

REDES DE COMPUTADORES

PROTOCOLO

DE

LIGAÇÃO DE DADOS



Henrique Silva – up202007242

Tiago Branquinho – up202005567

**SUMÁRIO**

Este relatório foi elaborado no âmbito do primeiro trabalho prático da unidade curricular “Redes de Computadores”, pertencente à Licenciatura em Engenharia Informática e Computação. Neste trabalho foi-nos proposta a transferência de dados usando uma aplicação que recorresse a um protocolo de ligação de dados fiável, mesmo em condições de interrupção e de adição de “ruído” à transmissão. Este trabalho foi implementado em C, no sistema operativo Linux, usando portas série RS-232.

Conclui-se que a proposta foi cumprida com sucesso pois todos os objetivos estabelecidos foram alcançados.

**INTRODUÇÃO**

O objetivo deste trabalho divide-se em duas partes, uma relativa ao protocolo de ligação de dados e outra à aplicação em si. O objetivo do protocolo de ligação de dados é fornecer um serviço de comunicação de dados fiável entre dois sistemas ligados por um canal de transmissão, neste caso uma porta de série. O objetivo da aplicação é desenvolver um protocolo de aplicação muito simples para transferência de um ficheiro, usando um serviço fiável oferecido pelo protocolo de ligação de dados.

O objetivo do relatório é documentar o código, explicando a nossa implementação em função do guião que nos foi apresentado. O relatório tem a seguinte estrutura:

* Arquitetura – Esclarecimento sobre os blocos funcionais e interfaces.
* Estrutura do código – Explicitação das *APIs*, das principais estruturas de dados e funções, e da sua relação com a arquitetura.
* Casos de uso principais – Identificação destes e das suas sequências de chamada de funções.
* Protocolo de ligação lógica – Identificação dos seus principais aspetos funcionais e descrição da estratégia de implementação dos mesmos, apresentando extratos de código.
* Protocolo de aplicação – Identificação dos seus principais aspetos funcionais e descrição da estratégia de implementação dos mesmos, apresentando extratos de código.
* Validação – Descrição dos testes efetuados, (com apresentação quantificada dos resultados)
* Eficiência do protocolo de ligação de dados – Caracterização da mesma, efetuada recorrendo a estatísticas medidas sobre o código desenvolvido. Comparação desta com a de um *protocolo Stop&Wait*.
* Conclusões – Síntese da informação apresentada e reflexão dos objetivos de aprendizagem alcançados.

**ARQUITETURA**

O trabalho segue uma arquitetura em camadas, já que é possível identificar duas camadas independentes: a da aplicação, e a da ligação de dados.

A camada da ligação de dados é a responsável pelo estabelecimento da ligação, garantindo a integridade do protocolo. Assim, é a responsável pela comunicação direta com a porta de série, gerindo a abertura, fecho, leitura e escrita na mesma. É nesta camada que ocorre a verificação de erros, garantindo-se assim a fiabilidade da transferência. A camada da ligação de dados oferece serviços à camada da aplicação, e, deste modo, é a de mais baixo nível deste projeto.

A camada da aplicação é a responsável pelo envio e receção de ficheiros, estando dividida em dois blocos, o emissor e o recetor. Esta camada acede aos serviços fornecidos pela camada de ligação de dados, invocando funções da mesma. Deste modo, é considerada a de mais alto nível deste projeto.

**ESTRUTURA DO CÓDIGO**

A camada da ligação de dados é representada por uma estrutura de ligação de dados. Esta estrutura contém o dispositivo associado à porta de série, a velocidade de transmissão, o número de tentativas em caso de falha, o valor do temporizador e a entidade que está a usufruir dos seus serviços, recetor ou emissor, que é representada por uma enumeração **– Anexo 2.1**.

As principais funções da camada de ligação de dados são llopen(), que abre uma conexão entre o recetor e o emissor, llread(), que permite a leitura de um pacote, llwrite(), que permite o envio de um pacote, llclose(), que fecha a conexão, permitindo a impressão de estatísticas referentes à mesma e, por fim, int set\_up\_port() e int close\_port(), que permitem a abertura e o fecho da porta de série, respetivamente. As duas últimas encontram-se no ficheiro *port\_operations.c* ***– Anexo 1.9 e 2.2***, enquanto as restantes se encontram no ficheiro *link\_layer.c* **– Anexo 1.4 e 2.2**.

A camada da aplicação é representada por uma estrutura de aplicação. Esta estrutura contém o descritor correspondente à porta de série e o estado, recetor ou emissor **– Anexo 2.3**.

As principais funções da camada da aplicação são receiveFile(), sendFile(), buildDataPacket(), parseDataPacket(), buildControlPacket(), parseControlPacket(), e applicationLayer(), a função principal. Esta última, para além de preencher a estrutura LinkLayer, vai invocar, de acordo com o *status*, sendFile(), que permite enviar um ficheiro através da porta de série, ou receiveFile(), que permite a receção de um ficheiro através da porta de série. As funções de *parse* são invocadas pelo recetor, enquanto as de *build* são invocadas pelo emissor **– Anexo 2.4**.

**CASOS DE USO PRINCIPAIS**

O utilizador pode escolher as configurações, o nome do ficheiro a ser enviado e o nome com que se deverá criar no recetor através da consola. Em alternativa pode recorrer ao *Makefile* existente, iniciando o sistema na configuração padrão.

No caso do emissor, o sistema irá invocar a função sendFile(), que, primeiramente, tenta abrir o ficheiro a ser lido. Em seguida, é invocada a função llopen(), de forma a estabelecer uma conexão entre o emissor e o recetor, através do envio de uma trama SET e receção de uma trama UA. Em seguida, é invocada a função buildControlPacket(), que cria um *control packet*, que contém a indicação de inicio de transmissão. Esse *packet* é posteriormente enviado para o recetor, através a função llwrite(). Após este início, é implementado um ciclo que vai lendo porções do ficheiro, criando *data packets* que as contêm, e enviando esses *packets*, recorrendo à função llwrite(). Seguidamente, envia um *control packet* criado pela função buildControlPacket() que sinaliza o final da transmissão. Por fim é invocada a função llclose(), de forma a fechar a conexão com o recetor.

No caso do recetor, o sistema irá invocar a função receiveFile(), que invoca a função llopen(), desta vez recebendo a trama SET e enviando a trama UA. Em seguida, é invocada a função llread() de forma a receber o *control packet* enviado pelo emissor, que posteriormente é processado, através da função parseControlPacket(). Depois disso, o ficheiro a ser criado é aberto, e é implementado um ciclo que vai recebendo e processando *data packets* através da função llread() e parseDataPacket(), respetivamente e, por fim, adicionando a informação presente nesses *packets* ao novo ficheiro criado. Seguidamente, é invocada a função llread() de forma a receber o *control packet* enviado pelo emissor, que posteriormente é processado através da função parseControlPacket(). Por fim, é invocada a função llclose(), de forma a fechar a conexão com o emissor.

**PROTOCOLO DE LIGAÇÃO LÓGICA**

No protocolo de ligação lógica, foram implementadas quatro funções: llopen(), llwrite(), llread() e llclose(). Estas funções servem como interface para o protocolo de aplicação utilizar as funcionalidades do protocolo de ligação lógica, e lidam com as quatro fases da transferência de dados. Começando pelo llopen(), onde é estabelecida a ligação entre o emissor e o recetor, seguem-se llwrite(), onde é enviada uma trama de informação até ser recebida uma resposta de confirmação, e llread(), onde é recebida uma trama de informação e enviada uma resposta de confirmação, e por fim llclose(), onde é finalizada a ligação entre o emissor e o recetor. As leituras de cada trama são feitas através de uma máquina de estados, que as recebe byte a byte, mudando o estado consoante o byte recebido, de modo a que apenas se chega ao estado final se a trama recebida tiver um formato válido.

A função llopen() começa por estabelecer ligação com a porta de série indicada, armazenar as informações enviadas pelo protocolo de aplicação e instalar o *alarm handler* **– Anexo 3.1**. Após realizadas as operações iniciais, esta função procederá de formas diferentes consoante o parâmetro *role*. Caso se trate do emissor será enviadauma trama SET e ficará a aguardar por uma resposta UA para confirmação **– Anexo 3.2**. Caso se trate do recetor, este aguardará por receber uma trama SET enviando por sua vez uma resposta UA após a sua receção para confirmação **– Anexo 3.3**.

