Protocolo de Ligação de Dados

Sara Marinha up201906805 Afonso Monteiro up201907284

Sumário

O trabalho foi realizado no âmbito da unidade curricular *Redes de Computadores* e tem como propósito o desenvolvimento de uma aplicação (em **C**) do protocolo de ligação de dados entre dois computadores com sistema operativo Linux e ligados por um cabo de série RS-232.

O trabalho foi realizado com sucesso, pois cumpriu-se o objetivo de desenvolver a aplicação requerida, capaz de transmitir um ficheiro entre 2 computadores, mesmo nos casos com erros ou de interrupção da transmissão.

1. Introdução

O trabalho consiste na transferência viável de um ficheiro, neste caso, gif, entre dois computadores ligados por um cabo de série com retransmissão de *frames* em caso de erro ou de perda de informação. Esta transferência segue o protocolo de ligação de dados, ou seja, implementa:

- sincronismo de *frames* (sendo estas delimitadas por *frames* 0x7E, denominada de **FLAG**).
- estabelecimento e terminação da ligação (respetivamente **llopen** e **llclose**)
- numeração de *frames* (terceiro byte da *frame*)
- confirmação positiva (**RR**) após a receção de um *frame* sem erros e na sequência correta
- controlo de erros (*Stop-and-Wait*)
    - temporizadores (*time-out*) - retransmissão é decidida pelo emissor
    - confirmação negativa (frames fora da sequência esperada ou com erro bcc no *header*) - retransmissão é pedida do lado do recetor
    - retransmissões podem originar duplicados, pelo que são ignorados
- controlo do fluxo O relatório visa descrever a implementação e funcionamento da aplicação, seguindo a componente teórica abordada. Respeitará a seguinte estrutura:
- **Arquitetura** , blocos funcionais e interfaces.
- **Estrutura do código** , APIs, principais estruturas de dados, principais funções e a sua relação com a arquitetura.
- **Casos de uso principais** , identificação; sequências de chamada de funções.
- **Protocolo de ligação lógica** , identificação dos principais aspetos funcionais; descrição da estratégia de implementação destes aspetos com apresentação de extratos de código.
- **Protocolo de aplicação** , identificação dos principais aspetos funcionais; descrição da estratégia de implementação destes aspetos com apresentação de extratos de código.
- **Validação**, descrição dos testes efetuados com apresentação quantificada dos resultados, se possível.
- **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.
- **Conclusões**, síntese da informação apresentada nas secções anteriores; reflexão sobre os objetivos de aprendizagem alcançados.

2. Arquitetura

blocos funcionais e interfaces. A arquitetura implementada segue o princípio de independência entre camadas, sendo estas a camada de ligação de dados e a da aplicação.

Camada da ligação de dados

- Abertura
- Calcula o bcc do pacote de dados
- Adiciona/remove o *frame header* e o *frame trailer*
- Stuffing e destuffing das *frames*
- Verifica se a trama tem erros
- Envia as frames de supervisão (S) e não numeradas (U)
- Leitura e escrita na porta série
- Retransmissão em caso de erro
- Terminação

Camada da aplicação

- Envio e receção de ficheiros
- Processamento dos pacotes de controlo e dados

3. Estrutura do código

llopen

- **Emissor** Abre a porta série , altera as suas configurações, envia uma frame de supervisão (**SET**) e espera receber a confirmação (**UA**).
- **Recetor** Abre a porta série , fica à espera pelo **SET** e envia o **UA**, no caso de sucesso.

llwrite

- **Emissor** Envia a *frame* passada pela aplicação adicionando-lhe um *header* e um *trailer*, aplica-lhe o mecanismo de byte stuffing e espera pela confirmação. No caso desta ser negativa ou de passarem 3 segundos retransmite a *frame*.

llread

- **Recetor** Ao receber a *frame* faz destuffing, verifica se a *frame* não tem erros tanto no BCC1 (xor entre A e C - ver imagem 1) como no BCC2 (depois de remover o header, faz o xor nos dados menos os dois ultimos, que correspondem ao BCC2 da *frame* e a FLAG final). Consoante o estado da *frame* manda a confirmação adequada.

llclose

- **Emissor** quando termina a transmissão envia um comando DISC e espera receber o DISC do recetor. Assim que isso aconteça, envia o UA. Por fim, repõe as configurações originais e fecha a porta série.
- **Recetor** quando recebe o DISC do emissor envia o seu comando DISC e espera receber o *UA*. Confirmada a receção, fecha a porta série.

4. Casos de uso principais

O trabalho realizado inclui o caso de uso:

1. Configurar a ligação;
2. Estabelecer ligação;
3. Envio de dados pelo transmissor;
4. Receção de dados pelo recetor e escrita de ficheiros no output;
5. Fecho dos ficheiros abertos;
6. Reversão das configurações de ligação;
7. Fecho de ligação.

5. Protocolo de ligação lógica

O protocolo de ligação lógico incorporado responsabiliza-se por:

- Fazer a ligação da porta série (guardando os parâmetros iniciais de ligação para depois substituir?)
- Transferir os dados pelas mesmas, fazendo o *stuffing* e *destuffing*.
- Fazer a verificação de erros e o controlo do fluxo.

Para uma implementação correta, recorreu-se às seguintes funções:

llopen

Abre a porta série e altera as suas configurações.

