Protocolo de Ligação de Dados

Relatório do 1º trabalho laboratorial de RC

Turma 6 Grupo 5
Maria José Valente da Silva Carneiro (up201907726)
Rodrigo Tuna de Andrade (up201904967)

Dezembro 2021

Conteúdo

Sumário

Este projeto foi elaborado como o primeiro projeto prático no contexto da unidade curricular de Redes de Computadores (RC) da Licenciatura em Engenharia Informática e Computação (L.EIC). Teve como objetivo desenvolver um protocolo de ligação de dados, robusto e resistente a erros, entre computadores que comunicam através de uma porta série de forma a que fosse possível enviar um ficheiro de um computador para o outro.

Todos os objetivos propostos para o trabalho foram cumpridos, tendo como resultado uma aplicação, implementada em C para PC's com Linux, capaz de enviar qualquer tipo ficheiros sem perda de dados.

1 Introdução

O trabalho desenvolvido prende-se na implementação de um protocolo de ligação de dados de modo a permitir a comunicação entre dois computadores, ligados por portas série RS-232 com comunicação assíncrona, e no design de uma aplicação simples e independente de transferência de ficheiros.

O presente relatório expõe o método utilizado para cumprir os objetivos propostos, bem como uma análise estatística à eficiência dos protocolos implementados. O relatório está então dividido em secções distintas: **Arquitetura**, que mostra os blocos principais bem como as interfaces utilizadas; **Estrutura do Código**, que generaliza a relação das principais funções, estruturas de dados e API's com a Arquitetura implementada; **Casos de Uso Principais**, que identifica esses casos e exemplifica a estrutura linear e sequencial da execução das funções; **Protocolo de Ligação Lógica**, que detalha os aspetos principais do funcionamento do Protocolo de Ligação Lógica; **Protocolo de Aplicação**, que aprofunda os aspetos principais do funcionamento do Protocolo de Aplicação; **Validação**, que descreve os testes efetuados com a apresentação quantificada dos seus resultados; e, por fim, **Eficiência do Protocolo de Ligação de Dados**, que caracteriza estatisticamente a eficiência do protocolo implementado, em comparação com um protocolo teórico de *Stop & Wait*.

2 Arquitetura

A arquitetura genérica do projeto divide-se em duas camadas independentes, definidas da seguinte forma:

- **Camada de Ligação de Dados**, que estabelece a comunicação com a porta série, de modo a permitir uma comunicação fiável de dados organizados em tramas, através de operações de abertura/fecho e escrita/leitura no driver da porta série (não diretamente na sua camada física). Também efetua o controlo de erros bem como do fluxo de execução, com recurso à numeração de tramas e confirmação positiva.
- **Camada de Aplicação**, que estabelece uma interface da linha de comandos que permite a transferência de um ficheiro entre os dois computadores, através da camada da ligação de dados. Os dados são transferidos em conjuntos de pacotes, protegidos por protocolos de controlo que garantem a consistência e linearidade no processo de transferência.

3 Estrutura do Código

O código está divido em duas camadas (Application e Data Link), como referido anteriormente, e dois ficheiros executáveis (*transmitter* e *receiver*).

3.1 Interface de Aplicação

O interface de Aplicação permite a execução do projeto pelo terminal e gera dois ficheiros executávéis:

- `./transmitter [COM] [file]`, que envia o ficheiro [file] pela porta série [COM]
- `./receiver [COM]`, que recebe um ficheiro pela porta série [COM]

[**COM**] representa a porta série a usar e [**file**] o path de um ficheiro a transmitir entre computadores. Esta camada efetua a transmissão de pacotes de dados e de controlo, com recurso às funções do interface do Linked Layer. Algumas das funções principais podem ser descritas por:

- `int application (int fd, status_t status, char* path)` - abre e fecha a comunicação com a porta série, recorrendo às funções da Linked Layer `llopen` e `llclose`, e envia/recebe o ficheiro de acordo com o status definido - *transmitter* ou *receiver*. Retorna 0 em sucesso e -1, caso contrário.
- `int app_send_file(char* path)` - envia o ficheiro através de pacotes de dados para a porta série, usando a função `app_send_data_packet`. Também envia pacotes de controlo `START` e `END`, usando a função `app_send_control_packet`. Recorre à função `llwrite` da Linked Layer para efetuar a comunicação com a porta série. Retorna 0 em sucesso e -1, caso contrário.
- `int app_receive_file()` - recebe o ficheiro através de pacotes de dados da porta série, usando a função `app_receive_data_packet`. Também recebe pacotes de controlo `START` e `END`, usando a função `app_receive_control_packet`, que usa para reconstruir o ficheiro e verificar a sua integridade. Recorre à função `llread` da Linked Layer para efetuar a comunicação com a porta série. Retorna 0 em sucesso e -1, caso contrário.

A porta série é configurada e restaurada após o envio/receção do ficheiro com as funções `int set_config(char * serial_port)` e `void reset_config(int fd)`, que se encontram em *transmitter*.c e *receiver*.c.

3.2 Interface de Linked Layer

O interface de Linked Layer efetua a comunicação com os drivers da porta série, necessária para a transmissão do ficheiro no interface de Aplicação, pelos ficheiros executáveis. Algumas das funções principais podem ser descritas por:

- `int ll_open(int fd, status_t st)` - inicia o protocolo de comunicação através do envio da trama de supervisão SET e da receção da trama não-numerada UA, por parte do *transmitter*, e vice-versa por parte do *receiver*. Retorna 0 em sucesso e -1, caso contrário.
- `int ll_read(int fd, uint8_t * buffer)` - lê a trama de informação recebida e envia a trama de supervisão RR, caso a leitura seja bem sucedida, e REJ caso contrário. Retorna o número de bytes lidos em sucesso ou um valor negativo, caso contrário.
- `int ll_write(int fd, uint8_t * buffer, int length)` - escreve a trama de informação e recebe as tramas de supervisão RR ou REJ, em caso de sucesso no envio ou de erro, respetivamente. Retorna o número de bytes escritos em sucesso, ou um valor negativo, caso contrário.
- `int ll_close(int fd)` - termina o protocolo de comunicação dependendo da máquina: o *transmitter* envia a trama de supervisão DISC, aguarda pela receção de DISC por parte do *receiver*, e após a receção, envia a trama não-numerada UA; já o *receiver*, recebe a trama de supervisão DISC por parte do *transmitter*, retorna DISC novamente e recebe a trama não-numerada UA. Retorna, em caso de sucesso, o número de bytes escritos ao enviar UA pelo *transmitter* ou 0 pelo *receiver*; caso contrário retorna um valor negativo.

4 Casos de Uso Principais

O caso de uso principal do programa é a transmissão de um ficheiro entre 2 máquinas em que uma atua como transmissora e outra como recetora. Para efeitos de exemplo, assumimos que o *transmitter* usa o driver 10 e o *receiver* o driver 11 e que estes se encontram conectados.

4.1 Execução

Primeiro, começamos por correr o *receiver* e ligámo-lo ao driver `11` para receber o ficheiro com a instrução seguinte:

`./receiver \dev\tty\S11`

De seguida, corremos o *transmitter* e ligámo-lo ao driver `10` com o nome do ficheiro que queremos enviar (`pinguim.txt`) com a instrução seguinte:

`./transmitter \dev\tty\S10 pinguim.gif`

4.2 Sequência de Execução

Após o início da execução com as instruções anteriores, a transmissão do ficheiro segue a seguinte linha de protocolos executados por `main()` em cada um dos ficheiros:

1. A conexão entre os dois computadores é configurada e aberta com recurso ao protocolo de `llopen` e a `set_config`
2. De modo a começar transferência do ficheiro, o *transmitter* envia um pacote de controlo que sinaliza o ínicio da transmissão ao *receiver*, que o recebe e verifica a sua integridade.
3. O ficheiro em si é enviado sequencialmente em pacotes de dados com recurso ao protocolo `llwrite` para o *receiver*, que os recebe e reconstrói, com recurso ao protocolo `llread`.
4. De modo a finalizar a transferência do ficheiro, o *transmitter* envia um pacote de controlo que sinaliza o fim da transmissão ao *receiver*, que o recebe e verifica a sua integridade.
5. A conexão entre os dois computadores é terminada com recurso ao protocolo de `llclose` e a `reset_config`.

5 Protocolo de Ligação Lógica

O protocolo de ligação lógica tem como função implementar a comunicação interna com os drivers da porta série, de modo a permitir a transmissão de um ficheiro. Este aplica uma combinação de diferentes protocolos, que garantem um mecanismo de controlo de erros através da retransmissão de tramas aquando de inconsistências nos dados transferidos entre máquinas, bem como de controlo de fluxo, através da sequencialização na transmissão (tramas são numeradas e apenas são mandadas depois da devida confirmação de receção das anteriores).