A função llwrite() será apenas chamada pelo emissor para enviar tramas e é composta por diversos passos. Em primeiro ocorre a criação de uma trama de informação utilizando a função createIFrame() **– Anexo 3.4**. Em seguida, realiza-se o *stuffing* da trama utilizando a função stuffIFrame() **– Anexo 3.5**. Depois disso, ocorre o envio da trama utilizando a função sendFrame() **– Anexo 3.6**. Por fim, ocorre a receção de uma resposta RR ou REJ utilizando a função readSFrame() **– Anexo 3.7**.Caso se trate de uma resposta RR o programa sairá da função. Caso se trate de uma resposta REJ será repetido o processo até receber uma resposta positiva.Se nunca for recebida uma resposta positiva irá eventualmente ocorrer um *timeout* após um número definido de tentativas, desencadeado pelo *alarm handler,* que manipula as variáveis globais que controlam o loop, *relay* e *stop* **– Anexo 3.8**.

A função llread() será apenas chamada pelo recetor para receber tramas e é composta por diversos passos, retornando o número de bytes da trama recebida. Em primeiro ocorre a leitura de uma trama de informação utilizando a função readIFrame() **– Anexo 3.9**, e em seguida, a realização do *unstuffing* da trama utilizando a função unstuffIFrame() **– Anexo 3.10**. Depois disso, ocorre a verificação do BCC2 e do *sequence number*, e determinação da resposta consoante estes :RR caso o BCC2 esteja correto independentemente do *sequence number* ou caso o BCC2 esteja incorreto e se trate de uma trama duplicada, REJ caso o BCC2 esteja incorreto e se trate de uma nova trama **– Anexo 3.11**. Finalmente ocorre a criação de uma trama de supervisão, tendo em conta a resposta adequada, utilizando a função createSFrame() **– Anexo 3.12**, e o envio dessa trama **– Anexo 3.13**.

A função llclose() procederá de forma diferente dependendo do parâmetro *role*. Caso se trate do emissor, será enviada uma trama de supervisão DISC, ficando a aguardar por outra trama de supervisão DISC enviada pelo recetor. Aquando da receção desta última trama será enviada uma resposta UA para confirmação **– Anexo 3.14**. Caso se trate do recetor, este aguardará por receber uma trama de supervisão DISC, enviando por sua vez outra trama de supervisão DISC após a sua receção, ficando a aguardar por uma resposta UA para confirmação **– Anexo 3.15**. No caso do recetor serão ainda impressas as estatísticas referentes à transferência de dados **– Anexo 3.16**.

**PROTOCOLO DE APLICAÇÃO**

No protocolo de aplicação, foram implementadas sete funções: buildDataPacket(), parseDataPacket(), buildControlPacket(), parseControlPacket(), sendFile(), receiveFile() e applicationLayer(). A função applicationLayer() será responsável pela distinção entre o emissor e recetor, chamando as funções sendFile() ou receiveFile(), respetivamente. Na função sendFile() serão chamadas as funções de construção de *packets*, buildDataPacket() e buildControlPacket(), e na função receiveFile() serão chamadas as funções de *parsing* de *packets*, parseDataPacket() e parseControlPacket().

A função applicationLayer() será responsável por armazenar os dados que mais tarde serão passados ao protocolo de ligação de dados e por distinguir o emissor do recetor **– Anexo 4.1**.

A função sendFile(), que retornará 0 caso a operação tenha sido bem-sucedida e 1 caso contrário, que será apenas chamada pelo emissor e é composta por diversos passos. Em primeiro ocorre a abertura do ficheiro que será enviado, utilizando a função openFile() **– Anexo 4.2**. Em seguida, ocorre o estabelecimento da ligação entre portas de série utilizando a função llopen() **– Anexo 4.3**. Depois disso, ocorre a construção e envio do *control packet* inicial utilizando a função buildControlPacket() e a função llwrite(), respetivamente **– Anexo 4.4**. Segue-se a criação e envio de tramas enquanto houver informação no ficheiro **– Anexo 4.5** e, após isso, a construção e envio do *control packet* final **– Anexo 4.6**. Por fim, a ligação entre as portas de série é fechada usando a função llclose() **– Anexo 4.7** e o ficheiro enviado é fechado **– Anexo 4.8**.

A função receiveFile(), que retornará 0 caso a operação tenha sido bem-sucedida e 1 caso contrário, será apenas chamada pelo recetor e é composta por diversos passos. Em primeiro ocorre o estabelecimento da ligação entre portas de série utilizando a função llopen() **– Anexo 4.3**. Em seguida, ocorre a leitura do *control packet* inicial utilizando a função llread() **– Anexo 4.9**. Após isso, o *control packet* é *parsado*, utilizando a função parseControlPacket() **– Anexo 4.10**. Depois ocorre a abertura do ficheiro de destino utilizando a função openFile() **– Anexo 4.11**, seguida pela leitura e *parsing* de cada cada trama recebida até ao *control packet* final, inclusive **– Anexo 4.12**. Seguidamente o ficheiro de destino é fechado utilizando a função closeFile() **– Anexo 4.13**, o *control packet* final é *parsado* utilizando a função parseControlPacket() **– Anexo 4.14** e ocorre a verificação de que o ficheiro original e o ficheiro de destino são idênticos **– Anexo 4.15**. Por fim a ligação entre as portas de série é fechada utilizando a função llclose() **– Anexo 4.16**.

**VALIDAÇÃO**

De forma a verificar a integridade do programa foram efetuados diversos testes, consistindo o mais simples no envio de um ficheiro sem interrupções. Foi também testado o envio de um ficheiro com a ligação interrompida em vários momentos da transferência, o envio de um ficheiro numa ligação com ruído em vários momentos da transferência, e por fim, o envio de um ficheiro com diferentes tamanhos para as tramas de informação.

Todos os testes realizados foram concluídos com sucesso, estando o projeto desenvolvido completamente funcional.

Para testar a eficiência do protocolo de dados foram desenhados testes que foram executados mais do que uma vez para melhor precisão estatística. O ficheiro utilizado para o efeito de teste foi *penguin.gif* e o tamanho máximo de pacote informação definido foi de 1024B.

Variando a percentagem de erros nos blocos BCC1 e BCC2 foram detetados resultados distintos. Isto pode ser explicado pelo facto de que, quando existe um erro no BCC1 irá ocorrer um *timeout* após 3 segundos, enquanto no BCC2 a trama será imediatamente reenviada. Assim, podemos verificar que nos dois casos o aumento da percentagem de erros na trama implica diminuições significativas na eficiência, sendo estas mais acentuadas no caso de erros no BCC1.

Variando o tempo de propagação de cada trama, utilizando a função usleep(), foi possível concluir que quanto maior o tempo de propagação menor a eficiência.

Variando a capacidade de transferência do protocolo, ao alterar a macro *BAUDRATE*, podemos observar que quando esta aumenta a eficiência diminui.

Variando o tamanho dos pacotes de informação, ao alterar a macro *MAX\_DATA\_SIZE*, podemos constatar que quando esta aumenta a eficiência também aumenta. Os valores do gráfico apresentado foram obtidos considerando um *Baudrate* de 19200, ao contrário de todos os outros, em que o *Baudrate* considerado era de 9600. Este novo valor é necessário quando os pacotes de informação transferidos são maiores.

**CONCLUSÕES**

A realização deste projeto permitiu-nos obter uma melhor compreensão de um protocolo de transferência de dados, nomeadamente ao nível da máquina de estados, construção de tramas e independência entre camadas. Concluímos também que o trabalho realizado cumpriu todas as expectativas ao superar todos os testes e ao distinguir corretamente a camada de ligação de dados da camada de aplicação.

**ANEXOS**

ANEXO 1 – Código fonte

* 1. –*Ficheiro main.c.*

// Main file of the serial port project.

// NOTE: This file must not be changed.

#include <stdio.h>

#include <stdlib.h>

#include "application\_layer.h"

#define BAUDRATE 9600

#define N\_TRIES 3

#define TIMEOUT 4

// Arguments:

// $1: /dev/ttySxx

// $2: tx | rx

// $3: filename

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

{

if (argc < 4)

{

printf("Usage: %s /dev/ttySxx tx|rx filename\n", argv[0]);

exit(1);

}

const char \*serialPort = argv[1];

const char \*role = argv[2];

const char \*filename = argv[3];

printf("Starting link-layer protocol application\n"

" - Serial port: %s\n"

" - Role: %s\n"

" - Baudrate: %d\n"

" - Number of tries: %d\n"

" - Timeout: %d\n"

" - Filename: %s\n",

serialPort,

role,

BAUDRATE,

N\_TRIES,

TIMEOUT,

filename);

applicationLayer(serialPort, role, BAUDRATE, N\_TRIES, TIMEOUT, filename);

return 0;

}

* 1. –*Ficheiro cable.c.*

// Virtual cable program to test serial port.

