Redes de Computadores

1º Trabalho Prático
Protocolo de ligação de dados

Guilherme Almeida - 202008866
Tiago Barbosa - 202004926

November 27, 2022

Índice

1 Sumário

Este trabalho foi realizado no âmbito da Unidade Curricular Redes de Computadores do curso de Engenharia Informática e Computação da FEUP. Neste primeiro trabalho laboratorial o objetivo era construir uma aplicação capaz de realizar transferência de ficheiros entre dois computadores através da porta série.

Todos os objetivos do trabalho foram alcançados, visto que foi possível utilizar esta aplicação na transferência de vários ficheiros sem qualquer perda de dados.

2 Introdução

O principal objetivo deste trabalho foi a criação de um protocolo de ligação de dados de acordo com um guião previamente fornecido para a transferência de ficheiros entre dois computadores através de uma porta série.

O relatório tem como propósito descrever a realização deste trabalho de forma a mostrar as estratégias adotadas, escolhas feitas e peripécias encontradas assim como fornecer uma revisão geral acerca da eficiência do protocolo. O relatório está assim dividido nas seguintes secções :

- **Arquitetura**

  Exibição dos blocos funcionais e interfaces.

- **Estrutura do código**

  Apresentação das APIs, principais estruturas de dados, principais funções e a sua relação com a arquitetura.

- **Casos de usos principais**

  Identificação destes e apresentação das sequências de chamada de funções.

- **Protocolo de ligação lógica**

  Identificação dos principais aspetos funcionais e descrição da estratégia de implementação destes aspetos com pequenos extratos de código.

- **Protocolo de aplicação**

  Identificação dos principais aspetos funcionais e descrição da estratégia de implementação destes aspetos com pequenos extratos de código.

- **Validação**

  Descrição dos testes efetuados com apresentação quantificada dos resultados.

- **Eficiência do protocolo de ligação de dados**

  Caraterização estatística da eficiência do protocolo recorrendo a medidas sobre o código desenvolvido.

- **Conclusões**

  Síntese da informação apresentada nas secções anteriores com reflexão sobre os objetivos de aprendizagem alcançados.

3 Arquitetura

O trabalho, apesar de não conter em termos de ficheiros diferenças para o emissor e o recetor, dentro da camada de aplicação estes papéis estão devidamente isolados para que seja possível separar a lógica de cada um. Assim sendo, na camada de aplicação existe o código apenas para o emissor e o código apenas para o emissor separados por uma condição verificada no princípio.

4 Estrutura do código

O código está assim dividido em duas partes : a camada de ligação lógica e a camada da aplicação. No ficheiro link_layer.c está representado o protocolo da ligação lógica com as seguintes funções :

- **llopen** - estabelece a ligação entre os dois computadores usando os parâmetros de conexão passados pela camada de aplicação. Esta função é a única da camada de ligação lógica que é chamada pelo emissor e recetor pelo que dentro dela, o código de cada papel está separado. Assim no emissor, a função envia o SET e espera receber o UA enquanto que no recetor lê a trama SET e envia a trama UA criando assim a ligação entre os dois PC's.
- **llwrite** - escreve para a porta série as tramas I enviadas pela camada de aplicação após realizar o "stuffing" das mesmas e espera receber uma confirmação do recetor na forma de uma trama RR ou uma não-confirmação na forma de REJ. Esta função é apenas chamada pelo emissor.
- **llread** - lê da porta série as tramas I enviadas pelo emissor e após fazer "destuffing" se a trama não contiver erros passa os dados para a camada de aplicação e envia uma confirmação para o emissor na forma de RR ou no caso de erro na trama I envia uma não-confirmação na forma de REJ. A função se ler uma trama DISC, enviada pelo emissor a indicar o fim da conexão, transmite uma nova trama DISC para o emissor e espera receber uma trama UA para confirmar o fim da ligação dos dois PC's terminando o programa. Esta função é apenas chamada pelo recetor.
- **llclose** - envia uma trama DISC para o recetor para indicar o terminar da ligação entre os dois PC's e fica a espera para ler outra trama DISC e enviar uma trama UA final para acabar o programa e a ligação. Esta função é apenas chamada pelo emissor.

Neste ficheiro estão definidas as seguintes estruturas de dados:

- **LinkLayerRole** typedef enum {LlTx,LlRx}
- **LinkLayer** typedef struct {char serialPort[50]; LinkLayerRole role;int baudRate;int nRetransmissions;int timeout;}

No ficheiro application_layer.c está representado o protocolo da camada de aplicação que está dividido na parte do emissor e recetor. Assim foram criados dois ficheiros : trasnmitter.c e receiver.c que contém as funções para cada um dos papéis.

No ficheiro trasnmitter.c está a seguinte função principal :

- **transmitter** - abre o ficheiro para trasnmitir e depois de construir pacotes de controlo com o nome e tamanho do ficheiro a enviar, lê o ficheiro aos bocados e envia pacotes para a camada de ligação lógica com os conteúdos do ficheiro usando a função llwrite.

No ficheiro receiver.c está a seguinte função principal :

- **receiver** - utiliza a funcção llread para ler pacotes enviados pelo emissor e depois faz um parse destes pacotes onde se estes forem de controlo cria o ficheiro novo e se forem de dados escreve esses dados para o novo ficheiro.

5 Casos de usos principais

Os principais casos de uso da aplicação são:

- **Interface** - permite ao utilizador escolher o seu role assim como definir as especificações do protocolo de ligação, selecionando o ficheiro a enviar/receber, o número de tentativas antes de abortar a conexão e o tempo em segundos antes de dar timeout.
  Para dar início ao programa, o utilizador pode correr o comando com os argumentos habituais, porta série, role e ficheiro, ficando o número de tentativas e o tempo de timeout com os valores default de 3. Correndo o comando sem argumentos o utilizador aciona assim a função **menu** situada no main.c, que lida com a parte da interface relacionada com as definições do protocolo de ligação, utilizando as funções **getInputString** e **getInputInt** para lidar com o input do utilizador. Por último a função **menu** utiliza **applicationLayer** para iniciar o protocolo, passando as especificações nos parâmetros dessa função.

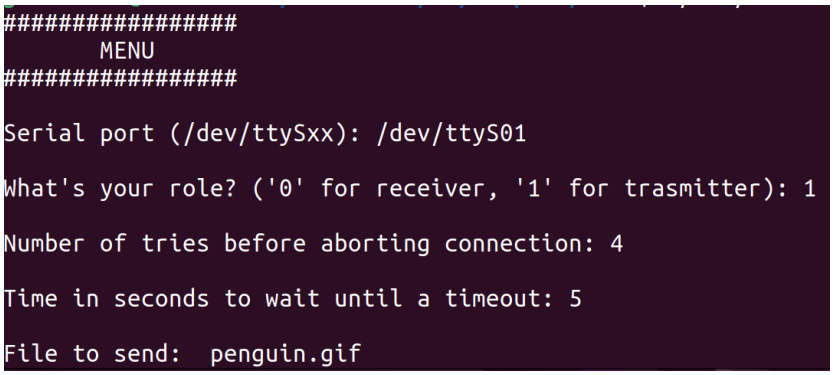

Figure 1: Menu Interface

- **Transmissão de Dados** - permite o utilizador enviar um ficheiro entre dois computadores, transmissor e recetor, via porta série. Neste caso de uso existe uma distinção nas funções usadas entre o transmissor e recetor.
  Em ambos os casos as funções **llopen** e **llclose** são utilizadas para estabelecer e terminar a ligação respetivamente. De seguida, no caso do transmissor a função **transmitter** é chamada e são utilizadas as funções **sendControlPacket**, **sendDataPacket** e **llwrite** para o envio dos dados. Do lado do recetor, a função **receiver** é utilizada para gerir a receção dos dados, utilizando as funções **llread**, **readPacket** e **readControlPacket** que guardam os dados num novo ficheiro.