Emissor

No lado do emissor, é responsável por criar uma *frame* não numerada com o campo de controlo SET. De seguida, espera receber uma *frame* não numerada com o campo de controlo UA e caso não a receba dentro de 3 segundos reenvia-a, no máximo, 3 vezes. Após retransmitir 3 vezes retorna um erro. Retorna o *file descriptor* no caso de sucesso.

```
int llopen(int porta, unsigned char flag)

  char path[12] = "/dev/ttyS";
  char port = porta + '0';
  memcpy(path + strlen(path) , &port,1);

  int fd = open(path, O_RDWR | O_NOCTTY );

  if (fd <0) {perror(path); return -1; }
  setSettings(fd);

  unsigned char ua_expected[5];
  unsigned char set[5];

  createFrame(ua_expected, CMD, UA);
  createFrame(set, CMD, SET);

  sendFrame(fd, set, 5);
```

```
  unsigned char res = checkingOneResponse(fd, set, CMD, UA);
  if(res == 1) return -1;
  }
  return fd;
}
```

Recetor

No lado do recetor, espera até receber a mensagem SET e no caso de a receber envia *frame* não numerada com o campo de controlo UA. Retorna o *file descriptor*.

```
int llopen(int porta, unsigned char flag)
{
  char path[12] = "/dev/ttyS";
  char port = porta + '0';
  unsigned char ua[5];

  memcpy(path + strlen(path) , &port,1);

  int fd = open(path, O_RDWR | O_NOCTTY );

  if (fd <0) {perror(path); exit(-1);}

  setSettings(fd);

  createFrame(ua, ANS, UA);
  checkingFrame(fd, ANS, SET);

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

  return fd;
}
```

llread

Esta função trata da leitura das *frames* recebidas, sendo elas de controlo ou de dados. Faz o destuffing, verificação do BCC1 e BCC2, remove o header e o trailer da *frame*. Retorna o número de bytes restantes da *frame* , no caso de não existirem erros. No caso de ser uma trama repetida, retorna 0 para a aplicação não escrever novamente os dados no ficheiro de output. No caso de erro :

- no BCC1 ou no BCC2 , envia uma resposta REJ com o n(r) da trama que quer que seja enviada.
- no número de sequência, ou seja, ser uma trama repetida , envia uma resposta RR de sucesso e com n(r) desejada para a *frame* seguinte. Se nenhum destes casos acontecer, envia uma resposta RR de sucesso e com n(r) desejada para a *frame* seguinte e atualiza o n(r) (n(r) seguinte é calculado pelo incremento de uma unidade módulo 2).

```
int llread(int fd, unsigned char * buffer)
{
  int acc = 2, i=0, res;
  int stop1 = 0, starting_frame = 0;
  memset(infoFrame, 0, 256);
  memset(buffer, 0, 256);

  while(!stop1)
  {
    res = read(fd, &buffer[i],1);
    if(buffer[i] == FLAG && !starting_frame)
    {
      i = 0;
      buffer[0] = FLAG;

      starting_frame = 1;
    }
    else if(buffer[i] == FLAG && starting_frame && buffer[i-1] != FLAG)
    {
      stop1 = 1;
    }
    else if(buffer[i] == FLAG && buffer[i-1] == FLAG)
    {
      i--;
    }
    i++;
  }

  int numBytesAfterDestuffing = byteDestuffing(buffer, i);
  unsigned char nsSender = infoFrame[2];
  unsigned char response[5];

  if(!checkBCC1(3))
  {
    printf("Erro no header\n");
    createFrame(response, ANS, REJ | (nreceiver << 7));
    write(fd, response,5);
    tcflush(fd, TCIOFLUSH);
    return -1;
  }

  removeFrameHeader(numBytesAfterDestuffing-4);

  printMsg(infoFrame, numBytesAfterDestuffing -4);
  if (((nsSender + 1)%2) != nreceiver)
  {
    createFrame(response, ANS, RR | (nreceiver << 7));
    write(fd, response,5);
    return 0;
  }

  else if((((nsSender + 1)%2) == nreceiver) && checkBCC2(numBytesAfterDestuffing-
6)) //significa que é uma trama diferente
```

```
  {
    createFrame(response, ANS, RR | (nreceiver << 7));
    write(fd, response,5);
    nreceiver = (nreceiver + 1) % 2;

  }
  else if((((nsSender + 1)%2) == nreceiver) && !checkBCC2(numBytesAfterDestuffing-
6))
  {
    createFrame(response, ANS, REJ | (nreceiver << 7));
    write(fd, response,5);
    tcflush(fd, TCIOFLUSH);
    return -1;
  } else
  {
    createFrame(response, ANS, RR | (nreceiver << 7));
    write(fd, response,5);
  }

  removeFrameTrailer(numBytesAfterDestuffing - 3);

  return numBytesAfterDestuffing - 5;
}
```

llwrite

Esta função adiciona o *header* e o *trailer* da *frame*, calcula o bcc do buffer passado como argumento, faz o stuffing da *frame*, envia para o recetor , espera para resposta e age conforme esta (RR - continua e REJ - retransmite). Caso passem 3 segundos sem resposta, volta a enviar e espera novamente 3 segundos, faz isto no máximo, 3 vezes.

```
int llwrite(int fd, unsigned char * buffer, int length)
{
  unsigned char frame [length + 6];

  addFrameHeader(frame);
  memcpy(frame + 4, buffer, length);

  unsigned bcc = calculateBCC(buffer, length);
  addFrameTrailer(frame, bcc,  4 + length);

  int nbytes = byteStuffing(frame, length + 6);

  info_frame_size = nbytes;
  int res = sendFrame(fd, infoFrame, info_frame_size);

  unsigned char response = checkingFourResponse(fd, CMD, RR, RR8, REJ, REJ8);
  if(response == 1) return -1;

  alarm(0);
```

```
  return res;
}
```

llclose

A política de retransmissão definidas nas funções anteriores também se aplica nesta função.

