Redes de Computadores

1º Trabalho Laboratorial
Protocolo de Ligação de Dados

Licenciatura em Engenharia Informática e Computação
1º Semestre - 2022/2023 | 3LEIC10

10 de novembro de 2022

Bruna Brasil Leão Marques – up202007191@fe.up.pt
Francisca Horta Guimarães – up202004229@fe.up.pt

Índice

1. Sumário

No âmbito da unidade curricular de Redes de Computadores do primeiro semestre do terceiro ano da Licenciatura em Engenharia Informática e Computação, foi-nos proposta a realização de um projeto cujo objetivo passa pela elaboração de um protocolo de ligação de dados via porta série com a finalidade de enviar ficheiros de um computador para outro.

Concluímos este trabalho com sucesso, visto que os objetivos propostos foram cumpridos.

2. Introdução

O objetivo deste projeto é implementar um protocolo de ligação de dados. Neste relatório iremos explicar o processo envolvido na sua criação.

Iremos começar por apresentar a arquitetura geral do projeto e a sua organização. De seguida, no ponto 4, iremos apresentar resumidamente a estrutura do código nas diferentes camadas e , de seguida, no ponto 5, discutiremos os casos de uso do nosso programa. Nos seguintes pontos, ponto 6 e ponto 7, os protocolos e as suas funções serão descritos com mais detalhe e, nos pontos seguintes, 8 e 9, descreveremos os testes realizados e os resultados obtidos. Por fim, concluiremos este relatório refletindo sobre os objetivos concluídos.

3. Arquitetura

O nosso projeto está dividido em dois blocos funcionais, a *Application Layer* e o *Link Layer*. A Application Layer faz uso das funções principais do Link Layer e das funções de criação dos pacotes de controlo e de informação, não tendo acesso a mais nada. Já na Link Layer são usadas as funções de configuração da porta série juntamente com as funções da Máquina de Estados e do Alarme.

4. Estrutura do Código

O nosso código está dividido em dois ficheiros de código, correspondentes às funções necessárias para execução do programa.

No ficheiro **application_layer.c** guardamos informação relativa ao ficheiro a enviar: a localização do destino do ficheiro, o tipo de aplicação, a localização da ligação ao cabo, o tempo para *time-out* e o número de retransmissões na estrutura de dados LinkLayer. Também é, neste ficheiro, que criamos as nossas tramas de controlo e de informação com ajuda das funções createControlPacket e createDataPacket, respetivamente. Este módulo é responsável pela chamada das funções contidas no Link Layer dependendo do tipo de aplicação que desejamos, receptor ou emissor.

De seguida, no ficheiro **link_layer.c** são criadas as funções responsáveis pela transmissão de dados entre o emissor e o receptor (llopen, llwrite, llread, llclose). Estas funções utilizam funções auxiliares de Máquina de Estados que nos ajudam a processar os dados recebidos (stateMachineSET, stateMachineUA, stateMachineDISC) bem como a confirmar se o tempo de receção de resposta máximo, tanto do lado do recetor como do lado do emissor, é cumprido com a ajuda das funções do Alarme (alarmHandler).

Existem header files para ambos os ficheiros e **macros.h**, que contém declarações importantes, nomeadamente para valores de flags e estados, usadas para efeitos de organização dos dados.

5. Casos de Uso Principais

Primeiramente, o utilizador deve compilar a nossa aplicação utilizando a *Makefile* disponibilizada e executando o comando "make". De seguida deverá executar o programa da seguinte forma: "./bin//main [porta] [rx/tx] [ficheiro]".

- [porta] – O número que representa a porta pela qual vamos fazer a nossa ligação. Depende do computador em questão e deverá ser substituído por "x" em "/dev/ttySx".
- [rx/tx] - Representa uma flag cujo valor rx ou tx indica se a aplicação será executada como receptor ou emissor (transmissor).
- [ficheiro] – Representa o nome do ficheiro do qual vamos ler ou escrever, dependendo do valor de [rx/tx].

