22 de dezembro de 2021

**Projeto de Redes de Computadores**

**Aplicação de Transferência de Ficheiros com Porta Série**

**Autores**

Mateus Silva, up201906232

Melissa Silva, up201905076

**Sumário**

O presente relatório aborda o funcionamento do primeiro projeto realizado para a unidade curricular de *Redes de Computadores*, onde se pretendia a elaboração de uma aplicação capaz de transferir ficheiros entre dois computadores com diferentes papéis (*recetor* e *transmissor*) ligados por uma porta série. A aplicação deveria seguir um modelo de camadas: o maior nível, com a aplicação em si e a sua interação com o utilizador (que lhe fornece argumentos); o menor nível, que efetue a transmissão de dados do ficheiro e finalmente a camada de ligação entre as duas últimas.

Após vários testes realizados através de portas série virtuais e nas portas séries físicas fornecidas em laboratório, concluímos que o trabalho foi realizado com sucesso e que cumprimos todos os objetivos estabelecidos no guião. O resultado é então uma aplicação completamente funcional capaz de transmitir diferentes tipos de ficheiros (imagens, documentos, texto, áudio e vídeo) entre dois computadores sem perda de dados.

**Introdução**

O projeto tinha como objetivos a elaboração de um mecanismo de transmissão de dados por camadas, em que uma destas intermediasse os dois níveis mais extremos, sendo estes os da aplicação e o *driver* da porta série.

O protocolo da camada intermédia deveria possuir um mecanismo de transparência por *byte stuffing* e criação de tramas de três possíveis tipos com componentes estruturais comuns (cabeçalho) mas que se diferenciam nas demais devido ao papel têm. Tramas de supervisão e não numeradas possuem proteção simples, enquanto tramas de informação possuem proteção dupla. A respeito da transmissão de tramas (*driver* da porta série), deveria usar-se um mecanismo de *Stop and Wait* com *timeout* e tentativa de retransmissão de tramas. Cada interveniente na transmissão (*emissor* e *recetor*) recebe e envia tramas; contudo, apenas o emissor pode transmitir pacotes – e, portanto, apenas ele pode transmitir tramas de informação. A aplicação deveria suportar e interpretar dois tipos de pacotes: de controlo e de dados. Os primeiros servem para sinalizar o início e fim de uma transmissão, tendo uma estrutura diferente.

Finalmente, espera-se que haja independência entre todas as camadas, pelo que a camada intermédia (ligação de dados) não pode processar cabeçalhos dos pacotes; a camada de aplicação não pode conhecer os detalhes da camada intermédia, apenas como lhe pode aceder; finalmente, a numeração de tramas de informação e pacotes de dados devem ser totalmente independentes.

As secções 5 e 6 detalham o funcionamento de cada uma das camadas do projeto.

**Arquitetura**

O projeto possui duas camadas independentes: aplicação, ligação de dados. A primeira encontra-se totalmente nos ficheiros app, tratando de leitura e criação de pacotes de dados e pacotes de controlo. A camada de ligação de dados encontra-se em *protocol\_app* (*llopen(), llclose()*, *llwrite()*, *llread()* e criação de tramas de informação), stuffing (para o mecanismo de transparência) e em data\_protocol (leitura e escrita de tramas de informação e comandos).

O utilizador chama um dos resultados da compilação dos ficheiros de código – *receiver* e *transmitter* *–* para iniciar o programa como recetor ou emissor, respetivamente.

**Estrutura do Código**

O código tem duas vertentes, uma para cada interveniente da transmissão. Isto resulta em dois executáveis, *receiver* e *transmitter*, cuja estrutura exploramos a seguir.

**Transmitter *(Emissor)***

A interação com este executável faz-se usando comandos com o seguinte formato no terminal.

./transmitter [porta\_série] (-t timeout) (-n tent\_falha) [-f nome do ficheiro]

[porta\_série] é um argumento obrigatório para a definição da ligação com o transmissor através da porta série.

(-t timeout) permite indicar um *timeout* em segundos usado no mecanismo de *Stop and Wait*. Se não indicado, existe um valor *default* de 3 segundos.

(-n tent\_falha) permite indicar um número máximo de tentativas no caso de ocorrerem erros na transmissão. Se não indicado, existe um valor *default* de 3 tentativas.

[-f nome\_ficheiro] é um argumento obrigatório para indicar o ficheiro a transmitir.

Da camada de ligação de dados, chama:

llopen – inicia o programa, abrindo a comunicação através das entradas na porta série, envia uma trama de supervisão SET e espera por uma trama não numerada UA de *receiver*.

a

llwrite – escreve as várias tramas de informação e está preparado para tentar retransmiti-las caso a informação não chegue corretamente a *receiver*. Precede-se a escrita com um processo de *byte stuffing* aplicado às tramas.

**I**

llclose – cessa a comunicação entre as entradas da porta série. Isto é precedido pelo envio de uma trama DISC a *receiver*, esperando-se por uma trama igual deste, após o que se envia uma trama UA.

Da camada da aplicação, chama:

create\_control\_package – cria um pacote de controlo, quer este seja de início ou fim de transmissão, com uma componente em formato TLV. A informação para um pacote provém do ficheiro fornecido a *transmitter*.

**I**

read\_data\_package – cria um pacote de dados, com informação proveniente do ficheiro fornecido a *transmitter*.

**Receiver *(Recetor)***

Invoca-se o executável com comandos do seguinte formato.

receiver [porta\_série] [-f nome\_ficheiro]

[porta\_série] é um argumento obrigatório para a definição da ligação com o transmissor através da porta série.

[-f nome\_ficheiro] é um argumento obrigatório para definir um novo nome para o ficheiro recebido.

Da camada de ligação de dados, chama:

o

llopen – inicia o programa, abrindo a comunicação através das entradas na porta série, recebe uma trama de supervisão SET e envia uma trama não numerada UA.

llread – lê tramas de informação enviadas por *transmitter*, efetua *byte destuffing* e valida a informação antes de a guardar. De outra forma, envia uma trama REJ, solicitando uma retransmissão.

Da camada da aplicação, chama:

llclose – cessa a comunicação através das entradas da porta série. Espera por uma trama DISC de *transmitter*, envia uma de volta a este e espera por uma trama UA para confirmar tal ação.

oi

read\_data\_package – lê e guarda a informação contida num pacote de dados.

read\_control\_package – lê e guarda a informação contida na componente de formato TLV do pacote de controlo, quer este seja de início ou de fim de transmissão.

**Estruturas de Dados**

struct linkLayer {

char port[20]; // Dispositivo /dev/ttySx, x = 0, 1

int baudRate; // Velocidade de transmissão

unsigned int sequenceNumber; // Nº de sequência da trama (0,1)

unsigned int timeout; // Valor do temporizador: 1s

unsigned int numTransmissions; // Tentativas em caso de falha

int fileDescriptor; // Descritor da porta série

int status; // TRANSMITTER | RECEIVER

};

struct applicationLayer {

int sequenceNumber; // Nº de sequência do pacote de dados esperado

};

typedef enum {

START,

FLAG\_RCV,

A\_RCV,

C\_RCV,

BCC\_OK,

D\_RCV,

STOP

} messageState

**Casos de Uso Principais**

Existem dois casos de uso principais: receção do ficheiro (*receiver*) ou envio do ficheiro (*transmitter*). Mostramos a seguir a sequência de chamadas das funções principais mencionadas na secção anterior.

