– Mestrado Integrado em Engenharia Informática e Computação –

– Redes de Computadores –

Protocolo de Ligação de dados

1º Trabalho Laboratorial

6 de Novembro de 2015

Grupo:
João Ramos - ei12062@fe.up.pt
Maria Miranda – ei12046@fe.up.pt
Rui Gonçalves - ei12185@fe.up.pt

Índice

Sumário:

O presente relatório serve de apoio ao projeto “Protocolo de Ligação de Dados” avaliado no âmbito da disciplina Redes de Computadores do 3º ano do Mestrado Integrado em Engenharia Informática e computação. Este projeto consiste em transferir um ficheiro entre dois computadores diferentes, através do uso da porto de série.

Introdução

Este trabalho prático foi realizado no âmbito da unidade curricular de Redes de Computadores do 3º ano do Mestrado Integrado em Engenharia Informática e Computação, e tinha como objetivo a implementação de um protocolo de ligação de dados especificado no guião do trabalho, assim como testar esse protocolo com uma simples aplicação de transferência de ficheiros, também a desenvolver pelos alunos.

No seguimento do trabalho prático, foi-nos pedido que realizássemos um relatório, que tinha como objetivo consolidar tanto a componente prática, como a componente teórica, também inerente ao trabalho desenvolvido. Neste relatório vão ser descritas todas as partes do trabalho, divididas nas seguintes categorias:

- Arquitetura: explicação dos blocos funcionais e interfaces
- Estrutura do código: onde serão descritas as principais APIs e estruturas de dados utilizadas, bem como as principais funções e a sua relação com a arquitetura.
- Casos de uso principais: fazer a sua identificação e as sequencias de chamadas de funções
- Protocolo de ligação lógica: identificar os principais aspetos funcionais e descrever a estratégia de implementação deste aspetos.
- Protocolo da aplicação: tal como no protocolo de ligação lógico, fazer uma identificação dos principais aspetos funcionais e descrever a estratégia de implementação deste aspetos
- Validação: descrever os testes que foram efetuados, com apresentação quantificada dos resultados.
- Elementos de valorização: identificação dos elementos de valorização implementados e descrição da estratégia de implementação.

- Conclusões: síntese da informação apresentada nas secções anteriores e também uma reflexão sobre os objetivos de aprendizagem alcançados.

Arquitetura

A aplicação desenvolvida está dividida em duas grandes partes: o protocolo de ligação de dados e o protocolo da aplicação. O protocolo de ligação de dados tem como objetivo fornecer um serviço de comunicação de dados fiável entre dos sistemas ligados por um meio de transmissão, que era, no caso do nosso trabalho, um cabo série. O protocolo da aplicação tem como objetivo fazer a separação do ficheiro a ser enviado em tramas, por fazer a chamada para o envio dessas tramas através da ligação lógica e posteriormente, fazer a junção das tramas, criando assim um ficheiro igual ao enviado.

No que diz respeito à interface com o utilizador, este deve inicialmente escrever o nome do ficheiro que pretende enviar e é o único *input* que o utilizador tem na aplicação. Existem várias mensagens de sucesso e erro implementadas ao longo do programa, com destaque para: a receção das tramas de informação, o envio das mesmas, a receção das tramas de supervisão e não numeradas ou o seu envio, sempre que existe uma escrita num ficheiro... Estas mensagens existem de modo a que o utilizador tenha informação sobre o que está a acontecer no programa. A outra interface existente liga a aplicação ao protocolo de ligação de dados, esta que é explicada mais pormenorizadamente na secção a seguir.

Para uma vista mais pormenorizada da arquitetura do nosso trabalho, ver anexos 4 e 6.

Estrutura do código

O estrutura do código está dividida em 4 partes principais, como se pode ver melhor pelo anexo 4: aplicação, interface, emissor e recetor. A parte da aplicação (ficheiros fileOpp.h e app.h) contém as funções de operações de dados e de leitura e escrita no ficheiro, que são todas as funções do ficheiro fileOpp.h, e a parte da aplicação em si, que são todas as funções do ficheiro app.h.

A interface (ficheiro interface.h) entre o protocolo de ligação de dados e a aplicação é feita no ficheiro interface.h e contém as funções principais do protocolo de ligação lógica: llopen(), que é usada para abrir as portas de série; llwrite(int control, char * buf, int seqNum), usada em *loop* pelo emissor para o envio dos dados e chamada pelo recetor apenas para

enviar as mensagens de aceitação ou rejeção quando recebe dados; llread(), chamada em *loop* pelo recetor logo após começar o programa e só termina quando receber uma trama de *disc*; e llclose(), que indica o fim da transmissão, o fecho da porta de série e o fim do programa. Para além destas funções, é guardado o descritor da porta de série e o estado (se é recetor ou emissor) numa *struct* applicationLayer, e é também guardado, numa *struct* linkLayer a porta usada, o *baudrate* o número de bytes da trama atual após *byte stuffing,* o tempo de timeout, o número de retransmissões e a trama a ser enviada ou a trama que foi recebida, para uso da parte da aplicação.

Todas as funções do protocolo de ligação lógica do lado do emissor estão no ficheiro transmitter.h e todas as funções do mesmo protocolo do lado do recetor estão no ficheiro receiver.h.

Casos de uso principais

Numa compilação e execução correta do programa o que deverá acontecer é o seguinte:

1. Emissor pergunta ao utilizador por um ficheiro e guarda as informações do ficheiro numa struct;
2. Recetor e emissor abrem a porta de série;
3. Recetor chama llread() e lê as tramas que vai recebendo (inicialmente, só aceita tramas *SET*;
4. Troca de mensagens *SET* e *UA* entre recetor e emissor;
5. Emissor cria pacotes de controlo e dados;
6. Emissor envia os pacotes um a um;
7. Recetor recebe os pacotes e guarda no novo ficheiro se for pacotes de dados;
8. Recetor envia confirmação de receção;
9. Ao terminar as tramas, trocas de mensagens DISC entre recetor e emissor;
10. Termina o programa.

Cada mensagem tem um timeout definido de 2 segundos e retransmite 3 vezes em caso de erro. Após ter retransmitido 3 vezes sem resposta, o programa termina com mensagem de erro.

Nos anexos 2 e 3 estão esquemas acerca da ordem de chamada das funções do emissor e do recetor, respetivamente.

Protocolo de ligação lógica

O protocolo de ligação lógica no nosso trabalho é composto pelas funções que estão nos ficheiros interface.h, que são as principais funções, transmitter.h e receiver.h.

Após fazer a abertura da porta de série através do llopen(), o recetor usa a função llread() em *loop* para fazer todas as leituras e guarda as tramas que recebe na *struct* linkLayer em *frame*. A maneira como o llread funciona, byte a byte, está mostrada no anexo 5, num diagrama.