// Creates a pair of virtual Tx / Rx serial ports using "socat".

//

// Author: Manuel Ricardo [mricardo@fe.up.pt]

// Modified by: Eduardo Nuno Almeida [enalmeida@fe.up.pt]

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <termios.h>

#include <unistd.h>

// Baudrate settings are defined in <asm/termbits.h>, which is

// included by <termios.h>

#define BAUDRATE B38400

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

#define FALSE 0

#define TRUE 1

#define BUF\_SIZE 2048

typedef enum

{

CableModeOn,

CableModeOff,

CableModeNoise,

} CableMode;

// Returns: serial port file descriptor (fd).

int openSerialPort(const char \*serialPort, struct termios \*oldtio, struct termios \*newtio)

{

int fd = open(serialPort, O\_RDWR | O\_NOCTTY);

if (fd < 0)

return -1;

// Save current port settings

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

return -1;

memset(newtio, 0, sizeof(\*newtio));

newtio->c\_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;

newtio->c\_iflag = IGNPAR;

newtio->c\_oflag = 0;

newtio->c\_lflag = 0;

newtio->c\_cc[VTIME] = 1; // Inter-character timer unused

newtio->c\_cc[VMIN] = 0; // Read without blocking

tcflush(fd, TCIOFLUSH);

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

return -1;

return fd;

}

// Add noise to a buffer, by flipping the byte in the "errorIndex" position.

void addNoiseToBuffer(unsigned char \*buf, size\_t errorIndex)

{

buf[errorIndex] ^= 0xFF;

}

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

{

printf("\n");

system("socat -dd PTY,link=/dev/ttyS10,mode=777 PTY,link=/dev/emulatorTx,mode=777 &");

sleep(1);

printf("\n");

system("socat -dd PTY,link=/dev/ttyS11,mode=777 PTY,link=/dev/emulatorRx,mode=777 &");

sleep(1);

printf("\n\n"

"Transmitter must open /dev/ttyS10\n"

"Receiver must open /dev/ttyS11\n"

"\n"

"The cable program is sensible to the following interactive commands:\n"

"--- on : connect the cable and data is exchanged (default state)\n"

"--- off : disconnect the cable disabling data to be exchanged\n"

"--- noise : add fixed noise to the cable\n"

"--- end : terminate the program\n"

"\n");

// Configure serial ports

struct termios oldtioTx;

struct termios newtioTx;

int fdTx = openSerialPort("/dev/emulatorTx", &oldtioTx, &newtioTx);

if (fdTx < 0)

{

perror("Opening Tx emulator serial port");

exit(-1);

}

struct termios oldtioRx;

struct termios newtioRx;

int fdRx = openSerialPort("/dev/emulatorRx", &oldtioRx, &newtioRx);

if (fdRx < 0)

{

perror("Opening Rx emulator serial port");

exit(-1);

}

// Configure stdin to receive commands to this program

int oldf = fcntl(STDIN\_FILENO, F\_GETFL, 0);

fcntl(STDIN\_FILENO, F\_SETFL, oldf | O\_NONBLOCK);

unsigned char tx2rx[BUF\_SIZE] = {0};

unsigned char rx2tx[BUF\_SIZE] = {0};

char rxStdin[BUF\_SIZE] = {0};

CableMode cableMode = CableModeOn;

volatile int STOP = FALSE;

printf("Cable ready\n");

while (STOP == FALSE)

{

// Read from Tx

int bytesFromTx = read(fdTx, tx2rx, BUF\_SIZE);

if (bytesFromTx > 0)

{

if (cableMode == CableModeOff)

{

printf("bytesFromTx=%d > bytesToRx=CONNECTION OFF\n", bytesFromTx);

}

else

{

if (cableMode == CableModeNoise)

{

addNoiseToBuffer(tx2rx, 0);

}

int bytesToRx = write(fdRx, tx2rx, bytesFromTx);

printf("bytesFromTx=%d > bytesToRx=%d\n", bytesFromTx, bytesToRx);

}

}

// Read from Rx

int bytesFromRx = read(fdRx, rx2tx, BUF\_SIZE);

if (bytesFromRx > 0)

{

if (cableMode == CableModeOff)

{

printf("bytesToTx=CONNECTION OFF < bytesFromRx=%d\n", bytesFromRx);

}

else

{

if (cableMode == CableModeNoise)

{

addNoiseToBuffer(rx2tx, 0);

}

int bytesToTx = write(fdTx, rx2tx, bytesFromRx);

printf("bytesToTx=%d < bytesFromRx=%d\n", bytesToTx, bytesFromRx);

}

}

// Read commands from STDIN to control the cable mode

int fromStdin = read(STDIN\_FILENO, rxStdin, BUF\_SIZE);

if (fromStdin > 0)

{

rxStdin[fromStdin - 1] = '\0';

if (strcmp(rxStdin, "off") == 0 || strcmp(rxStdin, "0") == 0)

{

printf("CONNECTION OFF\n");

cableMode = CableModeOff;

}

else if (strcmp(rxStdin, "on") == 0 || strcmp(rxStdin, "1") == 0)

{

printf("CONNECTION ON\n");

cableMode = CableModeOn;

}

else if (strcmp(rxStdin, "noise") == 0 || strcmp(rxStdin, "2") == 0)

{

printf("CONNECTION NOISE\n");

cableMode = CableModeNoise;

}

else if (strcmp(rxStdin, "end") == 0)

{

printf("END OF THE PROGRAM\n");

STOP = TRUE;

}

}

}

// Restore the old port settings

if (tcsetattr(fdRx, TCSANOW, &oldtioRx) == -1)

{

perror("tcsetattr");

exit(-1);

}

if (tcsetattr(fdTx, TCSANOW, &oldtioTx) == -1)

{

perror("tcsetattr");

exit(-1);

}

close(fdTx);

close(fdRx);

system("killall socat");

return 0;

}

* 1. *– Ficheiro application\_layer.c.*

// Application layer protocol implementation

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "application\_layer.h"

#include "link\_layer.h"

#include "file.h"

#include "frame.h"

LinkLayer linklayer;

// BUILDERS CALLED BY WRITER AND PARSERS CALLED BY READER

int buildDataPacket(unsigned char \*packet, int sequenceNumber, unsigned char \*data, int dataLength)

{

int shift = 4;

packet[0] = CTRL\_DATA;

packet[1] = (unsigned char)sequenceNumber;

unsigned char l1, l2;

util\_get\_MSB(dataLength, &l2);

util\_get\_LSB(dataLength, &l1);

packet[2] = (unsigned char)l2;

packet[3] = (unsigned char)l1;

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

packet[i + shift] = data[i];

return dataLength + shift; // returns len of buffer

}

int parseDataPacket(unsigned char \*packet, unsigned char \*data, int \*sequenceNumber)

{

if (packet[0] != CTRL\_DATA)

{

return 1;

}

\*sequenceNumber = (int)packet[1];

int size;

int shift = 4;

util\_join\_bytes(&size, packet[2], packet[3]);

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

{

data[i] = packet[i + shift];

}

return 0;

}

int buildControlPacket(unsigned char \*packet, unsigned char control, int fileSize, const char \*fileName)

{

packet[0] = control;

packet[1] = TYPE\_FILE\_SIZE;

int byteCount;

get\_size\_in\_bytes(fileSize, &byteCount);

packet[2] = byteCount;

int fileSizeStart = 3;

for (int i = byteCount - 1; i > 0; i--)

{

packet[fileSizeStart + i] = (unsigned char)(fileSize);

fileSize = fileSize >> 8;

}

packet[3 + byteCount] = TYPE\_FILE\_NAME;

int filenameSize = strlen(fileName) + 1;

packet[4 + byteCount] = (unsigned char)filenameSize;

int fileNameStart = 5 + byteCount;

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

{

packet[fileNameStart + i] = fileName[i];

}

return 5 + byteCount + filenameSize + 1; // returns len of buffer

}

int parseControlPacket(unsigned char \*packet, int \*fileSize, unsigned char \*fileName)

{

if (packet[0] != CTRL\_START && packet[0] != CTRL\_END)

{

return 1;

}

int lengthFileSize, lengthFileName, fileNameStart;

\*fileSize = 0;

if (packet[1] == TYPE\_FILE\_SIZE)

{

lengthFileSize = (int)packet[2];

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

{

\*fileSize = \*fileSize | (unsigned char)packet[3 + i];

if (i != lengthFileSize - 1)

{

\*fileSize = \*fileSize << 8;

}

}

}

else

{

return 1;

}

fileNameStart = 5 + lengthFileSize;

if (packet[fileNameStart - 2] == TYPE\_FILE\_NAME)

