Protocolo de Ligação de Dados

Mestrado Integrado em Engenharia Informática e Computação

Redes de Computadores

Turma 5:
Daniel Pereira Machado - 201506365
José Pedro Dias de Almeida Machado - 201504779
Sofia Catarina Bahamonde Alves - 201504570

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

9 de Novembro de 2017

1 Sumário

Este relatório tem como objetivo contextualizar o trabalho que tem vindo a ser desenvolvido na Unidade Curricular Redes de Computadores. Trata-se de um protocolo de envio de informação de um computador para outro, através da porta de série. Deste modo, foram implementadas um conjunto de funções de leitura, escrita e tratamento de dados.

O projeto foi concluído com sucesso. Foi desenvolvida uma aplicação capaz de enviar e receber os dados corretamente. Quando ocorre uma falha na transmissão de dados o programa é capaz de restabelecer a transmissão, tratando os erros devidamente.

2 Introdução

O objetivo do trabalho é implementar um dado protocolo de ligação de dados, especificado no guião fornecido, assim como testá-lo com uma aplicação simples de transferência de ficheiros. Esta implementação deve garantir um serviço de comunicação de dados fiável entre dois sistemas ligados por uma porta de série, assegurando que apesar de interrupções da comunicação e de interferências os dados são transmitidos sem qualquer problema.

Para isto, foi necessário desenvolver funções de criação e sincronização de tramas (*framing*), estabelecimento e terminação da ligação, numeração de tramas, confirmação de receção de uma trama sem erros e na sequência correta, controlo de erros e de tramas repetidas.

Este relatório será organizado da seguinte forma:

- Arquitetura - Demonstração dos blocos funcionais e interfaces.
- Estrutura do Código - Exposição das APIs, principais estruturas de dados, principais funções e a sua relação com a arquitetura.
- Casos de Uso Principais - Identificação dos casos de uso principais e sequências de chamada de funções.
- Protocolo de Ligação Lógica - Identificação dos principais aspetos funcionais e descrição da estratégia de implementação destes aspetos.
- Protocolo de Aplicação - Identificação dos principais aspetos funcionais; descrição da estratégia de implementação destes aspetos.
- Validação - Descrição dos testes efetuados com apresentação quantificada dos resultados.
- Eficiência do protocolo de ligação de dados - caraterização estatística da eficiência do protocolo, feita com recurso a medidas sobre o código desenvolvido. A caracterização teórica de um protocolo Stop&Wait, que deverá ser usada como termo de comparação, encontra-se descrita nos slides de Ligação Lógica das aulas teóricas
- Conclusão - Síntese da informação apresentada anteriormente e reflexão sobre os objetivos de aprendizagem alcançados.

3 Arquitetura

O nosso projeto está dividido em duas camadas: *Application Layer* e *Data Link Layer*. Estas duas camadas são responsáveis pelo bom funcionamento do mesmo. A parte relativa à aplicação em si está representada nos ficheiros *application_layer.h* e *application_layer.c*, a ligação lógica está representada nos ficheiros *data_link.h* e *data_link.c*. Além disso ainda utilizamos um ficheiro auxiliar onde estão declaradas as macros utilizadas ao longo do código, o ficheiro *utils.h*.

3.1 Camada de aplicação

A camada de aplicação, desenvolvida nos ficheiros *application_layer.h* e *application_layer.c* é a camada lógica situada diretamente acima da camada de ligação de dados, servindo de ponte entre esta e a interface com o utilizador. É responsável pela utilização da camada de lógica para comunicar e transferir ficheiros de acordo com os parâmetros que lhe são passados pelo utilizador.

3.2 Camada de ligação de dados

A camada de ligação de dados é a camada de mais baixo nível na nossa aplicação. Esta funciona como ponte entre a porta de série e a camada de aplicação, sendo responsável pela comunicação entre as duas máquinas. Deste modo, contém as funções que configuram, iniciam e terminam a ligação, assim como as que escrevem e lêem dados da porta de série, tratando das necessidades da comunicação – nomeadamente tratamento de erros e stuffing/destuffing de pacotes.

4 Estrutura do código

4.1 Application Layer

A Application Layer é representada através de uma struct, onde são guardados os dados relativos à mesma. Aqui encontra-se o descritor correspondente à porta de série, modo de trasmissão (emissor ou recetor) e ainda um inteiro (0 ou 1) que diferencia se estamos a usar a porta ttyS0 ou ttyS1.

```
typedef struct  {
  int port;
  int fileDescriptor;/*Descritor correspondente a porta serie*/
  status mode;/*TRANSMITTER | RECEIVER*/
}applicationLayer;
```

Nesta camada, as principais funções utilizadas são as seguintes:

```
void set_connection(char * port, char * mode);
void send_data(char * path, char* filename);
void receive_data();
```

Na função *set_ connection* incializa-se a struct *applicationLayer*, passando como dados a porta de série que vai ser utilizada e o modo de transmissão. Além disso establece-se a conexão entre o recetor e o transmissor, chamando a função *llopen()* da camada lógica.

Depois de ser estabelecida a conexão inicial, pode-se efetivamente enviar os pacotes com informação. Isto acontece na função *send_ data*, onde após ser escolhido o *path* de destino, são enviados os diferentes pacotes através das funções *send_ packets()* e *send_ control_ packet()*, esta última responsável por enviar o start e end packet que contêm informações essencias acerca do ficheiro a transmitir e delimitam a transmissão de informação.