As tramas partilhadas dividem-se em 3 tipos: de Informação (I) que contém os dados a transferir; de Supervisão (S), que supervisionam a comunicação e Não-Numeradas (U), que confirmam a transmissão de tramas. As tramas S enviam os comandos `SET`, `DISC`, `RR` e `REJ`, enquanto as tramas U enviam `UA`. Todas as tramas são delimitadas por `flags` e têm um cabeçalho comum:

- **Campo de endereço (A)**: identifica quem envia o comando ou a resposta: `0x03` em comandos do *transmitter* e respostas do *receiver*, e `0x01` em comandos do *receiver* e respostas do *transmitter*.
- **Campo de Controlo (C)**: identifica o tipo de trama: `0x03` para `SET`, `0x0B` para `DISC`, `0x07` para `UA`, `0b0S000000` para `I`, `0bR0000101` para RR e `0bR0000001` para REJ, onde `S = Ns` e `R = Nr`. `Ns` e `Nr` são enviadas junto com tramas I e são bits únicos que representam o número de sequência que permite identificar se a trama recebida é resultado de uma retransmissão ou não. Numa transmissão normal, quando `Ns = 0`, a resposta recebida deverá ter `Nr = 1`.
- **Block Check Character (BCC)**: identifica erros que ocorram devido à troca de bits e é dado pelo ou-exclusivo dos bytes dos campos A e C.

Para além disso, as tramas I apresentam um campo de dados codificado usando *byte stuffing* e um `BCC` a protegê-lo que é o ou-exclusivo entre todos os bytes de dados do campo anterior.

As diferentes tramas são recebidas com recurso a máquinas de estado que segmentam a receção dos bytes dos campos das tramas e permitem assim a verificação de erros e o desencadeamento de ações adequadas perante esses mesmos erros. A receção das tramas encontra-se protegida por um timeout, definido na configuração da porta série (`VTIME = 30` e `VMIN` = 0), que garante a espera de 3 segundos entre cada byte consecutivo. Caso esse tempo seja excedido, tentam-se retransmissões das tramas, com um número máximo de 3 por receção.

Como referido anteriormente, a transparência dos campos de dados é garantida através de *byte stuffing*, que consiste em definir um byte `ESC` (`0x7E`) e se o byte de dados for igual a `FLAG` ou `ESC`, é substituído por `ESC` seguido do ou-exclusivo entre o byte de dados e `0x20`. Para se proceder ao *destuff* da trama, basta fazer o ou-exclusivo entre o byte de dados e `0x20` que se encontre após um `ESC`. Este mecanismo permite então evitar o falso reconhecimento de uma `FLAG` no interior de uma trama.

5.1 Implementação do Mecanismo de Ligação Lógica

Dado que a implementação interna da receção e transmissão de tramas é crucial para o funcionamento do interface LL, é essencial detalhar o seu funcionamento. Respostas (RR, REJ e UA) e Comandos (I, SET, DISC) são enviados com recurso às funções `send_command`, `send_response` e `send_I_FRAME`:

- `send_command`: envia comandos do tipo `SET` e `DISC`, com recurso à função `write`. O campo de endereço varia de acordo com o *status* (se é *transmitter* ou *receiver*). Retorna o número de bytes escritos em caso de sucesso, ou -1 caso contrário.

- `send_response`: Semelhante à anterior mas envia respostas do tipo `RR`, `REJ` e `UA`. Retorna o número de bytes escritos em caso de sucesso, ou -1 caso contrário.

- `send_I_FRAME`: envia comandos do tipo `I`. O campo de endereço corresponde sempre ao *transmitter* e o campo de controlo varia de acordo com o `Ns` atual. O `BCC2` é calculado a partir do ou-exclusivo dos dados a enviar, que são posteriormente *stuffed*. Os diferentes componentes da trama são todos enviados com recurso a `write`. Retorna o tamanho do campo de dados enviado em caso de sucesso, ou -1 caso contrário.

A receção das tramas recorre a máquinas de estado que permitem a receção segmentada dos campos de dados, e é diferenciada trama a trama:

- `receive_U`: lê a informação da porta série byte a byte, com recurso à função `read`. Através da função `u_state_trans` processa o byte recebido e verifica a sua integridade, confrontando o seu valor com o esperado. Dado que tramas U apenas enviam `UA`, o campo de controlo deverá sinalizar apenas a receção desse tipo de tramas. A execução para quando o estado final é atingido e, caso não o seja, a função é terminada devido ao *timeout* estabelecido. Retorna 0 em caso de sucesso, ou -1 caso contrário.

- `receive_S`: Semelhante à anterior mas usa a função `s_state_trans` para efetuar as transições de estado e apenas recebe tramas do tipo `SET`, `DISC`, `RR` ou `REJ`. Retorna 0 em caso de sucesso, ou -1 caso contrário.

- `receive_I`: lê a informação da porta série byte a byte, com recurso à função `read`. Através da função `i_state_trans` processa o byte recebido e verifica a sua integridade, confrontando o seu valor com o esperado. Caso o campo de controlo apresente um `BCC` incorreto (semelhante ao anterior), estamos perante um duplicado, pelo que é enviado um `RR` de confirmação e o estado atual volta a ser o inicial. A execução da máquina de estados termina quando o estado final é atingido, e, caso não o seja, a função é terminada devido ao *timeout* estabelecido. Os dados vão sendo recebidos byte a byte pela

máquina de estados e são posteriormente *destuffed*, sendo confirmado o valor do `BCC2` com o seu valor esperado. Retorna o tamanho do campo de dados recebido em caso de sucesso, ou -1 caso contrário.

Estas funções permitem, então, a implementação do protocolo de ligação lógica e a partilha de tramas necessárias para as funções do interface LL, cujo funcionamento é explicado na seccção 3.2.

6 Protocolo de Aplicação

O protocolo de aplicação é a camada de mais alto nível do programa e tem como função a transmissão de ficheiros usando os métodos expostos pelo interface de Linked Layer. Pacotes de Dados e de Controlo são trocados entre o *transmitter* e o *receiver* que regularizam e permitem o envio e a receção do ficheiro a transferir.

Os pacotes de controlo sinalizam o início e o fim da transferência e contêm informação acerca do ficheiro, como o seu tamanho e nome. São interpretados como *arrays* com os seguintes campos (ordenados):

- **Campo de controlo (C)**: Indica o tipo do pacote: `0x02` para `START` e `0x03` para e `END`
- **Tipo, Tamanho e Valor (Ti, Li, Vi)**: Indicam o tipo de parâmetro, o seu tamanho em bytes e o valor desse parâmentro. O tipo do parâmetro que representa o tamanho do ficheiro é `0x00` e o que representa o nome do ficheiro é `0x01`.

Os pacotes de dados contêm o ficheiro em si pronto para ser enviado pelo protocolo da LL. São interpretados como *arrays* com os seguintes campos (ordenados):

- **Campo de controlo (C)**: Indica o tipo do pacote: `0x01` para `DATA`.
- **Número de sequência (N)**: Indica o número de sequência dos dados, com módulo 255.
- **Tamanho do pacote (K)**: Indica, usando 2 bytes (L1 e L2) o tamanho do pacote. (`K = L1 + L2*256`)
- **Campo de dados (P1..Pk)**: Guarda os bytes de dados do atual pacote a transferir.

O *transmitter* envia o pacote de controlo inicial, que constrói preenchendo os respetivos campos correspondentes aos parâmetros a enviar: o tamanho do ficheiro e o seu nome. De seguida, lê bytes de dados do ficheiro e envia-os em pacotes com um número de sequência particular, que é incrementado a cada grupo de bytes lido. Por fim, envia o pacote de controlo final, com os mesmos parâmetros que o primeiro (apenas diferem no campo de controlo).

Entretanto, o *receiver* recolhe informação do ficheiro usando o pacote de controlo que recebe inicialmente e que usa para reconstruir o ficheiro original com os pacotes de dados. Os pacotes de dados são lidos até ao número de bytes esperado ser recebido. A deteção de erros é efetuada em cada pacote recebido, verificando se a informação recolhida dos campos fixos é equivalente à esperada. No fim da transmissão, é efetuada a comparação entre os pacotes de controlo inicial e final, também como mecanismo de deteção de erros. Caso algum ocorra, o programa termina.

7 Validação

De forma a validar a implementação do protocolo foram realizados diferentes testes:

- Envio de diferentes tipos de ficheiros e com diferentes tamanhos.
- Interrupção de porta série durante o envio do ficheiro.
- Interferência na porta série durante o envio do ficheiro.

