FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Relatório do 1º Trabalho Laboratorial de Redes de Computadores - Protocolo de Ligação de Dados

Turma 3LEIC03, Grupo 8
Daniel José de Aguiar Gago (up202108791)
Francisco Silva Cunha Campos (up202108735)

Sumário

Este projeto foi realizado no âmbito da disciplina de Redes de Computadores, e tem como objetivo implementar um sistema de comunicação de dados que permita a transmissão de ficheiros através da Porta Série RS-232.

A estratégia utilizada para este protocolo foi a de Stop-and-Wait, implementando um protocolo de comunicação que permitisse o envio do ficheiro tanto em situações normais, como de desconexão e ruído que causasse interferência nos dados enviados e recebidos.

Introdução

O objetivo deste projeto foi desenvolver e testar o protocolo de ligação de dados, segundo a estratégia Stop-and-Wait, usado para enviar um ficheiro de um transmissor para um receptor. Quanto ao relatório, o objetivo é expor a forma como seguimos o guião do trabalho, e fazer a análise da eficiência do protocolo.
O relatório está dividido nas seguintes 8 secções:

- **Arquitetura:** Blocos funcionais e interfaces utilizadas;
- **Estrutura do código:** Descrição das APIs, principais estruturas de dados, principais funções e sua relação com a arquitetura;
- **Casos de Uso Principais:** Identificação dos casos de uso, e sequências de chamada de funções;
- **Protocolo de ligação lógica:** Identificação dos principais aspetos funcionais, e descrição da estratégia de implementação destes aspetos com apresentação de extratos de código;
- **Validação:** Descrição dos testes efetuados;
- **Eficiência do protocolo de ligação de dados:** ASFA
- **Conclusões:** Síntese da informação apresentada nas secções anteriores, e reflexão sobre os objetivos de aprendizagem alcançados.

No fim do relatório encontra-se disponível anexos com o código fonte do projeto.

Arquitetura

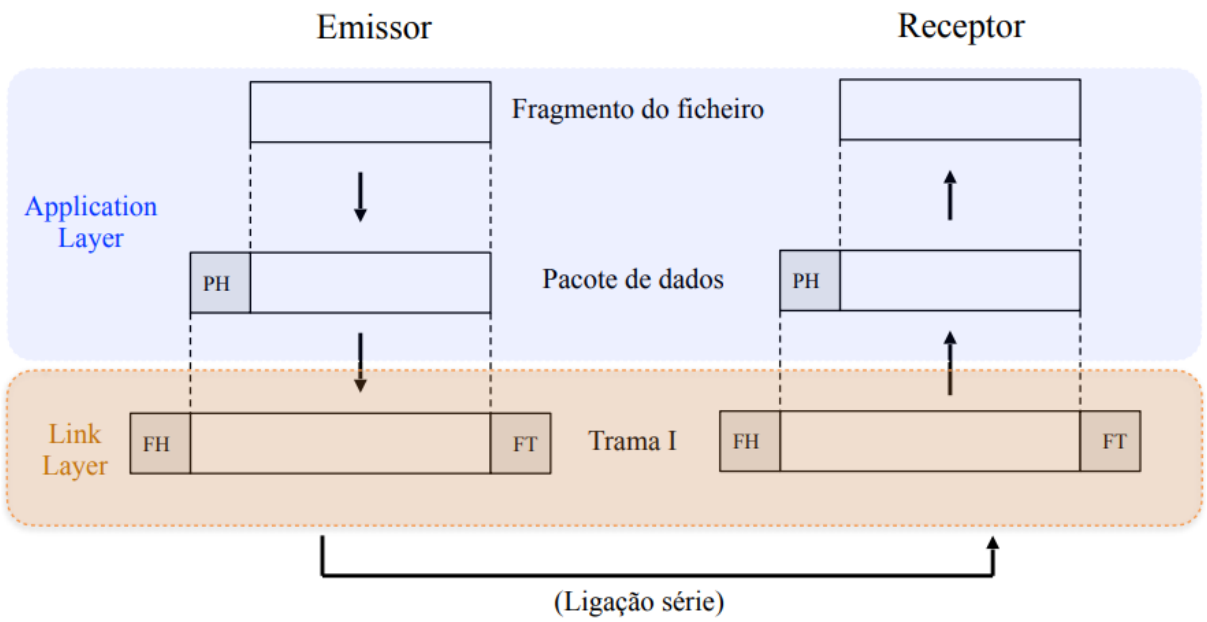
Blocos Funcionais

O projeto foi desenvolvido utilizando as duas camadas principais explicitadas no guião: a *LinkLayer* e a *ApplicationLayer*.

A *LinkLayer*, também chamada de camada de ligação de dados, é responsável pela ligação direta com a porta série, através das chamadas de read e write. Em termos lógicos, está responsável pela criação de tramas de informação através do pacote de dados fornecido pela Application Layer, e envio dessas mesmas tramas (do lado do transmissor), mas também além da validação e envio de mensagens a aceitar ou rejeitar a trama recebida (do lado do recetor). Esta camada está implementada nos ficheiros *link_layer.h* e *link_layer.c*.

A *ApplicationLayer*, também chamada de camada de aplicação, é a camada mais superficial, isto é, que está em maior contacto com o utilizador. Esta camada está responsável pela emissão e receção dos pacotes de dados do ficheiro em causa. A *ApplicationLayer* está implementada nos ficheiros *application_layer.h* e *application_layer.c*.

A seguinte imagem explica de forma mais visual o que foi anteriormente explicado:



Interfaces

O programa necessita de dois terminais para ser executado, um em cada computador, em que cada um executa um binário em modo transmissor, e outro em modo de receptor.

```
Unset
$ <PROGRAM> <SERIA_PORT> <ROLE> <FILE>

- PROGRAM: binário a executar
- SERIAL_PORT: designação da porta série a utilizar
- ROLE: 'tx' para transmissor, 'rx' para recetor
- FILE: nome do ficheiro a transferir/receber
```

Para facilitar, também é possível executar o comando **make run_tx** para correr o binário em modo de transmissão e **make run_rx** para modo de receptor.