{

lengthFileName = (int)packet[4 + lengthFileSize];

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

{

fileName[i] = packet[fileNameStart + i];

}

}

else

{

return 1;

}

return 0;

}

int sendFile(const char \*filename, char \*serialPort)

{

FILE \*file = openFile(filename, "r");

if (file == NULL)

{

printf("ERROR OPENING FILE!\n");

return 1;

}

if (llopen(linklayer) == -1)

{

return 1;

}

int fileSize = getFileSize(file);

unsigned char cSPacket[MAX\_PACK\_SIZE];

int packetSize = buildControlPacket(cSPacket, START\_TRANSFER, fileSize, filename);

if (llwrite(cSPacket, packetSize) == -1)

{

if (closeFile(file))

{

return 1;

}

return 1;

}

unsigned char data[MAX\_DATA\_SIZE];

int bytesRead;

int n = 0;

unsigned char dPacket[MAX\_PACK\_SIZE];

while (!feof(file))

{

bytesRead = readBytesFromFile(file, data);

if (bytesRead != MAX\_DATA\_SIZE && !feof(file))

{

printf("ERROR READING\n");

if (closeFile(file))

{

return 1;

}

return 1;

}

packetSize = buildDataPacket(dPacket, getSequenceNumber(n), data, bytesRead);

n++;

if (llwrite(dPacket, packetSize) == -1)

{

if (closeFile(file))

{

return 1;

}

return 1;

}

}

unsigned char cEPacket[MAX\_PACK\_SIZE];

packetSize = buildControlPacket(cEPacket, END\_TRANSFER, fileSize, filename);

if (llwrite(cEPacket, packetSize) == -1)

{

return 1;

}

if (llclose(1) == -1)

{

return 1;

}

if (closeFile(file))

{

return 1;

}

return 0;

}

int receiveFile(char \*filename, char \*serialPort){

unsigned char cPacket[MAX\_PACK\_SIZE];

unsigned char packetFilename[255];

int fileSize = 0;

int packetSize;

if(llopen(linklayer) == -1){

return 1;

}

if((packetSize = llread(cPacket)) < 0){

printf("Error reading control packet");

return 1;

}

if(parseControlPacket(cPacket, &fileSize, packetFilename) != 0 || cPacket[0] != START\_TRANSFER){

printf("Error parsing control packet\n");

return 1;

}

FILE \*file = openFile(filename, "w");

if(file == NULL){

return 1;

}

int n = 0;

int sequenceNumber;

unsigned char dPacket[MAX\_PACK\_SIZE];

unsigned char data[MAX\_DATA\_SIZE];

do

{

packetSize = llread(dPacket);

if(packetSize < 0){

closeFile(file);

return 1;

}

if(dPacket[0] == CTRL\_DATA){

if(parseDataPacket(dPacket, data, &sequenceNumber)){

closeFile(file);

return 1;

}

if(sequenceNumber != getSequenceNumber(n)){

printf("Different sequence numbers\n");

closeFile(file);

return 1;

}

n++;

}

if(dPacket[0] != CTRL\_END){

if((writeBytesToFile(file, data, packetSize - 4) != packetSize - 4)){

closeFile(file);

return 1;

}

}

} while (dPacket[0] != CTRL\_END);

closeFile(file);

int newFileSize = 0;

unsigned char newFileName[255];

if((parseControlPacket(dPacket, &newFileSize, newFileName) != 0) || dPacket[0] != END\_TRANSFER)

{

printf("Error parsing control packet\n");

return 1;

}

if((fileSize != newFileSize) || (strcmp(newFileName, packetFilename) != 0)){

printf("Files aren't the same\n");

return 1;

}

if(llclose(1) == -1){

printf("Error closing file\n");

return 1;

}

return 0;

}

void applicationLayer(const char \*serialPort, const char \*role, int baudRate,

int nTries, int timeout, const char \*filename)

{

strcpy(linklayer.serialPort, serialPort);

if(strcmp(role, "tx") == 0)

linklayer.role = LlTx;

else if(strcmp(role, "rx") == 0)

linklayer.role = LlRx;

else{

perror("Role not defined\n");

}

linklayer.baudRate = baudRate;

linklayer.nRetransmissions = nTries;

linklayer.timeout = timeout;

if (linklayer.role == LlTx)

{

if((sendFile(filename, serialPort) != 0)){

printf("Error sending file\n");

}

}

else if(linklayer.role == LlRx)

{

if((receiveFile(filename, serialPort) != 0)){

printf("Error receiving file\n");

}

}

else{

perror("Role not defined");

}

}

* 1. –*Ficheiro link\_layer.c.*

// Link layer protocol implementation

#include "unistd.h"

#include <sys/times.h>

#include <stdio.h>

#include "link\_layer.h"

#include "frame.h"

#include "port\_operations.h"

// MISC

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

int fd;

char \*serialPort;

LinkLayerRole role;

int baudRate;

int nRetransmissions;

int timeout;

unsigned char frame[MAX\_SIZE\_FRAME];

unsigned int frameLength;

int seqNumber;

extern int currentRetransmission, relay, stop;

int bitsReceived = 0;

struct timeval start, end;

////////////////////////////////////////////////

// LLOPEN

////////////////////////////////////////////////

int llopen(LinkLayer connectionParameters)

{

serialPort = connectionParameters.serialPort;

role = connectionParameters.role;

baudRate = connectionParameters.baudRate;

nRetransmissions = connectionParameters.nRetransmissions;

timeout = connectionParameters.timeout;

if((fd = set\_up\_port(serialPort)) == -1){

printf("Couldn't open port communication");

return -1;

}

alarmHandlerInstaller();

frameLength = BUF\_SIZE\_SF;

if(connectionParameters.role == LlTx){

unsigned char responseBuffer[BUF\_SIZE\_SF];

if (createSFrame(frame, FIELD\_A\_T\_INIT, SET) != 0){

close\_port(fd);

return -1;

}

if (sendFrame(frame, fd, frameLength) == -1){

close\_port(fd);

return -1;

}

printf("SET sent\n");

stop = FALSE;

int bytesRead = -1;

currentRetransmission = 0;

relay = FALSE;

alarm(timeout);

unsigned char controlByte[1] = {UA};

while(!stop){

bytesRead = readSFrame(responseBuffer, fd, controlByte, 1, FIELD\_A\_T\_INIT);

if(relay){

sendFrame(frame, fd, frameLength);

relay = FALSE;

}

if(bytesRead >= 0){

alarm(0);

stop = TRUE;

}

}

if(bytesRead == -1){

printf("Couldn't read UA. Closing file\n");

close\_port(fd);

return -1;

}

printf("UA received\n");

return 1;

}

if(connectionParameters.role == LlRx){

unsigned char controlByte[1] = {SET};

if (readSFrame(frame, fd, controlByte, 1, FIELD\_A\_T\_INIT) != 0){

close\_port(fd);

return -1;

}

printf("SET received\n");

if(createSFrame(frame, FIELD\_A\_T\_INIT, UA) != 0){

close\_port(fd);

return -1;

}

if(sendFrame(frame, fd, frameLength) < 0){

close\_port(fd);

return -1;

}

printf("UA sent\n");

gettimeofday(&start, NULL);

return 1;

}

perror("Invalid role");

close\_port(fd);

return -1;

}

////////////////////////////////////////////////

// LLWRITE

////////////////////////////////////////////////

int llwrite(const unsigned char \*buf, int bufSize)

{

unsigned char responseBuf[BUF\_SIZE\_SF];

unsigned char controlByte;

if(seqNumber == 0){

controlByte = NS0;

}

else{

controlByte = NS1;

}

if(createIFrame(frame, controlByte, buf, bufSize) != 0){

close\_port(fd);

return -1;

}

int length;

int frameSize = bufSize + 6;

if((length = stuffIFrame(frame, frameSize)) < 0){

close\_port(fd);

return -1;

}

frameLength = length;

int bytesWritten;

int finished = FALSE;

while(!finished){

if((bytesWritten = sendFrame(frame, fd, frameLength)) == -1){

close\_port(fd);

return -1;

}

printf("Information sent\n");

int bytesRead = -1;

stop = FALSE;

currentRetransmission = 0;

relay = FALSE;

alarm(timeout);

unsigned char controlBytes[2];

if(controlByte == 0){

controlBytes[0] = RR1;

controlBytes[1] = REJ0;

}

else{

controlBytes[0] = RR0;

controlBytes[1] = REJ1;

}

while(!stop){

bytesRead = readSFrame(responseBuf, fd, controlBytes, 2, FIELD\_A\_T\_INIT);

if(relay){

if(sendFrame(frame, fd, frameLength) < 0){

printf("Troubles resending. Closing file\n");

close\_port(fd);

return -1;