- Envio do ficheiro com diferentes valores de baudrate.
- Envio do ficheiro com diferentes valores de tempo de propagação.
- Envio do ficheiro com diferentes valores de tamanho do pacote.
- Envio do ficheiro com diferentes valores de probabilidade de erros na trama.

Todos os testes foram passados com sucesso.

8 Eficiência do Protocolo de Ligação de Dados

A eficiência do protocolo foi estudada variando os seguintes parâmetros:

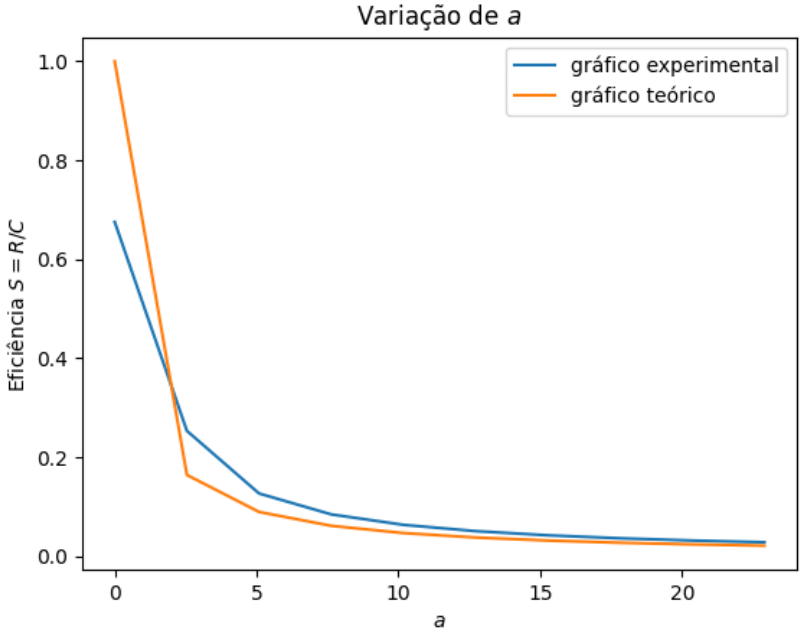
- O rácio entre o tempo de propagação e o tempo de envio de um frame ($a = T_{prop}/T_f$)
- A probabilidade de erro de um frame p_e
- A baudrate C
- O tamanho dos pacotes enviados S_p

A eficiência, S foi medida utilizando o tamanho do ficheiro S_f, o tempo de execução do programa t_e e a baudrate resultando em:

$$S = \frac{\frac{S_f}{t_e}}{C}$$

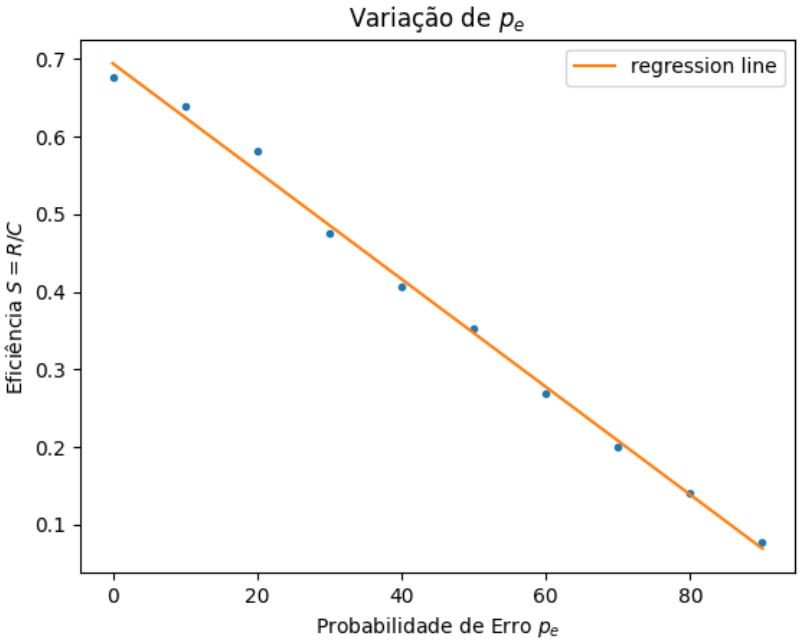
O tamanho dos pacotes enviados e a baudrate foram diretamente alterados, enquanto que a foi alterado alterando T_{prop} através de um delay artificial antes de receber cada frame. Para saber Tf foi assumido que T_{prop} sem delay seria muito próximo de 0 e assim calculado como o tempo para receber cada frame. Para $p(e)$ foi introduzido erro no BCC2 das tramas enviadas.

8.1 Variação de a



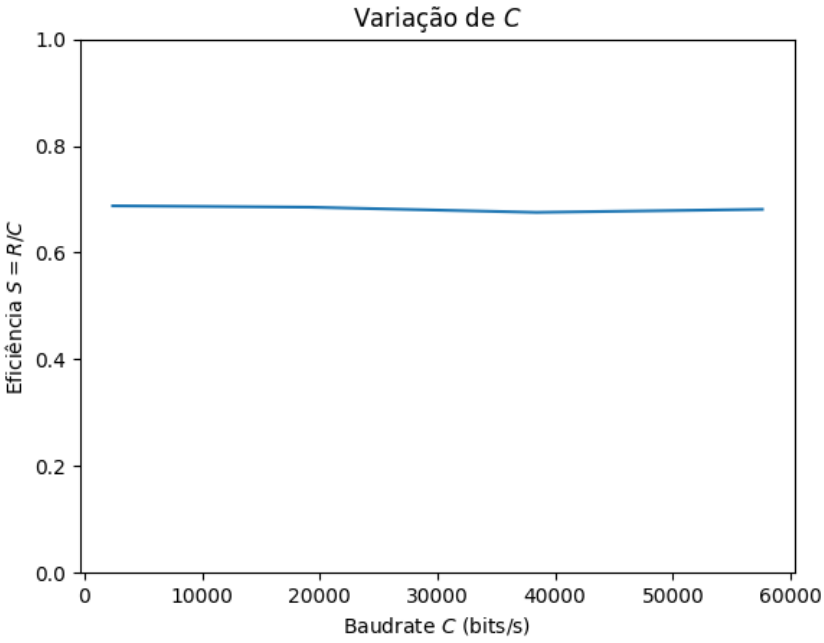
Comparando a eficiência medida com a sua formulação teórica $S = 1/(1+2a)$, verificamos que existe uma pequena discrepância nos primeiros valores o que pode ser explicado pela aproximação que realizamos de $T_{prop} = 0$, visto que para os valores menores de T_{prop} tem maior impacto.

8.2 Variação da probabilidade de erro



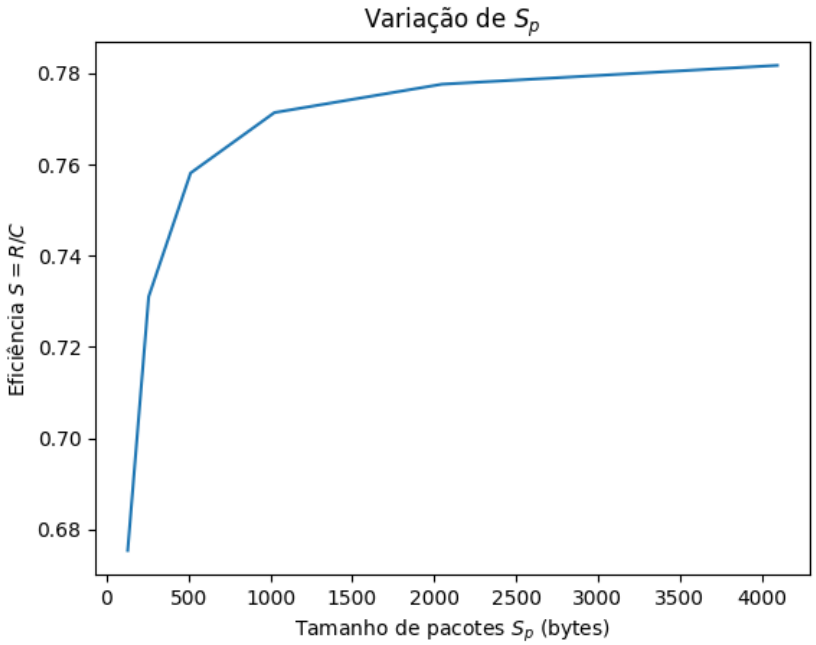
A relação entre p_e e S é dada por $S = (1 - p_e)/(1 + 2a)$, como o denominador permanece constante, a relação traduz - se numa reta, o que pode ser observado no gráfico.

8.3 Variação da baudrate



A variação da baudrate não tem nenhum impacto na eficiência. O que seria esperado visto que $S = (1 - p_e)/(1 + 2a)$, onde não entra C.

