U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Protocolo de Ligação de Dados

1º trabalho laboratorial

FEUP
Redes de Computadores

Daniel dos Santos Ferreira - up202108771
Mansur Mustafin - up202102355

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Sumário

Este projeto foi realizado no âmbito da unidade curricular Redes de Computadores para a fundamentação de alguns conceitos lecionados como transferência, uso de serviços e independência entre camadas, e o mecanismo ARQ *Stop & Wait*.

Ao implementar este projeto, tivemos de usar os conceitos descritos acima, sendo uma boa forma de os consolidar.

1. Introdução

Este projeto tem como objetivos implementar uma aplicação capaz de transferir um ficheiro entre duas máquinas distintas utilizando um protocolo de ligação de dados que deverá fornecer serviços à aplicação para que a transferência seja realizada corretamente com correção de eventuais erros a partir de uma porta série RS-232. Deverá também ser possível medir a eficiência do protocolo implementado.

- **Arquitetura**: descrição dos blocos funcionais e interfaces existentes no projeto;
- **Estrutura do código:** apresentação de APIs, principais estruturas de dados e principais funções usadas;
- **Casos de uso principais:** identificação da cadeia de funções usada nos casos de uso principais;
- **Protocolo de ligação lógica:** apresentação do funcionamento do protocolo de ligação de dados e estratégias utilizadas;
- **Protocolo de aplicação:** apresentação do funcionamento do protocolo de aplicação e estratégias utilizadas;
- **Validação:** descrição dos testes efetuados durante a execução do projeto;
- **Eficiência do protocolo de ligação de dados:** análise da eficiência do protocolo de ligação de dados com o uso do mecanismo *Stop & Wait*;
- **Conclusões:** síntese da informação apresentada nas secções anteriores; reflexão sobre os objetivos de aprendizagem alcançados.
- **Anexos I** - código.
- **Anexos II** - gráficos.

2. Arquitetura

A implementação do programa está estruturada em duas camadas distintas: `link_layer` e `application_layer`,com o objetivo de a independência entre nível da ligação de dados e nível da aplicação. Essa separação contribui para uma maior modularidade da arquitetura, permitindo a substituição de uma das camadas por outra do mesmo nível que forneça as funções necessárias sem comprometer o sistema.

A camada de ligação de dados `link_layer` oferece um serviço de comunicação de dados confiável e robusto entre dois computadores conectados por um meio físico. Isso

envolve o uso de uma API de porta série que garante a transmissão de pacotes sem erros e sem redundâncias indesejadas.

Camada de aplicação - `application_layer` é a camada que está mais perto do utilizador. Esta camada utiliza a API fornecida pela `link_layer` para efetuar a transferência de ficheiros, separando-os e enviando-os em frames. Confirma também a transmissão do ficheiro inteiro.

3. Estrutura do código

Foi usada a API da Porta Série

```
#include <fcntl.h>
#include <unistd.h>
#include <termios.h>
```

Application layer

As estruturas de dados principais usados:

```
enum state{                    // Máquina de estados.
      RECV_START,              // início de recepção
      RECV_CONT,               // estado de recepção de dados
      RECV_END                 // fim de recepção
};

typedef struct                 // Os características principais
{                              // do ficheiro.
      size_t file_size;
      char * file_name;
      size_t bytesRead;
} FileProps;
```

Funções principais:

```
// Função que utilizador chama
void applicationLayer(const char *serialPort, const char *role, int
baudRate,int nTries, int timeout, const char *filename);

int readPacketControl(unsigned char * buff);
int sendPacketControl(unsigned char C, const char * filename, size_t
file_size);

unsigned char* readPacketData(unsigned char *buff, size_t *newSize);
int sendPacketData(size_t nBytes, unsigned char *data);
```

Link Layer

As estruturas de dados principais usados:

```
typedef struct                  // Características principais da programma
{
      char serialPort[50];      // nome da porta
      LinkLayerRole role;       // role da programma: enum LlTx or LlRx
      int baudRate;
      int nRetransmissions;     // número máximo de retransmissões
      int timeout;              // tempo da espera entre retransmissão
} LinkLayer;

enum state{ // máquina de estados.
      START, FLAG_RCV, A_RCV, C_RCV, BCC_OK, DATA, STOP
};
```

Funções principais:

```
// Open a connection using the "port" parameters defined in struct
linkLayer.
int llopen(LinkLayer connectionParameters);
// Send data in buf with size bufSize.
int llwrite(const unsigned char *buf, int bufSize);
// Receive data in packet.
int llread(unsigned char *packet);
// Close previously opened connection.
int llclose(int showStatistics);
// envia o command packet com Adress e Control bytes
int send_packet_command( unsigned char A, unsigned char C);
// leia um pacote com técnica de retransmissão
int receivePacketRetransmission(unsigned char A_EXPECTED, unsigned char
C_EXPECTED, unsigned char A_TO_SEND, unsigned char C_TO_SEND);
// Mecanismo de Stuffing e Destuffing
const unsigned char * byteStuffing(const unsigned char *buf, int
bufSize, int *newSize);
int byteDestuffing(unsigned char *buf, int bufSize, int *newSize,
unsigned char *bcc2_received);
// Conecta e Desconecta com File Descriptor
int connectFD(LinkLayer connectionParametersApp);
int disconnectFD();
// funções de alarme
void alarmHandler(int signal); void alarmDisable();
```

4. Casos de uso principais

O nosso programa pode ser usado no modo de transmissor que enviará um ficheiro para uma outra máquina que estará a correr o programa no modo de recetor que receberá o ficheiro e o armazenará com o nome escolhido. As suas sequências de funções serão ligeiramente diferentes.

Em primeiro lugar, tanto o transmissor como o recetor irão chamar a função `llopen` para que a conexão entre eles seja estabelecida e a porta série seja corretamente configurada. Após isso o recetor chamará `llread` até receber um pacote que sinaliza o fim da transmissão do ficheiro, chamando `readPacketControl` e `ReadPacketData` para interpretar os pacotes recebidos do protocolo de ligação de dados através do `llread`. O transmissor irá por sua vez chamar a função `llwrite` repetidamente para enviar os pacotes de controlo que sinalizam o início e fim da transmissão do ficheiro, e os pacotes de dados que enviam os conteúdos do ficheiro a transmitir até ter enviado o ficheiro completo. Os pacotes são criados e enviados (chamando `llwrite`) com as funções `sendPacketControl` e `sendPacketData`. Por fim, ambas as máquinas chamam a função `llclose` para terminar a sua ligação e imprimir algumas estatísticas relevantes.

5. Protocolo de ligação lógica

O protocolo de ligação lógica centra-se nas quatro funcionalidades seguintes: estabelecer uma conexão com a função `llopen`, enviar uma trama para a outra máquina através do `llwrite`, receber uma trama de outra máquina através do `llread` e fecha a conexão com a função `llclose`. Descrevemos de seguida o que cada uma destas funções faz mais detalhadamente.

5.1. llopen: estabelecimento de conexão

A função `llopen` é definida da seguinte forma: `int llopen(LinkLayer connectionParameters)`.Antes de estabelecer a conexão entre o tranmissor e o recetor, ambas as máquinas guardam uma cópia do *struct* `LinkLayer` para ser usado no resto do programa, e configuram a porta série. Para estabelecer a conexão, o transmissor envia uma trama de supervisão SET, ficando à espera de receber uma trama de supervisão UA vinda do recetor. O recetor quando recebe a trama SET envia a trama UA. Quando o tranmissor recebe a trama UA a conexão foi efetuada com sucesso. Caso o recetor não envie a trama UA dentro do timeout definido em `LinkLayer`, o tranmissor tentará reenviar a trama SET um número pré-definido de vezes que está também definido em `LinkLayer`.

