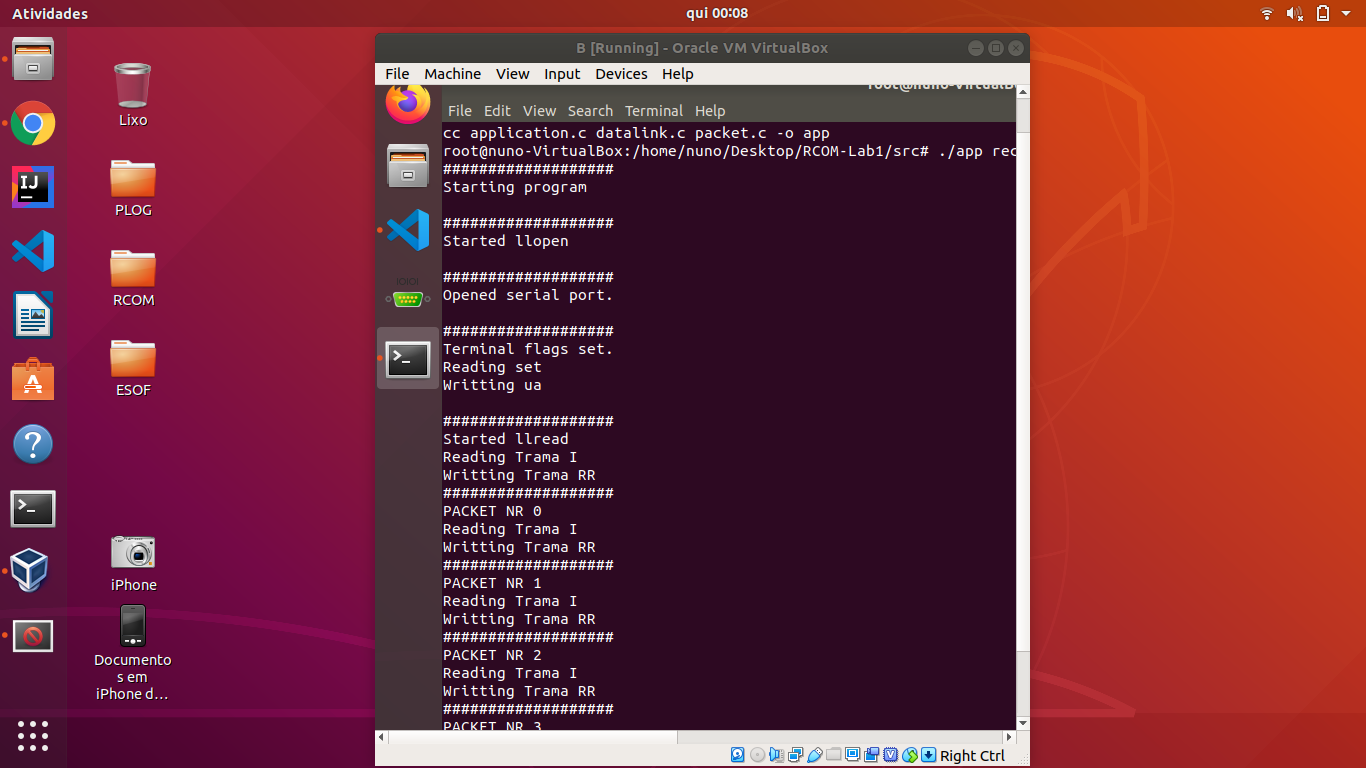
*Figura 8 - Estrutura de pacotes de dados*