8.4 Variação do tamanho dos pacotes enviados



É observada uma melhoria da eficência com o aumento do número de bytes em cada pacote. Este facto pode ser explicado pois, como diminuem o número de pacotes enviados, são enviados menos cabeçalhos, são realizados menos checks de BCC etc. o que faz com que o tempo de execução seja menor.

9 Conclusões

O presente projeto teve como objetivo a criação de um protocolo de ligação de dados e de um interface de aplicação simples para a transferência segura de um ficheiro entre duas máquinas, ligadas através de uma porta série. Conseguimos desenvolver com sucesso um protocolo robusto e consistente perante erros e eficiente perante diferentes condições, que avaliamos em comparação com o protocolo de *stop-and-wait* e garantimos um programa modularizado por camadas, independentes entre si. Deste modo, todos os objetivos propostos foram cumpridos.

Com a realização do projeto, tivemos então, acima de tudo, oportunidade de pôr em prática os conhecimentos teóricos aprendidos nas aulas de RCOM e de aprofundar o nosso conhecimento acerca do funcionamento de protocolos de rede e das suas complexidades.

Apêndice A Código Fonte

A.1 Application Layer

```
/**
 * @brief Application Layer
 */

#ifndef _APP_H_
#define _APP_H_

#include <stdio.h>
#include <stdint.h>

#include "ll.h"

#define DATA_PACKET_MAX_SIZE  4096

// Flags
#define DATA_CTRL 1
#define START_CTRL 2
#define END_CTRL 3

//Number of Fields in App Packets
#define NUM_FIELDS_CTRL 5
#define NUM_FIELDS_DATA 4

//Shift Offset
#define OFFSET 0xFF

//Control Packet Fields Flags
#define CTRL 0
#define FILE_SIZE_FLAG 0
#define FILE_NAME_FLAG 1
#define T1_SIZE 1
#define L1_SIZE 2
#define V1_SIZE 3
#define T2_NAME 7
#define L2_NAME 8
#define V2_NAME 9

//Data Packet Fields Flags
#define N_DATA 1
#define L2_DATA 2
#define L1_DATA 3
#define P_DATA 4

// File descriptor corresponding to the serial port
int fileDescriptor;

/**
 *
 * @brief Initializes the application
 *
 * @param fd File descriptor corresponding to the serial port
 * @param status Status of the application, Transmitter or Receiver
 * @param path Path of the file to transfer
 * @return 0 on success, -1 otherwise
 *
 * */
int application (int fd, status_t status, char* path);

```

```
59 /**
60  *
61  * @brief Sends the data packet
62  *
63  * @param data Data packet from the file to be transferred
64  * @param data_size Size of the data to send in bytes
65  * @param sequence_number Sequence Number of the data packet
66  * @return 0 on success, -1 otherwise
67  *
68  * */
69 int app_send_data_packet(char* data, uint32_t data_size, unsigned int sequence_number);
70
71 /**
72  *
73  * @brief Sends the control packet
74  *
75  * @param ctrl_flag Flag identifier of the control message to send, START or END
76  * @param data_size Size of the file to be transferred
77  * @param filename Name of the file to be transferred
78  * @return 0 on success, -1 otherwise
79  *
80  * */
81 int app_send_control_packet(int ctrl_flag, uint32_t file_size, const char* filename);
82
83 /**
84  *
85  * @brief Sends file
86  *
87  * @param path Path of the file to be transferred
88  * @return 0 on success, -1 otherwise
89  *
90  * */
91 int app_send_file(char* path);
92
93 /**
94  *
95  * @brief Receives the data packet
96  *
97  * @param data Pointer where the data will be stored
98  * @param sequence_number Expected Sequence Number
99  * @return Total size of the data packet received on success, -1 otherwise
100 *
101 * */
102 int app_receive_data_packet(char * data, int sequence_number);
103
104 /**
105  *
106  * @brief Sends the control packet
107  *
108  * @param ctrl_flag Flag identifier of the control message expected to be received, START
      or END
109  * @param file_size Pointer where the file size will be stored
110  * @param filename Pointer where the file name will be stored
111  * @return 0 on success, -1 otherwise
112  *
113  * */
114 int app_receive_control_packet(int ctrl_flag, unsigned int * file_size, char* filename);
115
116 /**
117  *
118  * @brief Receives file
119  *
```

```
 * @return 0 on success, -1 otherwise
*/
int app_receive_file();

#endif // _APP_H
```

Listing 1: app.h

```
#include "../header/app.h"

#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <libgen.h>

int application (int fd, status_t status, char* path) {

    if(ll_open(fd, status) < 0) return -1;

    fileDescriptor = fd;

    if(status == TRANSMITTER){
        if(app_send_file(path) < 0){
            fprintf(stderr, "Error: Couldn't send file to Receiver.\n");
            return -1;
        }
    }
    else {
        if(app_receive_file(fd) < 0){
            fprintf(stderr, "Error: Couldn't receive file from Transmitter.\n");
            return -1;
        }
    }

    if(ll_close(fd) < 0) return -1;
    fprintf(stdout, "File transmitted successfully.\n");
    return 0;
}

int app_send_data_packet(char* data, uint32_t data_size, unsigned int sequence_number){

    if(data_size > DATA_PACKET_MAX_SIZE){
        fprintf(stderr, "Error: Data packet has a larger size than the maximum allowed.\n
    ");
        return -1;
    }

    uint8_t * data_packet = (uint8_t *) malloc(data_size + NUM_FIELDS_DATA);

    data_packet[CTRL] = DATA_CTRL;
    data_packet[N_DATA] = sequence_number % 255;

    data_packet[L2_DATA] = (data_size >> 8) & OFFSET;
    data_packet[L1_DATA] = data_size & OFFSET;

    memcpy(data_packet + P_DATA, data, data_size);

    if(ll_write(fileDescriptor, data_packet, data_size + NUM_FIELDS_DATA) < 0){
        fprintf(stderr, "Error: Couldn't write data packet.\n");
        free(data_packet);
        return -1;
    }
```

```
54
55      free(data_packet);
56
57      return 0;
58  }
59
60  int app_send_control_packet(int ctrl_flag, uint32_t file_size, const char* filename){
61
62      uint32_t ctrl_packet_size = sizeof(uint32_t) + strlen(filename) + NUM_FIELDS_CTRL;
63      uint8_t * ctrl_packet = (uint8_t *) malloc(ctrl_packet_size);
64
65      ctrl_packet[CTRL] = ctrl_flag;
66      ctrl_packet[T1_SIZE] = FILE_SIZE_FLAG;
67      ctrl_packet[L1_SIZE] = sizeof(uint32_t);
68
69      ctrl_packet[V1_SIZE] = file_size & OFFSET;
70      ctrl_packet[V1_SIZE + 1] = (file_size >> 8) & OFFSET;
71      ctrl_packet[V1_SIZE + 2] = (file_size >> 16) & OFFSET;
72      ctrl_packet[V1_SIZE + 3] = (file_size >> 24) & OFFSET;
73
74      ctrl_packet[T2_NAME] = FILE_NAME_FLAG;
75      ctrl_packet[L2_NAME] = strlen(filename);
76
77      memcpy(ctrl_packet + V2_NAME, filename, strlen(filename));
78
79      if(ll_write(fileDescriptor, ctrl_packet, ctrl_packet_size) < 0){
80          fprintf(stderr, "Error: Couldn't write control packet.\n");
81          free(ctrl_packet);
82          return -1;
83      }
84
85      free(ctrl_packet);
86
87      return 0;
88  }
89
90  int app_send_file(char* path){
91
92      FILE* file;
93      unsigned int size = 0, sequence_number = 0;
94
95      if((file = fopen(path, "rb")) == NULL){
96          fprintf(stderr, "Error: Couldn't open file.\n");
97          return -1;
98      }
99
100     fseek(file, 0, SEEK_END);
101     uint32_t file_size = ftell(file);
102     fseek(file, 0, SEEK_SET);
103
104     char *filename = basename(path) + '\0';
105
106     if(app_send_control_packet(START_CTRL, file_size, path) < 0) return -1;
107
108     char *data_buffer = (char*)malloc(DATA_PACKET_MAX_SIZE);
109
110     while ((size = fread(data_buffer, sizeof(char),DATA_PACKET_MAX_SIZE,file)) > 0){
111
112         if(app_send_data_packet(data_buffer, size,sequence_number)) return -1;
113
114         sequence_number++;
115     }
```

```
116
117     if(app_send_control_packet(END_CTRL, file_size, filename) < 0) return -1;
118
119     return 0;
120 }
121
122 int app_receive_data_packet(char * data, int sequence_number){
123
124     uint8_t * data_packet = (uint8_t*) malloc(DATA_PACKET_MAX_SIZE + NUM_FIELDS_DATA);
125
126     if(ll_read(fileDescriptor, data_packet) < 0){
127         fprintf(stderr, "Error: Couldn't read data.\n");
128         free(data_packet);
129         return -1;
130     }
131
132     if(data_packet[CTRL] != DATA_CTRL){
133         fprintf(stderr, "Error: Control Field doesn't match.");
134         free(data_packet);
135         return -1;
136     }
137
138     if(data_packet[N_DATA] != (sequence_number % 255)){
139         fprintf(stderr, "Error: Sequence Number doesn't match.");
140         free(data_packet);
141         return -1;
142     }
143
144     int K = 256 * data_packet[L2_DATA] + data_packet[L1_DATA];
145
146     if(K > DATA_PACKET_MAX_SIZE){
147         fprintf(stderr, "Error: Data Recieved is larger than the maximum data size.");
148         free(data_packet);
149         return -1;
150     }
151
152     memcpy(data, data_packet + 4, K);
153
154     free(data_packet);
155
156     return K;
157 }
158
159 int app_receive_control_packet(int ctrl_flag, unsigned int * file_size, char* filename){
160
161     uint8_t ctrl_packet[NUM_FIELDS_CTRL + 255 + sizeof(unsigned int)];
162
163     if(ll_read(fileDescriptor, ctrl_packet) < 0){
164         fprintf(stderr, "Error: Couldn't read control packet.\n");
165         return -1;
166     }
167
168     if(ctrl_packet[CTRL] != ctrl_flag){
169         fprintf(stderr, "Error: Control Field doesn't match.\n");
170         return -1;
171     }
172
173     if(ctrl_packet[T1_SIZE] != FILE_SIZE_FLAG){
174         fprintf(stderr, "Error: Type of parameter doesn't match.\n");
175         return -1;
176     }
177
```

```
178     if(ctrl_packet[L1_SIZE] > sizeof(uint32_t)){
179         fprintf(stderr, "Error: File size in octates doesn't match (> 4).\n");
180         return -1;
181     }
182 
183     if(ctrl_packet[T2_NAME] != FILE_NAME_FLAG){
184         fprintf(stderr, "Error: Type of parameter doesn't match.\n");
185         return -1;
186     }
187 
188     memcpy(file_size, ctrl_packet + V1_SIZE, ctrl_packet[L1_SIZE]);
189 
190     for(int i = 0; i < ctrl_packet[L2_NAME]; i++){
191         filename[i] = (char) ctrl_packet[V2_NAME + i];
192     }
193 
194     filename[(int) ctrl_packet[L2_NAME]] = '\0';
195 
196     return 0;
197 }
198 
199 int app_receive_file(){
200     FILE * file;
201 
202     unsigned int file_size_start;
203     char filename_start[255];
204 
205     unsigned int file_size_end;
206     char filename_end[255];
207 
208     int K = 0;
209     unsigned int sequence_number = 0, bytes_read = 0;
210 
211     if(app_receive_control_packet(START_CTRL, &file_size_start, filename_start) < 0)
    return -1;
212 
213     if((file = fopen(filename_start, "wb")) == NULL){
214         fprintf(stderr, "Error: Couldn't open file.\n");
215         return -1;
216     }
217 
218     char *data_buffer = (char*)malloc(DATA_PACKET_MAX_SIZE);
219 
220     while(file_size_start > bytes_read){
221 
222         if((K = app_receive_data_packet(data_buffer,sequence_number)) < 0) {
223             fprintf(stderr, "Error: Couldn't receive data packet.\n");
224             return -1;
225         };
226 
227         fwrite(data_buffer, sizeof(uint8_t), K, file);
228 
229         bytes_read += K;
230         sequence_number++;
231 
232     }
233 
234     fclose(file);
235 
236     free(data_buffer);
237 
238     if(app_receive_control_packet(END_CTRL, &file_size_end, filename_end) < 0) return -1;
```

```
239
240     if(file_size_start != file_size_end || strcmp(filename_start, filename_end) != 0){
241         fprintf(stderr, "Error: Start and End Control packets don't match.");
242     }
243
244     return 0;
245 }
```