5.2. llwrite: envio de uma trama

A função `llwrite` é chamada da seguinte forma: `int llwrite(const unsigned char *buf, int bufSize)`. Para enviar os dados para o recetor, o transmissor tem de primeiro preparar a trama a ser enviada. Para tal, em primeiro lugar, é feito o stuffing dos dados e do campo BCC2 que o transmissor tem de enviar para que dados com valores hexadecimais iguais ao valor hexadecimal da *flag* não sejam interpretados como flags, o que causaria perda de dados. Após a operação de stuffing, o transmissor prepara a trama a ser enviada com a determinação de todos os campos necessários como os campos BCC1 e BCC2 que são respetivamente os campos de proteção do cabeçalho e dos dados da trama. Envia assim a trama para o recetor esperando pela sua resposta. Caso a resposta seja do tipo RR, a trama foi recebida com sucesso. Caso a resposta seja do tipo REJ, a trama foi recebida com um erro no cabeçalho ou no campo dos dados da trama e é reenviada pelo transmissor. Tal como na função `llopen`, reenvia a trama um número máximo de vezes caso não receba uma resposta num certo período de tempo.

5.3. llread: leitura de uma trama

A função `llread` é chamada da seguinte forma: `int llread(unsigned char *packet)`. Para ler uma trama, o recetor em primeiro lugar tenta ler, através de uma máquina de estados, uma trama de informação. Nesse processo, poderá ocorrer um erro no cabeçalho da trama. Caso isso aconteça, o recetor não envia qualquer resposta, esperando pela retransmissão da trama. Caso o recetor consiga ler uma trama de informação com sucesso, ele aplica destuffing no campo de dados e no BCC2, e verifica se houve algum erro no campo de dados. Se houver um erro é enviada uma trama REJ e o recetor espera pela retranmissão da trama. Se não houver erro, o campo de dados é extraído e enviado para a camada de aplicação assim como o número de bytes que foram lidos no campo de dados.

5.4. llclose: fecho da conexão

A função `llclose` é chamada da seguinte forma: `int llclose(int showStatistics)`. O fecho da conexão funciona de uma maneira semelhante à função `llopen`. Em primeiro lugar o transmissor envia uma trama de supervisão DISC, esperando por uma resposta com a trama DISC vinda do recetor. O recetor quando recebe a trama DISC, envia também ele uma trama DISC, esperando por uma resposta do transmissor com a trama UA. Quando o transmissor recebe a trama DISC, envia uma trama UA. Quando o recetor recebe a trama UA a conexão é fechada com sucesso. Por fim, tanto o transmissor como o recetor voltam a configurar a porta série como estava antes da execução do programa e apresentam estatísticas relevantes se `showStatistics` tiver o valor True.

6. Protocolo de aplicação

Ao nível da aplicação, é feita a interação direta com o ficheiro que se quer transferir. Assim, após o transmissor e o recetor terem estabelecido a conexão ao chamar `llopen`, o recetor entra num ciclo em que está sempre a chamar `llread`, até receber um pacote que sinaliza o final da transmissão do ficheiro. Para tal, a cada vez que um pacote é lido com `llread`, o recetor chama ou a função `readPacketControl` ou a função `readPacketData` para ler os conteúdos do pacote que foi recebido. O recetor vai escrevendo os conteúdos dos pacotes de dados num novo ficheiro. O transmissor, depois de estabelecer a conexão, envia em primeiro lugar um pacote de controle para sinalizar o início da transferência do ficheiro com a função `sendPacketControl`. Este pacote é codificado na forma TLV e envia o tamanho do ficheiro em bytes e o nome do ficheiro que vai enviar. Após o envio deste pacote, o transmissor envia o ficheiro em pacotes com um tamanho pré-determinado com o uso da função `sendPacketControl` para cada um dos pacotes. Por fim, envia um novo pacote de controlo para sinalizar o fim da transferência do ficheiro, tendo este pacote os mesmos conteúdos que o pacote enviado antes da transferência do ficheiro. Quando o ficheiro acaba de ser transferido, o recetor verifica que o número de bytes de pacotes de dados que leu é igual ao tamanho do ficheiro que era suposto ter recebido, verificando assim, ao nível da aplicação, se a transferência do ficheiro foi bem sucedida. Por fim, tanto o transmissor como o recetor fecham a conexão, terminando o programa.

7. Validação

Para validar o correto funcionamento do programa, foram conduzidos diversos testes, incluindo:

- Transferência de ficheiros em condições normais de operação;
- Transferência de ficheiros com interferência de ruído no cabo físico;
- Transferência de ficheiros com interrupção do cabo e subsequente reconexão;
- Transferência de ficheiros após uma tentativa de transferência anterior cancelada;
- Transferência utilizando taxas de transmissão (baud rates) e tempos de espera entre retransmissão (timeouts) variados, mas adequados;
- Transferência com uma taxa de erro de frame (FER) artificialmente induzida e não nula.
- Transferência de ficheiros com tamanhos diferentes.

Em todos casos com valores adequados o ficheiro foi transmitido com sucesso.
Tendo em conta o protocolo *Stop & Wait* a transmissão não era eficiente nos casos de ruído, interrupção do cabo e FER não nulo, por causa de retransmissões.

8. Eficiência do protocolo de ligação de dados

Para cada uma das variações descritas, nós mantivemos os outros campos constantes para não influenciarem os resultados. Os gráficos com os dados recolhidos encontram-se no Anexo II.

8.1. Variar tempo de propagação

O tempo de propagação foi variado com a introdução da função `usleep` para simular o tempo que os dados demoram a percorrer o meio físico. Podemos observar pelo nossos resultados que a eficiência diminui com o aumento do tempo de propagação. Do ponto de vista teórico isso faz sentido, pois a eficiência é inversamente proporcional ao tempo de propagação.

8.2. Variar o tamanho das tramas

O tamanho das tramas foi variado através da mudança do valor da variável `MAX_PAYLOAD_SIZE` no ficheiro `link_layer.h`. No nosso caso, como testamos com valores moderadamente altos, a eficiência foi constante. Isto porque corremos o programa com uma probabilidade de erro igual a 0. Num contexto real, quanto maior fosse a trama, maior a probabilidade de haver erros em cada trama e por isso a eficiência poderia ser menor.

8.3. Variar probabilidade de erro no cabeçalho das tramas de informação

Para adicionar uma probabilidade de ocorrer um erro numa trama de informação, nós usamos a linha `rand % 100` para calcular um valor aleatório entre 0 e 99 inclusive. Definimos também uma variável `FAKE_BCC1_ERR`, que pode ser um valor entre 0 e 100, que define a probabilidade de um erro falso no cabeçalho das tramas de informação. Assim, a cada vez que simulamos um erro destes nós fizemos o recetor não enviar qualquer resposta, como se tratasse de um erro real. Podemos verificar que, tal como de um ponto de vista teórico, a eficiência do programa desce com o aumento da probabilidade de erro. Podemos também reparar que a eficiência desce mais abruptamente que a eficiência ao variar o erro no campo de dados da trama de informação, pois um erro no cabeçalho leva a um *timeout* no transmissor, pois o recetor não envia qualquer resposta.

