Protocolo de ligação de dados

Mestrado Integrado em Engenharia Informática e de Computadores

Índice

Sumário

Este projeto foi elaborado no contexto do primeiro trabalho prático no âmbito da unidade curricular de Redes de Computadores do curso de Mestrado Integrado em Engenharia Informática e da Computação, o qual consiste em desenvolver um protocolo simples de transmissão de dados entre computadores operando através de uma porta de série assíncrona e protegido em relação a possíveis erros tanto relacionados aos dados propriamente dito, como também a erros de comunicação.

O trabalho foi realizado com sucesso, sendo que cumprimos com todos os objetivos estabelecidos no guião tendo como resultado uma aplicação totalmente funcional capaz de enviar qualquer tipo de ficheiro entre dois computadores e sem perda de dados.

1. Introdução

Este relatório é um complemento ao primeiro trabalho prático abordado sob uma esfera investigativa, ou seja, estendemos análises à respeito da teoria estudada no projeto, incluindo medições estatísticas de sua execução e esclarecimento de erros e suas validações.

O tema do trabalho engloba uma pluralidade de conceitos na transferência de ficheiros entre computadores por tramas e protocolos de ligação.

Este projeto está dividido nas seguintes partes principais:

Arquitetura: Exibição dos blocos funcionais e interfaces presentes.
Estrutura do Código: API's utilizadas, principais funções e suas relações com a arquitetura.
Casos de Uso: Identificação dos casos e exemplificação da linearidade seguida pela execução.
Protocolo de Ligação Lógica: Identificação dos principais aspectos funcionais; descrição da estratégia de implementação destes aspectos com apresentação de extratos de código.
Protocolo de Aplicação: Identificação dos principais aspectos funcionais; descrição da estratégia de implementação destes aspectos com apresentação de extratos de código.
Validação: Descrição dos testes efectuados com apresentação quantificada dos resultados.
Eficiência do Protocolo de Ligação de Dados: Caracterização estatística da eficiência do protocolo, feita com recurso a medidas sobre o código desenvolvido.

2. Arquitetura

Camadas (Layers)

Para estabelecer o protocolo com sucesso, o código é manipulado através de duas camadas: a lógica e a de aplicação.

A camada lógica (estruturada nos ficheiros *link_layer.c* e *link_layer.h*) contém as funções de envio e receção das tramas, ou seja, é nesta camada que acontece a conexão, verificação de erros, escrita e leitura na porta de série.

Já a camada de aplicação inclui funções responsáveis por fazer um tratamento dos dados antes de serem enviados e depois de serem recebidos, adicionando a eles protocolos de verificação/segurança garantindo a linearidade na transmissão dos dados.

Command Line Interface (CLI)

A aplicação é executada integralmente no terminal e pode ser compilada através do comando **make**, gerando os devidos executáveis (reader e writer).

Instruções para execução do reader:

```
./reader [porta_de_serie] <output_name_file>
```

porta_de_serie: parâmetro obrigatório, que indica a porta de série que vai receber o ficheiro.
output_name_file: parâmetro facultativo, indica o nome que deseja salvar o ficheiro recebido

Instruções para execução do writer:

```
./writer [porta_de_serie] [name_file]
```

porta_de_serie: parâmetro obrigatório, que indica a porta de série que vai enviar o ficheiro.
output_name_file: parâmetro obrigatório, indica o nome do ficheiro que deseja enviar.

3. Estrutura do código

O código está dividido em duas grandes camadas e dois ficheiros executáveis, em que cada um manipula à sua maneira ambas as camadas com algumas funções partilhadas entre si.

Executável reader:

Link Layer:

```
llopen - Início do protocolo recebendo a trama de supervisão SET e respondendo
com a trama UA.
```

```
llread - Lê em ciclo as tramas e faz destuffing dos bytes.
```

`llclose` - Encerra o protocolo recebendo a trama DISC, retorna DISC e recebe UA.

Ainda fazem parte desta camada funções auxiliares de construção de desconstrução das tramas.

App Layer:

`read_dataPackage` - Função chamada ao receber uma trama com pacote de dados.

`read_controlPackage` - Função chamada ao receber uma trama com pacote de controle inicial ou final.

Executável writer:

Link Layer:

`llopen` - Inícia do protocolo enviando a trama de supervisão SET e aguardando confirmação através do recebimento da trama UA.

`llwrite` - Utilizada para escrever em ciclo as tramas de dados, além das tramas de controle no início e no fim.

`llclose` - Encerra o protocolo enviando a trama DISC, aguarda por DISC e confirma enviando UA.

Ainda fazem parte desta camada funções auxiliares de construção de desconstrução das tramas.

App Layer:

`create_dataPackage` - Função chamada para enviar uma trama com pacote de dados.

`create_controlPackage` - Função chamada para enviar uma trama com pacote inicial ou final.

4. Casos de uso principais

O principal caso de uso do sistema desenvolvido é a transferência de um ficheiro a partir de um computador para outro sendo um o responsável por ser o transmissor e outro o receptor através de portas de série.

Para dar início à transferência, o receptor é executado na porta de série desejada (ex: /dev/ttyS0) e com indicação facultativa do nome do arquivo a ser salvo (por omissão utiliza-se o nome do ficheiro original), assim pode-se correr o transmissor indicando como parâmetro o ficheiro a ser enviado.

A transmissão de dados segue a seguinte linha de protocolos executados na função ***main()*** dos respectivos programas (reader e writer):

1. Receptor e transmissor são iniciados através de uma porta de série, o transmissor escolhe o ficheiro a ser enviado.
2. É feita a configuração da conexão entre os dois computadores através do protocolo llopen.
3. Com a conexão estabelecida, o transmissor sempre envia os dados através do protocolo llwrite.
4. A primeira trama enviada é da Control Package, que informa ao reader o início do ficheiro.
5. Em seguida, são enviados sequencialmente as Data Package, que estas já incluem conteúdo sobre o ficheiro em si.
6. Por fim, é enviado outra Control Package a indicar o fim da transferência.
7. O receptor sempre recebe os dados através do protocolo llread.
8. Após receber todos os dados, a conexão é terminada tanto no receptor como no transmissor através do protocolo llclose.

5. Protocolo de ligação lógica

LLOPEN

```
int llopen(char * port, int flag);
```

Esta função é responsável por estabelecer a ligação entre o emissor e receptor.

O argumento **`char* port`** recebe a porta como parâmetro (ex: /dev/ttyS0) e, como a função pode ser utilizada por ambos emissor e receptor, o argumento **`int flag`** identifica se é o emissor ou receptor quem requisita a função, uma vez que o comportamento do LLOPEN difere em ambos os casos.

No emissor o fluxo de código nesta função segue o seguinte raciocínio:

- Abre-se o descriptor responsável pela porta série.
- É enviado uma trama não numerada do emissor para o receptor, por intermédio da função **send_frame_nnsp** com o campo de controlo igual a **SET.** A função **send_frame_nnsp**, por sua vez, pode ser utilizada para enviar tramas não supervisionadas e não numeradas.
- Espera-se que o receptor envie uma resposta **UA**, por meio de uma trama não numerada, lida através da função **read_frame_not_supervision,** a qual implementa uma state machine.
- Caso não haja resposta num tempo **TIMEOUT** (macro com valor default igual a 3) então o alarme será acionado e será feita uma nova tentativa para enviar a trama. Caso haja falha em **TRIES** tentativas (macro com valor default o igual a 3), então o programa será encerrado.

Analogamente, no receptor o código segue da seguinte forma :

- Abertura do descriptor.
- Espera de uma trama não supervisionada com valor **SET** lida pela função descrita anteriormente, **read_frame_not_supervision.**
- Em caso de recebimento da trama com valor **SET** envia-se uma trama com valor UA, no intuito de confirmar o receção da trama **SET**.

