U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Relatório

1º Trabalho Laboratorial

Redes de Computadores

T06_G08

André Pereira – up201905650

Matilde Oliveira – up201906954

11/12/2021

Contents

Introdução

O presente trabalho tem como objetivo a implementação de um protocolo de ligação de dados que permita a transferência de dados entre dois sistemas. O objetivo é que o mesmo seja testado com recurso a uma aplicação que permita a transferência de ficheiros também por nós implementada.

O relatório que aqui apresentamos, servirá de apoio e descrição da implementação do protocolo e aplicação, bem como apresentará conclusões sobre a sua eficiência e validação.

Este documento seguirá a seguinte estrutura:

Arquitetura: blocos funcionais e interfaces.

Estrutura do Código: APIs, principais estruturas de dados, funções e relação com arquitetura.

Casos de uso: identifição dos principais e relação com arquitetura.

Protocolo de ligação lógica: descrição da estratégia, principais aspetos funcionais e implementação.

Protocolo de aplicação: descrição da estratégia, principais aspetos funcionais e implementação.

Validação: descrição de testes efetuados.

Eficiência do protocolo de ligação de dados: análise da eficiência do protocolo.

Arquitetura

De forma a resolver o problema proposto procuramos dividir o código em dois módulos **emissor** e **recetor**.

Dentro de cada um destes módulos o código encontra-se separado em duas camadas **aplicação** e **protocolo de transferência de dados**.

Estrutura do código

O código desenvolvido encontra-se divido em 4 ficheiros. Dentro da **camada da aplicação** temos o *emissor.c* e *recetor.c* que contém as funções utilizadas pela aplicação respetiva. Na camada do **protocolo de transferência de dados** apresentamos os restantes dois ficheiros o *dataLinkEmissor.c* que contem funções relevantes para transferir informação e o *dataLinkRecetor.c* com funções relevantes para receber informação ambas ao nível do protocolo de ligação de dados.

Principais estruturas de dados:

```
typedef struct {
    unsigned char *packet;
    int pSize;
    int sequenceNumber;
    enum PacketState packetState;
} AppPacket;
```

```
typedef struct {
    unsigned char * frame;
    int sizeFrame;
} Frame;
```

AppPacket – Usado ao nível da aplicação, para guardar o pacote recebido e a enviar.

Frame – Usado ao nível do protocolo de transferência de dados para guardar a trama recebida e a enviar.

Principais funções:

emissor.c

buildPacket() – Constrói o pacote de dados a partir, neste caso, do conteúdo do ficheiro lido e passado como parâmetro.

buildPacketStart() – Constrói o pacote de controlo inicial.

buildPacketStart() – Constrói o pacote de control final.

recetor.c

parsePacket() – Analisa o pacote recebido seja ele de dados ou de controlo.

dataLinkEmissor.c

dataLinkState() – máquina de estados que verifica se os bytes recebidos estão de acordo com o protocolo definido e as configurações corretas de tramas.

llopenEmissor() – inicia a comunicação enviando a trama de supervisão SET e recebendo a trama UA.

buildFrame() – constrói a trama a enviar e realiza o byte stuffing da mesma.

llwrite() – envia a trama para a porta de série e aguarda resposta analisando a trama recebida tomando decisões quanto à trama a enviar a seguir.

llclose() – termina a comunicação enviando a trama de supervisão DISC e espera por resposta afirmativa que se acontecer leva ao envio de UA.

dataLinkRecetor.c

llopenRecetor() – procura iniciar a comunicação da porta série aguardando o envio da trama de supervisão SET e respondendo com a trama UA.

dataLinkState() – máquina de estados que verifica se os bytes recebidos estão de acordo com o protocolo definido e as configurações corretas de tramas.

destuff() – realiza o byte destuffing da trama recebida.

llread() – lê a trama da porta de serie e envia consequente resposta RR ou REJ com o respetivo número de sequência.

llclose() – termina a comunicação esperando pela receção da trama de supervisão DISC e envia a trama DISC como resposta.

Casos de uso principais

Chamada do programa emissor

Permite ao utilizador escolher o ficheiro a enviar bem como escolher a porta de série que deseja utilizar para fazer a transferência do ficheiro "*emissor <serialPort> <filePath>*".

Chamada do programa recetor

Permite ao utilizador escolher a porta de série pela qual deseja realizar o download do ficheiro "*recetor <serialPort>*".

Sequência de chamadas de funções

A transmissão de dados ocorre entre dois processos, sendo a sequencia geral de eventos a seguinte:

1. Emissor abre o ficheiro a enviar e a porta de série enviando uma mensagem para começar a transferência.
2. Recetor abre a porta de série e recebe a mensagem.
3. Emissor vai lendo e enviando os dados, trama a trama.
4. Recetor vai recebendo e escrevendo no ficheiro os dados recebidos.
5. Emissor envia trama de finalização e termina.
6. Recetor recebe trama de finalização e comunicação termina.

Protocolo de ligação lógica

O **protocolo de ligação lógica** que implementamos é constituído pelas quatro funções principais que servem de interface para o protocolo de aplicação utilizar as funcionalidades da camada de ligação lógica. Cada função *llopen()*, *llwrite()*, *llread()* e *llclose()*, como indicado no guião do trabalho, trabalha mediante a aplicação em que está (emissor e recetor) e conforme a fase do protocolo de ligação de dados onde se encontram. Em ambos os programas, a leitura de qualquer trama é feita através de uma **máquina de estados**. Esta recebe a trama byte a byte e executa mudanças de estado conforme o byte lido ser o correto e o esperado, de modo a que apenas se chega ao estado final se a trama recebida tiver um formato válido para a fase onde o processo se encontra. A máquina de estados está na função *dataLinkState()* e os estados são representados pela enumeração *FrameState* (apresentados em baixo).

```
enum FrameState {
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    BCC_OK,
    END
};
```

```
/**
 * @brief depending on the global state of the protocol where the program this is
 * the state machine that deals with the reception of a frame and confirms the configuration and integrity is correct
 *
 * @param data - byte read from serial port
 * @param frameState - frame state of the recption frame state machine (configuration steps)
 * @param globalState - phase of the data link protocol (Establish, Data transfer and End)
 * @param frame - to be filled with the right frame configuration
 * @return int - 0 upon sucess
 */
int dataLinkState(unsigned char data, enum FrameState *frameState, int globalState, Frame * frame);
```

llopen()

Em ambos os programas, emissor e recetor, esta função começa por abrir a ligação à porta de série, definir as configurações necessárias e criar o respetivo descritor de ficheiro. No emissor, começa também por ativar o alarme de timeout, responsável por reenviar tramas no caso de perda de informação na porta de série.

De seguida, dependendo da aplicação que chamou a função, a função auxiliar *establish()* começa por:

- No emissor, enviar uma trama de supervisão SET (*sendControlFrame()*) e fica à espera de ler (*receiveMessage()*) a trama de supervisão UA da porta de série.
- No recetor, fica à espera de receber uma trama de supervisão SET (*receiveMessage()*) e quando a recebe envia a trama de supervisão UA (*sendControlFrame()*).

llwrite()

Esta função apenas está presente na aplicação emissora, de modo a enviar tramas de informação através da porta de série.