Estrutura do Código

ApplicationLayer

Nesta camada não foi necessária a criação de estruturas de dados adicionais. Esta foi a única função implementada:

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

Esta função tem um switch statement, que de acordo com o role apresentado no binário, faz com que o utilizador aceda a diferentes blocos do código:

- Se o terminal for transmissor (**LlTx**), vai chamar as funções llopen, llwrite e llclose, para além de **empacotar os fragmentos do ficheiro** e torná-los em pacotes de dados;
- Se o terminal for recetor (**LlRx**), vai chamar as funções llopen, llread e llclose, para além de **desempacotar os pacotes de dados**, para utilizar os fragmentos do ficheiro enviado;

LinkLayer

Nesta camada, foram utilizadas duas estruturas de dados: LinkLayer, onde são caracterizados os parâmetros iniciais definidos em main.c para a transferência de dados, e o LinkLayerRole, que identifica se o programa está em modo de transmissor ou receptor.

```C/C++
typedef enum
{
   LlTx,
   LlRx,
} LinkLayerRole;
```

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

Para além disso, estas foram as 4 funções desenvolvidas nesta camada:

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

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

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

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

Casos de Uso Principais

Como foi dito anteriormente, o main.c chama a função `applicationLayer`, e depois existe uma subdivisão do código, já explicada aqui. Abaixo encontra-se a explicação mais detalhada da ordem de execução e lógica das funções do projeto:

Transmissor

1. **llopen(...)**, lança a primeira trama para iniciar o contacto entre o transmissor e o recetor, e fica à espera da resposta por parte do recetor;
2. **llwrite(...)**, função responsável por enviar os pacotes recebidos no argumento por parte do ApplicationLayer, torná-los em tramas de informação, e enviar para o recetor. Depois aguarda resposta para saber se a trama foi aceite, ou rejeitada. Em caso de rejeição ou de timeout, volta a enviar a trama prévia.
3. **llclose(...)**, o transmissor entra nesta função quando envia todos os dados do ficheiro, mais um packet de controlo a anunciar o fim do ficheiro. Envia um pedido ao recetor para desconectar, e aguarda resposta.

Recetor

1. **llopen(...)**, recebe a primeira trama de estabelecimento de conexão (SET), e envia para o transmissor uma trama a confirmar a receção;

2. **llread(...)**, função responsável por enviar para o link layer os pacotes lidos na trama de informação. Primeiro lê a trama e valida os dados enviados. Depois, rejeita ou aceita a trama recebida. Caso aceite a trama em questão, usa o packet do argumento para enviar a informação para onde a função foi chamada (neste caso, o ApplicationLayer)
3. **llclose(...)**, o recetor entra nesta função quando recebe o packet final a indicar o fim do envio do ficheiro. Aguarda pelo pedido de desconexão do transmissor, e quando recebe envia o seu, e aguarda pela resposta UA por parte do transmissor.

Protocolo de Ligação Lógica

A camada de ligação de dados interage (*LinkLayer*) diretamente com a porta série (através da funções read() e write()). Assim, para garantir a devida comunicação entre o transmissor e receptor, é implementado o protocolo Stop-and-wait, para o começo e término da ligação, e claro, para enviar os dados do ficheiro em questão.

A conexão é iniciada na chamada da função **llopen**. No começo (após a configuração da porta série), o transmissor envia a trama de supervisão SET, e aguarda a trama de supervisão UA por parte do receptor. Caso a trama SET seja validada pelo receptor, e a UA pelo transmissor, a ligação entre o transmissor e receptor é bem sucedida, e o transmissor está pronto para começar a enviar dados, e o transmissor a lê-los.

De seguida, o envio dos packets estabelecidos na ApplicationLayer é feito na função **llwrite**. A função recebe um pacote de controlo ou de dados, e calcula logo o BCC2 do dados (operações XOR byte a byte). De seguida, realiza a estratégia de byte stuffing. Esta consiste em trocar todos os bytes com o mesmo valor das flags por um dois octetos. Esta técnica assegura a transparência da trama, de forma a garantir que só o início e o fim da trama tem os bytes correspondentes à flag (0x7E). Também é preciso dar stuffing ao octeto do escape, para garantir que sempre que esse octeto aparece, estamos na presença de um stuffed byte (em suma, 0x7E fica 0x7D 0x5E, e o 0x7D fica 0x7D 0x5D). Esta foi a nossa estratégia a nível de código para implementar este protocolo:

```C/C++
while (i < bufSize)
  {
    if (buf[i] == FLAG)
    {
      write_buf[j] = ESC;
      write_buf[j + 1] = 0x5E;
      j += 2;
    }
    else if (buf[i] == ESC)
    {
      write_buf[j] = ESC;
      write_buf[j + 1] = 0x5D;
      j += 2;
    }
```

```
    else
    {
      write_buf[j] = buf[i];
      j++;
    }
    i++;
  }
  if (bcc2 == FLAG)
  {
    write_buf[j] = ESC;
    write_buf[j + 1] = 0x5E;
    j += 2;
  }
  else if (bcc2 == ESC)
  {
    write_buf[j] = ESC;
    write_buf[j + 1] = 0x5D;
    j += 2;
  }
  else
  {
    write_buf[j] = bcc2;
    j++;
  }
```

Depois é realizado o framing, colocando um header e um footer no pacote, transformando-o assim numa trama, e é enviada para o receptor. O transmissor tem o número N de tentativas máximas para o envio do pacote. A trama é retransmitida em caso de timeout (muito tempo sem receber resposta do receptor), ou caso a trama seja rejeitada. Caso exceda o número máximo de tentativas, a comunicação é cancelada.

A leitura da informação é realizada no **llread**. Esta função lê a informação recebida na porta série, e verifica se existe algum erro, mandando uma mensagem de acordo com a validade da trama. Primeiro faz o destuffing, que é precisamente o contrário do destuffing: encontra octetos de escape e torna-os no respetivo dado original.
Aqui está a forma que nós utilizamos para fazer destuffing em código na máquina de estados (no estado em que estamos à procura do byte final):

C/C++
```
      if (read_byte == FLAG)
          {
               if (bcc2 == 0x00)
               {
                   packet[pos - 1] = '\0';
                   STOP = TRUE;
```

```
            if (N_local == I0 && writer_nlocal == I0)
            {
                new_packet = TRUE;
                response = RR1;
                N_local = I1;
            }
            else if (N_local == I1 && writer_nlocal == I1)
            {
                new_packet = TRUE;
                response = RR0;
                N_local = I0;
            }
            else if (N_local == I0 && writer_nlocal == I1)
            {
                response = RR0;
            }
            else if (N_local == I1 && writer_nlocal == I0)
            {
                response = RR1;
            }
        }
        else
        {
            printf("Error in BCC2\n");
            if (N_local == I0)
                response = REJ0;
            else if (N_local == I1)
                response = REJ1;
            pos = 0;
            STOP = TRUE;
        }
    }
    else if (escapeRead)
    {
        if (read_byte == 0x5D)
        {
            packet[pos] = ESC;
            bcc2 = bcc2 ^ ESC;
        }
        else if (read_byte == 0x5E)
        {
            packet[pos] = FLAG;
            bcc2 = bcc2 ^ FLAG;
        }
        escapeRead = FALSE;
        pos++;
    }
```

```
else if (read_byte == ESC)
{
    escapeRead = TRUE;
}
else
{
    packet[pos] = read_byte;
    bcc2 = bcc2 ^ read_byte;
    pos++;
}
```

De seguida, valida os campos do BCC1 e BCC2, para ver se ocorreu algum erro na transmissão, e envia a resposta tendo em conta a sua variável local se está a 0x00 (N = 0) ou 0x40 (N = 1). Se a posição de controlo da trama for igual ao valor da variável local, conseguimos detectar que a trama de informação é repetida, e portanto, que não devemos voltar a ler o packet, pois já temos os dados desse mesmo pacote de dados.