6 Protocolo de ligação lógica

O protocolo de ligação lógica (link layer) foi uma das camadas implementadas no projeto e da qual o protocolo de aplicação (application layer) depende. O **link layer** é responsável por estabelecer e terminar a ligação através da porta série; o envio e receção das tramas de informação; realizar stuff e destuff de pacotes da application layer.

Os principais aspetos funcionais são então as funções **llopen**, **llwrite**, **llread**, **llclose** e por fim **state_machine**.

- **llopen** - Como descrito na secção 4, a função **llopen** é responsável por estabelecer a ligação entre o transmissor e o recetor.

  Quando esta função é invocada pelo emissor, o SET é enviado e a resposta UA do recetor é aguardada. Se esta não chegar dentro de um tempo timeout, com a ajuda de um alarme, o SET é reenviado. Este mecanismo de retransmissão é repetido até o número de tentativas máximo for ultrapassado, caso em que o programa termina. Quando o recetor invoca a função, aguarda uma trama de controlo SET para poder responder com um UA. Em ambos os casos as tramas são enviadas utilizando a função **sendFrame**.

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

Figure 2: Função llopen

```
int sendFrame(int fd, unsigned char C, unsigned char BCC)
{
    unsigned char FRAME[5];

    FRAME[0] = FLAG;
    FRAME[1] = A;
    FRAME[2] = C;
    FRAME[3] = BCC;
    FRAME[4] = FLAG;

    return write(fd, FRAME, 5);
}
```

Figure 3: Função sendFrame

- **llwrite** - A função **llwrite** - é chamada apenas pelo transmissor e é responsável pelo envio das tramas e pelo stuffing das mesmas. Primeiramente é criado o framing da mensagem, onde é calculado o BCC utilizando a função **createBCC**. De seguida é feito o stuffing com o auxílio da função **byte_stuff**. Após estes processo é possível enviar a trama de informação através da função **sendInformationFrame**. Caso a resposta seja RR, a mensagem foi transmitida corretamente, caso seja o comando REJ, a mensagem não foi corretamente transmitida, ocorrendo uma retransmissão. O envio desta trama tem o mesmo mecanismo timeout e retransmissão descrito na função llopen anterior.

```
void createBCC(const unsigned char *src, unsigned char *newBuff, int bufSize)
{
    unsigned char BCC2 = 0;
    for (int i = 0; i < bufSize; i++)
    {
        newBuff[i] = src[i];
        BCC2 ^= src[i];
    }
    newBuff[bufSize] = BCC2;
```

Figure 4: Função createBCC

```c
int byte_stuffing(unsigned char *buf, int bufSize)
{
    int newBufSize = 0;
    unsigned char newBuff[bufSize * 2];
    for (int i = 0; i < bufSize; i++)
    {
        if (buf[i] == 0x7E)
        {
            newBuff[newBufSize] = 0X7D;
            newBufSize++;
            newBuff[newBufSize] = 0X5E;
            newBufSize++;
        }
        else if (buf[i] == 0x7D)
        {
            newBuff[newBufSize] = 0X7D;
            newBufSize++;
            newBuff[newBufSize] = 0X5D;
            newBufSize++;
        }
        else
        {
            newBuff[newBufSize] = buf[i];
            newBufSize++;
        }
    }
    memcpy(buf, newBuff, newBufSize);
    return newBufSize;
}
```

Figure 5: Função byte_stuff

```c
int sendInformationFrame(int fd, unsigned char C, unsigned char BCC, const unsigned char *buf, int bufSize)
{

    unsigned char FRAME[PACKET_SIZE * 2];
    int buf_cnt = 4;

    FRAME[0] = FLAG;
    FRAME[1] = A;
    FRAME[2] = C;
    FRAME[3] = BCC;

    for (int i = 0; i < bufSize; i++)
    {
        FRAME[buf_cnt] = buf[i];
        buf_cnt++;
    }

    FRAME[buf_cnt] = FLAG;

    return write(fd, FRAME, buf_cnt + 2);
}
```

Figure 6: Função sendInformationFrame

- **llread** - Esta função é utilizada apenas pelo recetor e é responsável pela receção das tramas e pelo destuffing das mesmas. Inicialmente realiza o destuffing da trama utilizando a função **byte_destuffing**, se a trama não tiver erros é enviado o RR, caso contrário o REJ. Ao ler uma trama DISC, significa que a ligação deve ser terminada e envia uma nova trama DISC para o emissor, esperando a receção de uma UA. As tramas são enviadas utilizando a função **sendFrame** mencionada previamente nesta secção.

```
int byte_destuffing(unsigned char *buf, int bufSize)
{
    int newBufSize = 0;
    unsigned char newBuff[bufSize * 2];
    for (int i = 0; i < bufSize; i++)
    {
        if (buf[i] == 0x7D)
        {
            if (buf[i + 1] == 0x5E)
            {
                newBuff[newBufSize] = 0X7E;
                newBufSize++;
                i++;
            }
            else if (buf[i + 1] == 0x5D)
            {
                newBuff[newBufSize] = 0X7D;
                newBufSize++;
                i++;
            }
        }
        else
        {
            newBuff[newBufSize] = buf[i];
            newBufSize++;
        }
    }
    memcpy(buf, newBuff, newBufSize);
    return newBufSize;
}
```

Figure 7: Função byte_destuffing

- **llclose** - Esta função é utilizada apenas pelo transmissor e tem é responsável por terminar a ligação através da porta série. É enviada a trama DISC utilizando a função **sendFrame**, aguarda pela receção do comando DISC e finalmente envia o comando UA. Ficando assim a ligação entre o emissor e o recetor efetivamente terminada.
- **state_machine** - As leituras de qualquer trama são feitas através de uma única máquina de estados, que recebe byte a byte a mensagem e executa as mudanças de estado, de modo a chegar ao estado final apenas se a trama recebida for válida. A função **state_machine** tem como parâmetros o byte de informação do buffer, o estado **state** atual, o data buffer onde se vai guardar a informação e o byte onde se vai escrever no data buffer. Estes dois últimos parâmetros são utilizados apenas pelo recetor. Com esta estratégia foi possível criar apenas uma máquina de estados de modo a ter um código mais limpo e eficiente.

```c
typedef enum
{
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    C_SET_RCV,
    C_RR_RCV,
    C_UA_RCV,
    C_REJ_RCV,
    C_DISC_RCV,
    C_DATA_RCV,
    C_DATA_RPT_RCV,
    BCC_OK,
    BCC_SET_OK,
    BCC_UA_OK,
    BCC_DISC_OK,
    BCC_DATA_OK,
    BCC_DATA_RPT_OK,
    BCC_RR_OK,
    BCC_REJ_OK,
    DATA_RCV,
    DATA_RPT_RCV,
    STOP_DISC,
    STOP_S,
    STOP_ALARM,
    STOP_SET,
    STOP_UA,
    STOP_RR,
    STOP_REJ,
    STOP_DATA,
    STOP_DATA_RPT
} STATE;

STATE state_machine(unsigned char buf ,STATE state, unsigned char * data,int* count_data);
```