Através do llopen(), o emissor envia uma mensagem *SET* e fica à espera de uma mensagem *UA* durante 2 segundos. Se, após reenviar 2 vezes a mensagem não tiver recebido uma mensagem *UA*, o programa termina com mensagem de erro.

Nós usamos uma estratégia em que, na transmissão de dados, há retransmissão da mesma trama com S = 0 e S = 1 e o recetor após receber as duas, compara-as. Se forem iguais aceita a receção, senão não aceita e o emissor tenta enviar novamente.

Por parte do emissor é feito inicialmente a função de *byte stuffing* e depois, tenta enviar a informação 3 vezes com S = 0, e, se tiver sucesso, envia outra vez com S = 1.

Quando toda a informação tiver sido enviada o emissor envia uma trama *DISC* e espera uma mesma trama de *DISC* de resposta do recetor. Se receber resposta, envia uma trama *UA* e ambos terminam o programa.

Protocolo da aplicação

No emissor, protócolo de aplicação inicia-se a perguntar qual o ficheiro que se pretende enviar. Se o ficheiro não existir a aplicação retorna erro e termina o programa. Senão, os dados do ficheiro, o tamanho do ficheiro e o número de segmentos a enviar são guardados numa *struct* data no ficheiro fileOpp.h e é apenas acessível ao protocolo da aplicação.

Ambos chamam a função llopen() para poderem dar início à transferência de dados.

Ao iniciar a transmissão de dados a primeira coisa que o emissor vai fazer é criar a trama de controlo inicial, através da função *createCtrlPackets* em fileOpp.h, que é guardada na *struct* controlData e enviá-la através da função llwrite(int control, char * buf, int seqNum). A *struct* controlData é apenas usada na criação e envio de tramas de controlo por parte da aplicação. Se a transmissão anterior for bem sucedida, inicia-se um ciclo para enviar toda a

informação do ficheiro, por segmentos através da função *createDataPackets*, que ficou guardada na *struct* data. Após o ciclo é enviada a trama de controlo final, criada, mais uma vez, pela função *createCtrlPackets*. Obtendo sucesso, é chamada a função llclose() e é terminado o programa.

Por parte do recetor, este inicia um loop a chamar a função llread(). O que esta retornar irá controlar o que o recetor faz a seguir. Ao retornar 1 ou 2, é chamada a função *checkControl*, que verifica se a trama está bem estruturada e que envia uma mensagem ao emissor de sucesso, através de llwrite(). No caso de essa trama ser de controlo, são guardados na *struct* controlData os dados de controlo: nome de ficheiro e tamanho. Se a trama for de dados e o retorno tiver sido 1, a aplicação apenas guarda na *struct* linkLayer, em interface.h, essa trama para comparação futura. No entanto, se a trama for de dados e o retorno tiver sido 2, após verificação da estrutura da trama e a comparação das tramas recebidas, a aplicação guarda a informação recebida no ficheiro através da função *saveChunk*. Quando o retorno da função llread for 3, é chamada a função llclose() que termina o programa.

Validação

Para efeitos de teste do nosso trabalho, foi-nos pedido que enviássemos um ficheiro no formato GIF (penguin.gif) para verificar se este era passado do computador emissor para o computador recetor sem problemas. Este passou, sem problemas, entre os dois computadores. Seguidamente, a professora retirou o cabo de série para testar se a ligação entre os dois computadores era interrompida devidamente, o que se verificou.

Elementos de valorização

Temos implementado, do lado do recetor, o REJ, que ocorre se a trama de informação recebida não for igual nas duas vezes que recebe, em S = 0 e S = 1. Se ocorrer o emissor tenta enviar outras duas vezes as duas tramas.

Conclusões

O trabalho prático elaborado durante as últimas semanas, assim como já foi dito em nas secções acima, teve como objetivo entender como ocorre a transmissão de tramas de informação de um computador para outro, e também como estabelecer um controlo de erros fiável Stop-and-Wait, através da implementação de um protocolo de ligação de dados e de um protocolo da aplicação.

Tivemos algumas dificuldades ao longo da elaboração do projeto, nomeadamente com o controlo de erros, especialmente numa fase inicial do trabalho. Mas, à medida que avançamos com o desenvolvimento do projeto, conseguimos ter uma melhor perceção de como implementar esse controlo.

A título de conclusão, os objetivos do trabalho foram alcançados, contribuindo para uma consolidação de alguns dos conceitos abordados nas aulas teóricas e para um conhecimento mais aprofundado do funcionamento das comunicações em rede, através do uso de uma porta de série.

Anexo 1 - Código Fonte

- **app.c**

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

#include "app.h"
#include "interface.h"
#include "fileOpp.h"

int checkFrames(char * buf) {
  int i;

  for(i = 0; i < ll.sequenceNumber; i++)
      if(ll.compFrame[i] != buf[i])
          return -1;

  return 0;
}

int create_packets_send() {
      int i, timeout = 0;

      printf("llwrite: sending cp1!\n");
      while (timeout < ll.numTransmissions) {
            strcpy(ll.frame,"");
            strcpy(ctrData.frame,"");
            createCtrlPackets(0);
```

```
            if(llwrite(2,ctrData.frame,ctrData.sequenceNumber) != 0)
                  printf("llwrite:   error   sending   cp1!   Try   number
%d\n",timeout+1);
            else
                  break;
            timeout++;
      }

      if(timeout == 3) {
            printf("llwrite: timed out cp1!\n");
            return -1;
      }
      for(i = 1; i <= fileData.numSeg; i++) {
            strcpy(ll.frame,"");
            strcpy(fileData.frame,"");
            createDataPacket(i);
            timeout=0;
            while (timeout < ll.numTransmissions) {
                  if(llwrite(2,fileData.frame,    fileData.sequenceNumber)
!= 0)
                        printf("llwrite:  error  sending  dataFrame!  Try
number %d\n",timeout+1);
                  else
                        break;
                  timeout++;
            }
            if(timeout == 3) {
                  printf("llwrite: timed out data sending!\n");
                  return -1;
            }
      }

      printf("llwrite: sending cp2!\n");
      timeout=0;
      while (timeout < ll.numTransmissions) {
```

```
            strcpy(ctrData.frame,"");
            strcpy(ll.frame,"");
            createCtrlPackets(1);
            if(llwrite(2, ctrData.frame, ctrData.sequenceNumber) != 0)
                  printf("llwrite:   error   sending   cp2!   Try   number
%d\n",timeout+1);
            else
                  break;
            timeout++;

      }
      if(timeout == 3) {
            printf("llwrite: timed out cp2!\n");
            return -1;
      }

      return 0;