Por fim, o término da ligação é realizado na função **llclose**, onde o transmissor envia uma trama DISC para anunciar ao recetor o fim da transferência. O recetor responde com outra trama DISC, e aguarda a resposta com uma trama UA por parte do transmissor, dando por concluída a ligação.

Protocolo de Aplicação

A camada de aplicação, por sua vez, irá interagir com a camada de ligação de dados (através das funções llopen(...), llwrite(...), llread(...) e llclose(...)). Esta camada é responsável por receber o ficheiro de input e dividi-lo em pacotes de informação para enviar no caso do transmissor e por processar os pacotes e criar o ficheiro recebido localmente no caso do receptor.

Começa por chamar **llopen**, de modo a iniciar a conexão entre as portas série, tal como descrito na secção anterior. Se isto for sucedido, a aplicação divide-se em 2 blocos distintos: o transmissor e o receptor, que irão enviar (usando o **llwrite**) e receber (usando o **llread**) o ficheiro, respetivamente, um pacote de cada vez. Após o envio completo do ficheiro, a aplicação chama **llclose** e termina a execução.

Transmissor

A aplicação do transmissor começa a enviar pacotes usando o **llwrite**, sendo que o primeiro será o pacote de arranque, identificado pelo número 2 no cabeçalho do pacote. Este contém informações acerca do ficheiro que será enviado em seguida (no nosso caso, apenas enviámos o tamanho do ficheiro, assim como o seu nome). Depois disto, os pacotes de informação começam a ser enviados. Cada pacote de informação contém um cabeçalho que informa ao receptor que o pacote a ser enviado é um pacote de informação (identificado pelo número 1) assim como o tamanho do pacote a ser enviado. Definimos um limite de

1000 *bytes* por pacote. Quando não existir mais informação para enviar, o transmissor informa o reader através de um pacote de fecho, identificado pelo número 3 no cabeçalho e que contém as mesmas informações que foram enviadas no pacote de arranque.

```C/C++
case LlTx:;
    FILE *input_file = fopen(filename, "rb");
    if (input_file == NULL)
    {
      perror("File not found\n");
      exit(-1);
    }

    fseek(input_file, 0, SEEK_END);
    long input_file_size = ftell(input_file);

    // Start packet
    int tx_control_packet_size = 5 + sizeof(long) + strlen(filename);

    unsigned char *tx_control_packet = (unsigned char
*)malloc(tx_control_packet_size * sizeof(unsigned char));
    int tx_control_packet_pos = 0;
    int timeout = FALSE;

    tx_control_packet[tx_control_packet_pos++] = 2;
    tx_control_packet[tx_control_packet_pos++] = 0;
    tx_control_packet[tx_control_packet_pos++] = sizeof(long);

    for (int i = 0; i < sizeof(long); i++)
    {
      tx_control_packet[tx_control_packet_pos++] = (unsigned
char)((input_file_size >> (i * 8)) & 0xFF);
    }

    tx_control_packet[tx_control_packet_pos++] = 1;
    tx_control_packet[tx_control_packet_pos++] = strlen(filename);

    for (size_t i = 0; i < strlen(filename); i++)
    {
      tx_control_packet[tx_control_packet_pos++] = filename[i];
    }

    if(llwrite(tx_control_packet, tx_control_packet_size) == FALSE)
    {
      printf("APPLICATION: START PACKET TIMEOUT. CLOSING...\n");
      break;
    }
    free(tx_control_packet);
```

```
printf("Start packet successfuly sent.\n");

// Data packets
fseek(input_file, 0, SEEK_SET);
unsigned char data[MAX_PAYLOAD_SIZE];
int bytes_read;
int tx_packet_counter = 1;
while ((bytes_read = fread(data, 1, sizeof(data), input_file)) > 0)
{
  unsigned char *tx_data_packet = (unsigned char *)malloc(3 + bytes_read);
  int tx_data_packet_pos = 0;

  tx_data_packet[tx_data_packet_pos++] = 1;
  tx_data_packet[tx_data_packet_pos++] = bytes_read >> 8 & 0xFF;
  tx_data_packet[tx_data_packet_pos++] = bytes_read & 0xFF;

  for (int i = 0; i < bytes_read; i++)
  {
    tx_data_packet[tx_data_packet_pos++] = data[i];
  }

  if(llwrite(tx_data_packet, bytes_read + 3) == FALSE)
  {
    printf("APPLICATION: DATA PACKET TIMEOUT. CLOSING...\n");
    timeout = TRUE;
    break;
  }
  free(tx_data_packet);
  printf("Data packet #%d successfuly sent.\n", tx_packet_counter);
  tx_packet_counter++;
}

if(timeout == TRUE)
  break;

// End packet
tx_control_packet[0] = 3;
if(llwrite(tx_control_packet, tx_control_packet_size) == FALSE)
{
  printf("APPLICATION: END PACKET TIMEOUT. CLOSING...\n");
  break;
}
printf("End packet successfuly sent.\n");

if(llclose(TRUE)==FALSE){
  printf("Failed to close connection");
}else{
  printf("Success.\n");
}
```

Receptor

A aplicação do receptor irá receber os pacotes que estão a ser enviados pelo transmissor usando o **llread**, o primeiro sendo o pacote de arranque. Este contém informações acerca do ficheiro que irá ser recebido e processado. Destas informações, apenas utilizamos o tamanho do ficheiro, de modo a garantir que o ficheiro que criamos localmente tem o mesmo tamanho que o ficheiro original, já que o nome é dado como argumento quando o programa é chamado. Depois disto, os pacotes de informação começam a ser recebidos e escritos localmente. Quando é detectado um erro no **llread**, o programa ignora o pacote e chama a função novamente. Isto continua até ser recebido o pacote de fecho. Nesse momento, o ficheiro é fechado e o protocolo chega ao fim.

```C/C++
  case LlRx:;
    // Start packet
    unsigned char rx_control_packet[MAX_PAYLOAD_SIZE];

    while(llread(rx_control_packet) == FALSE) {
      printf("APPLICATION: START PACKET NOT ACCEPTED. TRYING AGAIN...\n");
    };
    printf("Start packet successfuly received.\n");
    unsigned char L1 = rx_control_packet[2];
    long output_file_size = 0;
    for (unsigned int i = 0; i < L1; i++)
      output_file_size = (output_file_size << 8) | rx_control_packet[i + 3];

    // still need to parse name (dont know why)

    // Data packets
    FILE *output_file = fopen(filename, "wb");
    if (output_file == NULL)
    {
      perror("Could not open file\n");
      exit(-1);
    }

    unsigned char rx_data_packet[MAX_PAYLOAD_SIZE + 3];
    int rx_packet_counter = 1;
    while (TRUE)
    {
      while(llread(rx_data_packet) != TRUE) {
        printf("APPLICATION: DATA PACKET NOT ACCEPTED. TRYING AGAIN...\n");
      }
      long rx_data_size = 0;
      if (rx_data_packet[0] == 1)
```

```
      {
        rx_data_size = (rx_data_packet[1] << 8) | rx_data_packet[2];
        fwrite(&rx_data_packet[3], sizeof(char), rx_data_size, output_file);
        printf("Data packet #%d successfuly received and written to file.\n",
rx_packet_counter);
        totalDataBytesReceived += rx_data_size;
        rx_packet_counter++;
      }
      else if (rx_data_packet[0] == 3){
        printf("End packet successfuly received.\n");
        break;
      }
    }
    if(fclose(output_file))
      perror("File not closed correctly");
    if(llclose(TRUE) == FALSE){
      printf("Failed to close connection\n");
    }
    else{
      printf("Success.\n");
    }
```