*transmitter*

1. llopen()
2. create\_control\_package();
3. llwrite();
4. Ciclo:
   1. create\_data\_package();
   2. llwrite();
5. create\_control\_package();
6. llwrite();
7. llclose();

*receiver*

1. llopen()
2. llread()
3. read\_control\_package()
4. Ciclo
   1. llread()
   2. read\_data\_package();
5. llread();
6. read\_control\_package();
7. llclose();

**Protocolo de Ligação Lógica**

**llopen() – Estabelecimento da Ligação Lógica**

Se for *transmitter* a chamar esta função, abre-se a porta série e envia-se uma trama SET para *receiver*, usando a função send\_cmd(). Espera-se por uma trama UA como resposta de *receiver*, e esta será lida usando read\_cmd(), que implementa a máquina de estados do **Anexo I**. Ativa-se também o mecanismo de alarme e, caso ocorra *timeout*, ocorrerão até *n* tentativas de retransmissão. Após a *n-*ésimatentativa falhada, o programa termina.

Se for *receiver* que chama esta função, abre-se a porta série, lê-se a trama SET recebida com read\_cmd()e envia-se uma trama UA em resposta, usando send\_cmd().

**llwrite() – Escrita de Tramas de Informação**

Apenas *transmitter* chama esta aplicação. Primeiro, assegura-se que a informação a transmitir é guardada caso seja necessário retransmiti-la. Segue-se a criação da trama de informação com o formato adequado usando a função create\_info\_trama(). Esta última cria o componente BCC2 através de uma operação XOR de todos os *bits* presentes na componente dos dados, e ambas as componentes sofrerão um processo de *byte stuffing*. A trama é enviada ao *receiver* e espera-se a sua resposta. Pelo mecanismo de alarme, caso não recebamos uma resposta antes de terminar um período de *timeout*, retransmissão irá ocorrer.

Assumindo que não há *timeout*, verificamos a resposta de receiver: se for RR, então a informação enviada foi aceite, pelo que validamos o número de sequência esperado (que deve ser oposto ao atual em llwrite(), guardado numa variável estática para este propósito). Se for válido, desativamos o alarme, mudamos o valor de sequência de llwrite() e sai-se da função. Se não for válido, vamos retransmitir a informação até obtermos a resposta esperada ou excedermos o número de tentativas máximo. De outra forma (resposta REJ), desativamos o alarme e procederemos a retransmitir a informação até obtermos a resposta esperada.

**llread() - Leitura de Tramas de Informação**

Apenas *receiver* chama esta função. Primeiro, usa-se read\_info\_trama() para efetuar a leitura da trama de informação recebida de *transmitter.* A função mencionada implementa a máquina de estados do **Anexo 2**. Depois, sujeita-se a trama a *destuffing*. O resultado é usado para obter BCC2, efetuando uma operação XOR entre todos os bits da trama (exceto o último, já que esse é o verdadeiro valor de BCC2, para verificação).

Verifica-se primeiro se a informação recebida é duplicada, através de uma validação do número de sequência recebido com o valor esperado. Se for inválido, a informação é repetida e pediremos retransmissão enviando uma trama RR de número de sequência oposto ao recebido, descartando a informação recebida. De outra forma, efetuamos a validação entre os valores de BCC2 mencionados: se forem iguais, a informação chegou corretamente, pelo que a guardamos, enviamos para *transmitter* uma trama RR e mudamos o número de sequência esperado, saindo, depois, da função. Caso contrário, enviamos uma trama REJ como pedido de retransmissão.

**llclose()** - **Fecho da Ligação Lógica**

Se for *transmitter* a chamar esta função, enviamos uma trama DISC a *receiver*, usando a função send\_cmd(). Espera-se a resposta de *receiver*, que deverá ser uma trama DISC, que será lida usando read\_cmd() (**Anexo I**). Caso não haja uma resposta antes de ocorrer *timeout*, então o alarme será ativado e haverá, no máximo, *n* tentativas de retransmissão. Caso isto não surta sucesso, o programa termina por aqui. De outra forma, envia-se uma trama UA e procede-se ao fecho da porta série.

Se for *receiver* a chamar esta função, irá ler-se a trama DISC recebida através de read\_cmd(), enviar-se-á uma trama DISC em resposta usando send\_cmd() e ler-se-á a trama UA recebida, novamente com read\_cmd(), e, por fim, fecha-se a porta série.

**Protocolo da Aplicação**

**create\_control\_package()**

Esta função cria os dois tipos de pacotes de controlo para o início e fim de uma transmissão. É apenas chamada por *transmitter* e recebe deste a informação do ficheiro a transmitir, que é expressa no formato TLV (tipo, tamanho, valor) no pacote. A informação transmitida é o tamanho e o nome do ficheiro. Pacotes de controlo são expressos em *arrays* de *u\_int8\_t*.

**create\_data\_package()**

Chamada apenas por transmitter, recebendo como parâmetros o número do pacote a criar, o seu tamanho e uma porção do conteúdo do ficheiro. É feito o processamento destes dados e depois faz-se o pacote com o formato desejado.

Pacotes de dados são expressos em *arrays* de *u\_int8\_t*.

**read\_control\_package()**

É chamada por *receiver*. Lê pacotes de controlo e guarda a informação encontrada nos campos valor das componentes TLV destes.

**read\_data\_package()**

Apenas chamada apenas por *receiver*, esta função lê pacotes de dados. Para tal, começa por verificar se está a lidar com um pacote repetido (validando o número de sequência de pacote com o número de sequência esperado). Se verificar que um pacote é repetido, o programa é terminado pois houve perdas de informação. De outra forma, extrai-se a informação do pacote.

**Validação**

Testámos o nosso código em múltiplos cenários.

**1.** Transmissão da imagem sugerida: pinguim.gif;

**2.** Transmissão de imagens de tamanho maior que a sugerida;[[1]](#footnote-1)

**3.** Transmissão de ficheiros de outros formatos;[[2]](#footnote-2)

**4.** Interrupção da Transmissão (desconetando/reconectando a porta série);

**5.** Transmissão com Indução de Erros (aproximando barra de ferro à porta série);

**6.** Transmissão com Variação dos valores de *Baudrate;*

**7.** Transmissão com Variação dos valores em tramas de informação;

**8.** Transmissão com Variação dos valores de *T\_prop;*

**9.** Transmissão com Variação de FER em Percentagem.

O nosso código passou em todos os testes com sucesso.

**Eficiência do Protocolo de Ligação de Dados**

Todos os testes foram feitos com o ficheiro pinguim.gif (10968 *bytes*) e com uma *baudrate* constante de 38400 bit/s (excetuando o teste de eficiência em que tínhamos de variar este valor). Esta secção elenca as conclusões tiradas e os gráficos que nos levaram a estas podem ser vistos a partir do **Anexo XVI** (inclusive).

**FER**

Como sugerido, gerámos, aleatoriamente, erros em tramas de informação no seu cabeçalho (BCC1) e no campo de dados (BCC2). Concluímos, como esperado, que FER influencia negativamente a eficiência do programa.

Os erros em BCC1 produzem uma perda de eficiência mais significativa pois é necessário esperar o *timeout* (que tem como valor por defeito 3 segundos) para reenviar a trama. Já no caso dos erros no BCC2, apenas acontece o reenvio da trama, que ocorre de imediato.

**T\_prop**

Como esperado, observou-se que quanto maior for o atraso provocado à propagação de dados, menos eficiente é o programa. Isto deve-se ao facto que todas as tramas demoram mais tempo a serem transmitidas entre dispositivos, o que pode provocar que o alarme seja disparado (atrasando ainda mais o programa).

