Protocolo de Ligação de Dados

Redes de Computadores - 1º Trabalho Laboratorial

Class 9 group 2

up201906086@edu.fe.up.pt Marcelo Henriques Couto
up201907361@edu.fe.up.pt Francisco Pinto de Oliveira

Dezembro, 2021

Contents

1 Sumário **4**

2 Introdução **4**

3 Arquitetura **4**

4 Casos de Uso Principais **5**
4.1 Identificação 5
4.2 Sequência de Chamadas 5

5 Protocolo de Ligação Lógica **5**
5.1 Leitura da Porta Série 6
5.2 Escrita na Porta Série 8
5.3 Interface para uso na camada de aplicação 8
5.3.1 Estabelecimento da ligação 8
5.3.2 Término da conexão 8
5.3.3 Envio de dados 9
5.3.4 Receção de dados 9

6 Protocolo de Aplicação **9**
6.1 Pacotes de Controlo 9
6.2 Leitura e Escrita de dados 10
6.3 Leitura do ficheiro e construção dos pacotes 10
6.4 Interpretação dos pacotes e escrita do ficheiro 11

7 Validação **11**

8 Eficiência **11**
8.1 Protocolo utilizado 11
8.1.1 Funcionamento do protocolo 12
8.1.2 Mecanismos de deteção de erros 12
8.2 Análise de Dados 12
8.2.1 Definições 12
8.2.2 Gráficos 13

9 Conclusão **15**

10 Anexo 1 **16**
10.1 application.c 16
10.2 application.h 21
10.3 main.c 22
10.4 data _link.c 24
10.5 data _link.h 33
10.6 alarm.c 34
10.7 alarm.h 35

10.8 error.c 35
10.9 error.h 37
10.10macros.h 37
10.11state _machine.c 37
10.12state _machine.h 40
10.13file.c 40
10.14file.h 42

1 Sumário

Este relatório tem pretende descrever a implementação do 1º trabalho laboratorial de Redes de Computadores, expondo detalhes de funcionamento, implementação e eficiência.

O trabalho foi concluído com sucesso e a aplicação é capaz de transmitir ficheiros utilizando a porta série.

2 Introdução

O trabalho desenvolvido teve como objetivo desenvolver um protocolo de transmissão de dados através da porta série. Este relatório pretende descrever o funcionamento e detalhes de implementação.

O relatório será organizado nos seguintes módulos:

- **Arquitetura** – blocos funcionais e interfaces
- **Estrutura do código** – principais funções e estruturas de dados bem como a sua relação com a arquitetura
- **Casos de uso** – identificação e sequencia de chamada de funções
- **Protocolo de ligação lógica** – descrição de funcionamento do protocolo e exemplificação apresentando fragmentos de código
- **Protocolo da aplicação** – descrição do funcionamento da camada e exemplificação apresentando fragmento de código
- **Validação** – descrição dos testes efetuados
- **Eficiência do protocolo de ligação de dados** – caracterização da eficiência do protocolo de ligação de dados
- **Conclusões** – síntese da informação apresentada nas secções anteriores e reflexão sobre objetivos e aprendizagens

3 Arquitetura

Em termos de arquitetura a aplicação está dividida em **três camadas**: **aplicação**, **ligação de dados** e **protocolo**. Para além disso, ainda há **dois modos de funcionamento**: **emissor e recetor**. As camadas traduzem-se em módulos de código, mas os modos de funcionamento não. Isto acontece porque o utilizador utiliza o mesmo executável tanto para o emissor quanto para o recetor e o utilizador escolhe o modo de funcionamento utilizando argumentos de linha de comandos.

- A **camada da aplicação** é a responsável leitura/criação do novo ficheiro. O comportamento desta camada tem de ser especificado para determinar o seu comportamento
- A **camada de ligação de dados** é a responsável tanto por ler e escrever na porta série quanto por lidar com possíveis erros na transmissão das tramas. Esta camada não tem qualquer autonomia, só realiza as ações de transmissão e emissão quando solicitada
- A **camada de protocolo** já está implementada e a interação é feita através de system calls

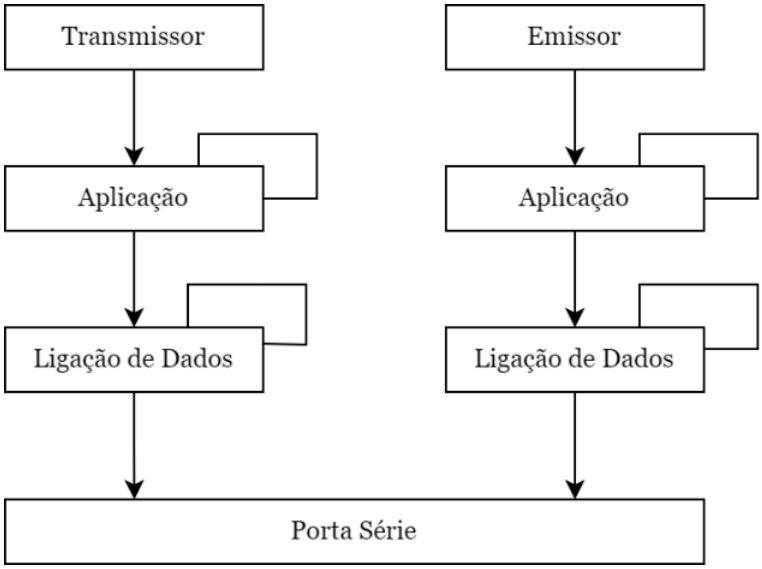


Figure 2: Arquitetura da aplicação

4 Casos de Uso Principais

4.1 Identificação

Para utilizar o programa o utilizador deve compilá-lo. Para isso deve executar o comando "make". Se o código já tiver sido previamente compilado deve ser executado "make clean".

Para transmitir um ficheiro o utilizador deve possuir dois terminais ligados por uma porta série.

- Comando a executar no terminal do transmissor - ./rcompy -t -s *caminho para a porta série* -f *caminho para o ficheiro*
- Comando a executar no terminal do recetor - ./rcompy -r -s *caminho para a porta série* -f *caminho para o novo ficheiro*
- Opções extra como -x *frequência de erros nos dados* ou -z *frequência de erros no cabeçalho* e -p *tempo de propagação* podem ser acrescentadas ao recetor para simulação de erros e tempo de propagação
- Para transmitir um ficheiro o utilizador deve possuir dois terminais ligados por uma porta série.

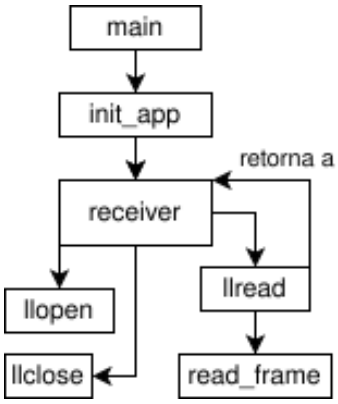
4.2 Sequência de Chamadas

A sequência de chamada a funções de cada componente da aplicação é apresentada nos seguintes diagramas.

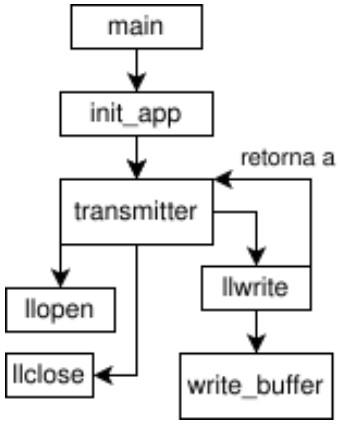
Apenas foram apresentadas as funções mais importantes, o fluxo completo pode ser obtido analisando o código e os seus comentários.

5 Protocolo de Ligação Lógica

Os objetivos principais da camada de ligação de dados são:



(a) Esquema de Chamadas do Recetor



(b) Esquema de Chamadas do Transmissor

Figure 3: Esquema de Chamadas do Programa

1. Redução e controlo dos erros de transmissão
2. Regulação do fluxo de dados
3. Disponibilização de uma interface para comunicação através da porta série à camada de aplicação

5.1 Leitura da Porta Série

Para leitura da porta série adaptada da máquina de estados fornecida, para suportar frames de dados, e tem como intuito ler as tramas recebidas na porta e verificar a sua validade.

A máquina de estados lê dados da porta série byte a byte. Depois da receção do BCC1, se não for detetada uma FLAG, sabemos que estamos perante uma frame de informação. Os dados são colocados num buffer e a leitura acaba quando é encontrada uma FLAG. O buffer contém os dados até à posição N - 1 pois o dado de posição N é o BCC2. Em seguida apresenta-se a sua implementação em pseudocódigo.

```
1    buffer <- Empty
2    While state != STOP:
3        byte_rcvd <- read_from_serial_port
4        if (state == START) then:
5            if byte_rcvd == FLAG then:
6                state <- FLAG_RCVD
7        if (state == FLAG_RCVD):
8            if (isA(byte_rcvd)) then:
9                state <- A_RCVD
10       if (state == A_RCVD) then:
11           if (isC(byte_rcvd)) then:
12               state <- A_RCVD
13           ElseIf (byte_rcvd == FLAG) then:
14               state = FLAG_RCVD
15           Else:
16               state = START
17       if (state == C_RCVD) then:
18           if (validBCC1()) then:
19               state = BCC_OK
```

```
            ElseIf (byte_rcvd == FLAG) then:
                state = FLAG_RCVD
            Else:
                state = START
        if (state == BCC_OK) then:
            if (byte_rcvd == FLAG_RCVD) then:
                state <- STOP
            Else:
                Insert(buffer, byte_rcvd)
                state = DATA
        if (state == DATA) then:
            if (byte_rcvd == FLAG) then:
                state = STOP
            Else:
                Insert(buffer, byte_rcvd)
```

Listing 1: Pseudocódigo da StateMachine

Quando a verificação do BCC1 falha não se envia qualquer resposta uma vez que é impossível determinar o tipo de frame recebido.

