U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Protocolo de Ligação de Dados

1º Trabalho Laboratorial
Redes de Computadores

Mestrado Integrado em Engenharia Informática e Computação

(5 de novembro de 2018)

Nuno Tiago Tavares Lopes.	up201605337@fe.up.pt
Amadeu Prazeres Pereira	up201605646@fe.up.pt
João Carlos Parada Alves.	up201605236@fe.up.pt
Mateus Pedroza Cortes Marques.	up201601876@fe.up.pt

Índice

Sumário

Este relatório foi elaborado no âmbito da unidade curricular de Redes e Computadores, do curso Mestrado Integrado em Engenharia Informática e Computação. O relatório é baseado no primeiro trabalho laboratorial cuja sua essência é a transferência de dados entre computadores utilizando uma porta de série. Utilizou-se para o caso um protocolo interiorizado nas aulas teóricas e laboratoriais.

O trabalho foi concluído com sucesso, sendo que todos os objetivos estabelecidos no guião foram cumpridos tendo como resultado uma aplicação totalmente funcional capaz de enviar qualquer tipo de ficheiro entre dois computadores utilizando a porta de série sem perda de dados.

Introdução

O objetivo deste trabalho é implementar um protocolo de ligação de dados, de acordo com o guião fornecido, e testar o protocolo com uma aplicação simples de transferência de ficheiros, recorrendo a uma porta de série. Este relatório está dividido nos seguintes tópicos:

- **Arquitetura**

  Exibição dos blocos funcionais e interfaces presentes.

- **Estrutura do código**

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

- **Casos de usos principais**

  Identificação dos mesmos e demonstração das sequências de chamada de funções.

- **Protocolo de ligação lógica**

  Identificação dos principais aspetos funcionais e descrição da estratégia de implementação dos mesmos com apresentação de extratos de código.

- **Protocolo de aplicação**

  Identificação dos principais aspetos funcionais e descrição da estratégia de implementação dos mesmos com apresentação de extratos de código.

- **Validação**

  Descrição dos testes efetuados com apresentação quantificada dos resultados.

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

  Caracterização estatística da eficiência do protocolo, feita com recurso a medidas sobre o código desenvolvido.

❑ **Conclusão**

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

Arquitetura

Layers (camadas)

Para realizar o trabalho, usamos uma camada lógica comum (*low level*, representada pelos ficheiros *logic.c* e *logic.h)* e uma camada de aplicação (*high level*), dividida em duas partes, uma para o ***sender*** (representada pelo ficheiro *sender.c*) e uma para o ***receiver*** (representada pelo ficheiro *receiver.c*).

As funções para o envio e receção de tramas estão contidas na camada lógica, ou seja, código de baixo nível que faz a conexão, verifica erros, escreve e lê da porta série.

As camadas superiores (camadas de aplicação) utilizam as funções da camada lógica para receber / ler dados, e fazem a lógica de escrever / ler de um ficheiro, e gerar / lidar com os dados a serem enviados e recebidos.

CLI (Command Line Interface)

Ao compilar o projeto com ***make*** são gerados dois executáveis, um para o sender e outro para o receiver, e para correr o programa num dos modos é preciso correr o executável desejado.

Execução:

- **sender:**
  ./sender [porta série] [ficheiro a ser enviado]
  Porta série: */dev/ttyS0* ou */dev/ttyS1.*
  Ficheiro a ser enviado: caminho até ao ficheiro que deseja enviar.

- **receiver:**
  ./receiver [porta série]
  Porta série: */dev/ttyS0* ou */dev/ttyS1.*

Estrutura de Código

Application Layer

A application layer encontra-se dividida em 2 ficheiros, o *sender.c* e o *receiver.c*, em que cada um implementa, respetivamente, a parte referente ao emissor e ao recetor.

As funções principais da application layer ao nível do emissor são:

```
int sendStartPackage (const char* filename, size_t fileSize);
```
- Gera e envia ao recetor um pacote *start* (início da transmissão).

```
int sendFPackages (const char* filename);
```
- Gera e envia a pacotes de dados.

```
int sendEndPackage (const char* filename, size_t fileSize);
```
- Gera e envia ao recetor um pacote *end* (fim da transmissão).

Ao nível do recetor são:

```
int handleStartPkg (char *buffer, int size)
```
- Função chamada ao receber um pacote *start*.

```
void handleIPkg (char *buffer, int size)
```
- Função chamada ao receber um pacote de dados.

```
void handleEndPkg (char *buffer, int size)
```
- Função chamada ao receber um pacote *end*.

Data Link Layer

A data link layer encontra-se desenvolvida nos ficheiros *logic.c* e *logic.h*.

Esta camada é representada por uma *struct* com todos os valores necessários para a ligação aí definidos.

```
typedef struct {
    char *port;                     /*Dispositivo /dev/ttySx, x = 0, 1*/
    int baudRate;                   /*Velocidade de transmissão*/
    unsigned int timeout;           /*Valor do temporizador: 1 s*/
    unsigned int numTransmissions;  /*Número de tentativas em caso defalha*/
} LinkLayer;
```

Também se encontra aqui definida uma outra *struct* onde se guarda toda a informação relativa ao que ocorre na ligação (número de tramas recebidas/enviadas, divididos em ***RR's*** e ***REJ's***, número de *time outs*, informação relativa ao tempo médio de envio para cada trama e percentagem de erros).

```
typedef struct {
    size_t sent;
    size_t received;
    size_t time_outs;
    size_t sentRR;
    size_t receivedRR;
    size_t sentREJ;
    size_t receivedREJ;

    size_t framesTotalTime;
    size_t framesCounter;
    float errorProbability_data;
    float errorProbability_header;
} Statistics;
```

Está aqui definido uma outra *struct* que contém informação relativa a uma trama (informação e tamanho):

```
typedef struct {
       char* msg;
       size_t length;
} Frame;
```

As principais funções aqui definidas são:

`int setup(char *port);`
- Função chamada tanto no recetor quanto no emissor, faz a configuração da conexão.

`int llopen(int type);`
- Faz a conexão entre o recetor e o emissor.

`int llwrite(char *buffer, int length);`
- Escreve dados na porta série. Responsável por adicionar *header* e fazer o processo de *stuffing*.

`int llread(char **buffer);`
- Lê dados que foram escritos na porta série. Remove o *header* e faz *destuffing*, evidenciando somente os dados recebidos.

`int llclose();`
- Fecha a conexão entre os dois computadores.

Casos de uso principais

O principal caso de uso da aplicação é o envio de um ficheiro, através da porta de série, entre dois computadores, sendo um o transmissor e o outro o recetor do ficheiro.

Para dar início à aplicação, o recetor é executado na porta de série desejada (ex: */dev/ttyS0*), assim, pode-se correr o transmissor na mesma porta de série e selecionar o ficheiro a ser enviado.

A transmissão de dados segue a seguinte sequência:

1. Recetor e transmissor é executado através de uma porta de série, o transmissor escolhe o ficheiro a ser enviado.
2. É feita a configuração da conexão entre os dois computadores através do protocolo ***llopen***.

3. Com a conexão estabelecida, o transmissor envia os dados através do protocolo ***llwrite***.
4. O recetor recebe os dados através do protocolo ***llread*.**
5. O recetor grava os dados num ficheiro com o mesmo nome do ficheiro enviado pelo transmissor.
6. Após receber todos os dados, a conexão é terminada tanto no recetor como no transmissor através do protocolo ***llclose***.