**C**

Concluímos que a eficiência é ligeiramente melhor com menores valores de *baudrate*. Cremos que isto acontece porque, com *baudrates* maiores, o tempo gasto em processamento, envio e receção de tramas tem maior impacto porque, neste tempo, poder-se-ia enviar um grande número de *bytes* ao invés de se processarem tramas. Isto perde o impacto em *baudrates* menores, já que não seria possível transferir tantos *bytes* no mesmo intervalo de tempo devido ao limite da *baudrate*, caso não houvesse interrupções causadas pelas mensagens de *acknowledgement.*

**Tamanho da Trama I**

Vimos facilmente que, quanto maior for o tamanho da trama de informação, maior será a eficiência do programa, uma vez que, quanto maior a trama, maior será o tamanho do pacote e mais informação é enviada de uma vez, não sendo preciso enviar tantos pacotes. Por consequência, menos tempo será gasto em preparação (*byte stuffing* e *destuffing*, por exemplo) para enviar um conjunto de *bytes*, o que também leva a que o *timeout* ocorra menos vezes.

**Protocolo *Stop & Wait***

Num protocolo *Stop & Wait*, o emissor transmite uma trama de informação a um recetor e espera então por um *acknowledgement* positivo (ACK) da parte deste ou um *acknowledgemente* negativo (NACK). Quando o recetor recebe uma trama, verifica que esta não possui erros; se isto se verificar, envia um ACK e, de outra forma, um NACK. Quando o recetor recebe um ACK, prosseguirá para o envio da trama seguinte (caso ainda não tenha transmitido a informação na sua totalidade) e, caso contrário, irá reenviar a mesma trama. Para além disto, caso a trama de informação ou os *acknowledgements* não sejam recebidos (isto é, perdidos), o programa deverá ter um mecanismo de *timeout* para que a trama I seja reenviada, de modo a assegurar o funcionamento correto do programa. Outro problema que pode existir será o recetor receber tramas repetidas. Para garantir que o recetor consegue identificar se uma trama é nova ou duplicada, estas são numeradas com um número de sequência. A mesma coisa acontece com os *acknowledgements* para o emissor, para que este tenha a certeza que recebeu os *acknowledgements* de todas as tramas e ainda saber quais as tramas que deve retransmitir. ACK(i) indica que o recetor está à espera da trama de informação (i).

Podemos afirmar que o nosso programa utiliza este protocolo: as tramas de informação são identificadas com 0 ou 1 e o emissor espera por um *acknowledgement* do recetor que pode ser RR, equivalente a ACK, ou REJ, equivalente a REJ – estes também são identificados com 0 ou 1. Caso o emissor receba um RR, irá enviar a próxima trama e, de outra forma, o emissor reenvia a mesma trama. Se não receber nenhum *acknowledgement*, será ativado um *timeout* e um reenvio de trama.

**Conclusão**

Durante o período de trabalho dedicado a este projeto, conseguimos implementar tudo aquilo que era esperado com sucesso, resultando numa aplicação completamente funcional e subsistente para a transferência de ficheiros entre dois computadores através de uma porta série de uma forma segura.

No geral, acreditamos que o projeto é de uma complexidade elevada, embora tenhamos conseguido atingir os seus objetivos. A parte onde encontrámos mais dificuldade foi para a obtenção de dados para o estudo estatístico feito na secção anterior, visto que só poderíamos obter medições em laboratório. Mesmo assim, essas dificuldades foram ultrapassadas e gostaríamos de demonstrar orgulho em ter completado este projeto.

**Anexo I –** Diagrama da Máquina de Estados Implementada em read\_cmd()

Diagram, schematic

Description automatically generated

**Anexo II –** Diagrama da Máquina de Estados Implementada em read\_info\_trama()

A picture containing text, device

Description automatically generated

**Anexo III –** Código fonte de alarm.h.

#ifndef ALARM\_H

#define ALARM\_H

/\*\*

\* Alarm handler: increases a counter of the number of tries of a certain action and checks if it has exceeded the number of tries permitted

\*/

void atende();

/\*\*

\* Installs alarm

\*/

void install\_alarm();

/\*\*

\* Deactivates the alarm

\*/

void deactivate\_alarm();

#endif

**Anexo IV –** Código fonte de alarm.c.

#include <unistd.h>

#include <signal.h>

#include <stdio.h>

#include <stdlib.h>

#include "protocol\_app.h"

#include "constants.h"

int conta = 0;

void atende() // atende alarme

{

conta++;

printf("alarme # %d\n", conta);

if (conta > link\_info.numTransmissions)

{

printf("Number of transmission tries exceed");

close(link\_info.fileDescriptor);

exit(-1);

}

}

void install\_alarm()

{

signal(SIGALRM, atende);

siginterrupt(SIGALRM, TRUE);

}

void deactivate\_alarm()

{

conta = 0;

alarm(0);

}

**Anexo V –** Código fonte de app.h.

#ifndef APP\_H

#define APP\_H

struct applicationLayer

{

int sequenceNumber;

};

struct applicationLayer app\_info;

/\*\*

\* Function that creates application data package.

\* @param n: sequence number.

\* @param length: size of data array.

\* @param data: contains bytes to be sent.

\* @param package: array to store built package.

\* @return: 0 or -1 in case of error.

\*/

int create\_data\_package(int n, int length, u\_int8\_t \*data, u\_int8\_t \*package);

/\*\*

\* Function that reads application data package.

\* @param data: array to store data contained in data package.

\* @param package: data package that function is going to read and extract information from.

\* @return: size of data read or -1 in case of error.

\*/

int read\_data\_package(u\_int8\_t\*data, u\_int8\_t \*package);

/\*\*

\* Function that creates application control package.

\* @param c: control byte (can be start or end).

\* @param file\_name: name of the file that will be sent.

\* @param length: size of the file that will be sent.

\* @param package: array to store built package.

\* @return: size of the control package or -1 in case of error.

\*/

int create\_control\_package(u\_int8\_t c, u\_int8\_t \*file\_name, int length, u\_int8\_t\*package);

/\*\*

\* Function that reads application control package.

\* @param package: control package to be read.

\* @param file\_name: variable used to store the name of the file (information in control package).

\* @param file\_size: variable used to store size of the file (information in control package).

\* @param package\_size: size of the control package.

\* @return: 0 or -1 in case of error.