Cada frame é lido num loop para o caso de haver erros na parte de informação e ser necessário receber outro frame. Se o BCC2 for válido e o frame corresponder ao número esperado um RR é enviado senão é enviado REJ e aguarda-se pelo novo frame.

```
while(!valid_frame) {
    new_frame = read_frame(fd, link_layer_data.frame, MAX_FRAME_SIZE);
    frame_size = destuff(link_layer_data.frame, new_frame.packet_size, aux_buffer,
    MAX_PACKET_SIZE + 1);

    if (frame_size == 0) {
        fprintf(stderr, "Error in data_link - llread: destuffing packet unsuccessfull\n"
    );
        return -1;
    }
    uint8_t bcc2 = aux_buffer[--frame_size];

    // Error
    if (new_frame.frame_type != expected_package || !validBCC2(bcc2, aux_buffer,
    frame_size)) {
        if (assemble_supervision_frame(expected_package ? REJ_1 : REJ_0, RECEIVER,
    response_frame) == -1) return -1;
    } else {
        memcpy(buffer, aux_buffer, frame_size);
        if (assemble_supervision_frame(expected_package ? RR_0 : RR_1, RECEIVER,
    response_frame) == -1) return -1;
        valid_frame = true;
    }
    if (write(fd, response_frame, SUPV_FRAME_SIZE) != 5) {
        fprintf(stderr, "Error in data_link - llread: did not write response frame
    correctly");
        return -1;
    }
}
```

Listing 2: Código em C do loop responsável pela leitura da porta série

5.2 Escrita na Porta Série

Quando um buffer é escrito, o utilizador da função deve especificar uma resposta esperada pelo outro lado da porta série. Quando o buffer é enviado, é acionado um alarme. Se o alarme for chamado uma flag é ativada e retira-se um valor ao número de tentativas de leitura. Se o número de tentativas for esgotado, assume-se que a trama não foi enviada.

```
/* Writes a buffer to the serial port and awaits a certain response
    fd -> Serial port file desciptor
    buffer -> buffer to write
    length -> length of the buffer to be written
    expected_response -> expected response sent by the other end of the serial port
*/
int write_buffer(int fd, uint8_t* buffer, size_t length, FrameType expected_response)
```

Listing 3: Código em C da função responsável pela escrita na porta série

5.3 Interface para uso na camada de aplicação

O programa foi desenhado de modo a manter independência entre a camada de aplicação e o protocolo de ligação de dados, de modo que este último é apenas visível à camada superior como 4 pontos de entrada.

5.3.1 Estabelecimento da ligação

Para o estabelecimento de ligação através de uma porta série, a função disponibilizada **llopen**.

- Emissor - configura-se a porta série, envia-se uma trama **SET** (pelo emissor) e aguarda-se uma trama **UA** que será enviada pelo recetor
- Recetor - configura-se a porta série, aguarda-se a receção de uma trama **SET**; quando esta é recebida corretamente, envia-se uma trama **UA** de modo a marcar o estabelecimento de ligação

```
int llopen(char* port, ConnectionType role)
```

Listing 4: llopen header

5.3.2 Término da conexão

Para o fecho da ligação, o protocolo de ligação de dados fornece a função **llclose**.

- Emissor - envia-se uma trama **DISC**, aguarda-se por uma trama **DISC** e finalmente envia-se uma trama **UA**.
- Recetor - aguarda-se uma trama **DISC**, envia-se uma trama **DISC** e finalmente aguarda-se uma trama **UA**.

```
int llclose(int fd)
```

Listing 5: llclose header

5.3.3 Envio de dados

Para o envio de dados (pela parte do emissor), existe a função **llwrite**.
Esta função apenas invoca a função `write_buffer` descrita acima, além de controlar o número de sequência da trama.

5.3.4 Receção de dados

Para a receção de dados (pela parte do recetor), é providenciada a função **llread**.
Esta função invoca a `state_machine` e controla o processo de deteção de erros descritos acima.

6 Protocolo de Aplicação

O Protocolo de Aplicação implementado tem como objetivos:

1. Envio, receção e construção dos pacotes de controlo de início e fim de transmissão
2. Leitura/Escrita de dados de/para uma porta série utilizando o protocolo de ligação lógica
3. Construção/Interpretação de pacotes de dados com *headers* válidos
4. Divisão de um ficheiro em fragmentos e posterior envio (transmissor)
5. Construção de um ficheiro através de dados recebidos (recetor)

6.1 Pacotes de Controlo

Os pacotes de controlo de transmissão são construídos pelo transmissor para denotar o inicio e fim de transmissão, bem como o envio de informação sobre os dados a serem enviados, tal como o tamanho do ficheiro e o seu nome.

```
uint8_t* build_control_packet(size_t* control_packet_size, char* file_name)
```

Listing 6: Função que constrói o pacote de controlo (transmissor)

```
void parse_control_packet(uint8_t* packet)
```

Listing 7: Função que interpreta o pacote de controlo recebido (recetor)

O tamanho do ficheiro, enviado no inicio da ligação no pacote de controlo, é usado para verificar, no fim da ligação, se o número de bytes recebidos corresponde ao esperado.

```
    if (bytes_received != app_data.transfer_file_size) {
        fprintf(stderr, "File size does not correspond to the expected\n");
    }
```

Listing 8: Verificação do tamanho do ficheiro recebido (receiver)

6.2 Leitura e Escrita de dados

A leitura e escrita de dados são ambos efetuados em loops nas funções principais do emissor e recetor, **transmitter** e **receiver** respetivamente.

```
while (bytes_read < app_data.transfer_file_size) {
    packet_no++;
    bytes_written = read_data_packet(packet_no, data_packet, oldFileDescriptor,
    bytes_read);
    bytes_read += bytes_written - 4;
    if ((llwrite(app_data.file_descriptor, data_packet, bytes_written)) == -1) exit(1);
    printf("Transmitter - packet %d sent\n", packet_no);
}
```

Listing 9: Envio de pacotes de dados

```
    while (true) {
        // READ
        if (llread(app_data.file_descriptor, data_packet, MAX_PACKET_SIZE) == -1) exit
    (1);
        if (data_packet[0] == 0x01) {
            bytes_received += write_data_packet(data_packet, newFileDescriptor, &
    packet_no);
            printf("Receiver - packet %d received\n", (int) data_packet[1]);
        } else if (data_packet[0] == 0x02) {
            parse_control_packet(data_packet);
            if (file_exists(app_data.transfer_file_dir, app_data.transfer_file_name)) {
                fprintf(stderr, "Error in application - receiver: file already exists\n"
    );
                exit(1);
            }
            newFileDescriptor = open_transfer_file();
            fileOpen = true;

        } else if (data_packet[0] == 0x03) {
            // CLOSE
            if ((llclose(app_data.file_descriptor)) != 0) exit(1);
            printf("Receiver - connection cut\n");
            break;
        } else {
            fprintf(stderr, "Error in application - receiver: C byte value invalid\n");
            exit(1);
        }
    }
```

Listing 10: Leitura dos pacotes de dados

Nos pacotes de dados possuem numeração, que permitem avisar qualquer inconsitência nos pacotes recebidos. Esta verificação é feita na função `read_data_packet`. **Nota:** esta funcionalidade foi implementada posteriormente à apresentação.

6.3 Leitura do ficheiro e construção dos pacotes

A leitura do ficheiro do **emissor** e subsequente construção do pacote de dados é feita incrementalmente, sendo a seguinte função chamada no loop de escrita referido anteriormente.

```
size_t read_data_packet(int packetNo, uint8_t* data_packet, int fd, int bytes_read)
```

Listing 11: Função que lê o ficheiro e constrói o pacote de dados

6.4 Interpretação dos pacotes e escrita do ficheiro

O **recetor** executa a interpretação e verificação dos dados recebidos e a escrita dos mesmos num ficheiro numa mesma função, chamada iterativamente no loop acima descrito.

```
size_t write_data_packet(uint8_t* data_packet, int fd, int* packet_no)
```

Listing 12: Função que interpreta o pacote de dados recebido e escreve no ficheiro

7 Validação

De forma a testar a aplicação e o protocolo desenvolvidos, foram executados os seguintes testes:

- Envio de ficheiros de diferentes tamanhos
- Interrupção da ligação por cabo entre as portas série
- Geração de ruído na ligação através de um curto circuito
- Interrupção momentânea da ligação a meio de um envio
- Envio de ficheiros com diferentes percentagens de erros simulados
- Variação dos tempos de propagação simulados
- Envio de ficheiros com diferentes tamanhos das tramas de Informação
- Envio de ficheiros para diferentes valores de capacidade de ligação (Baudrate)

O sucesso destes testes permitiu-nos concluir que o protocolo e a aplicação funcionam como esperado.