Listing 2: app.c

A.2 State Machines

```
1  /**
2   * @brief State Machine Communication
3   */
4
5
6  #ifndef _STATE_MACHINE_H_
7  #define _STATE_MACHINE_H_
8
9  #include <stdint.h>
10
11 /**
12  *
13  * @brief States to receive U-frame
14 */
15 typedef enum {
16     U_START,
17     U_FLAG_RCV,
18     U_A_RCV,
19     U_C_RCV,
20     U_BCC_OK,
21     U_END
22 } u_states_t;
23
24 /**
25  *
26  * @brief Performs state transitions in U-frames
27  *
28  * @param state Current state of the state machine
29  * @param byte Byte value of the current state
30  * @param a_rcv Pointer where the Address Field value will be stored
31  * @param c_rcv Pointer where the Control Field value will be stored
32  *
33  * @return 0 on success, -1 otherwise
34 */
35 int u_state_trans(u_states_t *state, uint8_t byte, uint8_t * a_rcv, uint8_t * c_rcv);
36
37 /**
38  *
39  * @brief States to receive I-frame
40 */
41 typedef enum {
42     I_START,
43     I_FLAG_RCV,
44     I_A_RCV,
45     I_C_RCV,
46     I_DATA,
47     I_END,
48     I_REP,
49     I_REP_END
50 } i_states_t;
51
```

```c
/**
 *
 * @brief Performs state transitions in I-frames
 *
 * @param state Current state of the state machine
 * @param byte Byte value of the current state
 * @param sz Current buffer index to store the data byte value
 * @param buf Buffer where the data byte value will be written
 *
 * @return 0 on success, -1 otherwise
*/
int i_state_trans(i_states_t *state, uint8_t byte, int *sz, uint8_t* buf);

/**
 *
 * @brief States to receive S-frame
*/
typedef enum {
    S_START,
    S_FLAG_RCV,
    S_A_RCV,
    S_C_RCV,
    S_BCC_OK,
    S_END
} s_states_t;

/**
 *
 * @brief Performs state transitions in S-frames
 *
 * @param state Current state of the state machine
 * @param byte Byte value of the current state
 * @param a_rcv Pointer where the Address Field value will be stored
 * @param c_rcv Pointer where the Control Field value will be stored
 *
 * @return 0 on success, -1 otherwise
*/
int s_state_trans(s_states_t *state, uint8_t byte, uint8_t * a_rcv, uint8_t * c_rcv);

#endif // _STATE_MACHINE_H
```

Listing 3: `state_machine.h`

```c
#include "../header/state_machine.h"

#include <stdint.h>

#include "../header/flag.h"
#include "../header/com.h"

int u_state_trans(u_states_t *state, uint8_t byte, uint8_t * a_rcv, uint8_t * c_rcv){
    switch(*state){
                case U_START:
                    if(byte == MSG_FLAG) *state = U_FLAG_RCV;
                    break;
                case U_FLAG_RCV:
                    if(byte == MSG_A_RECV || byte == MSG_A_SEND) {
                        *state = U_A_RCV;
                        *a_rcv = byte;
                    } else if (byte == MSG_FLAG) *state = U_FLAG_RCV;
                    else state = U_START;
                    break;
```

```
20              case U_A_RCV:
21                  if(byte == MSG_C_UA) {
22                      *state = U_C_RCV;
23                      *c_rcv = byte;
24                  } else if (byte == MSG_FLAG) *state = U_FLAG_RCV;
25                  else state = U_START;
26                  break;
27              case U_C_RCV:
28                  if(byte == (*a_rcv ^ *c_rcv)) *state = U_BCC_OK;
29                  else if (byte == MSG_FLAG) *state = U_FLAG_RCV;
30                  else *state = U_START;
31                  break;
32              case U_BCC_OK:
33                  if(byte == MSG_FLAG) *state = U_END;
34                  else *state = U_START;
35                  break;
36             case U_END:
37                  return -1;
38          }
39          return 0;
40  }
41
42  int i_state_trans(i_states_t *state, uint8_t byte, int *sz, uint8_t * buf){
43
44      switch(*state){
45              case I_START:
46                  if(byte == MSG_FLAG) *state = I_FLAG_RCV;
47                  break;
48              case I_FLAG_RCV:
49                  if(byte == MSG_A_SEND) *state = I_A_RCV;
50                  else if (byte == MSG_FLAG) *state = I_FLAG_RCV;
51                  else state = I_START;
52                  break;
53              case I_A_RCV:
54                  if(byte == MSG_C_I(Ns))  *state = I_C_RCV;
55                  else if (byte == MSG_C_I((Ns + 1)%2)) *state = I_REP;
56                  else if (byte == MSG_FLAG) *state = I_FLAG_RCV;
57                  else state = I_START;
58                  break;
59              case I_C_RCV:
60                  if(byte == (MSG_A_SEND ^ MSG_C_I(Ns))) *state = I_DATA;
61                  else if (byte == MSG_FLAG) *state = I_FLAG_RCV;
62                  else *state = I_START;
63                  break;
64              case I_DATA:
65                  if(byte == MSG_FLAG) *state = I_END;
66                  else {
67                      buf[(*sz)++] = byte;
68                  }
69                  break;
70              case I_REP:
71                  if(byte == MSG_FLAG) *state = I_REP_END;
72              case I_END:
73                  return -1;
74              case I_REP_END:
75                  return -1;
76          }
77          return 0;
78  }
79
80  int s_state_trans(s_states_t *state, uint8_t byte, uint8_t * a_rcv, uint8_t * c_rcv){
81      switch(*state){
```

```
82                case S_START:
83                    if(byte == MSG_FLAG) *state = S_FLAG_RCV;
84                    break;
85                case S_FLAG_RCV:
86                    if(byte == MSG_A_RECV || byte == MSG_A_SEND) {
87                        *state = S_A_RCV;
88                        *a_rcv = byte;
89                    } else if (byte == MSG_FLAG) *state = S_FLAG_RCV;
90                    else state = S_START;
91                    break;
92                case S_A_RCV:
93                    if(byte == MSG_C_SET || byte == MSG_C_DISC || byte == MSG_C_REJ(Nr) ||
    byte == MSG_C_RR(Nr)) {
94                        *state = S_C_RCV;
95                        *c_rcv = byte;
96                    } else if (byte == MSG_FLAG) *state = S_FLAG_RCV;
97                    else state = S_START;
98                    break;
99                case S_C_RCV:
100                   if(byte == (*a_rcv ^ *c_rcv)) *state = S_BCC_OK;
101                   else if (byte == MSG_FLAG) *state = S_FLAG_RCV;
102                   else *state = S_START;
103                   break;
104               case S_BCC_OK:
105                   if(byte == MSG_FLAG) *state = S_END;
106                   else *state = S_START;
107                   break;
108              case S_END:
109                   return -1;
110           }
111           return 0;
112   }
```