*Figura 9 - Estrutura de pacotes de controlo*

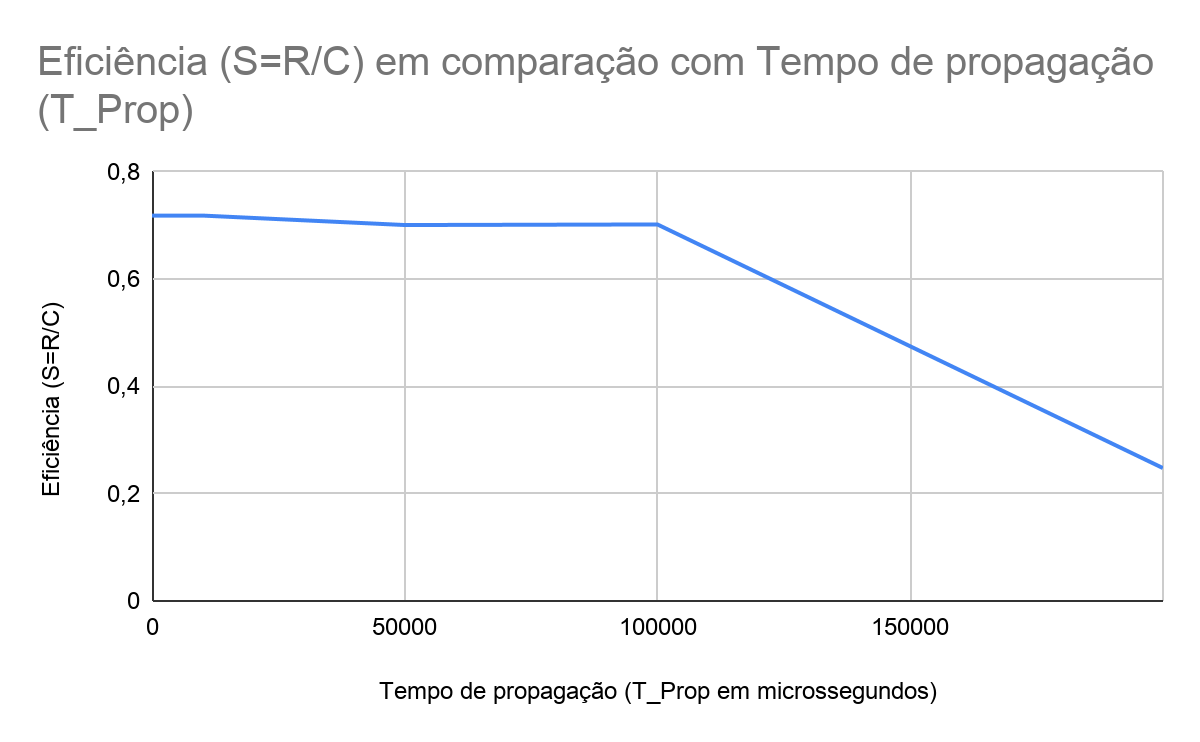
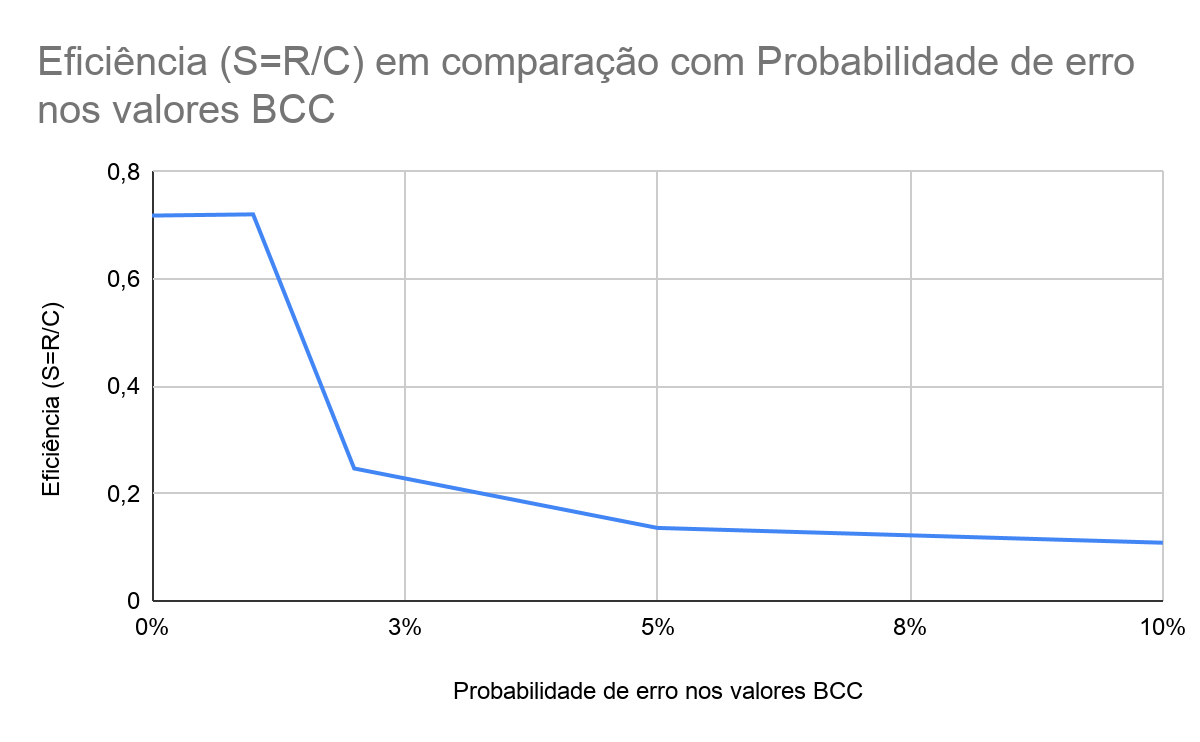
# **7 Validação**

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



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

# Eficiência (S=R/C) em comparação com Tamanho dos pacotes**Eficiência (S=R/C) em comparação com Capacidade de Ligação (C)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:

**/\* 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.");**

**}**

**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 |