Figure 8: Ficheiro statemachine.h

7 Protocolo de aplicação

A application layer é a camada de mais alto nível implementada neste projeto e é reponsável pelo envio/receção dos pacotes de controlo/dados e o envio do ficheiro especificado. Esta camada está dividida na parte de emissor e recetor, deste modo a função **applicationLayer** chama as funções **transmitter** e **receiver** dependendo do role do utilizador. Podemos então dividir as funcionalidades do protocolo de aplicação:

- **Transmissor**

  A função **transmitter** é a função principal na parte do transmissor na protocolo de aplicação. Primeira é aberto o ficheiro a enviar e é obtido o tamanho do mesmo. De seguida é construido e enviado o pacote de controlo utilizando a função **sendControlPacket** com o seu segundo argumento com o valor 2 - START. Depois é feita a fragmentação do ficheiro em pequenos pacotes, cujo tamanho está definido na variável PACKET_SIZE, de seguida é feito o envio por ordem de cada pacote de dados invocando a função **sendDataPacket**. Por fim é utilizada novamente a função para o envio do pacoto de controlo, mas desta vez o seu segundo argumento tem o valor de 3 - END. É de notar que ambas as funções **sendControlPacket** e **sendDataPacket** utilizam a função **llwrite** para finalizar o envio para a camada de ligação lógica.

```c
int sendControlPacket(int fd, unsigned char ctrl_field, unsigned file_size, const char *filename)
{
    unsigned L1 = sizeof(file_size);
    unsigned L2 = strlen(filename);
    unsigned packet_size = 5 + L1 + L2;

    unsigned char packet[packet_size];
    packet[0] = ctrl_field;
    packet[1] = 0;
    packet[2] = L1;
    memcpy(&packet[3], &file_size, L1);
    packet[3 + L1] = 1;
    packet[4 + L1] = L2;
    memcpy(&packet[5 + L1], filename, L2);

    return llwrite(fd, packet, packet_size);
}
```

Figure 9: Função sendControlPacket

```c
int sendDataPacket(int fd, int sequenceNr , int size, unsigned char * buffer){
    unsigned char packet[size];
    packet[0] = 1;
    packet[1] = sequenceNr % 255;
    packet[2] = size / 256;
    packet[3] = size % 256;
    memcpy(&packet[4], buffer, size);

    return llwrite(fd,packet,size + 4);
}
```

Figure 10: Função sendDataPacket

- **Recetor**

  A principal função na parte do recetor no protocolo de aplicação é a função **receiver**. É utilizada a função **llread** para ler os pacotes enviados pelo emissor. Para processar estes pacotes é utilizada a função **readPacket**, caso os pacotes sejam de controlo é invocada a função **readControlPacket** e um ficheiro novo é criado como nome e tamanho registados no pacote. Caso os pacotes sejam de dados, a sua ordem é confirmada com a ajuda da função **checkSequenceNr** e a sua informação é escrita no ficheiro criado de modo a juntar os fragmentos recebidos. Por fim o tamanho do ficheiro criado é comparado ao tamanho do ficheiro enviado através da função **checkFileSize**.

```c
int receiver(int fd);

int readPacket(unsigned char * packet);

int readControlPacket(unsigned char * packet, char* filename, int* filesize);

int checkSequenceNr(int number);
```

Figure 11: Ficheiro receiver.h

8 Validação

De forma a validarmos o nosso protocolo e garantir o seu bom funcionamento realizámos os seguintes testes :

- Transferência de ficheiros com vários tamanhos : 11KB , 20KB e 200KB
- Interrupção da ligação da porta série durante a transferência dos ficheiros
- Envio de ficheiros com ruído do lado do emissor e/ou ruído do lado do recetor
- Envio de ficheiros com diferentes probabilidades de erro nas tramas (FER) : 2%, 5%, 10% e 15%
- Envio de ficheiros com diferentes tempos de propagação do sinal : 50ms, 100ms, 150ms e 250ms
- Envio de ficheiros com diferentes capacidades de ligação (baudrate) : 4800bit/s, 9600bit/s, 19200bit/s, 38400bit/s e 57600bit/s.
- Envio de ficheiros com diferentes tamanhos dos pacotes de dados : 250B, 500B, 1000B, 2500B e 5000B

Todos estes testes foram concluídos com sucessor pelo o que podemos concluir que o nosso protocolo de transferência de ficheiros está totalmente operacional e sem qualquer falha.

9 Eficiência do protocolo de ligação de dados

De forma a avaliar a eficiência do nosso protocolo foram realizados 4 testes a diferentes variáveis. Estes testes foram realizados no nosso protocolo 5 vezes e foi realizado depois a média de eficiência de forma a diminuir o desvio dos dados. A cada um dos testes foi depois comparado esta eficiência com a esperada de um protocolo Stop&Wait : $\frac{Tf*(1-FER)}{Tf+2Tprop}$

9.1 Variação do FER

A geração de erros nas tramas irá afetar muito a eficiência do protocolo visto que os erros no BCC1 faz com que o recetor não responda levando a um timeout do lado emissor ficando o programa parado durante o tempo do timeout. Já no caso de um erro no BCC2 , este não terá tanto efeito visto que apenas levará a um reenvio da trama. Assim a eficiência revelou-se ser tal como esperada num protocolo Stop&Wait já que quanto maior o FER menor será a eficiência.

