U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Protocolo de Ligação de Dados

FEUP

Redes de Computadores

Trabalho realizado por:

Frederico Lopes

Pedro Pacheco

Índice

Sumário

Este projeto foi criado para a cadeira de Redes de Computadores (RC) com o objetivo de criar um protocolo de ligação de dados. Este projeto serviu para aplicar os conhecimentos adquiridos nas aulas teóricas dando assim uma componente pratica a temas que seriam de outra forma teóricos.

Introdução

Para este projeto foi criado um programa de acordo com as especificações dadas para transferir ficheiros entre dois computadores, ligados por uma porta de serie. Este relatório vai servir para apresentar os detalhes e especificações do programa, de acordo com a seguinte estrutura:

1. **Arquitetura**

   Blocos funcionais e interfaces.

2. **Estrutura do código**

   Principais funções, funcionamente destas e a sua relação com a arquitetura.

   2.1. **Protocolo de aplicação**

   2.2. **Protocolo de ligação lógica**

   2.3. **State machine**

3. **Casos de uso principais**

   Identificação; Sequências de chamada de funções.

4. **Validação**

   Descrição dos testes efetuados.

5. **Eficiência do protocolo de ligação de dados**

   Caraterização estatística da eficiência do protocolo, efetuada recorrendo a medidas sobre o código desenvolvido. A caraterização teórica de um protocolo Stop&Wait, que deverá ser empregue como termo de comparação, encontra-se descrita nos slides de Ligação Lógica das aulas teóricas.

6. **Conclusões**

   Síntese da informação apresentada nas secções anteriores; Reflexão sobre os objetivos de aprendizagem alcançados e principais dificuldades.

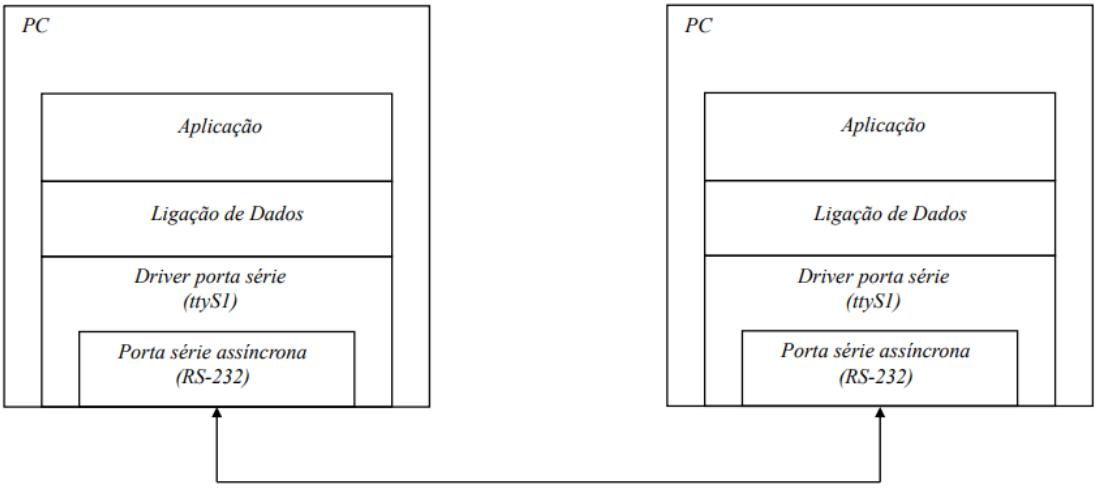
1.Arquitetura

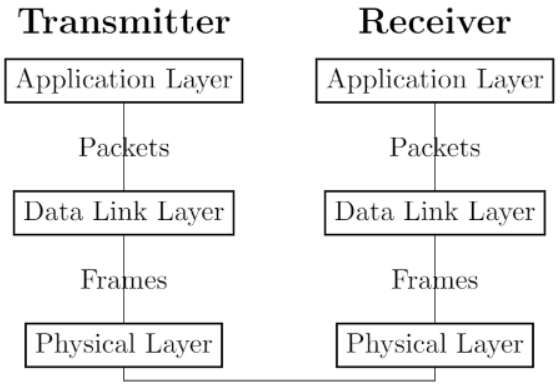
Para este programa, o objetivo da arquitetura escolhida foi criar várias camadas, sendo que cada uma consegue ter independência em relação as outras.