Protocolo de ligação lógica

As funções **llopen**, **llwrite**, **llread** e **llclose** constituem a API da porta de série e estão implementadas na camada de ligação de dados.

LLOPEN

```
int llopen(int type);
```

A responsabilidade desta função é estabelecer a ligação através da porta de série entre o emissor e o recetor.

Quando esta função é chamada, o emissor começa por enviar o comando **SET** e aguarda pela resposta do recetor, que neste caso é o comando **UA**. Se a resposta por parte do recetor não for recebida no emissor dentro de um tempo estabelecido (*time out*), o comando **SET** é reenviado. O comando **SET** é reenviado um número máximo de vezes preestabelecido, quando este valor for atingido o programa termina.

No caso do recetor, a função espera pela chegada da trama de controlo **SET** e responde com o comando **UA**.

Para a execução destas tarefas são utilizadas duas funções ***sendMsg*** que envia um comando e ***readFrame*** que o lê através da porta de série.

LLWRITE

```
int llwrite(char *buffer, int length);
```

Esta função recebe como parâmetros um *buffer* e o seu tamanho (*length*) para ser transmitido pela porta de série. É também nesta função que são realizados os devidos tratamentos à mensagem antes de ser enviada:

- BCC2: calculado e introduzido no buffer na função ***generateBCC2***.
- Stuffing: na função ***stuffing***.
- Cabeçalho e flag final: presente na função ***createFrame*** que retorna um objeto do tipo ***Frame***.

Depois da mensagem ter sido enviada utilizando a função ***sendFrame***, o emissor fica à espera de uma resposta. Para tal, é utilizado o mesmo mecanismo usado na função ***llopen***, se uma resposta não for recebida num intervalo de tempo pré definido em *time out*, é feita uma nova tentativa de envio da mensagem. Quando uma resposta for recebida, o emissor verifica o seu *Campo de Controlo* para saber o que deverá realizar de seguida. Se o *Campo de Controlo* representar o comando ***RR***, a mensagem foi transmitida corretamente; caso o comando recebido seja ***REJ***, então a mensagem não foi transmitida corretamente e é retransmitida para o recetor.

LLREAD

```
int llread(char **buffer);
```

A função ***llread*** é utilizada pelo recetor com o objetivo de receber uma mensagem através da porta de série.

O procedimento desta função é o inverso da função ***llwrite***, primeiro é removido o cabeçalho e a *flag* final (***deconstructFrame***), de seguida é realizado o ***destuffing*** e por fim é verificado o *BCC2* (***checkBCC2***). Durante este processo caso se verifique algum erro a resposta a ser enviada é ***REJ***, em caso contrário a resposta é ***RR***, recebendo assim o buffer corretamente.

Nesta função também é chamada uma função chamada ***simulateErrors*** que insere erros na trama recebida, tanto no cabeçalho como nos dados, para testar a boa funcionalidade do nosso protocolo.

LLCLOSE

```
int llclose();
```

Esta função é utilizada para terminar a ligação da porta de série entre o emissor e o recetor.

Quando o emissor chamar esta função o comando ***DISC*** é enviado e aguarda pela resposta do comando ***DISC*** por parte do recetor. Assim que recebido o comando ***UA*** é enviado e o emissor termina com sucesso.

No caso do recetor o comando ***DISC*** é recebido na função ***llread*** que de seguida executa a função ***llclose***. Assim que a função ***llclose*** é iniciada o recetor envia o comando ***DISC*** para o emissor e aguarda pela recepção do comando ***UA***. Quando tal acontecer o recetor termina a sua execução com sucesso.

Protocolo de aplicação

O protocolo de aplicação é a camada de mais alto nível responsável pelo envio e pela recepção de pacotes de controlo e de pacotes de dados.

Pacotes de controlo

Os pacotes de controlo START e END são os que marcam, respetivamente, o início e o fim do envio do ficheiro. Estes pacotes são tratados recorrendo às seguintes funções:

```
int sendStartPackage(const char* filename, size_t fileSize) {
    char* startPackage = NULL;
    int    startPackageSize    =    generateStartPackage(filename,    fileSize,
&startPackage);
    int ret;
    do{
      ret = llwrite(startPackage, startPackageSize);
    }while(ret == ERROR);
    free(startPackage);
    return ret;
}
```

```c
int handleStartPkg(char *buffer, int size){
  int i, j;
  size_t l1, l2;
  char * nameFile;
  for (i = 1; i < size; i++){
    //reads file size
    if(buffer[i] == 0x00){
      l1 = buffer[i+1];
      memcpy(&fileSize, buffer+i+2, l1);
      i = i + l1 + 2;
    }
    //reads file name
    if(buffer[i] == 0x01){
      l2 = buffer[i+1];
      i++;
      j= 0;
      nameFile = malloc(l2+1);
      for( ; j < (l2+1); i++, j++){
        if(i >= size)
          return ERROR;
        nameFile[j] = buffer[i+1];
      }
      nameFile[l2] = '\0';
    }
  }
  //Opens file
  file = fopen(nameFile, "wb");
  if (file == NULL){
      printf("Error opening file!\n");
      exit(1);
  }
  printf("Receiving a file named %s with %ld bytes\n", nameFile, fileSize);
  free(nameFile);
  return 0;
}
```

```c
int sendEndPackage(const char* filename, size_t fileSize) {
    char* endPackage = NULL;
    int endPackageSize = generateEndPackage(filename, fileSize, &endPackage);
    int ret;
    do{
      ret = llwrite(endPackage, endPackageSize);
    }while(ret == ERROR);
    free(endPackage);
    return ret;
```

```
void handleEndPkg(char *buffer, int size){
  fclose(file);
}
```

As funções de envio encarregam-se de criar os pacotes de controlo com o tamanho do ficheiro e o nome do ficheiro. Após estar criado este é enviado recorrendo à função ***llwrite***, implementada na camada de ligação de dados.

De forma semelhante, para a recepção dos pacotes de controlo é usada a função ***llread,*** implementada na camada inferior, e dependendo do tipo de pacote de controlo recebido, é chamada a função correspondente. Estas funções retiram do pacote as informações relativas ao nome e ao tamanho do ficheiro.

Pacotes de dados

O envio e a recepção dos pacotes de dados é conseguido recorrendo às seguintes funções:

```
int sendFPackages(const char* filename){
  FILE *file;
  char str[PACKAGE_DATA_SIZE];
  char * fPackage = NULL;
  int counter = 0;
  file = fopen(filename, "rb");
  if(file == NULL)
    return ERROR;
  char * buffer;
  buffer = (char*) malloc((fileSize+1)*sizeof(char));
  fread(buffer, fileSize, 1, file);
  int i = 0, j;
  int STOP = FALSE;
  while(STOP == FALSE){
    for(j = 0; j < PACKAGE_DATA_SIZE; j++, i++){
      if(i == fileSize){
        STOP = TRUE;
        break;
      }
      str[j] = buffer[i];
    }
    generateFPackages(str, &fPackage, counter, j);
    sizeSend += j;
    packageCounter++;
    printProgressBar(sizeSend, fileSize, packageCounter);
```

```
    int ret;
    ret = llwrite(fPackage, j+4);
    free(fPackage);
    if(ret == ERROR) return ERROR;
    counter++;
  }
  fclose(file);
  return 0;
}
```

```
void handleIPkg(char *buffer, int size){
  size_t N = packageCounter%255;
  if((unsigned char) buffer[1] != N){
    printf("Error: received frame with invalid number\n");
    exit(ERROR);
  }
  // parse L1  e L2
  size_t length = 256 * (unsigned char) buffer[2] + (unsigned char) buffer[3];
  if (file == NULL){
    printf("Error: No file opened.\n");
    exit(1);
  }
  char * temp;
  temp = malloc(length);
  int i;
  for(i = 0; i < length; i++) {
    temp[i] = buffer[i+4];
  }
  sizeReceived += length;
  packageCounter++;
  printProgressBar(sizeReceived, fileSize, packageCounter);
  write(fileno(file), temp, length);
}
```

A função de envio recebe o nome do ficheiro a enviar e divide-o em “pacotes”. Encarrega-se de criar um pacote de dados válido, conforme está especificado na documentação, e envia-o usando a função ***llwrite***.