8 Eficiência

8.1 Protocolo utilizado

O protocolo implementado baseia-se num sistema *Stop-and-Wait ARQ*. O seu nome advém da natureza do comportamento do emissor e do recetor:

- **Stop-and-Wait** porque o emissor espera pela resposta do recetor a cada trama
- **ARQ ou Automatic Repeat Request** porque o recetor automaticamente requer a repetição do envio da trama se esta contiver erros ou não corresponder ao esperado

8.1.1 Funcionamento do protocolo

1. O emissor envia uma trama e espera pela resposta do recetor
2. - Se o recetor receber a trama que espera, envia uma mensagem de acknowledgement ao emissor 'ACK'
   - Caso receba uma trama que não corresponde à esperada ou que contenha erros, envia uma mensagem de rejeição, de modo a que o emissor reenvie a mensagem 'NACK'
3. - Recomeça o processo para a próxima trama ou
   - Reenvia a trama

8.1.2 Mecanismos de deteção de erros

Através da numeração das tramas com os valores 0 ou 1, o recetor é capaz de distinguir se a trama recebida é aquela que se pretende ou se é apenas uma retransmissão ou se houve um lapso da trama.

BCC1 e BCC2 são dois componentes da trama cuja função é detetar um eventual bit flip no cabeçalho ou conteúdo da trama, respetivamente.

8.2 Análise de Dados

De modo a entender de que forma afetam a performance do protocolo alguns dos seus parâmetros, foi executada uma recolha e análise de dados.

O nosso protocolo baseia-se num sistema *Stop-and-Wait ARQ*.

8.2.1 Definições

Descrições

- **S** - eficiência
- **FER** ou $\boldsymbol{p_e}$ - probabilidade de erro numa trama
- **R** ou $\boldsymbol{T_f}$ - débito recebido
- $\boldsymbol{T_prop}$ - tempo de propagação da trama

Formulas

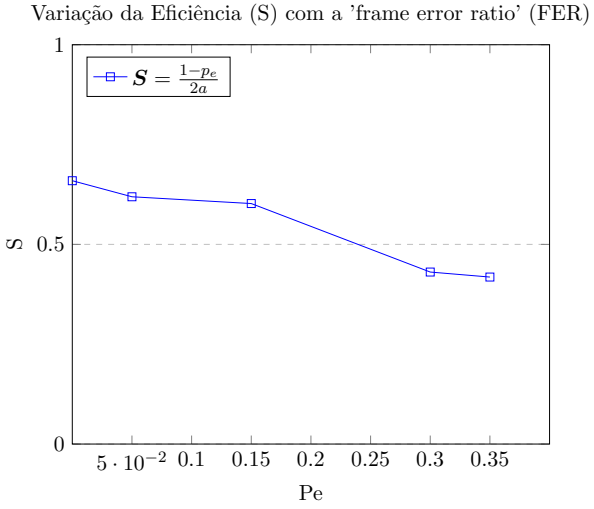
- $\boldsymbol{a} = \frac{T_prop}{T_f}$
- **Sem erros:** $\boldsymbol{S} = \frac{T_f}{T_prop+T_f+T_prop} = \frac{1}{2a}$
- **Com erros:** $\boldsymbol{S} = \frac{1-p_e}{2a}$

8.2.2 Gráficos

Para a recolha de dados, os valores por defeito são:

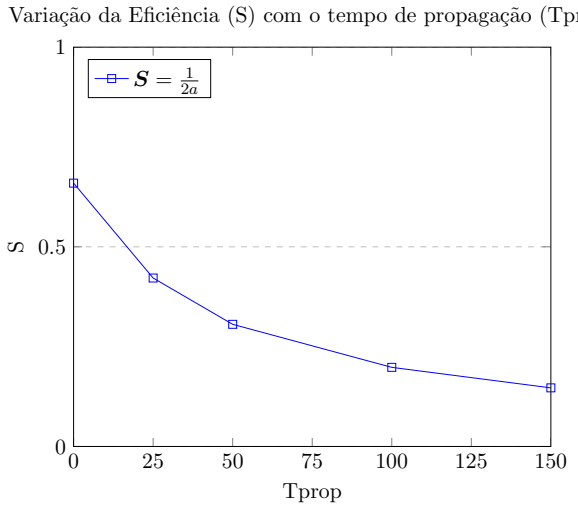
- $Baudrate = 38400$
- $Framesize = 150B$
- $T_prop = 0$
- $FER = 0$

FER Os resultados da nossa análise indicam-nos que as fórmulas a cima descritas, no que toca à influência dos erros na trama na eficiência do protocolo, estão corretos. Estas conclusões são suportadas pelo gráfico abaixo.

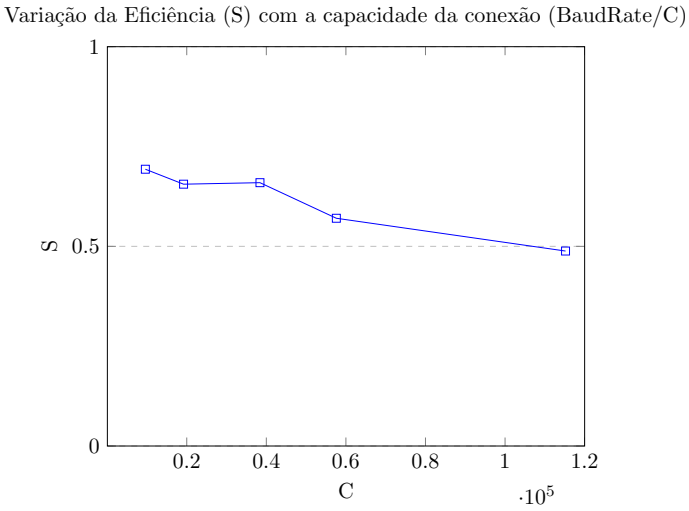


Como a regressão dos dados formou uma função linear, estes vão de encontro à fórmula prevista. A linearidade da função não é muito definida devido ao curto tamanho da amostra, que foi recolhida em laboratório em tempo limitado.

Tempo de propagação O próximo gráfico mostra a variação da eficiência consoante o tempo de propagação, o que valida as fórmulas no sentido de que S varia em proporcionalidade inversa com a.

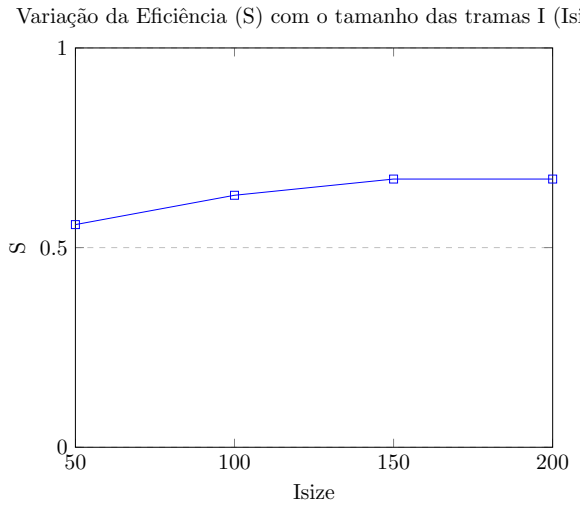


Baudrate



O gráfico permite-nos concluir que a eficiência do protocolo diminui com o aumento da capacidade da ligação de dados. Estima-se que esta viesse a subir até um dado valor. No entanto, infelizmente, essa subida não está representada na gama de valores que foram recolhidos.

Tamanho de trama



Por último conclui-se que o tamanho das tramas beneficia a eficiência mas apenas até um certo ponto, começando a estancar por volta dos 150 bytes.

9 Conclusão

Este trabalho tinha como objetivo o desenvolvimento de um protocolo de ligação de dados e de um protocolo de aplicação para o seu uso, que permitissem a comunicação e transmissão de dados entre dois computadores através de uma porta série.

O protocolo de ligação lógica é responsável pela preparação da ligação à porta série e pelo tratamento de erros. Este predispõe apenas de quatro pontos de entrada e mantém abstração e independência das camadas superiores (camada da aplicação).

A camada da aplicação tem como intuito a interpretação de argumentos e opções para a execução do programa, além da leitura e escrita do conteúdo no ficheiro.

Dados estes objetivos para o projeto, consideramos que o trabalho cumpre com os requisitos estipulados, sendo portanto uma ferramenta viável para o envio de dados entre dois computadores.