Validação

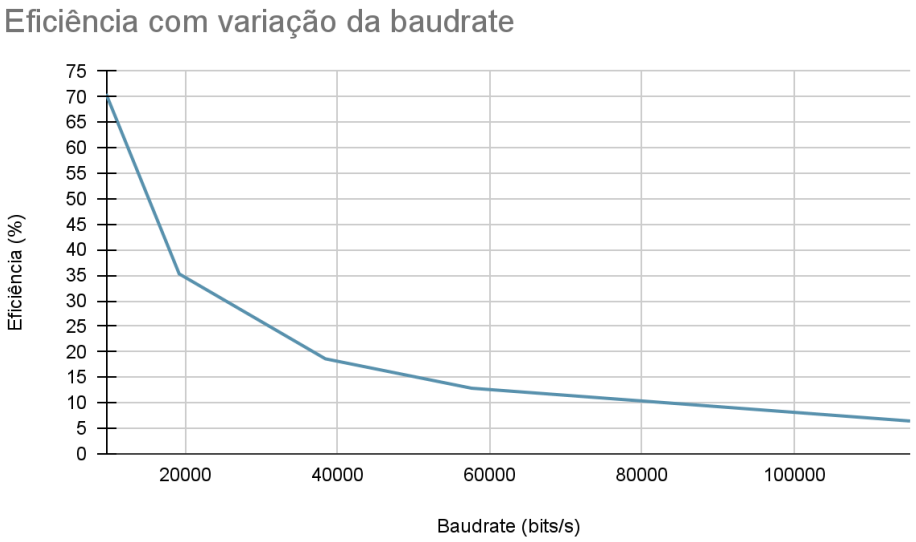
De modo a testar o programa desenvolvido, enviar uma imagem de 3 maneiras diferentes: envio simples, envio com interrupção da porta série e envio com ruído.

- **Envio simples:** o ficheiro é enviado e recebido sem problemas.
- **Envio com interrupção da porta série:** O envio de algumas tramas é interrompido, porém, o programa volta a enviar a mesma trama depois de o alarme ser acionado Quando a ligação com a porta série é restabelecida, o ficheiro continua a ser enviado e recebido sem problemas adicionais.
- **Envio com ruído:** Algumas tramas são recebidas com erros, tanto no header (confirmado pelo BCC1) como na informação (confirmado pelo BCC2). Quando isto acontece, o receptor ignora ou pede a retransmissão da trama, respetivamente. À parte disso, o ficheiro é enviado e recebido sem problemas adicionais.

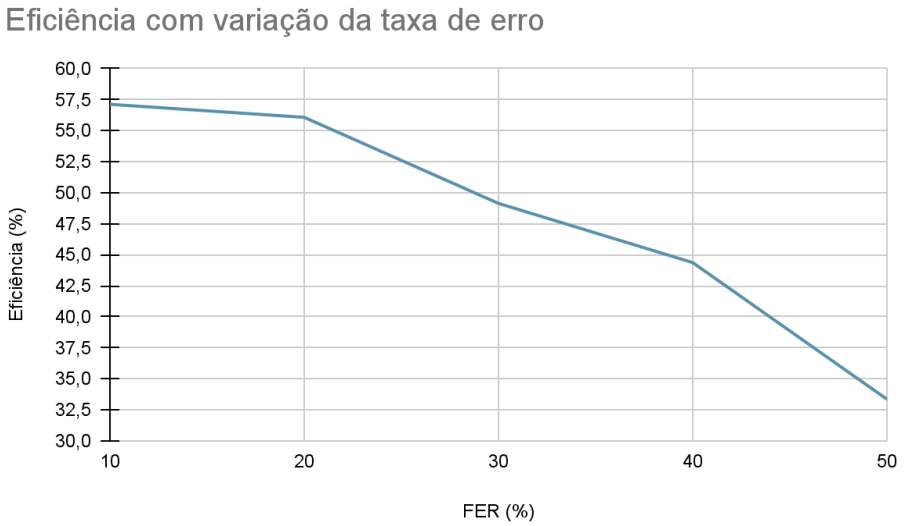
Eficiência do Protocolo de Ligação de Dados

De modo a testar a eficiência do nosso programa, calculamos, para diferentes valores da baudrate (R), diferentes taxas de erro (FER), diferentes tamanhos de trama (S) e diferentes distâncias de propagação, a sua eficiência (E) usando um cálculo simples:

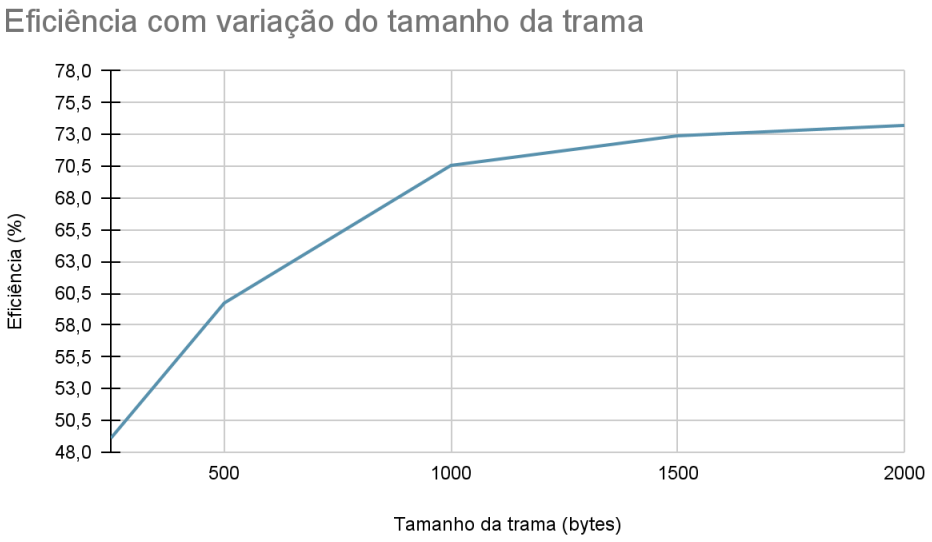
$$Rlab(bits/s) = \frac{FileSize\ (bits)}{Time\ (s)} \qquad Eficiência\ (\%) = \frac{Rlab\ (bits/s)}{R\ (bits/s)}$$