Na camada Data Link não existe processamento dos packets que recebe da camada Application.

A camada Application serve para aceder à camada Data Link mas não tem acesso aos seus comportamentos internos.

Esta separação entre camadas permite que apenas informação relevante seja passada a respetiva camada.





2.Estrutura do código

2.1. Application

llopen

Int llopen(char *port, int flag);

- Pede à camada Data Link para estabelecer ligação (transmitter ou receiver, dependendo da flag)

- Retorna file descriptor da porta de serie em caso de sucesso, -1 em caso de erro.

llwrite

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

-Pede à camada Data Link para transmitir a informação passada em buffer de tamanho length para file descriptor fd

-Retorna o número de bytes que enviou

llread

unsigned char* llread(int fd, int* size);

-Pede à camada Data link para ler a informação que foi passada para o file descriptor fd.

Retorna a mensagem lida e coloca o numero de bytes desta na variável size.

llclose

int llclose(int fd, int flag)

-Pede à camada Data Link para terminar a ligação.

-Fecha file descriptor da porta de serie

-Retorna 1 em caso de sucesso, -1 em caso de erro.

2.2. Data Link

byte_stuffing

unsigned char * byte_stuffing(unsigned char *packet, int *length);

-Faz stuffing da mensagem que recebe em packet de tamanho length

-Altera length de acordo com os bytes adicionados no stuffing

-Retorna packet depois do stuffing

byte_destuffing

unsigned char* byte_destuffing(unsigned char *packet, int *length);

-Faz destuffing da mensagem que recebe em packet de tamanho length

-Altera length de acordo com os bytes retirados no destuffing

-Retorna packet depois do destuffing

i_frame_write

int i_frame_write(int fd, char a, int length, unsigned char *data);

-Coloca dados data de tamanho length na frame.

- Escreve a frame no fd e liga o alarme.

- Espera até receber a mensagem de confirmação do tipo RR ou que o alarme toque. Se o alarme tocar reenvia os dados, se receber RR desliga o alarme e termina.

-Retorna o número de bytes escritos ou -1 em caso de erro.

read_i_frame

unsigned char* read_i_frame(int fd, int *size_read);

-Lê informação escrita no fd

-Tem maquina de estados implementada na função para confirmar primeiros bytes até receber inicio da information frame.

-No caso de receber os dados todos escreve no fd a mensagem de confirmação RR

-Atualiza a variável size_read para o número de bytes lidos

-Retorna os dados lidos

iniciate_connection

int iniciate_connection(char *port, int connection);

- Abre a porta de serie, guarda as configurações a atualiza-as

TRANSMITTER

-Envia mensagem SET e liga alarme

- Espera até receber a mensagem de confirmação do tipo UA ou que o alarme toque. Se o alarme tocar reenvia os dados, se receber UA desliga o alarme e termina.

RECEIVER

-Utiliza a maquina de estados para confirmar a receção da mensagem SET.

-Se receber SET envia mensagem de confirmação UA e termina.

-Ambos retornam fd em caso de sucesso e -1 em caso de erro.

terminate_connection

int terminate_connection(int *fd, int connection);

TRANSMITTER

-Envia mensagem DISC e liga alarme

- Espera até receber a mensagem de confirmação do tipo UA ou que o alarme toque. Se o alarme tocar reenvia os dados, se receber UA desliga o alarme, envia mensagem UA e termina.

RECEIVER

-Utiliza a máquina de estados para confirmar a receção da mensagem DISC.

-Se receber DISC envia mensagem de confirmação DISC.

-Utiliza a máquina de estados para confirmar a receção da mensagem UA, se receber termina.

-**Ambos** repõem as definições iniciais da porta de serie e fecham file descriptor da porta de serie

2.3. State machine

int state_machine(unsigned char *buf, int *state)

-Recebe o estado atual e um buffer com os últimos bytes lidos.

