Relatório do 1º Trabalho Laboratorial

Grupo 5

up201806551@fe.up.pt Beatriz Costa Silva Mendes
up201806528@fe.up.pt Clara Alves Martins

10 de novembro de 2020

Redes de Computadores - 2020/21 - MIEIC

Professor das Aulas Laboratoriais: Manuel Alberto Pereira Ricardo

Índice

Sumário

Este relatório tem como objetivo resumir todo o trabalho que foi realizado ao longo das aulas práticas. Este consite na criação de uma aplicação que auxilia na transferência de ficheiros através de uma porta de série, tendo-se, assim, criado diversas funções e estruturas de dados que tornam esta transferência possível.

Após vários testes realizados em casa através do comando fornecido de `socat`:

```
sudo socat -d -d PTY,link=/dev/ttyS0,mode=777 PTY,link=/dev/ttyS1,mode=777)
```

, através das portas série do laboratório (RS-232) e ainda portas série diferentes, concluimos que o nosso programa transmite de maneira fiável um ficheiro de qualquer dimensão, mesmo quando surgem erros, através de uma porta série.

1 Introdução

O objetivo do trabalho que nos foi proposto é criar um protocolo de aplicação simples para realizar a transferência de um ficheiro, usando o serviço fiável oferecido pelo procolo de ligação de dados. Para estabelecer esta comunicação utilizamos as portas de série (portas de série RS-232 - comunicação assíncrona) que nos foram disponibilizadas ao longo das aulas práticas. Todo o trabalho foi implementado de acordo com a informação que nos foi fornecida nos guiões de trabalho.

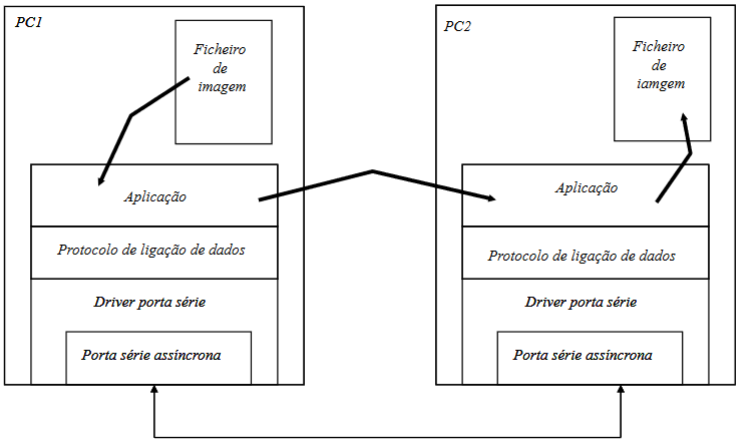

Figura 1: Aplicação Teste

O relatório apresenta a seguinte estrutura:

1. **Arquitetura**: blocos funcionais e interfaces da aplicação;
2. **Estrutura do Código**: APIs, principais estruturas de dados, principais funções e sua relação com a arquitetura;

3. **Casos de Uso Principais**: identificação dos casos de uso principais e sequências de chamadas de funções;
4. **Protocolo de Ligação Lógica**: identificação dos principais aspetos funcionais e descrição da estratégia de implementação destes aspetos com apresentação de extratos de código;
5. **Protocolo de Aplicação**: identificação dos principais aspetos funcionais e descrição da estratégia de implementação destes aspetos com apresentação de extratos de código;
6. **Validação**: descrição dos testes efectuados com apresentação quantificada dos resultados;
7. **Eficiência do Protocolo de Ligação de Dados**: caraterização estatística da eficiência do protocolo, feita com recurso a medidas sobre o código desenvolvido;
8. **Conclusões**: síntese da informação apresentada nas secções anteriores e reflexão sobre os objectivos de aprendizagem alcançados.

2 Arquitetura

2.1 Camadas

O programa desenvolvido pelo nosso grupo de trabalho tem duas camadas muito bem definidas: a **camada da aplicação** e a **camada da ligação de dados**.

A **camada da aplicação** é responsável pelo envio e receção de ficheiros. Relembrando a Figura 1 presente na secção anterior, esta camada encontra-se acima da camada de ligação de dados. Para além disso, esta ainda utiliza a interface disponibilizada pela camada de ligação de dados, chamando as suas funções para auxiliar no envio e receção de ficheiros. A camada da aplicação foi desenvolvida nos ficheiros `application_layer.c` e `file_interaction.c`.

A **camada da ligação lógica**, por outro lado, tem como principal objetivo estabelecer a ligação e assegurar a consistência do protocolo de ligação de dados que foi desenvolvido, sendo, assim, a camada de mais baixo nível da aplicação. Esta camada realiza a interação com a porta de série, fazendo a abertura, escrita, leitura e fecho desta, tratando ainda dos erros que poderão surgir na transmissão através da técnica de *byte stuffing*, garantindo assim a fiabilidade na transmissão dos ficheiros. Esta camada foi desenvolvida no ficheiro `logical_link.c`.

2.2 Interface

Na **interface** disponível para o utilizador, será apresentada informação relativamente ao estado do envio e da receção do ficheiro, nomeadamente se o recetor e o transmissor estão a ler ou a escrever tramas. Para além disso, também surgem mensagens de erro caso surja algum durante a transmissão dos ficheiros.

3 Estrutura do Código

Numa primeira fase, o nosso projeto tem dois ficheiros fundamentais: `write.c` e `read.c`. Estes ficheiros são responsáveis por interligar a camada de ligação de dados e a camada da aplicação de

modo a que a transferência de ficheiros seja possível. Posteriormente, os restantes ficheiros possuem funções e estruturas de dados que facilitam a transmissão, leitura e escrita de tramas.

3.1 Camada de Ligação de Dados

Estruturas de Dados:

Criação de uma *struct* `packet` para armazenar os *bytes* do pacote e o tamanho deste.

Consultar Anexo II.

Principais funções:

Consultar Anexo III.

Funções auxiliares:

Consultar Anexo IV.

3.2 Camada da Aplicação

Estruturas de Dados:

Criação de duas *structs*: `control_packet` e `data_packet`. A primeira possui a informação necessária para utilizar um pacote de controlo e a segunda para utilizar um pacote de informação.

Consultar Anexo V.

Principais funções:

Consultar Anexo VI.

Funções auxiliares:

Consultar Anexo VII.

4 Casos de Uso Principais

4.1 Transmissor (*Transmitter*)

O programa é executado em modo **transmissor**. Inicialmente, verifica-se se o ficheiro que se pretende transferir existe. Existindo, é estabelecida a ligação e, em seguida, vão sendo enviados pacotes de dados de acordo com o protocolo de ligação implementado. Após a conclusão do envio, a ligação é encerrada, terminando assim o programa. Em seguida apresenta-se a sequência de chamadas das funções principais:

1. `llopen`: abertura da porta de série para escrita e leitura, envio da trama SET e receção da trama UA;

2. `construct_control_packet`: construção do pacote de controlo com a flag de início da transmissão e os restantes parâmetros;
3. `llwrite`: construção da trama que inclui o pacote de controlo, escrita deste e verificação de erros;
4. Loop Principal:
    a) `read_from_file`: abertura do ficheiro que será transmitido e leitura deste;
    b) `construct_data_packet`: construção do pacote de informação com base no lido do ficheiro aberto anteriormente;
    c) `llwrite`: construção da trama que inclui o pacote de informação, escrita deste e verificação de erros;
5. `construct_control_packet`: construção do pacote de controlo com a flag de início da transmissão e os restantes parâmetros;
6. `llwrite`: construção da trama que inclui o pacote de controlo, escrita deste e verificação de erros;
7. `llclose`: envio da trama DISC, receção da trama DISC, envio da trama UA e fecho da porta de série.

4.2 Recetor (*Receiver*)

O programa é executado em modo **recetor**. É estabelecida a ligação com a porta de série. Seguidamente, começam a ser lidos os pacotes enviados pelo trasnmissor. No final da leitura, a ligação de dados é encerrada. Seguem-se as chamadas das funções principais por ordem de acontecimentos:

1. `llopen`: abertura da porta de série para escrita e leitura, receção da trama SET e envio da trama UA;
2. Loop Principal:
    a) `llread`: leitura das tramas que foram criadas pelo transmissor, quer sejam tramas com o pacote de controlo, quer sejam tramas com pacotes de informação, e verificação de erros;
        i. Se for lido um pacote de informação:
            A. `deconstruct_data_packet`: desconstrução do pacote de informação que foi enviado pela ligação de dados;
            B. `write_to_file`: escrita da informação recebida para o ficheiro;
        ii. Se for lido um pacote de controlo:
            A. `deconstruct_control_packet`: desconstrução do pacote de controlo que foi enviado pela ligação de dados, verificando se é o do início ou do fim da transmissão;
3. `llclose`: envio da trama DISC, receção da trama UA e fecho da porta de série.

5 Protocolo de Ligação Lógica

5.1 Aspetos Fundamentais e Estratégias de Implementação Adotadas

Estabelecimento/fecho da ligação lógica

No caso do transmissor (*transmitter*):

A ligação é estabelecida através do envio de uma trama SET e receção de uma trama UA.