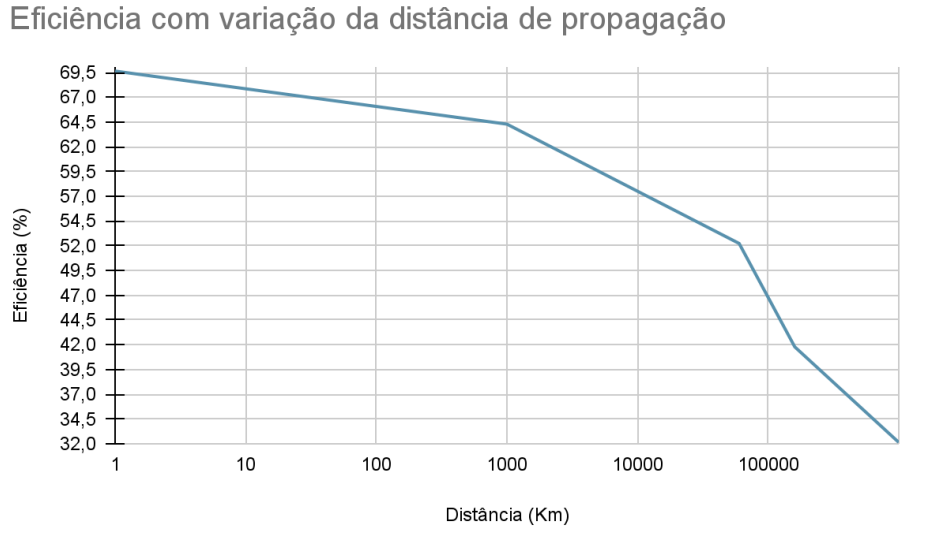
Como se consegue observar, quanto maior for o Baudrate, menor é a eficiência do protocolo. Aliás, a relação entre a Eficiência e o Baudrate aparenta ser no gráfico **inversamente proporcional**. Isto vai de encontro à teoria, visto que $Eficiência = \frac{T_f}{T_{prop}+T_f+T_{prop}} = \frac{1}{1+2a}$, $sendo\ a = \frac{T_{prop}}{T_f}$ e $T_f = \frac{L}{R}$, sendo R o Baudrate. Desta forma, conseguimos concluir que o gráfico obtido está correto de acordo com a informação teórica fornecida.



Visto que este programa segue o protocolo de ligação de dados Stop-and-Wait, sempre que existe um erro no envio de uma trama, esta será retransmitida, o que causará um aumento do tempo total da transmissão. Isto é confirmado pelo que observámos, já que, à medida que a FER (Frame Error Rate) aumenta, a eficiência diminui proporcionalmente, o que era esperado, visto que o cálculo da eficiência é dado por $Eficiência\ (\%) = \frac{FileSize}{Time * R}$.



Como observamos no gráfico, percebemos que a eficiência aumenta de acordo com o tamanho da trama. Isto porque quanto maior for a trama, menor será a quantidade de tempo gasta em operações de Stop-and-Wait, menor será o número de header e footers colocados nas tramas de informação, e assim, mais eficiente é a transmissão de informação do transmissor para o recetor. Assim, os resultados obtidos estão de acordo com a lógica da eficiência anteriormente descrita, e descrevem um relação de linearidade com a eficiência do protocolo, confirmada nas seguintes fórmulas $Eficiência = \frac{1}{1+2a}$, $sendo\ a = \frac{T_{prop}}{T_f}$ e $T_f = \frac{L}{R}$, sendo L o tamanho da trama.



As duas fórmulas que nos permitem saber teoricamente a influência que a distância tem na eficiência do protocolo são: $Eficiência = \frac{1}{1+2a}$, $sendo\ a = \frac{T_{prop}}{T_f}$ e $T_{prop} = \frac{d}{v}$, sendo d a distância entre os dois terminais, e $v = 5\mu s/km$ (constante). Assim, podemos ver que a

eficiência será **inversamente proporcional** à distância entre os dois computadores, cuja teoria é confirmada pelo gráfico acima.

Conclusões

Durante o desenvolvimento deste programa, observámos a interação direta da camada de aplicação com o utilizador e a imagem a ser enviada, enquanto a camada de ligação de dados lidava diretamente com a comunicação via porta série. Através deste processo, foi possível aprofundar o nosso conhecimento acerca dos aspectos essenciais de um protocolo de ligação de dados do estilo Stop-and-Wait.

Anexos I - application_layer.h

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

#ifndef _APPLICATION_LAYER_H_
#define _APPLICATION_LAYER_H_

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

#endif // _APPLICATION_LAYER_H_
```

Anexos II - application_layer.c

```C/C++
// Application layer protocol implementation

#include "application_layer.h"
```

```c
#include "link_layer.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern int totalDataBytesReceived;

void applicationLayer(const char *serialPort, const char *role, int baudRate,
          int nTries, int timeout, const char *filename)
{
  LinkLayer connectionParameters;

  connectionParameters.baudRate = baudRate;
  connectionParameters.nRetransmissions = nTries;
  if (role[0] == 't' && role[1] == 'x')
    connectionParameters.role = LlTx;
  else if (role[0] == 'r' && role[1] == 'x')
    connectionParameters.role = LlRx;
  else
    perror("Invalid role.\n");
  for (int i = 0; i < 12; i++)
    connectionParameters.serialPort[i] = serialPort[i];
  connectionParameters.timeout = timeout;

  if (llopen(connectionParameters) == FALSE)
  {
    perror("Connection failed.\n");
    exit(-1);
  }
  printf("Connection established.\n");

  switch (connectionParameters.role)
  {
  case LlTx:;
    FILE *input_file = fopen(filename, "rb");
    if (input_file == NULL)
    {
      perror("File not found\n");
      exit(-1);
    }

    fseek(input_file, 0, SEEK_END);
    long input_file_size = ftell(input_file);

    // Start packet
    int tx_control_packet_size = 5 + sizeof(long) + strlen(filename);

    unsigned char *tx_control_packet = (unsigned char
*)malloc(tx_control_packet_size * sizeof(unsigned char));
```

```
    int tx_control_packet_pos = 0;
    int timeout = FALSE;

    tx_control_packet[tx_control_packet_pos++] = 2;
    tx_control_packet[tx_control_packet_pos++] = 0;
    tx_control_packet[tx_control_packet_pos++] = sizeof(long);

    for (int i = 0; i < sizeof(long); i++)
    {
      tx_control_packet[tx_control_packet_pos++] = (unsigned
char)((input_file_size >> (i * 8)) & 0xFF);
    }

    tx_control_packet[tx_control_packet_pos++] = 1;
    tx_control_packet[tx_control_packet_pos++] = strlen(filename);

    for (size_t i = 0; i < strlen(filename); i++)
    {
      tx_control_packet[tx_control_packet_pos++] = filename[i];
    }

    if(llwrite(tx_control_packet, tx_control_packet_size) == FALSE)
    {
      printf("APPLICATION: START PACKET TIMEOUT. CLOSING...\n");
      break;
    }
    free(tx_control_packet);
    printf("Start packet successfuly sent.\n");