}

relay = FALSE;

}

if (bytesRead >= 0){

alarm(0);

stop = TRUE;

}

}

if (bytesRead < 0){

printf("Troubles reading. Closing file\n");

close\_port(fd);

return -1;

}

else if(bytesRead == 0){

finished = TRUE;

}

else{

finished = FALSE;

}

printf("Response read\n");

}

if(seqNumber == 0){

seqNumber = 1;

}

else{

seqNumber = 0;

}

return bytesWritten - 6;

}

////////////////////////////////////////////////

// LLREAD

////////////////////////////////////////////////

int llread(unsigned char \*packet)

{

unsigned char controlBytes[2];

controlBytes[0] = NS0;

controlBytes[1] = NS1;

int packetSize, bytesRead, bufferFull = FALSE;

while(!bufferFull){

bytesRead = readIFrame(frame, fd, controlBytes, 2, FIELD\_A\_T\_INIT);

if((packetSize = unstuffIFrame(frame, bytesRead)) < 0){

printf("Troubles unstuffing. Closing file\n");

close\_port(fd);

return -1;

}

int controlByte;

if(frame[2] == NS0){

controlByte = 0;

}

else if(frame[2] == NS1){

controlByte = 1;

}

else{

printf("Control byte wrongly defined. Closing file\n");

close\_port(fd);

return -1;

}

int responseByte;

if (frame[packetSize - 2] == dataBCC(&frame[4], packetSize - 6)){

if(controlByte != seqNumber){

if(controlByte == 0){

seqNumber = 1;

responseByte = RR1;

}

else{

seqNumber = 0;

responseByte = RR0;

}

}

else{

for (int i = 0; i < packetSize - 6; i++){

packet[i] = frame[4 + i];

}

bufferFull = TRUE;

if(controlByte == 0){

seqNumber = 1;

responseByte = RR1;

}

else{

seqNumber = 0;

responseByte = RR0;

}

}

}

else{

if(controlByte != seqNumber){

if(controlByte == 0){

seqNumber = 1;

responseByte = RR1;

}

else{

seqNumber = 0;

responseByte = RR0;

}

}

else{

if(controlByte == 0){

seqNumber = 0;

responseByte = REJ0;

}

else{

seqNumber = 1;

responseByte = REJ1;

}

}

}

if((createSFrame(frame, FIELD\_A\_T\_INIT, responseByte)) != 0){

printf("Troubles sending response. Closing file\n");

close\_port(fd);

return -1;

}

frameLength = BUF\_SIZE\_SF;

if(sendFrame(frame, fd, frameLength) < 0){

printf("Troubles sending response. Closing file\n");

close\_port(fd);

return -1;

}

printf("Response sent\n");

}

bitsReceived += (packetSize - 6) \* 8;

return packetSize - 6;

}

////////////////////////////////////////////////

// LLCLOSE

////////////////////////////////////////////////

int llclose(int showStatistics)

{

if(role == LlTx){

if((createSFrame(frame, FIELD\_A\_T\_INIT, DISC)) != 0){

printf("Troubles sending command. Closing file\n");

close\_port(fd);

return -1;

}

frameLength = BUF\_SIZE\_SF;

if(sendFrame(frame, fd, frameLength) < 0){

printf("Troubles sending command. Closing file\n");

close\_port(fd);

return -1;

}

printf("DISC sent\n");

stop = FALSE;

int bytesRead = -1;

currentRetransmission = 0;

relay = FALSE;

alarm(timeout);

unsigned char controlByte[1] = {DISC};

while(!stop){

if((bytesRead = readSFrame(frame, fd, controlByte, 1, FIELD\_A\_R\_INIT)) < 0){

printf("Error reading DISC\n");

return -1;

}

if(relay){

sendFrame(frame, fd, frameLength);

relay = FALSE;

}

if(bytesRead >= 0){

alarm(0);

stop = TRUE;

}

}

if(bytesRead < 0){

printf("Troubles reading command. Closing file\n");

close\_port(fd);

return -1;

}

printf("DISC received\n");

if((createSFrame(frame, FIELD\_A\_R\_INIT, UA)) != 0){

printf("Troubles sending response. Closing file\n");

close\_port(fd);

return -1;

}

if(sendFrame(frame, fd, frameLength) < 0){

printf("Troubles sending response. Closing file\n");

close\_port(fd);

return -1;

}

printf("UA sent\n");

}

else if(role == LlRx){

unsigned char controlByteCommand[1] = {DISC};

if (readSFrame(frame, fd, controlByteCommand, 1, FIELD\_A\_T\_INIT) != 0){

printf("Troubles receiving command. Closing file\n");

close\_port(fd);

return -1;

}

printf("DISC received\n");

if(createSFrame(frame, FIELD\_A\_R\_INIT, DISC) != 0){

printf("Troubles sending command. Closing file\n");

close\_port(fd);

return -1;

}

if(sendFrame(frame, fd, frameLength) < 0){

printf("Troubles sending command. Closing file\n");

close\_port(fd);

return -1;

}

printf("DISC sent\n");

stop = FALSE;

int bytesRead = -1;

currentRetransmission = 0;

relay = FALSE;

frameLength = BUF\_SIZE\_SF;

alarm(timeout);

unsigned char controlByteResponse[1] = {UA};

while(!stop){

if((bytesRead = readSFrame(frame, fd, controlByteResponse, 1, FIELD\_A\_R\_INIT)) < 0){

printf("Error reading UA\n");

return -1;

}

if(relay){

sendFrame(frame, fd, frameLength);

relay = FALSE;

}

if(bytesRead >= 0){

alarm(0);

stop = TRUE;

}

}

if(bytesRead < 0){

printf("Troubles reading response. Closing file\n");

close\_port(fd);

return -1;

}

printf("UA received\n");

gettimeofday(&end, NULL);

double time\_spent = ((end.tv\_sec - start.tv\_sec) \* 1e6 + (end.tv\_usec - start.tv\_usec)) \* 1e-6;

if(showStatistics == TRUE){

printf("Bits Received = %d\n", bitsReceived);

printf("Time spent = %f\n", time\_spent);

double R = bitsReceived/time\_spent;

double S = R / baudRate;

printf("Debit received = %lf\n", R);

printf("Efficiency S = %lf\n", S);

}

}

else{

perror("Invalid role");

close\_port(fd);

return -1;

}

if(close\_port(fd) < 0){

return -1;

}

return 1;

}

* 1. *–**Ficheiro alarm.c.*

#include "alarm.h"

int currentRetransmission, relay = FALSE, stop = FALSE;

extern int nRetransmissions, timeout;

void alarmHandler(int signal){

if(nRetransmissions > currentRetransmission){

relay = TRUE;

alarm(timeout);

currentRetransmission++;

printf("Timeout or invalide value: resending...\n");

}

else{

printf("Timeout or invalide value: number of tries exceeded!\n");

stop = TRUE;

}

}

* 1. *– Ficheiro auxiliar.c.*

#include "auxiliar.h"

int util\_get\_LSB (int val, unsigned char \*lsb) {

\*lsb = (unsigned char) val;

return 0;

}

int util\_get\_MSB (int val, unsigned char \*msb) {

\*msb = (unsigned char) (val >> 8);

return 0;

}

int util\_join\_bytes (int \*ret, unsigned char msb, unsigned char lsb) {

\*ret = msb;

\*ret = \*ret << 8;

\*ret = \*ret | lsb;

return 0;

}

int get\_size\_in\_bytes (int fileSize, int \*byteCount){

\*byteCount = fileSize / BYTE\_SIZE;

if(fileSize % BYTE\_SIZE > 0){

byteCount++;

}

return 1;

}

unsigned char headerBCC(unsigned char address, unsigned char control){

return address ^ control;

}

unsigned char dataBCC(const unsigned char \*data, int dataSize){

unsigned char dataInit = data[0];

for(int i = 1; i < dataSize; i++){

dataInit = dataInit ^ data[i];

}

return dataInit;

}

int stuffIFrame (unsigned char \*frame, int frameSize){

unsigned char cpy[frameSize];

int flagByteNo = frameSize - 1;

int shift = 0;

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

cpy[i] = frame[i];

}

for(int i = 4; i < frameSize + shift; i++){ // only stuff data field of I FRAME

if(cpy[i] == FLAG && i != flagByteNo){

frame[i + shift] = FLAG\_STUF1;

frame[i+1 + shift] = FLAG\_STUF2;

shift++;

}

else if(cpy[i] == ESC\_BYTE){

frame[i + shift] = ESC\_BYTE\_STUF1;

frame[i + 1 + shift] = ESC\_BYTE\_STUF2;

shift++;