O fecho da ligação acontece quando é enviada uma trama DISC, recebida uma trama DISC e enviada uma trama UA.

No caso do recetor (*receiver*):

A ligação é estabelecida através da receção de uma trama SET e envio de uma trama UA.

O fecho da ligação acontece quando é enviada uma trama DISC e recebida uma trama UA.

Consultar Anexo VIII.

Enviar/receber informação através da porta de série

O envio e receção de informação são tratados pelas funções `llwrite` e `llread` que se encontram no ficheiro `logical_link.c`. Estas mesmas funções acabam por fornecer a informação escrita/lida à camada aplicação.

Consultar Anexo IX.

Controlo de erros (BCC's e números de sequência)

O controlo de erros durante a transmissão de informação é feito através da verificação do BCC do cabeçalho e dos dados e através da utilização de números de sequência. Consultar Anexo X.

Confirmações

Após o processamento de qualquer trama, é enviado pelo recetor uma confirmação negativa (REJ) ou positiva (RR). A confirmação depende do número de sequência da trama e do resultado da verificação efetuada.

Consultar Anexo XI.

6 Protocolo de Aplicação

6.1 Aspetos Fundamentais e Estratégias de Implementação Adotadas

Fragmentação/desfragmentação de pacotes

A ligação por porta de série transmite a informação a partir de uma cadeia de bytes. Desta forma, ao transmitir, deveremos transformar a informação a ser transferida em cadeias de bytes. Da mesma forma, ao receber, deveremos ler as cadeias de bytes e, em seguida, retransformá-las em informação que possa ser usada e interpretada pelo resto do programa.

Consultar Anexo XII.

Envio/receção de pacotes de dados/controlo

São construídos pacotes de dados e de controlo ao longo do programa, sendo posteriormente enviados/recebidos.

Consultar Anexo XIII.

Leitura e escrita de ficheiros

Consultar Anexo XIV.

Criação de ficheiros

O ficheiro será criado na primeira chamada da função de escrita para ficheiros.

Consultar Anexo XV.

7 Validação

De modo a verificar a robustez do nosso programa, efetuamos um conjunto de testes. Estes testes incluiram:

- a transmissão de diferentes ficheiros;
- a transmissão com baudrates distintas;
- a transmissão com tamanho de informação útil variável;
- a introdução de atrasos na transmissão;
- a introdução de erros simulados;
- a interrupção temporária da transmissão;

- a interferência na transmissão, através da criação de um curto circuito no interior do cabo da porta de série (consultar o anexo XVI).

Todos estes testes foram bem sucedidos aquando da sua realização numa porta de série industrial. Infelizmente, não tivemos a oportunidade de testar no ambiente da FEUP devido à situação atual de pandemia.

8 Eficiência do Protocolo de Ligação de Dados

Todos os resultados obtidos ao longo deste capítulo foram conseguidos utilizando uma porta de série diferente das disponíveis nos laboratórios da FEUP.

8.1 Variação do Tamanho das Tramas

Usando uma imagem com o tamanho de 21936 bytes e um valor constante de 38400 para a *baudrate*, obtivemos os resultados visíveis no Anexo XVII.

Após uma análise destes, consideramos que a eficiência aumenta com o aumento do tamanho da trama de informação.

8.2 Variação da Capacidade de Ligação *Baudrate*

Usando uma imagem com o tamanho de 21936 bytes e 512 como o tamanho das tramas de informação, obtivemos os resultados visíveis no Anexo XVIII.

Tendo por base os resultados obtidos, pode-se concluir que, quando maior a capacidade de ligação, menor será a eficiência, apesar da transmissão terminar em menos espaço de tempo.

8.3 Geração do Atraso de Propagação

Usando uma imagem com o tamanho 21936 bytes, um valor constante de 38400 para a *baudrate* e 512 como o tamanho das tramas de informação, obtivemos os resultados presentes no Anexo XIX.

Analisando os dados obtidos ao longo do teste, verifica-se que quando maior o atraso que se introduz, menor será a eficiência do programa, tal como esperado. Como se verifica, o protocolo utilizado é bastante prejudicado pelo atraso, uma vez que os dados demoram mais tempo a ser transmitidos de um dispositivo para outro, assim como as tramas de confirmação e rejeição.

8.4 Geração Aleatória de Erros

Usando uma imagem com o tamanho 21936 bytes, um valor constante de 38400 para a *baudrate* e 512 como o tamanho das tramas de informação, obtivemos os resultados presentes no Anexo XX.

Após a análise dos resultados obtidos, verifica-se que a eficiência do programa criado diminui significativamente com o aumento da percentagem de erros gerados, tal como esperado. No entando, apesar de serem criados erros, o ficheiro é transmitido com sucesso.

9 Conclusões

Ao longo do período de trabalho dedicado a este projeto, conseguimos implementar todas as funcionalidades que nos foram propostas, o que foi bastante benéfico para a nossa compreensão da matéria lecionada ao longo das aulas. O protocolo implementado consegue passar vários ficheiros com dimensões diferentes, inclusive aquando da simulação de erros da porta de série, levando-nos a crer que o trabalho foi bem conseguido.

De uma forma geral, o nosso grupo acredita que o projeto apresentou uma complexidade elevada. No entanto, conseguimos atingir os objetivos que nos foram colocados com sucesso. Uma das maiores dificuldades encontradas foi simular nos nosso computadores os erros que acabariam por surgir no laboratório, já que numa situação de ensino remoto não nos era permitido ter acesso físico simultâneo à porta de série. Porém, estas foram ultrapassadas, através da utilização de uma porta de série industrial, levando ao sucesso do nosso projeto.

10 Anexos

10.1 Anexo I - Código Fonte

```
#ifndef CONSTRAINTS_H
#define CONSTRAINTS_H

// ---------------------------------------------------------------
// ---------------       Main Related       ---------------
// ---------------------------------------------------------------

#define SERIALPORT              "/dev/ttyS0"
#define MODEMDEVICE             "/dev/ttyS1"
#define COM1                    1
#define COM2                    2
#define _POSIX_SOURCE           1         // POSIX compliant source
#define FALSE                   0
#define TRUE                    1

// ---------------------------------------------------------------
// ---------------     Application Layer    ---------------
// ---------------------------------------------------------------

// ----- Packets : Formatting and Types -----
#define SIZE_PER_BYTE           256       // Max Size Per Byte

#define C_DATA_PACKET           1
#define C_START_PACKET          2
#define C_END_PACKET            3

// ---- Data Packet ----
//  C  |  N  |  L2  |  L1  |  P
#define NON_INFO_BYTES_PER_PACKET 4       // Bytes in Data Packet not used for Information
#define MAX_DATA_SIZE           (SIZE_PER_BYTE-1) * SIZE_PER_BYTE + (SIZE_PER_BYTE-1) //
    Max Information Size Per Packet
#define DATA_SIZE               4096      // Bytes of Information Per Packet

struct data_packet {
  unsigned char data[DATA_SIZE];
  int size;
};

// ---- Control Packet ----
//  C  |  T1  |  L1  |  V1  |  T2  |  L2  |  V2
#define T_LENGTH                0         // Set File Length
#define T_FILENAME              1         // Set File Name
#define MAX_FILENAME_SIZE       SIZE_PER_BYTE

struct control_packet {
  char filename[MAX_FILENAME_SIZE];
  int filename_size;
  int file_size;
};

// ---------------------------------------------------------------
```

```
// ---------------     File Interaction    ---------------
// -------------------------------------------------------------
#define FILENAME                "pinguim.gif"
#define MAX_READ_CHARS          DATA_SIZE       // Max Chars Read From a File At a Time

// -------------------------------------------------------------
// ---------------  Logical Link Connection ---------------
// -------------------------------------------------------------

// ----- Connection Device -----
#define UNDEFINED              -1
#define TRANSMITTER             0
#define RECEIVER                1

// ----- Connection Parameters -----
#define BAUDRATE             B38400
#define TIME                    3        // Inter-character timer - TIME seconds
#define MIN                     0        // Blocking read until MIN chars read
#define MAX_RETRANSMISSIONS     3        // Number of Alarm Interruptions
#define PROPAGATION_DELAY       2        // Propagation Time

// ----- Frames : Formatting and Types -----
#define MAX_INFORMATION_SIZE    DATA_SIZE + NON_INFO_BYTES_PER_PACKET    // Max Bytes
    Transfered Per Frame
#define MAX_INFORMATION_WRITE_SIZE 2 * MAX_INFORMATION_SIZE

struct packet {
  unsigned char packet_bytes[MAX_INFORMATION_WRITE_SIZE];
  int size;
};

#define FLAG                    0x7E     // Framing Delimiter

// ----- A Byte -----
#define A_CMD_SND               0x03     // Command sent by Transmitter
#define A_CMD_RCV               0x01     // Command sent by Receiver
#define A_ANS_SND               0x01     // Answer sent by Transmitter
#define A_ANS_RCV               0x03     // Answer sent by Receiver

// ----- C Byte -----
#define C_SET                   0x03     // SET (set up)
#define C_DISC                  0X0B     // DISC (disconnect)
#define C_UA                    0x07     // UA (unnumbered acknowledgment)
#define C_RR0                   0x05     // RR (receiver ready/positive ACK) with next
    sequence number 0
#define C_RR1                   0x85     // RR (receiver ready/positive ACK) with next
    sequence number 1
#define C_REJ0                  0x01     // REJ (reject/negative ACK) with next sequence
    number 0
#define C_REJ1                  0x81     // REJ (reject/negative ACK) with next sequence
    number 1
#define C_SEQUENCE_0            0x00     // Information Frame with sequence number 0
#define C_SEQUENCE_1            0x40     // Information Frame with sequence number 1
#define C_ERROR                 0xFF     // Error

// ----- Information Frame -----
```

```
// FLAG | A | C | BCC1 | D | BCC2 | FLAG

// ----- Index -----
#define I_START_FLAG_INDEX      0
#define I_A_INDEX               1
#define I_C_INDEX               2
#define I_BCC1_INDEX            3
#define I_INFORMATION_INDEX     4
#define I_BCC2_INDEX            5
#define I_FINISH_FLAG_INDEX     6
#define I_END_INDEX             7

// ----- Byte Stuffing -----
#define ESCAPE                  0x7D     // Escape Character
#define ESCAPE_XOR              0x20     // Used in a XOR to obtain the character that
    follows an escape character
#define ESCAPE_ESCAPE           ESCAPE_XOR ^ ESCAPE
#define ESCAPE_FLAG             ESCAPE_XOR ^ FLAG

// ----- Supervision Frame -----
// FLAG | A | C | BCC1 | FLAG

// ----- Index -----
#define S_START_FLAG_INDEX      0
#define S_A_INDEX               1
#define S_C_INDEX               2
#define S_BCC1_INDEX            3
#define S_FINISH_FLAG_INDEX     4
#define S_END_INDEX             5

#endif // CONSTRAINTS_H
```