    // Data packets
    fseek(input_file, 0, SEEK_SET);
    unsigned char data[MAX_PAYLOAD_SIZE];
    int bytes_read;
    int tx_packet_counter = 1;
    while ((bytes_read = fread(data, 1, sizeof(data), input_file)) > 0)
    {
      unsigned char *tx_data_packet = (unsigned char *)malloc(3 + bytes_read);
      int tx_data_packet_pos = 0;

      tx_data_packet[tx_data_packet_pos++] = 1;
      tx_data_packet[tx_data_packet_pos++] = bytes_read >> 8 & 0xFF;
      tx_data_packet[tx_data_packet_pos++] = bytes_read & 0xFF;

      for (int i = 0; i < bytes_read; i++)
      {
        tx_data_packet[tx_data_packet_pos++] = data[i];
      }

      if(llwrite(tx_data_packet, bytes_read + 3) == FALSE)
```

```
    {
      printf("APPLICATION: DATA PACKET TIMEOUT. CLOSING...\n");
      timeout = TRUE;
      break;
    }
    free(tx_data_packet);
    printf("Data packet #%d successfuly sent.\n", tx_packet_counter);
    tx_packet_counter++;
  }

  if(timeout == TRUE)
    break;

  // End packet
  tx_control_packet[0] = 3;
  if(llwrite(tx_control_packet, tx_control_packet_size) == FALSE)
  {
    printf("APPLICATION: END PACKET TIMEOUT. CLOSING...\n");
    break;
  }
  printf("End packet successfuly sent.\n");

  if(llclose(TRUE)==FALSE){
    printf("Failed to close connection");
  }else{
    printf("Success.\n");
  }

  break;
case LlRx:;
  // Start packet
  unsigned char rx_control_packet[MAX_PAYLOAD_SIZE];

  while(llread(rx_control_packet) == FALSE) {
    printf("APPLICATION: START PACKET NOT ACCEPTED. TRYING AGAIN...\n");
  };
  printf("Start packet successfuly received.\n");
  unsigned char L1 = rx_control_packet[2];
  long output_file_size = 0;
  for (unsigned int i = 0; i < L1; i++)
    output_file_size = (output_file_size << 8) | rx_control_packet[i + 3];

  // still need to parse name (dont know why)

  // Data packets
  FILE *output_file = fopen(filename, "wb");
  if (output_file == NULL)
  {
    perror("Could not open file\n");
```

```
      exit(-1);
    }

    unsigned char rx_data_packet[MAX_PAYLOAD_SIZE + 3];
    int rx_packet_counter = 1;
    while (TRUE)
    {
      while(llread(rx_data_packet) != TRUE) {
        printf("APPLICATION: DATA PACKET NOT ACCEPTED. TRYING AGAIN...\n");
      }
      long rx_data_size = 0;
      if (rx_data_packet[0] == 1)
      {
        rx_data_size = (rx_data_packet[1] << 8) | rx_data_packet[2];
        fwrite(&rx_data_packet[3], sizeof(char), rx_data_size, output_file);
        printf("Data packet #%d successfuly received and written to file.\n",
rx_packet_counter);
        totalDataBytesReceived += rx_data_size;
        rx_packet_counter++;
      }
      else if (rx_data_packet[0] == 3){
        printf("End packet successfuly received.\n");
        break;
      }
    }
    if(fclose(output_file))
      perror("File not closed correctly");
    if(llclose(TRUE) == FALSE){
      printf("Failed to close connection\n");
    }
    else{
      printf("Success.\n");
    }
    break;
  }
}
```

Anexos III - link_layer.h

C/C++

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

```c
#ifndef _LINK_LAYER_H_
#define _LINK_LAYER_H_

typedef enum
{
  LlTx,
  LlRx,
} LinkLayerRole;

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

// SIZE of maximum acceptable payload.
#define MAX_PAYLOAD_SIZE 1000

// MISC
#define FALSE 0
#define TRUE 1
#define FLAG 0x7E
#define ESC 0x7D
#define A_SENDER 0x03
#define A_RECEIVER 0x01
#define UA 0x07
#define SET 0x03
#define RR0 0x05
#define RR1 0x85
#define REJ0 0x01
#define REJ1 0x81
#define DISC 0x0B
#define I0 0x00
#define I1 0x40

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

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

// Receive data in packet.
// Return number of chars read, or "-1" on error.
```

```
int llread(unsigned char *packet);

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

#endif // _LINK_LAYER_H_
```

Anexos IV - link_layer.c

C/C++

```
// Link layer protocol implementation
#include "link_layer.h"

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

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

LinkLayerRole linkLayerRole;
struct termios oldtio;
int maxNReTransmissions = 3;
int timeout = 4;

int state = 0;
unsigned char N_local = I0;
int fd;

volatile int STOP = FALSE;

int alarmEnabled = FALSE;
int nReTransmissions = 0;
```

```c
//Statistics
int totalAlarms = 0;
int totalRejects = 0;
int totalReTransmissions = 0;
int totalBytesSent = 0;
int totalDataBytesReceived = 0;
struct timeval begin, end;

void alarmHandler(int signal)
{
  alarmEnabled = FALSE;
  nReTransmissions++;
  totalAlarms++;
  totalReTransmissions++;
  state = 0;
  STOP = TRUE;

  printf("Alarm activated (Retransmission #%d)\n", nReTransmissions);
}

////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters)
{
  gettimeofday(&begin, 0);
  fd = open(connectionParameters.serialPort, O_RDWR | O_NOCTTY);
  if (fd < 0)
  {
    perror(connectionParameters.serialPort);
    exit(-1);
  }

  struct termios newtio;
  if (tcgetattr(fd, &oldtio) == -1)
  {
    perror("tcgetattr");
    exit(-1);
  }

  memset(&newtio, 0, sizeof(newtio));

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

  newtio.c_lflag = 0;
  newtio.c_cc[VTIME] = 1; // Inter-character timer
```

```
newtio.c_cc[VMIN] = 0; // Blocking read until

tcflush(fd, TCIOFLUSH);

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

int success = FALSE;
unsigned char write_buf[5] = {0};
linkLayerRole = connectionParameters.role;
maxNReTransmissions = connectionParameters.nRetransmissions;
timeout = connectionParameters.timeout;
switch (linkLayerRole)
{
case LlTx:;
  printf("LlTx\n");
  write_buf[0] = FLAG;
  write_buf[1] = A_SENDER;
  write_buf[2] = SET;
  write_buf[3] = A_SENDER ^ SET;
  write_buf[4] = FLAG;

  (void)signal(SIGALRM, alarmHandler);

  while (nReTransmissions < maxNReTransmissions && state != 5)
  {
    if (alarmEnabled == FALSE)
    {
      STOP = FALSE;
      write(fd, write_buf, 5);
      alarm(timeout);
      alarmEnabled = TRUE;

      unsigned char read_byte;
      while (STOP == FALSE)
      {
        read(fd, &read_byte, 1);
        switch (state)
        {
        case 0:
          if (read_byte == FLAG)
            state = 1;
          else
            state = 0;
          break;
```