Vale realçar que a escrita é feita trama a trama e o recebimento delas é feita byte a byte.

LLWRITE

```
int llwrite(int fd, byte* data, int* data_length);
```

Está função é chamada pelo emissor, responsável pelo envio de tramas e byte stuffing destas.

No início do **llwrite** é instalado o alarme de timeout e a função **create_frame_i** é chamada para que, com base nos dados (**byte* data**) recebidos, construa-se uma trama de informação. Dentro da função **create_frame_i** a implementação segue a seguinte lógica:

- É feita a criação do BCC2, pela chamada da função **create_BCC2;**
- Realiza-se byte stuffing do BCC2 e campo de dados, com a função **byte_stuffing**;
- Uma vez realizado o byte stuffing a trama de informação é criado, acrescentando-se os campos que compõem a estrutura da trama (flag, endereço, …)

Com o retorno da função **create_frame_i** e criação da trama de informação, inicia-se o protocolo propriamente dito: a trama de informação é enviada e espera-se pela resposta do receptor, **read_frame_supervision.** O programa irá comportar-se de um jeito diferente dependendo da resposta do receptor:

- Caso seja recebido uma trama de informação com campo de controlo **REJ,** a última trama é reenviada imediatamente.
- Caso receba-se um campo de controlo com o **RR** correspondente a última trama enviada, desliga-se o alarme e a função **llwrite** retorna a trama lida.
- Assim como no **llopen**, caso não haja resposta por parte do receptor a trama é reenviada após um tempo **TIMEOUT** e, caso não haja respostas, **TRIES** vezes, então o transmissor é encerrado e o descriptor é fechado.

LLREAD

```
int llread(int fd, byte* data);
```

Esta função é responsável pela leitura da trama de informação e pelo byte destuffing dos dados e bcc2.

No início da função **llread** os frames são lidos pela função **read_frame_i**, a qual retorna tramas de informação sem erros no bcc1, pelo que tramas com erros neste campo são ignoradas e cabe ao **llwrite** reenviá-las após o timeout.

Após a leitura da trama de informação o campo de controlo é guardado e é feito o byte destuffing dos dados e bcc2 pela função **byte_destuffing**. Em sequência, cria-se um bcc2 com base nos campos de dados enviados pela trama de informação e compara-se este valor com o atual bcc2 recebido. Caso o bcc2 gerado e o bcc2 recebido sejam diferentes o código encontra-se perante um erro de envio de dados e, portanto, envia-se uma trama de supervisão com o **REJ** (com a função **send_frame_nnsp**) correspondente no campo de controlo. Nesta situação o llread irá fazer uma nova tentativa de leitura.

Caso a trama recebida não possua o campo de controlo correspondente ao esperado, o programa interpretará a trama recebida como uma trama repetida, por isso enviará uma trama de supervisão com campo de controlo com o **RR** (com a função **send_frame_nnsp**) correspondente para o emissor e irá repetir o processo de leitura. Por exemplo, caso espera-se que a próxima trama tenha um campo de controlo S=N(0), mas recebe-se uma com S=N(1), envia-se um RR R=N(0) ,pois estas é uma trama repetida.

Uma vez que não haja erros na trama será enviada uma mensagem com o RR correspondente e a função llread irá retornar o tamanho do campo de dados lido.

LLCLOSE

```
int llclose(int fd, int flag);
```

Assim como o **llopen**, a função **llclose** recebe o argumento **`int flag`** com o mesmo propósito descrito em **llopen** .

Para o caso o emissor tenha chamado a função, irá ser enviada uma trama não numerada para o receptor com campo de controlo igual a **DISC.** Em seguida, espera-se igualmente que o emissor envie uma trama não numerada **DISC**, onde a leitura desta é feita pela função **read_frame_not_supervision**, a qual implementa uma state machine. A partir daí temos dois cenários futuros: se o **DISC** for recebido o emissor irá enviar uma trama com campo de controlo **UA** e fechará o descriptor logo em seguida. Caso não haja resposta será feito um timeout na mesma lógica descrita nas funções anteriores.

Para o caso do receptor tenha chamado a função, será lida trama não supervisionada pela função **read_frame_not_supervision.** Em caso de sucesso enviará uma trama com campo de controlo **DISC** e aguardará o recebimento da trama com campo de controlo **UA.** O não recebimento desta última trama implica que o receptor não irá desligar.

6. Protocolo de aplicação

O protocolo de ligação é camada de mais alto nível neste código e possui quatro funções principais responsáveis por: criar o pacote de dados, ler o pacote de dados, criar o pacote de controlo e ler o este último.

create_controlPackage

```
int create_controlPackage(byte C, byte* nameFile, int length, byte* pack);
```

Esta função cria o pacote de controlo para sinalizar o início e fim da transferência de um ficheiro. Para tal o pacote de controlo que sinaliza o start (C = start) irá ter um campo com o nome do ficheiro e outro com o tamanho deste. E, assim como no início da transmissão, a sinalização do fim desta deve conter os campos contendo o nome do ficheiro e o campo com o tamanho do ficheiro. Mais tarde, estes pacotes serão enviados para o receptor por meio da função llwrite.

read_controlPackage

```
int read_controlPackage(byte pack, byte* nameFile, int* fileSize,int packSize);
```

Esta função lê os campos do pacote de controlo e retorna o nome e tamanho do ficheiro. Mais detalhes disponíveis em anexo.

read_dataPackage

```
int read_dataPackage(int seqNum, byte* info, byte* pack);
```

Esta é uma função que tem como objetivo ler o *data package.* A função apenas retorna o tamanho do pacote, definido pelos campos L1 e L2 e também armazena a informação do pacote na variável info.

create_dataPackage

```
int create_dataPackage(int seqNum, byte* info, int length, byte* frame);
```

Esta função é responsável por criar os *data packages*. A função é simples pelo que possui apenas 13 linhas nas quais atribui os valores necessários para compor a trama, no que diz respeito aos campos C, N, L2, L1 e dados, onde o valor C é atribuído como 1.

7. Validação

A fim de estudar o programa desenvolvido e a identificação de erros foram realizados os seguintes testes :

- Envio de ficheiros de tamanhos variados;
- Interrupção do ligação entre computadores durante o envio do ficheiro;
- Envio de informação num ambiente com ruído;

- Envio de ficheiros com diferentes valores de baudrate;
- Envio de ficheiros com diferentes tamanhos de trama;
- Envio de ficheiros com diferentes porcentagens de erros.

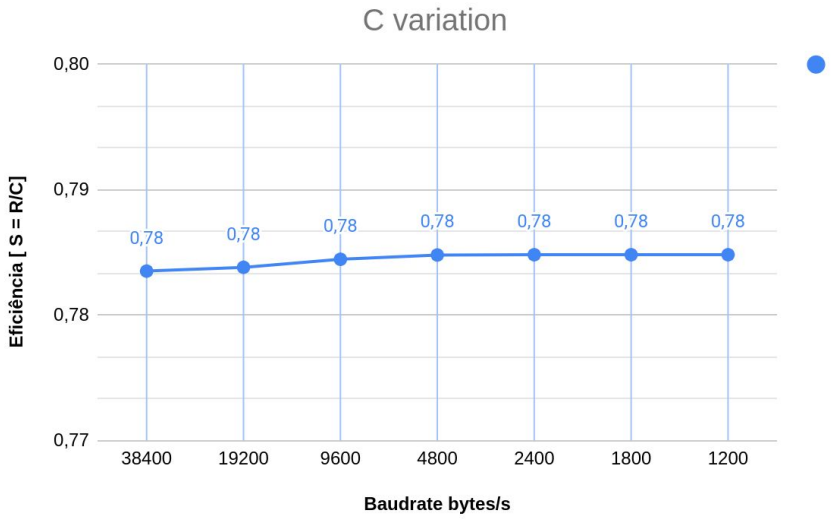
O programa passou em todos os testes com sucesso.