\*/

int read\_control\_package(u\_int8\_t \*package, u\_int8\_t \*file\_name, int \*file\_size, int package\_size);

#endif

**Anexo VI –** Código fonte de app.c.

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <termios.h>

#include <stdio.h>

#include <signal.h>

#include <string.h>

#include <stdlib.h>

#include "app.h"

#include "constants.h"

int create\_data\_package(int n, int length, u\_int8\_t \*data, u\_int8\_t \*package)

{

/\*We are gonna ask the user for stuff to fill out the struct linkLayer

Need to make two packets: control and data

Control packets have two types: 1) symbolizes transmission start

2) symbolizes transmission end

Contents of each packet->slide 23

Packages are sent by TRANSMITTER

\*/

package[0] = CTRL\_DATA; //C – campo de controlo (valor: 1 – dados)

package[1] = n % 255;

package[2] = length / 256; //L2

package[3] = length % 256; //L1

if (memcpy(&package[4], data, length) == NULL)

{ //memcpy returns destination pointer

printf("ERROR\n");

return -1;

}

return 0;

}

int create\_control\_package(u\_int8\_t c, u\_int8\_t \*file\_name, int file\_size, u\_int8\_t \*package)

{

int size = 0;

package[0] = c; //need to be informed if it's supposed to be the start (2) or end (3)

/\*Going to have two sets of TLV:

First one is about the size of the file

Second about the name of the file

\*/

char \*lstring = (char \*)malloc(sizeof(int));

sprintf(lstring, "%d", file\_size);

package[1] = 0; //file size (Type)

package[2] = strlen(lstring); //size of file size (Length)

if (memcpy(&package[3], lstring, strlen(lstring)) == NULL) //file size (Value)

{

printf("ERROR\n");

free(lstring);

return -1;

}

size = 3 + strlen(lstring);

package[size] = 1; //file name (Type)

size++;

package[size] = strlen(file\_name);//size of file name (Length)

size++;

if (memcpy(&package[size], file\_name, strlen(file\_name)) == NULL) //file name (Value)

{

printf("ERROR\n");

free(lstring);

return -1;

}

size += strlen(file\_name);

free(lstring);

return size;

}

int read\_data\_package(u\_int8\_t \*data, u\_int8\_t \*package)

{

/\*

package[2] & [3] are gonna be used to know the size of the info

Rest of the package is gonna be copied into u\_int8\_t\*data

\*/

/\*app\_info.sequenceNumber is where we store the sequence number we are expecting (packet wise).

This value is initially 0, thus the first package we expect has got to have 0 as a sequence number.

This is what's expected because we still haven't gotten that package. If we get it, we then update

the value of app\_info.sequenceNumber for future runs of read\_data\_package().

If we receive a value that is lower than the one in app\_info.sequenceNumber then it's

repeated: we don't want it because we already have it. If the value in the package received is

\*higher\* than expected, then it means a package was jumped over and, because of that, lost - thus

the file transmission has failed \*/

if (package[1] < app\_info.sequenceNumber)

{

return -1; //repeated packet

}

app\_info.sequenceNumber = (app\_info.sequenceNumber + 1) % 255; //Update

int size = 256 \* package[2] + package[3];

if (memcpy(data, &package[4], size) == NULL)

{

printf("ERROR\n");

return -1;

}

return size;

}

int read\_control\_package(u\_int8\_t \*package, u\_int8\_t \*file\_name, int \*file\_size, int package\_size)

{

u\_int8\_t \*sizes = (u\_int8\_t\*)malloc(sizeof(int));

int size;

for (int i = 1; i < package\_size; i++)

{

if (package[i] == 0)

{ //file size

i++;

size = package[i];

i++;

if (memcpy(sizes, &package[i], size) == NULL)

{

printf("ERROR\n");

free(sizes);

return -1;

}

sscanf(sizes, "%d", file\_size);

i += size;

}

if (package[i] == 1)

{ //file name

i++;

size = package[i];

i++;

if (memcpy(file\_name, &package[i], size) == NULL)

{

printf("ERROR\n");

free(sizes);

return -1;

}

i += size-1;

}

}

free(sizes);

return 0;

}

**Anexo VII –** Código fonte de constants.h.

#ifndef CONSTANTS\_H

#define CONSTANTS\_H

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <termios.h>

#include <unistd.h>

#include <stdlib.h>

#include <string.h>

#include <stdio.h>

#define BAUDRATE B38400

#define \_POSIX\_SOURCE 1 /\* POSIX compliant source \*/

#define FALSE 0

#define TRUE 1

#define BIT(n) 1 << n

#define MAX\_SIZE 255

#define MAX\_FRAME\_SIZE 1500

#define LINE\_SIZE 300 //Higher than 255

#define FLAG 0x7E

#define A\_E 0x03 //Comandos do Emissor e Respostas do Receptor

#define A\_R 0x01 //Comandos do Receptor e Respostas do Emissor

#define C\_SET 0x03

#define C\_DISC 0x0B

#define C\_UA 0x07

#define C\_RR\_ONE 0x85

#define C\_RR\_ZERO 0x05

#define C\_REJ\_ONE 0x11

#define C\_REJ\_ZERO 0x01

#define C\_I\_ONE 0x40

#define C\_I\_ZERO 0x00

#define BCC\_SET (A\_E^C\_SET) //SET is always sent by emissor

#define BCC\_UA\_E (A\_E^C\_UA)

#define BCC\_UA\_R (A\_R^C\_UA)

#define BCC\_DISC\_E (A\_E^C\_DISC)

#define BCC\_DISC\_R (A\_R^C\_DISC)

#define BCC\_C\_I\_ONE (A\_E^C\_I\_ONE)

#define BCC\_C\_I\_ZERO (A\_E^C\_I\_ZERO)

#define BCC\_RR\_ONE (A\_E^C\_RR\_ONE)

#define BCC\_RR\_ZERO (A\_E^C\_RR\_ZERO)

#define BCC\_REJ\_ONE (A\_E^C\_REJ\_ONE)

#define BCC\_REJ\_ZERO (A\_E^C\_REJ\_ZERO)

#define ESC 0x7D

#define FLAG\_FOUND 0x5E //FLAG ^ 0x20

#define ESC\_FOUND 0x5D //ESC ^ 0x20

/\*

Values for the variable status from the linkLayer struct

\*/

#define TRANSMITTER 0

#define RECEIVER 1

/\*

Values for control byte (application' control and data packages)

\*/

#define CTRL\_DATA 1

#define CTRL\_START 2

#define CTRL\_END 3

#endif

**Anexo VIII –** Código fonte de data\_protocol.h.

#ifndef DATA\_PROTOCOL\_H

#define DATA\_PROTOCOL\_H

typedef enum

{

START,

FLAG\_RCV,

A\_RCV,

C\_RCV,

BCC\_OK,

D\_RCV,

STOP

} messageState;

/\*\*

\* Function that writes supervision and non numbered tramas (SET, DISC, UA, RR, REJ).

\* @param command: int from 0 to 6, to specificy which trama is to be sent (SET, DISC, UA, RR with sequence number one,

RR with sequence number zero, REJ with sequence number one or REJ with sequence number zero).

\* @param sender: indicates who wishes to send the trama, TRANSMITTER or RECEIVER.

\* @param fd: file Descriptor of the serial port where to write the trama.

\* @return: number of bytes written (return value of function write) or -1 in case of error.

\*/

int send\_cmd(int command, int sender, int fd);

/\*\*

\* Function that reads the information trama and stores the information portion of it to an array.

\* @param info\_trama: array where information portion of information trama is stored.

\* @param cmd: variable where C (campo de controlo) is stored.

\* @param fd: file Descriptor of the serial port where to read trama from.

\* @return: size of info\_trama array or -1 in case of error.

\*/

int read\_info\_trama(u\_int8\_t\* info\_trama,u\_int8\_t\*cmd, int fd);

/\*\*

\* Function that reads supervision and non numbered tramas (SET, DISC, UA, RR, REJ).

\* @param cmd: variable where C (campo de controlo) is stored.

\* @param fd: file Descriptor of the serial port where to read trama from.

\* @return: 0 or -1 in case of error.