Emissor

Esta função envia: 1. Uma mensagem DISC 2. Espera receber a mensagem DISC do recetor 3. Quando o passo 2 acontece, envia a mensagem UA

```
int llclose(int fd)
{
  unsigned char disc[5];
  unsigned char ua[5];

  createFrame(disc, CMD, DISC);
  createFrame(ua, ANS, UA);

  sendFrame(fd, disc, 5);

  checkingOneResponse(fd, disc, ANS, DISC);
  sendFrame(fd, ua, 5);

  return 0;
}
```

Recetor

Esta função envia: 1. Espera receber a mensagem DISC do emissor 2. Quando o passo 1 acontece, envia a mensagem DISC 3. Espera receber a mensagem UA do emissor

```
int llclose(int fd)
{
  unsigned char disc[5];

  createFrame(disc, CMD, DISC);
  checkingFrame(fd, ANS ,DISC);
  write(fd, disc, 5);
  checkingFrame(fd, CMD, UA);

  return 0;
}
```

6. Protocolo de aplicação

identificação dos principais aspetos funcionais; descrição da estratégia de implementação destes aspetos com apresentação de extratos de código.

Emissor

Criação do pacote de controlo

```
int createControlPacket(unsigned char * packet, unsigned char control_field,
unsigned char type, unsigned char length,  unsigned char * value)
{
  packet[0] = control_field;
  packet[1] = type;
  packet[2] = length;
  memcpy(packet+3, value, length);
}
```

Preparação e envio para o camada de ligação de dados o pacote de controlo de START ou de END (controlo_field)

Retorna em função do llwrite.

```
int startAndEndApiConnection(int fd, unsigned char control_field, int img_size)
{
  unsigned char packet[5];
  unsigned char nbytesneeded;
  unsigned char result[5];

  nbytesneeded = intToArrayChar(img_size, result);

  createControlPacket(packet, control_field, 0, nbytesneeded , result);
  int lengthControloPacket = 3 + nbytesneeded;

  return llwrite(fd, packet, lengthControloPacket);
}
```

Preparação e envio para o camada de ligação de dados o pacote de dados

```
int apiWrite(int fd, unsigned char * buffer, int length)
{
  unsigned char packet[MAX_SIZE];
  unsigned char result[2];
  int nbytesneeded;

  nbytesneeded = intToArrayChar(length, result);
  if(nbytesneeded == 1)
  {
    result[1] = result[0];
```

```
    result[0] = 0x00;
  }

  int data_packet_size = createDataPacket(packet,result[1], result[0], buffer);

  return llwrite(fd, packet, data_packet_size);
}
```

Emissor

Trata a informação conforme o tipo de pacote

- Pacote de START : abre o ficheiro de output e guarda o tamanho da image que está ser enviada
- Pacote de END : chama o llclose
- Pacote de dados : verifica se está na sequência correta e se isto se verificar escreve no ficheiro de output o recebido pelo campo de dados

```
int apiread(int fd)
{
  if(infoFrame[0] == 1)
  {
    if(infoFrame[1] == nsq)
    {
      nsq += 1;
      int k = 256 * infoFrame[2] + infoFrame[3];

      memcpy(infoFrame, infoFrame + 4, k);

      write(supertux_fd, infoFrame, k);
    }
  }
  else if(infoFrame[0] == 2)
  {
    supertux_fd = open("supertux.gif", O_WRONLY |O_TRUNC | O_CREAT, 0777);
    unsigned char result [infoFrame[2]];
    memcpy(result, infoFrame + 3, 2);

    img_size = arrayCharToInt(result, infoFrame[2]);
  }
  else if (infoFrame[0] == 3)
  {
    llclose(fd);
    return 3;
  }
  else return -1;

}
```

7. Validação

Com o propósito de testar a nossa aplicação, fez-se os seguintes testes:

- ficheiro de texto com tamanhos variáveis;
- ficheiro de extensão *gif* fornecido pelo professor para teste;
- outro ficheiro da mesma extensão;
- interrupção da ligação da porta série;
- criação de ruído.

8. Eficiência do protocolo de ligação de dados

A caraterização teórica de um protocolo Stop&Wait, que deverá ser empregue como termo de comparação, encontra-se descrita nos slides de *Camada de Ligação de Dados* das aulas teóricas. As tabelas que originaram os gráficos apresentados encontram-se no **Anexo II**.

1. Variação do FER

Como se pode verificar, há uma relação inversa entre a variação do FER e a eficiência da transferência dos dados, como se poderá verificar na tabela abaixo:

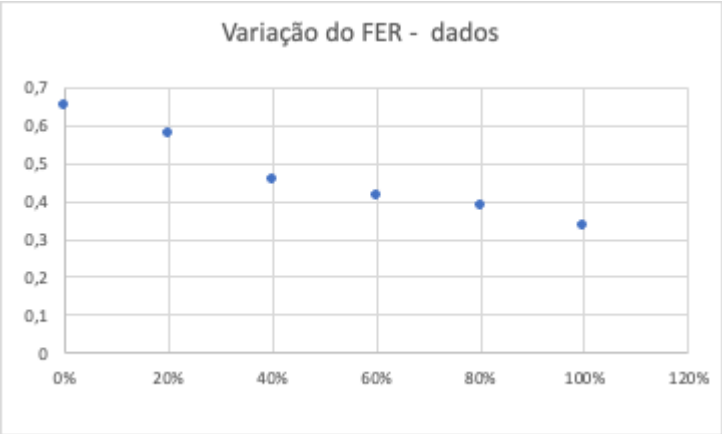


Tabela 1: Gráfico da relação entre eficiência e variação do FER.