1. Começa por encapsular o pacote recebido da aplicação numa trama de informação, na função *buildFrame()*, que também é responsável pelo stuffing da trama. Tendo em conta qual o número de sequência da trama (S0 e S1) anteriormente enviada, esta função também indica qual o número de sequência correto para a próxima a ser enviada e que está a ser construída.
2. Envio da trama de informação pela porta de série em *sendMessage()* que guarda qual a trama que foi enviada.
3. Espera por trama de supervisão de resposta, verificando se a configuração está correta aquando da receção.
4. Enquanto a resposta recebida não for um RR com o número de sequência esperado (receção de REJ ou RR com número de sequência não coerente), então é efetuado o reenvio da trama, quantas vezes forem necessárias para obter estes dois em conformidade e terminar a função.

Se em alguma das situações de envio ou receção de tramas houver perda de informação o alarme é ativado e reenvia a trama anteriormente enviada.

```
buildFrame(data, dataSize, &frameBackup);

do{
    frameResponseState = START;
    frameResponse.sizeFrame = 0;
    //send frame to be sent
    sendMessage(fd, frameBackup.frame, frameBackup.sizeFrame);

    //receive and confirm frame sent by recetor
    while(frameResponseState != END){
        receiveMessage(fd,buf);
        dataLinkState(buf[0], &frameResponseState, TRANSFER, &frameResponse);
        printf("%x ",buf[0]);
    }

    usleep(waitTime);

    if(frameResponse.frame[2] == C_RR1 && frameSequenceNumber == 0){
        printf("accepted RR1\n");
        end = TRUE;
        frameSequenceNumber = 1;
    }else if(frameResponse.frame[2] == C_RR0 && frameSequenceNumber == 1){
        printf("accepted RR0\n");
        end = TRUE;
        frameSequenceNumber = 0;
    }else if(frameResponse.frame[2] == C_REJ1){
        printf("accepted REJ1\n");
        frameSequenceNumber = 1;
    }else if(frameResponse.frame[2] == C_REJ0){
        printf("accepted REJ0\n");
        frameSequenceNumber = 0;
    }

}while( !end );
```

llread()

Esta função apenas está presente na aplicação recetora, de modo a receber tramas de informação através da porta de série.

1. Espera por receção de uma trama de informação através da porta de série, na função *receiveMessage()*, verificando à chegada se a configuração vai de encontro ao esperado em *dataLinkState()*.
2. Efetua o destuffing da trama de informação recebida e confirma o valor do BCC2, na função *destuff()*.
3. Se o destuffing não for bem sucedido, então envia trama de supervisão de REJ, com o número de sequência que está a espera de receber. Se este for bem sucedido e a trama recebida tiver o número de sequência em concordância com a que o recetor esperava receber, envia trama de supervisão RR com número de sequência contrário. Caso, o número de sequência recebido não for o que o recetor estava à espera, então reenvia RR com o número de sequência igual ao que estava à espera.

```
while( !end ){
    frameResponseState = START;
    frameResponse.sizeFrame = 0;

    while(frameResponseState != END){
        receiveMessage(fd,buf);
        dataLinkState(buf[0], &frameResponseState, TRANSFER, &frameResponse);
        printf("%x ",buf[0]);
    }

    usleep(waitTime);

    if(destuff(&frameResponse,data,dataSize) == -1){
        if(frameSequenceNumber == 0){
            sendControlFrame(fd, C_REJ0);
        }else if(frameSequenceNumber == 1){
            sendControlFrame(fd, C_REJ1);
        }
    }else if(frameResponse.frame[2] == C_S0){
        if(frameSequenceNumber == 0){
            frameSequenceNumber = 1;
            sendControlFrame(fd, C_RR1);
            end = TRUE;
        } else if(frameSequenceNumber == 1){
            sendControlFrame(fd, C_RR1);
        }
    }else if(frameResponse.frame[2] == C_S1){
        if(frameSequenceNumber == 1){
            frameSequenceNumber = 0;
            sendControlFrame(fd, C_RR0);
            end = TRUE;
        } else if(frameSequenceNumber == 0){
            sendControlFrame(fd, C_RR0);
        }
    }
}
```

llclose()

Esta função, dependendo da aplicação que chamou a função, age de acordo com a fase de terminação do protocolo de ligação de dados:

- No emissor, envia uma trama de supervisão DISC (*sendControlFrame()*) e fica à espera de ler (*receiveMessage()*) a trama de supervisão DISC, corretamente configurada, da porta de série. Termina com o envio final da trama de supervisão UA.
- No recetor, fica à espera de receber uma trama de supervisão DISC (*receiveMessage()*) e quando a recebe envia a trama de supervisão DISC (*sendControlFrame()*).

Em ambos os programas, emissor e recetor, esta função acaba depois de fechar a ligação à porta de série.

Protocolo de aplicação

O protocolo de aplicação tem em cada um dos módulos a função *appFunction()* que é chamada no main de cada um dos programas e define a estratégia de cada máquina.

Emissor

A aplicação do lado do emissor segue o seguinte conjunto de ações:

1. Abrir ficheiro a ser lido e enviado;

2. Estabelecimento da ligação e comunicação com o recetor, através da função *llopenEmissor()*, parte da interface Protocolo-Aplicação.
3. Construção e envio do primeiro pacote de início de transferência de dados (primeiro com a função *buildPacketStart()*, e depois recorrendo a *llwrite()*, membro da interface Protocolo-Aplicação).
4. Leitura de um número fixo de dados, neste caso 100 bytes, do ficheiro a ser transmitido, na função *readFromFile()*.
5. Construção e envio de pacote com os dados lidos, mantendo o número de sequência atualizado (primeiro com a função *buildPacket()*, e depois recorrendo a *llwrite()*, membro da interface Protocolo-Aplicação).
6. Repetição dos pontos 4 e 5 até a leitura do ficheiro retornar 0 bytes, quando a aplicação construirá o pacote final e envia-lo-á (primeiro com a função *buildPacketEnd()*, e depois recorrendo a *llwrite()*, membro da interface Protocolo-Aplicação).
7. Com o fim do envio do ficheiro, é terminada a ligação, através da função *llclose()*, parte da interface Protocolo-Aplicação.

```
fd = llopenEmissor(port);

while(appPacket.packetState != P_END){

    if(appPacket.packetState == P_START){
        buildPacketStart(finalName, &appPacket);
        appPacket.packetState = P_DATA;
    }else if(appPacket.packetState == P_DATA){
        bufSize = readFromFile(fileFd, buf);
        if(bufSize == 0){
            appPacket.packetState = P_END;
            printf("End of file reach\n");
            buildPacketEnd(&appPacket);
        }
        else if(bufSize == -1){
            printf("Error reading file\n");
            return -1;
        }else {
            buildPacket(buf, bufSize, &appPacket);
        }
    }

    int n = llwrite(fd, appPacket.packet , appPacket.pSize);
}

llclose(fd);
```

Recetor

A aplicação do lado do recetor segue o seguinte conjunto de ações:

1. Estabelecimento da ligação e comunicação com o emissor, através da função *llopenRecetor()*, parte da interface Protocolo-Aplicação.
2. Receção de informação proveniente da camada de ligação lógica, através da função *llread()*, parte da interface Protocolo-Aplicação.