\*/

int read\_cmd(u\_int8\_t\* cmd, int fd);

#endif

**Anexo IX –** Código fonte de data\_protocol.c.

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <termios.h>

#include <stdio.h>

#include <signal.h>

#include <string.h>

#include <stdlib.h>

#include "data\_protocol.h"

#include "constants.h"

#include "alarm.h"

int send\_cmd(int command, int sender, int fd)

{

/\*Function to use for every cmd except when it's an info trama\*/

u\_int8\_t cmd[5];

if (sender == TRANSMITTER)

{ //Emitter

switch (command)

{

case 0: // SET

cmd[0] = FLAG;

cmd[1] = A\_E;

cmd[2] = C\_SET;

cmd[3] = BCC\_SET;

cmd[4] = FLAG;

break;

case 1: // DISC

cmd[0] = FLAG;

cmd[1] = A\_E;

cmd[2] = C\_DISC;

cmd[3] = BCC\_DISC\_E;

cmd[4] = FLAG;

break;

case 2: // UA

cmd[0] = FLAG;

cmd[1] = A\_E;

cmd[2] = C\_UA;

cmd[3] = BCC\_UA\_E;

cmd[4] = FLAG;

break;

case 3: // RR

cmd[0] = FLAG;

cmd[1] = A\_E;

cmd[2] = C\_RR\_ONE;

cmd[3] = BCC\_RR\_ONE;

cmd[4] = FLAG;

break;

case 4: // RR

cmd[0] = FLAG;

cmd[1] = A\_E;

cmd[2] = C\_RR\_ZERO;

cmd[3] = BCC\_RR\_ZERO;

cmd[4] = FLAG;

break;

case 5:

cmd[0] = FLAG;

cmd[1] = A\_E;

cmd[2] = C\_REJ\_ONE;

cmd[3] = BCC\_REJ\_ONE;

cmd[4] = FLAG;

break;

case 6:

cmd[0] = FLAG;

cmd[1] = A\_E;

cmd[2] = C\_REJ\_ZERO;

cmd[3] = BCC\_REJ\_ZERO;

cmd[4] = FLAG;

break;

}

}

printf("I'M SENDING THIS COMMAND: %02x | %02x | %02x | %02x | %02x \n", cmd[0], cmd[1], cmd[2], cmd[3], cmd[4]);

int res = write(fd, cmd, 5);

if (res == -1)

{

printf("ERROR IN SENDING.\n");

}

return res;

}

int read\_info\_trama(u\_int8\_t \*info\_trama, u\_int8\_t \*cmd, int fd)

{

int r=-1;

u\_int8\_t byte\_received;

messageState state = START;

//should check the value of BCC in order to see if we can move on to BCC\_OK state

static int is\_bcc\_okay = 0;

while (TRUE)

{

if (read(fd, &byte\_received, 1) < 0)

{

printf("ERROR\n");

return -1;

}

switch (state)

{

case START:

r=0;

if (byte\_received == FLAG)

{

is\_bcc\_okay = 0;

state = FLAG\_RCV;

}

break;

case FLAG\_RCV:

if (byte\_received == A\_E)

{ //info trama is only sent by TRANSMITTER

is\_bcc\_okay ^= byte\_received;

state = A\_RCV;

}

else if (byte\_received != FLAG)

{

state = START;

}

break;

case A\_RCV:

//C Campo de Controlo 0 S 0 0 0 0 0 0 S = N(s) -->slide 7

if (byte\_received == C\_I\_ONE || byte\_received == C\_I\_ZERO)

{

is\_bcc\_okay ^= byte\_received;

cmd = byte\_received;

state = C\_RCV;

}

else if (byte\_received == FLAG)

{

state = FLAG\_RCV;

}

else

{

state = START;

}

break;

case C\_RCV:

is\_bcc\_okay ^= byte\_received;

if (is\_bcc\_okay == 0)

{

state = BCC\_OK;

}

else if (byte\_received == FLAG)

{

state = FLAG\_RCV;

}

else

{

state = START;

}

break;

case BCC\_OK:

if (byte\_received == FLAG)

{

state = FLAG\_RCV;

}

else

{

info\_trama[r] = byte\_received;

r++;

state = D\_RCV;

}

break;

case D\_RCV:

if (byte\_received == FLAG)

{

state = STOP;

return r;

}

else

{

info\_trama[r] = byte\_received;

r++;

}

break;

case STOP:

return r;

}

}

return -1;

}

int read\_cmd(u\_int8\_t\*cmd, int fd)

{

u\_int8\_t byte\_received;

messageState state = START;

//should check the value of BCC in order to see if we can move on to BCC\_OK state

static int is\_bcc\_okay = 0;

while (TRUE)

{

if (read(fd, &byte\_received, 1) < 0){

return -1;

}

switch (state)

{

case START:

if (byte\_received == FLAG)

{

is\_bcc\_okay = 0;

state = FLAG\_RCV;

}

break;

case FLAG\_RCV:

if (byte\_received == A\_E || byte\_received == A\_R)

{

is\_bcc\_okay ^= byte\_received;

state = A\_RCV;

}

else if (byte\_received != FLAG)

{ //according to the teacher's state machine if it's a FLAG we should stay in this state if not we should go to START

state = START;

}

break;

case A\_RCV:

if (byte\_received == C\_UA || byte\_received == C\_REJ\_ONE || byte\_received == C\_REJ\_ZERO || byte\_received == C\_RR\_ONE || byte\_received == C\_RR\_ZERO || byte\_received == C\_SET || byte\_received == C\_DISC)

{

\*cmd = byte\_received; //need this to know how the previous message was received

is\_bcc\_okay ^= byte\_received;

state = C\_RCV;

}

else if (byte\_received == FLAG)

{

state = FLAG\_RCV;

}

else

{

state = START;

}

break;

case C\_RCV:

is\_bcc\_okay ^= byte\_received;

if (is\_bcc\_okay == 0)

{

state = BCC\_OK;

}

else if (byte\_received == FLAG)

{

state = FLAG\_RCV;

}

else

{

state = START;

}

break;

case BCC\_OK:

if (byte\_received == FLAG)

{

state = STOP;

return 0;

}

else

{

state = START;

}

break;

}

}

return -1;

}

**Anexo X –** Código fonte de protocol\_app.h.

#ifndef PROTOCOL\_APP\_H

#define PROTOCOL\_APP\_H

struct linkLayer

{

char port[20]; /\*Dispositivo /dev/ttySx, x = 0, 1\*/

int baudRate; /\*Velocidade de transmissão\*/

unsigned int sequenceNumber; /\*Número de sequência da trama: 0, 1\*/

unsigned int timeout; /\*Valor do temporizador: 1 s\*/

unsigned int numTransmissions; /\*Número de tentativas em caso de falha\*/

int fileDescriptor; /\*File descriptor da porta série\*/

int status; /\*TRANSMITTER ou RECEIVER\*/

};

struct linkLayer link\_info;

/\*\*

\* Function that opens serial ports and connects transmitter and receiver.

\* @return: 0 or -1 in case of error.

\*/

int llopen();

/\*\*

\* Function that writes information from the buffer into the serial port, sending it to the receiver.

\* @param buffer: information to be written.

\* @param length: size of buffer.

\* @return: number of characters written in sucess or -1 in case of error.

\*/

int llwrite(u\_int8\_t\*buffer,int length);

/\*\*

\* Function that reads the information in the serial port.

\* @param buffer: array to store the information.

\* @return: number of characters read and stored in buffer or -1 in case of error.

\*/

int llread(u\_int8\_t\*buffer);

/\*\*

\* Function that closes the connection between transmitter and receiver and the serial ports.

\* @return: 0 or -1 in case of error.

\*/

int llclose();

/\*\*

\* Function that creates the information trama.