2. Variação do tempo de propagação

Tal como na relação anterior, mas desta vez o declive é mais acentuado, sendo tal percetível na imagem abaixo.

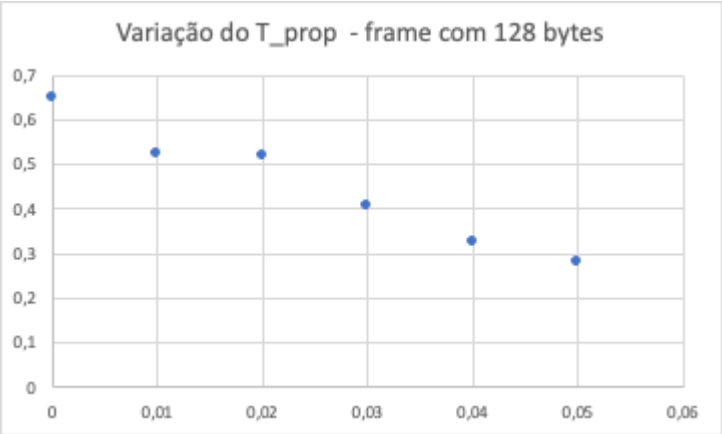


Tabela 2: Gráfico da relação entre eficiência e o tempo de propagação.

3. Variação da baudrate (C)

Esta relação revelou-se ser mais incomum, uma vez que a eficiência tem picos e quedas ao longo da variação do baudrate.

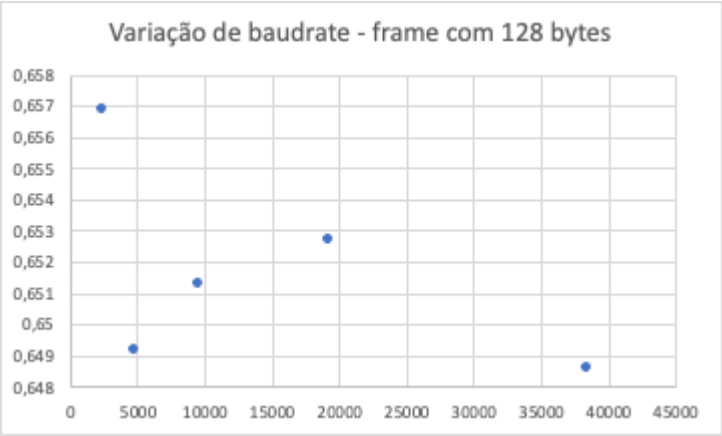
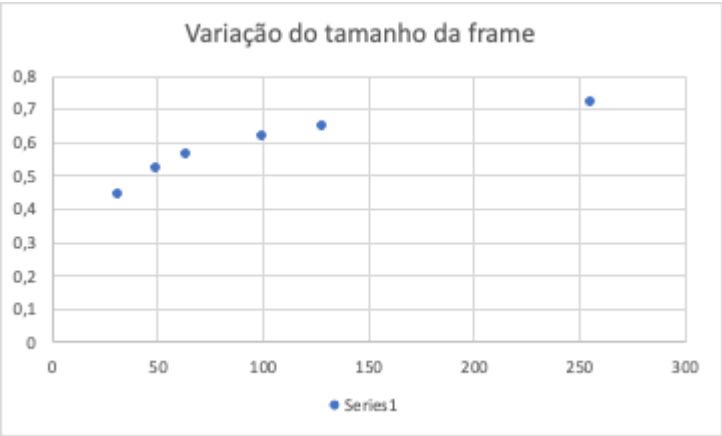


Tabela 3: Gráfico da relação entre eficiência e variação do baudrate.

4. Variação do tamanho das *frames* de informação

Na generalidade, há um aumento da eficiência com o aumento do tamanho das *frames*, sendo que esta tende a estabilizar.



9. Conclusões

Em suma, como foi referido o protocolo de ligação de dados visa garantir a independência entre as camadas, assumindo que a informação passada não contem erros.

Deste modo, foi concretizada com sucesso a aprendizagem sobre o modo de funcionamento da transmissão de ficheiros.

Anexo I - Código fonte

transmitter.c

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

#include <sys/types.h>
```

```c
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <math.h>

#define BAUDRATE B2400
#define MODEMDEVICE "/dev/ttyS1"
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FALSE 0
#define TRUE 1

#define FLAG 0x7E
#define CMD 0x03
#define ANS 0x01
#define SET 0x03
#define DISC 0x0B
#define UA 0x07
#define ESC 0x7D
#define RR 0x05
#define RR8 0x85
#define REJ 0x01
#define REJ8 0x81
#define TRANSMITTER 0
#define RECEIVER 1
#define DATA 1
#define START_ 2
#define END 3
#define NOTHING 0x00
#define MAX_SIZE 256


unsigned char nsequence = 0;
unsigned char nSequencePacket = 0;
int info_frame_size;

struct termios oldtio,newtio;

enum States
{
  START , FLAG_RCV, A_RCV, C_RCV, BCC_OK, STOP
};

unsigned char infoFrame[MAX_SIZE * 2];

int flag_signal=1, conta=1;

void atende()                   // atende alarme
{
    printf("alarme # %d\n", conta);
    flag_signal=1;
    conta++;
```