8. Eficiência do protocolo de ligação de dados

Para testar a eficiência do programa foram feitos 4 tipos de testes com suas respectivas tabelas de valores. Para cada teste foram feitas três ou duas amostras de tempo e o valor usado nos cálculos de eficiência foi a média destes tempos, a fim de diminuir a margem de erro dos dados. As tabelas estão disponíveis no anexo 2.

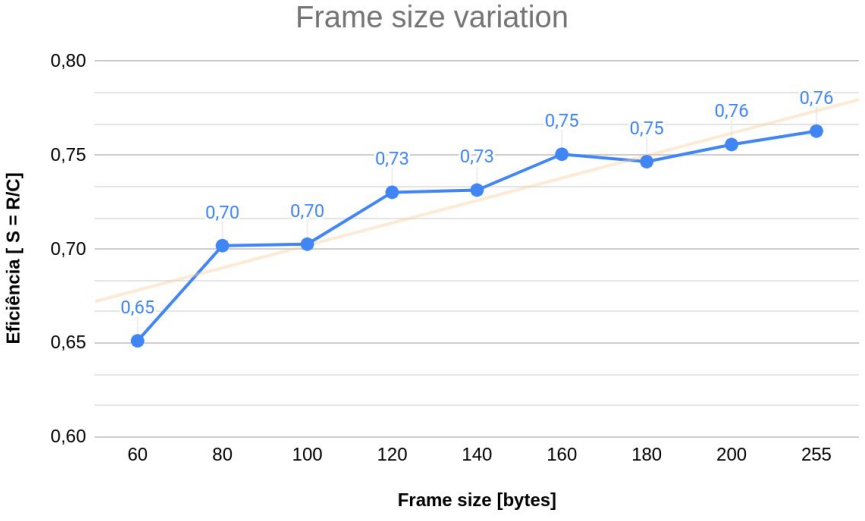
Variação da capacidade (C):

A conclusão que tiramos sobre o gráfico é que há uma melhora não significativa da performance quando o baudrate é diminuído.



Variação do tamanho das tramas de transmissão:

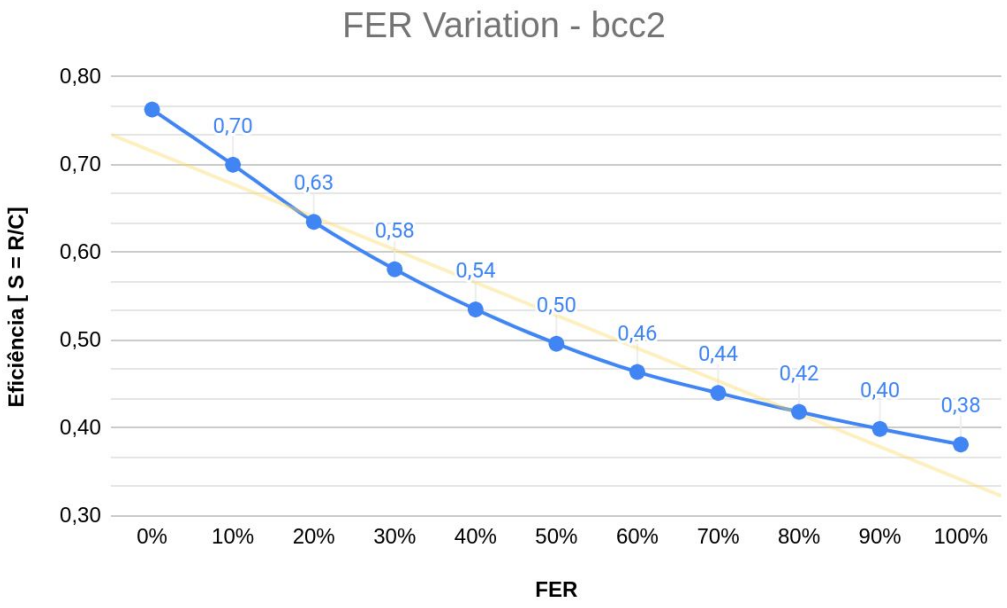
Com o aumento do tamanho das tramas de transmissão, verificamos uma melhora significativa do programa, uma vez que quanto maior o tamanho da trama, maior o tamanho do pacote. Por consequência será necessária menos preparação (e.g alocação, byte stuffing, etc) para enviar um conjunto de bytes.

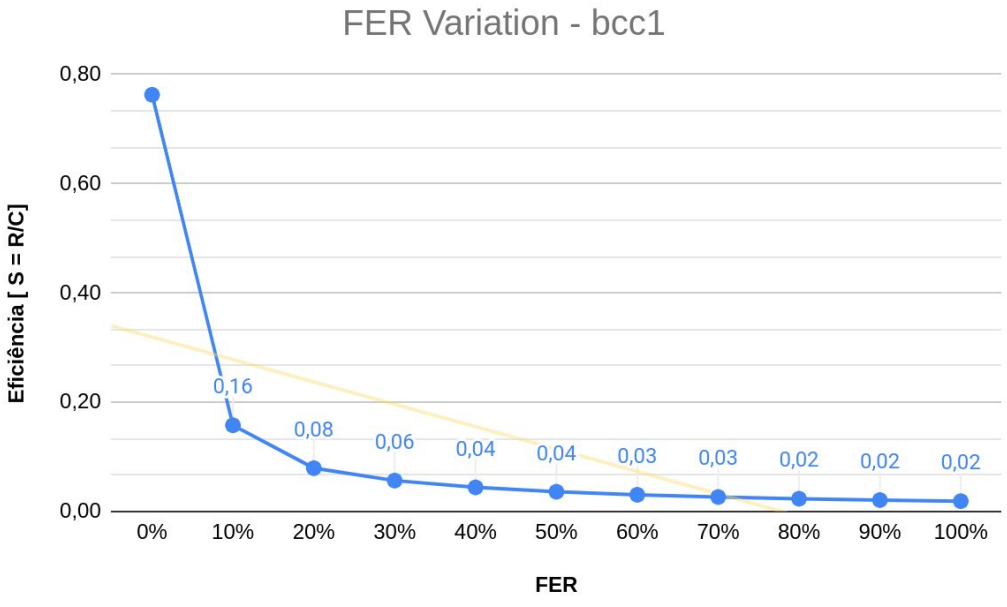


Variação do tamanho da FER

Foram feitos testes com a variação da porcentagem de erros das tramas. Se o valor da FER é igual a 50% significa 1 a cada duas tramas chega com erro no código. Foram feitos cálculos separados para a variação de erros no bcc1 e bcc2 para perceber qual a influência de cada um na performance do código.

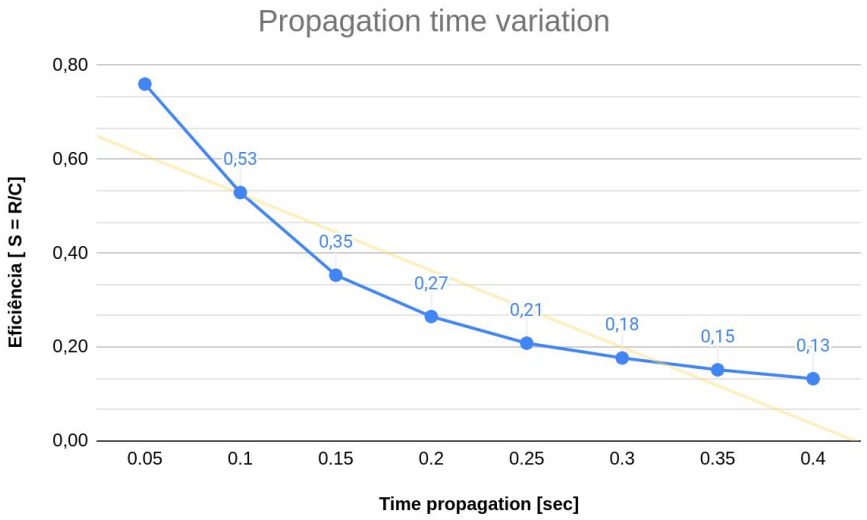
A partir do gráfico abaixo podemos concluir que o aumento de erros implica numa perda significativa de eficiência do programa. No caso da variação do bcc1 a perda de eficiência é mais significativa uma vez que é necessário esperar o timeout para reenviar a trama.