De forma igual à recepção de pacotes de controlo, os pacotes de dados são lidos através da função ***llread*** e invocando a respetiva função de *handle* para estes pacotes. Esta função trata de retirar a informação, que é para guardar no ficheiro, do pacote e guarda-a no ficheiro de output.

Validação

Com o intuito de testar a aplicação por nós implementada foram realizados vários testes. Além de enviarmos o ficheiro dado (pinguim.gif) enviamos também outros ficheiros. Estes ficheiros apresentavam diferentes tamanhos e eram de tipos diferentes (audio, imagem, gif). Todos os ficheiros foram corretamente enviados.

Também implementamos uma simulação para a ocorrência de erros na trama recebida pelo recetor, erros estes que podem aparecer no cabeçalho da trama (com uma certa probabilidade) e/ou no campo de dados desta (com uma probabilidade independente da do cabeçalho). Verificamos que ao aumentar a probabilidade de erro no cabeçalho da trama o número de *time outs* ocorridos durante a ligação aumenta e que ao aumentar a probabilidade de erros do campo de dados o número de **REJ** enviados aumentou. Também se verifica que ao aumentar estas percentagens de erro o tempo médio de envio de uma trama aumenta. Este comportamento está conforme o que está definido no protocolo de dados.

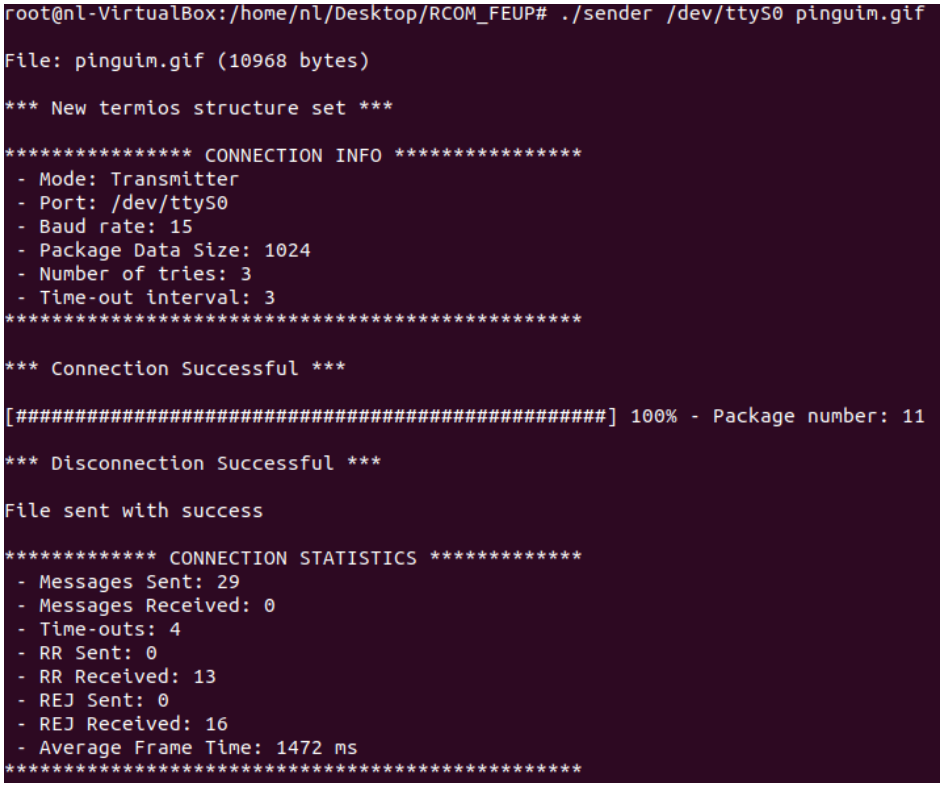


Figure 1 - Modo Emissor

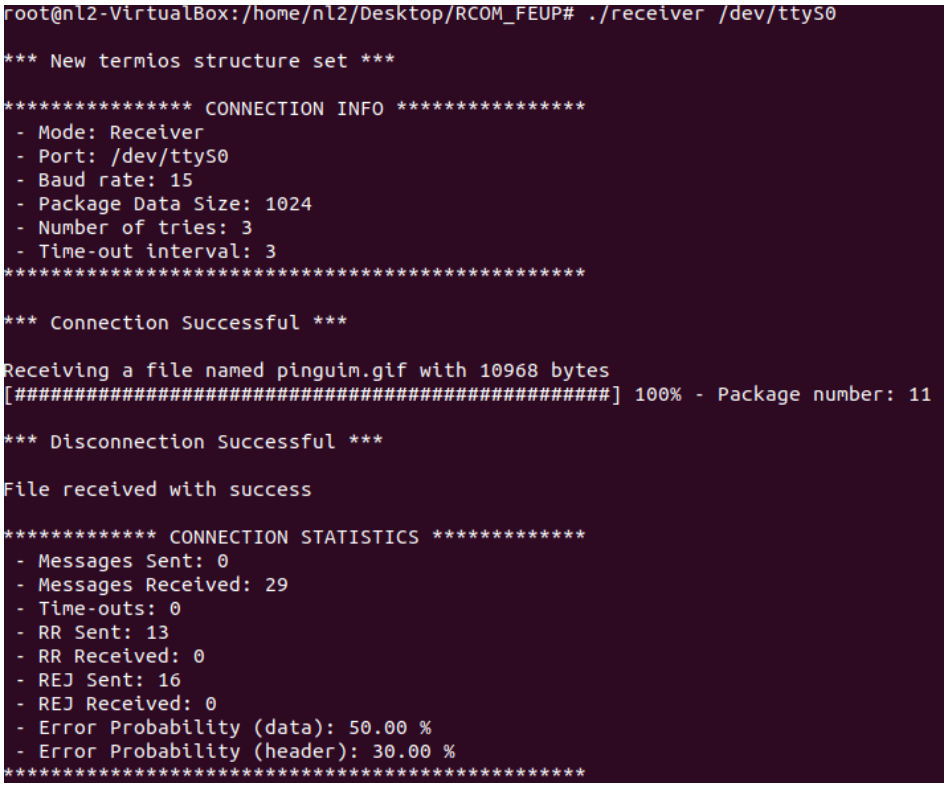


Figure 2 - Modo Recetor

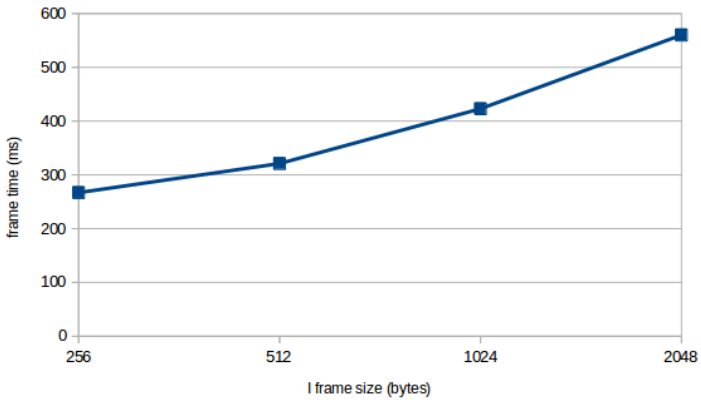
Todos os envios feitos foram realizados várias vezes variando o tamanho dos pacotes, interrompendo a ligação e causando curto circuitos manualmente várias vezes durante esse mesmo envio. Os ficheiros foram sempre enviados com sucesso.

Eficiência do protocolo de ligação de dados

Para a caracterização estatística da eficiência do protocolo foram feitos testes no programa desenvolvido de modo a obter resultados provenientes da alteração de certas variáveis.

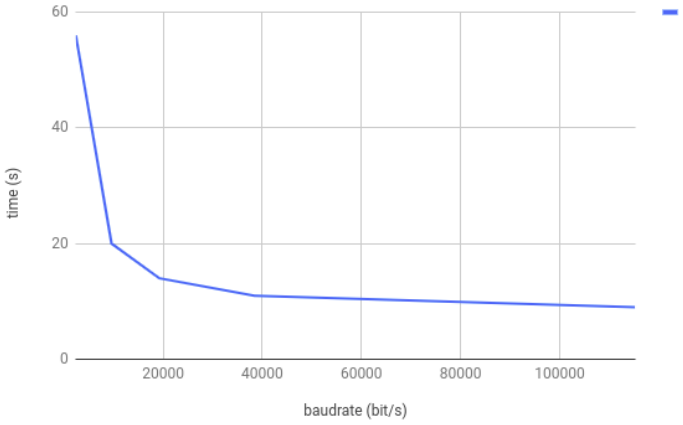
Variável: tamanho da trama I

Utilizando uma imagem de tamanho constante (10968 bytes) e usando um baud rate constante de 38400, fazendo alteração do tamanho da trama I, obteve-se:



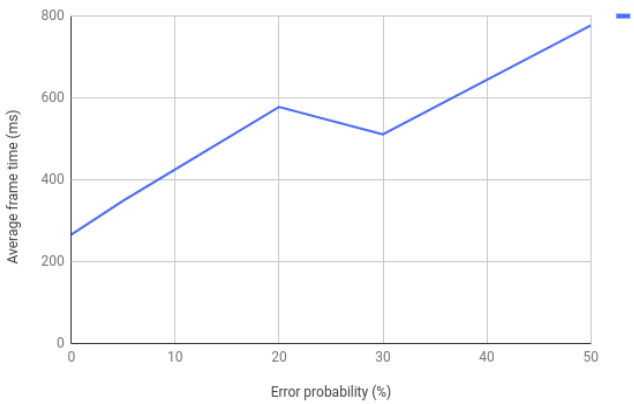
Variável: C (capacidade de ligação)

Utilizando uma imagem de tamanho constante (10968 bytes) e usando um tamanho fixo de 256 bytes por trama I, fazendo variar o *baud rate*, obteve-se:



Variável: FER

Utilizando uma imagem de tamanho constante (10968 bytes), um tamanho fixo de trama I de valor 256 bytes e um baud rate constante de 38400, fazendo variar a probabilidade de erros no cabeçalho da trama, obteve-se:



O protocolo utilizado é baseado num simples mecanismo de *ARQ* (*Automatic Repeat ReQuest*) conhecido como *Stop & Wait*.

Depois de cada envio de uma trama o emissor fica à espera até obter uma resposta por parte do recetor. Se em tramas I recebidas não forem detetados erros no cabeçalho e no campo de dados estas são aceites para processamento e o recetor confirma com ***RR*** (correspondente a um *ACK*), caso sejam detetados erros no campo de dados o recetor faz um pedido de retransmissão enviando um ***REJ*** (correspondente a um *NACK*). Na eventualidade de passar o tempo máximo de espera pela resposta há ocorrência de um *time out* e é efetuado um número máximo de tentativas de retransmissão (valor configurável).