Do lado do recetor é chamada a função *receive_ data()*. Deste modo, abre-se o descritor onde a informação vai ser registada.

4.2 Data Link Layer

Tal como acontece na Application Layer, a Data Link Layer é representada por uma struct, facilitando o armazenamento e acesso aos dados da mesma. Esta é consituída pelo descritor correspondente à porta de série, pela velocidade de transmissão, pelo número de sequência da trama (0 ou 1), pelo valor do temporizador, pelo número de tentativas em caso de falha, pelo modo de transmissão (emissor ou recetor) e por uma *struct* onde são guardadas as definições da porta de série.

```
typedef struct {
  char port[20];
  int baudRate;/*Velocidade de transmissao*/
  unsigned int sequenceNumber;    /*Numero de sequencia da trama: 0, 1*/
  unsigned int timeout;/*Valor do temporizador: 1 s*/
  unsigned int numTransmissions; /*Numero de tentativas em caso defalha*/
  status mode; /* RECEIVER || TRANSMITTER */
  struct termios portSettings;
}linkLayer;
```

Nesta camada, as principais funções utilizadas são as seguintes:

```
int llopen(int port,status mode);
int llread(int fd, unsigned char *packet);
int llwrite(int fd,  char * packet, int length);
int llclose(int fd);
```

A função *llopen()* é responsável por estabelecer a conexão inicial entre o recetor e o transmissor. Isto é feito através das tramas SET e UA. Depois são chamadas as funções *llread() llwrite()*, do lado do recetor e do emissor respetivamente. É aqui que é enviada a trama I e as respostas REJ e RR. Segue-se a função *llclose()*, onde se encontram os aspetos de terminação, nomeadamente o envio da trama DISC e UA.

5 Casos de uso principais

A nossa aplicação necessita de dois parâmetros para correr sendo que um deles é a porta série e o outro um carater (T ou R) para indicar se queremos correr o programa como transmissores de informação ou recetores. Ao correr o programa em qualquer um dos modos as funções que são invocadas em primeiro lugar são *init_ link_ layer* que vai iniciar a struct linkLayer com os parâmetros dados e logo de seguida é invocada a função *set_ connection* que por sua vez vai invocar llopen, cujo comportamento vai depender da forma como a porta série irá ser usada.

5.1 Modo de Transmissor

No caso de o programa estar a ser executado no modo de transmissor as funções mais importantes (por ordem de chamada) são:

- **send_data** que invoca **send_control_packet** que envia o start_packet contendo o nome e o tamanho do ficheiro, de seguida invoca **send_packets** e no final da transmissao envia novamente um pacote de controlo mas desta vez com o END_BYTE e de seguida termina com **llclose**.
- **send_packets** que vai lendo o ficheiro a enviar e invocando o **llwrite** a fimm de enviar os pacotes de dados.
- **create_Iframe** que vai fazer stuffing dos dados através de **stuff_frame** e preparar a trama de informação com as FLAGS necessárias, trama essa que é escrita na porta série por **write_packet**.
- **read_packet** responsável por ler a resposta à trama de dados enviada e depois é vericado por **valid_Sframe** que se trata de um REJ ou RR tendo em conta o sequence number.

5.2 Modo de Recetor

No caso de o programa estar a ser executado no modo de recetor as funções mais importantes (por ordem de chamada) são:

- **receive_data** que vai começar por invocar **receive_start_packet** que através da função **llread** vai tentar ler da porta de série até receber o control_packet com o START_BYTE. De seguida será novamente invocado **llread** num ciclo até receber o END_BYTE, que indica o fim da transmissão, que será terminada pela chamada à função **llclose**.
- **llread** que tenta ler até a trama ser válida da porta série através da função **valid_Iframe**, de seguida é invocada **destuff_frame** que vai fazer o destuffing da trama recebida é ainda feita a verificação do bcc2 e do sequence number através de **validBCC2** e de**valid_sequence_number**. A resposta a enviar é criada através da função **create_Sframe**.

6 Protocolo de Ligação Lógica

6.1 Principais Aspetos Funcionais

- Configuração da porta série;
- Estabelecimento e terminação da ligação através da porta série;
- Envio/receção de mensagens/comandos;
- Fazer stuff e destuff dos pacotes da camada de aplicação;

6.2 Implementação dos aspetos funcionais

6.2.1 llopen

A função **llopen** é responsável por estabelecer a ligação através da porta série em que inicialmente chama a função set_terminus que configura a porta série com as especificações pretendidadas. Quando o emissor executa escreve para a porta série o comando SET e aguarda que o recetor receba e envie de volta o comando UA que se for recebido faz com que retorne. Se UA não for recebido no tempo definido (timeOut em s) SET é reenviado e novamente aguarda-se por UA, isto até n vezes em que n é numTransmissions, se estas n vezes forem excedidas a aplicação aborta com uma mensagem de erro.

```
    while(state!=5 && !timedOut){
      if(read(fd,&c,1)==-1){
        printf("data_link - llopen: read error\n");
        exit(-1);
      }
      state = update_state(c,state,UA);
    }
  } while(timedOut && count < link_layer.numTransmissions);
```

6.2.2 llclose

A função **llclose** é responsável por terminar a ligação enviando o comando DISC e de forma análoga ao que acontece no **llopen** vai aguardar pela receção do DISC enviado pelo **llclose** do emissor e volta a reenviar tal como já foi explicado, se não receber o DISC aborta, caso receba envia o UA e retorna terminando assim a aplicação.