}

else{

frame[i + shift] = cpy[i];

}

}

return frameSize + shift; // return new len

}

int unstuffIFrame (unsigned char \*frame, int frameSize){

unsigned char cpy[frameSize];

int flagByteNo = frameSize - 2;

int shift = 0;

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

cpy[i] = frame[i];

}

for(int i = 4; i < frameSize; i++){ // only stuff data field of I FRAME

if(cpy[i] == FLAG\_STUF1 && cpy[i+1] == FLAG\_STUF2 && i != flagByteNo){

frame[i + shift] = FLAG;

shift--;

i++;

}

else if(cpy[i] == ESC\_BYTE\_STUF1 && cpy[i+1] == ESC\_BYTE\_STUF2){

frame[i + shift] = ESC\_BYTE;

shift--;

i++;

}

else{

frame[i + shift] = cpy[i];

}

}

return frameSize + shift; // return new len

}

int readByte(unsigned char\* byte, int fd) {

if(read(fd, byte, sizeof(unsigned char)) <= 0)

return -1;

return 0;

}

void alarmHandlerInstaller() {

struct sigaction action;

action.sa\_handler = alarmHandler;

if(sigemptyset(&action.sa\_mask) == -1){

perror("sigemptyset");

exit(-1);

}

action.sa\_flags = 0;

if(sigaction(SIGALRM, &action, NULL) != 0){

perror("sigaction");

exit(-1);

}

}

int getSequenceNumber (int number){

return number % 255;

}

* 1. –*Ficheiro file.c.*

#include "file.h"

FILE\* openFile(const char \*name, char\* mode){

FILE \*ret;

ret = fopen(name, mode);

if(ret == NULL){

printf("ERROR OPENING FILE\n");

}

return ret;

}

int closeFile(FILE \*file){

return fclose(file);

}

int getFileSize(FILE \*file){

fseek(file, 0L, SEEK\_END);

int fileSize = (int)ftell(file);

rewind(file);

return fileSize;

}

int readBytesFromFile(FILE \*file, unsigned char\* dest){

return fread(dest, sizeof(unsigned char), MAX\_DATA\_SIZE, file);

}

int writeBytesToFile(FILE \*file, unsigned char\* source, int len){

return fwrite(source, sizeof(unsigned char), len, file);

}

* 1. *– Ficheiro frame.c.*

#include "frame.h"

extern int currentRetransmission, relay, stop;

int sendFrame(unsigned char\* frame, int fd, int length) {

int bytesWritten;

if( (bytesWritten = write(fd, frame, length)) <= 0){

return -1;

}

sleep(1);

return bytesWritten;

}

int createIFrame(unsigned char\* frame, unsigned char control, const unsigned char\* data, int dataSize) {

frame[0] = FLAG;

frame[1] = FIELD\_A\_T\_INIT;

frame[2] = control;

frame[3] = headerBCC(FIELD\_A\_T\_INIT, control);

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

frame[i + 4] = data[i];

}

frame[dataSize + 4] = dataBCC(data, dataSize);

frame[dataSize + 5] = FLAG;

return 0;

}

int createSFrame(unsigned char\* frame, unsigned char address, unsigned char control) {

frame[0] = FLAG;

frame[1] = address;

frame[2] = control;

frame[3] = headerBCC(address, control);

frame[4] = FLAG;

return 0;

}

int readSFrame(unsigned char\* frame, int fd, unsigned char\* controlBytes, int controlBytesLength, unsigned char addressByte) {

state\_machine\_st \*sm = create\_state\_machine(controlBytes, controlBytesLength, addressByte);

unsigned char byte;

while(sm->state != STOP && stop != TRUE && !relay) {

if(readByte(&byte, fd) == 0){

event\_handler(sm, byte, frame, SUPERVISION);

}

}

int ret = sm->controlByteIndex;

destroy\_st(sm);

if(stop == TRUE || relay)

return -1;

return ret;

}

int readIFrame(unsigned char\* frame, int fd, unsigned char\* controlBytes, int controlBytesLength, unsigned char addressByte) {

state\_machine\_st \*sm = create\_state\_machine(controlBytes, controlBytesLength, addressByte);

unsigned char byte;

while(sm->state != STOP) {

if(readByte(&byte, fd) == 0)

event\_handler(sm, byte, frame, INFORMATION);

}

int ret = sm->dataLength;

destroy\_st(sm);

return ret + 5;

}

* 1. –*Ficheiro port\_operations.c.*

1. #include "port\_operations.h"
2. struct termios oldtio;
3. struct termios newtio;
4. int set\_up\_port(char \*serialPortName){
5. // Open serial port device for reading and writing, and not as controlling tty
6. // because we don't want to get killed if linenoise sends CTRL-C.
7. int fd = open(serialPortName, O\_RDWR | O\_NOCTTY);
8. if (fd < 0)
9. {
10. perror(serialPortName);
11. return(-1);
12. }
13. // Save current port settings
14. if (tcgetattr(fd, &oldtio) == -1)
15. {
16. perror("tcgetattr");
17. return(-1);
18. }
19. // Clear struct for new port settings
20. memset(&newtio, 0, sizeof(newtio));
21. newtio.c\_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
22. newtio.c\_iflag = IGNPAR;
23. newtio.c\_oflag = 0;
24. // Set input mode (non-canonical, no echo,...)
25. newtio.c\_lflag = 0;
26. newtio.c\_cc[VTIME] = 0; // Inter-character timer unused
27. newtio.c\_cc[VMIN] = 5; // Blocking read until 5 chars received
28. // VTIME e VMIN should be changed in order to protect with a
29. // timeout the reception of the following character(s)
30. // Now clean the line and activate the settings for the port
31. // tcflush() discards data written to the object referred to
32. // by fd but not transmitted, or data received but not read,
33. // depending on the value of queue\_selector:
34. // TCIFLUSH - flushes data received but not read.
35. tcflush(fd, TCIOFLUSH);
36. // Set new port settings
37. if (tcsetattr(fd, TCSANOW, &newtio) == -1)
38. {
39. perror("tcsetattr");
40. return(-1);
41. }
42. return fd;
43. }
44. int close\_port(int fd){
45. sleep(1);
46. if (tcsetattr(fd, TCSANOW, &oldtio) == -1){
47. perror("tcsetattr");
48. return(-1);
49. }
50. close(fd);
51. return 0;
52. }

*1.10 – Ficheiro state\_machine.c.*

#include "state\_machine.h"

int isControlByte(unsigned char byte, state\_machine\_st\* sm) {

for (int i = 0; i < sm->controlBytesLength; i++) {

if (sm->controlBytes[i] == byte)

return i;

}

return -1;

}

void change\_state(state\_machine\_st\* sm, state\_st st) {

sm->state = st;

}

state\_machine\_st\* create\_state\_machine(unsigned char\* controlBytes, int controlBytesLength, unsigned char addressByte) {

state\_machine\_st\* sm = malloc(sizeof(state\_machine\_st));

change\_state(sm, START);

sm->controlBytes = controlBytes;

sm->controlBytesLength = controlBytesLength;

sm->addressByte = addressByte;

sm->dataLength = 0;

return sm;

}

void event\_handler(state\_machine\_st\* sm, unsigned char byte, unsigned char\* frame, int mode) {

static int i = 0;

if(mode == SUPERVISION){

switch(sm->state) {

case START:

if (byte == FLAG) {

change\_state(sm, FLAG\_RCV);

frame[0] = byte;

}

break;

case FLAG\_RCV:

if (byte == FLAG)

break;

else if (byte == sm->addressByte) {

change\_state(sm, A\_RCV);

frame[1] = byte;

}

else

change\_state(sm, START);

break;

case A\_RCV:

if (byte == FLAG){

change\_state(sm, FLAG\_RCV);

}

else {

int n;

if ((n = isControlByte(byte, sm))>=0){

change\_state(sm, C\_RCV);

sm->controlByteIndex = n;

frame[2] = byte;

}

else

change\_state(sm, START);

}

break;

case C\_RCV:

if (byte == headerBCC(frame[1], frame[2])){

change\_state(sm, BCC\_OK);

frame[3] = byte;

}

else if (byte == FLAG)

change\_state(sm, FLAG\_RCV);

else

change\_state(sm, START);

break;

case BCC\_OK:

if (byte == FLAG){

change\_state(sm, STOP);

frame[4] = byte;

}

else

change\_state(sm, START);

break;

default:

break;

}

}

else if(mode == INFORMATION){

switch(sm->state) {

case START:

i = 0;

if (byte == FLAG) {

change\_state(sm, FLAG\_RCV);

frame[i++] = byte;