```
}

int sendFrame (int fd, unsigned char * frame, int length)
{
  int res = write(fd, frame, length);
  alarm(3);
  flag_signal = 0;
  return res;
}

int printMsg(unsigned char* msg, int length)
{
  //printf("--------------------\n");
  for(int i = 0; i < length; i++)
    {
        //printf("%02X\t", msg[i]);
    }
  //printf("--------------------\n");
}

unsigned char checkingOneResponse(int fd, unsigned char * frame_to_retransmit,
unsigned address_field, unsigned char control_field)
{
  enum States state;
  unsigned char buf[5];
  conta = 0;

  int res, i, no_error = 1;

  while(conta < 4){

    if (flag_signal){
      sendFrame(fd,frame_to_retransmit,5);
      no_error = 1;
    }

    if(no_error)
    {
      state = START;
      i=0;
      while(state != STOP && !flag_signal)
      {
        res = read(fd,&buf[i],1) ;

        switch (state)
        {
          case START:
            if(buf[i] == FLAG) state = FLAG_RCV;
            break;
          case FLAG_RCV:
            if(buf[i] == address_field) state = A_RCV;
            else if (buf[i] == FLAG) state = FLAG_RCV;
            else state = START;
            break;
```

```
            case A_RCV:
              if(buf[i] == control_field) state = C_RCV;
              else if (buf[i] == FLAG) state = FLAG_RCV;
              else state = START;
              break;
            case C_RCV:
              if(buf[i] == (address_field^control_field)) state = BCC_OK;
              else if (buf[i] == FLAG) state = FLAG_RCV;
              else state = START;
              break;
            case BCC_OK:
              if (buf[i] == FLAG) state = STOP;
              else state = START;
              break;
          }
        }
        if(state == STOP)
        {
          alarm(0);
          return 0;
        }
      }
    }
    return 1;
}

unsigned char checkingFourResponse(int fd, unsigned address_field, unsigned char
alt1, unsigned char alt2, unsigned char alt3, unsigned char alt4)
{
    enum States state;
    unsigned char buf[5];

    int res, i;
    unsigned char found = 0x00;
    conta = 1;

    while(conta < 4){

      if (flag_signal){
        alarm(0);
        sendFrame(fd,infoFrame,info_frame_size);

      }

      state = START;
      i=0;
      while(state != STOP && !flag_signal)
      {

        res = read(fd,&buf[i],1);

        switch (state)
        {
          case START:
```

```
            if(buf[i] == FLAG) state = FLAG_RCV;
            break;
          case FLAG_RCV:
            if(buf[i] == address_field) state = A_RCV;
            else if (buf[i] == FLAG) state = FLAG_RCV;
            else state = START;
            break;
          case A_RCV:
            if((buf[i] == alt1) || (buf[i] == alt2) || (buf[i] == alt3) || (buf[i]
== alt4))
            {
              state = C_RCV;
              found = buf[i];
            }
            else if (buf[i] == FLAG) state = FLAG_RCV;
            else state = START;
            break;
          case C_RCV:
            if(buf[i] == (address_field^found)) state = BCC_OK;
            else if (buf[i] == FLAG) state = FLAG_RCV;
            else state = START;
            break;
          case BCC_OK:
            if (buf[i] == FLAG) state = STOP;
            else state = START;
            break;
        }
      }
      if(state == STOP)
      {

        if(found == REJ || found == REJ8)
        {
          conta++;
          alarm(0);
          sendFrame(fd, infoFrame, info_frame_size);
          state = START;
          continue;
        }
        else
        {
          return found;
        }
      }
    }
    return 1;
}

int byteStuffing(unsigned char * buffer, int length)
{
    int j = 1;
    infoFrame[0] = buffer[0];
    int changes = 0;
```

```
  for (int i = 1; i < length - 1; i++)
  {
    if( buffer[i] == FLAG || buffer[i] == ESC)
    {
      infoFrame[j] = ESC;
      j++;
      changes++;
      infoFrame[j] = buffer[i] ^ 0x20;
    }
    else infoFrame[j] = buffer[i];
    j++;

  }
  infoFrame[j] = buffer[length - 1];
  j += 1;
  return j;
}

int setSettings(int fd)
{
  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]    = 0.1;   /* inter-character timer unused */
    newtio.c_cc[VMIN]     = 0;   /* blocking read until 5 chars received */

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

    tcflush(fd, TCIOFLUSH);

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


}

int createFrame(unsigned char * frame, unsigned char address_field, unsigned char
control_field)
{
```

```
  frame[0] = FLAG;
  frame[1] = address_field;
  frame[2] = control_field;
  frame[3] = address_field^control_field;
  frame[4] = FLAG;
}

revertSettings(int fd)
{
  if ( tcsetattr(fd,TCSANOW,&oldtio) == -1) {
      perror("tcsetattr");
      exit(-1);
    }
}


int llopen(int porta, unsigned char flag)

  char path[12] = "/dev/ttyS";
  char port = porta + '0';
  memcpy(path + strlen(path) , &port,1);

  int fd = open(path, O_RDWR | O_NOCTTY );

  if (fd <0) {perror(path); return -1; }
  setSettings(fd);

  unsigned char ua_expected[5];
  unsigned char set[5];

  createFrame(ua_expected, CMD, UA);
  createFrame(set, CMD, SET);

  sendFrame(fd, set, 5);

  unsigned char res = checkingOneResponse(fd, set, CMD, UA);
  if(res == 1) return -1;
  }
  return fd;
}

int createControlPacket(unsigned char * packet, unsigned char control_field,
unsigned char type, unsigned char length,  unsigned char * value)
{
  packet[0] = control_field;
  packet[1] = type;
  packet[2] = length;
  memcpy(packet+3, value, length);
}

int createDataPacket(unsigned char * packet, unsigned char l1, unsigned char l2,
unsigned char * data)
{
  packet[0] = DATA;
```