3. Análise do pacote recebido para obtenção do campo de dados, através da função *parsePacket()*:

   3.1. se se tratar de pacote inicial, abre novo ficheiro com o nome enviado no pacote;
   3.2. se se tratar de pacote de informação, escreve no ficheiro e atualiza número de sequência;
   3.3. se se tratar de pacote final, fecha o ficheiro construído e termina ciclo de receção de pacotes.
4. Com o fim da receção e construção do ficheiro, é terminada a ligação, através da função *llclose()*, parte da interface Protocolo-Aplicação.

```
fd = llopenRecetor(port);

while(appPacket.packetState != P_END){ //when we receive the end packet
    llread(fd, appPacket.packet, &appPacket.pSize);
    parsePacket(&fileFd, &appPacket);
}

llclose(fd);
```

Validação

Para garantir o correto funcionamento do nosso programa os seguintes testes foram realizados:

- Envio de ficheiros com diferentes tamanhos (x e 120000 bytes).
- Interrupção da ligação da porta de série.
- Geração de ruído na ligação da porta de série.

Estes três foram testados em aula e situação de avaliação. Os próximo serão avaliados em baixo e tiveram como base os seguintes testes:

Nota: os valores de referência que mantivemos constantes, apenas variando no respetivo teste, se este existir, foram: **Imagem** de 120000 **bytes**. **Baudrate** de 38400 **bit/s**. **Tamanho do pacote** de 100 **bytes**. Percentagem de **erro** simulado de **0%**. Simulação do **tempo de propagação** de 0 **ms**.

- Envio de ficheiros com diferentes tamanhos de pacotes de dados.

Tamanho do pacote (bytes)	Tempo médio de execução (s)
50	3.058067
100	2.646067
250	2.608833
500	2.536433
1000	2.297567

- Envio de ficheiros com erros simulados.

Simulação de erro (%)	Tempo médio de execução (s)
0%	2.646067
0.50%	36.0145
1%	86.015
2%	136.007

- Envio de ficheiros com diferentes valores de baudrate.

Baudrate (bit/s)	Tempo médio de execução (s)
4800	2.814527
9600	2.864667
19200	2.837233
38400	2.646067

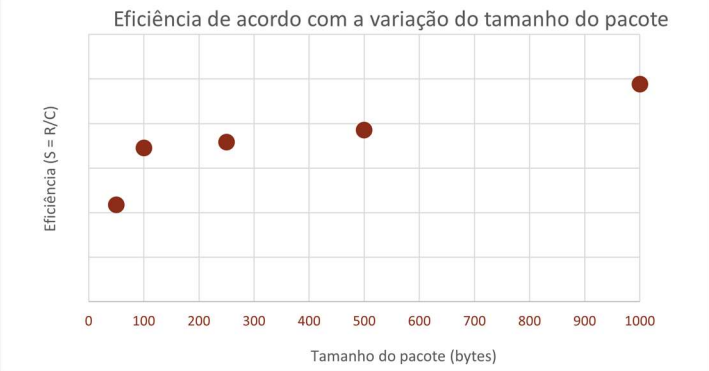
- Envio de ficheiros com diferentes tempos de propagação de pacotes.

Tempo de propagação (ms)	Tempo médio de execução (s)
0	2.646067
50	2.8494
100	3.06595
200	2.9539
500	3.675

Eficiência do protocolo de ligação de dados

Variação da eficiência relativamente ao tamanho do pacote de dados

O aumento do tamanho do pacote de dados desde 50, 100, 250, 500 até 1000 bytes mostra-nos que quanto maior o tamanho de cada pacote de dados, mais eficiente será o programa. A possível justificação para este resultado é a de que, como cada envio contém um maior conjunto de informação, para a transferência do mesmo ficheiro, o número de tramas a enviar reduz. Levando a que o programa e a transferência ocupem mais, o canal dedicado para realizar a comunicação, tornando-se mais eficiente e rápido. Outra conclusão será a de que com menos tramas de informação a enviar o time out ocorrerá relativamente menos vezes.



Variação da eficiência relativamente ao FER

Através do gráfico apresentado abaixo, podemos concluir que a introdução de erros simulados (ainda que em percentagens baixas) têm um grande impacto no valor da eficiência do programa. A introdução de erros no BCC1 têm um impacto significativo, visto que obrigará o sistema a reenviar a trama outra vez, após receber uma interrupção do alarme.

Variação da eficiência relativamente a diferentes valores de BaudRate (Capacidade de ligação)

Este gráfico apresentado em baixo comprova que com o aumento da capacidade da ligação a eficiência diminui como prova a fórmula de S = R/C.

Variação da eficiência relativamente a diferentes tempos de propagação

Como esperavamos, o tempo de propagação que se traduz numa melhor eficiência é quando este é nulo. Assim, com o aumento do tempo de propagação, o programa torna-se menos eficiente. Isto deve-se principalmente ao facto de a aplicação passar mais tempo sem enviar tramas, já que o envio e sucessiva receção demora mais tempo.

Comparação com protocolo Stop&Wait

Comparando o programa desenvolvido a um protocolo de *Stop&Wait,* podemos afirma que, na teoria, o comportamento implementado é o mesmo. Sempre que é enviada alguma informação o programa aguarda pela receção de um *ACK*, enviado pela parte to recetor, de modo a continuar a sua execução. Caso a trama envia sofra alguma perturbação, invalidando a informação nela presente, o processo recetor terá de responder com um *NACK* de modo a informar que ocorreu um erro durante a transferência da trama. O emissor ao receber esta resposta tem o dever de retransmitir a trama enviada anteriormente, podendo apenas continuar caso um ACK seja recebido. Da mesma maneira, caso a resposta esteja a demorar mais tempo que o previsto a chegar, o programa tem o dever de reenviar a mensagem. Este sistema de *timeout* previne ocasiões em que seja perdida informação enquanto esta está a ser transferida. De forma ao emissor saber a que trama se refere o *ACK/NACK* recebido, ambos a trama e o *ACK/NACK* são constituídos por um bit 0 ou 1, que desempenham um sistema de numeração. Estes números deveram ser alternados por cada trama de informação enviada. Adicionalmente esta estrategia permite ao recetor saber se a trama recebida é duplica.

Na nossa aplicação utilizamos um protocolo com base em *Stop&Wait* de forma a minizar os possíveis erros. As tramas de informação que enviamos estão, portanto, identificadas com um 0 ou 1. Em relação á resposta esta deverá ser um *RR* (*receiver ready*) caso a mensagem tenha sido bem recebida, equivalente ao *ACK*. Para uma resposta com erro deverá ser enviado um *REJ*, equivalente ao *NACK*. Ambas estas respostas, dependendo da trama de informação recebida, fazem-se acompanhadas do bit de identificação 0 ou 1.

Conclusão

Síntese

Este projeto consistiu no desenvolvimento de um serviço de comunicação entre duas máquinas ligadas através de uma porta série, implementando um protocolo de ligação de dados, apresentado e explicado ao longo deste relatório.