Exemplo de utilização: "./bin//main /dev/ttyS0 rx penguin-received.gif" e "./bin//main /dev/ttyS0 tx penguin.gif".

O programa receptor deverá ser o primeiro a ser iniciado esperando que o emissor inicie a ligação. Caso o emissor seja iniciado primeiro, tentará ligar-se ao receptor e, se exceder o número de tentativas, terminará a aplicação. Assim que a ligação é estabelecida, o emissor envia os dados do ficheiro e o receptor recebe-os, guardando-os num ficheiro com o nome especificado pelo utilizador. Na consola, poderão aparecer mensagens de erro ou de aviso e é apresentado o progresso da transferência do ficheiro com recurso à percentagem de transferência de dados concluída. Por fim, é terminada a ligação.

6. Protocolo de Ligação Lógica

O protocolo de ligação lógica tem como principais objetivos:

- Configuração da porta série;
- Estabelecimento de ligação pela porta série;
- Transferência de dados pela porta série, fazendo stuffing e destuffing dos mesmos;
- Recuperação de erros durante a transferência de dados.

Para isso, implementamos algumas funções:

6.1. llopen e llclose

Estas funções são as necessárias para iniciar e terminar a ligação pela porta de série.

Para isso, a função llopen começa por alterar as configurações da porta de série para as pretendidas. Após isto, se a aplicação for emissor, é criada e enviada uma trama **SET** e é esperada uma trama **UA** por parte do recetor. Lê byte a byte a trama recebida e utiliza a função stateMachineUA para verificar se recebe os valores esperados. Se não receber uma resposta dentro do tempo definido na applicationLayer (*time-out*), envia a trama novamente. Caso o número de retransmissões máximo for excedido, o llopen termina com um estado de erro -1,

informando a applicationLayer que não conseguiu estabelecer a comunicação com o receptor. Se a aplicação for recetor e se receber a trama **SET** corretamente, após verificação pela função stateMachineSET, responde com uma trama **UA** (Unnumbered Acknowledgment), estabelecendo assim, a ligação com sucesso.

Já a função llclose, do lado do emissor, tenta terminar a ligação ao enviar uma trama **DISC**. Ativa o alarme, assim como no llopen, e espera pela resposta do Receptor que terá de ser, também, uma trama **DISC**. Ao recebê-la, responde com uma trama **UA**, informando o Recetor que recebeu a sua intenção de finalizar a ligação. Após isto, repõe as configurações anteriores da porta de série e termina com sucesso. Do lado do receptor, espera até receber uma trama **DISC**, respondendo com uma trama do mesmo tipo. Após a transmissão da resposta, espera pela trama **UA** do emissor. Se a receber corretamente, o programa é terminado com sucesso. Para verificar os valores que recebe, tanto o emissor quanto o receptor utilizam a função stateMachineDISC.

6.2. llwrite e llread

Estas funções são as principais responsáveis pela escrita e leitura de tramas no decorrer da aplicação.

A função llwrite, utilizada exclusivamente pelo emissor, recebe um pacote, calcula o **BCC2** e faz o byte *stuffing* necessário para o pacote e para o **BCC2**. Após isso, acrescenta-lhe o cabeçalho (flag, campo de endereço, campo de controlo e o **BCC1**). É de salientar que consideramos o pior caso para byte *stuffing* para envio da trama, isto é, o pacote ser totalmente constituído por 0x7e e 0x7d. Sendo assim, é necessário o dobro do tamanho original (ex: para um pacote de 500 bytes, a trama após o *stuffing* é de 1000 bytes). No envio da trama, tal como no llopen, ativa o temporizador e espera por uma resposta de tipo **RR** ou **REJ** do receptor. Se não receber resposta neste intervalo de tempo (*time-out*), reenvia a trama e este processo repete-se até o número máximo de retransmissões. Se receber um **REJ**, reenvia a trama.