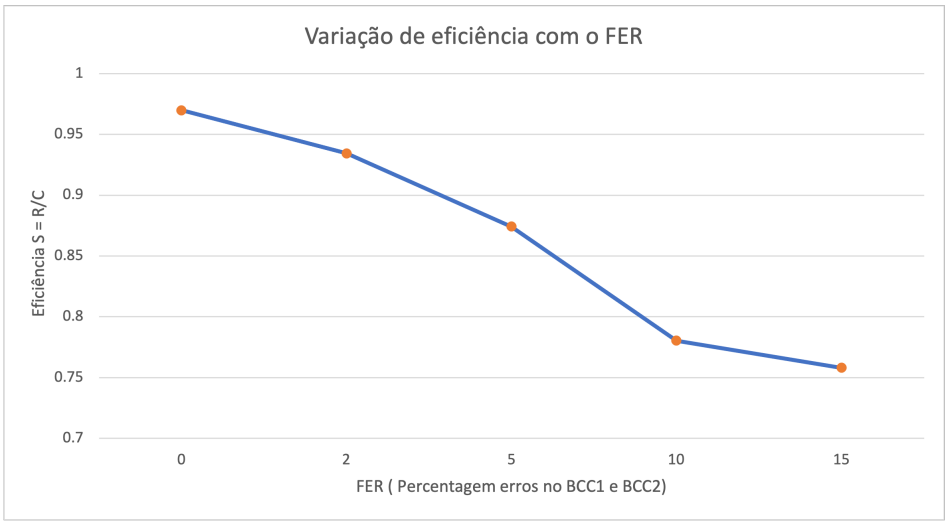

Figure 12: Variação da eficiência com o FER

9.2 Variação do T_PROP

Como seria de esperar o aumento do tempo de propagação das tramas diminui a eficiência do nosso protocolo visto que enquanto as tramas estão a propagar-se o protocolo está parado à espera de as receber. Assim tal como era esperado num protocolo Stop&Wait o aumento do tempo de propagação diminui bastante a eficiência visto que o tempo de propagação tem associado a si um fator de 2 já que as tramas têm que percorrer 2 vezes a porta série , uma do emissor para o recetor e outra para as respostas do recetor para o emissor.

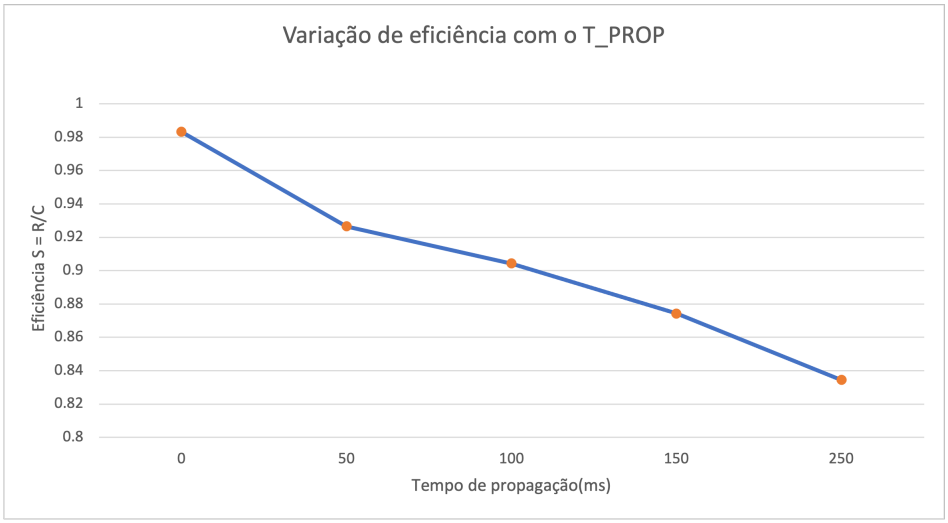

Figure 13: Variação da eficiência com o tempo de propagação

9.3 Variação da capacidade de ligação

O aumento da capacidade de ligação diminui a eficiência do nosso protocolo mas de uma forma muito subtil. Isto pode ser explicado pelo facto de se existir uma capacidade de ligação maior é mais provável um erro nas tramas levando a uma menor eficiência.

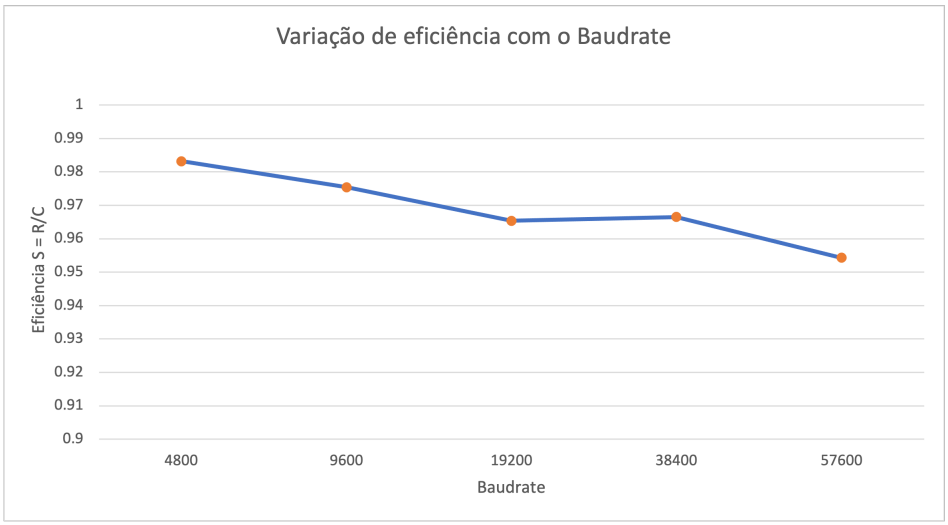

Figure 14: Variação da eficiência com a capacidade de ligação

9.4 Variação do tamanho das tramas I

Como seria de esperar o aumento do tamanho das tramas I aumenta a eficiência do nosso protocolo visto que serão precisas menos tramas I para transferir o ficheiro todo logo o protocolo será muito mais rápido e por consequência mais eficiente.

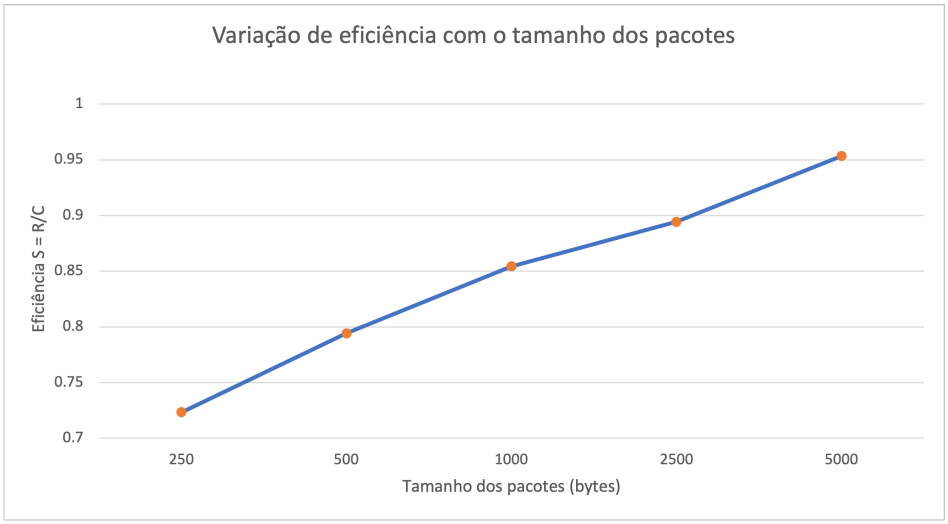

Figure 15: Variação da eficiência com o tamanho das tramas I

10 Conclusões

Este trabalho focou-se então na criação de um protocolo fiável de transferência de ficheiros entre dois PC's através da porta série. Permitiu-nos assim perceber como encapsular os dados de cada ficheiro em pacotes para envio e receção de informação. Também nos deu insights em termos como stuffing e destuffing das tramas de dados e na criação de uma máquina de estados que permitia receber os dados sequencialmente.