Reflexão

Para além de uma maior compreensão sobre a implementação e funcionamento de um protocolo de ligação de dados, este trabalho permitiu entender a importância e relevância do conceito e aplicabilidade da independência de camadas abordado em contexto de aula teórica. Embora inicialmente este funcionamento nos tenha feito um pouco de confusão, estamos certos de que o trabalho cumpre agora todas as regras neste sentido, assim como nos momentos de encapsulamento e envio/receção de informação.

Dado que são pedidos excertos de código para complementar a explicação de certas implementações do trabalho, o número de páginas final ultrapassa o pedido.

Anexo I - Código fonte

dataLinkEmissor.h

```
#ifndef DATALINKEMISSOR_H_
#define DATALINKEMISSOR_H_

enum FrameState {
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    BCC_OK,
    END
};

typedef struct {
    unsigned char * frame;
    int sizeFrame;
} Frame;

/**
 * @brief handles with alarm calls by sending the last message again
 *
 */
void alarmCall();

/**
 * @brief iniciates the communication of the serial port and sets its
configurations
 *
 * @param port - name of the serial port
 * @return int - 0 upon sucess
 */
int openSerialPort(char * port);

/**
 * @brief sends frame through the serial port and updates last message
sent with the one sent
 *
 * @param fd - serial port file descriptor
 * @param message - frame to be sent
 * @param size - message size in bytes
 * @return int - number of bytes written in fd
 */
int sendMessage(int fd, unsigned char * message, unsigned size);

/**
 * @brief reads message from serial port
 *
 * @param fd - serial port file descriptor
 * @param buf - to be completed with 1 bytes read
 * @return int - number of bytes read
 */
int receiveMessage(int fd,unsigned char * buf);
```

```
/**
 * @brief builds the control frame acording to configuration
 *
 * @param fd - file descriptor of the serial port
 * @param control - specific control flag
 * @return int - 0 upon sucess
 */
int sendControlFrame(int fd,unsigned char control);

/**
 * @brief depending on the global state of the protocol where the
program this is
 * the state machine that deals with the reception of a frame and
confirms the configuration and integrity is correct
 *
 * @param data - byte read from serial port
 * @param frameState - frame state of the recption frame state machine
(configuration steps)
 * @param globalState - phase of the data link protocol (Establish,
Data transfer and End)
 * @param frame - to be filled with the right frame configuration
 * @return int - 0 upon sucess
 */
int dataLinkState(unsigned char data, enum FrameState *frameState, int
globalState, Frame * frame);

/**
 * @brief represents the first fase of the data link protocol
 * where the sender sends a Set frame and waits for an UA one
 *
 * @return int - 0 upon sucess
 */
int establish();

/**
 * @brief initiates the alarm configs, opens serial port
 * and iniciates the data link protocol
 *
 * @param port - serial port path
 * @return int - file descriptor of serial port
 */
int llopenEmissor(char * port);

/**
 * @brief builds the frame to be sent with the data from the higher
layer
 * executing the byte stuffing on the frame
 *
 * @param data - information received from the higher layer to be put
in the information field of the frame
 * @param size - data size in bytes
 * @param frameBackup - information frame to be filled with the data
given and with the correct configuration
 * @return int - 0 upon sucess
 */
int buildFrame(unsigned char * data, int size, Frame *frameBackup);
```

```
/**
 * @brief sends frame through serial port waits for transmitter
response
 * and handles with rejection and receiver ready flags choosing which
frame to send next
 *
 * @param fd - file descriptor of the serial port
 * @param data - data received from higher layer
 * @param dataSize - data size in bytes
 * @return int - number of bytes of the frame sent
 */
int llwrite(int fd, unsigned char* data, int dataSize);

/**
 * @brief sends to the serial port the C_DISC supervision frame
indicating the end of the transfering process
 * receives the C_DISC from the transmitter and sends last frame C_UA
that ends the communication
 * closes the serial port at the end
 *
 * @param fd - file descriptor of the serial port
 * @return int - 0 upon sucess
 */
int llclose(int fd);

#endif // DATALINKEMISSOR_H_
```

dataLinkEmissor.c

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

#include "dataLinkEmissor.h"
#include "../VAR.h"

int waitTime = 0;

unsigned int alarmCalls = 0, flag = 1;

unsigned char lastMessage[2047];
unsigned int lastMessageSize = 0;
```

```
int fd; // Serial port file descriptor

unsigned int frameSequenceNumber = 0; // last frame sent switch between
0 and 1

struct termios oldtio,newtio;

void alarmCall(){                    // atende alarme
      printf("alarme # %d\n", alarmCalls);
      flag=1;
      alarmCalls++;
    sendMessage(fd,lastMessage,lastMessageSize);
}

int openSerialPort(char * port) {
    /*
        Open serial port device for reading and writing and not as
controlling tty
        because we don't want to get killed if linenoise sends CTRL-C.
    */

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


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

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

    /* set input mode (non-canonical, no echo,...) */
    newtio.c_lflag = 0;

    newtio.c_cc[VTIME]    = 0;   /* inter-character timer unused */
    newtio.c_cc[VMIN]     = 1;   /* blocking read until 5 chars
received */


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

    tcflush(fd, TCIOFLUSH);

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

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

    return 0;
}

int sendMessage(int fd, unsigned char * message, unsigned size){
    int res;

    if(flag){
      alarm(4);                 // activa alarme de 4s
      flag=0;
    }
    for(int i = 0; i <= size; i++){
        lastMessage[i] = message[i];
    }
    lastMessageSize = size;
    res = write(fd,message,size);
    printf("%d bytes written\n", res);
    return res;
}

int receiveMessage(int fd,unsigned char * buf){
    return read(fd,buf,1);
}

int sendControlFrame(int fd,unsigned char control){
    unsigned char CONTROL[5];

    CONTROL[0] = FLAG;
    CONTROL[1] = AE_SENT;
    CONTROL[2] = control;
    CONTROL[3] = CONTROL[1]^CONTROL[2]; //xor
    CONTROL[4] = FLAG;

    sendMessage(fd,CONTROL,5);

    return 0;
}

int dataLinkState(unsigned char data, enum FrameState *frameState, int
globalState, Frame * frame){
    switch(*frameState){
        case START:
            if(data == FLAG){
                *frameState = FLAG_RCV;
                frame->frame[frame->sizeFrame]=data;
                frame->sizeFrame = frame->sizeFrame + 1;
            }
            break;
        case FLAG_RCV:
            if(data == FLAG){
                *frameState = FLAG_RCV;
                frame->frame[0] = data;
                frame->sizeFrame = 1;
            }
            else if(data == AE_SENT){
                *frameState = A_RCV;
```