Variação do tamanho do Tempo de propagação

Para simular a variação do tempo de propagação foi posto um **usleep** antes do receptor ler a trama emitida pelo emissor. Neste cenário, pôde-se concluir que quanto maior o tempo de propagação menor a eficiência do código.



Comparação com protocolo stop & wait

Quanto ao protocolo stop & wait, podemos identificar semelhanças deste e o código implementado neste trabalho curricular.

No que diz respeito ao protocolo stop & wait, ao emissor enviar uma enviar uma trama ao receptor, espera-se que este envie uma resposta **ACK** (*acknowledgment*) ou **NACK**. Caso não hajam erros na trama recebida, se envia uma mensagem **ACK** ao emissor, caso contrário é enviada um **NACK**, o qual implica o reenvio imediato da mensagem antes recebida com erro.

No protocolo implementado em sala de aula o envio de uma trama (S, U ou I) requer que o emissor envie uma resposta ao transmissor podendo ser RR ou REJ. Por outro, lado o não recebimento de uma resposta por parte do emissor, assim como no protocolo stop & wait, também implica do **TIMEOUT** e por conseguinte no reenvio da trama. O RR desempenha as mesmas funções do **ACK**: atesta o recebimento da última trama e avisa que e receptor está pronto para uma nova. O **REJ** informa que houve um erro nos campos de dados da trama e espera que o emissor reenvie a trama imediatamente.

Vale realçar que as tramas **REJ** e **RR** devem possuir valores específicos de **Nr** a qual corresponde a !**Ns**. Assim, pode-se perceber se a trama recebida é repetida ou se é a esperada. Supondo que o **Ns** esperado seja 1 e que o emissor envie uma trama com Ns=1, espera-se que o emissor envie um RR ou REJ com Nr= !Ns = 0. Caso o Ns recebido não seja o esperado (Ns= 0), interpreta-se a trama recebida como uma repetida e esta é descartada, mas o emissor envia um RR com Ns = 0. Tal estratégia também é aplicada no protocolo stop & wait.

9. Conclusão

Por fim, o desenvolvimento primeiro trabalho prático da cadeira, o qual consistiu na implementação de um protocolo de ligação de dados, serviu para percebermos melhor o funcionamento protocolos de rede e suas complexidades.

Em termos menos gerais, foi possível transmitir informações entre dois computadores através de um cabo série de forma segura. Além disso, compreendemos a condição de independência entre camadas, onde a camada da link layer oferecia serviços a camada de aplicação, no entanto, ambas eram independentes entre si.

Em suma, todos os objetivos do trabalho foram completados e houve grande contribuição para o conhecimento teórico e prático da cadeira de redes de computadores.

Anexos

1. Código fonte

macros.h

```
define BAUDRATE B9600
#define MODEMDEVICE "/dev/ttyS1"
#define _POSIX_SOURCE       1


#define MAX_SIZE_ALLOC      16000           /* Max size for the vector.
always must be higher than (FRAME_SIZE-10)*2 */
#define FRAME_SIZE          400             /*This size must not be less
than 10*/

/* DEBUG MACROS ------------------------------------*/

#define DEBUG  1            /* Set zero to don't show prints. */
#define SHOW_OUTPUT 0      /* Show the information inside the state
machines. */
#define PRINT_ERR(format, ...) \
    do{ \
    if (DEBUG) \
    printf("\033[31;1mERR\033[0m: %s:%d\t\t::\033[31;1m" format
"\033[0m\n", __FILE__, __LINE__, ##__VA_ARGS__ ); \
    }while(0)
```

```
#define PRINT_SUC(format, ...) \
    do{ \
    if (DEBUG) \
    printf("\033[32;1mSUC\033[0m: %s:%d\t\t::\033[32;1m" format
"\033[0m\n", __FILE__, __LINE__, ##__VA_ARGS__ ); \
    }while(0)

#define PRINT_NOTE(format, ...) \
    do{ \
    if (DEBUG) \
    printf("\e[1;34mNOTE\e[0m: %s:%d\t\t::\e[1;34m" format "\e[0m\n",
__FILE__, __LINE__, ##__VA_ARGS__ ); \
    }while(0)

#define PRINTF(format, ...) \
    do{ \
    if (SHOW_OUTPUT) \
    printf("\e[1;34m"format "\e[0m", ##__VA_ARGS__ ); \
    }while(0)


/* MACROS FOR THE LLOPEN FLAG ---------------------------------*/

#define TRANSMITTER         0
#define RECEPTOR            1

/* MACROS FOR THE ALARM AND LOOPS-----------------------------*/

#define TRIES               3   /* Tries to read the receptor answers*/
//TODO: do I need this?
#define TIMEOUT             3   /* Time to wait for the receptor
answers*/
#define TRIES_READ          3   // TODO: do I need this?

/* MACROS FOR THE PROTOCOL--------------------------------------*/

#define FLAG                0x7E
#define ESC                 0x7D
#define XOR_STUFFING(N)     (N^0x20)
#define A                   0x03

/**Command Field*/

#define CMD_SET             0x03        /* SET command*/
#define CMD_DISC            0x0B        /* DISC command*/
#define CMD_UA              0x07        /* UA command*/
```

```
#define CMD_S(s)            (0x00 | ( s << 6 ))
#define CMD_RR(r)           (0x05 | ( r << 7 ))
#define CMD_REJ(r)          (0x01 | ( r << 7 ))

/* MACROS FOR THE APPLICATION ----------------------------------*/

#define CTRL_DATA           1
#define CTRL_START          2
#define CTRL_END            3

#define T_FILE_SIZE         0
#define T_FILE_NAME         1

/* OTHERS  -----------------------------------------*/
#define FALSE               0
#define TRUE                1
#define SWITCH(s)           !s
#define BIT(n)              (1 << n)
#define DELAY_US            0.2
```

writer.h

```
#ifndef WRITER_H
#define WRITER_H

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include "app_layer.h"


void open_file(byte* nameFile);

size_t get_fileSize();

#endi
```