}

int send_rr_app(int equalize, int segmentNumber, char* buf) {
  int i = 0;

  if(equalize == 0) {
    for(i = 0; i < ll.sequenceNumber; i++)
      ll.compFrame[i] = buf[i];
      return llwrite(equalize,buf,0);
  }
  else if (equalize == 1) {
      if(checkFrames(buf) == 0) {
      if(segmentNumber > 0)
            saveChunk(buf,ll.sequenceNumber);
    }
    return llwrite(equalize,buf,0);
  }
```

```
  return -1;
}

int checkControl(int equalize) {
  int i;
  unsigned int size;
  int segmentNumber = 0;
  char buf[MAX_SIZE*2];

  if(ll.frame[0] == C_START) {
    if(ll.frame[1] == C_SIZE_FILE) {
      size = (unsigned int) ll.frame[2];
      char flength[size];
      for(i = 3; i < (3+size); i++)
        flength[i] = ll.frame[i];

      ctrData.fileLength = ( (flength[0] << 24)
                  + (flength[1] << 16)
                  + (flength[2] << 8)
                  + (flength[3] ) );
      i = 3+size;
      if(ll.frame[i] == C_NAME_FILE) {
        i++;
        size = (unsigned int) ll.frame[i];
            i++;
        int curi = i;
        for(; i < (curi+size); i++) {
          ctrData.filePath[i-curi] = ll.frame[i];
        }
        ctrData.fpLength = i-curi+1;
      }
    }
    return send_rr_app(equalize,0,"");
  }
```

```
  else if (ll.frame[0] == C_END) {
    return send_rr_app(equalize,0,"");
  }
  else if(ll.frame[0] == C_DATA) {
    segmentNumber = ll.frame[1];
    ll.sequenceNumber = ll.sequenceNumber - 4;

    for(i=0; i < ll.sequenceNumber; i++)
            buf[i] = ll.frame[i+4];

    return send_rr_app(equalize,segmentNumber,buf);
  }

  return -1;
}


int main (int argc, char ** argv) {
      ll.baudRate = 9600;
      ll.timeout = 1;
      ll.numTransmissions = 3;

      int port = 0;
      int flag = 0;
      int read_ret = 0;

      if (argc < 3) {
            printf("Usage:\t./app PortNumber flag\n\tex: ./app 5 1\n");
            exit(1);
      }

      port = atoi(argv[1]);
      flag = atoi(argv[2]);

      if(port > 5 || port < 0) {
```

```
        printf("Port number must be between 0 and 5!\n");
        exit(1);
}

if(flag != 0 && flag != 1) {
        printf("Flag must be 0 or 1!\n");
        exit(1);
}

if(flag==0) {
        printf("Write the name of the file you want to copy: ");
        strcpy(ctrData.filePath,"");
        gets(ctrData.filePath);
        ctrData.fpLength = strlen(ctrData.filePath);
}

char sPort[20];

strcpy(sPort,"");
sprintf(sPort, "/dev/ttyS%d",port);

appLayer.status = flag;
strcpy(ll.port,sPort);

if(flag == 0)
        if(readData() != 0){
                printf("\nError: File doesn't exist!\n\n");
                return -1;
        }

if(llopen() == -1) {
        printf("Error llopen!\n");
        exit(1);
}
```

```
	if(appLayer.status == 0) {
		if(create_packets_send() == -1){
			printf("Error llwrite!\n");
			exit(1);
		}
	}
	else {
		int control = 0;
		while(TRUE) {
			read_ret = llread(control);

			if (read_ret == 1) {
				checkControl(read_ret-1);
				control = 1;
			}
			else if(read_ret == 2) {
				checkControl(read_ret-1);
			}
			else if(read_ret == 3) {
				if(llclose() != 0) {
					printf("\n\nError: llclose()!\n\n");
					exit(-1);
				}
			}
		}
	}

	llclose();

	return 0;
}
```

- **app.h**

```
#ifndef APP_H
```

```
#define APP_H

int checkFrames(char * buf);
int create_packets_send();
int send_rr_app(int equalize, int segmentNumber, char* buf);
int checkControl(int equalize);
int main (int argc, char ** argv);

#endif
```

- **fileOpp.c**

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include "fileOpp.h"

int readData() {
	FILE *my_file;
	unsigned int i;

	my_file = fopen (ctrData.filePath, "r");
	if (my_file == NULL) {
		printf ("File not found!\n");
		printf ("Or filepath is wrong!\n");
		return -1;
	}

	printf("Reading file and saving into struct data\n");
```

```
	fseek (my_file, 0, SEEK_END);
	fileData.dataLength = ftell (my_file);
	ctrData.fileLength = fileData.dataLength;
	fseek (my_file, 0, SEEK_SET);

	fileData.data = (unsigned char *) malloc (sizeof(unsigned char) *
fileData.dataLength);

	if(fileData.dataLength % MAX_SIZE_DATA == 0)
		fileData.numSeg = fileData.dataLength/MAX_SIZE_DATA;
	else
		fileData.numSeg = fileData.dataLength/MAX_SIZE_DATA + 1;

	printf("Num segments: %d\n", fileData.numSeg);
	printf("Length of file: %ld bytes\n", fileData.dataLength);

	for (i = 0; i < fileData.dataLength; i++)
		fileData.data[i] = getc(my_file);

	fclose (my_file);

	return 0;
}

int createCtrlPackets(int control) {
	int count = 0;
	int i = 0;

	unsigned char lengthValue[4];
	lengthValue[0] = (int)((fileData.dataLength >> 24) & 0xFF) ;
	lengthValue[1] = (int)((fileData.dataLength >> 16) & 0xFF) ;
	lengthValue[2] = (int)((fileData.dataLength >> 8) & 0XFF);
	lengthValue[3] = (int)((fileData.dataLength & 0XFF));
```

```
	if(control == 0) ctrData.frame[count] = C_START;
	else
		ctrData.frame[count] = C_END;

	count++;
	ctrData.frame[count] = C_SIZE_FILE;
	count++;
	ctrData.frame[count] = SIZE;
	count++;

	for(i = 0; i < 4; i++) {
		ctrData.frame[count] = lengthValue[i];
		count++;
	}

	ctrData.frame[count] = C_NAME_FILE;
	count++;
	ctrData.frame[count] = (unsigned char) ctrData.fpLength;
	count++;

	for(i = 0; i < ctrData.fpLength; i++) {
		ctrData.frame[count] = ctrData.filePath[i];
		count++;
	}

	ctrData.sequenceNumber = count;
	return 0;
}

int createDataPacket(int segment) {
	int count = 0;
	int i;
	unsigned int size = 0;
```

```
	if(segment == fileData.numSeg)
		size = fileData.dataLength % MAX_SIZE_DATA;
	else
		size = MAX_SIZE_DATA;

	unsigned char L1 = (short) size%256,L2 = (short) size/256;

	fileData.frame[count] = C_DATA;
	count++;

	printf("createDataPackets: segment: %d\n",segment);

	fileData.frame[count] = (unsigned char) segment;

	count++;
	fileData.frame[count] = L2;
	count++;
	fileData.frame[count] = '\0';
	fileData.frame[count] = (char) L1;
	count++;

	for(i = 0; i < size ; i++) {
		fileData.frame[count]	=	fileData.data[i+(segment-1)	*
MAX_SIZE_DATA];
		count++;
	}

	fileData.sequenceNumber = count;
	return 0;
}

int saveChunk(char* buf, int sequenceNumber) {
	FILE *my_file;
	unsigned int i;
```

```
      my_file = fopen (ctrData.filePath, "a");

    if(my_file == NULL) {
            printf("Creating and writing on new file with name:
%s\n",ctrData.filePath);
            my_file = fopen(ctrData.filePath,"w+");
    }
    else
      printf("Updating file with name: %s\n",ctrData.filePath);


      for(i = 0; i< sequenceNumber; i++)
            putc(buf[i],my_file);

      fclose(my_file);

      return 0;
}
```