8.4. Variar probabilidade de erro no campo de dados das tramas de informação

Para adicionar uma probabilidade de ocorrer um erro no campo de dados numa trama de informação, nós usamos a linha `rand % 100` para calcular um valor aleatório entre 0 e 99 inclusive. Definimos também uma variável `FAKE_BCC2_ERR`, que pode ser um

valor entre 0 e 100, que define a probabilidade de um erro falso no campo de dados das tramas de informação. Assim, a cada vez que simulamos um erro destes nós fizemos o recetor não enviar uma resposta REJ, como se tratasse de um erro real. Podemos verificar que, tal como de um ponto de vista teórico, a eficiência do programa desce com o aumento da probabilidade de erro.

8.5 Variar baudrate

Para variar o baudrate do programa modificámos o valor que passámos de baudrate no início do programa. Podemos reparar que no caso geral, a eficiência diminui com o aumento do baudrate, a eficiência que calculámos é inversamente proporcional ao valor do baudrate. Ou seja, em baudrates menores, o meio físico é melhor aproveitado que em baudrates maiores em que uma grande parte do meio físico se encontra vazio durante a transmissão dos dados.

9. Conclusões

Com este projeto tivemos uma vertente prática nos conceitos de transferência, uso de serviços e independência entre camadas, e o mecanismo ARQ *Stop & Wait*, sendo uma boa forma de consolidar tais conceitos. Concluímos também que o mecanismo ARQ *Stop & Wait* é bastante lento se compararmos as eficiências que tivemos na práticas com as eficiências que são esperadas na teoria.

O nosso programa foi desenvolvido de acordo com os requisitos fornecidos, sendo assim robusto e bem organizado em camadas.

Anexo I - Código fonte.

link_layer.h

```
// Link layer header.
// NOTE: This file must not be changed.

#ifndef _LINK_LAYER_H_
#define _LINK_LAYER_H_

typedef enum
{
      LlTx,
      LlRx,
} LinkLayerRole;

typedef struct
{
      char serialPort[50];
      LinkLayerRole role;
      int baudRate;
      int nRetransmissions;
      int timeout;
} LinkLayer;

// SIZE of maximum acceptable payload.
// Maximum number of bytes that application layer should send to link
layer
#define MAX_PAYLOAD_SIZE 1000

// MISC
#define FALSE 0
#define TRUE 1

// Open a connection using the "port" parameters defined in struct
linkLayer.
// Return "1" on success or "-1" on error.
int llopen(LinkLayer connectionParameters);
```

```
// Send data in buf with size bufSize.
// Return number of chars written, or "-1" on error.
int llwrite(const unsigned char *buf, int bufSize);

// Receive data in packet.
// Return number of chars read, or "-1" on error.
int llread(unsigned char *packet);

// Close previously opened connection.
// if showStatistics == TRUE, link layer should print statistics in the
console on close.
// Return "1" on success or "-1" on error.
int llclose(int showStatistics);

#endif // _LINK_LAYER_H_
```

---

link_layer.c

```
// Link layer protocol implementation

#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>
#include <signal.h>
#include <sys/time.h>
#include <time.h>

#include "link_layer.h"


// MISC
#define _POSIX_SOURCE 1 // POSIX compliant source
```

```
#define FALSE 0
#define TRUE 1

#define FRAME_SIZE 5

#define FLAG 0x7E
#define ESC 0x7D
#define ESC_FLAG 0x5E
#define ESC_ESC 0x5D
#define A_SEND 0x03
#define A_RECV 0x01
#define C_SET 0x03
#define C_UA 0x07
#define C_DISC 0x0B

#define C_INF0 0x00
#define C_INF1 0x40

#define RR0 0x05
#define RR1 0x85
#define REJ0 0x01
#define REJ1 0x81

// NOTE: only for report statistics
#define FAKE_BCC1_ERR 10.0
#define FAKE_BCC2_ERR 10.0
#define TPROP 500
#define FILE_SIZE 10968

typedef struct
{
size_t bytes_read; // Number of bytes read before any destuffing
unsigned int nFrames; // Number of good frames sent/received
unsigned int errorFrames;
unsigned int frames_size; // Size of good frames sent
double time_send_control; // Time spent on sending control frames
double time_send_data; // Time spent on sending data frames
struct timeval start; // When program starts
} Statistics;
```

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

LinkLayer connectionParameters;
Statistics statistics = {0, 0, 0, 0, 0.0, 0.0};
struct termios oldtio;
int alarmEnabled = FALSE;
int alarmCount = 0;
int fd;
unsigned char C_Ns = 0; // Ns
unsigned char C_Nr = 0; // Nr (o valor que ele espera de receber)

int get_boudrate(int boudrate){
switch (boudrate)
{
case 50 : return B50;
case 75: return B75;
case 110: return B110;
case 134: return B134;
case 150: return B150;
case 200: return B200;
case 300: return B300;
case 600: return B600;
case 1200: return B1200;
case 1800: return B1800;
case 2400: return B2400;
case 4800: return B4800;
case 9600: return B9600;
case 19200: return B19200;
case 38400: return B38400;
default: return B9600;
}
}
```

```
double get_time_difference(struct timeval ti, struct timeval tf) {
return (tf.tv_sec - ti.tv_sec) + (tf.tv_usec - ti.tv_usec) / 1e6;
}

int connectFD(LinkLayer connectionParametersApp)
{
if (connectionParametersApp.serialPort[0] == '\0') return -1;
fd = open(connectionParametersApp.serialPort, O_RDWR | O_NOCTTY);
if (fd < 0)
{
perror(connectionParametersApp.serialPort);
return -1;
}

struct termios newtio;

// Save current port settings
if (tcgetattr(fd, &oldtio) == -1)
{
perror("tcgetattr");
return -1;
}

// Clear struct for new port settings
memset(&newtio, 0, sizeof(newtio));

newtio.c_cflag = (get_boudrate(connectionParametersApp.baudRate)) | CS8
| CLOCAL | CREAD;
newtio.c_iflag = IGNPAR;
newtio.c_oflag = 0;

newtio.c_lflag = 0;

newtio.c_cc[VTIME] = 10 * connectionParameters.timeout;
newtio.c_cc[VMIN] = 0;

tcflush(fd, TCIOFLUSH);

if (tcsetattr(fd, TCSANOW, &newtio) == -1)
{
perror("tcsetattr");
```

```
return -1;
}

printf("New termios structure set\n");

return 0;
}

int disconnectFD()
{
if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
{
perror("tcsetattr");
return -1;
}
return 0;
}

void alarmHandler(int signal)
{
alarmCount++;
alarmEnabled = TRUE;
printf("Alarm count: %d\n", alarmCount);
}

void alarmDisable()
{
alarm(0);
alarmEnabled = FALSE;
alarmCount = 0;
}

int send_packet_command( unsigned char A, unsigned char C)
{
unsigned char buf[FRAME_SIZE] = {FLAG, A, C, 0, FLAG};
buf[3] = buf[1] ^ buf[2];
if(write(fd, buf, FRAME_SIZE) < 0)
{
perror("Error write send command");
return -1;
}
```

```
return 0;
}

int receivePacket(unsigned char A_EXPECTED, unsigned char C_EXPECTED)
{
enum state enum_state = START;
while (enum_state != STOP)
{
unsigned char byte = 0;
int bytes;
if((bytes = read(fd, &byte, sizeof(byte))) < 0)
{
perror("Error read DISC command");
return -1;
}
if(bytes > 0){
switch (enum_state)
{
case START:
if(byte == FLAG) enum_state = FLAG_RCV;
break;
case FLAG_RCV:
if(byte == FLAG) continue;
if(byte == A_EXPECTED) enum_state = A_RCV;
else enum_state = START;
break;
case A_RCV:
if(byte == C_EXPECTED) enum_state = C_RCV;
else if(byte == FLAG) enum_state = FLAG_RCV;
else enum_state = START;
break;
case C_RCV:
if(byte == (C_EXPECTED ^ A_EXPECTED)) enum_state = BCC_OK;
else if(byte == FLAG) enum_state = FLAG_RCV;
else enum_state = START;
break;
case BCC_OK:
if(byte == FLAG) enum_state = STOP;
else enum_state = START;
break;
default:
```