writer.c

```
/*Non-Canonical Input Processing*/

#include "../include/writer.h"

int fd;
FILE* fp;
int seqNum = 0;

int main(int argc, char **argv) {
    int contentSize = FRAME_SIZE - 10,  frameLength = 0,
actual_contentSize = 0;

    // ALLOCATING VARIABLES
    char * content[contentSize];
    byte* pack[MAX_SIZE_ALLOC];
    byte* frame_ = (byte*)malloc(sizeof(byte)*MAX_SIZE_ALLOC);

    // CHECK USAGE
    char firstLetters[9];
    if (argc < 3){
        PRINT_ERR("Usage: /dev/ttySX path_file!");
        exit(-1);
    }else{
        strncpy(firstLetters, argv[1], 9);
        if (strcmp(firstLetters, "/dev/ttyS") != 0){
            PRINT_ERR("Usage: /dev/ttySX path_file");
            exit(-1);
        }
    }

    // OPEN FILE
    byte * namefile = argv[2];
    open_file(namefile);


    int fileSize = get_fileSize();

   // SET CHANNEL
   install_alarm();
   fd = llopen(argv[1], TRANSMITTER);


    //CONTROL PACKAGE START
    int size = create_controlPackage(CTRL_START, namefile, fileSize,
pack);
```

```
    llwrite(fd, pack, &size);

    while(TRUE){
        if (fileSize - seqNum * contentSize < contentSize ) contentSize
= fileSize%contentSize;
        PRINT_NOTE("seqNum %d", seqNum);
        if ((actual_contentSize = fread(content, 1, contentSize, fp)) <=
0) {
            break;
        }

        if (create_dataPackage(seqNum, content, actual_contentSize,
frame_) <0){
            PRINT_ERR("create_dataPackage error");
            return -1;
        }


        frameLength = actual_contentSize + 4;
        if (llwrite(fd, frame_, &frameLength) < 0) {
            PRINT_ERR("LLWRITE error");
            return -1;
        }


        seqNum++;
    }


    //CONTROL PACKAGE END
    size = create_controlPackage(CTRL_END, namefile, fileSize, pack);
    llwrite(fd, pack, &size);

    free(frame_);

   //CLOSE
   llclose(fd, TRANSMITTER);


}

void open_file( byte* nameFile){
    if (( fp = fopen(nameFile, "rb")) == NULL){
        PRINT_ERR("Inexistent %s", nameFile);
        exit(-1);
    }
```

```
}


size_t get_fileSize(){
    // Get size of file.
    if (fseek(fp, 0L, SEEK_END) != 0){
        PRINT_ERR("%s", stderr);
        exit(-1);
    }

    int fileSize = ftell(fp);

    // Pointer back to the beggining.
    if (fseek(fp, 0L, SEEK_SET) != 0){
        PRINT_ERR("%s", stderr);
        exit(-1);
    }

    return fileSize;
}
```

reader.h

```
#ifndef READER_H
#define READER_H

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "app_layer.h"


#endi
```

reader.c

```
/*Non-Canonical Input Processing*/
```

```c
#include "../include/reader.h"

int fd;
int seqNum = 0;

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

    // Information about the file.
    char * namefile = (char*)malloc(sizeof(char)*MAX_SIZE_ALLOC);
    char * outputfile = (char*)malloc(sizeof(char)*MAX_SIZE_ALLOC);
    byte *package = (byte *) malloc(MAX_SIZE_ALLOC*sizeof(byte));
    byte* info = (byte*)malloc(sizeof(byte)*MAX_SIZE_ALLOC);
    FILE *fp;
    int length;

    int filesize;
    package[0] = 4;

    u_int8_t received_start = FALSE;
    // CHECK USAGE

    if(argc == 3){
        strcpy(outputfile, argv[2]);
        if (strncmp(argv[1], "/dev/ttyS", 9) != 0){
            PRINT_ERR("Usage: /dev/ttySX <path_file>");
            exit(-1);
        }
    }else if(argc == 2){
        outputfile = namefile;
        if (strncmp(argv[1], "/dev/ttyS", 9) != 0){
            PRINT_ERR("Usage: /dev/ttySX <path_file>");
            exit(-1);
        }
    }else{
        PRINT_ERR("Usage: /dev/ttySX <path_file>");
        exit(-1);
    }

    // SET CHANNEL
    PRINT_NOTE("LLOPEN CALL");
    if( (fd = llopen(argv[1], RECEPTOR)) == -1 ) {
        PRINT_ERR("Could not open descriptor on port %s.", argv[1]);
        exit(-1);
    }

    //RECEIVE START PACKAGE
```

```
    while(received_start != TRUE){
        length = llread(fd, package);
        if (package[0] == CTRL_START){
            PRINT_SUC("Received package start");
            read_controlPackage(package, namefile, &filesize, length);
            received_start = TRUE;
        }
    }

    if( (fp = fopen(outputfile, "wb")) == NULL ) {
        PRINT_ERR("%d", errno);
        exit(-1);
    }

    //READ DATA
    while(TRUE){
        if ( (length = llread(fd,package)) < 0){
            PRINT_ERR("Could not read file descriptor.");
        }

        if (package[0] == CTRL_DATA){
            length = read_dataPackage(&seqNum, info, package);
            PRINT_NOTE("seqNum %d",seqNum);
            fwrite(info, sizeof(byte), length, fp);
        }

        memset(info, 0, strlen(info));
        if (package[0] == CTRL_END){
            byte *end_outputfile[MAX_SIZE_ALLOC];
            int end_filesize;
            read_controlPackage(package, end_outputfile, &end_filesize,
length);
            if (strcmp(end_outputfile, namefile) != 0)
                PRINT_ERR("End file name: %s :-: Begin file name: %s",
end_outputfile, namefile);
            if (filesize != end_filesize)
                PRINT_ERR("End size: %d :-: Begin size: %d",
end_filesize, filesize);

            break;
        }


    }

    printf("NAMEFILE: %s\n", namefile);
```

```
    printf("FILESIZE: %d\n", filesize);

    PRINT_NOTE("LLCLOSE CALL");
    llclose(fd, RECEPTOR);
    free(info);
    free(package);
    free(namefile);

    if( (fclose(fp)) == EOF ) {
        PRINT_ERR("%s", stderr);
        exit(-1);
    }
}
```

link_layer.h

```
#ifndef GLOBALS_H
#define GLOBALS_H

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include "macros.h"
#include "utils.h"
#include <errno.h>


typedef unsigned char byte;

/* API
---------------------------------------------------------------------*
/
/**
 * @brief Set the port configuration and establish the connecction.
 *
 * @param port String containing the path of the string.
 * @param flag The plot of the caller, i.e TRANSMITTER, RECEPTOR
 * @return int -1 in case of error, number of the file descriptor
otherwise.
 */
```

```
int llopen(char * port, int flag);

int llwrite(int fd, byte* data, int *data_length);

int llread(int fd, byte * data);

int llclose(int fd, int flag);

/* AUX API
---------------------------------------------------------------------*/

/* Read and send frames */

int read_frame_supervision(int fd, byte *CMD);

int read_frame_not_supervision(int fd, byte CMD);

int send_frame_nnsp(int fd, byte ADDR, byte CMD);

int read_frame_i(int fd, byte *buffer, byte *CMD);

/* Create information frame */

int create_frame_i(byte * data, byte * frame, int data_length, byte
CMD);

int byte_stuffing(byte * frame, int *frame_length);

int byte_destuffing(byte * frame, int * frame_length);

void create_BCC2(byte * data, byte *buffer, int data_length);


/* ALARM
---------------------------------------------------------------------*
/

void install_alarm();

void handle_alarm_timeout();

void alarm_off();

/* OTHERS
---------------------------------------------------------------------*/
```

```
int openDescriptor(byte *port, struct termios *oldtio, struct termios
*newtio);

int closeDescriptor(int fd, struct termios * oldtio);


#endif
```