-Compara o byte correspondente ao estado atual e atualiza o estado com base no funcionamento da máquina de estados

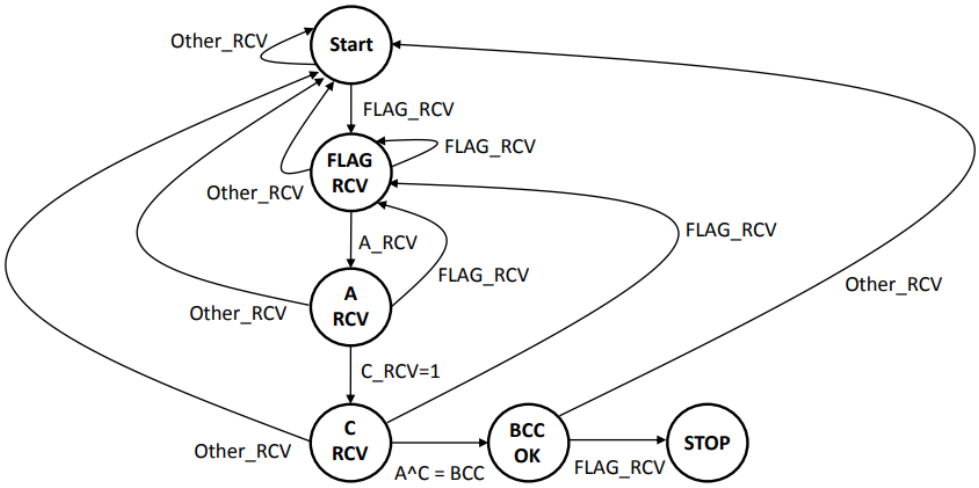


Figura 1: Máquina de estados para mensagem SET

3.Casos de uso principais

O nosso programa tem um número de argumentos variável, dependendo se é receiver ou transmitter, sendo eles:

TRANSMITTER:

- **Porta de serie :** /dev/ttySx
- **Ficheiro:** Path o ficheiro que o utilizador pretende transmitir

Receiver:

- **Porta de serie :** /dev/ttySx

Transmitter:

Exemplo de uso:

./main /dev/ttyS0 pinguim.gif

Sequência de chamada de funçoes:

Main

Application layer

Data link layer

1. llopen

    1.1.iniciate_connection

        1.1.1.su_frame_write

        1.1.2. state_machine

2. process_pic – Função para preparar envio de ficheiro
3. llwrite

    3.1.i_frame_write

        3.1.1 byte_stuffing

4. llwrite

    4.1.i_frame_write

        4.1.1.byte_stuffing

5. llwrite

    5.1. i_frame_write

        5.1.1byte_stuffing

6. llclose
    6.1. terminate_connection
        6.1.1.su_frame_write
        6.1. state_machine
        6.1.3.su_frame_write

Receiver:

Exemplo de uso:

./main /dev/ttyS0

Sequência de chamada de funçoes:

Main

Application layer

Data link layer

1. llopen
    1.1.iniciate_connection
        1.1.1.su_frame_write
2. llread
    2.1.read_i_frame
        2.1.1 byte_destuffing
3. llread
    3.1.read_i_frame
        3.1.1 byte_destuffing
4. llread
    4.1.read_i_frame
        4.1.1 byte_destuffing
7. llclose
    6.1.terminate_connection
        6.1.1.state_machine
        6.1.2.su_frame_write
        6.1.3.state_machine

4. Validação

Para testar o nosso programa foram feitos os seguintes testes:

- Teste de envio de ficheiro com socat
- Teste de envio de ficheiro de tamanho maior com socat
- Teste de envio de ficheiro com introdução de erros com socat
- Teste de envio de ficheiro normal e de tamanho maior com porta de serie
- Teste de envio de ficheiro com introdução de erros em porta de serie
- Teste de envio de ficheiro a desligar e ligar porta de serie durante a transmissão de dados

5. Eficiência do protocolo de ligação

No caso de não existir falhas de transmissão, um ficheiro de 10968 Bytes é enviado a cerca de 5.9 segundos para o transmitter e 6.4 segundos para o receiver (o transmitter foi corrido algumas decimas de segundo depois para dar tempo ao receiver de fazer setup das estruturas necessárias). Para um ficheiro com 29415 Bytes, ainda sem falhas de transmissão, este demorou cerca de 10.7 segundos para o transmitter e 11.2 para o receiver. O baudrate para estes testes foi de 38400.