```
enum_state = START;
}
}
}
return 0;
}

int receivePacketRetransmission(unsigned char A_EXPECTED, unsigned char
C_EXPECTED, unsigned char A_TO_SEND, unsigned char C_TO_SEND)
{
enum state enum_state = START;
(void)signal(SIGALRM, alarmHandler);
if(send_packet_command(A_TO_SEND, C_TO_SEND)) return -1;
alarm(connectionParameters.timeout);
while (enum_state != STOP && alarmCount <=
connectionParameters.nRetransmissions)
{
unsigned char byte = 0;
int bytes;
if((bytes = read(fd, &byte, sizeof(byte))) < 0)
{
perror("Error read UA command");
return -1;
}
if(bytes > 0){
switch (enum_state)
{
case START:
if(byte == FLAG) enum_state = FLAG_RCV;
break;
case FLAG_RCV:
if(byte == FLAG) continue;
if(byte == A_EXPECTED) enum_state = A_RCV;
else enum_state = START;
break;
case A_RCV:
if(byte == C_EXPECTED) enum_state = C_RCV;
else if(byte == FLAG) enum_state = FLAG_RCV;
else enum_state = START;
break;
case C_RCV:
```

```
if(byte == (C_EXPECTED ^ A_EXPECTED)) enum_state = BCC_OK;
else if(byte == FLAG) enum_state = FLAG_RCV;
else enum_state = START;
break;
case BCC_OK:
if(byte == FLAG) enum_state = STOP;
else enum_state = START;
break;
default:
enum_state = START;
}
}

if(enum_state == STOP)
{
alarmDisable();
return 0;
}
if(alarmEnabled)
{
alarmEnabled = FALSE;
if (alarmCount <= connectionParameters.nRetransmissions) {
if(send_packet_command(A_TO_SEND, C_TO_SEND)) return -1;
alarm(connectionParameters.timeout);
}
enum_state = START;
}
}

alarmDisable();
return -1;
}

int llopen(LinkLayer connectionParametersApp)
{
gettimeofday(&statistics.start, NULL);

memcpy(&connectionParameters, &connectionParametersApp,
sizeof(connectionParametersApp));

if (connectFD(connectionParameters) == -1)
```

```
return -1;

if(connectionParameters.role == LlTx){
struct timeval temp_start, temp_end;
gettimeofday(&temp_start, NULL);

if(receivePacketRetransmission(A_SEND, C_UA, A_SEND, C_SET)) return -1;

statistics.nFrames++;

gettimeofday(&temp_end, NULL);

statistics.time_send_control += get_time_difference(temp_start,
temp_end);

printf("Connection established\n");
}
if(connectionParameters.role == LlRx) {
srand(time(NULL));
if(receivePacket(A_SEND, C_SET)) return -1;
statistics.nFrames++;
statistics.bytes_read += FRAME_SIZE;
if(send_packet_command(A_SEND, C_UA)) return -1;
printf("Connection established\n");
}
return 0;
}

const unsigned char * byteStuffing(const unsigned char *buf, int
bufSize, int *newSize)
{
if(buf == NULL || newSize == NULL) return NULL;

unsigned char *result = (unsigned char *) malloc(bufSize * 2 + 1);
if(result == NULL) return NULL;
size_t j = 0;

for(size_t i = 0; i < bufSize; i++, j++){
if(buf[i] == FLAG){
result[j++] = ESC;
result[j] = ESC_FLAG;
```

```
}
else if (buf[i] == ESC) {
result[j++] = ESC;
result[j] = ESC_ESC;
}
else
result[j] = buf[i];
}

*newSize = (int) j;
result = realloc(result, j);

if (result == NULL) return NULL;

return result;
}

int llwrite(const unsigned char *buf, int bufSize)
{
if(buf == NULL) return -1;

int newSize;
const unsigned char *newBuf = byteStuffing(buf, bufSize, &newSize);
if(newBuf == NULL) return -1;
printf("Bytes sent: %d\n", newSize);
unsigned char *trama = (unsigned char *) malloc(newSize + 6);
if(trama == NULL){
free((unsigned char *) newBuf);
return -1;
}

trama[0] = FLAG;
trama[1] = A_SEND;
trama[2] = (C_Ns) ? C_INF1 : C_INF0;
trama[3] = trama[1] ^ trama[2];
memcpy(trama + 4, newBuf, newSize);


unsigned char bcc2 = 0x00;
for(size_t i = 0; i < bufSize; i++) bcc2 ^= buf[i];
trama[newSize + 4] = bcc2;
```

```
if(bcc2 == FLAG){
trama[newSize + 4] = ESC;
newSize++;
trama[newSize + 4] = ESC_FLAG;
trama = realloc(trama, newSize + 6);
trama[newSize + 5] = FLAG;
}else{
trama[newSize + 5] = FLAG;
}

enum state enum_state = START;
(void)signal(SIGALRM, alarmHandler);

struct timeval temp_start;
gettimeofday(&temp_start, NULL);

if(write(fd, trama, (newSize + 6)) < 0)
{
free(trama);
perror("Error write send command");
return -1;
}
alarm(connectionParameters.timeout);
unsigned char C_received = 0, A_received = 0;

while (enum_state != STOP && alarmCount <=
connectionParameters.nRetransmissions)
{
unsigned char byte = 0;
int bytes;
if((bytes = read(fd, &byte, sizeof(byte))) < 0)
{
free(trama);
perror("Error read command");
return -1;
}
if(bytes > 0){
switch (enum_state)
{
case START:
C_received = 0;
```

```
A_received = 0;
if(byte == FLAG) enum_state = FLAG_RCV;
break;
case FLAG_RCV:
if(byte == FLAG) continue;
if(byte == A_SEND || byte == A_RECV) {
enum_state = A_RCV;
A_received = byte;
}
else enum_state = START;
break;
case A_RCV:
if(byte == RR0 || byte == RR1 || byte == REJ0 || byte == REJ1) {
enum_state = C_RCV;
C_received = byte;
}
else if(byte == FLAG) enum_state = FLAG_RCV;
else enum_state = START;
break;
case C_RCV:
if(byte == (C_received ^ A_received)) enum_state = BCC_OK;
else if(byte == FLAG) enum_state = FLAG_RCV;
else enum_state = START;
break;
case BCC_OK:
if(byte == FLAG) enum_state = STOP;
else enum_state = START;
break;
default:
enum_state = START;
}
}

if(enum_state == STOP)
{
if(C_received == REJ0 || C_received == REJ1){
alarmEnabled = TRUE;
alarmCount = 0;
printf("Received reject; Second try.\n");
}
if(C_received == RR0 || C_received == RR1) {
```