10 Anexo 1

Modules

- **application** - camada de aplicação
- **main** - interpretação dos argumentos de chamada
- **file** - interpretação e validação do nome de ficheiro dado
- **error** - simulação de erros e do tempo de propagação
- **state** machine - máquina de estados
- **data** link - protocolo de ligação de dados
- **macros** - constantes

10.1 application.c

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <string.h>
#include <stdlib.h>
#include <stdint.h>
#include "file.h"
#include "data_link.h"
#include "application.h"

typedef struct {
    int file_descriptor;
    size_t transfer_file_size;
    ConnectionType status;
    char* transfer_file_name;
    char* transfer_file_dir;
    char* transfer_file_path;
} ApplicationLayer;

ApplicationLayer app_data;

int receiver(char* port_path);
int transmitter(char* port_path);

// RECEIVER AUX
size_t write_data_packet(uint8_t* data_packet, int fd, int* packet_no);
void parse_control_packet(uint8_t* packet);
int open_transfer_file();

// TRANSMITTER AUX
uint8_t* build_control_packet(size_t* control_packet_size, char* file_name);
size_t read_data_packet(int packetNo, uint8_t* data_packet, int fd, int bytes_read);
int read_file(char* file_path, uint8_t* buffer, size_t size);
```

```
35
36
37 void init_app(ConnectionType connection_type, char* file_path, char* serial_path) {
38     app_data.transfer_file_name = calloc(FILE_NAME_SIZE, sizeof(char));
39     app_data.transfer_file_dir = calloc(DIR_PATH_SIZE, sizeof(char));
40     app_data.status = connection_type;
41
42     path_parser(file_path, app_data.transfer_file_dir, app_data.transfer_file_name);
43
44     if (connection_type == TRANSMITTER) {
45         app_data.transfer_file_path = file_path;
46         transmitter(serial_path);
47     }
48     if (connection_type == RECEIVER) {
49         app_data.transfer_file_path = calloc(DIR_PATH_SIZE + FILE_NAME_SIZE, sizeof(char
    ));
50         receiver(serial_path);
51         free(app_data.transfer_file_path);
52     }
53
54     free(app_data.transfer_file_dir);
55     free(app_data.transfer_file_name);
56 }
57
58 void parse_control_packet(uint8_t* packet) {
59     int i = 1;
60     if (packet[i++] == 0x00) { // Filesize
61         size_t file_size = 0;
62         int n_bytes = (int) packet[i++];
63         n_bytes--;
64         while(n_bytes >= 0) {
65             file_size += (size_t) packet[i++] << (n_bytes * 8);
66             n_bytes--;
67         }
68         app_data.transfer_file_size = file_size;
69     } else {
70         fprintf(stderr, "Error in application - parse_control_packet: unexpected
    structure of the control_packet\n");
71         exit(1);
72     }
73     if (packet[i++] == 0x01) { // Filename
74         int n_bytes = (int) packet[i++];
75         if (strlen(app_data.transfer_file_name) == 0) {
76             strncpy(app_data.transfer_file_name, (char*) packet + i, n_bytes);
77         }
78         strncpy(app_data.transfer_file_path, app_data.transfer_file_dir, DIR_PATH_SIZE);
79         strcat(app_data.transfer_file_path, app_data.transfer_file_name);
80     } else {
81         fprintf(stderr, "Error in application - parse_control_packet: unexpected
    structure of the control_packet\n");
82         exit(1);
83     }
84 }
85
86 size_t write_data_packet(uint8_t* data_packet, int fd, int* packet_no) {
```

```
87
88      size_t packet_size = (size_t) (data_packet[2] << 8) + data_packet[3];
89      if (*packet_no == (int) data_packet[1]) {
90          (*packet_no)++;
91      } else {
92          fprintf(stderr, "Packet sequence number incorrect! Missed a packet\n");
93          (*packet_no) = (int) data_packet[1] + 1;
94      }
95
96      if ((write(fd, data_packet + 4, packet_size * sizeof(uint8_t))) != packet_size) {
97          perror("Error in application - write_data_packet");
98          exit(1);
99      }
100     return packet_size;
101 }
102
103 int receiver(char* port_path) {
104
105     int newFileDescriptor, packet_no = 0;
106     size_t bytes_received = 0;
107     bool fileOpen = false;
108     uint8_t *data_packet;
109     data_packet = calloc(MAX_PACKET_SIZE, sizeof(uint8_t));
110
111     // OPEN
112     if((app_data.file_descriptor = llopen(port_path, RECEIVER)) == -1) exit(1);
113
114     printf("Receiver - connection established\n");
115
116     while (true) {
117         // READ
118         if (llread(app_data.file_descriptor, data_packet, MAX_PACKET_SIZE) == -1) exit
    (1);
119         if (data_packet[0] == 0x01) {
120             bytes_received += write_data_packet(data_packet, newFileDescriptor, &
    packet_no);
121             printf("Receiver - packet %d received\n", (int) data_packet[1]);
122         } else if (data_packet[0] == 0x02) {
123             parse_control_packet(data_packet);
124             if (file_exists(app_data.transfer_file_dir, app_data.transfer_file_name)) {
125                 fprintf(stderr, "Error in application - receiver: file already exists\n"
    );
126                 exit(1);
127             }
128             newFileDescriptor = open_transfer_file();
129             fileOpen = true;
130
131         } else if (data_packet[0] == 0x03) {
132             // CLOSE
133             if ((llclose(app_data.file_descriptor)) != 0) exit(1);
134             printf("Receiver - connection cut\n");
135             break;
136         } else {
137             fprintf(stderr, "Error in application - receiver: C byte value invalid\n");
138             exit(1);
```

```
        }
    }

    if (bytes_received != app_data.transfer_file_size) {
        fprintf(stderr, "File size does not correspond to the expected\n");
    }

    if (fileOpen) {
        if ((close(newFileDescriptor) == -1)) {
            perror("Error in application - receiver");
            exit(1);
        }
    }
    free(data_packet);

    return 0;
}

size_t read_data_packet(int packetNo, uint8_t* data_packet, int fd, int bytes_read) {
    size_t bytes_to_write = (MAX_PACKET_SIZE > (app_data.transfer_file_size - bytes_read + 4)) ? (app_data.transfer_file_size - bytes_read) : MAX_PACKET_SIZE - 4; // In case of the file being in its end
    data_packet[0] = 0x01;
    data_packet[1] = (uint8_t) packetNo;
    data_packet[2] = (uint8_t) (bytes_to_write & 0xFF00) >> 8; // Amount of data
    data_packet[3] = (uint8_t) bytes_to_write & 0x00FF;
    if ((read(fd, data_packet + 4, bytes_to_write * sizeof(uint8_t))) != bytes_to_write) {
        perror("Error in application - read_data_packet");
        exit(1);
    }
    return bytes_to_write + 4;
}

uint8_t* build_control_packet(size_t* control_packet_size, char* file_name) {
    size_t file_name_size = strlen(file_name) + 1; //To actually copy \0

    if (file_name_size > MAX_PACKET_SIZE - 13) {
        fprintf(stderr, "Error in application - build_control_packet: filename too big\n");
        exit(1);
    }

    *control_packet_size = file_name_size + 13;
    uint8_t* buffer = calloc((*control_packet_size), sizeof(uint8_t));

    buffer[0] = 0x02; // C
    buffer[1] = 0; // T
    buffer[2] = (uint8_t) sizeof(app_data.transfer_file_size); // L
    buffer[3] = (app_data.transfer_file_size & 0xFF00000000000000) >> 56; // filesize
    buffer[4] = (app_data.transfer_file_size & 0x00FF000000000000) >> 48;
    buffer[5] = (app_data.transfer_file_size & 0x0000FF0000000000) >> 40;
    buffer[6] = (app_data.transfer_file_size & 0x000000FF00000000) >> 32;
    buffer[7] = (app_data.transfer_file_size & 0x00000000FF000000) >> 24;
    buffer[8] = (app_data.transfer_file_size & 0x0000000000FF0000) >> 16;
```

```
190     buffer[9] = (app_data.transfer_file_size & 0x000000000000FF00) >> 8;
191     buffer[10] = app_data.transfer_file_size & 0x00000000000000FF;
192     buffer[11] = 1; // T
193     buffer[12] = (uint8_t) (sizeof(char) * file_name_size); // L
194     memcpy((char*) buffer + 13, file_name, file_name_size); // filename
195     return buffer;
196 }
197
198 int transmitter(char* port_path) {
199     uint8_t *control_packet, *data_packet;
200     struct stat s;
201     size_t control_packet_size, bytes_read = 0, bytes_written;
202     int packet_no = 0, oldFileDescriptor;
203
204     if (stat(app_data.transfer_file_path, &s) == -1) {
205         perror("Error in application - transmitter - stat");
206         exit(1);
207     }
208     app_data.transfer_file_size = s.st_size;
209     data_packet = calloc(MAX_PACKET_SIZE, sizeof(uint8_t));
210
211     oldFileDescriptor = open_transfer_file();
212
213     // OPEN
214     if( (app_data.file_descriptor = llopen(port_path, TRANSMITTER)) == -1) exit(1);
215     control_packet = build_control_packet(&control_packet_size, app_data.
    transfer_file_name);
216
217     printf("Transmitter - connection established\n");
218
219     // START
220     if ((llwrite(app_data.file_descriptor, control_packet, control_packet_size)) == -1)
    exit(1);
221
222     printf("Transmitter - transmition begun\n");
223
224     // WRITING
225     while (bytes_read < app_data.transfer_file_size) {
226         packet_no++;
227         bytes_written = read_data_packet(packet_no, data_packet, oldFileDescriptor,
    bytes_read);
228         bytes_read += bytes_written - 4;
229         if ((llwrite(app_data.file_descriptor, data_packet, bytes_written)) == -1) exit
    (1);
230         printf("Transmitter - packet %d sent\n", packet_no);
231     }
232
233     // FINISH
234     control_packet[0] = 0x03;
235     if ((llwrite(app_data.file_descriptor, control_packet, control_packet_size)) == -1)
    {
236         fprintf(stderr, "Error in application - transmitter - fatal error when writting
    control packet\n");
237         exit(1);
238     }
```