Para o caso do baudrate a 19200 o ficheiro de 10968 Bytes demorou 8.8 segundos para o transmitter e 10.1 para o receiver.

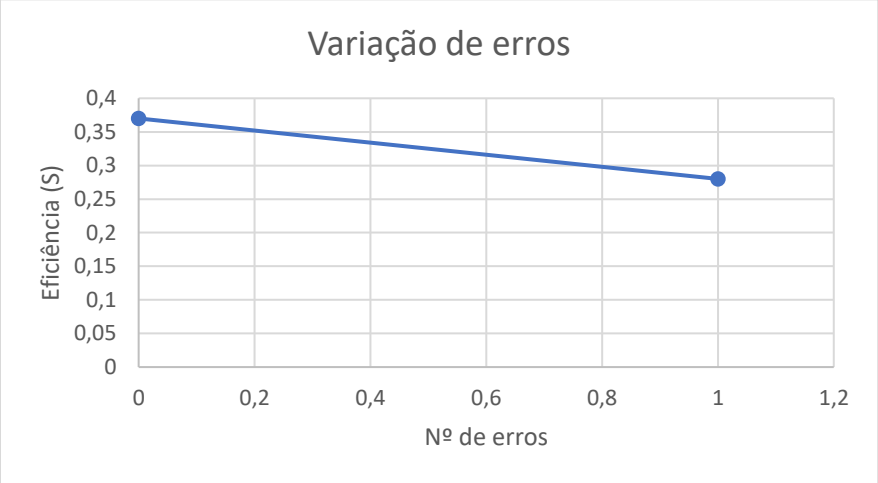
O tempo de transmissão quando existe deteção de erros aumenta significativamente devido ao facto que o ficheiro inteiro é retransmitido. O ficheiro de 10968 Bytes passou a demorar 16.9 segundos no caso do transmitter quando foi introduzido um erro que levou a uma repetição da trama e o ficheiro com 29415 Bytes nas mesmas condições aumentou para 21.7 segundos. (nota: o transmitter tem um timeout de 10seg, ou seja, se o receiver não confirmar a receção da trama o transmitter espera 10 segundos antes de a reenviar.

R = 14152 bit/s

S = R/C ⇔ S = 14152 / 38400 = 0.37 (ficheiro mais pequeno, sem falha de transmissão)

R = 10844 bit/s

S = R/C ⇔S = 10844 / 38400 = 0.28 (ficheiro maior, uma retransmissão)



6. Conclusões

Para concluir o que foi dito acima, consideramos que o nosso projeto seguiu as indicações de desenvolvimento dadas (com exceção da divisão do ficheiro por tramas) e que apresentou um bom resultado em todos os testes.

Achamos que este projeto foi uma boa maneira de fortalecer o nosso conhecimento de redes e compreender algumas das formas de como é feita a analise de erros e reenvio de dados.

Um dos maiores obstáculos no desenvolvimento deste projeto foi o facto do programa correr de forma bastante diferente no socat e na porta de serie. Isto deveu-se a vários fatores como compiladores dos PCs diferentes, não reposição das definições da porta de serie no final do programa, entre outras. Ainda assim, conseguimos ultrapassar todos estes problemas e no final conseguimos que o programa passasse em todos os testes a ele apresentados, o que bastante recompensador.