```
  packet[1] = nSequencePacket;
  packet[2] = l2;
  packet[3] = l1;

  int k = l2*256 + l1;
  nSequencePacket = (nSequencePacket + 1) % 256;
  memcpy(packet+4, data, k);

  return k + 4;
}

void addFrameHeader(unsigned char * frame)
{
  frame[0] = FLAG;
  frame[1] = CMD;
  frame[2] = nsequence;
  frame[3] = CMD ^ nsequence;
  nsequence = (nsequence + 1)%2;
}

unsigned char calculateBCC (unsigned char * frame, int length)
{
  unsigned char bcc = 0;
  for(int i = 0; i < length; i++)
  {
    bcc = bcc ^ frame[i];
  }
  return bcc;
}

void addFrameTrailer(unsigned char * frame, unsigned char bcc,  int pos)
{
  frame[pos] = bcc;
  frame[pos+1] = FLAG;
}

int llwrite(int fd, unsigned char * buffer, int length)
{
  unsigned char frame [length + 6];

  addFrameHeader(frame);
  memcpy(frame + 4, buffer, length);

  unsigned bcc = calculateBCC(buffer, length);
  addFrameTrailer(frame, bcc,  4 + length);

  int nbytes = byteStuffing(frame, length + 6);

  info_frame_size = nbytes;
  int res = sendFrame(fd, infoFrame, info_frame_size);

  unsigned char response = checkingFourResponse(fd, CMD, RR, RR8, REJ, REJ8);
  if(response == 1) return -1;
```

```
  alarm(0);

  return res;
}

int llclose(int fd)
{
  unsigned char disc[5];
  unsigned char ua[5];

  createFrame(disc, CMD, DISC);
  createFrame(ua, ANS, UA);

  sendFrame(fd, disc, 5);

  checkingOneResponse(fd, disc, ANS, DISC);
  sendFrame(fd, ua, 5);

  return 0;
}

int intToArrayChar(int value, unsigned char * result)
{
  int i = 0;
  while(value > 0)
  {
    result[i] = value % 256;
    value /= 256;
    i++;
  }
  return i;
}

int startAndEndApiConnection(int fd, unsigned char control_field, int img_size)
{
  unsigned char packet[5];
  unsigned char nbytesneeded;
  unsigned char result[5];

  nbytesneeded = intToArrayChar(img_size, result);

  createControlPacket(packet, control_field, 0, nbytesneeded , result);
  int lengthControloPacket = 3 + nbytesneeded;

  return llwrite(fd, packet, lengthControloPacket);
}

int apiWrite(int fd, unsigned char * buffer, int length)
{
  unsigned char packet[MAX_SIZE];
  unsigned char result[2];
  int nbytesneeded;

  nbytesneeded = intToArrayChar(length, result);
```

```
  if(nbytesneeded == 1)
  {
    result[1] = result[0];
    result[0] = 0x00;
  }

  int data_packet_size = createDataPacket(packet,result[1], result[0], buffer);

  return llwrite(fd, packet, data_packet_size);
}


int main(int argc, char** argv)
{
    int fd,c, res;

    (void) signal(SIGALRM, atende);

    int i = 0, sum = 0, speed = 0;

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

  /*
    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.
  */

    int porta = ((int) (argv[1][strlen(argv[1]) - 1]) - '0');


    fd = llopen(porta, TRANSMITTER);
    if(fd < 0){ perror("error when trying openning.."); return -1;}

    int fdImg = open("pinguim.gif", O_RDONLY);
    struct stat sb;

    if(fstat(fdImg, &sb) == -1)
    {
      perror("stat");
      exit(1);
    }

    int img_size = (int) sb.st_size;

    if(startAndEndApiConnection(fd, START_, img_size) < 0) {
        perror("tried 3 times\n");
        return -1;
      }
```

```
    unsigned char buffer[10968];
    int end = 0;

    int frame_size = 128;
    int mod = img_size % frame_size;
    int j = 0;

    struct timeval before_sending;
    struct timeval after_sending;

    gettimeofday(&before_sending, NULL);
    while(end < (img_size - mod))
    {
      for(int i = 0; i < frame_size; i++)
      {
        read(fdImg, &buffer[i], 1);
      }

      if(apiWrite(fd, buffer, frame_size) < 0)
      {
        perror("tried 3 times\n");
        return -1;
      }

      end += frame_size;
      j++;
    }

    for(int i = 0; i < mod; i++) read(fdImg, &buffer[i], 1);

    apiWrite(fd, buffer, mod);

    gettimeofday(&after_sending, NULL);
    time_t numSec = after_sending.tv_sec - before_sending.tv_sec;
    suseconds_t uSec = after_sending.tv_usec - before_sending.tv_usec;
    printf("tempo decorrido em segundos : %ld \t tempo em usec : %ld", numSec,
uSec);

    startAndEndApiConnection(fd, END, img_size);

    llclose(fd);

    revertSettings(fd);

    close(fd);
    close(fdImg);