```
struct timeval temp_end;
gettimeofday(&temp_end, NULL);

statistics.time_send_data += get_time_difference(temp_start, temp_end);

alarmDisable();
C_Ns = 1 - C_Ns;
statistics.nFrames++;
free(trama);
return bufSize;
}
}
if(alarmEnabled)
{
alarmEnabled = FALSE;

if (alarmCount <= connectionParameters.nRetransmissions) {
if(write(fd, trama, (newSize + 6)) < 0)
{
perror("Error write send command");
return -1;
}
alarm(connectionParameters.timeout);
}

enum_state = START;
}
}

alarmDisable();
free(trama);

return -1;
}

int byteDestuffing(unsigned char *buf, int bufSize, int *newSize,
unsigned char *bcc2_received)
{
if (buf == NULL || newSize == NULL) return -1;
if (bufSize < 1) return 0;
```

```
unsigned char *read = buf; // Pointer for reading from buf
unsigned char *write = buf; // Pointer for writing to buf

while (read < buf + bufSize) {
if (*read != ESC) {
*write++ = *read++;
} else {
if (*(read + 1) == ESC_FLAG) {
*write++ = FLAG;
} else if (*(read + 1) == ESC_ESC) {
*write++ = ESC;
}
read += 2;
}
}

*bcc2_received = *(write - 1);
*newSize = write - buf - 1;
return 0;
}

int llread(unsigned char *packet)
{
usleep(TPROP * 1000);

enum state enum_state = START;
unsigned char C_received = 0;
size_t pkt_indx = 0;
while (enum_state != STOP)
{
unsigned char byte = 0;
int bytes;
if((bytes = read(fd,&byte, sizeof(byte))) < 0)
{
perror("Error read DISC command");
return -1;
}
if(bytes > 0){
switch (enum_state)
{
case START:
```

```
C_received = 0;
pkt_indx = 0;
if(byte == FLAG) enum_state = FLAG_RCV;
break;
case FLAG_RCV:
if(byte == FLAG) continue;
if(byte == A_SEND) enum_state = A_RCV;
else enum_state = START;
break;
case A_RCV:
if(byte == C_INF0 || byte == C_INF1){
enum_state = C_RCV;
C_received = byte;
}
else if(byte == FLAG) enum_state = FLAG_RCV;
else enum_state = START;
break;
case C_RCV:
if (byte == (C_received ^ A_SEND)) enum_state = DATA; // need to check
A_SEND
else {
statistics.errorFrames++;
if(byte == FLAG) enum_state = FLAG_RCV;
else enum_state = START;
}
break;
case DATA:
if(byte == FLAG) {
int newSize = 0;
unsigned char bcc2_received = 0;
if (byteDestuffing(packet, pkt_indx, &newSize, &bcc2_received)) return
-1;

unsigned char bcc2 = 0x00;
for (size_t i = 0; i < newSize; i++) bcc2 ^= packet[i];

unsigned char C_respons, A_respons;

if (bcc2 == bcc2_received) {
C_respons = (C_received == C_INF0)? RR1 : RR0;
A_respons = A_SEND;
```

```
}
else {
if ((C_Nr == 0 && C_received == C_INF1) || (C_Nr == 1 && C_received ==
C_INF0)) {
C_respons = (C_received == C_INF0)? RR1 : RR0;
A_respons = A_SEND;
}
else {
C_respons = (C_received == C_INF0) ? REJ0 : REJ1;
A_respons = A_SEND;
}
}

enum_state = START;

int error_in_bcc1 = rand() % 100;
int error_in_bcc2 = rand() % 100;


if ((C_Nr == 0 && C_received == C_INF0) || (C_Nr == 1 && C_received ==
C_INF1)) {
if (error_in_bcc1 <= FAKE_BCC1_ERR - 1) {
statistics.errorFrames++;
break;
}

if (error_in_bcc2 <= FAKE_BCC2_ERR - 1) {
C_respons = (C_received == C_INF0) ? REJ0 : REJ1;
A_respons = A_SEND;
}
}

usleep(TPROP * 1000);

if (send_packet_command(A_respons, C_respons)) return -1;

if (C_respons == REJ0 || C_respons == REJ1) {
statistics.errorFrames++;
break;
}
```

```
if ((C_Nr == 0 && C_received == C_INF0) || (C_Nr == 1 && C_received ==
C_INF1)){
C_Nr = 1 - C_Nr;
printf("Bytes received: %d\n", newSize);
statistics.bytes_read += newSize + 6;
statistics.nFrames++;
return newSize;
}
printf("Received duplicate\n");
} else packet[pkt_indx++] = byte;
break;
default:
enum_state = START;
}
}
}
return -1;
}

void printStatistics()
{
printf("\n======== Statistics ========\n");

if (connectionParameters.role == LlRx) {
printf("MAXPAYLOAD: %d\n", MAX_PAYLOAD_SIZE);

float a = (float) ((float) TPROP / 1000) / (float) ((float)
MAX_PAYLOAD_SIZE * 8.0 / (float) connectionParameters.baudRate);

float REAL_FER = (float) statistics.errorFrames / (statistics.nFrames +
statistics.errorFrames);

float EXPECTED_FER = FAKE_BCC1_ERR/100.0 + (100.0 -
FAKE_BCC1_ERR)/100.0 * FAKE_BCC2_ERR/100.0;

printf("EXPECTED FER: %f\n", EXPECTED_FER);

printf("EXPECTED A: %f\n", a);

printf("\nNumber of bytes received (after destuffing): %lu\n",
statistics.bytes_read);
```

```
printf("\nNumber of good frames received: %d frames\n",
statistics.nFrames);

struct timeval end;
gettimeofday(&end, NULL);

printf("\nTime taken to download file: %f seconds\n",
get_time_difference(statistics.start, end));

printf("\nAverage size of a frame: %ld bytes per frame\n",
statistics.bytes_read / statistics.nFrames);

printf("\nDébito recebido (bits/s): %f\n", (float)
statistics.bytes_read * 8.0 / get_time_difference(statistics.start,
end));

printf("\nBaudrate real: %d\n", connectionParameters.baudRate);

printf("\nFER: %f\n", REAL_FER);

printf("\nEficiencia teorica: %f", (1.0 - EXPECTED_FER) / (1 + 2*a));

printf("\nEficiencia pratica: %f", (FILE_SIZE * 8.0) /
get_time_difference(statistics.start, end));

printf("\nEficiencia pedida %f\n", ((float) ((float) FILE_SIZE * 8.0) /
get_time_difference(statistics.start, end)) / (float)
connectionParameters.baudRate);
}

else {
printf("\nNumber of good frames sent: %d frames\n",
statistics.nFrames);

printf("\nTime taken to send and receive confirmation of receival of
control frames: %f seconds\n", statistics.time_send_control);

printf("\nTime taken to send and receive confirmation of receival of
data frames: %f seconds\n", statistics.time_send_data);
```

```
printf("\nAverage time taken to send a frame: %f seconds\n",
(statistics.time_send_data + statistics.time_send_control) /
statistics.nFrames);
}

printf("\n============================\n");
}

int llclose(int showStatistics)
{
if(connectionParameters.role == LlTx)
{
struct timeval temp_start, temp_end;

gettimeofday(&temp_start, NULL);

if(receivePacketRetransmission(A_RECV, C_DISC, A_SEND, C_DISC))
return disconnectFD();

statistics.nFrames++;
gettimeofday(&temp_end, NULL);

statistics.time_send_control += get_time_difference(temp_start,
temp_end);
statistics.nFrames++;

if(send_packet_command(A_RECV, C_UA)) return disconnectFD();
printf("Disconnected\n");
}
if(connectionParameters.role == LlRx)
{
if(receivePacket(A_SEND, C_DISC)) return disconnectFD();
statistics.nFrames++;
statistics.bytes_read += FRAME_SIZE;


if(receivePacketRetransmission(A_RECV, C_UA, A_RECV, C_DISC)) return
disconnectFD();
statistics.nFrames++; // Retirar este linha se se mudar em cima para
send_packet_command
```

```
statistics.bytes_read += FRAME_SIZE; // Retirar este linha se se mudar
em cima para send_packet_command

printf("Disconnected\n");
// if(send_packet_command(A_RECV, C_DISC)) return disconnectFD();
}

if(showStatistics) printStatistics();
return disconnectFD();
}
```

---

application_layer.h

```
// Application layer protocol header.
// NOTE: This file must not be changed.