- **fileOpp.h**

```
#ifndef FILEOPP_H
#define FILEOPP_H

#define C_START 0x01
#define C_END 0x02
#define C_SIZE_FILE 0x00
#define C_NAME_FILE 0x01
#define SIZE 0x04
#define C_DATA 0x00

#define MAX_SIZE 256
#define MAX_SIZE_DATA 230
```

```
struct data{

	unsigned long dataLength;
	unsigned char * data;
	unsigned int numSeg;
	char frame[MAX_SIZE*2];
	unsigned int sequenceNumber;

};

struct controlData{

	unsigned long fileLength;
	char filePath[MAX_SIZE_DATA];
	unsigned int fpLength;
	char frame[MAX_SIZE*2];
	unsigned int sequenceNumber;

};

int readData();
int createCtrlPackets(int control);
int createDataPacket(int segment);
int saveChunk(char* buf, int sequenceNumber);

struct data fileData;
struct controlData ctrData;

#endif
```

- **interface.c**

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

```c
#include <termios.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

#include "transmitter.h"
#include "receiver.h"
#include "interface.h"

int setNum = 0;

int llopen() {

	if(appLayer.status == 0) {
		printf("llopen: opening ports and sending SET bytes!\n");
		return saveConfig();
	}

	printf("llopen: opening ports!\n");
	return saveConfigNC();
}

int llread(int control) {

    int state = 0;
    int equalize = 0;
    int res = 0;
    int count_buf = 0;
    char buf[MAX_SIZE];
    char ant[1];

    //RECEIVE INF
    strcpy(buf,"");
```

```
    while(TRUE) {
          res = read(appLayer.fd,buf,1);

        switch(state) {
            case 0:
                  if(buf[res-1] == FLAG)
                   state++;
                count_buf = 0;
                ll.sequenceNumber = 0;
            break;

            case 1:
                if(buf[res-1] == A_SEND)
                    state++;
                else if(buf[res-1] == FLAG)
                    break;
                else
                    state=0;
                strcpy(ll.frame,"");
                count_buf = 0;
                ll.sequenceNumber = 0;
                ant[0] = '\0';
            break;

            case 2:
                if(buf[res-1] == C_SET)
                    state++;
                else if(buf[res-1] == C_SI && setNum == 1)
                    state = 5;
                  else if(buf[res-1] == C_SF && control == 1 && setNum ==
1)

                    state=5;
                else if (buf[res-1] == C_DISC)
                    state=8;
```

```
    else if(buf[res-1] == FLAG)
        state=1;
    else
        state=0;
break;

case 3:
    if(buf[res-1] == (A_SEND^C_SET))
        state++;
    else if (buf[res-1] == FLAG)
         state = 1;
    else
        state = 0;
break;

case 4:
    if (buf[res-1] == FLAG) {
        printf("Received SET!\n");
        state = 0;
          setNum = 1;
        return send_ua();
    }
    else
        state = 0;
break;

case 5:
    if(buf[res-1] == (C_SI^A_SEND)) {
        state = 6;
        equalize = 0;
    }
    else if(buf[res-1] == (C_SF^A_SEND)) {
        state = 6;
        equalize = 1;
    }
```

```
    else if (buf[res-1] == FLAG)
        state = 1;
    else
        state = 0;
    ant[0] = buf[res-1];
break;

case 6:
    if(buf[res-1] == FLAG)
        state = 1;
    else if(buf[res-1] == AFT_FLAG) {
          if(ant[0] == ESC) {
            ll.frame[count_buf] = FLAG;
            count_buf++;
             ant[0] = '\0';
          }
          else {
            ll.frame[count_buf] = AFT_FLAG;
            count_buf++;
            ant[0] = AFT_FLAG;
          }
    }
    else if(buf[res-1] == AFT_ESC){
          if(ant[0] == ESC){
            ll.frame[count_buf] = ESC;
            count_buf++;
                ant[0] = '\0';
          }
          else {
                ll.frame[count_buf] = AFT_ESC;
            count_buf++;
            ant[0] = AFT_ESC;
          }
    }
    else if(buf[res-1] == ESC) {
```

```
                    ant[0] = ESC;
                    break;
                }
                    else if(buf[res-1] == (C_SI^A_SEND) || buf[res-1] ==
(C_SF^A_SEND)){
                    state = 7;
                    ant[0] = buf[res-1];
                }
                else {
                    ll.frame[count_buf] = buf[res-1];
                    count_buf++;
                    ant[0] = buf[res-1];
                }
            break;

            case 7:
                if(buf[res-1] == FLAG){
                          printf("Successfully destuffed bytes and saved
message! %d bytes\n", count_buf);
                    ant[0]='\0';
                    state = 0;
                    ll.sequenceNumber = count_buf;
                      return equalize+1;
                }
                else{
                    ll.frame[count_buf] = ant[0];
                    count_buf++;
                    ll.frame[count_buf] = buf[res-1];
                    ant[0]=buf[res-1];
                    count_buf++;
                    state=6;
                }
            break;

            case 8:
```

```
                if(buf[res-1] == (A_SEND^C_DISC))
                    state=9;
                else if (buf[res-1] == FLAG)
                    state = 1;
                else
                    state = 0;
            break;

            case 9:
                if(buf[res-1] == FLAG){
                    printf("Sending DISC and shutting down!\n");
                    return 3; //send_disc_nc();
                }
                else
                    state = 0;
            break;

            default:
                printf("Error llread: state is not between 0 and 9!\n");
            break;
        }
  }

  return -1;
}

int llwrite(int control, char * buf, int seqNum) {

      switch(control) {
            case 0:
                  return send_rr(control,buf);
                  break;

            case 1:
                  return send_rr(control,buf);
```

```
                break;

            case 2:
                return prepare_inf(buf, seqNum);
                break;
      }
      return -1;
}

int llclose() {

      if(appLayer.status == 0)
            return prepare_send_disc();

      send_disc_nc();
      closeConfigNC();

      return 0;
}
```