main

```
#include "application.h"


unsigned char * process_pic(char* path, int* size){
   FILE *f = fopen(path, "r");
   struct stat st;
   if (stat(path, &st) == 0)
      *size = st.st_size;
   //*size = ftell(f); //qts bytes tem o ficheiro
   unsigned char *data = (unsigned char*)malloc(*size+4);
   unsigned char *buffer = (unsigned char *)malloc(*size);



   //fseek(f, 0, SEEK_SET);
   fread(buffer, 1, *size, f);
   fclose(f);

            data[0] = C_REJ;      // C
            data[1] = 0; // N
            data[2] = *size / 255;              // L2
            data[3] = *size % 255; // L1

            for (int i = 0; i < *size; i++) {
                     data[i+4] = buffer[i];
            }
   printf("main- total file bytes = %d\n", *size);

            return data;
}



int main(int argc, char** argv)
{
   fflush(stdout);
```

```
char *port;
char *img;
unsigned char control[100];

int fd, res, length = 5;
char buf[255];



if(argc < 2 || argc > 3){
  printf("Invalid Usage:\tInvalid number of arguments0");
  exit(1);
}

   if(strcmp(MODEMDEVICE_0, argv[1]) == 0 || strcmp(MODEMDEVICE_1, argv[1]) == 0 ||
strcmp(SOCAT_MODEMDEVICE_10, argv[1]) == 0 || strcmp(SOCAT_MODEMDEVICE_11, argv[1]) == 0 ){
      port = argv[1];
   }
if(argc == 2)
   img = NULL;
else if(argc == 3){
            int accessableImg = access(argv[2], F_OK);
            if (accessableImg == 0) {
                        img = argv[2];
            }
   else {
                        printf("Invalid Usage:\tInvalid arguments1\n");
                        exit(1);
            }
}

//img = NULL;
if(img == NULL){ // Open comunications for receiver
printf("receiver\n");
   if((fd = llopen(port, RECEIVER))){
      printf("after ua received\n");
```

```
    unsigned char *msg = NULL;
    msg = (unsigned char *)malloc(800000);
    unsigned char *buffer;
    unsigned char *msg_start;
    unsigned char *msg_end;
    printf("receiver reading first control packet\n");
    int control_size = 0;
    msg_start = llread(fd, &control_size);

    printf("receiver reading first data packet\n");
    if(msg_start[0] == REJ){
       printf("Received start\n");
       }
       int size = 0;
       msg = llread(fd, &size);
       printf("receiver reading last control packet\n");
       msg_end = llread(fd, &control_size);

    llclose(fd, RECEIVER);
    //int number_frames = sizeof(*msg) / (MAX_SIZE - 6);
    printf("SIZE OF READ IS %d\n\n\n", size);
    FILE *f = fopen("return_file.gif", "w");
    for(int i = 4; i< size; i++){
       fputc(msg[i], f);
    }
    // int written = fwrite(msg, sizeof(unsigned char), sizeof(msg), f);
    // if (written == 0) {
    //     printf("Error during writing to file !");
    // }
    fclose(f);

  }
}
else if(fd = llopen(port, TRANSMITTER)){
  printf("transmitter");
```

```
    unsigned char * buffer = process_pic(img, &length);
    if(length <= 5){ // demand at least a byte, the rest is the header
       printf("Error processing image\n");
       exit(1);
    }
    printf(" length in main is %d\n", length);
    //CONTROL packet
    control[0] = 2; // C_BEGIN
    control[1] = 0; // T_FILESIZE
    control[2] = 1;
    control[3] = length;
    int tempLength = length;
    int l1 = (length / 255) + 1;
    int i;
    for(i = 0; i < l1; i++){
    control[4+i] =  (tempLength >> 8*(i+1) & 0xff);   //so o ultimo byte do numero
  }
    printf("starting transmission of CONTROL packet\n");
    printf("fd = %d\n", fd);
    if(llwrite(fd, control, i + 4)){    //escreve control packet
       printf("starting transmission of I packet\n");
       if(llwrite(fd, buffer, length)){
          control[0] = 3;
          printf("starting transmission of last CONTROL packet\n");
          llwrite(fd, control, 4);
       }
    }

    llclose(fd, TRANSMITTER);
  }



        return 0;
}
```