6.2.3 llwrite

A função **llwrite** primeiramente começa por criar as tramas de informação e escreve na porta série o packet que recebe como parâmetro e aguarda por uma resposta, se não receber qualquer resposta no tempo definido vai voltar a tentar o número de vezes definido,se receber o comando RR vai retornar terminando com sucesso, caso seja rejeitada através do comando REJ e neste caso é retransmitida até atingir o número máximo de tentativas definido.

```
do{
  if(write_packet(fd,frame,frame_length)<0){
    printf("Failed sending packet.\n");
    return -1;
```

```
    }
    timedOut = false;
    alarm(link_layer.timeout);

    while(!timedOut){

    if(read_packet(fd,response,&response_len)==0){

      if(valid_Sframe(response,response_len,RR)){
        alarm(0);
        link_layer.sequenceNumber =!link_layer.sequenceNumber;
        return 0;
      }

      if(valid_Sframe(response,response_len,REJ)){
        alarm(0);
        count=0;
        timedOut = true;
        }
      }
    }
  }while(timedOut && count<link_layer.numTransmissions);
```

6.2.4 llread

A função **llread** tenta ler da porta série até receber uma frame válida verificando os primeiros bytes da frame inclusive o BCC, de seguida verifica se o BCC2 é válido, se este for é verifcado o sequence number(se for válido é criada a resposta REJ e se não for significa que foi encontrado um duplicado), caso contrário a mensagem é rejeitada enviando o comando REJ.

7 Protocolo de Aplicação

O protocolo de aplicação encontra-se implementado na Application Layer, estando dependente dos ficheiros respetivos a essa camada, mas também da camada inferior, a Data Link Layer.

7.1 Principais Aspetos Funcionais

- Geração e transferência dos pacotes de controlo e de dados
- Leitura e escrita do ficheiro a transferir

7.2 Implementação dos aspetos funcionais

A implementação relativa à Application Layer inicia-se com a função *send_ data()*, sendo esta a responsável pelo comportamento do transmissor, enviando dados para a camada inferior, e esta envia para a porta de série. Nesta função são chamadas as funções que enviam os pacotes de controlo (*send_ packets()* e *send_ control_ packet()*) e os pacotes de dados.

```
char data_packet[PACKET_SIZE];
int packet_size= num_chars + PACKET_HEADER_SIZE;

data_packet[0] = DATA_BYTE;
data_packet[1] = i % 256;
data_packet[2] = 0;
data_packet[3] = num_chars;

memcpy(data_packet +PACKET_HEADER_SIZE, data, num_chars);

if(llwrite(app_layer.fileDescriptor,data_packet,packet_size)==-1)
exit(1);
```

No excerto acima encontra-se a especificação relativa à função *send_ packets()*,ficando cada byte com a informação respetiva. Relativamente à função *send_ control_ packets()* o esquema é relativamente semelhante e encontra-se em baixo.

```
start_packet[0] = control_byte;

start_packet[1] = FILE_SIZE_BYTE;
start_packet[2] = sizeof(info.st_size);
*((off_t *)(start_packet + 3)) = file_size;

start_packet[3 + sizeof(info.st_size)] = FILE_NAME_BYTE;
start_packet[4 + sizeof(info.st_size)] = filename_len;
strcat(start_packet + 5 + sizeof(info.st_size),filename);

llwrite(app_layer.fileDescriptor, start_packet, start_packet_len);
```

A função *receive_ data()* é resposável pelo comportamento principal do transmissor , recebe dados da camada inferior para compor o ficheiro lido.Deste modo, a função começa por ler, usando a função *llread()* da camada inferior, o pacote de controlo e daí tirar o nome do ficheiro, as suas permissões e o seu tamanho final. Depois enquanto receber pacotes válidos que não sejam o pacote final,

continua a escrever os bytes de informação vindos de *llread()* para o ficheiro, acabando quando receber o pacote final, o que se pode verificar neste excerto de código.

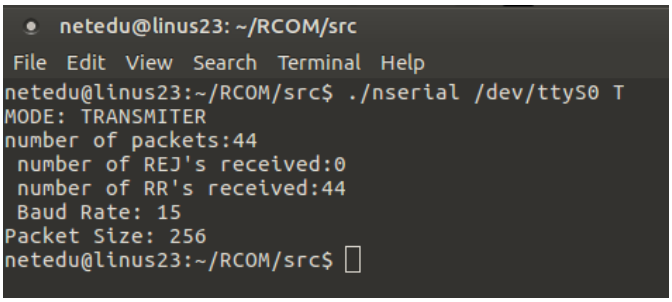
```
while(true){

  do{
    packet_length = llread(app_layer.fileDescriptor, packet);

  }while(packet_length ==0);

  if (packet_length < 0) {
    printf("app_layer - receive_data: error llread\n");
    close(fd);
    exit(-1);
  }

  if(packet[0] == END_BYTE)
  break;

  unsigned int data_len = packet[2] * 256 + packet[3];

  if (write(fd, packet + 4, data_len) != data_len) {
    printf("app_layer - receive_data: write error\n");
    close(fd);
    exit(-1);
  }
}
```

8 Validação

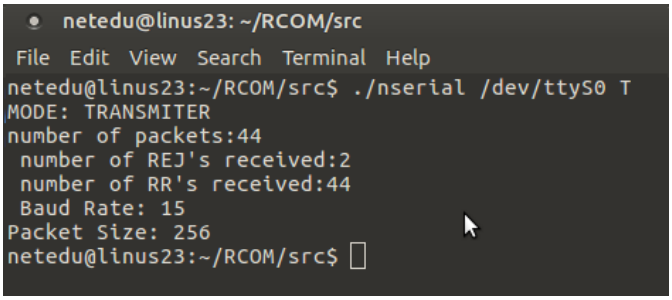
Para verificar a robustez e o bom funcionamento da aplicação foram implementados os seguintes testes:

- Enviar um ficheiro
- Enviar um ficheiro, primir o botão de interrupção durante a transmissão e reabrindo a transmissão depois.
- Enviar um ficheiro e introduzir erros na ligação com o cabo de cobre

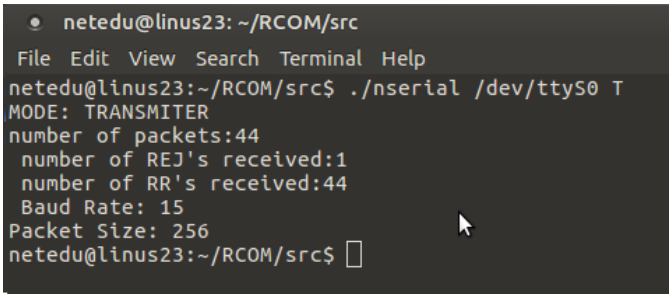
No primeiro teste o resultado foi o seguinte. Não houve qualquer interferência durante a transmissão e, deste modo não foi recebido nenhum REJ e foram recebidos tantos RR's quanto o numero de packets enviado.



O resultado à direita é referente ao segundo teste. Primiu-se o botão que interrompe a ligação 2 vezes e, como tal foram recebidos dois REJs. O número de RRs foi igual ao número de packets enviados.



No terceiro teste o resultado foi o seguinte. Passou-se um cabo de cobre descarnado através dos pinos metálicos que se encontram na placa que liga as duas portas de série. Assim, foram recebidos 2 REJs e o número de RRs foi igual ao número de packets enviados.



9 Eficiência do protocolo de ligação de dados

Um protocolo Stop and Wait é um método de transmissão para enviar informações entre dois dispositivos conectados, Este protocolo garante que a informação não é perdida devido a pacotes descartados e que os pacotes são recebidos na ordem correta. Isto deve-se ao facto de de após o transmissor enviar um pacote de dados não envia mais nada até receber um sinal de confirmação. Depois de receber um pacote válido o recetor envia a confirmação que tem um tempo limite (estabelecido pelo protocolo) para ser recebida, caso não seja recebida será enviado novamente o mesmo pacote. Normalmente neste tipo rudimentar de protocolo é adicionado um número de verificação de redundância (BCC1 e BCC2) se este número estiver errado o recetor não envia resposta agindo como se o pacote tivesse sido perdido e não apenas danificado. O problema deste tipo de protocolo está no facto de que se a resposta enviada pelo recetor tiver erros, o emissor vai pensar que não foi enviado o pacote e vai reenvia-lo ficando com 2 pacotes repetidos.

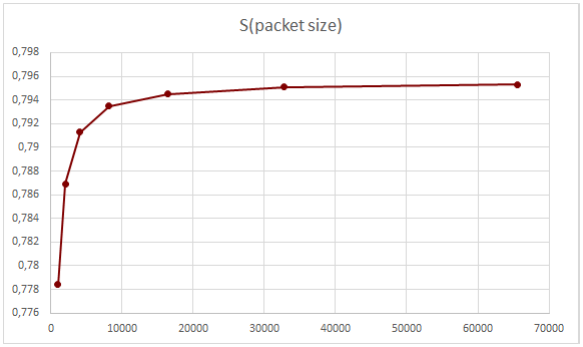


Gráfico 1 - Gráfico da eficiência relativamente ao tamanho dos pacotes

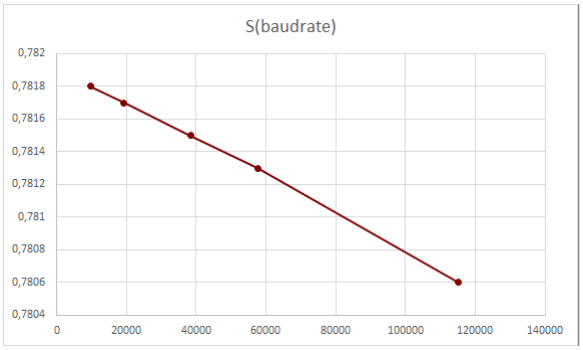


Gráfico 2 - Gráfico da eficiência relativamente à taxa de transmissão

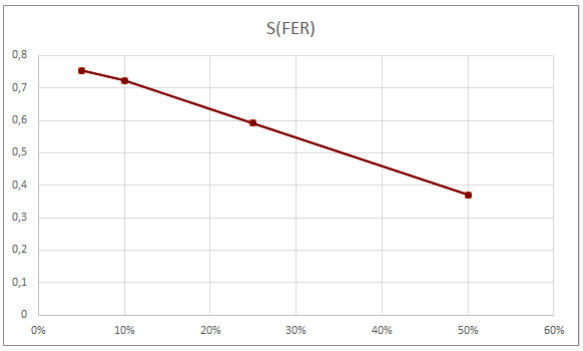


Gráfico 3 - Gráfico da eficiência relativamente à frequência de erros

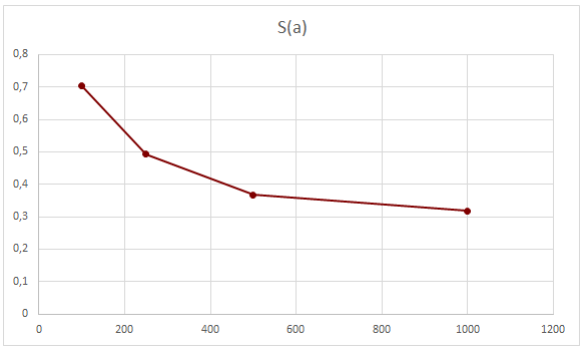


Gráfico 4 - Gráfico da eficiência relativamente ao parâmetro a

10 Conclusão

De modo geral, foram atingidos os principais objetivos deste projeto. Conseguimos interiorizar os conceitos abordados ao longo trabalho e elaborar código robusto o suficiente para fazer face à bateria de testes apresentada.