#ifndef _APPLICATION_LAYER_H_
#define _APPLICATION_LAYER_H_

// Application layer main function.
// Arguments:
// serialPort: Serial port name (e.g., /dev/ttyS0).
// role: Application role {"tx", "rx"}.
// baudrate: Baudrate of the serial port.
// nTries: Maximum number of frame retries.
// timeout: Frame timeout.
// filename: Name of the file to send / receive.
void applicationLayer(const char *serialPort, const char *role, int
baudRate,
int nTries, int timeout, const char *filename);

#endif // _APPLICATION_LAYER_H_
```

---

application_layer.c

```
// Application layer protocol implementation

#include "application_layer.h"
#include "link_layer.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include<unistd.h>

#define DATA 1
#define C_START 2
#define C_END 3

#define T_FILESIZE 0
#define T_FILENAME 1

#define MAX_FILENAME 50

enum state{
RECV_START,
RECV_CONT,
RECV_END
};

typedef struct
{
size_t file_size;
char * file_name;
size_t bytesRead;
} FileProps;

enum state stateReceive = RECV_START;
FileProps fileProps = {0, "", 0};


int sendPacketData(size_t nBytes, unsigned char *data)
```

```
{
if(data == NULL) return -1;
unsigned char *packet = (unsigned char *) malloc(nBytes + 3);
if(packet == NULL) return -1;
packet[0] = DATA;
packet[1] = nBytes >> 8;
packet[2] = nBytes & 0xFF;

memcpy(packet + 3, data, nBytes);

int result = llwrite(packet, nBytes + 3);

free(packet);

return result;
}

unsigned char * itouchar(size_t value, unsigned char *size)
{
if (size == NULL) return NULL;
size_t tmp_value = value;
size_t length = 0;
do {
length++;
tmp_value >>= 8;
} while (tmp_value);

unsigned char *bytes = malloc(length);
if (bytes == NULL) return NULL;

for (size_t i = 0; i < length; i++, value >>= 8)
bytes[i] = value & 0xFF;

*size = length;
return bytes;
}

size_t uchartoi (unsigned char n, unsigned char * numbers)
{
if(numbers == NULL) return 0;
size_t value = 0;
```

```
size_t power = 1;
for(int i = 0; i < n; i++, power <<= 8){
value += numbers[i] * power;
}
return value;
}

int sendPacketControl(unsigned char C, const char * filename, size_t
file_size)
{
if(filename == NULL) return -1;
unsigned char L1 = 0;
unsigned char * V1 = itouchar(file_size, &L1);
if(V1 == NULL) return -1;

unsigned char L2 = (unsigned char) strlen(filename);
unsigned char *packet = (unsigned char *) malloc(5 + L1 + L2);
if(packet == NULL) {
free(V1);
return -1;
}

size_t indx = 0;
packet[indx++] = C;
packet[indx++] = T_FILESIZE;
packet[indx++] = L1;
memcpy(packet + indx, V1, L1); indx += L1;
packet[indx++] = T_FILENAME;
packet[indx++] = L2;
memcpy(packet + indx, filename, L2); indx += L2;
free(V1);
int res = llwrite(packet, (int) indx);

free(packet);

return res;
}

unsigned char * readPacketData(unsigned char *buff, size_t *newSize)
{
if (buff == NULL) return NULL;
```

```
if (buff[0] != DATA) return NULL;

*newSize = buff[1] * 256 + buff[2];

return buff + 3;
}

int readPacketControl(unsigned char * buff)
{
if (buff == NULL) return -1;
size_t indx = 0;

char * file_name = malloc(MAX_FILENAME);
if(file_name == NULL) return -1;

if(buff[indx] == C_START) stateReceive = RECV_CONT;
else if(buff[indx] == C_END) stateReceive = RECV_END;
else {
free(file_name);
return -1;
}

indx++;
if (buff[indx++] != T_FILESIZE) return -1;
unsigned char L1 = buff[indx++];
unsigned char * V1 = malloc(L1);
if(V1 == NULL) return -1;
memcpy(V1, buff + indx, L1); indx += L1;
size_t file_size = uchartoi(L1, V1);
free(V1);

if(buff[indx++] != T_FILENAME) return -1;
unsigned char L2 = buff[indx++];
memcpy(file_name, buff + indx, L2);
file_name[L2] = '\0';

if(buff[0] == C_START){
fileProps.file_size = file_size;
fileProps.file_name = file_name;
printf("[INFO] Started receiving file: '%s'\n", file_name);
}
```

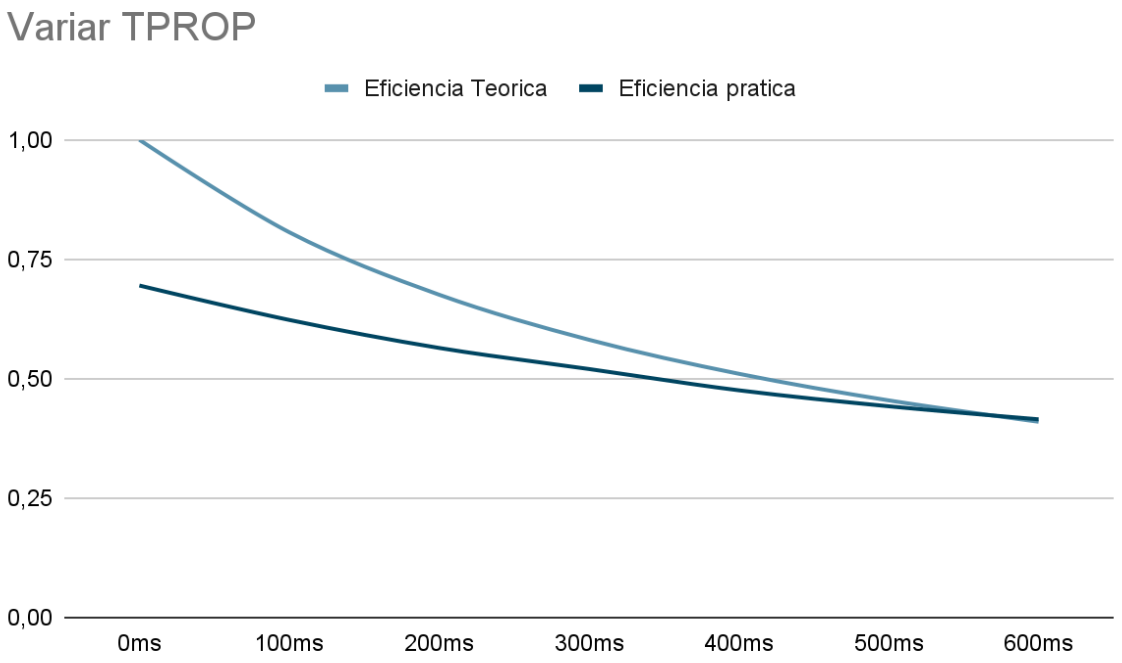
```
if(buff[0] == C_END){
if (fileProps.file_size != fileProps.bytesRead) {
perror("Number of bytes read doesn't match size of file\n");
}
if(strcmp(fileProps.file_name, file_name)){
perror("Names of file given in the start and end packets don't
match\n");
}
printf("[INFO] Finished receiving file: '%s'\n", file_name);
}
free(file_name);
return 0;
}