\* @param buffer: array that contains the information to be placed in the information part of the trama.

\* @param trama: array to store the trama contents.

\* @param length: size of buffer.

\* @param s\_writer: current sequece number.

\* @return: size of trama array.

\*/

int create\_info\_trama(u\_int8\_t\*buffer,u\_int8\_t\*trama,int length,int s\_writer);

/\*\*

\* Function that creates BCC2 by doing the XOR operation with all the components of the info\_trama array (that has data).

\* @param info\_trama: array that contains data.

\* @param BCC2: array to store the value of BCC2.

\* @param length: size of info\_trama array

\*/

void check\_BCC2(u\_int8\_t \* info\_trama, u\_int8\_t\* BCC2, int length);

#endif

**Anexo XI –** Código fonte de protocol\_app.c.

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <termios.h>

#include <stdio.h>

#include <signal.h>

#include "constants.h"

#include "data\_protocol.h"

#include "stuffing.h"

#include "protocol\_app.h"

#include "alarm.h"

struct termios newtio, oldtio;

int llopen()

{ //Don't need any of the arguments in the slides because all the data is stored in data structures

if (link\_info.status != TRANSMITTER && link\_info.status != RECEIVER)

{ //sender will be TRANSMITTER or RECEIVER

printf("ERROR");

return -1;

}

u\_int8\_t cmd;

int res = -1;

int fd = open(link\_info.port, O\_RDWR | O\_NOCTTY);

if (fd < 0)

{

perror(link\_info.port);

return -1;

}

link\_info.fileDescriptor=fd;

if (tcgetattr(fd, &oldtio) == -1)

{

printf("ERROR\n");

return -1;

}

bzero(&newtio, sizeof(newtio));

if(link\_info.baudRate==-1){

newtio.c\_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;

}

else{

newtio.c\_cflag = link\_info.baudRate | CS8 | CLOCAL | CREAD;

}

newtio.c\_iflag = IGNPAR;

newtio.c\_oflag = 0;

/\* set input mode (non-canonical, no echo,...) \*/

newtio.c\_lflag = 0;

newtio.c\_cc[VTIME] = 0; /\* inter-character timer unused \*/

newtio.c\_cc[VMIN] = 1; /\* blocking read until 1 chars received \*/

tcflush(fd, TCIOFLUSH);

if (tcsetattr(fd, TCSANOW, &newtio) == -1)

{

printf("ERROR\n");

return -1;

}

if (link\_info.status == TRANSMITTER)

{

while (res != 0)

{

alarm(link\_info.timeout);

if (send\_cmd(0, TRANSMITTER, fd) < 0)

{ //Send SET

printf("ERROR\n");

}

else

{

printf("Wrote SET\n");

}

if ((res = read\_cmd(&cmd, fd)) >= 0){

printf("Received UA\n");

}

}

if (res == 0){

deactivate\_alarm(); //We got a response so timeout is over

}

}

else

{

while (res < 0)

{

read\_cmd(&cmd, fd);

printf("Read SET\n");

if ((res = send\_cmd(2, TRANSMITTER, fd)) < 0)

{

printf("ERROR\n");

}

else

{

printf("Sent UA\n");

}

}

}

return 0;

}

int llwrite(u\_int8\_t \*buffer, int length)

{

/\*Orders protocol to send the Info trauma

In order to make this function work->need to build the information trama

F A C BCC1 buffer BCC2 F

TRANSMITTER is the only one who calls this function

\*/

static int seqNum = 0; //keeps track of sequence number

static int retransmit=0;

u\_int8\_t \*trama = (u\_int8\_t \*)malloc(MAX\_FRAME\_SIZE \* sizeof(u\_int8\_t)); //Allocs space to write info trama

u\_int8\_t \*copy = (u\_int8\_t \*)malloc(MAX\_FRAME\_SIZE \* sizeof(u\_int8\_t)); //Allocs space to write info trama

if (length < 0)

{

printf("Value should be positive in order to actually transfer data\n");

free(trama);

return -1;

}

int write\_length;

u\_int8\_t cmd;

memcpy(copy,buffer,length);

//Need to create info trauma + send it

while (TRUE)

{

if(retransmit){ //ensure we have information for potential retransmissions

memcpy(buffer,copy,length);

memcpy(copy,buffer,length);

}

memset(trama, 0, strlen(trama)); //initialize trama array

write\_length = create\_info\_trama(buffer, trama, length,seqNum);

alarm(link\_info.timeout);

if (write(link\_info.fileDescriptor, trama, write\_length) < 0)

{

printf("ERROR");

}

else

{

printf("Sent message with sequence number %d\n", seqNum);

}

if (read\_cmd(&cmd, link\_info.fileDescriptor) < 0)

{

printf("ERROR");

}

else

{

printf("Read ACK with sequence number %d\n", !seqNum);

}

//RR->means it was accepted

if ((cmd == C\_RR\_ONE && seqNum == 0) || (cmd == C\_RR\_ZERO && seqNum == 1))

{

deactivate\_alarm();

if(seqNum == 0){

seqNum=1;

}

else{

seqNum=0;

}

free(trama);

retransmit=0;

return write\_length;

}

else if (cmd == C\_REJ\_ZERO || cmd == C\_REJ\_ONE)

{

deactivate\_alarm();

printf("Received REJ\n");//Need to retransmit message

retransmit = 1;

}

else{

retransmit=1;

}

}

}

void check\_BCC2(u\_int8\_t \* info\_trama, u\_int8\_t\* BCC2, int length)

{

//Creates BCC2 by doing the XOR operation with all the components of the trama(data) array

for (int i = 0 ; i < length; i++){

\*BCC2 ^= info\_trama[i];

}

}

int llread(u\_int8\_t \*buffer)

{

/\* RECEIVER is the only one who calls this function

\*/

static int seqNum=0; //keeps track of the sequence Number

int length;

u\_int8\_t cmd;

while (TRUE)

{

//read info trama-->can't use read\_cmd because of the data segment

if ((length = read\_info\_trama(buffer,&cmd, link\_info.fileDescriptor)) < 0)

{

}

else

{

printf("Read info message with sequence number %d\n", seqNum);

}

//Need to destuff before storing

int new\_length=destuffing(buffer, length);

//Check if BCC2 is correct, if not dump info

//First we create BCC2 based on the info and then we check if it matches up with the last bit of info trama (that should be BCC2)

u\_int8\_t BCC2;

BCC2 = 0x00;

check\_BCC2(buffer,&BCC2,new\_length-1);

/\*Okay let's interpret slide 11:

We need to check if BCC2 is right however maybe this is not the first thing we need to do

We send REJ if BCC2 is wrong and if info is new

We send RR: 1) if BCC2 is wrong but info is duplicated

2) if BCC2 is right but info is duplicated-->just dump it afterwards

3) if BCC2 is right and info is new-->save it

\*/

int bcc2\_is\_not\_okay = FALSE;

if (BCC2 != buffer[new\_length - 1])

{

printf("Wrong BCC2 - gonna send negative ACK (REJ)\n");

bcc2\_is\_not\_okay = TRUE;

}

//Info is duplicated

if ((cmd == C\_I\_ZERO && seqNum == 1) || (cmd == C\_I\_ONE && seqNum == 0))

{

//Send RR - Slide 14

if (cmd==C\_I\_ZERO) //I(Ns=0)

{

send\_cmd(3, TRANSMITTER, link\_info.fileDescriptor); //RR(Nr=1)

}

else //I(Ns=1)