Conclusões

O grupo tinha como objetivo implementar um protocolo de ligação de dados desenvolvendo um serviço de comunicação de dados fiável entre dois sistemas ligados por um canal de transmissão (cabo série).

Relativamente ao trabalho desenvolvido foi possível consolidar conhecimentos referentes a canais de comunicação e controlo de ligação de dados, alguns dos quais como sincronismo, estabelecimento/terminação da ligação e controlo de erros.

As dificuldades sentidas durante a construção do código são referentes à divisão de camadas e por sua vez ao conceito de Independência entre camadas e também à otimização do controlo de erros desenvolvido na parte correspondente à ligação de dados.

Após finalização, o grupo conclui que todos os objetivos relacionados com a estrutura e robustez do trabalho foram atingidos e que vários conceitos e aspetos relativos ao tema em questão, adquiridos em aulas teóricas, foram aprofundados.

Anexo I

Logic.h

```
// Logic -- Data Link Layer

#include <stddef.h>

#define PACKAGE_DATA_SIZE 1024
#define BAUDRATE B38400
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FALSE 0
#define TRUE 1
#define TRANSMITTER 1
#define RECEIVER 2
#define ERROR -1
#define BUFFER_MAX_SIZE 255
#define L2 0x08
#define XOR(a, b) a^b
#define ERROR_PROBABILITY_DATA 50     //percentage
#define ERROR_PROBABILITY_HEADER 30   //percentage

#define NUMBER_OF_TRIES 3

#define F 0x7e
#define A1 0x03
#define A2 0x01

#define SET 1
#define SET_C 0x03
#define SET_BCC1 0x00

#define DISC 2
#define DISC_C 0x0b

#define UA 3
#define UA_C 0x07
#define UA_BCC1 A1 ^ UA_C

#define RR0 4
#define RR0_C 0x05

#define RR1 5
#define RR1_C 0x85
```

```c
#define REJ0 6
#define REJ0_C 0x01

#define REJ1 7
#define REJ1_C 0x81

#define I0 8
#define I0_C 0x00

#define I1 9
#define I1_C 0x40

typedef struct {
    char* msg;
    size_t length;
} Frame;

typedef struct {
    char *port;                     /*Dispositivo /dev/ttySx, x = 0, 1*/
    int baudRate;                   /*Velocidade de transmissão*/
    unsigned int timeout;           /*Valor do temporizador: 1 s*/
    unsigned int numTransmissions;  /*Número de tentativas em caso defalha*/
} LinkLayer;

typedef struct {
    size_t sent;
    size_t received;
    size_t time_outs;
    size_t sentRR;
    size_t receivedRR;
    size_t sentREJ;
    size_t receivedREJ;

    size_t framesTotalTime;
    size_t framesCounter;
    float errorProbability_data;
    float errorProbability_header;
} Statistics;

int setup(char *port);
int llopen(int type);
int llwrite(char *buffer, int length);
int llread(char **buffer);
int llclose();
void copyBuffer(char *dest, char *source, int length);
void alarm_function();
void printProgressBar(int sizeReceived, int fileSize, size_t packageNumber);
void connectionStatistics();
```