Relativamente à implementação, para nós, o grande desafio foi resolver todos os problemas de alocação de memória. Consideramos que fizemos uma adequada distinção entre camada lógica e camada de aplicação (Data Link Layer e Application Layer), o que nos ajudou bastante a organizar e implementar todo o projeto.

A Código fonte

A.1 application_layer.h

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/stat.h>

#include "data_link.h"



typedef struct  {
  int port;
  int fileDescriptor;/*Descritor correspondente a porta serie*/
  status mode;/*TRANSMITTER | RECEIVER*/
}applicationLayer;

applicationLayer app_layer;

void set_connection(char * port, char * mode);

void send_data(char * path, char* filename);
void receive_data();

void send_control_packet(int fd, char* filename, unsigned char control_byte);
void send_packets(int fd, char* filename);
char* receive_start_packet(off_t* file_size);
```

A.2 application_layer.c

```
x#include "application_layer.h"


void set_connection(char * port, char * stat){

  if(strcmp(port,COM1_PORT)==0)
  app_layer.port = COM1;

  if(strcmp(port,COM2_PORT)== 0)
  app_layer.port =COM2;

  if(strcmp(stat,"T")==0){
    app_layer.mode = TRANSMITTER;
    printf("TRANSMITTER\n");
  }

  if(strcmp(stat,"R")== 0){
    app_layer.mode = RECEIVER;
    printf("RECEIVER\n");
  }

  app_layer.fileDescriptor = llopen(app_layer.port,app_layer.mode);

  if(app_layer.fileDescriptor< 0){
    printf("app_layer - set_connection(): invalid file descriptor\n");
    exit(-1);
  }


}

void send_data(char * path, char* filename){
  char *full_path =
  (char *)malloc(sizeof(char) * (strlen(path) + 1 + strlen(filename)));

  strcpy(full_path, path);
  strcat(full_path, "/");
  strcat(full_path, filename);

  int fd = open(full_path, O_RDONLY);
  if (fd <0) {
    printf("app_layer - send_data: invalid file decriptor\n");
    exit(-1);
  }

  send_control_packet(fd,filename, START_BYTE);
  send_packets(fd,filename);
  send_control_packet(fd,filename, END_BYTE);
```

```
  llclose(app_layer.fileDescriptor);


}

void send_packets(int fd, char* filename){

  struct stat info;
  fstat(fd, &info);

  off_t file_size = info.st_size;


  char data[DATA_PACKET_SIZE];
  int i = 0;
  off_t bytes_to_read = file_size;

  while(bytes_to_read>0){
    int num_chars=read(fd,data,DATA_PACKET_SIZE);

    if(num_chars <0){
      printf("app_layer - send_packets: error reading data\n" );
      exit(-1);
    }

    char data_packet[PACKET_SIZE];
    int packet_size= num_chars + PACKET_HEADER_SIZE;

    data_packet[0] = DATA_BYTE;
    data_packet[1] = i % 256;
    data_packet[2] = 0;
    data_packet[3] = num_chars;

    memcpy(data_packet +PACKET_HEADER_SIZE, data, num_chars);

    if(llwrite(app_layer.fileDescriptor,data_packet,packet_size)==-1)
    exit(1);
    bytes_to_read -= num_chars;
    i++;

  }



}

void send_control_packet(int fd, char* filename,unsigned char control_byte){

  struct stat info;
  fstat(fd, &info);

  int filename_len = strlen(filename);
```

```
  off_t file_size = info.st_size;


  int start_packet_len = 5 + sizeof(info.st_size) + filename_len;
  char *start_packet = ( char *)malloc(sizeof(char) * start_packet_len);

  start_packet[0] = control_byte;

  start_packet[1] = FILE_SIZE_BYTE;
  start_packet[2] = sizeof(info.st_size);
  *((off_t *)(start_packet + 3)) = file_size;

  start_packet[3 + sizeof(info.st_size)] = FILE_NAME_BYTE;
  start_packet[4 + sizeof(info.st_size)] = filename_len;
  strcat(start_packet + 5 + sizeof(info.st_size),filename);

  llwrite(app_layer.fileDescriptor, start_packet, start_packet_len);

}

void receive_data(){

  // receive start packet

  off_t file_size;
  char* file_name;

  file_name = receive_start_packet(&file_size);

  strcpy(file_name, "p.gif");

  int fd = open(file_name, O_RDWR | O_CREAT | O_TRUNC);

  if (fd <0) {
    printf("app_layer - receive_data: invalid file descriptor\n");
    exit(-1);
  }


  unsigned char packet[PACKET_SIZE];
  int packet_length;

  while(true){

    do{
      packet_length = llread(app_layer.fileDescriptor, packet);