De tudo é importante salientar o termo **Independência entre camadas** onde a camada de ligação lógica não recorre a qualquer dado processado na camada de aplicação assim como esta não sabe a implementação feita pela camada de ligação lógica apenas podendo usar as funções disponibilizadas por esta camada.

Side note: O número máximo de páginas permitido no enunciado do relatório foi excedido ligeiramente devido ao tamanho e posicionamento das imagens inseridas. Por vezes era também necessário realizar page breaks de modo a não desformatar o conteúdo. Deste modo, em termos de texto este relatório não excede as 8 páginas máximas permitidas.

11 Anexo

main.c

```
// Main file of the serial port project.
// NOTE: This file must not be changed.

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

#include "application_layer.h"
#include "utils.h"

#define N_TRIES 3
#define TIMEOUT 4
#define BAUD_RATE B38400

// Arguments:
//   $1: /dev/ttySxx
//   $2: tx | rx
//   $3: filename

void menu()
{
    printf("###################\n");
    printf("       MENU       \n");
    printf("###################\n\n");


    char* port;
    int role;
    int ntries;
    int timeout;
    char* filename;

    printf("Serial port (/dev/ttySxx): ");
    port = getInputString();


    printf("\nWhat's your role? ('0' for receiver, '1' for trasmitter): ");
    role = getInputInt(0, 1);

    printf("\nNumber of tries before aborting connection: ");
    ntries = getInputInt(1, 15);

    printf("\nTime in seconds to wait until a timeout: ");
    timeout = getInputInt(1, 15);

    if (role == 0) {
        printf("\nName for the received file: ");
        filename = getInputString();

        applicationLayer(port, "rx", BAUD_RATE, ntries, timeout, filename);
    }

    else {
        printf("\nFile to send:  ");
```

```
        filename = getInputString();

        applicationLayer(port, "tx", BAUD_RATE, ntries, timeout, filename);
    }

}

int main(int argc, char *argv[])
{
    if (argc != 1 && argc != 4) {
        printf("Usage: %s /dev/ttySxx tx|rx filename\n\n", argv[0]);
        printf("Usage for menu: %s \n", argv[0]);
        exit(1);
    }

    if (argc == 1) {
        menu();
    }
    else if (argc == 4) {
        const char *serialPort = argv[1];
        const char *role = argv[2];
        const char *filename = argv[3];
        applicationLayer(serialPort, role, BAUD_RATE, N_TRIES, TIMEOUT, filename);
    }


    return 0;
}
```

application_layer.c

```
// Application layer protocol implementation
#include "application_layer.h"
#include <sys/time.h>

#define _POSIX_SOURCE 1 // POSIX compliant source
#define FALSE 0
#define TRUE 1

LinkLayer connectionParameters;

int fd;
int finish = FALSE;
char *FileName;

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

                      int nTries, int timeout, const char *filename)

{
    printf("Starting link-layer protocol application\n"
                    "  - Serial port: %s\n"
                    "  - Role:  %s\n"
                    "  - Baudrate: %d\n"
                    "  - Number of tries: %d\n"
                    "  - Timeout: %d\n"
                    "  - Filename: %s\n",
                    serialPort,
                    role,
                    baudRate,
                    nTries,
                    timeout,
                    filename);

    sleep(2);

    struct timeval r_start, t_start, r_end, t_end;
    FileName = malloc(sizeof(filename));

    strcpy(FileName, filename);
    strcpy(connectionParameters.serialPort, serialPort);
    if (strcmp(role, "tx") == 0)
    {
        connectionParameters.role = LlTx;
    }
    else if (strcmp(role, "rx") == 0)
    {
        connectionParameters.role = LlRx;
    }
    connectionParameters.baudRate = baudRate;
    connectionParameters.nRetransmissions = nTries;
    connectionParameters.timeout = timeout;

    fd = llopen(connectionParameters);

    if (connectionParameters.role)
```

```
    { // if recetor
        gettimeofday(&r_start, NULL);

        while (!finish)
        {
            receiver(fd);
        }

        gettimeofday(&r_end, NULL);
        double time_spent = (r_end.tv_sec - r_start.tv_sec) * 1e6;
        time_spent = (time_spent + (r_end.tv_usec - r_start.tv_usec)) * 1e-6;
        printf("Time spent: %f seconds\n", time_spent);
        printf("\nFINISHING PROGRAM \n");
    }

    if (!connectionParameters.role)
    { // if emissor

        gettimeofday(&t_start, NULL);

        transmitter(fd, filename);
        llclose(fd);

        gettimeofday(&t_end, NULL);
        double time_spent = (t_end.tv_sec - t_start.tv_sec) * 1e6;
        time_spent = (time_spent + (t_end.tv_usec - t_start.tv_usec)) * 1e-6;
        printf("Time spent: %f seconds\n", time_spent);
    }
}
```