}

break;

case FLAG\_RCV:

if (byte == FLAG)

break;

else if (byte == sm->addressByte) {

change\_state(sm, A\_RCV);

frame[i++] = byte;

}

else {

change\_state(sm, START);

i = (int) START;

}

break;

case A\_RCV:

if (byte == FLAG) {

change\_state(sm, FLAG\_RCV);

i = (int) FLAG\_RCV;

}

else {

if (isControlByte(byte, sm) >= 0){

change\_state(sm, C\_RCV);

frame[i++] = byte;

}

else {

change\_state(sm, START);

i = (int) START;

}

}

break;

case C\_RCV:

if (byte == headerBCC(frame[1], frame[2])){

change\_state(sm, BCC\_OK);

frame[i++] = byte;

}

else if (byte == FLAG){

change\_state(sm, FLAG\_RCV);

i = (int) FLAG\_RCV;

}

else{

change\_state(sm, START);

i = (int) START;

}

break;

case BCC\_OK:

if(byte == FLAG){

frame[i] = byte;

change\_state(sm, STOP);

sm->dataLength = i-4;

}

else{

frame[i++] = byte;

}

break;

default:

break;

}

}

}

void destroy\_st(state\_machine\_st\* sm) {

free(sm);

}

ANEXO 2 – Estrutura do código

*2.1 – Estrutura da ligação de dados.*

typedef enum

{

LlTx,

LlRx,

} LinkLayerRole;

typedef struct

{

char serialPort[50];

LinkLayerRole role;

int baudRate;

int nRetransmissions;

int timeout;

} LinkLayer;

*2.2 – Principais funções da camada da ligação de dados.*

int llopen(LinkLayer connectionParameters);

int llwrite(const unsigned char \*buf, int bufSize);

int llread(unsigned char \*packet);

int llclose(int showStatistics);

int set\_up\_port(char \*serialPortName);

int close\_port(int fd);

*2.3 – Estrutura da camada da aplicação.*

struct applicationLayer {

int fileDescriptor;

int status;

};

*2.4 – Principais funções da camada da aplicação.*

int buildDataPacket(unsigned char \*packet, int sequenceNumber, unsigned char \*data,

int dataLength);

int parseDataPacket(unsigned char \*packet, unsigned char \*data, int \*sequenceNumber);

int buildControlPacket(unsigned char \*packet, unsigned char control, int fileSize, const char \*fileName);

int parseControlPacket(unsigned char \*packet, int \*fileSize, unsigned char \*fileName);

int sendFile(const char \*filename, char \*serialPort);

int receiveFile(char \*filename, char \*serialPort);

void applicationLayer(const char \*serialPort, const char \*role, int baudRate,

int nTries, int timeout, const char \*filename);

ANEXO 3 – Protocolo de ligação lógica

*3.1 – Operações iniciais da função llopen().*

serialPort = connectionParameters.serialPort;

    role = connectionParameters.role;

    baudRate = connectionParameters.baudRate;

    nRetransmissions = connectionParameters.nRetransmissions;

    timeout = connectionParameters.timeout;

    if((fd = set\_up\_port(serialPort)) == -1){

        printf("Couldn't open port communication");

        return -1;

    }

    alarmHandlerInstaller();

*3.2 – Função llopen() – emissor.*

if(connectionParameters.role == LlTx){

        unsigned char responseBuffer[BUF\_SIZE\_SF];

        if (createSFrame(frame, FIELD\_A\_T\_INIT, SET) != 0){

            close\_port(fd);

            return -1;

        }

        if (sendFrame(frame, fd, frameLength) == -1){

            close\_port(fd);

            return -1;

        }

        printf("SET sent\n");

        stop = FALSE;

        int bytesRead = -1;

        currentRetransmission = 0;

        relay = FALSE;

        alarm(timeout);

        unsigned char controlByte[1] = {UA};

        while(!stop){

            bytesRead = readSFrame(responseBuffer, fd, controlByte, 1, FIELD\_A\_T\_INIT);

            if(relay){

                sendFrame(frame, fd, frameLength);

                relay = FALSE;

            }

            if(bytesRead >= 0){

                alarm(0);

                stop = TRUE;

            }

        }

        if(bytesRead == -1){

            printf("Couldn't read UA. Closing file\n");

            close\_port(fd);

            return -1;

        }

        printf("UA received\n");

        return 1;

    }

*3.3 – Função llopen() – recetor*

if(connectionParameters.role == LlRx){

        unsigned char controlByte[1] = {SET};

        if (readSFrame(frame, fd, controlByte, 1, FIELD\_A\_T\_INIT) != 0){

            close\_port(fd);

            return -1;

        }

        printf("SET received\n");

        if(createSFrame(frame, FIELD\_A\_T\_INIT, UA) != 0){

            close\_port(fd);

            return -1;

        }

        if(sendFrame(frame, fd, frameLength) < 0){

            close\_port(fd);

            return -1;

        }

        printf("UA sent\n");

        gettimeofday(&start, NULL);

        return 1;

    }

*3.4 – Função llwrite() - criação de uma trama de informação.*

if(createIFrame(frame, controlByte, buf, bufSize) != 0){

        close\_port(fd);

        return -1;

    }

*3.5 – Função llwrite() – stuffing da trama de informação.*

if((length = stuffIFrame(frame, frameSize)) < 0){

        close\_port(fd);

        return -1;

    }

*3.6 – Função llwrite() – envio da trama de informação.*

 if((bytesWritten = sendFrame(frame, fd, frameLength)) == -1){

            close\_port(fd);

            return -1;

    }

*3.7 – Função llwrite() - receção de uma resposta.*

bytesRead = readSFrame(responseBuf, fd, controlBytes, 2, FIELD\_A\_T\_INIT);

*3.8 – Função llwrite() – processamento da resposta recebida.*

while(!stop){

            bytesRead = readSFrame(responseBuf, fd, controlBytes, 2, FIELD\_A\_T\_INIT);

            if(relay){

                if(sendFrame(frame, fd, frameLength) < 0){

                    printf("Troubles resending. Closing file\n");

                    close\_port(fd);

                    return -1;

                }

                relay = FALSE;

            }

            if (bytesRead >= 0){

                alarm(0);

                stop = TRUE;

            }

        }

*3.9 – Função llread() -* leitura de uma trama de informação.

bytesRead = readIFrame(frame, fd, controlBytes, 2, FIELD\_A\_T\_INIT);

*3.10 – Função llread() - unstuffing uma trama de informação.*

if((packetSize = unstuffIFrame(frame, bytesRead)) < 0){

            printf("Troubles unstuffing. Closing file\n");

            close\_port(fd);

            return -1;

        }

*3.11 – Função llread() – construção de uma resposta de acordo com o BBC2 e com o sequence number.*

if (frame[packetSize - 2] == dataBCC(&frame[4], packetSize - 6)){

            if(controlByte != seqNumber){

                if(controlByte == 0){

                    seqNumber = 1;

                    responseByte = RR1;

                }

                else{

                    seqNumber = 0;

                    responseByte = RR0;

                }

            }

            else{

                for (int i = 0; i < packetSize - 6; i++){

                    packet[i] = frame[4 + i];

                }

                bufferFull = TRUE;

                if(controlByte == 0){

                    seqNumber = 1;

                    responseByte = RR1;

                }

                else{

                    seqNumber = 0;

                    responseByte = RR0;

                }

            }

        }

        else{

            if(controlByte != seqNumber){

                if(controlByte == 0){

                    seqNumber = 1;

                    responseByte = RR1;

                }

                else{

                    seqNumber = 0;

                    responseByte = RR0;

                }

            }

            else{

                if(controlByte == 0){

                    seqNumber = 0;

                    responseByte = REJ0;

                }

                else{

                    seqNumber = 1;

                    responseByte = REJ1;

                }

            }

        }

*3.12 – Função llread() –* criação de uma trama de supervisão em função da resposta.

if((createSFrame(frame, FIELD\_A\_T\_INIT, responseByte)) != 0){

            printf("Troubles sending response. Closing file\n");

            close\_port(fd);

            return -1;        }

*3.13 – Função llread() –* envio da trama de supervisão.

if(sendFrame(frame, fd, frameLength) < 0){

printf("Troubles sending response. Closing file\n");

close\_port(fd);

return -1;

}

*3.14 – Função llclose() – emissor.*

if(role == LlTx){

        if((createSFrame(frame, FIELD\_A\_T\_INIT, DISC)) != 0){

            printf("Troubles sending command. Closing file\n");

            close\_port(fd);

            return -1;