{

send\_cmd(4, TRANSMITTER, link\_info.fileDescriptor);//RR(Nr=0)

}

}

else

{ //info is new

if (bcc2\_is\_not\_okay)

{

//Send REJ

if (cmd==C\_I\_ZERO) //I(Ns=0)

{

send\_cmd(5, TRANSMITTER, link\_info.fileDescriptor); //REJ(Nr=1)

}

else //I(Ns=1)

{

send\_cmd(6, TRANSMITTER, link\_info.fileDescriptor); //REJ(Nr=0)

}

continue;

}

else

{

//Send RR + store info

//Change sequence number

if (seqNum==0) //I(Ns=0)

{

send\_cmd(3, TRANSMITTER, link\_info.fileDescriptor); //RR(Nr=1)

seqNum=1;

}

else //I(Ns=1)

{

send\_cmd(4, TRANSMITTER, link\_info.fileDescriptor);//RR(Nr=0)

seqNum=0;

}

return length;

}

}

}

return -1;

}

int llclose() //Don't need any of the arguments in the slides because all the data is stored in data structures

{

int cmd\_received = FALSE;

u\_int8\_t cmd;

if (link\_info.status!= TRANSMITTER && link\_info.status != RECEIVER)

{ //sender will be TRANSMITTER or RECEIVER

printf("ERROR\n");

}

if (link\_info.status == TRANSMITTER)

{

while (!cmd\_received)

{

alarm(link\_info.timeout);

if (send\_cmd(1, TRANSMITTER, link\_info.fileDescriptor) < 0) //Send DISC

{

printf("ERROR\n");

}

if (read\_cmd(&cmd, link\_info.fileDescriptor) < 0) //Read DISC from Receiver

{

printf("Try again\n");

}

if (cmd == C\_DISC)

{

deactivate\_alarm();

cmd\_received = TRUE;

}

if (send\_cmd(2, TRANSMITTER, link\_info.fileDescriptor) < 0) //Send UA

{

printf("ERROR\n");

}

}

}

else

{

while (!cmd\_received)

{

if (read\_cmd(&cmd, link\_info.fileDescriptor) < 0) //Read DISC

{

printf("ERROR\n");

}

else

{

if (cmd == C\_DISC)

{

cmd\_received = TRUE;

}

}

}

if (send\_cmd(1, TRANSMITTER, link\_info.fileDescriptor) < 0) //Send DISC back

{

printf("ERROR\n");

continue;

}

if (read\_cmd(&cmd, link\_info.fileDescriptor) < 0) //Read UA

{

printf("ERROR\n");

}

}

//close fd

if (tcsetattr(link\_info.fileDescriptor, TCSANOW, &oldtio) == -1)

{

perror("tcsetattr");

return -1;

}

close(link\_info.fileDescriptor);

return 0;

}

int create\_info\_trama(u\_int8\_t \*buffer, u\_int8\_t\*trama, int length,int seqNum)

{

//Let's start by defining BCC2-->it will need to be stuffed (like data) but according to slide 7 and 13, it is created before

u\_int8\_t\* BCC2 =(u\_int8\_t\*)malloc(sizeof(u\_int8\_t));

BCC2[0] = 0x00; //neutral operand

check\_BCC2(buffer,BCC2,length);

//We need to stuff the data + BCC2

int bcc2\_length=1;

int new\_bcc2\_length = stuffing(BCC2, bcc2\_length);

int new\_data\_length = stuffing(buffer, length);

//Assemble info trama

int new\_length = new\_data\_length + new\_bcc2\_length + 5; //5 because F A C BCC1 F

trama[0] = FLAG;

trama[1] = A\_E;

if (seqNum == 0)

{

trama[2] = C\_I\_ZERO;

trama[3] = BCC\_C\_I\_ZERO;

}

else

{

trama[2] = C\_I\_ONE;

trama[3] = BCC\_C\_I\_ONE;

}

memcpy(&trama[4],buffer,new\_data\_length);

int index\_bcc2 = new\_data\_length + 4;

memcpy(&trama[index\_bcc2], BCC2, new\_bcc2\_length);

trama[new\_length - 1] = FLAG; //second flag is at the end of the frame

return new\_length;

}

**Anexo XII –** Código fonte de receiver.c.

//RECEIVER

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <termios.h>

#include <stdio.h>

#include <signal.h>

#include <string.h>

#include <stdlib.h>

#include "app.h"

#include "protocol\_app.h"

#include "constants.h"

// ./receiver /dev/ttySx -f file name

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

{

u\_int8\_t \*package = (u\_int8\_t \*)malloc(sizeof(u\_int8\_t) \* MAX\_FRAME\_SIZE);

u\_int8\_t \*frame = (u\_int8\_t \*)malloc(sizeof(u\_int8\_t) \* MAX\_FRAME\_SIZE);

link\_info.status = RECEIVER;

char \*filename = (char\*)malloc(sizeof(char)\*MAX\_FRAME\_SIZE);

char \*new\_file = (char\*)malloc(sizeof(char)\*MAX\_FRAME\_SIZE);

int file\_size;

int index=-1;

FILE \*fptr;

//Parse arguments

for (int i = 1; i < argc; i++)

{

if (strcmp(argv[i], "") != 0)

{

if (!strncmp(argv[i], "/dev/ttyS", 9))

{

strcpy(link\_info.port, argv[i]);

}

if (!strcmp(argv[i], "-f"))

{

index=i+1;

}

}

}

//Open connection between app and data protocol

//And connection between TRANSMITTER and RECEIVER

if(index==-1){

printf("ERROR: Please try again and input a name for the new file so we can start the transmission process\n");

return 1;

}

else{

new\_file=argv[index];

}

llopen();

int size;

int received\_ctrl\_pack\_start = FALSE;

//Gonna receive a Control Package with START

while (!received\_ctrl\_pack\_start)

{

if ((size = llread(package)) < 0)

{

printf("ERROR\n");

free(package);

free(frame);

return 1;

}

if (package[0] == CTRL\_START)

{

if (read\_control\_package(package, filename, &file\_size, size) < 0)

{

printf("ERROR\n");

free(package);

free(frame);

return 1;

}

received\_ctrl\_pack\_start = TRUE;

}

}

if ((fptr = fopen(new\_file, "w")) == NULL)

{

printf("ERROR\n");

free(package);

free(frame);

return 1;

}

//Gonna receive Data Packages as well and a Control package with END

int not\_end = FALSE;

while (!not\_end)

{

if ((size = llread(package)) < 0)

{

printf("ERROR\n");

free(package);

free(frame);

return 1;

}

if (package[0] == CTRL\_DATA)

{

if ((size = read\_data\_package(frame, package)) < 0)

{

printf("ERROR\n");

free(package);

free(frame);

return 1;

}

fwrite(frame, sizeof(u\_int8\_t), size, fptr);

}

memset(frame,0,strlen(frame));

if (package[0] == CTRL\_END)

{

u\_int8\_t\*end[MAX\_FRAME\_SIZE];

int end\_fsize;

if (read\_control\_package(package, end, &end\_fsize, size) < 0)

{

printf("ERROR\n");

free(package);

free(frame);

return 1;

}

not\_end = TRUE;

//If everything went well with the transmission (no data lost), the size of the og file and the new one will be the same

if(file\_size != end\_fsize){

printf("ERROR\n");

}

}

}

if( (fclose(fptr)) == EOF ) {

printf("ERROR\n");

return 1;

}

//Close connection

if (llclose() < 0)

{

printf("ERROR\n");

free(package);

free(frame);

return 1;

}

free(package);

free(frame);

free(filename);

return 0;

}

**Anexo XIII –** Código fonte de stuffing.h.

#ifndef STUFFING\_H

#define STUFFING\_H

/\*\*

\* Function that executes the stuffing process in a frame, adding new bytes to it depending on the bytes found in it. Called before trama is sent to receiver.