Listing 4: `state_machine.c`

A.3 Linked Layer Communication

```
1  /**
2   * @brief Link Layer Communication
3   */
4
5  #ifndef _COM_H_
6  #define _COM_H_
7
8  #include "ll.h"
9
10 // Sequence Numbers of the Receiver and Transmitter
11 int Ns;
12 int Nr;
13
14
15 /**
16 @brief Sends commands
17  *
18  * @param fd File Descriptor of the Serial Port to write to
19  * @param status Status of the application, Transmitter or Receiver
20  * @param ctrl Control Field to be sent, SET and DISC
21  *
22  * @return Number of bytes written to the serial port on success, -1 otherwise
23 */
24 int send_command(int fd, status_t status, uint8_t ctrl);
25
26 /**
```

```
 *
 * @brief Sends responses
 *
 * @param fd File Descriptor of the Serial Port to write to
 * @param status Status of the application, Transmitter or Receiver
 * @param ctrl Control Field to be sent, RR, REJ and UA
 *
 * @return Number of bytes written to the serial port on success, -1 otherwise
*/
int send_response(int fd, status_t status, uint8_t ctrl);

/**
 *
 * @brief Sends I frame
 *
 * @param fd File Descriptor of the Serial Port to write to
 * @param buf Data Buffer with the data to be transferred
 * @param size Size of the data to be transferred in the frame
 *
 * @return The size of the data field sent on success, -1 otherwise
*/
int send_I_FRAME(int fd, uint8_t * buf, int size);

/**
 * @brief Receives U frame
 *
 * @param fd File Descriptor of the Serial Port to read from
 * @param a_rcv Pointer where the Address Field value will be stored
 * @param c_rcv Pointer where the Control Field value will be stored
 *
 * @return 0 on success, -1 otherwise
*/
int receive_U(int fd, uint8_t* a_rcv, uint8_t* c_rcv);

/**
 * @brief Receives S frame
 *
 * @param fd File Descriptor of the Serial Port to read from
 * @param a_rcv Pointer where the Address Field value will be stored
 * @param c_rcv Pointer where the Control Field value will be stored
 *
 * @return 0 on success, -1 otherwise
*/
int receive_S(int fd, uint8_t* a_rcv, uint8_t* c_rcv);

/**
 * @brief Receives I frame
 *
 * @param fd File Descriptor of the Serial Port to read from
 * @param a_rcv Pointer where the Address Field value will be stored
 * @param c_rcv Pointer where the Control Field value will be stored
 *
 * @return The size of the data field received on success, -1 otherwise
*/
int receive_I(int fd, uint8_t* buffer);

#endif // _COM_H
```

Listing 5: `com.h`

```
#include "../header/com.h"

```

```c
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdint.h>

#include "../header/flag.h"
#include "../header/state_machine.h"
#include "../header/utils.h"

int send_command(int fd, status_t status, uint8_t ctrl){
    int res;

    uint8_t msg_buf[5];
    msg_buf[0] = MSG_FLAG;
    msg_buf[1] = (status == RECEIVER ? MSG_A_RECV : MSG_A_SEND);
    msg_buf[2] = ctrl;
    msg_buf[3] = msg_buf[1] ^ msg_buf[2];
    msg_buf[4] = MSG_FLAG;

    res = write(fd, msg_buf, 5);
    if(res < 0){
        fprintf(stderr, "Error: Write.\n");
        return -1;
    }
    return res;
}

int send_response(int fd, status_t status, uint8_t ctrl){
    int res;

    uint8_t msg_buf[5];
    msg_buf[0] = MSG_FLAG;
    msg_buf[1] = (status == RECEIVER ? MSG_A_SEND : MSG_A_RECV);
    msg_buf[2] = ctrl;
    msg_buf[3] = msg_buf[1] ^ msg_buf[2];
    msg_buf[4] = MSG_FLAG;

    res = write(fd, msg_buf, 5);
    if(res < 0){
        fprintf(stderr, "Error: Write.\n");
        return -1;
    }
    return res;
}


int send_I_FRAME(int fd, uint8_t * buf, int size){
    int res;

    uint8_t header[4];
    header[0] = MSG_FLAG;
    header[1] = MSG_A_SEND;
    header[2] = MSG_C_I(Ns);
    header[3] = header[2] ^ header[1];

    res = write(fd, header, 4);
    if(res < 0){
        fprintf(stderr, "Error: Write.\n");
        return -1;
    }

    uint8_t bcc2 = bcc_buf(buf, size);
```

```
    buf[size] = bcc2;

    uint8_t stuff_buf[2*MAX_SIZE];
    int stuff_sz = stuff_frame(stuff_buf, buf, size+1);

    res = write(fd, stuff_buf, stuff_sz);
    if(res < 0){
        fprintf(stderr, "Error: Write.\n");
        return -1;
    }

    uint8_t tail[1];
    tail[0] = MSG_FLAG;

    res = write(fd, tail, 1);
    if(res < 0){
        fprintf(stderr, "Error: Write.\n");
        return -1;
    }

    return size;
}

int receive_U(int fd, uint8_t *a_rcv, uint8_t *c_rcv){
    u_states_t state = U_START;
    uint8_t byte;

    while(state != U_END){
        if (read(fd, &byte, 1) != 1) {
            fprintf(stderr, "Error: Exceeded read time.\n");
            return -1;
        }
        if (u_state_trans(&state, byte, a_rcv, c_rcv) < 0) return -1;
    }

    return 0;
}

int receive_S(int fd, uint8_t* a_rcv, uint8_t* c_rcv) {
    s_states_t state = S_START;
    uint8_t byte;

    while(state != S_END){
        if (read(fd, &byte, 1) != 1) {
            fprintf(stderr, "Error: Exceeded read time.\n");
            return -1;
        }
        if (s_state_trans(&state, byte, a_rcv, c_rcv) < 0) return -1;
    }

    return 0;
}

int receive_I(int fd, uint8_t* buffer) {
    i_states_t state = I_START;
    uint8_t byte;
    uint8_t buf[2*MAX_SIZE];
    int sz = 0;
    while (state != I_END){
        while(state != I_END && state != I_REP_END){
            if (read(fd, &byte, 1) != 1) {
                fprintf(stderr, "Error: Exceeded read time.\n");
```

```
                return -1;
            }
            if (i_state_trans(&state, byte, &sz, buf) < 0) return -1;
        }
        if(state == I_REP_END){
            send_response(fd, RECEIVER, MSG_C_RR((Nr + 1)%2));
            fprintf(stderr, "Error: Received repeated Information Frame, sending RR.\n");
            state = I_START;
        }
    }

    uint8_t buf_destuff [MAX_SIZE];
    int destuff_sz = destuff_frame(buf_destuff, buf, sz);

    if(buf_destuff[destuff_sz - 1] != bcc_buf(buf_destuff, destuff_sz-1)){
        fprintf(stderr, "Error: Wrong BCC2.\n");
        return -1;
    }

    memcpy(buffer, buf_destuff, destuff_sz-1);

    return sz-1;
}
```