void applicationLayer(const char *serialPort, const char *role, int
baudRate,
int nTries, int timeout, const char *filename)
{
if(serialPort == NULL || role == NULL || filename == NULL){
perror("Initialization error: One or more required arguments are
NULL.");
return;
}

if (strlen(filename) > MAX_FILENAME) {
printf("The lenght of the given file name is greater than what is
supported: %d characters'\n", MAX_FILENAME);
return;
}
LinkLayer connectionParametersApp;
strncpy(connectionParametersApp.serialPort, serialPort,
sizeof(connectionParametersApp.serialPort)-1);
connectionParametersApp.role = strcmp(role, "tx") ? LlRx : LlTx;
connectionParametersApp.baudRate = baudRate;
connectionParametersApp.nRetransmissions = nTries;
connectionParametersApp.timeout = timeout;

if (llopen(connectionParametersApp) == -1) {
perror("Link layer error: Failed to open the connection.");
llclose(FALSE);
return;
```

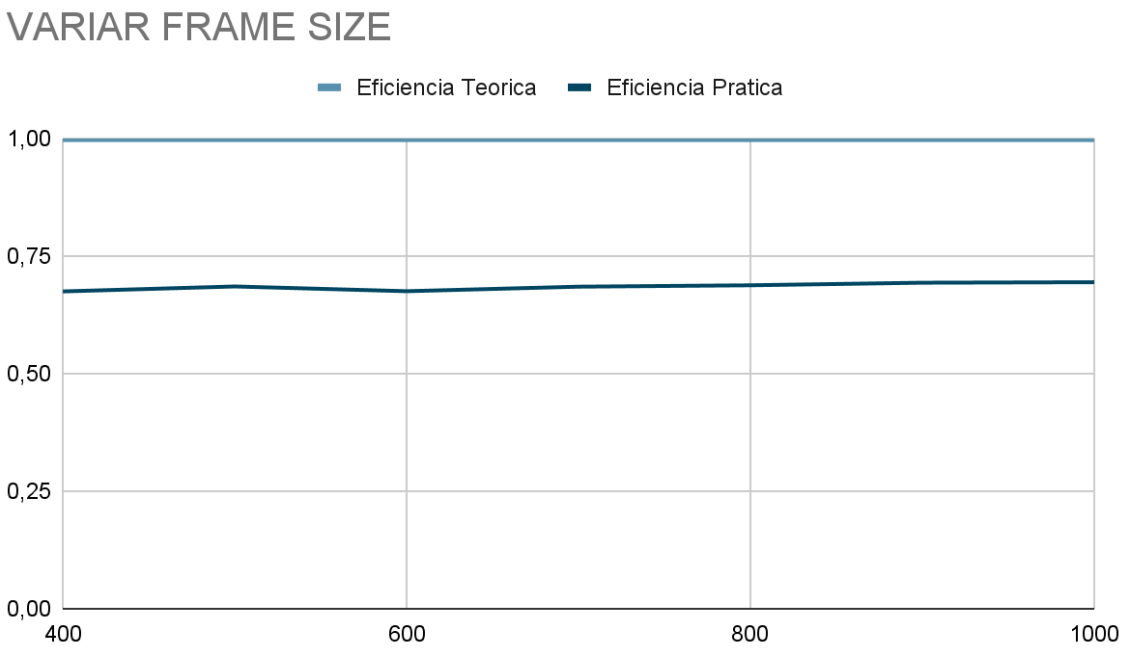
```
}
if (connectionParametersApp.role == LlTx) {
size_t bytesRead = 0;
unsigned char *buffer = (unsigned char *) malloc(MAX_PAYLOAD_SIZE +
20);
if(buffer == NULL) {
perror("Memory allocation error at buffer creation.");
llclose(FALSE);
return;
}

FILE* file = fopen(filename, "rb");
if(file == NULL) {
perror("File error: Unable to open the file for reading.");
fclose(file);
free(buffer);
llclose(FALSE);
return;
}

fseek(file, 0, SEEK_END);
size_t file_size = ftell(file);
rewind(file);

if(sendPacketControl(C_START, filename, file_size) == -1) {
perror("Transmission error: Failed to send the START packet control.");
fclose(file);
llclose(FALSE);
return;
}

while ((bytesRead = fread(buffer, 1, MAX_PAYLOAD_SIZE, file)) > 0) {
size_t sended_bytes = 0;
sended_bytes = sendPacketData(bytesRead, buffer);
if(sended_bytes == -1){
perror("Transmission error: Failed to send the DATA packet control.");
fclose(file);
llclose(FALSE);
return;
}
}
```

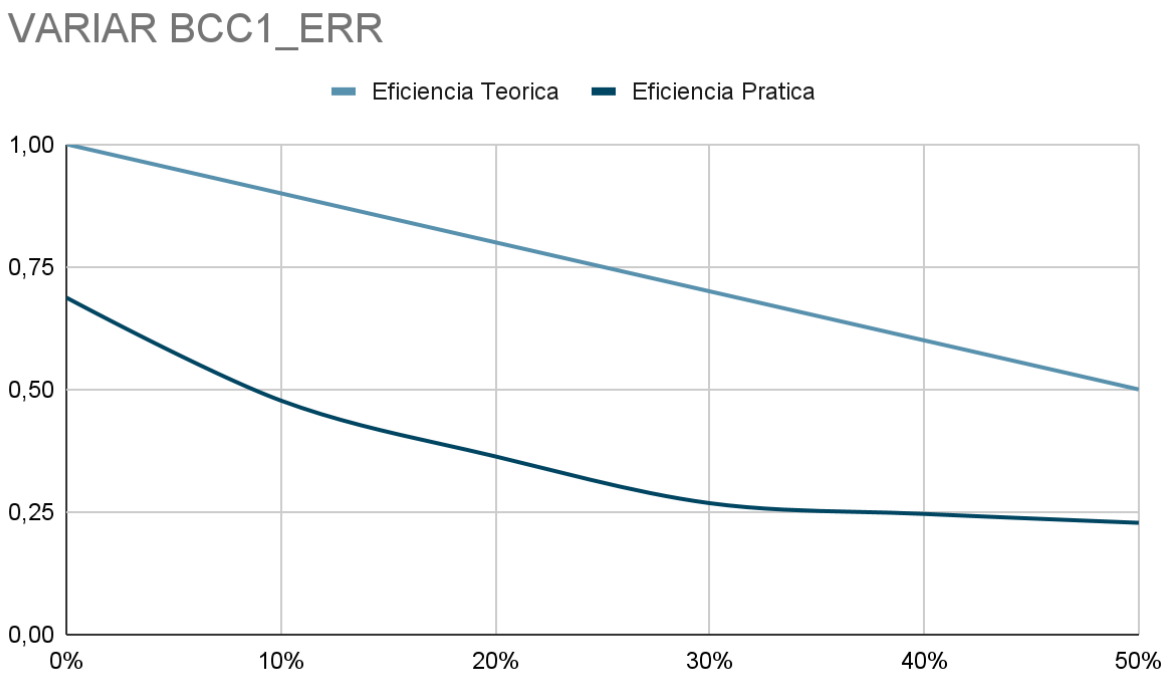
```
if(sendPacketControl(C_END, filename, file_size) == -1){
perror("Transmission error: Failed to send the END packet control.");
fclose(file);
llclose(FALSE);
return;
}

fclose(file);
}
if (connectionParametersApp.role == LlRx) {
unsigned char * buf = malloc(MAX_PAYLOAD_SIZE + 20);
unsigned char * packet = malloc(MAX_PAYLOAD_SIZE + 20);
if(buf == NULL || packet == NULL){
perror("Initialization error: One or more buffers pointers are NULL.");
llclose(FALSE);
return;
}