```
                frame->frame[frame->sizeFrame] = data;
                frame->sizeFrame = frame->sizeFrame+1;
            }
            else{
                *frameState=START;
                frame->sizeFrame = 0;
            }
            break;
        case A_RCV:
            if(data == FLAG){
                *frameState = FLAG_RCV;
                frame->frame[0] = data;
                frame->sizeFrame = 1;
            }
            else if(globalState == ESTABLISH && data == C_UA){
                *frameState=C_RCV;
                frame->frame[frame->sizeFrame]=data;
                frame->sizeFrame = frame->sizeFrame+1;
            }
            else if(globalState == TERMINATE && data == C_DISC){
                *frameState=C_RCV;
                frame->frame[frame->sizeFrame]=data;
                frame->sizeFrame = frame->sizeFrame+1;
            }else if(globalState == TRANSFER && (data == C_REJ0 || data
== C_REJ1 || data == C_RR1 || data == C_RR0)){
                *frameState=C_RCV;
                frame->frame[frame->sizeFrame]=data;
                frame->sizeFrame = frame->sizeFrame+1;
            }
            else{
                *frameState=START;
                frame->sizeFrame = 0;
            }
            break;
        case C_RCV:
            if(data == FLAG){
                *frameState = FLAG_RCV;
                frame->frame[0]=data;
                frame->sizeFrame = 1;
            }
            else if(data == frame->frame[1]^frame->frame[2]){
                *frameState = BCC_OK;
                frame->frame[frame->sizeFrame] = data;
                frame->sizeFrame = frame->sizeFrame+1;
            }
            else{
                *frameState = START;
                frame->sizeFrame = 0;
            }
            break;
        case BCC_OK:
            if(data==FLAG){
                *frameState=END;
                frame->frame[frame->sizeFrame]=data;
                frame->sizeFrame = 0;
            }
            else{
                *frameState=START;
```

```
                frame->sizeFrame = 0;
            }
            break;
        case END:
            break;
    }
    return 0;
}

int establish(){
    unsigned char buf[2047];
    enum FrameState frameState = START;

    Frame frameResponse;
    frameResponse.frame = malloc (2047 * sizeof (unsigned char));
    frameResponse.sizeFrame = 0;

    sendControlFrame(fd, C_SET);

    //Read control frame sent by receiver (C_UA)
    while(frameState != END){
        receiveMessage(fd,buf);
        dataLinkState(buf[0], &frameState, ESTABLISH, &frameResponse);
        printf("%x ",buf[0]);
    }

    usleep(waitTime);

    free(frameResponse.frame);
    return 0;
}

int llopenEmissor(char * port){

    (void) signal(SIGALRM, alarmCall);

    if(openSerialPort(port)!= 0){
        perror("openSerialPort");
        exit(-1);
    }

    if(establish()!= 0){
        perror("establish");
        exit(-1);
    }

    return fd;
}

int buildFrame(unsigned char * data, int size, Frame *frameBackup){
    unsigned char *FRAME = malloc (2047 * sizeof (unsigned char));
    unsigned int currFrame = 0;
    unsigned char bcc;

    FRAME[0] = FLAG;
    FRAME[1] = AE_SENT;
    if(frameSequenceNumber == 0){
       FRAME[2] = C_S0;
```

```
    } else if (frameSequenceNumber == 1){
       FRAME[2] = C_S1;
    }
    FRAME[3] = FRAME[1]^FRAME[2]; //xor
    currFrame = 4;

    //start byte stuffing
    for(int i = 0; i < size; i++){
        if(i == 0){
            bcc = data[i];
        }
        else {
           bcc = bcc ^ data[i];
        }

        if(data[i] == FLAG){
            FRAME[currFrame] = O_ESC;
            currFrame++;
            FRAME[currFrame] = O_FST;
            currFrame++;
        }
        else if ( data[i] == O_ESC){
            FRAME[currFrame] = O_ESC;
            currFrame++;
            FRAME[currFrame] = O_SND;
            currFrame++;
        }
        else {
            FRAME[currFrame] = data[i];
            currFrame++;
        }
    }

    if(bcc == FLAG){
        FRAME[currFrame] = O_ESC;
        currFrame++;
        FRAME[currFrame] = O_FST;
        currFrame++;
    }
    else if (bcc == O_ESC){
        FRAME[currFrame] = O_ESC;
        currFrame++;
        FRAME[currFrame] = O_SND;
        currFrame++;
    } else {
        FRAME[currFrame] = bcc;
        currFrame++;
    }
    //end byte stuffing

    FRAME[currFrame] = FLAG;
    currFrame++;

    frameBackup->frame = FRAME;
    frameBackup->sizeFrame = currFrame;

    return 0;
}
```

```
//retorna o numero de caracteres escritos
int llwrite(int fd, unsigned char* data, int dataSize){
    int end = FALSE;
    //frame to be built and sent
    Frame frameBackup;

    //frame to be received and confirmed
    unsigned char buf[2047];
    enum FrameState frameResponseState = START;
    Frame frameResponse;
    frameResponse.frame = malloc (2047 * sizeof (unsigned char));
    frameResponse.sizeFrame = 0;

    //frame to be built
    buildFrame(data, dataSize, &frameBackup);

    do{
        frameResponseState = START;
        frameResponse.sizeFrame = 0;
        //send frame to be sent
        sendMessage(fd, frameBackup.frame, frameBackup.sizeFrame);

        //receive and confirm frame sent by recetor
        while(frameResponseState != END){
            receiveMessage(fd,buf);
            dataLinkState(buf[0], &frameResponseState, TRANSFER,
&frameResponse);
            printf("%x ",buf[0]);
        }

        usleep(waitTime);

        if(frameResponse.frame[2] == C_RR1 && frameSequenceNumber ==
0){
            printf("accepted RR1\n");
            end = TRUE;
            frameSequenceNumber = 1;
        }else if(frameResponse.frame[2] == C_RR0 && frameSequenceNumber
== 1){
            printf("accepted RR0\n");
            end = TRUE;
            frameSequenceNumber = 0;
        }else if(frameResponse.frame[2] == C_REJ1){
            printf("accepted REJ1\n");
            frameSequenceNumber = 1;
        }else if(frameResponse.frame[2] == C_REJ0){
            printf("accepted REJ0\n");
            frameSequenceNumber = 0;
        }

    }while( !end );

    free(frameBackup.frame);
    free(frameResponse.frame);

    return frameBackup.sizeFrame;
}
```

```
int llclose(int fd){
    unsigned char buf[2047];
    enum FrameState frameState = START;

    Frame frameResponse;
    frameResponse.frame = malloc (2047 * sizeof (unsigned char));
    frameResponse.sizeFrame = 0;

    sendControlFrame(fd, C_DISC);

    //Read control frame sent by receiver (C_UA)
    while(frameState != END){
        receiveMessage(fd,buf);
        dataLinkState(buf[0], &frameState, TERMINATE, &frameResponse);
        printf("%x ",buf[0]);
    }

    usleep(waitTime);

    sendControlFrame(fd, C_UA);

    alarm(0);
    free(frameResponse.frame);


    sleep(2);