Logic.c

```c
// Logic -- Data Link Layer

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

struct termios oldtio, newtio;
LinkLayer linkLayer;
Statistics statistics;
struct timeval timerStart, timerEnd;

int flag = 0;
int counter = 0;
int fd;
char C_FLAG = 0x0;
int program;

int checkFrame(Frame f);
int sendMsg(Frame f);
int llclose_Receiver();
int llclose_Sender();
void connectionInfo();

char getCFlag(){
    if (C_FLAG == 0x00){
        C_FLAG = 0x40;
        return 0x0;
    } else {
        C_FLAG = 0x00;
        return 0x40;
    }
}
```

```c
void alarm_function(){
    statistics.time_outs++;
    flag=1;
    counter++;
}

void createStructLinkLayer(char *port){
    linkLayer.baudRate = BAUDRATE;
    linkLayer.numTransmissions = NUMBER_OF_TRIES;
    linkLayer.port = port;
    linkLayer.timeout = 3;
}

void initializeStatistics(){
    statistics.sent = 0;
    statistics.received = 0;
    statistics.time_outs = 0;
    statistics.sentRR = 0;
    statistics.receivedRR = 0;
    statistics.sentREJ = 0;
    statistics.receivedREJ = 0;
    statistics.framesTotalTime = 0;
    statistics.framesCounter = 0;
    statistics.errorProbability_data = (float)ERROR_PROBABILITY_DATA;
    statistics.errorProbability_header = (float)ERROR_PROBABILITY_HEADER;
}

int setup(char *port) {
  createStructLinkLayer(port);
  initializeStatistics();
  fd = open(linkLayer.port, O_RDWR | O_NOCTTY );
  if (fd <0) {perror(linkLayer.port); return ERROR; }

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

  bzero(&newtio, sizeof(newtio));
  newtio.c_cflag = linkLayer.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]    = 1;   /* inter-character timer unused */
  newtio.c_cc[VMIN]     = 0;   /* blocking read until 5 chars received */
```

```
  tcflush(fd, TCIOFLUSH);

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

  printf("\n*** New termios structure set ***\n");
  return fd;
}

int readFrame(Frame* f){

    char* buf = malloc(sizeof(char) * 10);
    int bufferSize = 10;
    int res, i = 0;
    int state = 0;

    while(flag == 0){
        if(i == bufferSize) {
            bufferSize *= 2;
            buf = realloc(buf, sizeof(char) * bufferSize);
        }

        res = read(fd,buf+i,1);

        switch(state) {
        case 0:
            if(res > 0 && *(buf+i) == F) {
                i++;
                state++;
            }
            break;
        case 1:
            if(res > 0 && *(buf+i) != F) {
                i++;
                state++;
            }
            break;
        case 2:
            if(res > 0 && *(buf+i) != F) {
                i++;
            }
            else if (res > 0 && *(buf+i) == F) {
                state++;
                i++;
            }
```

```
            break;
        default:
            f->msg = buf;
            f->length = i;
            return 0;
        }
    }

    return ERROR;
}

int rejectFrame(char cflag){
    char c;
    if (cflag == I1_C){
        c = REJ1_C;
    }
    else if (cflag == I0_C){
         c = REJ0_C;
    }
    else{
        return ERROR;
    }

    char* temp = malloc(sizeof(char) * 5);
    temp[0] = F; temp[1] = A1; temp[2] = c; temp[3] = XOR(A1, c); temp[4] = F;

    Frame rej = {
        .msg = temp,
        .length = 5
    };

    // write reject
    int ret = sendMsg(rej);
    free(rej.msg);

    statistics.sentREJ++;

    return ret;
}

int acceptFrame(char cflag){
    char c;
    if (cflag == I1_C){
        c = RR0_C;
    }
    else if (cflag == I0_C){
         c = RR1_C;
    }
```

```
    else{
        return ERROR;
    }

    char* temp = malloc(sizeof(char) * 5);
    temp[0] = F; temp[1] = A1; temp[2] = c; temp[3] = XOR(A1, c); temp[4] = F;

    Frame rr = {
        .msg = temp,
        .length = 5
    };

    // write accept
    int ret = sendMsg(rr);
    free(rr.msg);

    statistics.sentRR++;

    return ret;
}

char* destuffing(char* buf, int* size) {
    int length = *size;
    int new_size = 0;
    int i, j;

    for(i = 0; i < length; i++) {
        if (buf[i] == 0x7d) {
            i++;
        }
        new_size++;
    }

    char* ret = malloc(sizeof(char) * new_size);

    for (i = 0, j = 0; i < length; i++, j++) {
        if(buf[i] == 0x7d) {
            if(buf[i+1] == 0x5e) {
                ret[j] = 0x7e;
                i++;
            }
            else if(buf[i+1] == 0x5d) {
                ret[j] = 0x7d;
                i++;
            }
            else
                return NULL;
        }
```

```
        else {
            ret[j] = buf[i];
        }
    }

    *size = new_size;
    return ret;
}

char* deconstructFrame(Frame f, int* size) {
    int new_size = f.length - 5;
    char* ret = malloc(sizeof(char) * new_size);
    ret = memcpy(ret, f.msg + 4, new_size);

    *size = new_size;
    return ret;
}

int checkBCC2(char* buf, int size) {
    int i, bcc2 = buf[0];

    for(i = 1; i < size - 1; i++) {
        bcc2 = XOR(bcc2, buf[i]);
    }

    if(bcc2 == buf[size - 1]) {
        return TRUE;
    }
    else {
        return FALSE;
    }
}

void simulateErrors(Frame* f) {
    size_t length = f->length;

    srand((unsigned int) time(NULL));

    float r = rand() / ((float) RAND_MAX);
    r *= 100;

    //Insert error in data
    if(r < statistics.errorProbability_data) {
        int index = rand() % (length - 6);
        index += 4;
        f->msg[index] = rand();
    }
```

```
    //Insert error in frame headers
    r = rand() / ((float) RAND_MAX);
    r *= 100;
    if(r < statistics.errorProbability_header) {
        int index = rand() % 4;
        index += 1;
        if(index == 4) index = length - 2;
        f->msg[index] = rand();
    }
}

int llread(char **buffer){

    Frame f;

    int ret = readFrame(&f);

    if(ret == ERROR)
        return ERROR;

    if((f.msg[2] == I0_C || f.msg[2] == I1_C) && f.msg[4] == 0x01) {
        simulateErrors(&f);
    }

    int frame_type = checkFrame(f);

    if(frame_type == ERROR) {
        return 0;
    }

    if(frame_type == DISC && f.msg[1] == A1) {
        return -2;
    }

    statistics.received++;
    if(frame_type != I0 && frame_type != I1) {
        rejectFrame(C_FLAG);
        return 0;
    }
    if (f.msg[2] != C_FLAG) {
        rejectFrame(C_FLAG);
        return 0;
    }

    int size;

    // remover cabeçalho e flag inicial
    char* buf1 = deconstructFrame(f, &size);
```

```
    free(f.msg);

    // extrair pacote de comando da trama - destuffing
    char* buf2 = destuffing(buf1, &size);
    free(buf1);

    if(buf2 == NULL) {
        rejectFrame(C_FLAG);
        return 0;
    }

    // ver valor do bcc2 se está correcto
    if(checkBCC2(buf2, size)) {
        size--;
        buf2 = realloc(buf2, sizeof(char) * size);
    }
    else {
        rejectFrame(C_FLAG);
        return 0;
    }

    //check data L1 and L2 with actual size
    if(buf2[0] == 0x01) {
        size_t data_size = 256 * (unsigned char) buf2[2] + (unsigned char) buf2[3];
        if(data_size != size - 4) {
            rejectFrame(C_FLAG);
            return 0;
        }
    }

    *buffer = buf2;
    acceptFrame(C_FLAG);

    getCFlag();

    return size;
}

int llopen_Receiver(){

    char* temp = malloc(sizeof(char) * 5);
    temp[0] = F; temp[1] = A1; temp[2] = UA_C; temp[3] = UA_BCC1; temp[4] = F;

    Frame ua = {
        .msg = temp,
        .length = 5
    };
```

```
    Frame f;
    int ret;

    while(1){
        ret = readFrame(&f);

        if (ret != ERROR) {
            int frame_type = checkFrame(f);
            if(frame_type == SET  && f.msg[1] == A1) {
                sendMsg(ua);
                free(f.msg);
                free(ua.msg);
                return 0;
            }
        }
     }

    return ERROR;
}

int sendMsg(Frame f) {

    // write on serial port
    int res = write(fd, f.msg, sizeof(char) * f.length);
    return res;
}

int sendFrame(Frame f, Frame* response){
    int STOP = FALSE;

    // reset global counter
    counter = 0;

    while (STOP==FALSE && counter < linkLayer.numTransmissions) {
        sendMsg(f);
        alarm(linkLayer.timeout);
        flag = 0;
        if (readFrame(response) != ERROR){
            alarm(0);
            STOP = TRUE;
        }
  }

  if(STOP != TRUE)
    return ERROR;

  return 0;
}
```

```
int checkFrame(Frame f) {
    if(f.msg[3] != (XOR(f.msg[1], f.msg[2]))) {
        return ERROR;