link_layer.c

```
// Link layer protocol implementation

#include "link_layer.h"

#define NO_DATA nodata
#define NO_COUNT 0

volatile int STOP = FALSE;

extern int finish;

STATE state = START;

int alarmEnabled = FALSE;
int alarmCount = 0;
int ns = 0;
int nr = 1;
int RRsent = 0;
int REJsent = 0;
int RRreceived = 0;
int REJreceived = 0;

unsigned char nodata[0];

struct termios oldtio;
struct termios newtio;

int baudRate;
int nTries;
int timeout;

// Alarm function handler
void alarmHandler(int signal)
{
    alarmEnabled = FALSE;
    alarmCount++;
    state = STOP_ALARM;

    printf("Alarm #%d\n", alarmCount);
}

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

////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters)
{
    state = START;
    int UA_RCV = FALSE;
    // Set alarm function handler
    (void)signal(SIGALRM, alarmHandler);

    int fd = open(connectionParameters.serialPort, O_RDWR | O_NOCTTY);
```

```
if (fd < 0)
{
    perror(connectionParameters.serialPort);
    exit(-1);
}

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

timeout = connectionParameters.timeout;
baudRate = connectionParameters.baudRate;
nTries = connectionParameters.nRetransmissions;

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

newtio.c_cflag = baudRate | CS8 | CLOCAL | CREAD;
newtio.c_iflag = IGNPAR;
newtio.c_oflag = 0;

newtio.c_lflag = 0;
newtio.c_cc[VTIME] = 0;
newtio.c_cc[VMIN] = 0;

tcflush(fd, TCIOFLUSH);

if (tcsetattr(fd, TCSANOW, &newtio) == -1)
{
    perror("tcsetattr");
    exit(-1);
}
printf("New termios structure set\n");


if (connectionParameters.role) // if recetor
{
    unsigned char buf[1] = {0};

    int bytes;
    while (state != STOP_ALARM && state != STOP_SET)
    {
        bytes = read(fd, buf, 1);
        // printf("%d\n", buf[0]);
        if (bytes > 0)
        {
            state = state_machine(buf[0], state, NO_DATA, NO_COUNT);
        }
    }

    //printf("Success SET received\n");
    sendFrame(fd, C_UA, BCC_UA);
```

```
        //printf("Sent UA -> %d bytes written\n", bytes);
    }

    if (!connectionParameters.role) // if emissor
    {
        unsigned char buf[1] = {0};

        while (!UA_RCV)
        {
            if (alarmCount >= nTries)
            {
                printf("Maximum tries reached. Ending program\n");
                return -1;
            }
            state = START;

            sendFrame(fd, C_SET, BCC_SET);
            //printf("Sent SET -> %d bytes written\n", bytes);

            // Wait until all bytes have been written to the serial port
            sleep(1);

            alarm(timeout);
            alarmEnabled = TRUE;

            while (state != STOP_ALARM && state != STOP_UA)
            {
                int bytes = read(fd, buf, 1);
                if (bytes > 0)
                {
                    state = state_machine(buf[0], state, NO_DATA, NO_COUNT);
                }
            }

            if (state == STOP_UA)
            {
                UA_RCV = TRUE;
                //printf("Success UA received\n");
            }
        }
    }
    return fd;
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(int fd, const unsigned char *buf, int bufSize)
{
    unsigned char buffer[1] = {0}; // +1: Save space for the final '\0' char
    int RR_RCV = FALSE;
    alarmCount = 0;
    unsigned char newBuff[PACKET_SIZE * 2] = {0};

    createBCC(buf, newBuff, bufSize);
    int size = byte_stuffing(newBuff, bufSize + 1);
```

```
    while (!RR_RCV)
    {
        if (alarmCount >= nTries)
        {
            printf("Maximum tries reached. Ending program\n");
            return -1;
        }
        state = START;
        //printf("size: %d\n",size);
        sendInformationFrame(fd, ns << 6, A ^ (ns << 6), newBuff, size);
        //printf("Sent I -> %d bytes written\n", bytes);

        sleep(1);

        alarm(timeout);
        alarmEnabled = TRUE;

        while (state != STOP_ALARM && state != STOP_RR && state != STOP_REJ)
        {
            int bytes = read(fd, buffer, 1);
            if (bytes > 0)
            {
                state = state_machine(buffer[0], state, NO_DATA, NO_COUNT);
            }
        }

        if (state == STOP_RR)
        {
            RR_RCV = TRUE;
            ns ^= 1;
            RRreceived++;
            //printf("Success RR received\n");
        }
        else if (state == STOP_REJ)
        {
            REJreceived++;
            //printf("Success REJ received\n");
        }
    }

    return 0;
}

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
int llread(int fd, unsigned char *packet)
{
    state = START;
    int UA_RCV = FALSE;
    alarmCount = 0;
    unsigned char buf[1] = {0};
    unsigned char data[PACKET_SIZE * 2] = {0};
    int count_data = 0;
```

```
int bytes;

while (state != STOP_ALARM && state != STOP_SET && state != STOP_DISC && state != STO
{
    bytes = read(fd, buf, 1);
    if (bytes > 0)
    {
        state = state_machine(buf[0], state, data, &count_data);
    }
}

int size = byte_destuffing(data, count_data);

if (state == STOP_SET)
{
    //printf("Success SET received\n");
    sendFrame(fd, C_UA, BCC_UA);
    //printf("Sent UA -> %d bytes written\n", bytes);
}
else if (state == STOP_DISC)
{
    //printf("Success DISC received\n");

    while (!UA_RCV)
    {
        if (alarmCount >= nTries)
        {
            printf("Maximum tries reached. Ending program\n");
            finish = TRUE;
            return -1;
        }
        state = START;
        sendFrame(fd, C_DISC, BCC_DISC);
        //printf("Sent DISC -> %d bytes written\n", bytes);
        sleep(1);

        alarm(timeout);
        alarmEnabled = TRUE;

        while (state != STOP_ALARM && state != STOP_UA)
        {
            bytes = read(fd, buf, 1);
            if (bytes > 0)
            {
                state = state_machine(buf[0], state, NO_DATA, NO_COUNT);
            }
        }

        if (state == STOP_UA)
        {
            UA_RCV = TRUE;
            //printf("Success UA received\n");
            finish = TRUE;

            if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
            {
```

```
                perror("tcsetattr");
                exit(-1);
            }

            close(fd);
            printStatistics(0,RRsent,REJsent);
            return 0;
        }
    }
}
else if (state == STOP_DATA)
{
    //printf("Success DATA received\n");
    unsigned char BCC2 = 0;

    for (int i = 0; i < size - 1; i++)
    {
        BCC2 = BCC2 ^ data[i];
    }

    if (BCC2 == data[size - 1])
    {
        //printf("BCC2 ok!\n");
        for (int i = 0; i < size - 1; i++)
        {
            packet[i] = data[i];
        }
        sendFrame(fd, C_RR ^ (nr << 7), A ^ (C_RR ^ (nr << 7)));
        nr ^= 1;
        RRsent++;
        //printf("sent RR -> %d bytes written\n", bytes);
        sleep(1);
    }
    else
    {
        //printf("BCC2 FAILED!\n");
        sendFrame(fd, C_REJ ^ (nr << 7), A ^ (C_REJ ^ (nr << 7)));
        REJsent++;
        //printf("sent REJ -> %d bytes written\n", bytes);
        sleep(1);
    }
}
else if (state == STOP_DATA_RPT)
{
    //printf("Success DATA_RPT received\n");
    unsigned char BCC2 = 0;

    for (int i = 0; i < size - 1; i++)
    {
        BCC2 = BCC2 ^ data[i];
    }

    if (BCC2 == data[size - 1])
    {
        //printf("BCC2 ok!\n");
        sendFrame(fd, C_RR ^ ((nr ^ 1) << 7), A ^ (C_RR ^ ((nr ^ 1) << 7)));
```

```
            //printf("sent RR -> %d bytes written\n", bytes);
            RRsent++;
            sleep(1);
        }
        else
        {
            //printf("BCC2 FAILED!\n");
            sendFrame(fd, C_RR ^ ((nr ^ 1) << 7), A ^ (C_RR ^ ((nr ^ 1) << 7)));
            //printf("sent RR -> %d bytes written\n", bytes);
            RRsent++;
            sleep(1);
        }
    }

    return 0;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int fd)
{
    unsigned char buf[1] = {0};
    alarmCount = 0;
    int DISC_RCV = FALSE;
    int bytes;
    alarmCount = 0;

    while (!DISC_RCV)
    {
        if (alarmCount >= nTries)
        {
            printf("Maximum tries reached. Ending program\n");
            return -1;
        }
        state = START;

        sendFrame(fd, C_DISC, BCC_DISC);
        //printf("Sent DISC -> %d bytes written\n", bytes);

        // Wait until all bytes have been written to the serial port
        sleep(1);

        alarm(timeout);
        alarmEnabled = TRUE;

        while (state != STOP_ALARM && state != STOP_DISC)
        {
            bytes = read(fd, buf, 1);
            if (bytes > 0)
            {
                state = state_machine(buf[0], state, NO_DATA, NO_COUNT);
            }
        }

        if (state == STOP_DISC)
```

```
            {
                //printf("success DISC received\n");
                DISC_RCV = TRUE;
                sendFrame(fd, C_UA, BCC_UA);
                //printf("Sent UA -> %d bytes written\n", bytes);
                sleep(1);
            }
        }

        printStatistics(1,RRreceived,REJreceived);

        return 0;
}
```