- **interface.h**

```
#ifndef INTERFACE_H
#define INTERFACE_H

#define _POSIX_SOURCE 1 // POSIX compliant source
#define FALSE 0
#define TRUE 1

#define MAX_SIZE 256
#define MAX_SIZE_DATA 230


struct applicationLayer {
```

```
      int fd;     /*Descritor correspondente à porta série*/
      int status; /*TRANSMITTER | RECEIVER*/

};

struct linkLayer {

      char port[20];    /*Dispositivo /dev/ttySx, x = 0, 1*/
      int baudRate;     /*Velocidade de transmissão*/
      unsigned int sequenceNumber;   /*Número de bytes após stuffing */
      unsigned int timeout;  /*Valor do temporizador: 1 s*/
      unsigned int numTransmissions;     /*Número de tentativas em caso
de falha*/
      char frame[MAX_SIZE*2];/*Trama*/
      char compFrame[MAX_SIZE*2]; /*Trama para comparar*/ //TESTAR

};


int checkFrames(char*buf);

int llopen();
int llread(int control);
int llwrite(int control, char * buf, int seqNum);
int llclose();

struct applicationLayer appLayer;
struct linkLayer ll;



#endif
```

- **receiver.c**

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

#include "interface.h"
#include "receiver.h"

unsigned int segmentNumber;

int receive_ua_nc() {
    int res;
    char buf[MAX_SIZE];
    int count = 0;

    strcpy(buf,"");

    while (1) {
        res = read(appLayer.fd,buf,1);

        switch(count) {
            case 0:
            if(buf[res-1] == FLAG)
               count++;
            break;

            case 1:
            if(buf[res-1] == A_SEND)
               count++;
            else if(buf[res-1] == FLAG)
```

```
            break;
        else
            count=0;
        break;

        case 2:
        if(buf[res-1] == C_UA)
            count++;
        else if(buf[res-1] == FLAG)
            count=1;
        else
            count=0;
        break;

        case 3:
        if(buf[res-1] == (C_UA^A_SEND))
            count++;
        else if(buf[res-1] == FLAG)
            count=1;
        else
            count=0;
        break;

        case 4:
        if(buf[res-1] == FLAG) {
            printf("Received UA and closing down!\n");
            return 0;
        }
        else
            count=0;
        break;
    }
}

return -1;
```

```
}

int send_disc_nc() {
    char DISC[5];
    DISC[0] = FLAG;
    DISC[1] = A_REC;
    DISC[2] = C_DISC;
    DISC[3] = DISC[1]^DISC[2];
    DISC[4] = FLAG;
    write(appLayer.fd,DISC,6);

    receive_ua_nc();
    return 1;
}



int send_rr(int equalize, char* buf) {
    char RR[5];

    RR[0] = FLAG;
    RR[1] = A_REC;

    if(equalize == 0)
        RR[2] = C_RRI;
    else if (equalize == 1) {
        if(checkFrames(buf) == 0) {
            RR[2] = C_RRF;
        }
        else{
            RR[2] = C_REJ;
        }
    }

    RR[3] = RR[1]^RR[2];
```

```
    RR[4] = FLAG;

    printf("send_rr: Sending rr number %d\n\n", equalize);
    write(appLayer.fd,RR, 6);

    return 0;
}


int send_ua() {
    unsigned char UA[5];
    UA[0] = FLAG;
    UA[1] = A_REC;
    UA[2] = C_UA;
    UA[3] = UA[1]^UA[2];
    UA[4] = FLAG;

    //SEND UA
    write(appLayer.fd,UA,5);
    printf("Sent response (UA)!\n\n");

    return 0;
}


int saveConfigNC() {
    appLayer.fd = open(ll.port, O_RDWR | O_NOCTTY );

    if (appLayer.fd <0) {
        perror(ll.port);
        exit(-1);
    }

     if ( tcgetattr(appLayer.fd,&oldtio) == -1) { /* save current port
settings */
```

```
        perror("tcgetattr");
        exit(-1);
    }

    //RECEIVE SET
    return newConfigNC();
}


int newConfigNC() {
    struct termios newtio;

    bzero(&newtio, sizeof(newtio));
    newtio.c_cflag = ll.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) proximo(s) caracter(es)
    */

    tcflush(appLayer.fd, TCIOFLUSH);

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

```
    }

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

    return 0;
}

int closeConfigNC() {
    printf("Ending program!\n");

    sleep(5);

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

    close(appLayer.fd);

    exit(0);
}
```

- **receiver.h**

```
#ifndef RECEIVER_H
#define RECEIVER_H


#define FLAG 0x7e
#define ESC 0x7d
#define AFT_ESC 0x5d
#define AFT_FLAG 0x5e
#define A_SEND 0x03   //sender comand
#define A_REC 0x01  // receiver command
#define C_SET 0x07
```

```
#define C_UA 0x03
#define C_SI 0x00
#define C_SF 0x20
#define C_RRI 0x21
#define C_RRF 0x01
#define C_REJ 0x25
#define C_DISC 0x0b


struct termios oldtio;

int receive_ua_nc();
int send_disc_nc();
int send_rr(int equalize, char * buf);
int send_ua();
int receive_inf(int control);
int saveConfigNC();
int newConfigNC();
int closeConfigNC();

#endif
```

- **transmitter.c**

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

#include "interface.h"
```

```c
#include "transmitter.h"

volatile int STOP_REC=FALSE;

int re_send = 0, re_send_ti = 0, re_send_tf = 0, flag_al = 1;

void alarm_handler() {
    flag_al = 1;
    re_send++;
}

int send_final_ua() {
    unsigned char UA[5];
    UA[0] = FLAG;
    UA[1] = A_SEND;
    UA[2] = C_UA;
    UA[3] = UA[1]^UA[2];
    UA[4] = FLAG;

    //SEND UA

    write(appLayer.fd,UA,6);
    printf("Sent response!\n");

    return 0;
}

int prepare_send_final_ua() {
    printf("\nWaiting and then sending FINAL UA\n");
    sleep(3);
    re_send = 0;

    while (re_send < ll.numTransmissions) {
        if(flag_al) {
            alarm(ll.timeout);
```