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

#include "constraints.h"
#include "logical_link.c"
#include "application_layer.c"
#include "file_interaction.c"

// ---------------------------------------------------------------
// ---------------     GLOBAL VARIABLES     ---------------
// ---------------------------------------------------------------

extern int finished;

// ---------------------------------------------------------------
// ---------------          MAIN            ---------------
// ---------------------------------------------------------------

void description(int filesize, double time) {
  printf("-------------------------------------\n");
  printf(" * Filename: %s\n", FILENAME);
  printf(" * FileSize: %d\n", filesize);
  printf(" * Baudrate: %d\n", BAUDRATE);
  printf(" * Retransmission attempts: %d\n", MAX_RETRANSMISSIONS);
  printf(" * Propagation Delay: %d\n", PROPAGATION_DELAY);
  printf(" * Timeout time: %d\n", TIME);
  printf(" * Chars read per time: %d\n", MIN);
  printf(" * Data Per Packet: %d\n", DATA_SIZE);
  printf("-------------------------------------\n");
  printf(" * Time: %f\n", time);
  printf("-------------------------------------\n");
}

int main(int argc, char** argv) {
  if ((argc < 2) || ((strcmp(SERIALPORT, argv[1])!=0) && (strcmp(MODEMDEVICE,
      argv[1])!=0))) {
    printf("Usage:\tnserial SerialPort\n\tex: nserial %s\n", MODEMDEVICE);
    return 1;
  }

  if (DATA_SIZE > MAX_DATA_SIZE) {
    printf("Invalid number of bytes per packet.\n");
    return 1;
  }

  // Time
  time_t start_time = time(NULL);

  char * filename = FILENAME;
  if (verify_file(filename) != 0) {
    printf("Couldn't find file.\n");
    return 1;
```

```
}

//Open Logical Link Connection
int com = (strcmp(SERIALPORT, argv[1]) == 0) ? COM1 : COM2;
int fd = llopen(com, TRANSMITTER);
if (fd < 0) {
  printf("Error when stablishing connection.\n");
  return 1;
}

struct packet p;

// Send Starting Control Packet
struct control_packet c;
c.file_size = get_size_from_file(filename);
int filesize = c.file_size;
if (c.file_size == 0) {
  printf("Error when obtaining the file size.\n");
  llclose(fd);
  return 1;
}
c.filename_size = sizeof(FILENAME);
strcpy(c.filename, filename);

int error = FALSE;

p = construct_control_packet(c);
if (p.size <= 0) {
  printf("Error when constructing the control packet.\n");
  error = TRUE;
}

if (llwrite(fd, p.packet_bytes, p.size) < 0) {
  printf("Error when sending the starting control packet.\n");
  error = TRUE;
}

// Send Data Through Logical Link
struct data_packet d; d.size = 0;

while (finished == FALSE) {
  d = read_from_file(filename);
  if (d.size > 0) {
    p = construct_data_packet(d);
    if (p.size <= 0) {
      printf("Error when constructing the data packet.\n");
      error = TRUE;
    }

    // Simulate Delay
    sleep(PROPAGATION_DELAY);
    if (llwrite(fd, p.packet_bytes, p.size) < 0) {
      printf("Error when sending message.\n");
      error = TRUE;
    }
  }
```

```
  }

  // Send Ending Control Packet
  p = construct_control_packet(c);
  if (p.size <= 0) {
    printf("Error when constructing the control packet.\n");
    error = TRUE;
  }
  if (llwrite(fd, p.packet_bytes, p.size) < 0) {
    printf("Error when sending the ending control packet.\n");
    error = TRUE;
  }

  // End of Logical Link Connection
  if (llclose(fd) != 0) {
    printf("Error when closing the connection.\n");
    error = TRUE;
  }

  // Time
  time_t stop_time = time(NULL);
  double time = difftime(stop_time, start_time);
  description(filesize, time);

  if (error) { return 1; }
  return 0;
}
```

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

#include "constraints.h"
#include "logical_link.c"
#include "application_layer.c"
#include "file_interaction.c"

// ---------------------------------------------------------------
// ---------------     GLOBAL VARIABLES     ---------------
// ---------------------------------------------------------------

extern int disconnect;

// ---------------------------------------------------------------
// ---------------          MAIN            ---------------
// ---------------------------------------------------------------

int main(int argc, char** argv) {
  if ((argc < 2) || ((strcmp(SERIALPORT, argv[1])!=0) && (strcmp(MODEMDEVICE,
      argv[1])!=0))) {
    printf("Usage:\tnserial SerialPort\n\tex: nserial %s\n", SERIALPORT);
    return 1;
  }

  //Open Logical Link Connection
  int com = (strcmp(SERIALPORT, argv[1]) == 0) ? COM1 : COM2;
  int fd = llopen(com, RECEIVER);
  if (fd < 0) {
    printf("Error when stablishing connection.\n");
    return 1;
  }

  // Receive Data from Logical Link
  struct packet p;
  struct data_packet d;
  struct control_packet c;
  strcpy(c.filename, FILENAME);

  while (disconnect == FALSE) {
    p.size = llread(fd, p.packet_bytes);
    if (p.size != -1) {
      if (is_data_packet(p)) {
        d = deconstruct_data_packet(p);
        if (d.size == 0) { printf("Error when obtaining information from logical link.\n");
            }
        if (write_to_file(c.filename, d) == -1) { printf("Error when writing to file.\n"); }
      }
      else {
        c = deconstruct_control_packet(p);
        if (c.file_size == 0) { printf("Error when obtaining control information from
            logical link.\n"); }
        if (c.filename_size == 0) { strcpy(c.filename, FILENAME); }
      }
```

```
    }
  }

  // Verify Data
  if (verify() != 0) {
    printf("Error in data.\n");
    llclose(fd);
    return 1;
  }

  // End of Logical Link Connection
  if (llclose(fd) != 0) {
    printf("Error when closing the connection.\n");
    return 1;
  }
  return 0;
}
```

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

#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>

#include "constraints.h"

// ---------------------------------------------------------------
// ---------------     GLOBAL VARIABLES     ---------------
// ---------------------------------------------------------------

int last_sequence_number = -1;
static struct termios oldtio;
int side = UNDEFINED;

int time_limit = FALSE; // TRANSMITTER

int disconnect = FALSE; // RECEIVER

// ---------------------------------------------------------------
// ---------------     HELPER FUNCTIONS     ---------------
// ---------------------------------------------------------------

/**
 * Send a Message (C)
 *
 * Return Values :
 *       success  -> TRUE
 *       unsuccess -> FALSE
**/
int send(int fd, unsigned char a, unsigned char c) {
  unsigned char message[5];
  message[0] = FLAG;
  message[1] = a;
  message[2] = c;
  message[3] = a ^ c;
  message[4] = FLAG;

  int res = write(fd, message, 5);

  if (side == TRANSMITTER) { printf("Emitter"); }
  else                     { printf("Receiver"); }
  printf(" write %d bytes, %x %x %x %x %x\n", res, message[0], message[1], message[2],
      message[3], message[4]);

  return (res == 5);
}

/**
 * Attempts to retrieve C Byte from a frame.
 *
 * Return Values:
 *       success  -> C
```