        }

        frameLength = BUF\_SIZE\_SF;

        if(sendFrame(frame, fd, frameLength) < 0){

            printf("Troubles sending command. Closing file\n");

            close\_port(fd);

            return -1;

        }

        printf("DISC sent\n");

        stop = FALSE;

        int bytesRead = -1;

        currentRetransmission = 0;

        relay = FALSE;

        alarm(timeout);

        unsigned char controlByte[1] = {DISC};

        while(!stop){

            if((bytesRead = readSFrame(frame, fd, controlByte, 1, FIELD\_A\_R\_INIT)) < 0){

                printf("Error reading DISC\n");

                return -1;

            }

            if(relay){

                sendFrame(frame, fd, frameLength);

                relay = FALSE;

            }

            if(bytesRead >= 0){

                alarm(0);

                stop = TRUE;

            }

        }

        if(bytesRead < 0){

            printf("Troubles reading command. Closing file\n");

            close\_port(fd);

            return -1;

        }

        printf("DISC received\n");

        if((createSFrame(frame, FIELD\_A\_R\_INIT, UA)) != 0){

            printf("Troubles sending response. Closing file\n");

            close\_port(fd);

            return -1;

        }

        if(sendFrame(frame, fd, frameLength) < 0){

            printf("Troubles sending response. Closing file\n");

            close\_port(fd);

            return -1;

        }

        printf("UA sent\n");

    }

*3.15 – Função llclose() – recetor.*

else if(role == LlRx){

        unsigned char controlByteCommand[1] = {DISC};

        if (readSFrame(frame, fd, controlByteCommand, 1, FIELD\_A\_T\_INIT) != 0){

            printf("Troubles receiving command. Closing file\n");

            close\_port(fd);

            return -1;

        }

        printf("DISC received\n");

        if(createSFrame(frame, FIELD\_A\_R\_INIT, DISC) != 0){

            printf("Troubles sending command. Closing file\n");

            close\_port(fd);

            return -1;

        }

        if(sendFrame(frame, fd, frameLength) < 0){

            printf("Troubles sending command. Closing file\n");

            close\_port(fd);

            return -1;

        }

        printf("DISC sent\n");

        stop = FALSE;

        int bytesRead = -1;

        currentRetransmission = 0;

        relay = FALSE;

        frameLength = BUF\_SIZE\_SF;

        alarm(timeout);

        unsigned char controlByteResponse[1] = {UA};

        while(!stop){

            if((bytesRead = readSFrame(frame, fd, controlByteResponse, 1, FIELD\_A\_R\_INIT)) < 0){

                printf("Error reading UA\n");

                return -1;

            }

            if(relay){

                sendFrame(frame, fd, frameLength);

                relay = FALSE;

            }

            if(bytesRead >= 0){

                alarm(0);

                stop = TRUE;

            }

        }

        if(bytesRead < 0){

            printf("Troubles reading response. Closing file\n");

            close\_port(fd);

            return -1;

        }

        printf("UA received\n");

*3.16 – Função llclose() – estatísticas – recetor.*

gettimeofday(&end, NULL);

        double time\_spent = ((end.tv\_sec - start.tv\_sec) \* 1e6 + (end.tv\_usec - start.tv\_usec)) \* 1e-6;

        if(showStatistics == TRUE){

            printf("Bits Received =  %d\n", bitsReceived);

            printf("Time spent =  %f\n", time\_spent);

            double R =  bitsReceived/time\_spent;

            double S = R / baudRate;

            printf("Debit received = %lf\n", R);

            printf("Efficiency S = %lf\n", S);

        }

    }

ANEXO 4 – Protocolo de aplicação

*4.1 – Função applicationLayer.*

strcpy(linklayer.serialPort, serialPort);

    if(strcmp(role, "tx") == 0)

        linklayer.role = LlTx;

    else if(strcmp(role, "rx") == 0)

        linklayer.role = LlRx;

    else{

        perror("Role not defined\n");

    }

    linklayer.baudRate = baudRate;

    linklayer.nRetransmissions = nTries;

    linklayer.timeout = timeout;

    if (linklayer.role == LlTx)

    {

        if((sendFile(filename, serialPort) != 0)){

            printf("Error sending file\n");

        }

    }

    else if(linklayer.role == LlRx)

    {

        if((receiveFile(filename, serialPort) != 0)){

            printf("Error receiving file\n");

        }

    }

    else{

        perror("Role not defined");

    }

*4.2 – Função sendFile() – abertura do ficheiro.*

FILE \*file = openFile(filename, "r");

    if (file == NULL)

    {

        printf("ERROR OPENING FILE!\n");

        return 1;

    }

*4.3 – Função sendFile() e receiveFile() – estabelecimento ligação.*

if (llopen(linklayer) == -1)

    {

        return 1;

    }

*4.4 – Função sendFile() – construção e envio o control packet inicial.*

int fileSize = getFileSize(file);

    unsigned char cSPacket[MAX\_PACK\_SIZE];

    int packetSize = buildControlPacket(cSPacket, START\_TRANSFER, fileSize, filename);

if (llwrite(cSPacket, packetSize) == -1)

    {

        if (closeFile(file))

        {

            return 1;

        }

        return 1;

    }

*4.5 – Função sendFile() – criação de tramas.*

while (!feof(file))

    {

        bytesRead = readBytesFromFile(file, data);

        if (bytesRead != MAX\_DATA\_SIZE && !feof(file))

        {

            printf("ERROR READING\n");

            if (closeFile(file))

            {

                return 1;

            }

            return 1;

        }

        packetSize = buildDataPacket(dPacket, getSequenceNumber(n), data,bytesRead);

        n++;

        if (llwrite(dPacket, packetSize) == -1)

        {

            if (closeFile(file))

            {

                return 1;

            }

            return 1;

        }

    }

*4.6 – Função sendFile() construção e envio o control packet final.*

unsigned char cEPacket[MAX\_PACK\_SIZE];

    packetSize = buildControlPacket(cEPacket, END\_TRANSFER, fileSize,filename);

    if (llwrite(cEPacket, packetSize) == -1)

    {

        return 1;

    }

*4.7 – Função sendFile() – fecho da ligação.*

if (llclose(1) == -1)

    {

        return 1;

    }

*4.8 – Função sendFile() – fehco do ficheiro.*

if (closeFile(file))

    {

        return 1;

    }

    return 0;

*4.9 – Função receiveFile() – leitura do control packet inicial.*

if((packetSize = llread(cPacket)) < 0){

        printf("Error reading control packet");

        return 1;

    }

*4.10 – Função receiveFile() – parsing do control packet inicial.*

if(parseControlPacket(cPacket, &fileSize, packetFilename) != 0 ||

cPacket[0] != START\_TRANSFER){

        printf("Error parsing control packet\n");

        return 1;

    }

*4.11 – Função receiveFile() – abertura do ficheiro de destino.*

FILE \*file = openFile(filename, "w");

    if(file == NULL){

        return 1;

    }

*4.12 – Função receiveFile() – Leitura e parsing das tramas recebidas até ao control packet final.*

if((packetSize = llread(cPacket)) < 0){

int n = 0;

    int sequenceNumber;

    unsigned char dPacket[MAX\_PACK\_SIZE];

    unsigned char data[MAX\_DATA\_SIZE];

    do

    {

        packetSize = llread(dPacket);

        if(packetSize < 0){

            closeFile(file);

            return 1;

        }

        if(dPacket[0] == CTRL\_DATA){

            if(parseDataPacket(dPacket, data, &sequenceNumber)){

                closeFile(file);

                return 1;

            }

            if(sequenceNumber != getSequenceNumber(n)){

                printf("Different sequence numbers\n");

                closeFile(file);

                return 1;

            }

            n++;

        }

        if(dPacket[0] != CTRL\_END){

            if((writeBytesToFile(file, data, packetSize - 4) != packetSize -4)){

                closeFile(file);

                return 1;

            }

        }

    } while (dPacket[0] != CTRL\_END);

*4.13 – Função receiveFile() – fecho do ficheiro de destino.*

closeFile(file);

*4.14 – Função receiveFile() – parsing do control packet final.*

int newFileSize = 0;

    unsigned char newFileName[255];

    if((parseControlPacket(dPacket, &newFileSize, newFileName) != 0) ||dPacket[0] != END\_TRANSFER)

    {

        printf("Error parsing control packet\n");

        return 1;

    }

*4.15 – Função receiveFile() verificação da igualdade dos ficheiros.*

if((fileSize != newFileSize) || (strcmp(newFileName, packetFilename) != 0)){

        printf("Files aren't the same\n");

        return 1;

    }

*4.16 – Função receiveFile() – fecho da ligação.*

if(llclose(1) == -1){

        printf("Error closing file\n");

        return 1;

    }