Listing 6: `com.c`

A.4 Linked Layer

```
/**
 * @brief Link Layer
 */

#ifndef _LL_H_
#define _LL_H_

#include <stdint.h>

/**
 *
 * @brief Application Status structure
*/
typedef enum {
    TRANSMITTER, // Transfers the file
    RECEIVER // Receives the File
} status_t;

#define MAX_SIZE 5001

/**
 *
 * @brief Initializes the Communication Protocol
 *
 * @param fd File descriptor corresponding to the serial port
 * @param status Status of the application, Transmitter or Receiver
 *
 * @return 0 on success, -1 otherwise
*/
int ll_open(int fd, status_t status);

/**
 *
 * @brief Reads from the Serial Port
 *
```

```
 * @param fd File descriptor corresponding to the serial port
 * @param buffer Buffer where the data read will be stored
 *
 * @return Number of bytes read on success, a negative number otherwise
*/
int ll_read(int fd, uint8_t * buffer);

/**
 *
 * @brief Writes to the Serial Port
 *
 * @param fd File descriptor corresponding to the serial port
 * @param buffer Buffer where the data will be written
 * @param length Size of the data to write
 *
 * @return Number of bytes written on success, a negative number otherwise
*/
int ll_write(int fd, uint8_t * buffer, int length);

/**
 *
 * @brief
 *
 * @param fd File descriptor corresponding to the serial port
 *
 * @return 0 if the present status is RECEIVER and the number of bytes written
 * sending UA if the status is TRANSMITTER, a negative number otherwise
*/
int ll_close(int fd);

#endif // _LL_H
```

Listing 7: `ll.h`

```
#include "../header/ll.h"

#include <stdio.h>

#include "../header/com.h"
#include "../header/flag.h"

status_t status;

int ll_open(int fd, status_t st){
    status = st;
    if(status == TRANSMITTER){
        int tries;
        for(tries = 0; tries < NUM_RETRANS; tries++){
            if (send_command(fd,status, MSG_C_SET) < 0){
                fprintf(stderr, "Error: Sending SET.\n");
                continue;
            }

            uint8_t a_rcv, c_rcv;
            if (receive_U(fd, &a_rcv, &c_rcv) < 0){
                fprintf(stderr, "Error: Couldn't receive UA, retrying.\n");
                continue;
            } else  if(a_rcv == MSG_A_SEND && c_rcv == MSG_C_UA){
                break;
            } else {
                fprintf(stderr, "Error: Received wrong UA.\n");
            }
```

```
29          }
30          if(tries == NUM_RETRANS) {
31              fprintf(stderr, "Error: Exceeded number of tries in llopen.\n");
32              return -1;
33          }
34
35      } else if (status == RECEIVER){
36          int tries;
37          for(tries = 0; tries < NUM_RETRANS; tries++){
38              uint8_t a_rcv, c_rcv;
39              if (receive_S(fd, &a_rcv, &c_rcv) < 0){
40                  fprintf(stderr, "Error: Couldn't receive SET, retrying.\n");
41                  continue;
42              }
43
44              if(a_rcv == MSG_A_SEND && c_rcv == MSG_C_SET){
45                  if (send_response(fd, status, MSG_C_UA) < 0){
46                      fprintf(stderr, "Error: Sending UA.\n");
47                      return -1;
48                  }
49                  break;
50              } else {
51                  fprintf(stderr, "Error: Received wrong SET.\n");
52              }
53          }
54          if(tries == NUM_RETRANS) {
55              fprintf(stderr, "Error: Exceeded number of tries in llopen.\n");
56              return -1;
57          }
58      }
59      Ns = 1; Nr = 0;
60      return 0;
61  }
62
63  int ll_read(int fd, uint8_t * buffer) {
64      Ns = (Ns + 1)%2;
65      Nr = (Nr + 1)%2;
66
67      int tries,res;
68      for(tries = 0; tries < NUM_RETRANS; tries++){
69          res = receive_I(fd, buffer);
70
71          if(res <= 0){
72              send_response(fd, status, MSG_C_REJ(Nr));
73              fprintf(stderr, "Error: Couldn't receive Information Frame, retrying.\n");
74          } else {
75              send_response(fd, status, MSG_C_RR(Nr));
76              break;
77          }
78      }
79      if(tries == NUM_RETRANS) {
80              fprintf(stderr, "Error: Exceeded number of tries in llread.\n");
81              return -1;
82      }
83      return res;
84  }
85  int ll_write(int fd, uint8_t * buffer, int length){
86      Ns = (Ns + 1)%2;
87      Nr = (Nr + 1)%2;
88
89      int tries,res;
90      for(tries = 0; tries < NUM_RETRANS; tries++){
```

```
91          res = send_I_FRAME(fd, buffer, length);
92
93          if(res != length){
94              fprintf(stderr, "Error: Sending Information Frame, retrying.\n");
95              continue;
96          }
97
98          uint8_t a_rcv, c_rcv;
99          res = receive_S(fd, &a_rcv, &c_rcv);
100
101         if(res < 0){
102             fprintf(stderr, "Error: Couldn't receive answer, retrying.\n");
103             continue;
104         }
105         if(a_rcv == MSG_A_SEND){
106             if(c_rcv == MSG_C_RR(Nr)) break;
107             else if(c_rcv == MSG_C_REJ(Nr)) {
108                 fprintf(stderr, "Error: Received REJ, retrying.\n");
109                 continue;
110             }
111         }
112     }
113     if(tries == NUM_RETRANS) {
114             fprintf(stderr, "Error: Exceeded number of tries in llwrite.\n");
115             return -1;
116     }
117     return res;
118 }
119
120 int ll_close(int fd){
121     int tries,res;
122     uint8_t a_rcv, c_rcv;
123     if(status == TRANSMITTER){
124         for(tries = 0; tries < NUM_RETRANS; tries++){
125             if (send_command(fd,status, MSG_C_DISC) < 0){
126                 fprintf(stderr, "Error: Sending DISC.\n");
127                 return -1;
128             }
129
130             if((res = receive_S(fd, &a_rcv, &c_rcv)) < 0) {
131                 fprintf(stderr, "Error: Couldn't receive DISC, retrying.\n");
132                 continue;
133             }
134             else if(a_rcv == MSG_A_RECV && c_rcv == MSG_C_DISC){
135                 break;
136             }else {
137                 fprintf(stderr, "Error: Received wrong DISC, retrying.\n");
138                 continue;
139             }
140         }
141         if(tries == NUM_RETRANS) {
142             fprintf(stderr, "Error: Exceeded number of tries in llclose.\n");
143             return -1;
144         }
145
146         res = send_response(fd, status, MSG_C_UA);
147     } else if (status == RECEIVER){
148         for(tries = 0; tries < NUM_RETRANS; tries++){
149             res = receive_S(fd, &a_rcv, &c_rcv);
150
151             if(res < 0) {
152                 fprintf(stderr, "Error: Couldn't receive DISC, retrying.\n");
```

```
153                 continue;
154             }
155             if(a_rcv == MSG_A_SEND && c_rcv == MSG_C_DISC){
156                 if((res = send_command(fd, status, MSG_C_DISC)) < 0){
157                     fprintf(stderr, "Error: Sending DISC.\n");
158                     return -1;
159                 }
160                 break;
161             } else {
162                 fprintf(stderr, "Error: Received wrong DISC.\n");
163                 continue;
164             }
165         }
166         if(tries == NUM_RETRANS) {
167             fprintf(stderr, "Error: Exceeded number of tries in llclose.\n");
168             return -1;
169         }
170         res = receive_U(fd, &a_rcv, &c_rcv);
171 
172         if(res < 0) {
173             fprintf(stderr, "Error: Couldn't receive UA.\n");
174             return -1;
175         }
176 
177         if(a_rcv != MSG_A_RECV || c_rcv != MSG_C_UA){
178             fprintf(stderr, "Error: Received wrong UA.\n");
179             return -1;
180         }
181     }
182     return res;
183 }
```