receiver.c

```
#include "receiver.h"
#include "utils.h"

int receiver(int fd)
{
    unsigned char packet[PACKET_SIZE] = {0};
    llread(fd, packet);
    readPacket(packet);
    return 0;
}

int readPacket(unsigned char *packet)
{
    static int newFile;
    static int filesize = 0;
    static int currentsize = 0;
    char filename[30];

    if (packet[0] == 2)
    {
        readControlPacket(packet, filename, &filesize);
        if ((newFile = open(filename, O_WRONLY | O_CREAT, 0777)) < 0)
        {
            printf("Error opening new file!");
            return -1;
        }
    }
    else if (packet[0] == 3)
    {
        if (close(newFile) < 0)
        {
            printf("Error closing file!");
            return -1;
        }
        extern char *FileName;
        if (checkFileSize(filesize, FileName) != 0)
        {
            printf("File size different from expected\n");
            return -1;
        }
        else
            printf("File size expected!\n");
    }
    else if (packet[0] == 1)
    {
        if (checkSequenceNr(packet[1]) != 0)
        {
            printf("Error in sequence number\n");
            return -1;
        }
        int datasize = packet[3] + 256 * packet[2];
        currentsize += datasize;
        printProgressBar(currentsize,filesize);
        if (write(newFile, &packet[4], datasize) < 0)
        {
```

```
            printf("Error writing to file!");
            return -1;
        }
    }

    return 0;
}

int readControlPacket(unsigned char *packet, char *filename, int *filesize)
{
    unsigned L1 = packet[2];
    if (packet[1] == 0)
    {
        for (int i = 0; i < L1; i++)
        {
            *filesize |= (packet[3 + i] << (i * 8));
        }
    }
    if (packet[L1 + 3] == 1)
    {
        unsigned L2 = packet[L1 + 4];
        char name[L2 + 1];
        for (int i = 0; i < L2; i++)
        {
            name[i] = packet[L1 + 5 + i];
        }
        name[L2] = '\0';
        strcpy(filename, name);
    }

    return 0;
}

int checkSequenceNr(int number)
{
    static int sequenceNr = 0;

    if (number == sequenceNr)
    {
        sequenceNr = (sequenceNr + 1) % 255;
        return 0;
    }

    return -1;
}
```

transmitter.c

```
#include "transmitter.h"
#include "utils.h"


int transmitter(int fd, const char *filename)
{
    struct stat file_stat;
    int file_fd;

    // Reads file info using stat
    if (stat(filename, &file_stat) < 0)
    {
        printf("Error getting file size.");
    }

    // Opens file to transmit
    if ((file_fd = open(filename, O_RDONLY)) < 0)
    {
        printf("Error opening file.");
    }


    sendControlPacket(fd, 2, file_stat.st_size, filename);

    unsigned char buffer[PACKET_SIZE];
    int bytes;
    int sequenceNr = 0;
    float writtenBytes = 0;

    while ((bytes = read(file_fd, buffer, PACKET_SIZE-4)) > 0){
        sendDataPacket(fd,sequenceNr,bytes,buffer);

        writtenBytes += bytes;
        printProgressBar(writtenBytes, file_stat.st_size);
        sequenceNr++;
    }

    sendControlPacket(fd, 3, file_stat.st_size, filename);

    if (close(file_fd) < 0)
    {
        printf("Error closing file. ");
        return -1;
    }

    return 0;
}

int sendControlPacket(int fd, unsigned char ctrl_field, unsigned file_size, const char *
{
    unsigned L1 = sizeof(file_size);
    unsigned L2 = strlen(filename);
    unsigned packet_size = 5 + L1 + L2;
```

```
    unsigned char packet[packet_size];
    packet[0] = ctrl_field;
    packet[1] = 0;
    packet[2] = L1;
    memcpy(&packet[3], &file_size, L1);
    packet[3 + L1] = 1;
    packet[4 + L1] = L2;
    memcpy(&packet[5 + L1], filename, L2);

    return llwrite(fd, packet, packet_size);
}

int sendDataPacket(int fd, int sequenceNr , int size, unsigned char * buffer){
    unsigned char packet[size];
    packet[0] = 1;
    packet[1] = sequenceNr % 255;
    packet[2] = size / 256;
    packet[3] = size % 256;
    memcpy(&packet[4], buffer, size);

    return llwrite(fd,packet,size + 4);
}
```

state_machine.c

```
#include "state_machine.h"

extern int ns;
extern int nr;

STATE state_machine(unsigned char buf, STATE state, unsigned char *data, int *count_data
{

    switch (state)
    {
    case START:
        if (buf == FLAG)
        {
            state = FLAG_RCV;
        }
        break;

    case FLAG_RCV:
        if (buf == A)
        {
            state = A_RCV;
        }
        else if (buf == FLAG)
        {
            state = FLAG_RCV;
        }
        else
        {
            state = START;
        }

        break;

    case A_RCV:
        if (buf == C_SET)
        {
            state = C_SET_RCV;
        }
        else if (buf == C_UA)
        {
            state = C_UA_RCV;
        }
        else if (buf == (C_RR ^ ((ns ^ 1) << 7)))
        {
            state = C_RR_RCV;
        }
        else if (buf == (C_REJ ^ ((ns ^ 1) << 7)))
        {
            state = C_REJ_RCV;
        }
        else if (buf == ((nr ^ 1) << 6))
        {
            state = C_DATA_RCV;
        }
        else if (buf == (nr << 6))
```

```
        {
            state = C_DATA_RPT_RCV;
        }
        else if (buf == C_DISC)
        {
            state = C_DISC_RCV;
        }
        else if (buf == FLAG)
        {
            state = FLAG_RCV;
        }
        else
        {
            state = START;
        }
        break;

    case C_SET_RCV:
        if (buf == (BCC_SET))
        {
            state = BCC_SET_OK;
        }
        else if (buf == FLAG)
        {
            state = FLAG_RCV;
        }
        else
        {
            state = START;
        }

        break;

    case C_UA_RCV:
        if (buf == (BCC_UA))
        {
            state = BCC_UA_OK;
        }
        else if (buf == FLAG)
        {
            state = FLAG_RCV;
        }
        else
        {
            state = START;
        }

        break;

    case C_RR_RCV:
        if (buf == (C_RR ^ ((ns ^ 1) << 7) ^ A))
        {
            state = BCC_RR_OK;
        }
        else if (buf == FLAG)
        {
```

```
            state = FLAG_RCV;
        }
        else
        {
            state = START;
        }

        break;

    case C_REJ_RCV:
        if (buf == (C_REJ ^ ((ns ^ 1) << 7) ^ A))
        {
            state = BCC_REJ_OK;
        }
        else if (buf == FLAG)
        {
            state = FLAG_RCV;
        }
        else
        {
            state = START;
        }

        break;

    case C_DATA_RCV:
        if (buf == (A ^ ((nr ^ 1) << 6)))
        {
            state = BCC_DATA_OK;
        }
        else if (buf == FLAG)
        {
            state = FLAG_RCV;
        }
        else
        {
            state = START;
        }

        break;

    case C_DATA_RPT_RCV:
        if (buf == (A ^ (nr << 6)))
        {
            state = BCC_DATA_RPT_OK;
        }
        else if (buf == FLAG)
        {
            state = FLAG_RCV;
        }
        else
        {
            state = START;
        }

        break;
```

```
case C_DISC_RCV:
    if (buf == (BCC_DISC))
    {
        state = BCC_DISC_OK;
    }
    else if (buf == FLAG)
    {
        state = FLAG_RCV;
    }
    else
    {
        state = START;
    }

    break;

case BCC_SET_OK:
    if (buf == FLAG)
    {
        state = STOP_SET;
    }
    else
    {
        state = START;
    }
    break;

case BCC_UA_OK:
    if (buf == FLAG)
    {
        state = STOP_UA;
    }
    else
    {
        state = START;
    }
    break;

case BCC_RR_OK:
    if (buf == FLAG)
    {
        state = STOP_RR;
    }
    else
    {
        state = START;
    }
    break;

case BCC_REJ_OK:
    if (buf == FLAG)
    {
        state = STOP_REJ;
    }
    else
```

```
        {
            state = START;
        }
        break;

    case BCC_DATA_OK:
        if (buf == FLAG)
        {
            state = FLAG_RCV;
        }
        else
        {
            data[*count_data] = buf;
            (*count_data)++;
            state = DATA_RCV;
        }

        break;

    case BCC_DATA_RPT_OK:
        if (buf == FLAG)
        {
            state = FLAG_RCV;
        }
        else
        {
            data[*count_data] = buf;
            (*count_data)++;
            state = DATA_RPT_RCV;
        }

        break;

    case BCC_DISC_OK:
        if (buf == FLAG)
        {
            state = STOP_DISC;
        }
        else
        {
            state = START;
        }

        break;

    case DATA_RCV:
        if (buf == FLAG)
        {
            state = STOP_DATA;
        }
        else
        {
            data[*count_data] = buf;
            (*count_data)++;
        }
        break;
```

```
    case DATA_RPT_RCV:
        if (buf == FLAG)
        {
            state = STOP_DATA_RPT;
        }
        else
        {
            data[*count_data] = buf;
            (*count_data)++;
        }
        break;

    default:
        break;
    }
    return state;
}
```