```
 *       unsuccess -> error (0xFF)
**/
unsigned char retrieve(int fd, unsigned char a) {
  const int MAX_CHARS_PER_READ = 1;
  unsigned char c = C_ERROR;
  unsigned char frame[6] = {0, 0, 0, 0, 0, 0}; //TODO : maybe remove
  unsigned char buf[MAX_CHARS_PER_READ];

  int index = 0;
  int res;

  int timeout = FALSE;
  time_t now, start = time(NULL);
  double t;

  int stop = FALSE;
  while (stop == FALSE) {
    res = read(fd, buf, MAX_CHARS_PER_READ);
    if (res == 1) {
      switch(index) {
        case S_START_FLAG_INDEX:
          if (buf[0] == FLAG) { frame[index] = buf[0]; ++ index; }
          break;

        case S_A_INDEX:
          if (buf[0] == a) { frame[index] = buf[0]; ++ index; }
          else if (buf[0] != FLAG) { index = S_START_FLAG_INDEX; }
          break;

        case S_C_INDEX:
          if (buf[0] == FLAG) { index = S_START_FLAG_INDEX; }
          else                { frame[index] = buf[0]; c = buf[0]; ++ index; }
          break;

        case S_BCC1_INDEX:
          if (buf[0] == (a ^ c)) { frame[index] = buf[0]; ++ index; }
          else { return C_ERROR; }
          break;

        case S_FINISH_FLAG_INDEX:
          frame[index] = buf[0];
          index ++;
          break;

        default:
          break;
      }
    }
    if (time_limit && (TIME != 0)) {
      now = time(NULL);
      t = difftime(now, start);
      timeout = (t > TIME * 1.1);
    }
    if (timeout && index == S_START_FLAG_INDEX) stop = TRUE;
    if (index == S_END_INDEX) stop = TRUE;
  }
```

```
  if (side == TRANSMITTER) { printf("Emitter"); }
  else                     { printf("Receiver"); }
  printf(" read %d bytes, %x %x %x %x %x\n", index, frame[0], frame[1], frame[2], frame[3],
      frame[4]);

  return timeout ? C_ERROR : c;
}

/**
 * Send a Command Message (C)
 *
 * Return Values :
 *       success  -> TRUE
 *       unsuccess -> FALSE
**/
int send_command(int fd, unsigned char c) {
  unsigned char a;
  switch (side) {
    case TRANSMITTER:
      a = A_CMD_SND;
      break;
    case RECEIVER:
      a = A_CMD_RCV;
      break;
    default:
      // ERROR
      return -1;
  }
  return send(fd, a, c);
}

/**
 * Send a Confirmation Response (C)
 *
 * Return Values :
 *       success  -> TRUE
 *       unsuccess -> FALSE
**/
int send_confirmation(int fd, unsigned char c) {
  unsigned char a;
  switch (side) {
    case TRANSMITTER:
      a = A_ANS_SND;
      break;
    case RECEIVER:
      a = A_ANS_RCV;
      break;
    default:
      // ERROR
      return -1;
  }
  return send(fd, a, c);
}

/**
```

```
 * Attempts to retrieve C Command.
 *
 * Return Values:
 *       success  -> C command
 *       unsuccess -> error (0xFF)
**/
unsigned char retrieve_command(int fd) {
  unsigned char a;
  switch (side) {
    case TRANSMITTER:
      a = A_CMD_RCV;
      break;
    case RECEIVER:
      a = A_CMD_SND;
      break;
    default:
      // ERROR
      return C_ERROR;
  }
  return retrieve(fd, a);
}

/**
 * Attempts to retrieve C Response.
 *
 * Return Values:
 *       success  -> C response
 *       unsuccess -> error (0xFF)
**/
unsigned char retrieve_confirmation(int fd) {
  unsigned char a;
  switch (side) {
    case TRANSMITTER:
      a = A_ANS_RCV;
      break;
    case RECEIVER:
      a = A_ANS_SND;
      break;
    default:
      // ERROR
      return C_ERROR;
  }
  return retrieve(fd, a);
}

// ------------------------------------------------------------
// ---------------  Logical Link Connection ---------------
// ------------------------------------------------------------

/**
 * Establishment of the Logical Link
 *
 * TRANSMITTER:
 * Sends the SET message.
 * Attempts to receive UA message.
 *
```

```
 * RECEIVER:
 * Receives the SET message.
 * When successful, sends the UA message.
 *
 * Return Values:
 *       success  -> fd
 *       unsuccess -> -1
**/
int llopen(int com, int machine_side) {
  /*
    Open serial port device for reading and writing and not as controlling tty
    because we don't want to get killed if linenoise sends CTRL-C.
  */
  char * arg = (com == COM1) ? SERIALPORT: MODEMDEVICE;
  static struct termios newtio;
  int fd = open(arg, O_RDWR | O_NOCTTY );
  if (fd < 0) { perror(arg); return -1; }

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

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

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

  newtio.c_cc[VTIME]  = TIME;  /* inter-character timer */
  newtio.c_cc[VMIN]   = MIN;   /* blocking read until MIN chars received */

  /*
    VTIME e VMIN should be altered in order to protect
    the reading of the next chars with a timer
  */

  tcflush(fd, TCIOFLUSH);

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

  printf("New termios structure set\n");

  side = machine_side;
  if (side == TRANSMITTER) {
    int retransmission_count = 1;
    time_limit = TRUE;
    while (retransmission_count <= MAX_RETRANSMISSIONS) {
      send_command(fd, C_SET);
      if (retrieve_confirmation(fd) == C_UA) { break; }
```

```
      ++ retransmission_count;
    }
    time_limit = FALSE;
    if (retransmission_count > MAX_RETRANSMISSIONS) {
      printf("UA not Received!\n");
      return -1;
    }
    printf("UA Received.\n");
    return fd;
  }
  else { // side == RECEIVER
    unsigned char set = retrieve_command(fd);
    if (set == C_SET) { send_confirmation(fd, C_UA); return fd; }
  }
  return -1;
}

/**
 * Transmitting of Information Through the Logical Link
 *
 * Reads a Message.
 * Detects if the Emitter wants to Disconnect.
 * Send a Adequate Response.
 *
 * Return Values:
 *       size of the message received
 *       -1 if no message is received
**/
int llread(int fd, unsigned char * buffer) {
  const int MAX_CHARS_PER_READ = 1;
  int size = MAX_INFORMATION_SIZE;
  bzero(buffer, size);

  unsigned char buf[MAX_CHARS_PER_READ];
  int res;
  int loop_index = 0, index = -1;
  int sequence_number = -1;
  unsigned char lastChar = 0x00;
  int escapedChar = FALSE;
  int bcc1error = FALSE, bcc2error = FALSE;
  unsigned char bcc2 = 0x00;

  int stop = FALSE;
  while (stop == FALSE) {
    res = read(fd, buf, MAX_CHARS_PER_READ);
    if (res == 1) {
      switch (loop_index) {
        case I_START_FLAG_INDEX:
          if (buf[0] == FLAG)           { ++ loop_index; }
          break;

        case I_A_INDEX:
          if      (buf[0] == A_CMD_SND) { ++ loop_index; }
          else if (buf[0] != FLAG)      { loop_index = I_START_FLAG_INDEX; }
          break;
```

```
      case I_C_INDEX:
        if      (buf[0] == C_SEQUENCE_0) { sequence_number = 0; ++ loop_index; }
        else if (buf[0] == C_SEQUENCE_1) { sequence_number = 1; ++ loop_index; }
        else if (buf[0] == FLAG)      { loop_index = I_A_INDEX; }
        else if (buf[0] == C_DISC)    { disconnect = TRUE; ++ loop_index; }
        else                          { loop_index = I_START_FLAG_INDEX; }
        break;

      case I_BCC1_INDEX:
        if      ((sequence_number == 0) && (buf[0] == (A_CMD_SND ^ C_SEQUENCE_0))) { ++
            loop_index; }
        else if ((sequence_number == 1) && (buf[0] == (A_CMD_SND ^ C_SEQUENCE_1))) { ++
            loop_index; }
        else if ((    disconnect    ) && (buf[0] == (A_CMD_SND ^ C_DISC)))   { ++
            loop_index; }
        else { bcc1error = TRUE; /* BCC1 error */ }
        break;

      case I_INFORMATION_INDEX:
        if (buf[0] == FLAG) {
          stop = TRUE;
          if (disconnect) { break; }
          // BCC2 is the lastChar
          if (index > size) { bcc2error = TRUE; break; /* BCC2 error */ }
          for (int i = 0; i < index; ++ i) { bcc2 = bcc2 ^ buffer[i]; }
          if (bcc2 != lastChar) { bcc2error = TRUE; /* BCC2 error */ }
        }
        else if ((lastChar == ESCAPE) && (!escapedChar)) { lastChar = buf[0] ^ ESCAPE_XOR;
            escapedChar = TRUE;}
        else {
          if ((index >= 0) && (index < size)) { buffer[index] = lastChar; }
          ++ index;
          lastChar = buf[0];
          escapedChar = FALSE;
        }
        break;
      default:
        break;
    }
  }
}

switch (sequence_number) {
  case 0:
    if (bcc1error) { printf("Error with BCC1\n"); return -1; }
    else if (bcc2error) {
      if (last_sequence_number == sequence_number) { send_confirmation(fd, C_RR1 ); } /*
          Duplicated Data */
      else                                    { send_confirmation(fd, C_REJ1); } /* New
          Data */
      printf("Error with BCC2 %x %x\n", bcc2, lastChar); return -1;
    }
    else { send_confirmation(fd, C_RR1); last_sequence_number = sequence_number; return
        index; }
    break;
  case 1:
```