Listing 8: `ll.c`

A.5 Serial Port Configuration

```
1  /**
2   * @brief Serial Port Configuration
3   */
4 
5  #ifndef _CONFIG_H_
6  #define _CONFIG_H_
7 
8  //Baudrate value
9  #define BAUDRATE B38400
10 
11 /**
12  *
13  * @brief Sets the serial port configuration
14  *
15  * @param serial_port Port name to be configured
16  *
17  * @return The file descriptor corresponding to the serial port on success, -1 otherwise
18 */
19 int set_config(char * serial_port);
20 
21 /**
22  *
23  * @brief Resets the serial port's configuration
24  *
25  * @param fd File descriptor of the serial port
26 */
27 void reset_config(int fd);
```

```
28
29
30 #endif // _CONFIG_H
```

Listing 9: `config.h`

```
1  #include "../header/config.h"
2
3  #include <fcntl.h>
4  #include <termios.h>
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <strings.h>
8  #include <unistd.h>
9
10 struct termios oldtio,newtio;
11
12 int set_config(char * serial_port){
13     int fd = open(serial_port, O_RDWR | O_NOCTTY );
14
15     if (fd <0) {perror(serial_port); exit(-1); }
16
17     if ( tcgetattr(fd,&oldtio) == -1) { /* save current port settings */
18       perror("tcgetattr");
19       exit(-1);
20     }
21
22     bzero(&newtio, sizeof(newtio));
23     newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
24     newtio.c_iflag = IGNPAR;
25     newtio.c_oflag = 0;
26
27     /* set input mode (non-canonical, no echo,...) */
28     newtio.c_lflag = 0;
29
30     newtio.c_cc[VTIME]    = 30;   /* inter-character timer unused */
31     newtio.c_cc[VMIN]     = 0;   /* blocking read until 5 chars received */
32   /*
33     VTIME e VMIN devem ser alterados de forma a proteger com um temporizador a
34     leitura do(s)  p r ximo (s) caracter(es)
35   */
36     tcflush(fd, TCIOFLUSH);
37
38     if ( tcsetattr(fd,TCSANOW,&newtio) == -1) {
39       perror("tcsetattr");
40       exit(-1);
41     }
42
43     return fd;
44 }
45
46 void reset_config(int fd) {
47     sleep(2);
48
49     if (tcsetattr(fd, TCSANOW, &oldtio) == -1) {
50         perror("tcsetattr reset");
51         exit(-1);
52     }
53
54     close(fd);
55 }
```

Listing 10: `config.c`

A.6 Transmitter and Receiver Main

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

#include "../header/config.h"
#include "../header/app.h"

int main(int argc, char ** argv){
    if ( (argc < 2) ||
         ((strcmp("/dev/ttyS0", argv[1])!=0) &&
          (strcmp("/dev/ttyS1", argv[1])!=0) &&
          (strcmp("/dev/ttyS11", argv[1])!=0) &&
          (strcmp("/dev/ttyS10", argv[1])!=0) ) ) {
      printf("Usage:\tnserial SerialPort\n\tex: nserial /dev/ttyS1\n");
      exit(1);
    }

    int fd = set_config(argv[1]);

    if(application(fd, TRANSMITTER, argv[2]) != 0){
        fprintf(stderr, "Error transmitting File");
        exit(-1);
    }

    reset_config(fd);
}
```

Listing 11: `transmitter.c`

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

#include "../header/config.h"
#include "../header/app.h"

int main(int argc, char ** argv){
    if ( (argc < 2) ||
         ((strcmp("/dev/ttyS0", argv[1])!=0) &&
          (strcmp("/dev/ttyS1", argv[1])!=0) &&
          (strcmp("/dev/ttyS11", argv[1])!=0) &&
          (strcmp("/dev/ttyS10", argv[1])!=0) ) ) {
      printf("Usage:\tnserial SerialPort\n\tex: nserial /dev/ttyS1\n");
      exit(1);
    }

    int fd = set_config(argv[1]);

     if(application(fd, RECEIVER, argv[2]) != 0){
        fprintf(stderr, "Error recieving File");
        exit(-1);
    }

    reset_config(fd);
}
```

Listing 12: `receiver.c`

A.7 Stuffing Utilitaries

```c
/**
 * @brief Stuffing Utilitaries
 */

#ifndef _UTILS_H_
#define _UTILS_H_

#include <stdint.h>

/**
 *
 * @brief Destuffs frame
 *
 * @param destuff_buf Pointer that holds the buffer after destuffing
 * @param buf Buffer to destuff
 * @param sz Size of the buffer to destuff
 *
 * @return Size of destuffed buffer
*/
int destuff_frame(uint8_t * destuff_buf, uint8_t * buf, int sz);

/**
 *
 * @brief Stuffs frame
 *
 * @param stuf_buf Pointer that holds the buffer after stuffing
 * @param buf Buffer to stuff
 * @param sz Size of the buffer to stuff
 *
 * @return Size of stuffed buffer
*/
int stuff_frame(uint8_t * stuf_buf, uint8_t * buf, int sz);

/**
 *
 * @brief Calculates the BCC2 of a data buffer
 *
 * @param buf Data buffer to calculate the BCC2 from
 * @param sz Size of the data buffer
 *
 * @return Value of the BCC2
*/
uint8_t bcc_buf(uint8_t * buf, int sz);

#endif // _UTILS_H
```

Listing 13: `utils.h`

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


#include "../header/flag.h"


int destuff_frame(uint8_t * destuff_buf, uint8_t * buf, int sz){
    int destuff_sz = 0;
    for(int i = 0; i < sz; i++){
        if(buf[i] == MSG_ESC){
            destuff_buf[destuff_sz++] = MSG_STUFF(buf[++i]);
        } else destuff_buf[destuff_sz++] = buf[i];
    }
    return destuff_sz;
```

```
15
16 }
17
18 int stuff_frame(uint8_t * stuff_buf, uint8_t * buf, int sz){
19     int stuff_sz = 0;
20     for(int i = 0; i < sz; i++){
21         if(buf[i] == MSG_FLAG || buf[i] == MSG_ESC){
22             stuff_buf[stuff_sz++] = MSG_ESC;
23             stuff_buf[stuff_sz++] = MSG_STUFF(buf[i]);
24         } else stuff_buf[stuff_sz++] = buf[i];
25     }
26     return stuff_sz;
27 }
28
29 uint8_t bcc_buf(uint8_t * buf, int sz){
30     uint8_t res = 0;
31     for(int i = 0; i < sz; i++){
32         res ^= buf[i];
33     }
34     return res;
35 }
```

Listing 14: `utils.c`

A.8 Flag Macros

```
1  /**
2   * @brief Link Layer Flag Macros
3   */
4
5  #ifndef _FLAG_H_
6  #define _FLAG_H_
7
8  #define MSG_FLAG          0x7E
9  #define MSG_A_SEND        0x03
10 #define MSG_A_RECV        0x01
11 #define MSG_C_SET         0x03
12 #define MSG_C_DISC        0x0B
13 #define MSG_C_UA          0x07
14 #define MSG_C_I(N)        ((N) << 6)
15 #define MSG_C_RR(N)       (0x05  | ((N) << 7))
16 #define MSG_C_REJ(N)      (0x01 | ((N) << 7))
17 #define MSG_ESC           0x7D
18 #define MSG_STUFF(N)      ((N) ^  0x20)
19
20
21 #define NUM_RETRANS 3
22
23 #endif // _FLAG_H
```

Listing 15: `flag.h`

Apêndice B Estatísticas

Os valores predefinidos são $T_{prop} = 0$ s, $C = 38400$ bits/s, $S_p = 128$ B, $p_e = 0$, e o ficheiro enviado foi pinguim.gif (10968 bytes), o tempo de excução é medido em segundos.

B.1 Variação da probabilidade de erro

p_e	t_e	S
0	3.383284	0.67537931
10	3.579960	0.63827529
20	3.924144	0.5822926
30	4.806502	0.4753977
40	5.624793	0.40623717
50	6.464980	0.3534427
60	8.493095	0.26904209
70	11.437231	0.19978612
80	16.273438	0.14041286
90	29.733669	0.07684891

B.2 Variação de a

t_{prop}	t_e	S
0	3.383284	0.67537931
0.10	9.020170	0.25332117
0.20	18.020442	0.12680044
0.30	27.020345	0.08456591
0.40	36.020752	0.06343566
0.50	45.021054	0.05075403
0.60	54.021304	0.04229813
0.70	63.021822	0.03625728
0.80	72.022278	0.03172629
0.90	81.022747	0.02820196

B.3 Variação da baudrate

C	t_e	S
2400	53.168603	0.68762386
4800	26.596021	0.68732086
9600	13.311601	0.68661914
19200	6.671105	0.685043931
38400	3.383284	0.67537931
57600	2.236670	0.68107201

B.4 Variação do tamanho dos pacotes enviados

S_p	t_e	S
128	3.383284	0.67537931
256	3.125657	0.7310463
512	3.013858	0.75816445
1024	2.962073	0.77141921
2048	2.938361	0.77764441
4096	2.922868	0.7817664