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

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

Emissor.h

```
#ifndef EMISSOR_H_
#define EMISSOR_H_

enum PacketState {
    P_START,
    P_DATA,
    P_END
```

```
};

typedef struct {
    unsigned char *packet;
    int pSize;
    int sequenceNumber;
    enum PacketState packetState;
} AppPacket;

/**
 * @brief builds the start control packet
 * that is going to be send with the read fileName
 *
 * @param fileName - name of the file to be sent and created by the
transmitter
 * @param appPacket - packet to be completed with the start packet
configurations
 * @return int - 0 upon sucess
 */
int buildPacketStart(unsigned char * fileName, AppPacket * appPacket);

/**
 * @brief copies the information received in the parameter buf
 * to the packet to be sent and completes the packet with the
configuration needed
 *
 * @param buf - information read from file to complete the data field
of the packet
 * @param bufSize - buf number of bytes
 * @param appPacket - paket to be completed with the information from
buf and other configurations
 * @return int - 0 upon sucess
 */
int buildPacket(unsigned char * buf, int bufSize, AppPacket *
appPacket);

/**
 * @brief builds the ending control packet
 * with the configuration needed to determine the ending of the
transfer process
 *
 * @param appPacket - packet to be filled with the specific
configuration
 * @return int - 0 upon sucess
 */
int buildPacketEnd(AppPacket * appPacket);

/**
 * @brief fills buf with the information read from file (100 bytes)
 *
 * @param fileFd - fd from file to be read
 * @param buf - array to be filled with the read info
 * @return int - number of bytes read
 */
int readFromFile(int fileFd,unsigned char *buf);

/**
```

```
 * @brief controls the app cycle that connects the app with the data
link protocol
 * by reading data from file and sending it to the lower layer
 *
 * @param port - serial port path
 * @param fileName - file name of the file that will be trasnfered
 * @return int - 0 upon sucess
 */
int appFunction(char *port, char *fileName);

#endif // EMISSOR_H_
```

Emissor.c

```
//sudo socat -d  -d  PTY,link=/dev/ttyS0,mode=777
PTY,link=/dev/ttyS1,mode=777

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

#include "emissor.h"
#include "dataLinkEmissor.h"
#include "../VAR.h"

//unsigned char *fileName = "./pinguim1.gif";
//unsigned char *fileNameOrg = "./pinguim.gif";

int buildPacketStart(unsigned char * fileName, AppPacket * appPacket){
    appPacket->packet[0] = 0x02;
    appPacket->packet[1] = 0x01;
    appPacket->packet[2] = strlen(fileName) + 1;
    for (int i = 0; i <  strlen(fileName); i++){
        appPacket->packet[i + 3] = fileName[i];
    }
    appPacket->packet[strlen(fileName) + 3] = '\0';
    appPacket->pSize = strlen(fileName) + 4;

    appPacket->sequenceNumber = 0;
    return 0;
}
```

```
int buildPacket(unsigned char * buf, int bufSize, AppPacket *
appPacket){
    appPacket->packet[0] = 0x01;
    appPacket->packet[1] = appPacket->sequenceNumber;
    appPacket->packet[2] = (0x0ff00 & bufSize) >> 2;
    appPacket->packet[3] = 0xff & bufSize;
    for(int i = 0; i < bufSize; i++){
        appPacket->packet[i+4] = buf[i];
    }

    appPacket->pSize = bufSize + 4;
    appPacket->sequenceNumber = appPacket->sequenceNumber + 1;
    return 0;
}

int buildPacketEnd(AppPacket * appPacket){
    appPacket->packet[0] = 0x03;
    appPacket->pSize = 1;
    return 0;
}

int readFromFile(int fileFd,unsigned char *buf){
    return read(fileFd,buf,100);
}

int appFunction(char *port, char *fileName){
    int fileFd;
    fileFd = open(fileName, O_RDONLY);
    if (fileFd < 0) {perror("Erro while opening test.txt"); exit(-1);}

    char finalName[255] = "clone_";

    strtok(fileName, "/");

    strcat(finalName, strtok(NULL, "/"));


    int fd, bufSize;
    AppPacket appPacket;
    appPacket.pSize = 0;
    appPacket.sequenceNumber = 0;
    appPacket.packetState = P_START;
    appPacket.packet = malloc (2047 * sizeof (unsigned char));

    unsigned char buf[2047];

    fd = llopenEmissor(port);

    while(appPacket.packetState != P_END){

        if(appPacket.packetState == P_START){
            buildPacketStart(finalName, &appPacket);
            appPacket.packetState = P_DATA;
        }else if(appPacket.packetState == P_DATA){
            bufSize = readFromFile(fileFd, buf);
            if(bufSize == 0){
                appPacket.packetState = P_END;
                printf("End of file reach\n");
```

```
                buildPacketEnd(&appPacket);
            }
            else if(bufSize == -1){
                printf("Error reading file\n");
                return -1;
            }else {
                buildPacket(buf, bufSize, &appPacket);
            }
        }

        int n = llwrite(fd, appPacket.packet , appPacket.pSize);
//returns number of chars written
    }

    llclose(fd);
}

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

    if (( argc < 3 )){
        printf("A file must be specifed\n");
        exit(1);
    }

    struct timeval start, end;
    double elapsedTime;
    clock_t startP, endP;

    startP = clock();
    gettimeofday(&start, NULL);

    //start sending data
    if (appFunction(argv[1], argv[2]) != 0){
        perror("communication error");
        exit(-1);
    }

    gettimeofday(&end, NULL);

    elapsedTime = (end.tv_sec - start.tv_sec) * 1000.0;
    elapsedTime += (end.tv_usec - start.tv_usec) / 1000.0;

    endP = clock();

    printf("Sender execution time - %f\n",elapsedTime * 1.0e-3);
    printf("Sender process execution time - %f\n",((double) (endP -
startP)) / CLOCKS_PER_SEC);
    return 0;
}
```