```
      if (bcc1error) { printf("Error with BCC1\n"); return -1; }
      else if (bcc2error) {
        if (last_sequence_number == sequence_number) { send_confirmation(fd, C_RR0 ); } /*
            Duplicated Data */
        else                                    { send_confirmation(fd, C_REJ0); } /* New
            Data */
        printf("Error with BCC2\n"); return -1;
      }
      else { send_confirmation(fd, C_RR0); last_sequence_number = sequence_number; return
          index; }
      break;
    default:
      // Means it received DISC and is now preparing to disconnect
      break;
  }

  return index;
}

/**
 * Transmitting of Information Through the Logical Link
 *
 * Sends a message.
 * Receives the UA message.
 *
 * Return Values:
 *       success  -> 0
 *       unsuccess -> -1
**/
int llwrite(int fd, unsigned char * buf, int size) {
  // Message : F | A | C | Bcc1 | D | Bcc2 | F

  if (size > MAX_INFORMATION_SIZE) {
    printf("Too many chars read.\n");
    return -1;
  }

  int sequence_number = ((last_sequence_number == 0) ? 1 : 0);

  int MAX_CHARS = MAX_INFORMATION_WRITE_SIZE + I_FINISH_FLAG_INDEX;
  int i = 0, j = 0;

  unsigned char msg[MAX_CHARS]; // Frame to be sent

  msg[i++] = FLAG;                                                    // Flag
  msg[i++] = A_CMD_SND;                                               // A
  msg[i++] = (sequence_number == 0) ? C_SEQUENCE_0 : C_SEQUENCE_1;    // C
  msg[i++] = A_CMD_SND ^ ((sequence_number == 0) ? C_SEQUENCE_0 : C_SEQUENCE_1); // BCC1

  unsigned char info_frame[MAX_INFORMATION_WRITE_SIZE];
  unsigned char bcc2 = 0x00;

  /*
  *      Byte Stuffing and buffer size adjusting - Information
  *
  * If inside the information occurs the FLAG pattern,
```

```
*   it should be replaced by the sequence ESCAPE ESCAPE_FLAG.
*
* If inside the information occurs the ESCAPE pattern,
*   it should be replaced by the sequence ESCAPE ESCAPE_ESCAPE.
*/
for (int k = 0; k < size; ++ k) {
  if (buf[k] == FLAG) {
    info_frame[j++] = ESCAPE;
    info_frame[j++] = ESCAPE_FLAG;
  }
  else if (buf[k] == ESCAPE) {
    info_frame[j++] = ESCAPE;
    info_frame[j++] = ESCAPE_ESCAPE;
  }
  else {
    info_frame[j++] = buf[k];
  }

  bcc2 = buf[k] ^ bcc2;
}

// D
int p = 0;
for (p = 0; p < j; ++ p) {
  msg[i+p] = info_frame[p];
}
i = i + p;

// Byte Stuffing - BCC2
if (bcc2 == FLAG) {
  msg[i++] = ESCAPE;
  msg[i++] = ESCAPE_FLAG;
}
else if (bcc2 == ESCAPE) {
  msg[i++] = ESCAPE;
  msg[i++] = ESCAPE_ESCAPE;
}
else {
  msg[i++] = bcc2;
}

msg[i++] = FLAG;              // Flag

unsigned char confirmation;
int confirmed = FALSE;
int retransmission_count = 1;
while ((retransmission_count <= MAX_RETRANSMISSIONS) && (confirmed == FALSE)) {
  write(fd, msg, i);
  ++ retransmission_count;

  time_limit = TRUE;
  confirmation = retrieve_confirmation(fd);
  time_limit = FALSE;

  switch (confirmation) {
    case C_RR0:
```

```
        if (sequence_number == 1) { confirmed = TRUE; }
        break;
      case C_RR1:
        if (sequence_number == 0) { confirmed = TRUE; }
        break;
      default:
        /*
          Goes for C_REJ0, C_REJ1 and error
          Also C_RR0 or C_RR1 out of sequence
        */
        break;
    }
  }

  if (confirmed == FALSE) {
    printf("Confirmation not Received!\n");
    return -1;
  }

  last_sequence_number = sequence_number;
  printf("Confirmation Received.\n");
  return 0;
}

/**
 * Closing of Logical Link
 *
 * TRANSMITTER:
 * Sends command DISC.
 * Receives DISC command.
 * Sends UA message.
 *
 * RECEIVER:
 * Sends command DISC.
 * Receives UA message.
 *
 * Return Values:
 *       success  -> 0
 *       unsuccess -> -1
**/
int llclose(int fd) {
  if (side == TRANSMITTER) {
    send_command(fd, C_DISC);
    time_limit = TRUE;
    unsigned char disc = retrieve_command(fd);
    time_limit = FALSE;
    if (disc != C_DISC) { printf("Error on DISC.\n"); return -1; }
    send_confirmation(fd, C_UA);
  }
  else { // side == RECEIVER
    send_command(fd, C_DISC);
    unsigned char ua = retrieve_confirmation(fd);
    if (ua != C_UA) { printf("Error on UA.\n"); }
   if (tcsetattr(fd,TCSANOW,&oldtio) == -1) {
      printf("Error on tcsetattr\n");
      perror("tcsetattr");
```

```
      return -1;
    }
  }


  close(fd);
  return 0;
}
```

```c
#include <sys/stat.h>
#include <fcntl.h>

#include <stdio.h>

#include "constraints.h"

// ------------------------------------------------------------
// ---------------     GLOBAL VARIABLES     ---------------
// ------------------------------------------------------------

FILE * aux_f = NULL; // TRANSMITTER
int finished = FALSE; // TRANSMITTER

// ------------------------------------------------------------
// ---------------      FILE RELATED       ---------------
// ------------------------------------------------------------

/**
 * Read from the file
 *
 * Reading information from the file
 * to a data struct
 *
 * Return Values :
 *       success  -> d.size >  0
 *       unsuccess -> d.size == 0
 *       finished -> d.size == -1
**/
struct data_packet read_from_file(char * filename) {
  struct data_packet d;
  d.size = 0;

  if (finished) {
    d.size = -1;
    return d;
  }

  // Open the file in read only mode
  FILE *f;
  if (aux_f == NULL) {
    f = fopen(filename, "rb");
    if (f == NULL) { printf("Couldn't read from file.\n"); finished = TRUE; return d; }
  }
  else f = aux_f;

  // Read from file to a data_packet
  d.size = fread(d.data, 1, MAX_READ_CHARS, f);
  if (ferror(f)) { printf("Error when reading from file.\n"); return d; }

  if (d.size < MAX_READ_CHARS) {
    aux_f = NULL;
    finished = TRUE;
    if (fclose(f) != 0) { printf("Error when closing the file.\n"); }
  }
```

```
  else aux_f = f;

  return d;
}

/**
 * Write to the file
 *
 * Takes the information from the data packet
 * Appends that information in the file
 *
 * Return Values :
 *       success  -> 0
 *       unsuccess -> -1
**/
int write_to_file(char * filename, struct data_packet d) {
  // Open the file in append mode (Creating it if it doesn't exit)
  FILE * f = fopen(filename, "a");
  if (f == NULL) { printf("Couldnt't open file.\n"); }

  // Write the data to the file (appending it)
  int res = fwrite(d.data, 1, d.size, f);
  if (res != d.size) { printf("Error when writing to file.\n"); }

  // Close the file
  res = fclose(f);
  if (res != 0) { printf("Error when closing the file.\n"); }
  return 0;
}

/**
 * Get Size from a File
 *
 * Return Values :
 *       success  -> size from file
 *       unsuccess -> 0
**/
off_t get_size_from_file(char * filename){
  struct stat info;
  stat(filename, &info);
  return info.st_size;
}

/**
 * Verify if there is a file with name filename
 *
 * Return Values :
 *       success  -> 0
 *       unsuccess -> -1
**/
int verify_file(char * filename) {
  FILE * f = fopen(filename, "r");
  if (f == NULL) { return -1; }
  fclose(f);
  return 0;
}
```

```
#include <string.h>

#include "constraints.h"

// ------------------------------------------------------------
// ---------------     GLOBAL VARIABLES     ---------------
// ------------------------------------------------------------

int last_packet_number = -1;
int started = FALSE;
struct control_packet start_packet; // RECEIVER
int size = 0;                       // RECEIVER
int control_packet_count = 0;    // RECEIVER