utils.c

```
#include <sys/stat.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "utils.h"
#include "application_layer.h"

void createBCC(const unsigned char *src, unsigned char *newBuff, int bufSize)
{
    unsigned char BCC2 = 0;
    for (int i = 0; i < bufSize; i++)
    {
        newBuff[i] = src[i];
        BCC2 ^= src[i];
    }
    newBuff[bufSize] = BCC2;
}

int byte_stuffing(unsigned char *buf, int bufSize)
{
    int newBufSize = 0;
    unsigned char newBuff[bufSize * 2];
    for (int i = 0; i < bufSize; i++)
    {
        if (buf[i] == 0x7E)
        {
            newBuff[newBufSize] = 0X7D;
            newBufSize++;
            newBuff[newBufSize] = 0X5E;
            newBufSize++;
        }
        else if (buf[i] == 0x7D)
        {
            newBuff[newBufSize] = 0X7D;
            newBufSize++;
            newBuff[newBufSize] = 0X5D;
            newBufSize++;
        }
        else
        {
            newBuff[newBufSize] = buf[i];
            newBufSize++;
        }
    }
    memcpy(buf, newBuff, newBufSize);
    return newBufSize;
}

int byte_destuffing(unsigned char *buf, int bufSize)
{
    int newBufSize = 0;
    unsigned char newBuff[bufSize * 2];
    for (int i = 0; i < bufSize; i++)
    {
        if (buf[i] == 0x7D)
```

```
        {
            if (buf[i + 1] == 0x5E)
            {
                newBuff[newBufSize] = 0X7E;
                newBufSize++;
                i++;
            }
            else if (buf[i + 1] == 0x5D)
            {
                newBuff[newBufSize] = 0X7D;
                newBufSize++;
                i++;
            }
        }
        else
        {
            newBuff[newBufSize] = buf[i];
            newBufSize++;
        }
    }
    memcpy(buf, newBuff, newBufSize);
    return newBufSize;
}

char *getFilename(char *path)
{
    char *filename = path, *p;
    for (p = path; *p; p++)
    {
        if (*p == '/' || *p == '\\' || *p == ':')
        {
            filename = p;
        }
    }
    return filename;
}

int sendFrame(int fd, unsigned char C, unsigned char BCC)
{
    unsigned char FRAME[5];

    FRAME[0] = FLAG;
    FRAME[1] = A;
    FRAME[2] = C;
    FRAME[3] = BCC;
    FRAME[4] = FLAG;

    return write(fd, FRAME, 5);
}

int sendInformationFrame(int fd, unsigned char C, unsigned char BCC, const unsigned char
{

    unsigned char FRAME[PACKET_SIZE * 2];
    int buf_cnt = 4;
```

```
    FRAME[0] = FLAG;
    FRAME[1] = A;
    FRAME[2] = C;
    FRAME[3] = BCC;

    for (int i = 0; i < bufSize; i++)
    {
        FRAME[buf_cnt] = buf[i];
        buf_cnt++;
    }

    FRAME[buf_cnt] = FLAG;

    return write(fd, FRAME, buf_cnt + 2);
}

int checkFileSize(int filesize, const char *filename)
{

    struct stat file_stat;

    if (stat(filename, &file_stat) < 0)
    {
        printf("Error getting file information.");
        return -1;
    }

    return (filesize != file_stat.st_size);
}

void printProgressBar(float current, float total)
{

    float percentage = 100.0 * current / total;

    printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
    printf("\n%.2f%% [", percentage);

    int len = 50;
    int pos = percentage * len / 100.0;
    for (int i = 0; i < len; i++)
        i <= pos ? printf("#") : printf(" ");

    printf("]\n");
}

void printStatistics(int role, int RR, int REJ)
{
    if (role)
    {
        printf("\nSTATISTICS\n\n");
        printf("Number of  RR received: %d\n", RR);
        printf("Number of  REJ received: %d\n", REJ);
    }
    else
    {
```

```
        printf("\nSTATISTICS\n\n");
        printf("Number of  RR sent: %d\n", RR);
        printf("Number of  REJ sent: %d\n", REJ);
    }
}


char* getInputString()
{
    int done = 0;
    char* inputs = (char*) malloc(256 * sizeof(char));

    while (!done) {
        if (scanf("%s", inputs) == 1) done = 1;
        else {
            printf("\nInvalid input, try again!\n");
        }

        clearInputBuffer();
    }

    return inputs;
}


int getInputInt(int start, int end)
{
    int input;
    int done = 0;
    while (!done) {
       if (scanf("%d", &input) == 1 && input >= start && input <= end) {
           done = 1;
       }
       else printf("\nInvalid input, try again!\n");

       clearInputBuffer();
    }

    return input;
}

void clearInputBuffer() {
        int c;
        while ((c = getchar()) != '\n' && c != EOF);
}
```