Application

```
#include "application.h"



int llopen(char *port,int flag) {


   // appL->fileDescriptor = iniciate_connection(port, flag);

   // appL->status = flag;

   return iniciate_connection(port, flag);

}



int llwrite(int fd, char *buffer, int length){

   //1º preparar buffer

   //fazer malloc

   return i_frame_write(fd, A_E, length, buffer);

}


unsigned char* llread(int fd, int *size){

   return read_i_frame(fd, size);

}


int llclose(int fd, int flag){

   return terminate_connection(&fd, flag);

}
```

link

```
#include "link.h"


struct termios oldtio, newtio;


int timerfd = 0;


unsigned int sequenceNumber = 0;   /*Número de sequência da trama: 0, 1*/
unsigned int timeout = 10;         /*Valor do temporizador: 1 s*/
unsigned int numTransmissions = 3; /*Número de tentativas em caso de falha*/
int size_of_read = 0;


int alarmFlag = 0;
int alarmCount = 0;


void sig_handler(int signum){


  alarmFlag = 1;
  alarmCount++;
  fcntl(timerfd, F_SETFL, O_NONBLOCK);
}


void change_sequenceNumber(){
    if(sequenceNumber)
        sequenceNumber = 0;
    else sequenceNumber = 1;
}


unsigned char * byte_stuffing(unsigned char *packet, int *length){
    unsigned char *stuffed_packet = NULL;
    stuffed_packet = (unsigned char *)malloc( *length * 2);
    int j = 0;
    for(int i = 0; i < *length; i++){
        if(packet[i] == FLAG){
            stuffed_packet[j] = 0x7d;
```

```
            stuffed_packet[++j] = 0x5e;
        }
        else if(packet[i] == ESCAPE_OCTET){
            stuffed_packet[j] = 0x7d;
            stuffed_packet[++j] = 0x5d;
        }
        else stuffed_packet[j] = packet[i];

        j++;
    //printf("data = %x, stuffed data = %x\n",  packet[i], stuffed_packet[i] );
    }
    *length = j;
    return stuffed_packet;

}

//////////////////////////////////////////////////////
unsigned char * byte_destuffing(unsigned char *packet, int *length){
    //so do data no i packet
    unsigned char *destuffed_packet = NULL;
    destuffed_packet = (unsigned char *)malloc(*length * 2);
    int j = 0;
    for (int i = 0; i < *length; i++) {
        if (packet[i] == ESCAPE_OCTET)
            destuffed_packet[j] = packet[++i] ^ 0x20;
        else
            destuffed_packet[j] = packet[i];

        j++;
    }
    *length = j;
    return destuffed_packet;

}
```

```
int su_frame_write(int fd, char a, char c) {
    unsigned char buf[5];

    buf[0] = FLAG;
    buf[1] = a;
    buf[2] = c;
    buf[3] = a ^ c;
    buf[4] = FLAG;

    return write(fd, buf, 5);
}

int i_frame_write(int fd, char a, int length, unsigned char *data) {
    (void) signal (SIGALRM, sig_handler);
    //bff2 before stuffing
    timerfd = fd;
    alarmFlag = FALSE;
    alarmCount = 0;
    unsigned char bcc2 = data[0];
    for(int i = 1; i < length; i++){
        bcc2 ^= data[i];
    }
    unsigned char *framed_data = (unsigned char*)malloc(sizeof(unsigned char) * (length + 7));
    //byte stuffing
    unsigned char *stuffed_data = byte_stuffing(data, &length);
    //put stuffed data into frame
    framed_data[0] = FLAG;
    framed_data[1] = a;
    framed_data[2] = sequenceNumber;
    framed_data[3] = a^ sequenceNumber;
    int j = 4;

    for(int i = 0; i < length; i++){
        framed_data[j] = stuffed_data[i];       //começa no buf[2]
        //printf("   bcc2: %x   ", stuffed_data[j]);
```

```
   j++;
}
framed_data[j+1] = bcc2;
framed_data[j+2] = FLAG;

//write frame
int frame_length = j+2+1; //+1 bcd 0 index
int written_length = 0;
int state = START;
alarmCount = 0;
unsigned char buf[5];
int flag = FALSE;
////////////////////////////////////////
//fcntl(fd, F_SETFL, O_NONBLOCK);
////////////////////////////////////////
printf("frame length = %d", frame_length);
do{
      fcntl(fd, F_SETFL, 0);
      alarmFlag = FALSE;
      alarm(timeout);
      if( (written_length = write(fd, framed_data, frame_length)) < 0){
         printf("written_length = %d ", written_length);
         perror("i frame failed\n");
      }
      flag = FALSE;
      while(!alarmFlag && state != BCC_OK ){
         if(read(fd, &buf[state], 1) <= 0)
            sleep(1);
         state_machine(buf, &state);
      }
      if(state == BCC_OK){
         alarm(0);
         break;
      }
```

```
        }
        while(alarmFlag && (alarmCount <   numTransmissions));