```
239
240     printf("Transmitter - transmition ended\n");
241
242     // CLOSE
243     if ((llclose(app_data.file_descriptor)) != 0) exit(1);
244
245     printf("Transmitter - connection cut\n");
246
247     if ((close(oldFileDescriptor) == -1)) {
248         perror("Error in application - transmitter");
249         exit(1);
250     }
251
252     free(data_packet);
253     free(control_packet);
254
255     return 0;
256 }
257
258 int open_transfer_file() {
259     int fd;
260
261     switch (app_data.status) {
262         case TRANSMITTER: {
263             if ((fd = open(app_data.transfer_file_path, O_RDONLY)) == -1) {
264                 perror("Error in application - transmitter - open_transfer_file");
265                 exit(1);
266             }
267             break;
268         }
269         case RECEIVER: {
270             if ((fd = creat(app_data.transfer_file_path, S_IRUSR | S_IWUSR | S_IRGRP |
    S_IROTH)) == -1) {
271                 perror("Error in application - receiver - open_transfer_file");
272                 exit(1);
273             }
274             break;
275         }
276
277         default: {
278             fprintf(stderr, "Error in application - open_transfer_file: wrong type of
    connection\n");
279             exit(1);
280         }
281     }
282     return fd;
283 }
```

Listing 13: application.c

10.2 application.h

```
1 #pragma once
2
3 #include "macros.h"
```

```
4
5  void init_app(ConnectionType connection_type, char* file_path, char* serial_path);
```

Listing 14: application.h

10.3 main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4  #include <getopt.h>
5  #include <time.h>
6  #include "error.h"
7  #include "state_machine.h"
8  #include "application.h"
9
10 void print_cmd_args() {
11     printf("Usage: \n");
12     printf("    rcompy -r/-t --file <argument> --serial_port <argument> [--fer <argument
       > --t_prop <argument> --help]\n");
13     printf("Options: \n");
14     printf("    -r               Receiver mode\n");
15     printf("    -t               Transmitter mode\n");
16     printf("    -f --file        Choose destination if receiver or source if transmitter
       \n");
17     printf("    -s --serial_port Choose serial port path\n");
18     printf("    -z --header_error_ratio        Choose error generation probability in
       information packets\n");
19     printf("    -x --data_error_ratio         Choose error generation probability in
       information packets\n");
20     printf("    -p --t_prop      Choose propagation delay in milliseconds\n");
21     printf("    -h --help        Show this help message\n");
22 }
23
24 // Just parsing
25 int main(int argc, char** argv) {
26
27     srandom(time(NULL));
28     struct option options[] = {
29         {"file", required_argument, 0, 'f'},
30         {"serial_port", required_argument, 0, 's'},
31         {"header_error_ratio", required_argument, 0, 'z'},
32         {"data_error_ratio", required_argument, 0, 'x'},
33         {"t_prop", required_argument, 0, 'p'},
34         {"help", required_argument, 0, 'h'},
35         {0, 0, 0, 0}
36     };
37
38     bool is_trans, is_rec, had_file_path, had_serial_path, had_header_error,
       had_data_error, had_tprop;
39     is_trans = is_rec = had_file_path = had_serial_path = had_header_error =
       had_data_error = had_tprop = false;
40
41     char* file_path;
42     char* serial_port_path;
```

```
43      int t_prop = 0;
44      double header_error = 0;
45      double data_error = 0;
46
47      int c = 0;
48      int opt_index = 0;
49      while((c = getopt_long(argc, argv, "htrf:p:x:z:s:", options, &opt_index)) != -1) {
50          switch (c) {
51              case 't': {
52                  is_trans = true;
53                  if (is_rec) {
54                      print_cmd_args();
55                      return 1;
56                  }
57                  break;
58              }
59              case 'r': {
60                  is_rec = true;
61                  if (is_trans) {
62                      print_cmd_args();
63                      return 1;
64                  }
65                  break;
66              }
67              case 'f': {
68                  if (had_file_path) {
69                      print_cmd_args();
70                      return 1;
71                  }
72                  had_file_path = true;
73                  file_path = optarg;
74                  break;
75              }
76              case 's': {
77                  if (had_serial_path) {
78                      print_cmd_args();
79                      return 1;
80                  }
81                  had_serial_path = true;
82                  serial_port_path = optarg;
83                  break;
84              }
85              case 'p': {
86                  if (had_tprop) {
87                      print_cmd_args();
88                      return 1;
89                  }
90                  had_tprop = true;
91                  t_prop = atoi(optarg);
92                  break;
93              }
94              case 'z': {
95                  if (had_header_error) {
96                      print_cmd_args();
97                      return 1;
```

```
                }
                had_header_error = true;
                header_error = strtod(optarg, NULL);
                break;
            }
            case 'x': {
                if (had_data_error) {
                    print_cmd_args();
                    return 1;
                }
                had_data_error = true;
                data_error = strtod(optarg, NULL);
                break;
            }
            case 'h': {
                print_cmd_args();
                return 1;
            }
            default: {
                print_cmd_args();
                return 1;
            }
        }
    }
    if (!had_file_path || !had_serial_path || (!is_rec && !is_trans)) {
        print_cmd_args();
        return 1;
    }

    set_error_rates(header_error, data_error);
    set_prop_time(t_prop);
    init_app(is_rec ? RECEIVER : TRANSMITTER, file_path, serial_port_path);

    return 0;
}
```

Listing 15: main.c

10.4 data_link.c

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <strings.h>
#include <termios.h>
#include <stdbool.h>
#include "data_link.h"
#include "alarm.h"
#include "state_machine.h"
#include "macros.h"


static struct termios old_terminal;
static struct LinkLayerData link_layer_data;
```

```
16 static uint8_t* aux_buffer;
17 static bool alarm_called = false;
18
19 int setup_terminal(int fd);
20 int restore_terminal(int fd);
21
22 int assemble_supervision_frame(FrameType frame_type, ConnectionType connection_type,
       uint8_t* frame_buffer);
23 uint32_t assemble_information_frame(uint8_t* packet_data, size_t packet_size, uint8_t*
       frame, uint8_t frame_number);
24 int write_buffer(int fd, uint8_t* buffer, size_t length, FrameType expected_response);
25 bool validBCC2(uint8_t bcc2, uint8_t* buffer, size_t buffer_size);
26 uint8_t compute_bcc2(uint8_t* buffer, size_t buffer_size);
27
28 int establish_transmitter_connection(int fd);
29 int wait_transmitter_connection(int fd);
30 int disconnect_transmitter(int fd);
31 int disconnect_receiver(int fd);
32
33 void dl_alarm_callback() {
34     alarm_called = true;
35 }
36
37 int llopen(char* port, ConnectionType role) {
38
39     int fd = 0;
40     alarm_called = false;
41
42     // Configure port
43     snprintf(link_layer_data.port, sizeof(char)*MAX_PORT_SIZE, "%s", port);
44     link_layer_data.role = role;
45     link_layer_data.baud_rate = BAUDRATE;
46     link_layer_data.num_transmissions = NUM_TX;
47     link_layer_data.timeout = TIMEOUT;
48
49     if ((link_layer_data.frame = calloc(MAX_FRAME_SIZE, sizeof(uint8_t))) == NULL) {
50         return -1;
51     }
52     if ((aux_buffer = calloc(MAX_FRAME_SIZE + 1, sizeof(uint8_t))) == NULL) {
53         return -1;
54     }
55
56     fd = open(link_layer_data.port, O_RDWR | O_NOCTTY );
57     if (fd < 0) {
58         perror("Error in data_link - llopen - opening port");
59         return -1;
60     }
61
62     if (setup_terminal(fd) == -1) {
63         perror("Error in data_link - llopen");
64         return -1;
65     }
66
67     if (setup_alarm_handler() == -1) {
68         fprintf(stderr, "Error in data_link - llopen: setting up alarm handler\n");
```

```
69          return -1;
70      }
71
72      if (subscribe_alarm(dl_alarm_callback) == -1) { // Subscribing a specific function
        to be called in alarm
73          fprintf(stderr, "Error in data_link - llopen: subscribing alarm event\n");
74          return -1;
75      }
76
77      switch(role) {
78          case TRANSMITTER: {
79              if ((establish_transmitter_connection(fd)) == -1) return -1;
80              break;
81          }
82          case RECEIVER: {
83              if ((wait_transmitter_connection(fd)) == -1) return -1;
84              break;
85          }
86          default: {
87              fprintf(stderr, "Error in data_link - llopen: tried to open a connection
        with invalid role\n");
88              return -1;
89          }
90      }
91
92      return fd;
93  }
94
95  int llclose(int fd) {
96      if (link_layer_data.role == TRANSMITTER) {
97          disconnect_transmitter(fd);
98      } else if (link_layer_data.role == RECEIVER) {
99          disconnect_receiver(fd);
100     }
101     if (restore_terminal(fd) == -1) {
102         perror("Error in data_link - llclose");
103         return -1;
104     }
105     if (restore_alarm_handler() == -1) {
106         fprintf(stderr, "Error in data_link - llclose: restoring alarm handlers\n");
107         return -1;
108     }
109     free(aux_buffer);
110     free(link_layer_data.frame);
111     return close(fd);
112 }
113
114 int llwrite(int fd, uint8_t* packet, size_t packet_length) {
115     static uint8_t package_to_send = 0;
116     uint8_t frame_length;
117     FrameType expected_response;
118
119     // Numero de sequ ncia
120     if (package_to_send == 0) {
121         expected_response = RR_1;
```