```
        case 1:
          if (read_byte == FLAG)
            state = 1;
          else if (read_byte == A_RECEIVER)
            state = 2;
          else
            state = 0;
          break;
        case 2:
          if (read_byte == FLAG)
            state = 1;
          else if (read_byte == UA)
            state = 3;
          else
            state = 0;
          break;
        case 3:
          if (read_byte == FLAG)
            state = 1;
          else if (read_byte == (A_RECEIVER ^ UA))
            state = 4;
          else
            state = 0;
          break;
        case 4:
          if (read_byte == FLAG)
          {
            STOP = TRUE;
            success = TRUE;
            alarm(0);
            state = 5;
          }
          else
            state = 0;
          break;
        default:
          break;
        }
      }
    }
  }
  break;
case LlRx:;
  printf("LlRx\n");
  unsigned char read_byte;
  while (STOP == FALSE)
  {
    read(fd, &read_byte, 1);
```

```
  switch (state)
  {
  case 0:
    if (read_byte == FLAG)
      state = 1;
    else
      state = 0;
    break;
  case 1:
    if (read_byte == FLAG)
      state = 1;
    else if (read_byte == A_SENDER)
      state = 2;
    else
      state = 0;
    break;
  case 2:
    if (read_byte == FLAG)
      state = 1;
    else if (read_byte == SET)
      state = 3;
    else
      state = 0;
    break;
  case 3:
    if (read_byte == FLAG)
      state = 1;
    else if (read_byte == (A_SENDER ^ SET))
      state = 4;
    else
      state = 0;
    break;
  case 4:
    if (read_byte == FLAG)
    {
      STOP = TRUE;
      success = TRUE;
    }
    else
      state = 0;
    break;
  }
}

write_buf[0] = FLAG;
write_buf[1] = A_RECEIVER;
write_buf[2] = UA;
write_buf[3] = A_RECEIVER ^ UA;
```

```
    write_buf[4] = FLAG;

    write(fd, write_buf, 5);

    break;
  default:
    break;
  }
  return success;
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(const unsigned char *buf, int bufSize)
{
  // sleep(1);
  nReTransmissions = 0;
  state = 0;
  alarmEnabled = FALSE;
  unsigned char *write_buf = (unsigned char *)malloc(bufSize * 2);
  write_buf[0] = FLAG;
  write_buf[1] = A_SENDER;
  if (N_local == I0)
    write_buf[2] = I0;
  else
    write_buf[2] = I1;
  write_buf[3] = A_SENDER ^ write_buf[2]; // BCC1

  // BCC2 = P1^P2^...^Pn
  unsigned char bcc2 = buf[0];
  for (int k = 1; k < bufSize; k++)
  {
    bcc2 = bcc2 ^ buf[k];
  }

  // Stuffing
  int j = 4;
  int i = 0;
  while (i < bufSize)
  {
    if (buf[i] == FLAG)
    {
      write_buf[j] = ESC;
      write_buf[j + 1] = 0x5E;
      j += 2;
    }
    else if (buf[i] == ESC)
```

```
  {
    write_buf[j] = ESC;
    write_buf[j + 1] = 0x5D;
    j += 2;
  }
  else
  {
    write_buf[j] = buf[i];
    j++;
  }
  i++;
}
if (bcc2 == FLAG)
{
  write_buf[j] = ESC;
  write_buf[j + 1] = 0x5E;
  j += 2;
}
else if (bcc2 == ESC)
{
  write_buf[j] = ESC;
  write_buf[j + 1] = 0x5D;
  j += 2;
}
else
{
  write_buf[j] = bcc2;
  j++;
}
write_buf[j] = FLAG;
j++;

(void)signal(SIGALRM, alarmHandler);

while (nReTransmissions < maxNReTransmissions && state != 5)
{
  if (alarmEnabled == FALSE)
  {
    STOP = FALSE;
    write(fd, write_buf, j + 1);
    alarm(timeout);
    alarmEnabled = TRUE;

    unsigned char read_byte;
    unsigned char control;
    while (STOP == FALSE)
    {
```

```
        read(fd, &read_byte, 1);

        switch (state)
        {
        case 0:
          if (read_byte == FLAG)
            state = 1;
          else
            state = 0;
          break;
        case 1:
          if (read_byte == FLAG)
            state = 1;
          else if (read_byte == A_RECEIVER)
            state = 2;
          else
            state = 0;
          break;
        case 2:
          if (read_byte == FLAG)
            state = 1;
          else if (read_byte == RR1 || read_byte == RR0 || read_byte == REJ1 ||
read_byte == REJ0)
          {
            control = read_byte;
            state = 3;
          }
          else
            state = 0;
          break;
        case 3:
          if (read_byte == FLAG)
            state = 1;
          else if (read_byte == (A_RECEIVER ^ control))
            state = 4;
          else
            state = 0;
          break;
        case 4:
          if (read_byte == FLAG)
          {
            if (control == RR0 || control == RR1)
            {
              STOP = TRUE;
              alarm(0);
              alarmEnabled = FALSE;
              totalBytesSent += j;
              state = 5;
```

```
            if ((N_local == I0 && control == RR1) || (N_local == I1 && control ==
RR0))
            {
              if (N_local == I1)
                N_local = I0;
              else if (N_local == I0)
                N_local = I1;
            }
          }
          else if (control == REJ0 || control == REJ1)
          {
            STOP = TRUE;
            alarm(0);
            alarmEnabled = FALSE;
            nReTransmissions++;
            totalReTransmissions++;
            state = 0;
            printf("Sent frame got rejected (Retransmission #%d)\n",
nReTransmissions);
          }
        }
        else
          state = 0;
        break;
      default:
        break;
      }
    }
  }
  free(write_buf);

  return nReTransmissions < maxNReTransmissions;
}

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
int llread(unsigned char *packet)
{
  // sleep(1);
  STOP = FALSE;
  state = 0;
  unsigned char response;
  unsigned char writer_nlocal;
  int pos = 0;
  unsigned char read_byte;
  int new_packet = FALSE;
```

```
int escapeRead = FALSE;
unsigned char bcc2 = 0x00;
while (STOP == FALSE)
{
  read(fd, &read_byte, 1);

  switch (state)
  {
  case 0:
    if (read_byte == FLAG)
      state = 1;
    break;
  case 1:
    if (read_byte == A_SENDER)
      state = 2;
    else if (read_byte == FLAG)
      state = 1;
    else
      state = 0;
    break;
  case 2:

    if (read_byte == I0 || read_byte == I1)
    {
      writer_nlocal = read_byte;
      state = 3;
    }
    else if (read_byte == FLAG)
      state = 1;
    else
      state = 0;

    break;
  case 3:
    if (read_byte == (A_SENDER ^ writer_nlocal))
      state = 4;
    else if (read_byte == FLAG)
      state = 1;
    else
      state = 0;
    break;
  case 4:
    if (read_byte == FLAG)
    {
      if (bcc2 == 0x00)
      {
        packet[pos - 1] = '\0';
        STOP = TRUE;
```