link_layer.c

```
#include "../include/link_layer.h"

int numTransmissions = 0;
int fd_transmitter = 0;
struct termios oldtio_transmitter;
struct termios oldtio_receiver;


int llopen(char *port, int flag)
{
    int fd;
    int res = -1;
    struct termios newtio;

    if (flag != TRANSMITTER && flag != RECEPTOR){
        PRINT_ERR("Actual flag %d. Must be 1 or 0.", flag);
        return -1;
    }

    if (TRANSMITTER == flag) {

        // Set the file descriptor.
        fd = openDescriptor(port, &oldtio_transmitter, &newtio);
        fd_transmitter = fd;

        // Establishment of the connection.
        while (res != 0) {
            alarm(TIMEOUT);
            if (send_frame_nnsp(fd, A, CMD_SET) < 0)
                PRINT_ERR("Not possible to send CMD_SET. Sending again
after timeout...");
            else PRINT_SUC("Written CMD_SET.");

            if((res = read_frame_not_supervision(fd, CMD_UA)) >= 0)
```

```
                PRINT_SUC("Received UA.");
        }

        if (res == 0) alarm_off();
    }

    else if (RECEPTOR == flag)
    {
        //Set the file descriptor.
        fd = openDescriptor(port, &oldtio_receiver, &newtio);

        while(res < 0){
            // Establishment of the connection.
            read_frame_not_supervision(fd, CMD_SET);
            PRINT_SUC("Received CMD_SET with success.");

            if ((res = send_frame_nnsp(fd, A, CMD_UA)) < 0)
                PRINT_ERR("Error sending answer to the emissor.");
        }
    }
    return fd;
}

int llwrite(int fd, byte *data, int *data_length) {
    static int s_writer = 0;
    int frame_length;
    byte CMD;

    byte * frame  = (byte*) malloc(MAX_SIZE_ALLOC*sizeof(byte));    //
Alloc max size.
    byte * data_cpy = (byte*) malloc(MAX_SIZE_ALLOC*sizeof(byte));    //
Alloc max size.
    // Check input function errors.
    if (*data_length < 0) {
        PRINT_ERR("Length must be positive, actual: %d", *data_length);
        return -1;
    }
    memcpy(data_cpy, data, *data_length);
    // Send info.
    while(TRUE){

        // Creating the info to send.
        memcpy(data, data_cpy, *data_length);
        memset(frame, 0, strlen(frame));

        frame_length = create_frame_i(data, frame, *data_length,
```

```
CMD_S(s_writer));

        alarm(TIMEOUT);
        if (write(fd, frame, frame_length) < 0) {
            PRINT_ERR("Not possible to write info frame. Sending again
after timeout...");
            continue;
        }
        else PRINT_SUC("Sent frame with S=%d", s_writer);

        if (read_frame_supervision(fd, &CMD) < 0) {
            PRINT_ERR("Not possible to read info frame. Sending
again...");
            continue;
        }

        else PRINT_SUC("Read CMD=%02x with R=%d", CMD, !s_writer);

        if ((CMD == CMD_RR(1) && s_writer == 0) || (CMD == CMD_RR(0) &&
s_writer == 1)){
            alarm_off();
            s_writer = SWITCH(s_writer);
            free(frame);
            free(data_cpy);
            return 0;
        }

        if (CMD == CMD_REJ(!s_writer) || CMD == CMD_REJ(s_writer)){
            alarm_off();
            PRINT_ERR("Received REJ");
            continue;
        }


    }

}

int llread(int fd, byte * data){
    int data_length = -1;
    static int s_reader = 0, curr_s = 0;
    byte  check_BCC2, CMD;

    // Will not leave the loop until has a new message.
    while(TRUE){
        if ((data_length = read_frame_i(fd, data, &CMD)) < 0){
```

```
            sleep(DELAY_US);
            PRINT_NOTE("Trying to read again.");
            continue;
        }else PRINT_SUC("Received CMD_S=%02x, S=%d", CMD_S(s_reader),
s_reader);

        // Get the s received.
        if (CMD == CMD_S(0)) curr_s = 0;
        else if (CMD == CMD_S(1)) curr_s = 1;

        byte_destuffing(data, &data_length);

        // Check the bcc2.
        check_BCC2 = 0x00;
        create_BCC2(data, &check_BCC2, data_length-1);

        // If wrong bcc2 send CMD REJ.
        if (check_BCC2 != data[data_length-1]) {
            PRINT_ERR("Wrong bcc2. Expected: %02x, Received: %02x.",
check_BCC2, data[data_length-1]);
            PRINT_NOTE("Sending CMD_REJ.");
            if (CMD == CMD_S(0))
                send_frame_nnsp(fd, A, CMD_REJ(1));
            else if (CMD == CMD_S(1))
                send_frame_nnsp(fd, A, CMD_REJ(0));
            continue;
        }else {

            PRINT_SUC("Correct bcc2. Expected: %02x, Received: %02x.",
check_BCC2, data[data_length-1]);
            PRINT_SUC("BCC2 ok!");
        }


        // It's not the desired message.
        if (CMD != CMD_S(s_reader)){
            PRINT_NOTE("Undesired message s_reader %d", s_reader);
            PRINT_SUC("Sending RR %d", !curr_s);
            send_frame_nnsp(fd, A, CMD_RR(!curr_s));
            continue;       // Discard the message.
        }

        // Desired message, save the info.
        if (send_frame_nnsp(fd, A, CMD_RR(!s_reader)) > 0){
            PRINT_SUC("CMD_RR with R=%d sent.", !s_reader);
            s_reader = SWITCH(s_reader);
```

```
            return data_length;
        }

    }
    return -1;
}

int llclose(int fd, int flag){

    int res = -1;

    if (flag != TRANSMITTER && flag != RECEPTOR){
        PRINT_ERR("Invalid flag.");
        exit(-1);
    }

    if (flag == TRANSMITTER){
        while(res < 0){
            alarm(TIMEOUT);
            if ((res = send_frame_nnsp(fd, A, CMD_DISC)) < 0){
                PRINT_ERR("Failed sending CMD_DISC. Sending again...");
                sleep(DELAY_US);   /* Wait a little before sending
again.*/
            }else PRINT_SUC("Sent CMD_DISC.");

            if((res = read_frame_not_supervision(fd, CMD_DISC)) < 0){
                PRINT_ERR("Failed in receive CMD_DISC. Sending CMD_DISC
again...");
                continue;
            } else {
                PRINT_SUC("Read CMD_DISC.");
                alarm_off();
            }

            // Here doesn't matter if it was sent or not. The emissor
must turn off.
            if (send_frame_nnsp(fd, A, CMD_UA) < 0 ){
                PRINT_ERR("Failed in sending CMD_UA. Turning off...");
            }
            else{
                sleep(1);
                PRINT_SUC("Sent CMD_UA.");
            }

        }
```

```
        return closeDescriptor(fd, &oldtio_transmitter);

    }else if (flag == RECEPTOR){
        while(res < 0){
            if((res = read_frame_not_supervision(fd, CMD_DISC)) < 0){
                PRINT_ERR("Failed reading CMD_DISC");
                continue;
            } else PRINT_SUC("Read CMD_DISC");

            if ((res = send_frame_nnsp(fd, A, CMD_DISC)) < 0){
                PRINT_ERR("Failed sending CMD_DISC. Reading CMD_DISC
again...");
                continue;
            } else PRINT_SUC("Sent CMD_DISC");

            if ((res = read_frame_not_supervision(fd, CMD_UA))){
                PRINT_ERR("Not able to receive CMD_UA. Reading CMD_DISC
again...");
                continue;
            }else PRINT_SUC("Received CMD_UA");

                sleep(1);
            return closeDescriptor(fd, &oldtio_receiver);
        }
    }
    return -1;
}

int read_frame_supervision(int fd, byte *CMD){
   int curr_state = 0; /* byte that is being read. From 0 to 4.*/
    byte byte;

    PRINTF("--READ SUPERVISION FRAME [RR, REJ]--\n");
    while (TRUE)
    {
        if (read(fd, &byte, 1) == -1)
            return -1;

        switch (curr_state)
        {
        // RECEIVE FLAG
        case 0:

            PRINTF("case 0: %02x\n", byte);
            if (byte == FLAG)
                curr_state++;
```

```
            break;

        // RECEIVE ADDR
        case 1:
            PRINTF("case 1: %02x\n", byte);
            if (byte == A)
                curr_state++;
            else if (byte != FLAG)
                curr_state = 0;
            break;

        // RECEIVE CMD
        case 2:
            PRINTF("case 2: %02x\n", byte);
            if (byte == CMD_REJ(1) || byte == CMD_RR(1) || byte ==
CMD_RR(0) || byte == CMD_REJ(0)){
                *CMD = byte;
                curr_state++;
            }
            else if (byte == FLAG)
                curr_state = 1;
            else
                curr_state = 0;
            break;
        // RECEIVE BCC
        case 3:
            PRINTF("case 3: %02x\n", byte);
            if (byte == (*CMD ^ A))
                curr_state++;
            else if (byte == FLAG)
                curr_state = 1;