```
122         frame_length = assemble_information_frame(packet, packet_length, link_layer_data
    .frame, 0);
123     } else {
124         expected_response = RR_0;
125         frame_length = assemble_information_frame(packet, packet_length, link_layer_data
    .frame, 1);
126     }
127 
128     // Writing
129     int res = write_buffer(fd, link_layer_data.frame, frame_length, expected_response);
130     if ( res == -2) {
131         fprintf(stderr, "Error in data_link - llwrite: no. tries exceeded\n");
132         return -1;
133     } else if ( res == -1) {
134         perror("Error in data_link - llwrite");
135         return -1;
136     } else if ( res != frame_length) {
137         fprintf(stderr, "Error in data_link - llwrite: no. bytes written not matching
    expected\n");
138         return -1;
139     }
140     package_to_send = package_to_send ? 0 : 1;
141     return res;
142 }
143 
144 /*
145     This function reads into a buffer a packet that was read from the serial port
146     fd -> Serial port file descriptor
147     buffer -> Buffer that will receive the packet
148     max_buffer_size -> Max size of buffer variable, used to check if a packet can fit
    into the provided buffer
149 */
150 int llread(int fd, uint8_t* buffer, size_t max_buffer_size) {
151     if (max_buffer_size < MAX_PACKET_SIZE) {
152 
153         fprintf(stderr, "Error in data_link - llread: provided buffer does not meet size
     requirements\n");
154         return -1;
155     }
156 
157     size_t frame_size;
158     static uint8_t expected_package;
159     uint8_t response_frame[SUPV_FRAME_SIZE];
160     StateMachineResult new_frame;
161     bool valid_frame = false;
162 
163     while(!valid_frame) {
164 
165         new_frame = read_frame(fd, link_layer_data.frame, MAX_FRAME_SIZE);
166         frame_size = destuff(link_layer_data.frame, new_frame.packet_size, aux_buffer,
    MAX_PACKET_SIZE + 1);
167 
168         if (frame_size == 0) {
169             fprintf(stderr, "Error in data_link - llread: destuffing packet
    unsuccessfull\n");
```

```
170             return -1;
171         }
172         uint8_t bcc2 = aux_buffer[--frame_size];
173 
174         // Error
175         if (new_frame.frame_type != expected_package || !validBCC2(bcc2, aux_buffer,
    frame_size)) {
176             if (assemble_supervision_frame(expected_package ? REJ_1 : REJ_0, RECEIVER,
    response_frame) == -1) return -1;
177         } else {
178             memcpy(buffer, aux_buffer, frame_size);
179             if (assemble_supervision_frame(expected_package ? RR_0 : RR_1, RECEIVER,
    response_frame) == -1) return -1;
180             valid_frame = true;
181         }
182         if (write(fd, response_frame, SUPV_FRAME_SIZE) != 5) {
183             fprintf(stderr, "Error in data_link - llread: did not write response frame
    correctly");
184             return -1;
185         }
186     }
187 
188     expected_package = expected_package ? 0 : 1;
189     return frame_size;
190 }
191 
192 int setup_terminal(int fd) {
193     struct termios new_terminal;
194 
195     if ( tcgetattr(fd, &old_terminal) == -1) {
196       return -1;
197     }
198 
199     bzero(&new_terminal, sizeof(new_terminal));
200     new_terminal.c_cflag = link_layer_data.baud_rate | CS8 | CLOCAL | CREAD;
201     new_terminal.c_iflag = IGNPAR;
202     new_terminal.c_oflag = 0;
203     new_terminal.c_lflag = 0;
204 
205     new_terminal.c_cc[VTIME] = VTIME_VALUE;
206     new_terminal.c_cc[VMIN] = VMIN_VALUE;
207 
208     tcflush(fd, TCIOFLUSH);
209 
210     if ( tcsetattr(fd,TCSANOW,&new_terminal) == -1) {
211       return -1;
212     }
213 
214     return 0;
215 }
216 
217 int restore_terminal(int fd) {
218     if (tcsetattr(fd,TCSANOW,&old_terminal) == -1) {
219       return -1;
220     }
```

```
221     return 0;
222 }
223 
224 int assemble_supervision_frame(FrameType frame_type, ConnectionType connection_type,
    uint8_t* frame_buffer) {
225     frame_buffer[0] = FLAG;
226     switch (connection_type) {
227         case TRANSMITTER: {
228             frame_buffer[1] = 0x03;
229             if (frame_type == UA) {
230                 frame_buffer[1] = 0x01;
231             }
232             if (frame_type == SET || frame_type == DISC || frame_type == UA) {
233                 frame_buffer[2] = frame_type;
234             } else {
235                 fprintf(stderr, "Error: Data_link - assemble_supervision_frame - invalyd
     frame_type: %x\n", frame_type);
236                 return -1; //Invalid frame_type for this function
237             }
238             break;
239         }
240         case RECEIVER: {
241             frame_buffer[1] = 0x03;
242             if (frame_type == DISC) {
243                 frame_buffer[1] = 0x01;
244             }
245             if (frame_type == UA || frame_type == DISC || frame_type == RR_0 ||
    frame_type == RR_1 || frame_type == REJ_0 || frame_type == REJ_1) {
246                 frame_buffer[2] = frame_type;
247             } else {
248                 fprintf(stderr, "Error: Data_link - assemble_frame - invalyd frame_type\
    n");
249                 return -1; //Invalid frame_type for this function
250             }
251             break;
252         }
253     }
254     frame_buffer[3] = frame_buffer[1] ^ frame_buffer[2];
255     frame_buffer[4] = FLAG;
256     return 0;
257 }
258 
259 uint32_t assemble_information_frame(uint8_t* packet_data, size_t packet_size, uint8_t*
    frame, uint8_t frame_number) {
260     size_t frame_size = 4;
261     frame[0] = FLAG;
262     frame[1] = 0x03;
263     frame[2] = frame_number;
264     frame[3] = frame[1] ^ frame[2];
265     frame_size = stuff(packet_data, packet_size, frame, frame_size);
266     uint8_t bcc2[2];
267     bcc2[0] = compute_bcc2(packet_data, packet_size);
268     frame_size = stuff(bcc2, 1, frame, frame_size);
269     frame[frame_size++] = FLAG;
270     return frame_size;
```

```
}

uint32_t stuff(uint8_t* packet, size_t length, uint8_t* frame, size_t occupied_bytes) {
    size_t stuffed_length = occupied_bytes;

    int new_buffer_i = occupied_bytes;

    for (int i = 0; i < length; i++) {
        uint8_t byte = packet[i];
        if (byte == FLAG || byte == ESC_HEX) {
            frame[new_buffer_i++] = ESC_HEX;
            frame[new_buffer_i++] =  byte ^ STUFF_XOR_HEX;
            stuffed_length++;
        } else {
            frame[new_buffer_i++] = byte;
        }
        stuffed_length++;
    }
    return stuffed_length;
}

uint32_t destuff(uint8_t* stuffed_packet, size_t stuffed_length, uint8_t*
    unstuffed_packet, size_t max_unstuffed_packet_size) {
    size_t unstuffed_length = stuffed_length;

    int64_t new_buffer_i = 0;

    for (int i = 0; i < stuffed_length; i++) {
        uint8_t byte = stuffed_packet[i];
        if (new_buffer_i >= max_unstuffed_packet_size) {
            fprintf(stderr, "Error in data_link - destuffing: unstuffed packet is bigger
     than the destiny buffer! Tried to access %ld on a %ld bytes buffer\n", new_buffer_i
    , max_unstuffed_packet_size);
            return 0;
        }
        if (byte == ESC_HEX && i < stuffed_length - 1) {
            if (stuffed_packet[i + 1] == (FLAG ^ STUFF_XOR_HEX)) {
                unstuffed_packet[new_buffer_i++] = FLAG;
            } else if (stuffed_packet[i + 1] == (ESC_HEX ^ STUFF_XOR_HEX)) {
                unstuffed_packet[new_buffer_i++] = ESC_HEX;
            }
            unstuffed_length--;
            i++;
        } else {
            unstuffed_packet[new_buffer_i++] = byte;
        }
    }
    return unstuffed_length;
}

int establish_transmitter_connection(int fd) {
    uint8_t supervision_buffer[SUPV_FRAME_SIZE];

    if ((assemble_supervision_frame(SET, TRANSMITTER, supervision_buffer)) != 0) {
        return -1;
```