    }

    switch(f.msg[2]) {
        case (char)SET_C:
            return SET;
        case (char)DISC_C:
            return DISC;
        case (char)UA_C:
            return UA;
        case (char)RR0_C:
            return RR0;
        case (char)RR1_C:
            return RR1;
        case (char)REJ0_C:
            return REJ0;
        case (char)REJ1_C:
            return REJ1;
        case (char)I0_C:
            return I0;
        case (char)I1_C:
            return I1;
        default:
            return ERROR;
    }
}

int llopen_Sender(){
    char* temp = malloc(sizeof(char) * 5);
    temp[0] = F; temp[1] = A1; temp[2] = SET_C; temp[3] = SET_BCC1; temp[4] = F;

    Frame set = {
        .msg = temp,
        .length = 5
    };

    Frame response;

    int ret = sendFrame(set, &response);

    free(set.msg);

    if (ret != ERROR) {
        int frame_type = checkFrame(response);
        if(frame_type == UA  && response.msg[1] == A1) {
```

```
            free(response.msg);
            return 0;
        }
    }

    return ERROR;
}

int llopen(int type){
  program = type;

  connectionInfo();

  if(program == TRANSMITTER){
    return llopen_Sender();
  }
  else if (program == RECEIVER){
    return llopen_Receiver();
  }

  return ERROR;
}

char *generateBCC2(char *buffer, int *size){
    int length = *size;
    char *buff1 = malloc(sizeof(char) * (length + 1));

    int i;
    int bcc2 = buffer[0];
    for(i = 1; i < length; i++) {
        bcc2 = XOR(buffer[i], bcc2);

    }

    memcpy(buff1, buffer, length);

    // add bcc2 to buff1
    buff1[length] = bcc2;

    *size = length + 1;

    return buff1;
}

char *stuffing(char *buffer, int *finalSize){
    int length = *finalSize;
    int i, j, new_size = 0;
```

```
    // count number of 0x7e and 0x7d
    for(i = 0; i < length; i++){
        if(buffer[i] == 0x7e || buffer[i] == 0x7d){
            new_size++;
        }
        new_size++;
    }

    char *buff = malloc(sizeof(char) * new_size);

    for(i = 0, j= 0; i < length; i++){
        if(buffer[i] == 0x7e){
            buff[j] = 0x7d;
            buff[j+1] = 0x5e;
            j++;
        }
        else if (buffer[i] == 0x7d){
            buff[j] = 0x7d;
            buff[j+1] = 0x5d;
            j++;
        } else {
            buff[j] = buffer[i];
        }
        j++;
    }

    *finalSize = new_size;
    return buff;
}

Frame createFrame(char *buffer, int size){
    int length = size;
    char *msg = malloc(length + 5);

    msg[0] = F;
    msg[1] = A1;
    msg[2] = getCFlag();

    // add bcc1
    msg[3] = XOR(msg[1], msg[2]);

    // copy buffer
    memcpy(msg+4, buffer, length);

    msg[length + 4] = F;

    Frame frame = {
        .msg = msg,
```

```
        .length = length + 5
    };

    return frame;
}

int llwrite(char *buffer, int length){
    int finalSize = length;

    // calculate BCC2
    char *buff1 = generateBCC2(buffer, &finalSize);

    //stuffing
    char *buff2 = stuffing(buff1, &finalSize);
    free(buff1);

    // junta cabecalho e flag final
    Frame f = createFrame(buff2, finalSize);
    free(buff2);

    Frame response;

    int frame_type_send = checkFrame(f);
    int rej = 1;

    gettimeofday(&timerStart, NULL);

    do {
        int ret = sendFrame(f, &response);
        if(ret != ERROR) {
            statistics.sent++;
            int frame_type_response = checkFrame(response);

            if((frame_type_response == RR0) || (frame_type_response == RR1))
                statistics.receivedRR++;
            if((frame_type_response == REJ0) || (frame_type_response == REJ1))
                statistics.receivedREJ++;

            if(frame_type_response != ERROR){
                if ((frame_type_send == I1 && frame_type_response == RR0) ||
                    (frame_type_send == I0 && frame_type_response == RR1)) {
                    rej = 0;
                }
                else if ((frame_type_send == I1 && frame_type_response == REJ1) ||
                    (frame_type_send == I0 && frame_type_response == REJ0)) {
                    rej = 1;
                }
                else {
```

```
                    rej = 0;
                }
            }
            else{
                rej = 1;
            }
        }
        else {
            return ERROR;
        }
    } while(rej);

    gettimeofday(&timerEnd, NULL);
    statistics.framesTotalTime += (timerEnd.tv_sec - timerStart.tv_sec)*1000.0f +
                                                                    (timerEnd.tv_usec    -
timerStart.tv_usec)/1000.0f;
    statistics.framesCounter++;

    free(f.msg);
    return 0;
}

int llclose(){
  int ret;

  if(program == TRANSMITTER){
    ret = llclose_Sender();
  }
  else if (program == RECEIVER){
    ret = llclose_Receiver();
  }

  if ( tcsetattr(fd,TCSANOW,&oldtio) == ERROR) {
    perror("tcsetattr");
    return ERROR;
  }

  if(close(fd) < 0 || ret < 0) {
    return ERROR;
  }
  return 0;
}

int llclose_Receiver() {
  char* temp = malloc(sizeof(char) * 5);
  temp[0] = F; temp[1] = A2; temp[2] = DISC_C; temp[3] = XOR(A2, DISC_C); temp[4] = F;

  Frame disc = {
```

```
    .msg = temp,
    .length = 5
  };

  Frame f;
  int ret;
    sendMsg(disc);
    free(disc.msg);

  ret = readFrame(&f);
  if(ret != ERROR) {
    int frame_type = checkFrame(f);
    if(frame_type == UA && f.msg[1] == A2) {
      free(f.msg);
      return 0;
    }
  }

  return ERROR;
}

int llclose_Sender() {
  char* temp = malloc(sizeof(char) * 5);
  temp[0] = F; temp[1] = A1; temp[2] = DISC_C; temp[3] = XOR(A1, DISC_C); temp[4] = F;

  Frame disc = {
    .msg = temp,
    .length = 5
  };

  Frame response;
  int ret = sendFrame(disc, &response);
  free(disc.msg);

  if(ret != ERROR) {
    int frame_type = checkFrame(response);
    if(frame_type == DISC && response.msg[1] == A2) {
      free(response.msg);

      //creating ua frame
      char* temp1 = malloc(sizeof(char) * 5);
      temp1[0] = F; temp1[1] = A2; temp1[2] = UA_C; temp1[3] = XOR(A2, UA_C); temp1[4]
= F;

      Frame ua = {
        .msg = temp1,
        .length = 5
      };
```

```
      ret = sendMsg(ua);

      return 0;
    }
  }
  return ERROR;
}

void printProgressBar(int sizeReceived, int fileSize, size_t packageNumber){
    int j, n, m;
    m = sizeReceived*100/fileSize;
    printf("\r[");
    n = m*50/100;
    if(sizeReceived >= fileSize){
        m=100;
    }
    for(j = 0; j < 50; j++){
        if(n >= 0){
            printf("#");
            n--;
        }
        else{
            printf(" ");
        }
    }
    printf("] %d%% - Package number: %ld", m, packageNumber);
    if(m==100)
        printf("\n\n");
    fflush(stdout);
}

void connectionInfo(){
    char *mode;
    if(program == TRANSMITTER){
        mode = "Transmitter";
    }
    else{
        mode = "Receiver";
    }
    printf("\n");
    printf("**************** CONNECTION INFO ****************\n");
    printf(" - Mode: %s\n", mode);
    printf(" - Port: %s\n", linkLayer.port);
    printf(" - Baud rate: %d\n", linkLayer.baudRate);
    printf(" - Package Data Size: %d\n", PACKAGE_DATA_SIZE);
    printf(" - Number of tries: %d\n", linkLayer.numTransmissions);
    printf(" - Time-out interval: %d\n", linkLayer.timeout);
```

```
    printf("************************************************\n");
    printf("\n");
}

void connectionStatistics(){
    printf("\n");
    printf("************* CONNECTION STATISTICS *************\n");
    printf(" - Messages Sent: %ld\n", statistics.sent);
    printf(" - Messages Received: %ld\n", statistics.received);
    printf(" - Time-outs: %ld\n", statistics.time_outs);
    printf(" - RR Sent: %ld\n", statistics.sentRR);
    printf(" - RR Received: %ld\n", statistics.receivedRR);
    printf(" - REJ Sent: %ld\n", statistics.sentREJ);
    printf(" - REJ Received: %ld\n", statistics.receivedREJ);
    if(program == TRANSMITTER)
        printf(" - Average Frame Time: %ld ms\n", statistics.framesTotalTime /
statistics.framesCounter);
    if(program == RECEIVER) {
        printf("     -     Error     Probability     (data):     %.2f     %%     \n",
statistics.errorProbability_data);
        printf("     -     Error     Probability     (header):     %.2f     %%     \n",
statistics.errorProbability_header);
    }
    printf("************************************************\n");
    printf("\n");
}
```