dataLinkRecetor.h

```
#ifndef DATALINKRECETOR_H_
#define DATALINKRECETOR_H_

enum FrameState {
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    BCC_OK,
    DATA_RCV,
    END
};

typedef struct {
    unsigned char * frame;
    int sizeFrame;
} Frame;

/**
 * @brief iniciates the communication of the serial port and sets its
configurations
 *
 * @param port - name of the serial port
 * @return int - 0 upon sucess
 */
int openSerialPort(char * port);

/**
 * @brief sends frame through the serial port
 *
 * @param fd - serial port file descriptor
 * @param message - frame to be sent
 * @param size - message size in bytes
 * @return int - number of bytes written in fd
 */
int sendMessage(int fd,unsigned char* message, int size);

/**
 * @brief reads message from serial port
 *
 * @param fd - serial port file descriptor
 * @param buf - to be completed with 1 bytes read
 * @return int - number of bytes read
 */
int receiveMessage(int fd,unsigned char * buf);

/**
 * @brief depending on the global state of the protocol where the
program is
 * it is the state machine that deals with the reception of a frame and
confirms the configuration and integrity is correct
```

```
 *
 * @param data - byte read from serial port
 * @param frameState - frame state of the recption frame state machine
(configuration steps)
 * @param globalState - fase of the data link protocol (Establish, Data
transfer and End)
 * @param frame - to be built with the specific config
 * @return int - 0 upon sucess
 */
int dataLinkState(unsigned char data, enum FrameState *frameState, int
globalState, Frame * frame);

/**
 * @brief builds the control frame acording to configuration
 *
 * @param fd - file descriptor of the serial port
 * @param control - specific control flag
 * @return int - 0 upon sucess
 */
int sendControlFrame(int fd,unsigned char control);

/**
 * @brief - iniciate the data link protocol waiting for the C_SET
supervision frame
 * and sending the C_UA supervion frame
 *
 * @param fd - file descriptor of the serial port
 * @return int - 0 upon sucess
 */
int establish(int fd);

/**
 * @brief opens serial port
 * and waits for the iniciation of data link protocol
 *
 * @param port - serial port path
 * @return int - file descriptor of serial port
 */
int llopenRecetor(char* port);

/**
 * @brief destuffs the received frame verifying it's integrity
recording it in data parameter
 *
 * @param frame - frame received from the other process
 * @param data - data content of received frame (destuffed)
 * @param dataSize - size of data in bytes
 * @return int - 0 upon sucess
 */
int destuff(Frame *frame, unsigned char *data, int *dataSize);

/**
 * @brief reads frame from the serial port analyze it and send an
according response
 * Receiver ready + next sequence number
 * Reject + reject frame sequence number
 * If frame received is OK the data is sent to the upper layer
 *
```

```
 * @param fd - file descriptor of serial port
 * @param data - data content of received frame (destuffed)
 * @param dataSize - size of data in bytes
 * @return int - 0 upon sucess
 */
int llread(int fd, unsigned char *data, int *dataSize);

/**
 * @brief - receives C_DISC supervision frame indicating the end of the
transfering process
 * sends back the C_DISC frame informing that the message was received
upon received supervision frame C_UA
 * closes the serial port at the end
 *
 * @param fd - file descriptor of serial port
 * @return int - 0 upon sucess
 */
int llclose(int fd);

#endif // DATALINKRECETOR_H_
```

dataLinkRecetor.c

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

#include "dataLinkRecetor.h"
#include "../VAR.h"

struct termios oldtio,newtio;

unsigned int frameSequenceNumber = 0; // number that must be received
switch between 0 and 1

int errorRate = 5;
long int errorCounter = 100;

int waitTime = 0;

int openSerialPort(char * port) {
    /*
```

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

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

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

    /* set input mode (non-canonical, no echo,...) */
    newtio.c_lflag = 0;

    newtio.c_cc[VTIME]    = 0;   /* inter-character timer unused */
    newtio.c_cc[VMIN]     = 1;   /* blocking read until 5 chars
received */

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

    tcflush(fd, TCIOFLUSH);

    if ( tcsetattr(fd,TCSANOW,&newtio) == -1) {
      perror("tcsetattr");
      exit(-1);
    }
    printf("New termios structure set\n");

    return fd;
}

int sendMessage(int fd,unsigned char* message, int size){
    int res;
    res = write(fd,message,size);
    printf("sent: %s, %u\n",message, res);
    return 0;
}

int receiveMessage(int fd,unsigned char * buf){
    return read(fd,buf,1);
}
```

```
int dataLinkState(unsigned char data, enum FrameState *frameState, int
globalState, Frame * frame){
    switch(*frameState){
        case START:
            if(data == FLAG){
                *frameState = FLAG_RCV;
                frame->frame[0]=data;
                frame->sizeFrame = 1;
            }
            break;
        case FLAG_RCV:
            if(data == FLAG){
                *frameState = FLAG_RCV;
                frame->frame[0]=data;
                frame->sizeFrame = 1;
            }
            else if(data == AE_SENT){
                *frameState = A_RCV;
                frame->frame[frame->sizeFrame]=data;
                frame->sizeFrame = frame->sizeFrame+1;
            }
            else{
                *frameState=START;
                frame->sizeFrame = 0;
            }
            break;
        case A_RCV:
            if(data == FLAG){
                *frameState = FLAG_RCV;
                frame->frame[0]=data;
                frame->sizeFrame = 1;
            }
            else if(data == C_SET && globalState == ESTABLISH){
                *frameState=C_RCV;
                frame->frame[frame->sizeFrame]=data;
                frame->sizeFrame = frame->sizeFrame+1;
            }
            else if((data == C_DISC || data == C_UA) &&
globalState==TERMINATE){
                *frameState=C_RCV;
                frame->frame[frame->sizeFrame]=data;
                frame->sizeFrame = frame->sizeFrame+1;
            }
            else if((data == C_S0 || data == C_S1) && globalState ==
TRANSFER){
                *frameState=C_RCV;
                frame->frame[frame->sizeFrame]=data;
                frame->sizeFrame = frame->sizeFrame+1;
            }
            else{
                *frameState=START;
                frame->sizeFrame = 0;
            }
            break;
        case C_RCV:
            if(data == FLAG){
                *frameState = FLAG_RCV;
                frame->frame[0]=data;
```

```
                frame->sizeFrame = 1;
            }
            else if(data == frame->frame[1]^frame->frame[2] /*&&
(errorCounter > errorRate)*/){
                *frameState = BCC_OK;
                frame->frame[frame->sizeFrame]=data;
                frame->sizeFrame = frame->sizeFrame+1;
                errorCounter = rand() % 1000;
            }
            else{
                errorCounter = rand() % 1000;
                *frameState=START;
                frame->sizeFrame = 0;
            }
            break;
        case BCC_OK:
            if(data==FLAG){
                *frameState=END;
                frame->frame[frame->sizeFrame]=data;
            }
            else if (globalState == TRANSFER){
                *frameState = DATA_RCV;
                frame->frame[frame->sizeFrame]=data;
                frame->sizeFrame = frame->sizeFrame+1;
            }
            else{
                *frameState=START;
                frame->sizeFrame = 0;
            }
            break;
        case DATA_RCV:
            if(data==FLAG){
                *frameState=END;
                frame->frame[frame->sizeFrame]=data;
            }
            else {
                frame->frame[frame->sizeFrame]=data;
                frame->sizeFrame = frame->sizeFrame+1;
            }
            break;
        case END:
            break;
    }
}

int sendControlFrame(int fd,unsigned char control){
    unsigned char CONTROL[5];

    CONTROL[0] = FLAG;
    CONTROL[1] = AE_SENT;
    CONTROL[2] = control;
    CONTROL[3] = CONTROL[1]^CONTROL[2]; //xor
    CONTROL[4] = FLAG;

    sendMessage(fd,CONTROL,5);

    return 0;
}
```

```
int establish(int fd){
    unsigned char buf[2047];
    enum FrameState frameState = START;

    Frame frameResponse;
    frameResponse.frame = malloc (2047 * sizeof (unsigned char));
    frameResponse.sizeFrame = 0;

    //Read control frame sent by transmitter (C_SET)
    while(frameState != END){
        receiveMessage(fd,buf);
        dataLinkState(buf[0], &frameState, ESTABLISH, &frameResponse);
        printf("%x ",buf[0]);
    }

    usleep(waitTime);

    sendControlFrame(fd, C_UA);

    free(frameResponse.frame);
    return 0;

}

int llopenRecetor(char* port){
    int fd = openSerialPort(port);
    if(fd <= 0){
        perror("openSerialPort");
        exit(-1);
    }

    if(establish(fd)!= 0){
        perror("establish");
        exit(-1);
    }

    return fd;
}

int destuff(Frame *frame, unsigned char *data, int *dataSize){
    int i = 4;
    unsigned char bcc;
    *dataSize = 0;
    while(i < frame->sizeFrame){
        if(frame->frame[i]==O_ESC){
            i++;
            if(frame->frame[i]==O_FST){
                data[*dataSize]=FLAG;
            }
            else if (frame->frame[i]==O_SND){
                data[*dataSize]=O_ESC;
            }
        }
        else {
            data[*dataSize]=frame->frame[i];
        }
        i++;
```

```
        *dataSize = *dataSize+1;
    }

    *dataSize = *dataSize - 1;
    bcc = data[0];
    for(int i=1; i<(*dataSize); i++){
        bcc = data[i]^bcc;
    }
    if(bcc != data[*dataSize]){
        return -1;
    }
    return 0;
}

int llread(int fd, unsigned char *data, int *dataSize){

    int end = FALSE;

    unsigned char buf[2047];
    enum FrameState frameResponseState = START;
    Frame frameResponse;
    frameResponse.frame = malloc (2047 * sizeof (unsigned char));
    frameResponse.sizeFrame = 0;

    while( !end ){
        frameResponseState = START;
        frameResponse.sizeFrame = 0;

        while(frameResponseState != END){
            receiveMessage(fd,buf);
            dataLinkState(buf[0], &frameResponseState, TRANSFER,
&frameResponse);
            printf("%x ",buf[0]);
        }

        usleep(waitTime);

        if(destuff(&frameResponse,data,dataSize) == -1){
            if(frameSequenceNumber == 0){
                sendControlFrame(fd, C_REJ0);
            }else if(frameSequenceNumber == 1){
                sendControlFrame(fd, C_REJ1);
            }
        }else if(frameResponse.frame[2] == C_S0){
            if(frameSequenceNumber == 0){
                frameSequenceNumber = 1;
                sendControlFrame(fd, C_RR1);
                end = TRUE;
            } else if(frameSequenceNumber == 1){
                sendControlFrame(fd, C_RR1);
            }
        }else if(frameResponse.frame[2] == C_S1){
            if(frameSequenceNumber == 1){
                frameSequenceNumber = 0;
                sendControlFrame(fd, C_RR0);
                end = TRUE;
            } else if(frameSequenceNumber == 0){
                sendControlFrame(fd, C_RR0);
```

```
            }
        }
    }

    return *dataSize;
}

int llclose(int fd){
    unsigned char buf[2047];
    enum FrameState frameState = START;

    Frame frameResponse;
    frameResponse.frame = malloc (2047 * sizeof (unsigned char));
    frameResponse.sizeFrame = 0;


    //Read control frame sent by receiver (C_DISC))
    while(frameState != END){
        receiveMessage(fd,buf);
        dataLinkState(buf[0], &frameState, TERMINATE, &frameResponse);
        printf("%x ",buf[0]);
    }

    usleep(waitTime);

    sendControlFrame(fd, C_DISC);
    free(frameResponse.frame);


    sleep(2);

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

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

Recetor.h