        if(alarmCount ==  numTransmissions){
            perror("Error sending i packet, too many attempts\n");
            return -1;
        }
        else{
            printf("RR from i message recieved\n");
        }

      //sequenceNumber =  sequenceNumber ^ 1;
    return written_length;

}

unsigned char* read_i_frame(int fd, int *size_read){
    unsigned char *temp = NULL;
    int state = START;
    int data_size = 0;
    unsigned char buffer;
    //unsigned char *data_received = (unsigned char*)malloc(data_size);
    unsigned char data_received[100000];
    int all_data_received = FALSE;
    int data_couter = 0;
    int testCount = 0;
    while(!all_data_received){
        if(read(fd, &buffer, 1) < 0)
            perror("failed to read i frame\n");
        else{
            //ver estado
            switch(state){

                case START:
```

```
    //printf("buffer: %x state :start\n", buffer);
    //printf("data counter = %d\n", data_couter);
    data_couter++;
    if(buffer == FLAG)
      state = FLAG_RCV;
    break;
  case FLAG_RCV:
    //printf("buffer: %x state :flag_rcv\n", buffer);
    // printf("data counter = %d\n", data_couter);
    data_couter++;
    if(buffer == 0x01 || buffer == 0x03)
      state = A_RCV;
    else if(buffer == 0x7e)
      state = FLAG_RCV;
    else state = START;
    break;
  case A_RCV:
    //printf("buffer: %x state :a_rcv\n", buffer);
    // printf("data counter = %d\n", data_couter);
    data_couter++;
    //printf("sequenceNumber = %d\n", sequenceNumber);
    if(buffer ==  sequenceNumber)
      state = C_RCV;
    else if(buffer == 0x7e)
      state = FLAG_RCV;
    else
      state = START;
    break;
  case C_RCV:
    //printf("buffer: %x state :c_rcv\n", buffer);
    // printf("data counter = %d\n", data_couter);
    data_couter++;
    if(buffer == 0x01 ^ sequenceNumber)
      state = DATA;
    else if(buffer == 0x7e)
```

```
            state = FLAG_RCV;
        else
            state = START;
        break;
    case DATA:
        data_couter++;
        if(buffer == FLAG){     //finished transmitting data
            printf("received final flag!\n");

            temp = byte_destuffing(data_received, &data_size);  //data size starts in 0

            unsigned char post_transmission_bcc2 = data_received[0];
            for(int i = 1; i < data_size - 2; i++){
                post_transmission_bcc2 ^= temp[i];
                //printf("   bcc2: %x   ", data_received[i]);
            }
            unsigned char bcc2 = temp[data_size-1];

            if(bcc2 == post_transmission_bcc2){
                printf("data packet received!\n");
                all_data_received = TRUE;
            }
            else{
                perror("BCC2 dont match in llread\n");
                printf("Will now try to re-send\n");
                data_size = 0;
                all_data_received = FALSE;
                data_couter = 0;
                state = START;
            }
        }
        else{
            data_size++;
            data_received[data_size - 1] = buffer;
            printf("buffer = %x", buffer);
```

```
            }
            break;
        }
      }
    }

    printf("receiver received packet!\n");
    unsigned char *final_array = (unsigned char*)malloc(sizeof(data_received));
    if(data_received[0])
       su_frame_write(fd, A_R, C_RR);

    *size_read = data_size;
    printf("data size is %d\n", data_size);
    return temp;
}

int iniciate_connection(char *port, int connection)
{


    int fd,c, res;
    char buf[5];
    alarmCount = 0;
    alarmFlag = FALSE;
    int i, sum = 0, speed = 0;

    ////////////////////////////////////////////////////////////
    /*
    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.
    */

    fd = open(port, O_RDWR | O_NOCTTY );
    if (fd <0) {perror(port); exit(-1); }
```

```
   sleep(1);
   if ( tcgetattr(fd,&oldtio) == -1) { /* save current port settings */
   perror("tcgetattr");
   exit(-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] = 0; /* inter-character timer unused */
   newtio.c_cc[VMIN] = 1; /* blocking read until 5 chars received */




   /*
   VTIME e VMIN devem ser alterados de forma a proteger com um temporizador a
   leitura do(s) próximo(s) caracter(es)
   */



   tcflush(fd, TCIOFLUSH);

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