FILE *file = fopen(filename, "wb");
// To save the file with the same name as the one sent, use:
// FILE *file = fopen(fileProps.file_name, "wb");
if(file == NULL) {
perror("File error: Unable to open the file for writing.");
fclose(file);
llclose(FALSE);
return;
}

size_t bytes_readed = 0;
while(stateReceive != RECV_END){
bytes_readed = llread(buf);

if(bytes_readed == -1) {
perror("Link layer error: Failed to read from the link.");
fclose(file);
llclose(FALSE);
return;
}
if(buf[0] == C_START || buf[0] == C_END){
if(readPacketControl(buf) == -1) {
```

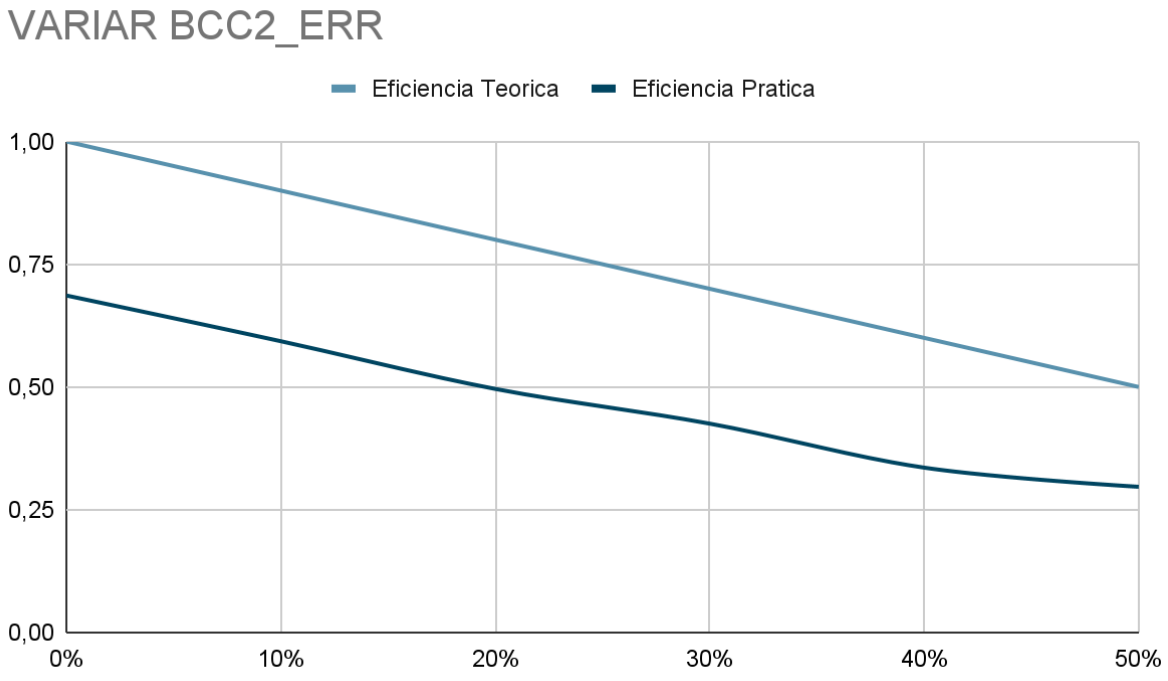
```
perror("Packet error: Failed to read control packet.");
fclose(file);
llclose(FALSE);
return;
}
}else if(buf[0] == DATA){
packet = readPacketData(buf, &bytes_readed);
if(packet == NULL) {
perror("Packet error: Failed to read data packet.");
fclose(file);
llclose(FALSE);
return;
}
fwrite(packet, 1, bytes_readed, file);
fileProps.bytesRead += bytes_readed;
}
}

fclose(file);
}


if (llclose(TRUE) == -1) {
perror("Link layer error: Failed to close the connection.");
return;
}
}
```

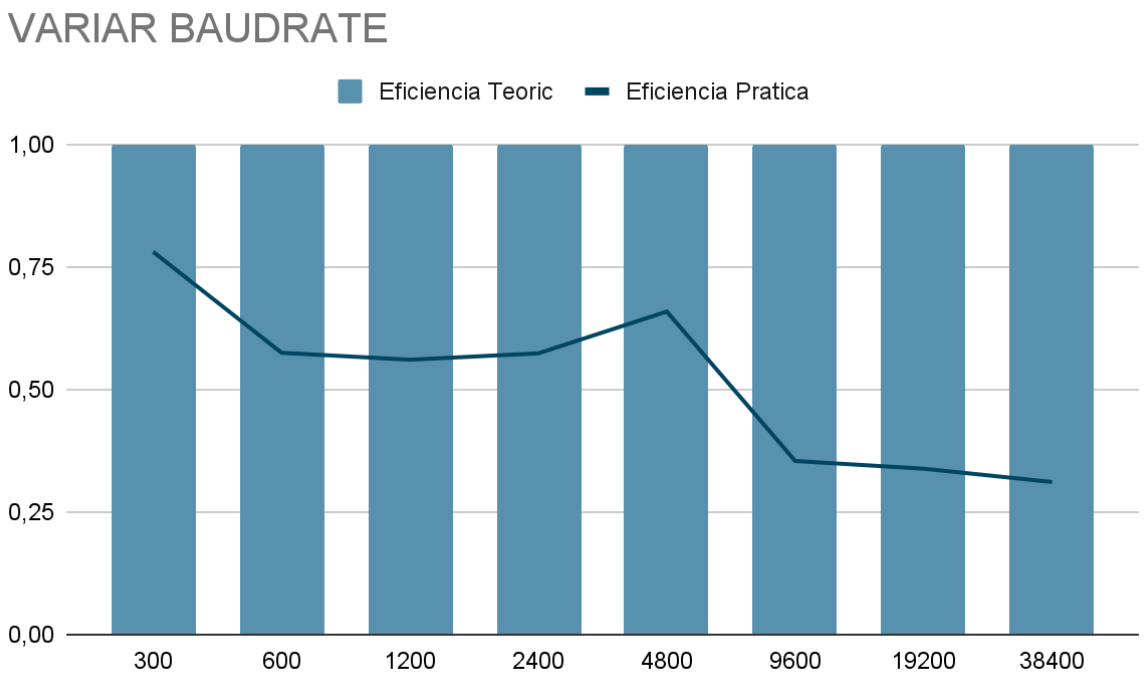
Anexo II - Gráficos.






VARIAR BCC1_ERR
Eficiencia Teorica
Eficiencia Pratica
1,00
0,75
0,50
0,25
0,00
0%
10%
20%
30%
40%
50%


VARIAR BCC2_ERR
Eficiencia Teorica
Eficiencia Pratica
1,00
0,75
0,50
0,25
0,00
0%
10%
20%
30%
40%
50%


VARIAR BAUDRATE
Eficiencia Teoric
Eficiencia Pratica
1,00
0,75
0,50
0,25
0,00
300
600
1200
2400
4800
9600
19200
38400