```
323     }
324     int res = write_buffer(fd, supervision_buffer, SUPV_FRAME_SIZE, UA);
325     if (res == -2) {
326         fprintf(stderr, "Error in data_link - establish_transmitter_connection: no.
    tries exceeded\n");
327         return -1;
328     } else if ( res == -1) {
329         perror("Error in data_link - disconnect_receiver");
330         return -1;
331     } else if (res != 5) {
332         fprintf(stderr, "Error in data_link - establish_transmitter_connection: no.
    bytes written not matching expected\n");
333         return -1;
334     }
335     return 0;
336 }
337 
338 int wait_transmitter_connection(int fd) {
339     StateMachineResult res = read_frame(fd, link_layer_data.frame, MAX_FRAME_SIZE);
340     if (res.frame_type == SET) {
341         uint8_t supervision_buffer[SUPV_FRAME_SIZE];
342         assemble_supervision_frame(UA, RECEIVER, supervision_buffer);
343         if (write(fd, supervision_buffer, SUPV_FRAME_SIZE) != 5) {
344             fprintf(stderr, "Error in data_link - wait_transmiiter_connection: could no
    write ACK to transmitter\n");
345             return -1;
346         }
347         return 0;
348     } else {
349         fprintf(stderr, "Error in data_link - wait_transmiiter_connection: received
    wrong type of frame\n");
350         return -1;
351     }
352 }
353 
354 int disconnect_transmitter(int fd) {
355     uint8_t supervision_buffer[SUPV_FRAME_SIZE];
356     assemble_supervision_frame(DISC, TRANSMITTER, supervision_buffer);
357     int res = write_buffer(fd, supervision_buffer, SUPV_FRAME_SIZE, DISC);
358     if ( res == -2) {
359         fprintf(stderr, "Error in data_link - disconnect_transmitter: no. tries exceeded
    \n");
360         return -1;
361     } else if ( res == -1) {
362         perror("Error in data_link - disconnect_receiver");
363         return -1;
364     } else if ( res != 5) {
365         fprintf(stderr, "Error in data_link - disconnect_transmitter: no. bytes written
    not matching expected\n");
366         return -1;
367     }
368     assemble_supervision_frame(UA, TRANSMITTER, supervision_buffer);
369     if (write(fd, supervision_buffer, SUPV_FRAME_SIZE) != 5) {
370         return -1;
371     }
```

```
372     return 0;
373 }
374 
375 int disconnect_receiver(int fd) {
376 
377     StateMachineResult res = read_frame(fd, link_layer_data.frame, MAX_FRAME_SIZE);
378     if (res.frame_type == DISC) {
379         uint8_t supervision_buffer[SUPV_FRAME_SIZE];
380         assemble_supervision_frame(DISC, TRANSMITTER, supervision_buffer);
381         int res = write_buffer(fd, supervision_buffer, SUPV_FRAME_SIZE, UA);
382         if ( res == -2) {
383             fprintf(stderr, "Error in data_link - disconnect_receiver: no. tries
    exceeded\n");
384             return -1;
385         } else if ( res == -1) {
386             perror("Error in data_link - disconnect_receiver");
387             return -1;
388         } else if ( res != 5) {
389             fprintf(stderr, "Error in data_link - disconnect_receiver: no. bytes written
     not matching expected\n");
390             return -1;
391         }
392         return 0;
393     } else {
394         fprintf(stderr, "Error in data_link - disconnect_receiver: unexpected frame type
    \n");
395         return -1;
396     }
397 }
398 
399 /* Writes a buffer to the serial port and awaits a certain response
400     fd -> Serial port file desciptor
401     buffer -> buffer to write
402     length -> length of the buffer to be written
403     expected_response -> expected response sent by the other end of the serial port
404 */
405 int write_buffer(int fd, uint8_t* buffer, size_t length, FrameType expected_response) {
406     StateMachineResult res_frame = {};
407     int tries = NUM_TX;
408     int res = 0;
409     while (tries > 0) {
410         if((res = write(fd, buffer, length * sizeof(uint8_t))) == -1) return -1; //
    Writes buffer to serial port
411         alarm(3);
412         res_frame = read_frame(fd, link_layer_data.frame, MAX_FRAME_SIZE); // Reads
    response
413         if (alarm_called) { // Timeout
414             tries--;
415             alarm_called = !alarm_called;
416             fprintf(stderr, "Alarm called, response timed out, %d tries left\n", tries);
417         } else {
418             alarm(0);
419             if (res_frame.frame_type == expected_response) {
420                 //Only break if frame received and data was correct
421                 break;
```

```
            }
        }
    }
    if (tries <= 0) { // Timeout limit
        return -2;
    } else {
        return res;
    }
}

bool validBCC2(uint8_t bcc2, uint8_t* buffer, size_t buffer_size) {
    uint8_t side_bcc2 = compute_bcc2(buffer, buffer_size);
    return ((side_bcc2 ^ bcc2) == 0);
}

uint8_t compute_bcc2(uint8_t* buffer, size_t buffer_size) {
    uint8_t bcc2 = buffer[0];
    for (int i = 1; i < buffer_size; i++) {
        bcc2 ^= buffer[i];
    }
    return bcc2;
}
```

Listing 16: data_link.c

10.5 data _link.h

```
#pragma once

#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include "macros.h"
#include "state_machine.h"

#define MAX_FRAME_SIZE ((MAX_PACKET_SIZE * 2) + 7)

struct LinkLayerData {
    ConnectionType role;
    char port[MAX_PORT_SIZE];
    unsigned int baud_rate;
    unsigned int sequence_number;
    unsigned int timeout;
    unsigned int num_transmissions;
    uint8_t* frame;
} LinkLayerData;

uint32_t stuff(uint8_t* packet, size_t length, uint8_t* frame, size_t occupied_bytes);
uint32_t destuff(uint8_t* stuffed_packet, size_t stuffed_length, uint8_t*
    unstuffed_packet, size_t max_unstuffed_packet_size);

int llopen(char* port, ConnectionType role);
int llwrite(int fd, uint8_t* buffer, size_t length);
int llread(int fd, uint8_t* buffer, size_t max_buffer_size);
```

```
27 int llclose(int fd);
```

Listing 17: data_link.h

10.6 alarm.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <signal.h>
4  #include <stdbool.h>
5  #include "alarm.h"
6
7  static struct sigaction old_action;
8  static AlarmListener* alarm_listeners = NULL;
9  static size_t number_of_listeners = 0;
10
11 // Alarm handler
12 void alarm_callback(int signo) {
13     // Calls 'subhandlers' that were subscribed (behaviour like observer pattern in OP)
14     for (int i = 0; i < number_of_listeners; i++) {
15         alarm_listeners[i]();
16     }
17 }
18
19 int setup_alarm_handler() {
20     struct sigaction new_action;
21     sigset_t smask;
22     if (sigemptyset(&smask)==-1) {
23         return -1;
24     }
25
26     new_action.sa_mask = smask;
27     new_action.sa_handler = alarm_callback;
28     new_action.sa_flags = 0;
29
30     if (sigaction(SIGALRM, &new_action, &old_action) == -1) {
31         perror("Error in alarm - setup_alarm_handler");
32         return -1;
33     }
34     return 0;
35 }
36
37 // Add a new subhandler/listener to the functions to be called when the alarm 'rings'
38 int subscribe_alarm(AlarmListener alarm_listener) {
39     number_of_listeners++;
40     if (alarm_listeners == NULL) {
41         if ((alarm_listeners = (AlarmListener*) malloc(sizeof(AlarmListener))) == NULL)
    {
42             perror("Error in alarm - subscribe_alarm");
43             return -1;
44         }
45         alarm_listeners[0] = alarm_listener;
46     } else {
47         AlarmListener* new_alarm_listeners;
```

```
48          if ((new_alarm_listeners = realloc(alarm_listeners, sizeof(AlarmListener) *
    number_of_listeners)) == NULL) {
49              perror("Error in alarm - subscribe_alarm");
50              return -1;
51          }
52          alarm_listeners = new_alarm_listeners;
53          alarm_listeners[number_of_listeners - 1] = alarm_listener;
54      }
55      return 0;
56  }
57
58  int restore_alarm_handler() {
59      free(alarm_listeners);
60      number_of_listeners = 0;
61      if (sigaction(SIGALRM, &old_action, NULL) == -1) {
62          return -1;
63      }
64      return 0;
65  }
```

Listing 18: alarm.c

10.7 alarm.h

```
1  #pragma once
2
3  typedef void (*AlarmListener)();
4
5  int setup_alarm_handler();
6  int subscribe_alarm(AlarmListener alarm_listener);
7  int restore_alarm_handler();
```

Listing 19: alarm.h

10.8 error.c

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "error.h"
5  #include "macros.h"
6
7
8  static double header_error_rate = 0;
9  static double data_error_rate = 0;
10 static int prop_time = 0;
11
12
13 bool should_corrupt(double error_rate);
14 uint8_t gen_corrupted_byte();
15
16
17
```