```
            printf("SENDING UA n%d!\n",(re_send+1));
            flag_al=0;
            send_final_ua();
        }
        else
            break;
    }

    return closeConfig();
}

int receive_disc() {
    char buf[1];
    int res;
    int count = 0;
    printf("Receiving DISC...\n");

    while (1) {
        res = read(appLayer.fd,buf,1);

        if(res == 0)
            return -1;

        switch(count) {
            case 0:
            if(buf[res-1] == FLAG)
                count++;
            break;

            case 1:
            if(buf[res-1] == A_REC)
                count++;
            else if(buf[res-1] == FLAG)
                break;
            else
```

```
                count = 0;
            break;

            case 2:
            if(buf[res-1] == C_DISC)
                count++;
            else if(buf[res-1] == FLAG)
                count=1;
            else
                count = 0;
            break;

            case 3:
            if(buf[res-1] == (A_REC^C_DISC))
                count++;
            else if(buf[res-1] == FLAG)
                count=1;
            else
                count = 0;
            break;
            case 4:
            if(buf[res-1] == FLAG)
                return 0;
            else
                count = 0;
            break;
        }
    }

    return -1;
}


int send_disc() {
    char DISC[5];
```

```
    DISC[0] = FLAG;
    DISC[1] = A_SEND;
    DISC[2] = C_DISC;
    DISC[3] = DISC[1]^DISC[2];
    DISC[4] = FLAG;
    write(appLayer.fd,DISC, 6);

    return receive_disc();
}

int prepare_send_disc() {

    int returnInt;

    printf("Waiting and then sending DISC\n");
    sleep(3);
    re_send = 0;

    while (re_send < ll.numTransmissions) {
        if(flag_al) {
            alarm(ll.timeout);
            printf("SENDING DISC n%d!\n",(re_send+1));
            flag_al=0;

            if((returnInt=send_disc()) != 0)
                    printf("Error: no acknowledgment received from data
sending (DISC)!\n");
            else
                break;
        }
    }

    return prepare_send_final_ua();
}
```

```
int byte_stuffing(char* seq, int seqNum) {
    int count1 = 0;
    int count2 = 4;

    strcpy(ll.frame,"");

    ll.frame[0] = FLAG;
    ll.frame[1] = A_SEND;
    ll.frame[2] = C_SI;
    ll.frame[3] = ll.frame[1]^ll.frame[2];

    while (count1 < seqNum) {

        if(seq[count1] == FLAG) {
            ll.frame[count2] = ESC;
            count2++;
            ll.frame[count2] = 0x5e;
            count2++;
        }
        else if(seq[count1] == ESC) {
            ll.frame[count2] = ESC;
            count2++;
            ll.frame[count2] = 0x5d;
            count2++;
        }
        else{
            ll.frame[count2] = seq[count1];
            count2++;
        }
        count1++;
    }

    ll.frame[count2] = ll.frame[3];
    ll.frame[count2+1] = FLAG;
    printf("Stuffed %d bytes!\n",count2+2);
```

```
    ll.sequenceNumber = count2+2;

    return 0;
}


int receive_RR(int control) {

      char buf[1];
      int res;
      int count = 0;

    if(control == 0) {
        while (1) {
              res = read(appLayer.fd,buf,1);

              if(res == 0)
                return -1;

            switch(count) {
                case 0:
                if(buf[res-1] == FLAG)
                    count++;
                break;

                case 1:
                if(buf[res-1] == A_REC)
                    count++;
                else if(buf[res-1] == FLAG)
                    break;
                else
                    count=0;
                break;

                case 2:
```

```
            if(buf[res-1] == C_RRI)
                count++;
            else if(buf[res-1] == FLAG)
                count=1;
            else
                count=0;
            break;

            case 3:
            if(buf[res-1] == (C_RRI^A_REC))
                count++;
            else if(buf[res-1] == FLAG)
                count=1;
            else
                count=0;
            break;

            case 4:
            if(buf[res-1] == FLAG)
                return 0;
            else
                count=0;
            break;
        }
    }
}

else if(control == 1) {
    while (1) {
        res = read(appLayer.fd,buf,1);

        if(res == 0)
            return -1;

        switch(count){
```

```
case 0:
if(buf[res-1] == FLAG)
    count++;
break;

case 1:
if(buf[res-1] == A_REC)
    count++;
else if(buf[res-1] == FLAG)
    break;
else
    count=0;
break;

case 2:
if(buf[res-1] == C_RRF)
    count++;
else if(buf[res-1] == FLAG)
    count=1;
else
    count=0;
break;

case 3:
if(buf[res-1] == (C_RRF^A_REC))
    count++;
else if(buf[res-1] == FLAG)
    count=1;
else
    count=0;
break;

case 4:
if(buf[res-1] == FLAG)
    return 0;
```

```
                else
                    count=0;
                break;
            }
        }
    }

    return -1;
}


int send_inf(int control) {
    printf("Sending %d bytes!\n",ll.sequenceNumber);
    write(appLayer.fd,ll.frame,ll.sequenceNumber);

    return receive_RR(control);
}


int prepare_inf(char* inf, int seqNum) {
    byte_stuffing(inf, seqNum);
    int returnInt = -2;
    printf("Waiting and then sending TI\n");
    sleep(ll.timeout);
    re_send=0;

    while (re_send < ll.numTransmissions) {
        if(flag_al) {
            alarm(ll.timeout);
            printf("SENDING DATA n%d!\n",(re_send+1));
            flag_al=0;
            if((returnInt=send_inf(0)) != 0) {
                    printf("Error: no acknowledgment received from data
sending (TI)!\n");
            }
```

```
            else
                break;
        }
    }

    if(returnInt != 0 )
        printf("Error during data sending\n");
    else {
        printf("Waiting and then sending TF\n");
        sleep(ll.timeout);
        ll.frame[2] = C_SF;
        ll.frame[3] = C_SF^A_SEND;
        ll.frame[ll.sequenceNumber-2] = ll.frame[3];
        re_send = 0;

        while (re_send < ll.numTransmissions) {
            if(flag_al) {
                alarm(ll.timeout);
                printf("SENDING DATA n%d!\n",(re_send+1));
                flag_al=0;

                if((returnInt=send_inf(1)) != 0) {
                      printf("Error: no acknowledgment received from data
sending (TF)!\n");
                    if(returnInt == 1) {
                          printf("Data received was not the same, trying
again!\n");
                        return prepare_inf(inf,seqNum);
                    }
                }
                else
                    break;
            }
        }
    }
```

```
    if(returnInt == 0)
        printf("Sent information data successfully!\n\n");

    return returnInt;
}

int receive_ua() {
    int res;
    char buf[MAX_SIZE];
    int count = 0;

    strcpy(buf,"");

    while (STOP_REC == FALSE) {
        res = read(appLayer.fd,buf,1);

        if(res == 0)
            return -1;

        switch(count) {
            case 0:
            if(buf[res-1] == FLAG)
                count++;
            break;

            case 1:
            if(buf[res-1] == A_REC)
                count++;
            else if(buf[res-1] == FLAG)
                break;
            else
                count=0;
            break;
```