    }while(packet_length ==0);

    if (packet_length < 0) {
      printf("app_layer - receive_data: error llread\n");
      close(fd);
```

```
      exit(-1);
    }

    if(packet[0] == END_BYTE)
    break;

    unsigned int data_len = packet[2] * 256 + packet[3];

    if (write(fd, packet + 4, data_len) != data_len) {
      printf("app_layer - receive_data: write error\n");
      close(fd);
      exit(-1);
    }
  }

  llclose(app_layer.fileDescriptor);

  close(fd);

}

char* receive_start_packet(off_t* file_size){

  unsigned char packet[PACKET_SIZE];
  int packet_length;


  do {

    packet_length = llread(app_layer.fileDescriptor, packet);

    if ( packet_length <0) {
      printf("app_layer - receive_data - receive_start_packet: error.\n");
      exit(-1);
    }

  }while(packet[0] != (unsigned char)START_BYTE || packet_length==0);



  int i;
  // get file size
  i = 1;
  while (i < packet_length) {
    if (packet[i] == FILE_SIZE_BYTE){
      *file_size = *((off_t *)(packet + i + 2));
      break;
    }

    i += 2 + packet[i + 1];
  }
```

```
  // get file name
  i = 1;
  while (i < packet_length) {
    if (packet[i] == FILE_NAME_BYTE) {
      char * file_name = (char *)malloc((packet[i + 1] + 1) * sizeof(char));
      memcpy(file_name, packet + i + 2, packet[i + 1]);
      file_name[(packet[i + 1] + 1)] = 0;
      return file_name;
    }

    i += 2 + packet[i + 1];
  }


  return NULL;

}
```

A.3 data_link.h

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <termios.h>
#include <fcntl.h>
#include <stdbool.h>
#include <signal.h>
#include <unistd.h>
#include <errno.h>

#include "utils.h"


typedef struct {
  char port[20];
  int baudRate;/*Velocidade de transmissao*/
  unsigned int sequenceNumber;   /*Numero de sequencia da trama: 0, 1*/
  unsigned int timeout;/*Valor do temporizador: 1 s*/
  unsigned int numTransmissions; /*Numero de tentativas em caso defalha*/
  status mode; /* RECEIVER || TRANSMITTER */
  struct termios portSettings;
  int wrongPackets;
}linkLayer;

linkLayer link_layer;

void set_wrong_packets(int numPackets);

void init_link_layer(int timeout,int numTransmissions, int baudRate);

int set_terminus(int fd);

int update_state(unsigned char c,int state,char * msg);

int llopen(int port,status mode);
int llopen_transmitter(int fd);
int llopen_receiver(int fd);

int llread(int fd, unsigned char *packet);
int llwrite(int fd,  char * packet, int length, int * rej_counter);

int llclose(int fd);
int llclose_transmitter(int fd);
int llclose_receiver(int fd);

unsigned char *create_Iframe(int *frame_len, char *packet, int packet_len);
unsigned char * create_Sframe(char control_byte);

unsigned char *stuff_frame( char *packet, int *packet_len);
unsigned char *destuff_frame(unsigned char *packet,  int *packet_len);
```

```
int read_packet(int fd, unsigned char *frame, int *frame_length);
int write_packet(int fd, unsigned char * buffer,int buf_length);


bool valid_Iframe(unsigned char * frame);
bool valid_sequence_number(char control_byte);
bool validBCC2(unsigned char * packet,int packet_length,unsigned char expected)
bool valid_Sframe(unsigned char *response, int response_len, unsigned char C);

bool DISC_frame(unsigned char * reply);
```