```
18
19 bool should_corrupt(double error_rate) {
20     double prob = (double) random()/ (double) RAND_MAX;
21     return (prob <= error_rate);
22 }
23
24 bool should_corrupt_data() {
25     return should_corrupt(data_error_rate);
26 }
27
28 bool should_corrupt_header() {
29     return should_corrupt(header_error_rate);
30 }
31
32 void corrupt_header(uint8_t* a, uint8_t* c, uint8_t* bcc1) {
33     printf("Generated header error\n");
34     uint64_t corrupted_byte = random() % 3;
35     switch (corrupted_byte) {
36         case 0:
37             *a = gen_corrupted_byte();
38             break;
39         case 1:
40             *c = gen_corrupted_byte();
41             break;
42         case 2:
43             *bcc1 = gen_corrupted_byte();
44             break;
45         default:
46             fprintf(stderr, "Error in error.c - corrupt_header: unexpected
    corrupted_byte\n");
47             break;
48     }
49 }
50
51 void corrupt_data_buffer(uint8_t* data_buffer, size_t buffer_size) {
52     printf("Generated data error\n");
53     size_t corrupted_byte_idx = random() % buffer_size;
54     data_buffer[corrupted_byte_idx] = gen_corrupted_byte();
55 }
56
57 uint8_t gen_corrupted_byte() {
58     return ((uint8_t) random());
59 }
60
61 void set_error_rates(double h_error, double d_error) {
62     header_error_rate = h_error;
63     data_error_rate = d_error;
64 }
65
66 void set_prop_time(int t_prop) {
67     prop_time = t_prop;
68 }
69
70 void delay() {
71     usleep(1000*prop_time);
```

```
}
```

Listing 20: error.c

10.9 error.h

```
#pragma once

#include <stdbool.h>
#include <stdint.h>


void delay();
void set_error_rates(double h_error, double d_error);
void set_prop_time(int t_prop);
bool should_corrupt_data();
bool should_corrupt_header();
void corrupt_header(uint8_t* a, uint8_t* c, uint8_t* bcc1);
void corrupt_data_buffer(uint8_t* data_buffer, size_t buffer_size);
```

Listing 21: error.h

10.10 macros.h

```
#pragma once

typedef enum { TRANSMITTER, RECEIVER } ConnectionType;

#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FLAG 0x7E
#define BAUDRATE B38400
#define NUM_TX 3 //Number of tries
#define VMIN_VALUE 5
#define VTIME_VALUE 0
#define TIMEOUT 2 //Timeout in seconds
#define FALSE 0
#define TRUE 1
#define MAX_PACKET_SIZE 150
#define MAX_PORT_SIZE 20
#define SUPV_FRAME_SIZE 5
#define INF_DATA_LAYER_SIZE 6
#define STUFF_XOR_HEX 0x20
#define ESC_HEX 0x7d
```

Listing 22: macros.h

10.11 state _machine.c

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

```c
#include "error.h"
#include "macros.h"
#include "alarm.h"
#include "state_machine.h"

enum StateType_ {START, FLAG_RCV, A_RCV, C_RCV, BCC_OK, DATA, STOP};
typedef enum StateType_ StateType;

/*
    Functions declared here because they are private
*/
bool isA(uint8_t a);
bool isC(uint8_t c);

// State machine to read the serial port and check the input validity
StateMachineResult read_frame(int fd, uint8_t* frame, size_t frame_size) {
    StateMachineResult state_machine_res  = {
        .frame_type = ERROR,
        .packet_size = 0,
    };
    uint8_t byte_rcvd, a, c;
    StateType state = START;
    a = byte_rcvd = c = 0;

    while (state != STOP) {
        if(read(fd, &byte_rcvd, sizeof(uint8_t)) == -1) {
            return state_machine_res;
        }
        switch (state) {
            case START: {
                if (byte_rcvd == FLAG) {
                    state = FLAG_RCV;
                }
                break;
            }
            case FLAG_RCV: {
                if (isA(byte_rcvd)) {
                    a = byte_rcvd;
                    state = A_RCV;
                } else if (byte_rcvd != FLAG) {
                    state = START;
                }
                break;
            }
            case A_RCV: {
                if (isC(byte_rcvd)) {
                    c = byte_rcvd;
                    state = C_RCV;
                } else if (byte_rcvd == FLAG) {
                    state = FLAG_RCV;
                } else {
                    state = START;
                }
                break;
            }
```

```
60              case C_RCV: {
61                  if ((c == I_0 || c == I_1) && should_corrupt_header()) {
62                      corrupt_header(&a, &c, &byte_rcvd);
63                  }
64                  if ((a ^ c ^ byte_rcvd) == 0) {
65                      state = BCC_OK;
66                  } else if (byte_rcvd == FLAG) {
67                      state = FLAG_RCV;
68                  } else {
69                      state = START;
70                  }
71                  break;
72              }
73              case BCC_OK: {
74                  if (byte_rcvd == FLAG) {
75                      state = STOP;
76                  } else {
77                      state_machine_res.packet_size++;
78                      frame[0] = byte_rcvd;
79                      state = DATA;
80                  }
81                  break;
82              }
83              case DATA: {
84                  if (byte_rcvd == FLAG) {
85                      state = STOP;
86                  } else {
87                      state_machine_res.packet_size++;
88                      if (state_machine_res.packet_size == frame_size) {
89                          fprintf(stderr, "Error in data_link - state_machine: receiving
    more bytes than the max frame size\n");
90                          return state_machine_res;
91                      }
92                      frame[state_machine_res.packet_size - 1] = byte_rcvd;
93                  }
94                  break;
95              }
96              case STOP:
97                  break;
98          }
99      }
100     state_machine_res.frame_type = c;
101
102     // Random errors
103     if ((state_machine_res.frame_type == I_0 || state_machine_res.frame_type == I_1) &&
    should_corrupt_data()) {
104         corrupt_data_buffer(frame, state_machine_res.packet_size);
105     }
106
107     delay(); // T_PROP
108
109     return state_machine_res;
110 }
111
112 bool isA(uint8_t a) {
```

```
113     return (a == 0x03) || (a == 0x01);
114 }
115 
116 bool isC(uint8_t c) {
117     c &= 0x0F; // Removes useless package data
118     return (c == SET) || (c == DISC) || (c == UA) || (c == RR_0) || (c == REJ_0) || (c
        == I_0) || (c == I_1);
119 }
```

Listing 23: state$_m$achine.c

10.12 state _machine.h

```
1  #pragma once
2  
3  #include <stdlib.h>
4  #include <stdint.h>
5  
6  typedef enum {SET=0x03, UA=0x07, I_0 = 0x00, I_1 = 0x01, RR_0=0x05, RR_1=0x85, DISC=0x0D
       , ERROR, REJ_0=0x01, REJ_1=0x81, NONE} FrameType;
7  
8  typedef struct {
9      FrameType frame_type;
10     size_t packet_size;
11 } StateMachineResult;
12 
13 StateMachineResult read_frame(int fd, uint8_t* frame, size_t frame_size);
```

Listing 24: state$_m$achine.h

10.13 file.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdint.h>
4  #include <string.h>
5  #include <errno.h>
6  #include <sys/stat.h>
7  #include <fcntl.h>
8  #include "file.h"
9  
10 // Create Dir and Path for file
11 int path_parser(char* path, char* dir, char* file_name) {
12     file_name[0] = '\0';
13     dir[0] = '\0';
14 
15     char* last_dir_sep = strrchr(path, '/');
16     char* cur_ptr = path;
17 
18     // Dir separation
19     if (last_dir_sep == NULL) {
20         strcpy(dir, "./");
21         strcpy(file_name, path);
```

```
    } else {
        char* cur_ptr = path;
        size_t dir_idx = 0;
        while (cur_ptr != last_dir_sep) {
            dir[dir_idx] = *cur_ptr;
            cur_ptr++;
            dir_idx++;
        }
        dir[dir_idx] = '/';
        dir[dir_idx + 1] = '\0';
        cur_ptr = ++last_dir_sep;
    }

    // Path separation
    size_t file_name_idx = 0;
    while (*cur_ptr != '\0') {
        file_name[file_name_idx] = *cur_ptr;
        cur_ptr++;
        file_name_idx++;
    }
    *cur_ptr = '\0';

    struct stat stat_buf;
    int status;
    if ((status = stat(dir, &stat_buf)) == -1) {
        perror("Error in path_parser");
        return -1;
    }
    if (S_ISDIR(stat_buf.st_mode)) {
        return 0;
    } else if (S_ISREG(stat_buf.st_mode)) {
        fprintf(stderr, "Error in path_parser - file already exists!\n");
        return -1;
    } else {
        fprintf(stderr, "Error in path_parser - nvalid file path\n");
        return -1;
    }

}

bool file_exists(char* dir, char* file_name) {
    char* file_path = calloc(DIR_PATH_SIZE + FILE_NAME_SIZE, sizeof(uint8_t));
    strncpy(file_path, dir, DIR_PATH_SIZE);
    strcat(file_path, file_name);

    struct stat stat_buf;

    if ((stat(file_path, &stat_buf) == -1) && errno == ENOENT) {
        free(file_path);
        return false;
    }
    free(file_path);
    return true;
}
```

Listing 25: file.c

10.14 file.h

```
#pragma once

#include <stdbool.h>

#define DIR_PATH_SIZE (4096 - FILE_NAME_SIZE)
#define FILE_NAME_SIZE 255

int path_parser(char* path, char* dir, char* file_name);
bool file_exists(char* dir, char* file_name);
```

Listing 26: file.h