```
            case 2:
            if(buf[res-1] == C_UA)
                count++;
            else if(buf[res-1] == FLAG)
                count=1;
            else
                count=0;
            break;

            case 3:
            if(buf[res-1] == (C_UA^A_REC))
                count++;
            else if(buf[res-1] == FLAG)
                count=1;
            else
                count=0;
            break;

            case 4:
            if(buf[res-1] == FLAG)
                return 0;
            else
                count=0;
            break;
        }
    }

    return -1;
}


int send_set() {
    unsigned char SET[5];
    SET[0] = FLAG;
    SET[1] = A_SEND;
```

```
    SET[2] = C_SET;
    SET[3] = SET[1]^SET[2];
    SET[4] = FLAG;

    //SEND SET
    write(appLayer.fd,SET,6);
    printf("Sent SET\n");

    //RECEIVE UA
    return receive_ua();
}


int prepare_set() {

    int returnInt;
    while (re_send < ll.numTransmissions) {
        if(flag_al) {
            alarm(ll.timeout);
            printf("SENDING n%d!\n",(re_send+1));
            flag_al=0;

            if((returnInt=send_set()) != 0){
                printf("Error: no acknowledgment received (UA)!\n");
            }
            else
                break;
        }
    }

    return returnInt;
}

int saveConfig() {
```

```
    (void) signal(SIGALRM, alarm_handler);
    appLayer.fd = open(ll.port, O_RDWR | O_NOCTTY );

    if (appLayer.fd <0) {
        perror(ll.port);
        exit(-1);
    }

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

    return newConfig();
}

int newConfig() {

    struct termios newtio;
    bzero(&newtio, sizeof(newtio));
    newtio.c_cflag = ll.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] = ll.timeout * 10;    /* 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) proximo(s) caracter(es)
    */
```

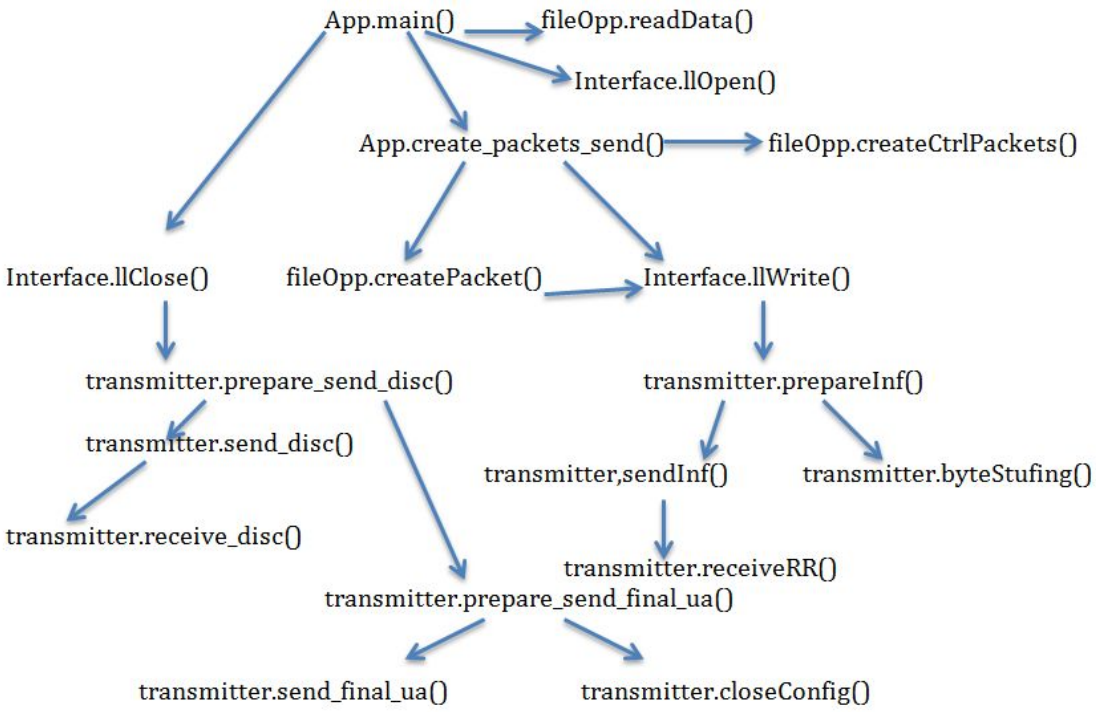
```
    tcflush(appLayer.fd, TCIOFLUSH);
    if ( tcsetattr(appLayer.fd,TCSANOW,&newtio) == -1) {
        perror("tcsetattr");
        exit(-1);
    }

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


int closeConfig() {

    printf("Ending program!\n");
    sleep(5);

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

    close(appLayer.fd);

    return 0;
}
```

- **transmitter.h**

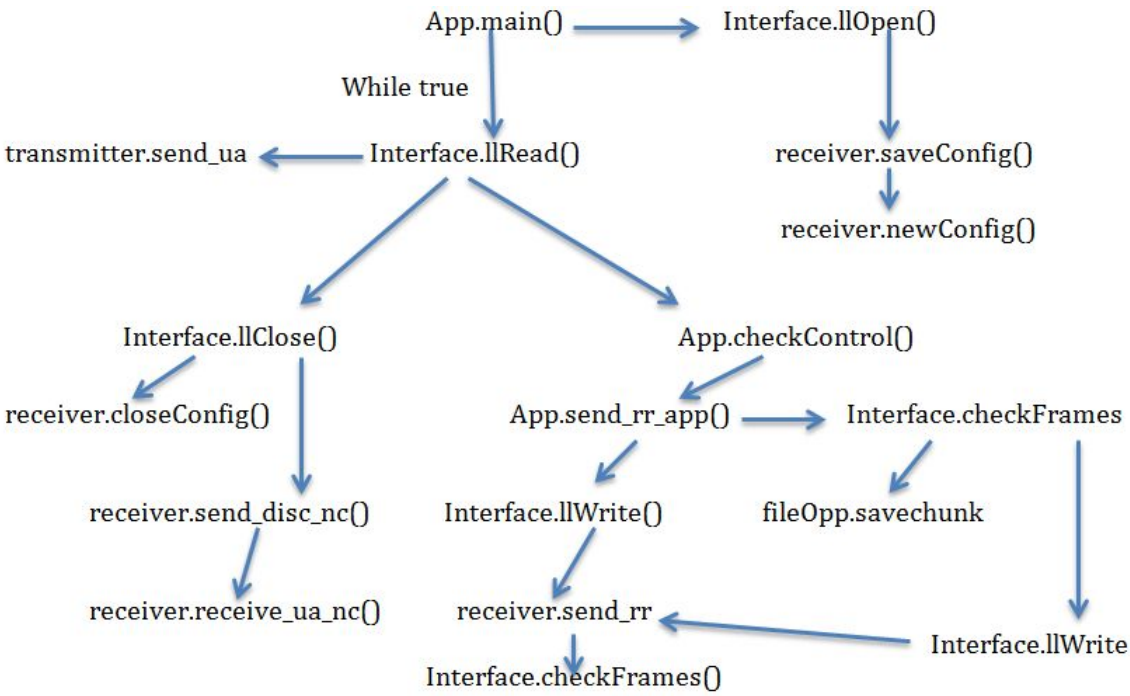
```
#ifndef TRANSMITTER_H
#define TRANSMITTER_H