```
      if (N_local == I0 && writer_nlocal == I0)
      {
        new_packet = TRUE;
        response = RR1;
        N_local = I1;
      }
      else if (N_local == I1 && writer_nlocal == I1)
      {
        new_packet = TRUE;
        response = RR0;
        N_local = I0;
      }
      else if (N_local == I0 && writer_nlocal == I1)
      {
        response = RR0;
      }
      else if (N_local == I1 && writer_nlocal == I0)
      {
        response = RR1;
      }
    }
    else
    {
      printf("Error in BCC2\n");
      if (N_local == I0)
        response = REJ0;
      else if (N_local == I1)
        response = REJ1;
      pos = 0;
      STOP = TRUE;
    }
  }
  else if (escapeRead)
  {
    if (read_byte == 0x5D)
    {
      packet[pos] = ESC;
      bcc2 = bcc2 ^ ESC;
    }
    else if (read_byte == 0x5E)
    {
      packet[pos] = FLAG;
      bcc2 = bcc2 ^ FLAG;
    }
    escapeRead = FALSE;
    pos++;
  }
```

```
      else if (read_byte == ESC)
      {
        escapeRead = TRUE;
      }
      else
      {
        packet[pos] = read_byte;
        bcc2 = bcc2 ^ read_byte;
        pos++;
      }

      break;
    default:
      break;
    }
  }

  unsigned char write_buf[5] = {0};
  write_buf[0] = FLAG;
  write_buf[1] = A_RECEIVER;
  write_buf[2] = response;
  write_buf[3] = A_RECEIVER ^ response;
  write_buf[4] = FLAG;

  write(fd, write_buf, 5);
  return new_packet;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics)
{
  printf("Starting closing procedures...\n");
  state = 0;
  nReTransmissions = 0;
  alarmEnabled = FALSE;
  STOP = FALSE;
  unsigned char write_buf[5] = {0};
  switch (linkLayerRole)
  {
  case LlTx:;
    write_buf[0] = FLAG;
    write_buf[1] = A_SENDER;
    write_buf[2] = DISC;
    write_buf[3] = A_SENDER ^ DISC;
    write_buf[4] = FLAG;
```

```
(void)signal(SIGALRM, alarmHandler);

while (nReTransmissions < maxNReTransmissions && state != 5)
{
  if (alarmEnabled == FALSE)
  {
    STOP = FALSE;
    write(fd, write_buf, 5);
    alarm(timeout);
    alarmEnabled = TRUE;

    unsigned char read_byte;
    while (STOP == FALSE)
    {
      read(fd, &read_byte, 1);
      switch (state)
      {
      case 0:
        if (read_byte == FLAG)
          state = 1;
        else
          state = 0;
        break;
      case 1:
        if (read_byte == FLAG)
          state = 1;
        else if (read_byte == A_RECEIVER)
          state = 2;
        else
          state = 0;
        break;
      case 2:
        if (read_byte == FLAG)
          state = 1;
        else if (read_byte == DISC)
          state = 3;
        else
          state = 0;
        break;
      case 3:
        if (read_byte == FLAG)
          state = 1;
        else if (read_byte == (A_RECEIVER ^ DISC))
          state = 4;
        else
          state = 0;
        break;
      case 4:
```

```
          if (read_byte == FLAG)
          {
            STOP = TRUE;
            alarm(0);
            state = 5;
          }
          else
            state = 0;
          break;
        default:
          break;
        }
      }
    }
  }

  write_buf[0] = FLAG;
  write_buf[1] = A_SENDER;
  write_buf[2] = UA;
  write_buf[3] = A_SENDER ^ UA;
  write_buf[4] = FLAG;
  write(fd, write_buf, 5);

  break;
case LlRx:;
  unsigned char read_byte;
  while (STOP == FALSE)
  {
    read(fd, &read_byte, 1);
    switch (state)
    {
    case 0:
      if (read_byte == FLAG)
        state = 1;
      else
        state = 0;
      break;
    case 1:
      if (read_byte == FLAG)
        state = 1;
      else if (read_byte == A_SENDER)
        state = 2;
      else
        state = 0;
      break;
    case 2:
      if (read_byte == FLAG)
        state = 1;
```

```
    else if (read_byte == DISC)
      state = 3;
    else
      state = 0;
    break;
  case 3:
    if (read_byte == FLAG)
      state = 1;
    else if (read_byte == (A_SENDER ^ DISC))
      state = 4;
    else
      state = 0;
    break;
  case 4:
    if (read_byte == FLAG)
    {
      STOP = TRUE;
    }
    else
      state = 0;
    break;
  }
}

write_buf[0] = FLAG;
write_buf[1] = A_RECEIVER;
write_buf[2] = DISC;
write_buf[3] = A_RECEIVER ^ DISC;
write_buf[4] = FLAG;

write(fd, write_buf, 5);
state = 0;
nReTransmissions = 0;
alarmEnabled = FALSE;
STOP = FALSE;

while (STOP == FALSE)
{
  read(fd, &read_byte, 1);
  switch (state)
  {
  case 0:
    if (read_byte == FLAG)
      state = 1;
    else
      state = 0;
    break;
  case 1:
```

```
      if (read_byte == FLAG)
        state = 1;
      else if (read_byte == A_SENDER)
        state = 2;
      else
        state = 0;
      break;
    case 2:
      if (read_byte == FLAG)
        state = 1;
      else if (read_byte == UA)
        state = 3;
      else
        state = 0;
      break;
    case 3:
      if (read_byte == FLAG)
        state = 1;
      else if (read_byte == (A_SENDER ^ UA))
        state = 4;
      else
        state = 0;
      break;
    case 4:
      if (read_byte == FLAG)
      {
        STOP = TRUE;
      }
      else
        state = 0;
      break;
    }
  }

  break;
default:
  break;
}

// Restore the old port settings
if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
{
  perror("tcsetattr");
  exit(-1);
}

gettimeofday(&end, 0);
long seconds = end.tv_sec - begin.tv_sec;
```

```
  long microseconds = end.tv_usec - begin.tv_usec;
  double timeElapsed = seconds + microseconds * 1e-6;

  if (showStatistics == TRUE)
  {
    printf("Time elapsed: %f seconds\n", timeElapsed);
    if (linkLayerRole == LlTx)
    {
      printf("Total alarms activated: %d\n", totalAlarms);
      printf("Total frame rejections: %d\n", totalReTransmissions -
totalAlarms);
      printf("Total re-transmissions: %d\n", totalReTransmissions);
      printf("Total bytes sent: %d bytes\n", totalBytesSent);
    }
    else if (linkLayerRole == LlRx)
      printf("Total data bytes received: %d bytes\n",
totalDataBytesReceived);
  }

  close(fd);

  return nReTransmissions < maxNReTransmissions;
}
```