Sender.c

```
// Sender -- Application Layer

#include "logic.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <signal.h>

#define START_C 0x02
#define START_T_FILESIZE 0x00
#define START_T_FILENAME 0x01
#define PACKAGE_C 0x01
#define END_C 0x03
#define END_T_FILESIZE 0x00
#define END_T_FILENAME 0x01
#define RIGHT_SHIFT_CALC(size, index) 8*(size - index - 1)

size_t getFileSize(const char* filename);
int sendStartPackage(const char* filename, size_t fileSize);
int generateStartPackage(const char* filename, size_t fileSize, char** startPackage);
int sendFPackages(const char* filename);
int  generateFPackages(char  *  filedata,  char  **  fPackage,  int  packageCounter,  int
size_package);
int sendEndPackage(const char* filename, size_t fileSize);
int generateEndPackage(const char* filename, size_t fileSize, char** endPackage);

int fd;
size_t fileSize;
size_t packageCounter = 0;
int sizeSend = 0;

int main(int argc, char** argv){

  // parse arguments
  if ((argc < 3) ||
      ((strcmp("/dev/ttyS0", argv[1])!=0) &&
      (strcmp("/dev/ttyS1", argv[1])!=0))) {
    printf("Usage:%s [serial port] [file path]\n", argv[0]);
    exit(-1);
  }

  struct sigaction action;
  action.sa_handler = alarm_function;
```

```
sigemptyset(&action.sa_mask);
action.sa_flags = 0;
sigaction(SIGALRM, &action, NULL);

// ler dados do ficheiro
fileSize = getFileSize(argv[2]);
if(fileSize == -1) {
  printf("Error: could not get file size\n");
  exit(-1);
}
else if(fileSize == -2) {
  printf("Error: %s is not a file\n", argv[2]);
  exit(-1);
}
else {
  printf("\nFile: %s (%ld bytes)\n", argv[2], fileSize);
}

// setup serial port
if ((fd = setup(argv[1])) == ERROR){
  printf("Error: could not setup serial port %s.\n", argv[1]);
  exit(-1);
}

if(llopen(TRANSMITTER) == ERROR){
  printf("Error: could not connect to the receiver\n");
  exit(-1);
}
else {
    printf("*** Connection Successful ***\n\n");
}

// gerar pacote start
if(sendStartPackage(argv[2], fileSize) == ERROR){
  printf("\nError: could not send Start package\n");
  return ERROR;
}

// gerar n pacotes com k dados lidos do ficheiro
if(sendFPackages(argv[2]) == ERROR){
  printf("\nError: could not send F packages\n");
  return ERROR;
}

// gerar pacote start
if(sendEndPackage(argv[2], fileSize) == ERROR){
  printf("\nError: could not send End package\n");
  return ERROR;
```

```
  }

  if(llclose() == ERROR) {
    printf("\nError: could not disconnect\n");
    return ERROR;
  }
  else{
    printf("*** Disconnection Successful ***\n\n");
  }

  printf("File sent with success\n");

  connectionStatistics();
  return 0;
}

size_t getFileSize(const char* filename){
  struct stat fileStat;
  if (stat(filename, &fileStat) < 0 ){
    return -1;
  }
  if (!S_ISREG(fileStat.st_mode)) {
    return -2;
  }
  return fileStat.st_size;
}

int sendStartPackage(const char* filename, size_t fileSize) {
    char* startPackage = NULL;
    int startPackageSize = generateStartPackage(filename, fileSize, &startPackage);

    int ret;
    do{
      ret = llwrite(startPackage, startPackageSize);
    }while(ret == ERROR);

    free(startPackage);
    return ret;
}

int generateStartPackage(const char* filename, const size_t filesize, char** start){
  char* temp;
  int i = 0, j;
  int startPackageSize = 1;

  //filesize tlv
  int filesize_s = sizeof(filesize);
  startPackageSize += 2 + filesize_s;
```

```
  //filename tlv
  int filename_s = strlen(filename) * sizeof(char);
  startPackageSize += 2 + filename_s;

  temp = malloc(startPackageSize);
  temp[i++] = START_C;               //C
  temp[i++] = START_T_FILESIZE;      //T
  temp[i++] = filesize_s;            //L

  memcpy(temp+i, &filesize, filesize_s);
  i = i + filesize_s;

  temp[i++] = START_T_FILENAME;  //T
  temp[i++] = filename_s;        //L

  for(j = 0; j < filename_s; j++, i++) {
    temp[i] = *(filename + j);   //V
  }

  *start = temp;
  return startPackageSize;
}

int sendFPackages(const char* filename){
  FILE *file;
  char str[PACKAGE_DATA_SIZE];
  char * fPackage = NULL;
  int counter = 0;

  file = fopen(filename, "rb");
  if(file == NULL)
    return ERROR;

  char * buffer;
  buffer = (char*) malloc((fileSize+1)*sizeof(char));
  fread(buffer, fileSize, 1, file);

  int i = 0, j;
  int STOP = FALSE;

  while(STOP == FALSE){
    for(j = 0; j < PACKAGE_DATA_SIZE; j++, i++){
      if(i == fileSize){
        STOP = TRUE;
        break;
      }
      str[j] = buffer[i];
```

```
    }

    generateFPackages(str, &fPackage, counter, j);

    sizeSend += j;
    packageCounter++;

    printProgressBar(sizeSend, fileSize, packageCounter);

    int ret;
    ret = llwrite(fPackage, j+4);
    free(fPackage);
    if(ret == ERROR) return ERROR;

    counter++;
  }

  fclose(file);
  return 0;
}

int generateFPackages(char * filedata, char ** fPackage, int packageCounter, int
size_package){
  char* temp;
  int i = 0, j;

  temp = malloc(size_package + 4);
  temp[i++] = PACKAGE_C;
  temp[i++] = packageCounter % 255;
  temp[i++] = size_package / 256;
  temp[i++] = size_package % 256;

  for(j = 0; j < size_package; j++, i++) {
    temp[i] = filedata[j];
  }

  *fPackage = temp;
  return 0;
}

int sendEndPackage(const char* filename, size_t fileSize) {
    char* endPackage = NULL;
    int endPackageSize = generateEndPackage(filename, fileSize, &endPackage);

    int ret;
    do{
      ret = llwrite(endPackage, endPackageSize);
    }while(ret == ERROR);
```

```
    free(endPackage);
    return ret;
}

int generateEndPackage(const char* filename, const size_t filesize, char** end){
  char* temp;
  int i = 0, j;
  int endPackageSize = 1;

  //filesize tlv
  int filesize_s = sizeof(filesize);
  endPackageSize += 2 + filesize_s;

  //filename tlv
  int filename_s = strlen(filename) * sizeof(char);
  endPackageSize += 2 + filename_s;

  temp = malloc(endPackageSize);
  temp[i++] = END_C;               //C
  temp[i++] = END_T_FILESIZE;      //T
  temp[i++] = filesize_s;            //L

  memcpy(temp+i, &filesize, filesize_s);
  i = i + filesize_s;

  temp[i++] = END_T_FILENAME;  //T
  temp[i++] = filename_s;        //L

  for(j = 0; j < filename_s; j++, i++) {
    temp[i] = *(filename + j);   //V
  }

  *end = temp;
  return endPackageSize;
}
```