            else
                curr_state = 0;
            break;

        // RECEIVE FLAG
        case 4:
            PRINTF("case 4: %02x\n", byte);
            if (byte == FLAG)
                return 0;
            else
                curr_state = 0;
        }
    }
    return curr_state;
}
```

```
int read_frame_not_supervision(int fd, byte CMD)
{
    int curr_state = 0; /* byte that is being read. From 0 to 4.*/
    byte data;
    PRINTF("--READ NOT SUPERVISION FRAME [UA, DISC, SET]--\n");
    while (TRUE) {

        if (read(fd, &data, 1) == -1)
            return -1;

        switch (curr_state) {

        // RECEIVE FLAG
        case 0:

            PRINTF("case 0: %02x\n", data);
            if (data == FLAG)
                curr_state++;
            break;

        // RECEIVE ADDR
        case 1:
            PRINTF("case 1: %02x\n", data);
            if (data == A)
                curr_state++;
            else if (data != FLAG)
                curr_state = 0;
            break;

        // RECEIVE CMD
        case 2:
            PRINTF("case 2: %02x\n", data);
            if (data == CMD)
                curr_state++;
            else if (data == FLAG)
                curr_state = 1;
            else
                curr_state = 0;
            break;
        // RECEIVE BCC
        case 3:
            PRINTF("case 3: %02x\n", data);
            if (data == (CMD ^ A))
                curr_state++;
            else if (data == FLAG)
```

```
                curr_state = 1;
            else
                curr_state = 0;
            break;

        // RECEIVE FLAG
        case 4:
            PRINTF("case 4: %02x\n", data);
            if (data == FLAG) return 0;
            else curr_state = 0;
        }
    }
    return -1;
}

int send_frame_nnsp(int fd, byte ADDR, byte CMD)
{
    byte frame[5];
    frame[0] = FLAG;
    frame[1] = ADDR;
    frame[2] = CMD;
    frame[3] = frame[1] ^ frame[2];
    frame[4] = FLAG;

    return write(fd, frame, 5);
}

int read_frame_i(int fd, byte *buffer, byte *CMD){
    int curr_state= 0, info_length = -1;
    byte byte;

    PRINTF("--READ FRAME I--\n");
    while(curr_state < 5){
        if (read(fd, &byte, 1) == -1)
            return -1;

        switch (curr_state)
        {
            // RECEIVE FLAG
            case 0:
                info_length = 0;
                PRINTF("case 0: %02x\n", byte);
                if (FLAG == byte)
                    curr_state ++;
                break;
            // RECEIVE ADDR
```

```
            case 1:
                PRINTF("case 1: %02x\n", byte);
                if (A == byte)
                    curr_state ++;
                else if (FLAG != byte)
                    curr_state = 0;

                break;

            // RECEIVE CMD
            case 2:
                PRINTF("case 2: %02x\n", byte);
                if (byte == CMD_S(0) || byte == CMD_S(1)){
                    *CMD = byte;
                    curr_state++;
                }
                else if (byte == FLAG)
                    curr_state = 1;
                else curr_state = 0;

                break;

            // RECEIVE BCC1
            case 3:
                PRINTF("case 3: %02x\n", byte);
                if (byte == (*CMD ^ A))
                    curr_state ++;
                else if (byte == FLAG)
                    curr_state = 1;
                else
                    curr_state = 0;
                break;
            // RECEIVE INFO
            case 4:
                PRINTF("case 4: %02x\n", byte);
                if (byte != FLAG){
                    buffer[info_length++] = byte;
                }
                else curr_state ++;

        }
    }
    return info_length;
}

int create_frame_i(byte *data, byte *frame, int data_length, byte CMD)
```

```
{
    int frame_length, bcc_length = 1;

    // Stuffing bcc2 and data.
    byte *BCC2 = (byte*)malloc(sizeof(byte));
    BCC2[0] = 0x00;

    create_BCC2(data, BCC2, data_length);
    PRINT_NOTE("BCC %02x", *BCC2);
    byte_stuffing(data, &data_length);
    byte_stuffing(BCC2, &bcc_length);

    // Store information
    frame_length = 5  + bcc_length + data_length;
    frame[0] = FLAG;
    frame[1] = A;
    frame[2] = CMD;
    frame[3] = frame[1]^frame[2];
    // BCC
    memcpy(&frame[4], data, data_length);
    memcpy(&frame[4 + data_length], BCC2, bcc_length);

    frame[frame_length-1] = FLAG;

    free(BCC2);
    PRINT_NOTE("Created frame i");
    return frame_length;
}

void create_BCC2(byte * data, byte* buffer, int data_length)
{
    for (int i = 0 ; i < data_length; i++){
        *buffer ^= data[i];
    }
}

int byte_stuffing(byte * frame, int* frame_length)
{
    byte * new_frame;
    int extra_space = 0;        /* The extra space needed to be added to
the frame. */
    int new_frame_length;   /* The new length of the string frame (extra
+ length). */
    int actual_pos;         /* Position in the new_frame. */
    int counter = 0;            /* Number of escapes and flags found in
the second iteration. */
```

```
    //  First find all the flags and scapes to avoid multiple reallocs.
    for (int i = 0 ; i < *frame_length; i++)
        if (frame[i] == FLAG || frame[i] == ESC) extra_space++;


    new_frame_length = extra_space + *frame_length;
    new_frame = (byte *)malloc(sizeof(byte) * new_frame_length);

    for (int i = 0 ; i< *frame_length; i++){
        actual_pos = i + counter;
        if (frame[i] == FLAG){
            new_frame[actual_pos] =  ESC;
            new_frame[actual_pos+1] = XOR_STUFFING(FLAG);

            counter ++;
        }
        else if (frame[i] == ESC){
            new_frame[actual_pos] = ESC;
            new_frame[actual_pos+1] = XOR_STUFFING(ESC);

            counter ++;
        }
        else new_frame[actual_pos] = frame[i];
    }


    //frame = realloc(frame, new_frame_length);
    * frame_length  = new_frame_length;

    memcpy(frame, new_frame, new_frame_length);
    free(new_frame);
    return 0;
}

int byte_destuffing(byte * frame, int * frame_length){

    int new_frame_pos = 0;
    byte * new_frame = (byte*)malloc(sizeof(byte)*(*frame_length));

    for (int i = 0 ; i < *frame_length; i++){
        if (frame[i] == ESC){
            if (frame[i+1] == XOR_STUFFING(FLAG))
                new_frame[new_frame_pos] = FLAG;
            else if (frame[i+1] == XOR_STUFFING(ESC))
                new_frame[new_frame_pos] = ESC;
```

```
            i++;
        }
        else new_frame[new_frame_pos] = frame[i];
        new_frame_pos ++;
    }


    memcpy(frame, new_frame, new_frame_pos);
    *frame_length = new_frame_pos;
    free(new_frame);
    return 0;
}

int openDescriptor(byte *port, struct termios *oldtio, struct termios
*newtio)
{
    int fd;
    if((fd = open(port, O_RDWR | O_NOCTTY)) < 0) {
        PRINT_ERR("Invalid port: %s", port);
        exit(-1);
    }

    if (tcgetattr(fd, oldtio) == -1) {
        PRINT_ERR("tcgetattr.");
        exit(-1);
    }

    bzero(newtio, sizeof(newtio));
    newtio->c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
    newtio->c_iflag = IGNPAR;
    newtio->c_oflag = 0;

    /* set input mode (non-canonical, no echo,...) */
    newtio->c_lflag = 0;

    newtio->c_cc[VTIME] = 0; /* inter-character timer unused */
    newtio->c_cc[VMIN] = 1;  /* blocking read until 1 chars received */

    tcflush(fd, TCIOFLUSH);

    if (tcsetattr(fd, TCSANOW, newtio) == -1) {
        PRINT_ERR("tcsetattr");
        exit(-1);
    }
```

```
    PRINT_SUC("Function openDescriptor executed with success.");

    return fd;
}

int closeDescriptor(int fd, struct termios * oldtio){
    PRINT_NOTE("Closing descriptor");
    if (tcsetattr(fd, TCSANOW, oldtio) == -1) {
        perror("tcsetattr");
        exit(-1);
    }
    PRINT_SUC("Restoured oldtio");
    return close(fd);
}

void install_alarm() {
    if (signal(SIGALRM, handle_alarm_timeout) == SIG_ERR)
    {
        PRINT_ERR("Not possible to install signal, SIG_ERR.");
        llclose(fd_transmitter, CMD_DISC);
    }
    siginterrupt(SIGALRM, TRUE);
}


void handle_alarm_timeout() {
    numTransmissions++;

    PRINT_ERR("Time out #%d", numTransmissions);

    if (numTransmissions > TRIES)
    {
        PRINT_ERR("Number of tries exceeded\n");
        closeDescriptor(fd_transmitter, &oldtio_transmitter);
        exit(-1);
    }
}


void alarm_off() {
    numTransmissions = 0;
    alarm(0);
}
```