    return 0;
}
```

receiver.c

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

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <string.h>


#define BAUDRATE B2400
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FALSE 0
#define TRUE 1
#define FLAG 0x7E
#define FLAG_STUFFED_BYTE_0 0x7D
#define FLAG_STUFFED_BYTE_1 0x5E
#define ESC 0x7D
#define CMD 0x01
#define ANS 0x03
#define UA 0x07
#define SET 0x03 // Alterar
#define RR 0x05
#define REJ 0x01
#define DISC 0x0B
#define MAX_SIZE 256

#define TRANSMITTER 0
#define RECEIVER 1

unsigned char infoFrame[MAX_SIZE];
unsigned char nreceiver = 1;
unsigned char nsq = 0;
int img_size;
int supertux_fd;
struct termios oldtio,newtio;

enum States
{
  START , FLAG_RCV, A_RCV, C_RCV, BCC_OK, STOP
};

unsigned char calculateBCC (unsigned char * frame, int length)
{
  unsigned char bcc = 0;
  for(int i = 0; i < length; i++)
  {
    bcc = bcc ^ frame[i];
  }
  return bcc;
```

```
}

int createFrame(unsigned char * frame, unsigned char address_field, unsigned char
control_field)
{
  frame[0] = FLAG;
  frame[1] = address_field;
  frame[2] = control_field;
  frame[3] = address_field^control_field;
  frame[4] = FLAG;
}

revertSettings(int fd)
{
  if ( tcsetattr(fd,TCSANOW,&oldtio) == -1) {
      perror("tcsetattr");
      exit(-1);
    }
}

int setSettings(int fd)
{
  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]    = 0.1;   /* inter-character timer unused */
    newtio.c_cc[VMIN]     = 1;   /* blocking read until 1 chars received */

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

    tcflush(fd, TCIOFLUSH);

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


int checkingFrame(int fd, unsigned char address_field, unsigned char
control_field)
```

```
{
  enum States state = START;
  unsigned char buf[5];
  int res, i = 0;

  while(state != STOP)
  {
    res = read(fd,&buf[i],1) ;

    switch (state)
    {
      case START:
        if(buf[i] == FLAG) state = FLAG_RCV;
        break;
      case FLAG_RCV:
        if(buf[i] == address_field) state = A_RCV;
        else if (buf[i] == FLAG) state = FLAG_RCV;
        else state = START;
        break;
      case A_RCV:
        if(buf[i] == control_field) state = C_RCV;
        else if (buf[i] == FLAG) state = FLAG_RCV;
        else state = START;
        break;
      case C_RCV:
        if(buf[i] == (address_field^control_field)) state = BCC_OK;
        else if (buf[i] == FLAG) state = FLAG_RCV;
        else state = START;
        break;
      case BCC_OK:
        if (buf[i] == FLAG) state = STOP;
        else state = START;
        break;
    }
  }
  return 0;
}

int llopen(int porta, unsigned char flag)
{
  char path[12] = "/dev/ttyS";
  char port = porta + '0';
  unsigned char ua[5];

  memcpy(path + strlen(path) , &port,1);

  int fd = open(path, O_RDWR | O_NOCTTY );

  if (fd <0) {perror(path); exit(-1);}

  setSettings(fd);

  createFrame(ua, ANS, UA);
  checkingFrame(fd, ANS, SET);
```

```
  int res = write(fd,ua,5);

  return fd;
}

int removeFrameTrailer(int pos)
{
  infoFrame[pos-1] = 0x00;
  infoFrame[pos-2] = 0x00;
  return 0;
}

int removeFrameHeader(int nbytesToCpy)
{
  memcpy(infoFrame, infoFrame + 4, nbytesToCpy);
  return 0;
}

int removePacketHeader(int pos, int nbytesToCpy)
{
  memcpy(infoFrame, infoFrame + pos, nbytesToCpy);
  return 0;
}

int llwrite(int fd, char * buffer, int length)
{
  write(fd,buffer, length);
  return 1;
}

int pot(int x , int n)
{
  int p = 1;
  for(int i  =0; i < n;i++)
  {
    p = p * x;
  }
  return p;
}

int arrayCharToInt(unsigned char * value, unsigned char length)
{
  int res = 0;
  for(int i = 0; i< length;i++)
  {
    res += value[i] * pot(256, i);
  }
  return res;
}

int apiread(int fd)
{
  if(infoFrame[0] == 1)
```

```
    {
      if(infoFrame[1] == nsq)
      {
        nsq += 1;
        int k = 256 * infoFrame[2] + infoFrame[3];

        memcpy(infoFrame, infoFrame + 4, k);

        write(supertux_fd, infoFrame, k);
      }
    }
    else if(infoFrame[0] == 2)
    {
      supertux_fd = open("supertux.gif", O_WRONLY |O_TRUNC | O_CREAT, 0777);
      unsigned char result [infoFrame[2]];
      memcpy(result, infoFrame + 3, 2);

      img_size = arrayCharToInt(result, infoFrame[2]);
    }
    else if (infoFrame[0] == 3)
    {
      llclose(fd);
      return 3;
    }
    else return -1;

  }

  int checkBCC1(int pos)
  {
    return (infoFrame[pos-2] ^ infoFrame[pos-1]) == infoFrame[pos];
  }

  int checkBCC2(int pos)
  {
    unsigned char bcc = calculateBCC(infoFrame, pos);
    return bcc == infoFrame[pos];
  }

  int llread(int fd, unsigned char * buffer)
  {
    int acc = 2, i=0, res;