\* @param buffer: frame.

\* @param length: size of frame buffer.

\* @return: size of the new frame.

\*/

int stuffing(u\_int8\_t\* buffer, int length);

/\*\*

\* Function that executes the destuffing process in a frame, removing specific bytes from it. Called by the receiver in order to store information.

\* @param buffer: frame.

\* @param length: size of frame buffer.

\* @return: size of the new frame.

\*/

int destuffing(u\_int8\_t\* buffer, int length);

#endif

**Anexo XIV –** Código fonte de stuffing.c.

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <termios.h>

#include <stdio.h>

#include <signal.h>

#include <string.h>

#include <stdlib.h>

#include "constants.h"

int stuffing(u\_int8\_t \*buffer, int length)

{

/\*The new frame needs to have a different length than the og frame

and by different I mean bigger, everytime it finds ESC or FLAG we need to replace it with FLAG and ESC (respectively)

and add a ESC\_FOUND and FLAG\_FOUND (respectively)

\*/

int more=0;

for(int i=0;i<length;i++){

if(buffer[i] == FLAG || buffer[i] == ESC){ //sees if it can find any Flags or ESCs inside of trama, if so this means one extra space because we have to add an ESC\_FOUND or FLAG\_FOUND

more++;

}

}

int nl = more + length;

u\_int8\_t\*frame = (u\_int8\_t\*)malloc(sizeof(u\_int8\_t)\*nl);

int counter=0;

int index=0;

for (int i = 0; i < length; i++)

{

index = i + counter;

if (buffer[i] == FLAG)

{ //Found 0x7E inside the trama

frame[index] = ESC;

frame[index + 1] = FLAG\_FOUND;

counter ++;

}

else if (buffer[i] == ESC)

{ //Found 0x7D inside the trama

frame[index] = ESC;

frame[index + 1] = ESC\_FOUND;

counter++;

}

else

{

frame[index] = buffer[i];

}

}

memcpy(buffer,frame,nl);

return nl;

}

int destuffing(u\_int8\_t \*buffer, int length)

{

//With stuffing the frame goes back to its original size

u\_int8\_t\*frame = (u\_int8\_t \*)malloc(sizeof(u\_int8\_t)\*length);

int new\_size = 0;

for (int i = 0; i < length; i++)

{

if (buffer[i] == ESC) //common point

{

if(buffer[i+1] == FLAG\_FOUND){

frame[new\_size]=FLAG;

}

else if(buffer[i+1] == ESC\_FOUND){

frame[new\_size]=ESC;

}

i++;

}

else

{

frame[new\_size] = buffer[i];

}

new\_size++;

}

memcpy(buffer,frame,new\_size);

free(frame);

return new\_size;

}

**Anexo XV –** Código fonte de transmitter.c.

//TRANSMITTER

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <termios.h>

#include <stdio.h>

#include <signal.h>

#include <string.h>

#include <stdlib.h>

#include "app.h"

#include "constants.h"

#include "protocol\_app.h"

#include "alarm.h"

// ./transmitter /dev/ttySx -t TIMEOUT -n numTries -f filename

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

{

u\_int8\_t \*package[MAX\_FRAME\_SIZE]; //it should be bigger than 255 because K=256\*L2+L1

link\_info.status = TRANSMITTER;

int index\_t=-1,index\_n=-1,index\_f=-1;

FILE \*fprt;

//Parse arguments

for (int i = 1; i < argc; i++)

{

if (strcmp(argv[i], "") != 0)

{

if (!strncmp(argv[i], "/dev/ttyS", 9))

{

strcpy(link\_info.port, argv[i]);

}

if (!strcmp(argv[i], "-t"))

{

link\_info.timeout = atoi(argv[i + 1]);

index\_t = i+1;

}

if (!strcmp(argv[i], "-n"))

{

link\_info.numTransmissions = atoi(argv[i + 1]);

index\_n = i+1;

}

if (!strcmp(argv[i], "-f"))

{

index\_f=i+1;

}

}

}

if(index\_f < 0){

printf("ERROR: Please introduce a file so we can start the transmission process\n");

return 1;

}

u\_int8\_t \* filename= argv[index\_f];

if(index\_n < 0){

link\_info.numTransmissions = 3; //default value

}

if(index\_t < 0){

link\_info.timeout = 3; //default

}

struct stat st;

stat(filename, &st);

int size = st.st\_size; //get file's size

//Open file

if ((fprt = fopen(filename, "r")) == NULL)

{

printf("ERROR: file doesn't exist\n");

return -1;

}

install\_alarm();

//Open connection between app and data protocol

//And connection between TRANSMITTER and RECEIVER

llopen();

//Send control package with START

int package\_size;

if ((package\_size = create\_control\_package(CTRL\_START, filename, size, package)) < 0)

{

printf("ERROR\n");

return 1;

}

int write\_length;

if ((write\_length = llwrite(package, package\_size)) < 0)

{

printf("ERROR\n");

return 1;

}

//Keep sending Data packages until the end of the file

int c\_size = LINE\_SIZE;

int n = 0, line\_size = 0;

char\*line[c\_size];

u\_int8\_t \*frame = (u\_int8\_t \*)malloc(sizeof(u\_int8\_t) \* MAX\_FRAME\_SIZE);

while (TRUE)

{

if (size - n \* c\_size < c\_size) //File size - sequence number \* line size

{

c\_size = size % c\_size;

}

if ((line\_size = fread(line, 1, c\_size, fprt)) <= 0)

{ //could be an error or could be EOF

break;

}

if (create\_data\_package(n, line\_size, line, frame) < 0)

{

printf("ERROR\n");

return 1;

}

int frame\_size = line\_size + 4; //4 because line\_size only has the size of P1...Pk, we need it to have C, N, L2 and L1 into consideration

if (llwrite(frame, frame\_size) < 0)

{

printf("ERROR\n");

return 1;

}

n++;

}

//Send control package with END

if ((package\_size = create\_control\_package(CTRL\_END, filename, size, package)) < 0)

{

printf("ERROR\n");

return 1;

}

if ((write\_length = llwrite(package, package\_size)) < 0)

{

printf("ERROR\n");

return 1;

}

//Close connection

if (llclose() < 0)

{

printf("ERROR\n");

return 1;

}

return 0;

}

**Anexo XVI –** Gráfico de Evolução da Eficiência S em função da Percentagem de Erros Aleatórios em BCC1.

**Anexo XVII –** Gráfico de Evolução da Eficiência S em função da Percentagem de Erros Aleatórios em BCC2.

**Anexo XVIII –** Gráfico de Evolução da Eficiência S em função do tempo de propagação.

**Anexo XIX –** Gráfico de Evolução da Eficiência S em função da *Baudrate* (capacidade de ligação).

**Anexo XX –** Gráfico de Evolução da Eficiência S em função do Tamanho da Trama de Informação.

1. 92KB, 200MB, 602MB, 1.8MB, 3.3MB, 7.8MB. [↑](#footnote-ref-1)
2. MP3, JPG, PNG, MP4. [↑](#footnote-ref-2)