// ------------------------------------------------------------
// ---------------    PACKET CONSTRUCTION   ---------------
// ------------------------------------------------------------

/**
 * Turns the received array into a data packet
 *
 * Return Values :
 *       success  -> data_packet with size > 0
 *       unsuccess -> data_packet with size <= 0
**/
struct data_packet deconstruct_data_packet(struct packet p) {
  // C  |  N  |  L2  |  L1  |  P
  struct data_packet d;
  bzero(d.data, DATA_SIZE);
  d.size = 0;
  int index = 0;

  if (p.packet_bytes[index ++] != C_DATA_PACKET) {
    printf("This packet is not a data packet.\n");
    d.size = -1;
    return d;
  }

  int packet_number = (last_packet_number + 1) % SIZE_PER_BYTE;
  if (packet_number != p.packet_bytes[index ++]) {
    printf("This data packet is out of sequence.\n");
    return d;
  }

  d.size = p.packet_bytes[index ++] * SIZE_PER_BYTE;
  d.size = d.size + p.packet_bytes[index ++];

  for (int i = 0; i < d.size; ++ i) {
    d.data[i] = p.packet_bytes[index ++];
  }

  last_packet_number = packet_number;
  size = size + d.size;
  return d;
}
```

```
/**
 * Turns the received array into a control packet
 *
 * Return Values :
 *       success  -> control_packet with file_size > 0
 *       unsuccess -> control_packet with file_size <= 0
**/
struct control_packet deconstruct_control_packet(struct packet p) {
  // C  |  T1  |  L1  |  V1  |  T2  |  L2  |  V2
  struct control_packet c;
  bzero(c.filename, MAX_FILENAME_SIZE);
  c.file_size = 0;
  c.filename_size = 0;
  int index = 0;

  ++ control_packet_count;
  if (control_packet_count > 2) {
    printf("Too much control packets.\n");
    return c;
  }

  unsigned char c_byte = p.packet_bytes[index ++];

  if (c_byte == C_DATA_PACKET) {
    printf("This packet is not a control packet.\n");
    c.file_size = -1;
    return c;
  }
  else if ((!started) && (c_byte != C_START_PACKET)) {
    printf("This control packet is out of order.\n");
    return c;
  }
  else if ((started) && (c_byte != C_END_PACKET)) {
    printf("This control packet is out of order.\n");
    return c;
  }

  unsigned char t;
  int l;
  while (index < p.size) {
    t = p.packet_bytes[index ++];
    l = p.packet_bytes[index ++];

    switch(t) {
      case T_FILENAME:
        c.filename_size = l;
        for (int i = 0; i < l; ++ i) {
          c.filename[i] = p.packet_bytes[index ++];
        }
        break;
      case T_LENGTH:
        c.file_size = 0;
        for (int i = 0; i < l; ++ i) {
          c.file_size = c.file_size * SIZE_PER_BYTE + p.packet_bytes[index ++];
        }
```

```
        break;
      default:
        printf("This type is not defined.\n");
        break;
    }
  }

  if (!started) { started = TRUE; start_packet = c; }
  else if ((started) && ((strcmp(start_packet.filename, c.filename) != 0) ||
      (start_packet.file_size != c.file_size))) {
    printf("Start Packet did not match Finish Packet.\n");
    c.file_size = 0;
  }
  else if ((started) && (size != start_packet.file_size)) {
    printf("Received size doesn't match control size.\n");
    c.file_size = -2;
  }
  return c;
}

/**
 * Turns the data packet into the outgoing array
 *
 * Return Values :
 *       success  -> packet with size > 0
 *       unsuccess -> packet with size == 0
**/
struct packet construct_data_packet(struct data_packet d) {
  // C  |  N  |  L2  |  L1  |  P
  int packet_number = (last_packet_number + 1) % SIZE_PER_BYTE;
  struct packet p;
  p.size = 0;

  if ((d.size == 0) || (d.size > DATA_SIZE)) {
    printf("Error with data packet size.\n");
    return p;
  }

  p.packet_bytes[p.size ++] = C_DATA_PACKET;
  p.packet_bytes[p.size ++] = packet_number;
  p.packet_bytes[p.size ++] = d.size / SIZE_PER_BYTE;
  p.packet_bytes[p.size ++] = d.size % SIZE_PER_BYTE;

  for (int index = 0; index < d.size; ++ index) {
    p.packet_bytes[p.size ++] = d.data[index];
  }

  last_packet_number = packet_number;
  return p;
}

/**
 * Turns the control packet into the outgoing array
 *
 * Return Values :
 *       success  -> packet with size > 0
```

```
 *       unsuccess -> packet with size == 0
**/
struct packet construct_control_packet(struct control_packet c) {
  // C  |  T1  |  L1  |  V1  |  T2  |  L2  |  V2

  struct packet p;
  p.size = 0;
  unsigned char array[SIZE_PER_BYTE];
  int size_aux = c.file_size;
  int m = SIZE_PER_BYTE, divisions = 0;

  if (!started) { p.packet_bytes[p.size ++] = C_START_PACKET; started = TRUE; }
  else          { p.packet_bytes[p.size ++] = C_END_PACKET; }
  p.packet_bytes[p.size ++] = T_LENGTH;

  while (size_aux > 0) {
    array[-- m] = size_aux % SIZE_PER_BYTE;
    size_aux    = size_aux / SIZE_PER_BYTE;
    ++ divisions;
  }

  p.packet_bytes[p.size ++] = divisions;

  for (int i = 0; i < divisions; ++ i) {
    p.packet_bytes[p.size ++] = array[m ++];
  }

  p.packet_bytes[p.size ++] = T_FILENAME;
  p.packet_bytes[p.size ++] = c.filename_size;
  for (int i = 0; i < c.filename_size; ++ i) {
    p.packet_bytes[p.size ++] = c.filename[i];
  }

  return p;
}

/**
 * Determines if the packet is a data packet or not
 *
 * Return Values :
 *       success  -> TRUE
 *       unsuccess -> FALSE
**/
int is_data_packet(struct packet p) {
  return (p.packet_bytes[0] == C_DATA_PACKET);
}

/**
 * Verifies data coherence
 *
 * Return Values :
 *       success  ->  0
 *       unsuccess -> -1
**/
int verify() {
  if (control_packet_count != 2) {
```

```
    printf("Wrong Number of Control Packets.\n");
    return -1;
  }
  if (size != start_packet.file_size) {
    printf("Wrong File Size.\n");
    return -1;
  }
  return 0;
}
```

10.2 Anexo II - Estruturas de Dados da Camada de Ligação de Dados

```
struct packet {
    unsigned char packet_bytes[MAX_INFORMATION_WRITE_SIZE];
    int size;
};
```

10.3 Anexo III - Principais Funções da Camada de Ligação de Dados

```
int llopen(int com, int machine_side);
int llread(int fd, unsigned char * buffer);
int llwrite(int fd, unsigned char * buf, int size);
int llclose(int fd);
```

10.4 Anexo IV - Funções Auxiliares da Camada de Ligação de Dados

```
int send(int fd, unsigned char a, unsigned char c);
unsigned char retrieve(int fd, unsigned char a);
int send_command(int fd, unsigned char c);
int send_confirmation(int fd, unsigned char c);
unsigned char retrieve_command(int fd);
unsigned char retrieve_confirmation(int fd);
```

- `send`: enviar uma mensagem
- `retrieve`: receber um byte de uma trama
- `send_command`: enviar um comando
- `send_confirmation`: enviar uma resposta/confirmação
- `retrieve_command`: receber um comando
- `retrieve_confirmation`: receber uma resposta/confirmação

10.5 Anexo V - Estruturas de Dados da Camada da Aplicação

```
struct control_packet {
    char filename[MAX_FILENAME_SIZE];
    int filename_size;
    int file_size;
};

struct data_packet {
    unsigned char data[MAX_DATA_SIZE];
    int size;
};
```

10.6 Anexo VI - Principais Funções da Camada da Aplicação

Interação com Ficheiros

```
struct data_packet read_from_file(char * filename);
int write_to_file(char * filename, struct data_packet d);
```

Camada da Aplicação

```
struct data_packet deconstruct_data_packet(struct packet p);
struct control_packet deconstruct_control_packet(struct packet p);
struct packet construct_data_packet(struct data_packet d);
struct packet construct_control_packet(struct control_packet c);
```

10.7 Anexo VII - Funções Auxiliares da Camada da Aplicação

Interação com Ficheiros

```
off_t get_size_from_file(char * filename);
int verify_file(char * filename);
```

Camada da Aplicação

```
int is_data_packet(struct packet p);
int verify();
```

- `get_size_from_file`: obter tamanho do ficheiro
- `verify_file`: verifica se existe o ficheiro com o nome que foi passado como argumento
- `is_data_packet`: verifica se o pacote é um pacote de informação
- `verify`: verificações necessárias para saber se não houve nenhum erro

10.8 Anexo VIII - Estabelecimento/fecho da ligação lógica

Estabelecimento da Ligação

Na função `llopen`:

```
side = machine_side;
if (side == TRANSMITTER) {
  int retransmission_count = 1;
  time_limit = TRUE;
  while (retransmission_count <= MAX_RETRANSMISSIONS) {
    send_command(fd, C_SET);
    if (retrieve_confirmation(fd) == C_UA) { break; }
    ++ retransmission_count;
  }
  time_limit = FALSE;
  if (retransmission_count > MAX_RETRANSMISSIONS) {
```

```
    printf("UA not Received!\n");
    return -1;
  }
  printf("UA Received.\n");
  return fd;
}
else { // side == RECEIVER
  unsigned char set = retrieve_command(fd);
  if (set == C_SET) { send_confirmation(fd, C_UA); return fd; }
}
```

Fecho da Ligação

Na função `llclose`:

```
if (side == TRANSMITTER) {
    send_command(fd, C_DISC);
    time_limit = TRUE;
    unsigned char disc = retrieve_command(fd);
    time_limit = FALSE;
    if (disc != C_DISC) { printf("Error on DISC.\n"); return -1; }
    send_confirmation(fd, C_UA);
}
else { // side == RECEIVER
    send_command(fd, C_DISC);
    unsigned char ua = retrieve_confirmation(fd);
    if (ua != C_UA) { printf("Error on UA.\n"); }
    if (tcsetattr(fd,TCSANOW,&oldtio) == -1) {
        printf("Error on tcsetattr\n");
        perror("tcsetattr");
        return -1;
    }
}
```

10.9 Anexo IX - Enviar/receber informação através da porta de série

Envio de Informação

```
int llwrite(int fd, unsigned char * buf, int size);
```

Receção de Informação

```
int llread(int fd, unsigned char * buffer);
```

10.10 Anexo X - Controlo de erros (BCC's e números de sequência)

Na função `llread`:

```
switch (sequence_number) {
    case 0:
      if (bcc1error) { printf("Error with BCC1\n"); return -1; }
      else if (bcc2error) {
        if (last_sequence_number == sequence_number) { send_confirmation(fd, C_RR1 ); }
            /* Duplicated Data */
        else                                        { send_confirmation(fd, C_REJ1); } /*
            New Data */
        printf("Error with BCC2 %x %x\n", bcc2, lastChar); return -1;
      }
      else { send_confirmation(fd, C_RR1); last_sequence_number = sequence_number;
          return index; }
      break;
    case 1:
      if (bcc1error) { printf("Error with BCC1\n"); return -1; }
      else if (bcc2error) {
        if (last_sequence_number == sequence_number) { send_confirmation(fd, C_RR0 ); }
            /* Duplicated Data */
        else                                        { send_confirmation(fd, C_REJ0); } /*
            New Data */
        printf("Error with BCC2\n"); return -1;
      }
      else { send_confirmation(fd, C_RR0); last_sequence_number = sequence_number;
          return index; }
      break;
    default:
      // Means it received DISC and is now preparing to disconnect
      break;
}
```

10.11 Anexo XI - Confirmações

Na função `llwrite`:

```
unsigned char confirmation;
int confirmed = FALSE;
int retransmission_count = 1;
while ((retransmission_count <= MAX_RETRANSMISSIONS) && (confirmed == FALSE)) {
  write(fd, msg, i);
  ++ retransmission_count;

  time_limit = TRUE;
  confirmation = retrieve_confirmation(fd);
  time_limit = FALSE;

  switch (confirmation) {
    case C_RR0:
      if (sequence_number == 1) { confirmed = TRUE; }
      break;
    case C_RR1:
      if (sequence_number == 0) { confirmed = TRUE; }
      break;
    default:
```

```
        /*
          Goes for C_REJ0, C_REJ1 and error
          Also C_RR0 or C_RR1 out of sequence
        */
        break;
    }
  }

  if (confirmed == FALSE) {
    printf("Confirmation not Received!\n");
    return -1;
  }

  last_sequence_number = sequence_number;
  printf("Confirmation Received.\n");
```

10.12 Anexo XII - Fragmentação/desfragmentação de pacotes

Descontrução de pacotes de dados

```
struct data_packet d;
bzero(d.data, MAX_DATA_SIZE);
d.size = 0;
int index = 0;

if (p.packet_bytes[index ++] != C_DATA_PACKET) {
  printf("This packet is not a data packet.\n");
  d.size = -1;
  return d;
}

int packet_number = (last_packet_number + 1) % SIZE_PER_BYTE;
if (packet_number != p.packet_bytes[index ++]) {
  printf("This data packet is out of sequence.\n");
  return d;
}

d.size = p.packet_bytes[index ++] * SIZE_PER_BYTE;
d.size = d.size + p.packet_bytes[index ++];

for (int i = 0; i < d.size; ++ i) {
  d.data[i] = p.packet_bytes[index ++];
}

last_packet_number = packet_number;
size = size + d.size;
```

Descontrução de pacotes de controlo

```
struct control_packet c;
bzero(c.filename, MAX_FILENAME_SIZE);
c.file_size = 0;
c.filename_size = 0;
int index = 0;
```

```
++ control_packet_count;
if (control_packet_count > 2) {
  printf("Too much control packets.\n");
  return c;
}

unsigned char c_byte = p.packet_bytes[index ++];

if (c_byte == C_DATA_PACKET) {
  printf("This packet is not a control packet.\n");
  c.file_size = -1;
  return c;
}
else if ((!started) && (c_byte != C_START_PACKET)) {
  printf("This control packet is out of order.\n");
  return c;
}
else if ((started) && (c_byte != C_END_PACKET)) {
  printf("This control packet is out of order.\n");
  return c;
}

unsigned char t;
int l;
while (index < p.size) {
  t = p.packet_bytes[index ++];
  l = p.packet_bytes[index ++];

  switch(t) {
    case T_FILENAME:
      c.filename_size = l;
      for (int i = 0; i < l; ++ i) {
        c.filename[i] = p.packet_bytes[index ++];
      }
      break;
    case T_LENGTH:
      c.file_size = 0;
      for (int i = 0; i < l; ++ i) {
        c.file_size = c.file_size * SIZE_PER_BYTE + p.packet_bytes[index ++];
      }
      break;
    default:
      printf("This type is not defined.\n");
      break;
  }
}

if (!started) { started = TRUE; start_packet = c; }
else if ((started) && ((strcmp(start_packet.filename, c.filename) != 0) ||
    (start_packet.file_size != c.file_size))) {
  printf("Start Packet did not match Finish Packet.\n");
  c.file_size = 0;
}
else if ((started) && (size != start_packet.file_size)) {
  printf("Received size doesn't match control size.\n");
```

```
  c.file_size = -2;
}
```

Construção de pacotes de dados

```
int packet_number = (last_packet_number + 1) % SIZE_PER_BYTE;
struct packet p;
p.size = 0;

if ((d.size == 0) || (d.size > MAX_DATA_SIZE)) {
  printf("Error with data packet size.\n");
  return p;
}

p.packet_bytes[p.size ++] = C_DATA_PACKET;
p.packet_bytes[p.size ++] = packet_number;
p.packet_bytes[p.size ++] = d.size / SIZE_PER_BYTE;
p.packet_bytes[p.size ++] = d.size % SIZE_PER_BYTE;

for (int index = 0; index < d.size; ++ index) {
  p.packet_bytes[p.size ++] = d.data[index];
}

last_packet_number = packet_number;
```

Construção de pacotes de controlo

```
struct packet p;
p.size = 0;
unsigned char array[SIZE_PER_BYTE];
int size_aux = c.file_size;
int m = SIZE_PER_BYTE, divisions = 0;

if (!started) { p.packet_bytes[p.size ++] = C_START_PACKET; started = TRUE; }
else          { p.packet_bytes[p.size ++] = C_END_PACKET; }
p.packet_bytes[p.size ++] = T_LENGTH;

while (size_aux > 0) {
  array[-- m] = size_aux % SIZE_PER_BYTE;
  size_aux    = size_aux / SIZE_PER_BYTE;
  ++ divisions;
}

p.packet_bytes[p.size ++] = divisions;

for (int i = 0; i < divisions; ++ i) {
  p.packet_bytes[p.size ++] = array[m ++];
}

p.packet_bytes[p.size ++] = T_FILENAME;
p.packet_bytes[p.size ++] = c.filename_size;
for (int i = 0; i < c.filename_size; ++ i) {
  p.packet_bytes[p.size ++] = c.filename[i];
}
```

10.13 Anexo XIII - Envio/receção de pacotes de dados/controlo

Envio de Pacotes de Controlo

```
struct control_packet c;
c.file_size = get_size_from_file(filename);
if (c.file_size == 0) {
  printf("Error when obtaining the file size.\n");
  llclose(fd);
  return 1;
}
c.filename_size = sizeof(FILENAME);
strcpy(c.filename, filename);

p = construct_control_packet(c);
if (p.size <= 0) {
  printf("Error when constructing the control packet.\n");
  llclose(fd);
  return 1;
}
if (llwrite(fd, p.packet_bytes, p.size) < 0) {
  printf("Error when sending the starting control packet.\n");
  llclose(fd);
  return 1;
}
```

Envio de Pacotes de Dados