```
#ifndef RECETOR_H_
#define RECETOR_H_
```

```
enum PacketState {
    P_START,
    P_DATA,
    P_END
};

typedef struct {
    unsigned char *packet;
    int pSize;
    int sequenceNumber;
    enum PacketState packetState;
} AppPacket;



/**
 * @brief parse the parcket received and does the desired actions
depending on the packet state received
 * opens a new file to write and assigns the file descriptor to the
variabel fileFd (Control Packet Start received)
 * writes in file pointed by the file descritor the data that's coming
in the packet (Data Packet received)
 * closes the file pointed by the file descriptor (Control Packet End
received)
 *
 * @param fileFd - file descriptor of the file that is being writen
 * @param packet - packet that contains data send by the other process
 * @return int
 */
int parsePacket(int *fileFd, AppPacket * packet);

/**
 * @brief controls the app cycle that connects the app with the data
link protocol
 * by waiting for data from the lower layer and writting it to the file
 *
 * @param port - serial port path
 * @return int - 0 upon sucess
 */
int appFunction(char *port);

#endif // RECETOR_H_
```

Recetor.c

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

```
#include <termios.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>


#include "dataLinkRecetor.h"
#include "../VAR.h"
#include "recetor.h"


int parsePacket(int *fileFd, AppPacket * packet){
    printf("sq n : %d------------ packet 1: %d\n", packet-
>sequenceNumber, packet->packet[1]);

    //Control Packet Start
    if (packet->packet[0] == 0x02){
        if( packet->packet[1] == 0x01){
            int size = packet->packet[2];
            unsigned char fileName[255];
            for (int i = 0 ; i < size; i++){
                fileName[i] = packet->packet[i+3];
            }
            printf("filename -> %s\n",fileName);
            *fileFd = open(fileName, O_WRONLY | O_CREAT | O_TRUNC,
0666);
            if (*fileFd < 0) {printf("Erro while opening %s",fileName);
return(-1);}
            packet->sequenceNumber = 255;
            packet->packetState = P_DATA;
            return 0;
        }
    }

    //DATA Packet
    else if(packet->packet[0] == 0x01 && packet->packet[1] == (packet-
>sequenceNumber + 1) % 256){
        int size = (packet->packet[2] << 2) | packet->packet[3];
        printf("size - %d\n", size);

        if(write(*fileFd,&(packet->packet[4]),size) == -1){
            perror("Error while writing");
            return -1;
        }
        packet->sequenceNumber = (packet->sequenceNumber + 1) % 256;
        return 0;
    }

    //Control Packet End
    else if (packet->packet[0] == 0x03){
        close(*fileFd);
        packet->packetState = P_END;
        return 0;
    }
```

```
    return -1;
}

int appFunction(char *port){

    //Application
    int fd, fileFd, bufSize;
    unsigned char buf[2047];

    AppPacket appPacket;
    appPacket.pSize = 0;
    appPacket.sequenceNumber = 255;
    appPacket.packetState = P_START;
    appPacket.packet = malloc (2047 * sizeof (unsigned char));


    fd = llopenRecetor(port);

    while(appPacket.packetState != P_END){ //when we receive the end
packet
        llread(fd, appPacket.packet, &appPacket.pSize);
        parsePacket(&fileFd, &appPacket);
    }

    llclose(fd);
}

int main(int argc, char** argv){
    srand(time(0));
    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);
    }

    struct timeval start, end;
    double elapsedTime;
    clock_t startP, endP;

    startP = clock();
    gettimeofday(&start, NULL);

    //start sending data
    if (appFunction(argv[1]) != 0){
        perror("communication error");
        exit(-1);
    }

    gettimeofday(&end, NULL);

    elapsedTime = (end.tv_sec - start.tv_sec) * 1000.0;
    elapsedTime += (end.tv_usec - start.tv_usec) / 1000.0;

    endP = clock();

    printf("Receiver execution time - %f\n",elapsedTime * 1.0e-3);
```

```
    printf("Receiver process execution time - %f\n",((double) (endP -
startP)) / CLOCKS_PER_SEC);
    return 0;
}
```