A.4 data_link.c

```
#include "data_link.h"

char SET[5] = {FLAG, A_SEND, C_SET, A_SEND ^ C_SET, FLAG};
char UA[5] = {FLAG, A_SEND, C_UA, A_SEND ^ C_UA, FLAG};
char DISC[5] ={FLAG, A_SEND, C_DISC, A_SEND ^ C_DISC, FLAG};

bool timedOut=false;
int count=0;
bool ignore_flag= false;

void alarmHandler(int sig){
  timedOut=true;
  count ++;
}

void init_link_layer(int timeout,int numTransmissions, int baudRate){

  link_layer.timeout = timeout;
  link_layer.numTransmissions = numTransmissions;
  link_layer.baudRate = baudRate;

}

int llopen(int port,status mode){

  int fd;

  link_layer.mode = mode;

  switch (port) {
    case COM1:
    strcpy(link_layer.port, COM1_PORT);
    break;

    case COM2:
    strcpy(link_layer.port, COM2_PORT);
    break;

    default:
    printf("data_link - llopen(): invalid port!\n");
    return -1;
  }

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

  if (set_terminus(fd) != 0) {
    printf("data_link - llopen() - set_terminus: error\n");
    return -1;
  }
```

```
  link_layer.mode = mode;

  if(mode == TRANSMITTER)
  if(llopen_transmitter(fd) <0)
  return -1;


  if(mode == RECEIVER)
  if(llopen_receiver(fd) < 0)
  return -1;

  link_layer.sequenceNumber =0;

  printf("llopen success!\n");
  return fd;

}

int set_terminus(int fd){

  struct termios oldtio,newtio;

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

  link_layer.portSettings = oldtio;

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


  if(link_layer.mode == TRANSMITTER){
    newtio.c_cc[VTIME]    = 5;   /* inter-character timer unused */
    newtio.c_cc[VMIN]     = 0;   /* blocking read until 5 chars received */
  }else{
    newtio.c_cc[VTIME]    = 0;   /* inter-character timer unused */
    newtio.c_cc[VMIN]     = 1;   /* blocking read until 5 chars received */
  }


  tcflush(fd, TCIOFLUSH);

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

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

  return 0;

}

int llopen_transmitter(int fd){


  unsigned char c;
  int state=0;

  signal(SIGALRM, alarmHandler);

  do{

    if(write(fd,SET,5) != 5){
      printf("data_link - llopen: error writting SET\n");
      exit(-1);
    }

    timedOut = false;
    alarm(link_layer.timeout);


    while(state!=5 && !timedOut){
      if(read(fd,&c,1)==-1){
        printf("data_link - llopen: read error\n");
        exit(-1);
      }
      state = update_state(c,state,UA);
    }
  } while(timedOut && count < link_layer.numTransmissions);

  if(count == link_layer.numTransmissions)
  return -1;
  else
  return 0;

}

int llopen_receiver(int fd){

  unsigned char c;
  int state =0;

  while(state != 5 ){

    if (read(fd,&c,1) == -1){
      printf("data_link - llopen: read error\n");
      return 1;
```

```
    }

    state = update_state(c,state,SET);

  }

  if(write(fd,UA,5) != 5){
    printf("data_link - llopen: error writting UA\n");
    exit(-1);
  }


  return 0;

}

int update_state(unsigned char c,int state,char * msg){



  switch (state) {

    case 0:
    if(c == msg[0])
    return 1;

    break;

    case 1:
    if(c == msg[1])
    return 2;

    if(c!= msg[0])
    return 0;

    break;

    case 2:
    if(c == msg[2])
    return 3;

    if(c!= msg[0])
    return 0;
    else
    return 1;

    break;

    case 3:
    if( c == (msg[2]^msg[1]))
    return 4;
```

```
      if(c!= msg[0])
      return 0;
      if( c== msg[0])
      return 1;

      break;

      case 4:
      if(c != msg[0])
      return 0;
      else
      return 5;

      break;

      default:
      break;

    }


    return 0;

}

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

    int frame_length;
    unsigned char *frame = create_Iframe(&frame_length, packet, length);

    count=0;
    unsigned char response[255];
    int response_len;


    do{
      if(write_packet(fd,frame,frame_length)<0){
        printf("Failed sending packet.\n");
        return -1;
      }
      timedOut = false;
      alarm(link_layer.timeout);

      while(!timedOut){

      if(read_packet(fd,response,&response_len)==0){

        if(valid_Sframe(response,response_len,RR)){
          alarm(0);
          link_layer.sequenceNumber =!link_layer.sequenceNumber;
          return 0;
        }
```

```
        if(valid_Sframe(response,response_len,REJ)){
          alarm(0);
          count=0;
          timedOut = true;
          }
        }
      }
    }while(timedOut && count<link_layer.numTransmissions);

    return -1;

}

bool valid_Sframe(unsigned char *response, int response_len, unsigned char C){

    if(response_len <5)
      return false;

    if(response[0]==(unsigned char)FLAG &&
    response[1]==(unsigned char)A_SEND &&
    response[3]==(unsigned char)(response[1]^response[2])&&
    response[4] == (unsigned char)FLAG&&
    ((C== RR  && response[2]==(unsigned char)(!link_layer.sequenceNumber << 7|C))
    (C== REJ && response[2]==(unsigned char)(link_layer.sequenceNumber << 7|C))))
    return true;
    else
    return false;
}

int write_packet(int fd, unsigned char * buffer,int buf_length){
    int total_chars = 0;
    int chars = 0;

    while (total_chars < buf_length) {
      chars = write(fd, buffer, buf_length);

      if( chars <= 0) {
        printf("error in write");
        return -1;
      }

      total_chars += chars;
    }
    return 0;
}

unsigned char *create_Iframe(int *frame_len, char *packet, int packet_len){

    unsigned char *stuff_packet = stuff_frame(packet, &packet_len);

    *frame_len = 5 + packet_len;
```

```
  unsigned char *frame = (unsigned char *)malloc(*frame_len * sizeof(char));

  frame[0] = FLAG;
  frame[1] = A_SEND;
  frame[2] = link_layer.sequenceNumber <<6;
  frame[3] = frame[1]^frame[2];

  memcpy(frame + 4, stuff_packet, packet_len);

  frame[*frame_len-1] = FLAG;

  return frame;

}