    int stop1 = 0, starting_frame = 0;
    memset(infoFrame, 0, 256);
    memset(buffer, 0, 256);

    while(!stop1)
    {
      res = read(fd, &buffer[i],1);
      if(buffer[i] == FLAG && !starting_frame)
      {
        i = 0;
        buffer[0] = FLAG;
```

```
      starting_frame = 1;
    }
    else if(buffer[i] == FLAG && starting_frame && buffer[i-1] != FLAG)
    {
      stop1 = 1;
    }
    else if(buffer[i] == FLAG && buffer[i-1] == FLAG)
    {
      i--;
    }
    i++;
  }

  int numBytesAfterDestuffing = byteDestuffing(buffer, i);
  unsigned char nsSender = infoFrame[2];
  unsigned char response[5];

  if(!checkBCC1(3))
  {
    createFrame(response, ANS, REJ | (nreceiver << 7));
    write(fd, response,5);
    tcflush(fd, TCIOFLUSH);
    return -1;
  }
  removeFrameHeader(numBytesAfterDestuffing-4);

  if (((nsSender + 1)%2) != nreceiver)
  {
    createFrame(response, ANS, RR | (nreceiver << 7));
    int x = write(fd, response,5);
    return 0;
  }

  else if((((nsSender + 1)%2) == nreceiver) && checkBCC2(numBytesAfterDestuffing-
6))
  {
    createFrame(response, ANS, RR | (nreceiver << 7));
    write(fd, response,5);
    nreceiver = (nreceiver + 1) % 2;

  }
  else if((((nsSender + 1)%2) == nreceiver) && !checkBCC2(numBytesAfterDestuffing-
6))
  {
    createFrame(response, ANS, REJ | (nreceiver << 7));
    write(fd, response,5);
    tcflush(fd, TCIOFLUSH);
    return -1;
  } else
  {
    createFrame(response, ANS, RR | (nreceiver << 7));
    write(fd, response,5);
  }
```

```
  removeFrameTrailer(numBytesAfterDestuffing - 3);

  return numBytesAfterDestuffing - 5;
}

int llclose(int fd)
{
  unsigned char disc[5];

  createFrame(disc, CMD, DISC);
  checkingFrame(fd, ANS ,DISC);
  write(fd, disc, 5);
  checkingFrame(fd, CMD, UA);

  return 0;
}

int byteDestuffing(unsigned char * buffer, int length)
{
  int i = 0, j = 0;

  for (int i = 0; i < length; i++)
  {
    if(buffer[i] == ESC)
    {
      if(buffer[i+1] == 0x5E)
      {

        infoFrame[j] = FLAG;
      }
      else infoFrame[j] = ESC;
      i++;
    }
    else infoFrame[j] = buffer[i];
    j++;

  }
  return j;
}

int main(int argc, char** argv)
{
    int fd,c;
    struct termios oldtio,newtio;
    unsigned char buf[1];

    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);
    }

  /*
```

```
    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.
  */
    int porta = ((int) (argv[1][strlen(argv[1]) - 1]) - '0');

    fd = llopen(porta, RECEIVER);

    unsigned char buffer [MAX_SIZE];
    int res = 0, i = 0;

    while(res != 3)
    {
      int res1 = llread(fd, buffer);
      if(res1 > 1) res1 = apiread(fd);
    }

    sleep(1);
    revertSettings(fd);

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

Anexo II - Tabelas de medições

Variação do T_prop - frame com 128 bytes					
Tempo de propação (ms)	Tamanho do ficheiro (bits)	Capacidade de ligação (C = bit/s)	Tempo de transferência (s)	Débito recebido (R = bit/s)	Eficiência (S = R/C)
0	87744	38400	3,523099	24905,34612	0,648576722
10	87744	38400	4,383432	20017,19201	0,521281042
20	87744	38400	4,423672	19835,10532	0,516539201
30	87744	38400	5,613295	15631,46067	0,407069288
40	87744	38400	7,06817	12413,96288	0,323280283
50	87744	38400	8,217249	10678,02619	0,278073599

Variação do FER - dados					
FER	Tamanho do ficheiro (bits)	Capacidade de ligação (C = bit/s)	Tempo de transferência (s)	Débito recebido (R = bit/s)	Eficiência (S = R/C)

Variação do FER - dados

0%	87744	38400	3,523099	24905,34612	0,648576722
20%	87744	38400	3,945632	22238,26246	0,579121418
40%	87744	38400	5,023454	17466,86642	0,454866313
60%	87744	38400	5,546343	15820,15393	0,411983175
80%	87744	38400	5,903745	14862,43054	0,387042462
100%	87744	38400	6,786343	12929,49678	0,336705645

Variação do tamanho da frame

Tamanho da trama	Tamanho do ficheiro (bits)	Capacidade de ligação (C = bit/s)	Tempo de transferência (s)	Débito recebido (R = bit/s)	Eficiência (S = R/C)
32	87744	38400	5,143676	17058,61722	0,444234823
50	87744	38400	4,373215	20063,95752	0,522498894
64	87744	38400	4,064093	21590,05712	0,562241071
100	87744	38400	3,719669	23589,19571	0,614301971
128	87744	38400	3,523099	24905,34612	0,648576722
256	87744	38400	3,178789	27602,96453	0,718827201

Variação de baudrate - frame com 128 bytes

Tamanho da trama	Tamanho do ficheiro (bits)	Capacidade de ligação (C = bit/s)	Tempo de transferência (s)	Débito recebido (R = bit/s)	Eficiência (S = R/C)
128	87744	38400	3,523099	24905,34612	0,648576722
128	87744	19200	7,001516	12532,14304	0,652715783
128	87744	9600	14,033059	6252,663799	0,651319146
128	87744	4800	28,158454	3116,080165	0,649183368
128	87744	2400	55,656783	1576,519433	0,656883097