app_layer.h

```
#include "link_layer.h"


int create_dataPackage(int seqNum, byte* info, int length, byte* pack);

int read_dataPackage(int* seqNum, byte* info, byte* pack);

int create_controlPackage(byte C, byte* nameFile, int length, byte*
pack);

int read_controlPackage(byte* pack, byte* nameFile, int *fileSize, int
packSize);
```

app_layer.c

```
#include "../include/app_layer.h"

int create_dataPackage(int seqNum, byte* info, int length, byte* frame){

    frame[0] = CTRL_DATA;
    frame[1] = seqNum % 256;
    frame[2] = length / 256; // L2
    frame[3] = length % 256;

    if (memcpy(&frame[4], info, length) == NULL){
        PRINT_ERR("Error while copying package.");
        return -1;
    }
    PRINT_SUC("Created data package");
    return 0;
}

int read_dataPackage(int* seqNum, byte* info, byte* pack){

    *seqNum = pack[1];
    int infoSize = pack[2]*256 + pack[3];

    if (memcpy(info, &pack[4], infoSize) == NULL){
        PRINT_ERR("Error parsing info.");
        return -1;
    }
```

```
    return infoSize;
}

int create_controlPackage(byte C, byte* nameFile, int length, byte*
pack){
    int size_nameFile = strlen(nameFile), curr_pos = 0;

    pack[0]= C;
    pack[1] = T_FILE_NAME;
    pack[2]= strlen(nameFile);

    if (memcpy(&pack[3] , nameFile, size_nameFile) == NULL){
        PRINT_ERR("Not possible to copy file name");
        return -1;
    }

    curr_pos = 3 + size_nameFile;
    char * length_string = (char*)malloc(sizeof(int));
    sprintf(length_string, "%d", length);                        // Int
to string.

    pack[curr_pos] = T_FILE_SIZE;
    pack[curr_pos+1] = strlen(length_string);

    if (memcpy(&pack[curr_pos+2], length_string, strlen(length_string))
== NULL){
        PRINT_ERR("Not possible to copy size of file");
        return -1;
    }

    PRINT_SUC("Created control package.");
    return curr_pos + strlen(length_string) + 2;
}

int read_controlPackage(byte* pack, byte* nameFile, int *fileSize, int
packSize){
    char * fileSize_string = (char*)malloc(sizeof(int));

    for (int i = 0 ; i < packSize; i++){
        if (T_FILE_NAME == pack[i]){
            i++;
            int sizeT = pack[i++];
            if (memcpy(nameFile, &pack[i], sizeT) == NULL){
                PRINT_ERR("Not possible to parse file name");
                return -1;
            }
```

```
            i+= sizeT-1;
        }
        else if (T_FILE_SIZE == pack[i]){
            i++;
            int sizeT = pack[i++];
            if (memcpy(fileSize_string, &pack[i], sizeT) == NULL){
                PRINT_ERR("Not possible to parse size of file");
                return -1;
            }

            sscanf(fileSize_string, "%d", fileSize);        // Parse the
size
            i+= sizeT;
        }

    }
    PRINT_SUC("Read control package.");
    free(fileSize_string);
    return 0;
}
```

utils.h

```
#include  <stdio.h>
#include <stdlib.h>


void print_hex(char *s, int length);
```

utils.c

```
#include "../include/utils.h"

void print_hex(char *s, int length) {
    for (int i = 0 ; i < length; i++ )
        printf("%02x\n", s[i]);
    printf("\n");
}
```

2. Tabelas para medição de eficiência

Variação da capacidade (C):

Frame Size	Total frames sent
255,00	45,00

C variation				
Time 1	**Time 2**	**Time 3**	**Baudrate**	**Efficiency**
3,05	3,05	3,05	38400	0,78
6,10	6,10	6,10	19200	0,78
12,19	12,19	12,19	9600	0,78
24,37	24,37	24,37	4800	0,78
48,73627	48,736223	48,736272	2400	0,78
64,98168	64,98168	64,98168	1800	0,78
97,47249	97,472461	97,47249	1200	0,78

Variação do tamanho da trama I:

Trama size variation					
Time 1	**Time 2**	**Baudrate**	**Efficiency**	**Frame Size**	**Num packets**
3,133895	3,133372	38400	0,76	255	45
3,197569	3,1979	38400	0,76	200	58
3,264702	3,26459	38400	0,75	180	65
3,286776	3,286159	38400	0,75	160	74
3,38887	3,390161	38400	0,73	140	85
3,423648	3,424009	38400	0,73	120	100
3,61634	3,616862	38400	0,70	100	122
3,725837	3,731804	38400	0,70	80	157
4,222523	4,224036	38400	0,65	60	220

Variação da FER BCC2

Frame Size	Total frames sent
255,00	45,00

FER - BCC2				
Time 1	**Time 2**	**Baudrate**	**Efficiency**	**FER**
3,133895	3,133372	38400	0,76	0%
3,415749	3,414871	38400	0,70	10%
3,764624	3,76574	38400	0,63	20%
4,114841	4,115505	38400	0,58	30%
4,467477	4,46611	38400	0,54	40%
4,817926	4,818865	38400	0,50	50%
5,151713	5,15224	38400	0,46	60%
5,431908	5,432315	38400	0,44	70%
5,710929	5,712676	38400	0,42	80%
5,98947	5,990685	38400	0,40	90%
6,267678	6,268375	38400	0,38	100%

Variação da FER BCC1

Frame Size	Total frames sent
255,00	45,00

FER - BCC1				
Time 1	**Time 2**	**Baudrate**	**Efficiency**	**FER**
3,133895	3,133372	38400	0,76	0%
15,144352	15,143363	38400	0,16	10%
30,144078	30,144186	38400	0,08	20%
42,45264	42,144275	38400	0,06	30%
54,145151	54,144959	38400	0,04	40%
66,145447	66,145697	38400	0,04	50%
78,145763	78,145273	38400	0,03	60%
90,146671	90,145118	38400	0,03	70%
102,146434	102,146588	38400	0,02	80%
114,146987	114,146595	38400	0,02	90%

126,146713	126,146671	38400	0,02	100%

Variação do tempo de propagação

Frame Size	Total frames sent
255,00	45,00

Propagation time - T_prop				
Time 1	**Time 2**	**Baudrate**	**Efficiency**	**T_prop**
3,143725	3,145858	38400	0,76	0,05
4,518211	4,518257	38400	0,53	0,1
6,772458	6,768078	38400	0,35	0,15
9,018209	9,019122	38400	0,27	0,2
11,68101	11,268089	38400	0,21	0,25
13,518107	13,518354	38400	0,18	0,3
15,76858	15,768235	38400	0,15	0,35
18,018069	18,018113	38400	0,13	0,4