#define FLAG 0x7e
#define ESC 0x7d
#define AFT_ESC 0x5d
```

```
#define AFT_FLAG 0x5e
#define A_SEND 0x03   //sender comand
#define A_REC 0x01  // receiver command
#define C_SET 0x07
#define C_UA 0x03
#define C_SI 0x00
#define C_SF 0x20
#define C_RRI 0x21
#define C_RRF 0x01
#define C_REJ 0x25
#define C_DISC 0x0b

struct termios oldtio;

int send_final_ua();
int receive_disc();
int send_disc();
int prepare_send_disc();
int byte_stuffing(char* seq,int seqNum);
int receive_RR(int control);
int send_inf(int control);
int prepare_inf(char* inf, int seqNum);
int send_set();
int receive_ua();
int prepare_set();
int saveConfig();
int newConfig();
int closeConfig();


#endif
```

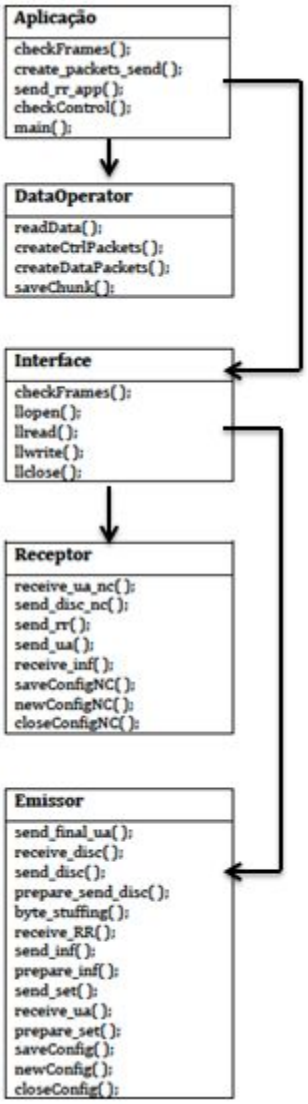
Anexo 2 - Emissor



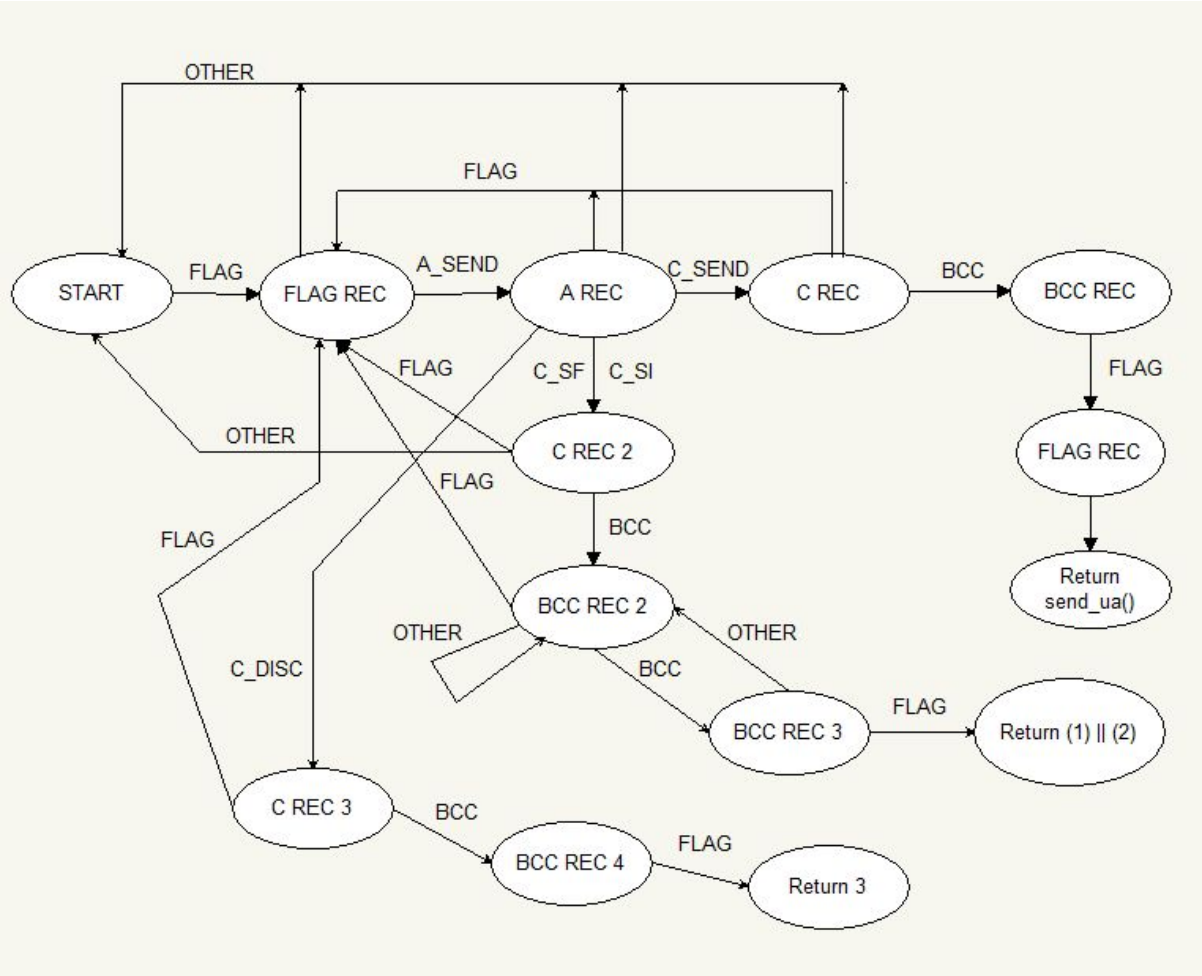
Anexo 3 - Recetor



Anexo 4 - Arquitetura



Anexo 5 - Diagrama llread()



Anexo 6 - Arquitetura Geral