Receiver.c

```c
// Receiver -- Application Layer

#include "logic.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

//states
#define START 0
#define PACKAGE 1
#define END 2
//----

int state;
FILE *file;
size_t fileSize;
size_t packageCounter = 0;
int sizeReceived = 0;

void handleRead(char *buffer, int size);

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

  if(setup(argv[1]) == ERROR) {
    printf("Error: Could not setup serial port.\n");
    exit(1);
  }

  if(llopen(RECEIVER) == ERROR){
    printf("Unable to connect to the sender!\n");
    exit(-1);
  }
  else {
    printf("*** Connection Successful ***\n\n");
  }
```

```
  int readResult;

  char * buffer;
  do {
    readResult = llread(&buffer);

    if(readResult > 0) {
      handleRead(buffer, readResult);
      free(buffer);
    }

  } while(readResult != -2);

  if(llclose() == ERROR){
    printf("Error: could not disconnect!\n");
    exit(-1);
  }
  else{
    printf("*** Disconnection Successful ***\n\n");
  }

  printf("File received with success\n");

  connectionStatistics();
  return 0;
}

int handleStartPkg(char *buffer, int size){
  int i, j;
  size_t l1, l2;
  char * nameFile;

  for (i = 1; i < size; i++){
    //reads file size
    if(buffer[i] == 0x00){
      l1 = buffer[i+1];
      memcpy(&fileSize, buffer+i+2, l1);
      i = i + l1 + 2;
    }
    //reads file name
    if(buffer[i] == 0x01){
      l2 = buffer[i+1];
      i++;
      j= 0;
      nameFile = malloc(l2+1);
      for( ; j < (l2+1); i++, j++){
        if(i >= size)
          return ERROR;
```

```
        nameFile[j] = buffer[i+1];
      }
      nameFile[l2] = '\0';
    }
  }

  //Opens file
  file = fopen(nameFile, "wb");

  if (file == NULL){
      printf("Error opening file!\n");
      exit(1);
  }

  printf("Receiving a file named %s with %ld bytes\n", nameFile, fileSize);
  free(nameFile);
  return 0;
}

void handleIPkg(char *buffer, int size){

  size_t N = packageCounter%255;
  if((unsigned char) buffer[1] != N){
    printf("Error: received frame with invalid number\n");
    exit(ERROR);
  }

  // parse L1  e L2
  size_t length = 256 * (unsigned char) buffer[2] + (unsigned char) buffer[3];

  if (file == NULL){
    printf("Error: No file opened.\n");
    exit(1);
  }

  char * temp;
  temp = malloc(length);
  int i;
  for(i = 0; i < length; i++) {
    temp[i] = buffer[i+4];
  }

  sizeReceived += length;

  packageCounter++;
  printProgressBar(sizeReceived, fileSize, packageCounter);

  write(fileno(file), temp, length);
```

```
}

void handleEndPkg(char *buffer, int size){
  fclose(file);
}

void handleRead(char *buffer, int size){

  //lidar com o c2 do pacote de comando
  char c2 = buffer[0];

  switch(c2){
    case 0x02:
      handleStartPkg(buffer, size);
    break;
    case 0x01:
      handleIPkg(buffer, size);
    break;
    case 0x03:
      handleEndPkg(buffer, size);
    break;
    default:
    break;
  }
}
```