unsigned char *stuff_frame( char *packet, int *packet_len) {

  unsigned char* stuffed = (unsigned char *)malloc(256 * sizeof(char));

  unsigned char bcc2 = 0;
  int i = 0;
  int j = 0;

  for (i = 0; i < *packet_len; i++)
  bcc2 ^= packet[i];

  packet[*packet_len]=bcc2;
  *packet_len = *packet_len +1;

  for (i=0; i < *packet_len; i++) {

    if (packet[i] == ESC || packet[i] == FLAG) {
      stuffed[j] = ESC;
      stuffed[++j] = packet[i] ^ STUFF_BYTE;

    } else
    stuffed[j] = packet[i];
    j++;
  }

  *packet_len = j;

  return stuffed;
}

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

  unsigned char frame[MAX_SIZE];
  unsigned char * reply;
  int frame_length;
  int packet_length;
```

```
do{
    if(read_packet(fd, frame, &frame_length)<0){
      printf("data_link - llread: error reading frame\n");
      exit(-1);
    }

}while(!valid_Iframe(frame));

unsigned char expected;


if (frame[frame_length - 3] == ESC)
expected= frame[frame_length - 2] ^ STUFF_BYTE;
else
expected = frame[frame_length - 2];

// seeting actual packet size
packet_length = frame_length - HEADER_SIZE;


if (frame[frame_length - 3] == ESC)
packet_length--;


// destuffing frame and update packet value
unsigned char *destuffed = destuff_frame(frame+4, &packet_length);
memcpy(packet,destuffed , packet_length);


// check BB2
if(validBCC2(packet,packet_length,expected)){


  // check for repeated frames
  if(valid_sequence_number(frame[2])){
    link_layer.sequenceNumber = !link_layer.sequenceNumber;
  }else{
    packet_length=0; // found duplicate

  }
  reply = create_Sframe(RR);

}else{

  printf("invalid BCC2\n");
      ignore_flag =1;

  // invalid BCC2
  // check for repeated frames
  if (valid_sequence_number(frame[2])){
  reply = create_Sframe(REJ);
  }else
```

```
    reply = create_Sframe(RR);


    packet_length =0;

  }

  if(write(fd, reply, S_FRAME_LENGTH) != S_FRAME_LENGTH) {
    printf("data_link - llread: write error\n");
    return -1;
  }

  return packet_length;

}

int read_packet(int fd, unsigned char *frame, int *frame_length){

  bool STOP = false;
  char buf;
  *frame_length = 0;
  int flag_count = 0;

  while (!STOP) {
    if (read(fd, &buf, 1) >0) {
      if (buf == FLAG) {
        if(!ignore_flag){
        flag_count++;

        if(flag_count == 2)
        STOP = true;

        frame[*frame_length] = buf;
        (*frame_length)++;
      }else
        ignore_flag= false;

      }else {
        if(flag_count>0) {
          frame[*frame_length] = buf;
          (*frame_length)++;
        }
      }
    }else
    return -1;
  }

  return 0;

}

unsigned char *destuff_frame(unsigned char *packet,  int *packet_len){
```

```
  unsigned char *destuffed = (unsigned char *)malloc(((*packet_len)) * sizeof(u
  int i = 0;
  int j = 0;

  for (; i < *packet_len; i++) {
    if (packet[i] == ESC) {
      destuffed[j] = packet[i + 1] ^ STUFF_BYTE;
      i++;
    } else
    destuffed[j] = packet[i];

    j++;
  }

  *packet_len = j;

  return destuffed;
}


bool validBCC2(unsigned char * packet,int packet_length, unsigned char expected

  unsigned char actual = 0;

  int i;
  for (i = 0; i < packet_length; i++)
  actual ^= packet[i];

  return(actual == expected);

}


bool DISC_frame(unsigned char * reply){

  return(reply[0] == FLAG &&
    reply[1] == ((link_layer.mode == TRANSMITTER) ? A_RECEIVE : A_SEND)&&
    reply[2] == C_DISC &&
    reply[3] == (reply[1] ^ reply[2]) &&
    reply[4] == FLAG);

  }

  bool valid_Iframe(unsigned char * frame){

    if(frame[0] == FLAG &&
      frame[1] == A_SEND &&
      frame[3] == (frame[1] ^ frame[2]))

      return true;
```

```
      else
      return false;


    }

  unsigned char * create_Sframe(char control_byte){
      unsigned char * reply =(unsigned char *)malloc(S_FRAME_LENGTH * sizeof(ch

      reply[0] = FLAG;
      reply[1] = A_SEND;
      reply[2] = (link_layer.sequenceNumber << 7) | control_byte;
      reply[3] = reply[1]^reply[2];
      reply[4] = FLAG;



      return reply;
    }

    bool valid_sequence_number(char control_byte) {
      return (control_byte == (link_layer.sequenceNumber << 6));
    }

int llclose(int fd){


if(link_layer.mode == TRANSMITTER)
  if(llclose_transmitter(fd)<0)
    return -1;

if(link_layer.mode == RECEIVER)
  if(llclose_receiver(fd)<0)
    return -1;


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

close(fd);

return 0;

}


int llclose_transmitter(int fd){

  unsigned char c;
  int state =0;
```

```
  do{

    if(write(fd,DISC,5) != 5){
      printf("data_link - llclose: error writting DISC\n");
      return -1;
    }

    timedOut = false;
    alarm(link_layer.timeout);


    while(state!=5 && !timedOut){

      if(read(fd,&c,1)==-1){
        printf("data_link - llclose: read error\n");
        return -1;
      }

      state = update_state(c,state,DISC);

    }

  } while(timedOut && count < link_layer.numTransmissions);

  if(write(fd,UA,5) != 5){
    printf("data_link - llclose: error writting UA\n");
    exit(-1);
  }

  return 0;
}

int llclose_receiver(int fd){


    unsigned char c;
    int state =0;

    while(state != 5 ){

      if (read(fd,&c,1) == -1){
        printf("data_link - llopen: read error\n");
        return -1;
      }

      state = update_state(c,state,DISC);

    }

    if(write(fd,DISC,5) != 5){
      printf("data_link - llopen: error writting UA\n");
```

```
        return -1;
    }

    state =0;

    while(state != 5 ){

        if (read(fd,&c,1) == -1){
            printf("data_link - llopen: read error\n");
            return -1;
        }

        state = update_state(c,state,UA);

    }


    return 0;

}
```

A.5 makefile

```
all: main.c data_link.c application_layer.c data_link.h application_layer.h uti
	gcc -Wall main.c data_link.c application_layer.c -o nserial
```