```
d = read_from_file(filename);
if (d.size > 0) {
  p = construct_data_packet(d);
  if (p.size <= 0) {
    printf("Error when constructing the data packet.\n");
    llclose(fd);
    return 1;
  }
  if (llwrite(fd, p.packet_bytes, p.size) < 0) {
    printf("Error when sending message.\n");
    llclose(fd);
    return 1;
  }
}
```

Receção de Pacotes de Controlo/Dados

```
p.size = llread(fd, p.packet_bytes);
if (p.size != -1) {
    if (is_data_packet(p)) {
    d = deconstruct_data_packet(p);
    if (d.size == 0) { printf("Error when obtaining information from logical link.\n");
        }
    if (write_to_file(c.filename, d) == -1) { printf("Error when writing to file.\n"); }
    }
else {
    c = deconstruct_control_packet(p);
```

```
    if (c.file_size == 0) { printf("Error when obtaining control information from
        logical link.\n"); }
    if (c.filename_size == 0) { strcpy(c.filename, FILENAME); }
}
```

10.14 Anexo XIV - Leitura e escrita de ficheiros

Leitura de um ficheiro

```
d.size = fread(d.data, 1, MAX_READ_CHARS, f);
if (ferror(f)) { printf("Error when reading from file.\n"); return d; }

if (d.size < MAX_READ_CHARS) {
  aux_f = NULL;
  finished = TRUE;
  if (fclose(f) != 0) { printf("Error when closing the file.\n"); }
}
else aux_f = f;
```

Escrita de um ficheiro

```
int res = fwrite(d.data, 1, d.size, f);
if (res != d.size) { printf("Error when writing to file.\n"); }
```

10.15 Anexo XV - Criação de ficheiros

```
FILE * f = fopen(filename, "a");
if (f == NULL) { printf("Couldnt't open file.\n"); }
```

10.16 Anexo XVI - Validação

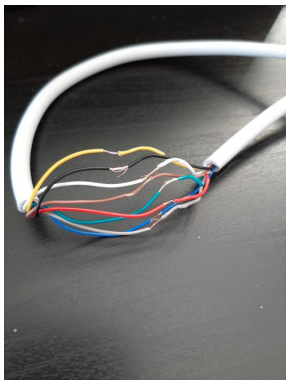

Figura 2: Cabos Utilizados para o Curto Circuito

10.17 Anexo XVII - Variação do Tamanho das Tramas

Tamanho da Trama I	R/C	Tempo (s)	R (bits/s)
64	0.006585	694	252.8645533
128	0.0130199	351	499.965812
256	0.0255307	179	980.3798883
512	0.0496739	92	1907.478261
1024	0.0914	50	3509.76
2048	0.1632143	28	6267.428571
4096	0.2538889	18	9749.333333
6144	0.3264286	14	12534.85714
8192	0.4154545	11	15953.45455
9216	0.3808333	12	14624
10240	0.3808333	12	14624
11264	0.457	10	17548.8
12288	0.5077778	9	19498.66667

Figura 3: Tabela da Variação do Tamanho das Tramas

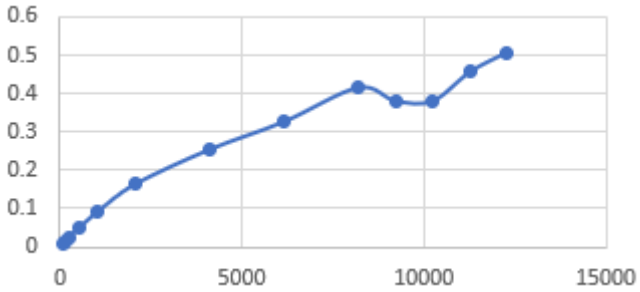

Figura 4: Gráfico da Variação do Tamanho das Tramas

10.18 Anexo XVIII - Variação da Capacidade de Ligação *Baudrate*

Baudrate	R/C	Tempo (s)	R (bits/s)
4800	0.6303448	58	3025.655172
9600	0.5077778	36	4874.666667
19200	0.3808333	24	7312
38400	0.2538889	18	9749.333333
57600	0.1904167	16	10968
115200	0.1088095	14	12534.85714

Figura 5: Tabela da Variação da Capacidade de Ligação

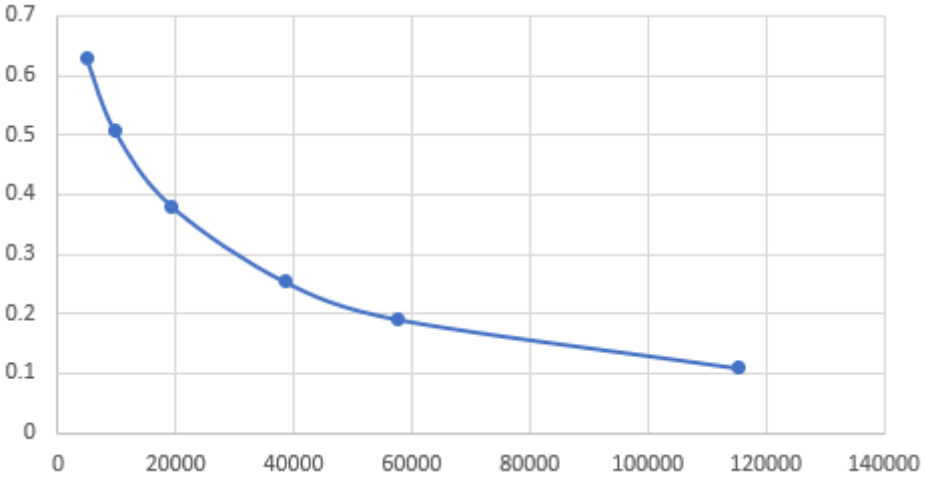

Figura 6: Gráfico da Variação da Capacidade de Ligação

10.19 Anexo XIX - Geração do Atraso de Propagação

Atraso	R/C	Tempo (s)	R (bit/s)
0	2.285	2	87744
1	0.57125	8	21936
2	0.2538889	18	9749.333333
3	0.2285	20	8774.4

Figura 7: Tabela da Geração do Atraso de Propagação

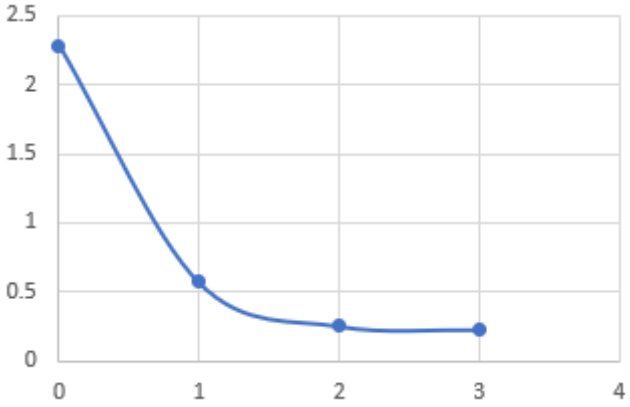

Figura 8: Gráfico da Geração do Atraso de Propagação

10.20 Anexo XX - Geração Aleatória de Erros

Percentagem de Erros	R/C	Tempo (s)	R (bit/s)
10	0.2538889	18	9749.333333
20	0.1904167	24	7312
30	0.1904167	24	7312
40	0.1757692	26	6749.538462
50	0.1757692	26	6749.538462
60	0.1202632	38	4618.105263

Figura 9: Tabela da Geração Aleatória de Erros

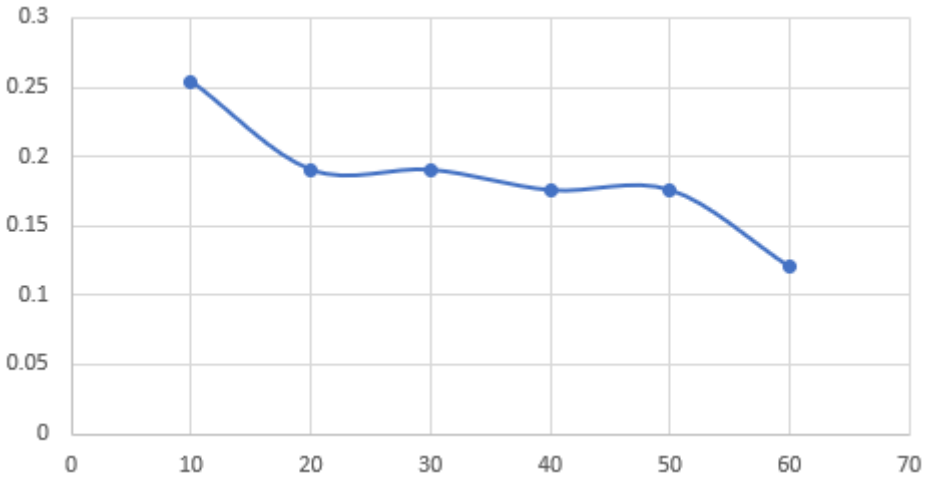

Figura 10: Gráfico da Geração Aleatória de Erros