```
(void) signal(SIGALRM, sig_handler);    //Register signal handler


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



int state = START;
if(connection == TRANSMITTER){
   int flag = TRUE;

   //re-send message if no confirmation
   do{
      if(su_frame_write(fd, A_E, C_SET) < 0){
         perror("set message failed\n");
      }
      alarm(  timeout);
      flag = FALSE;
      while(!alarmFlag && state != BCC_OK ){
         read(fd, &buf[state], 1);
         state_machine(buf, &state);
      }
      if(state == BCC_OK){
         alarm(0);
         break;
      }



   }
   while(alarmFlag && alarmCount <   numTransmissions);

   if(alarmCount ==   numTransmissions){
      perror("Error establishing connection, too many attempts\n");
      return -1;
   }
```

```
        else(printf("UA from SET message recieved\n"));

    }

    else if(connection == RECEIVER){
    printf("entrou no receiver\n");
        while(state != BCC_OK){
            if (read(fd, &buf[state], 1) < 0) { // Receive SET message
                perror("Failed to read SET message.");
            } else {
                state_machine(buf, &state);
            }
        }
        printf("establish connection - SET recieved!\n");
        su_frame_write(fd, A_E, C_UA);

    }
    else {
        printf("invalid type of connection!\n");
        return -1;
    }
    return fd;

}

int terminate_connection(int *fd, int connection)
{
    char buf[5];
    alarmCount = 0;
    alarmFlag = FALSE;
    int state = START;
    if(connection == TRANSMITTER){
        int flag = TRUE;
        printf("terminate connection(transmitter) starting\n");
        //re-send message if no confirmation
```

```
    //send and check if recieved DISC msg
    do{
        if(su_frame_write(*fd, A_E, C_DISC) < 0){
            sleep(3);
            perror("disc message failed\n");
        }
        alarm(  timeout);
        flag = FALSE;
        while(!alarmFlag && state != BCC_OK ){
            read(*fd, &buf[state], 1);
            state_machine(buf, &state);
        }
        if(state == BCC_OK){
            alarm(0);
            break;
        }


    }
    while(alarmFlag && alarmCount <   numTransmissions);

    if(alarmCount ==   numTransmissions){
        perror("Error establishing connection, too many attempts\n");
        return -1;
    }
    else{
        printf("DISC from DISC message recieved\n");
        su_frame_write(*fd, A_E, C_UA);
    }
}

else if(connection == RECEIVER){
    printf("terminate connection(receiver) starting\n");
    while(state != BCC_OK){
        if (read(*fd, &buf[state], 1) < 0) { // Receive SET message
```

```
        sleep(10);
            perror("Failed to read DISC message.");
        } else {
            state_machine(buf, &state);
        }
    }
    printf("DISC recieved!\n");

    if(su_frame_write(*fd, A_E, C_DISC) < 0){    //write
        perror("ua message failed\n");
        return -1;
    }

    state = START;
    while(state != BCC_OK){
        if (read(*fd, &buf[state], 1) < 0) { // Receive UA message
            perror("Failed to read SET message.");
        } else {
            state_machine(buf, &state);
        }
    }
    printf("UA recieved!\n");

}
else {
    printf("invalid type of connection!\n");
    return -1;
}

sleep(1);
if (tcsetattr(*fd, TCSANOW, &oldtio) == -1) {
                perror("tcsetattr");
                exit(-1);
            }
```

```
            close(*fd);

    fflush(stdout);

    return 1;

}
```