A função llread lê byte a byte e confere a validade do cabeçalho. Envia uma resposta **REJ** se for inválido. Caso contrário, faz *destuffing*, confere o **BCC2** e verifica se recebeu uma trama duplicada através do número de sequência. Responde com **REJ** se **BCC2** for inválido ou **RR** se for válido.

7. Protocolo de Aplicação

O protocolo de ligação lógica tem como principais objetivos:

- A geração e transferência dos pacotes de controlo e de dados;
- Leitura e Escrita do ficheiro a transferir.

Para isso, implementamos algumas funções:

7.1 createControlPacket e createDataPacket

A função createControlPacket é usada na applicationLayer e cria um pacote de controlo, que contém a informação codificada em TLVs (Type, Length, Value), isto é, para cada parâmetro a passar nesse pacote, é necessário passar o tipo do parâmetro (tamanho ou nome do ficheiro), depois o seu tamanho e só depois o valor do parâmetro em si. Na aplicação desenvolvida, é

passada neste pacote a indicação de que é um pacote que sinaliza início ou fim da transmissão (campo de controlo), o nome do ficheiro e o seu tamanho.

A função createDataPacket recebe os dados lidos do ficheiro pela applicationLayer e cria o pacote com o campo de controlo (0x01 para envio de dados), número de sequência, número de octetos do campo de dados (L2 e L1) e os dados.

7.2 applicationLayer

Esta é uma função que trata todo o processo de leitura e escrita do ficheiro. Começa por chamar a função llopen da Link Layer que inicia a conexão.

Caso seja um emissor, a aplicação abre o ficheiro, chama a função createControlPacket, e envia o pacote criado através do llwrite. De seguida, lê um número determinado de bytes do ficheiro, cria o pacote de dados, com a ajuda da função createDataPacket, e envia através do llwrite. Depois de ler e enviar todo o documento, a aplicação envia um pacote de controlo que sinaliza o final da transmissão e chama, por fim, o llclose para fechar a ligação.

Caso seja um recetor, após o llopen, a aplicação lê os pacotes através do llread e interpreta os seus dados. Caso seja um pacote de controlo inicial, é criado um ficheiro com o nome presente no pacote; caso seja um pacote de dados, escreve byte a byte o conteúdo do pacote num ficheiro; caso seja um pacote de controlo final, fecha o documento e chama o llclose.

8. Validação

Efetuamos os seguintes testes para averiguar os limites da nossa aplicação:

- Envio do ficheiro proposto pelos docentes (penguin.gif);
- Interrupção da ligação antes de enviar o ficheiro (llopen);
- Interrupção da ligação enquanto enviamos o ficheiro (llwrite/llread);
- Interrupção da ligação voltando, de seguida, a estabelecê-la enquanto enviamos o ficheiro (llwrite/llread);
- Geração de ruído causado por curto-circuito enquanto enviamos o ficheiro;
- Envio de ficheiros de formato diferente;
- Envio de tramas duplicadas;
- Envio de um ficheiro com variação do tamanho de pacotes;
- Envio de ficheiros de vários tamanhos. (*)

Todos os testes foram concluídos com sucesso.

(*) - Para alguns ficheiros, nomeadamente os maiores que 2MB, a aplicação apresenta algumas limitações.

9. Eficiência do Protocolo de Ligação de Dados

Com o objetivo de avaliar a eficiência da nossa aplicação, realizamos os seguintes testes de envio do ficheiro pinguim.gif, traduzindo-os num gráfico. Foram registados os tempos de execução tanto para o lado do recetor como do emissor, tendo sido feita a média dos dois para mitigar o desvio dos dados.

Variação do Tamanho das Tramas I

Podemos confirmar, com o seguinte gráfico, que quanto maior o tamanho de cada pacote, mais eficiente é a nossa aplicação. Com este aumento, é enviada mais informação de cada vez, o que se traduz num menor número de tramas a ser enviado e numa execução mais rápida do nosso programa.

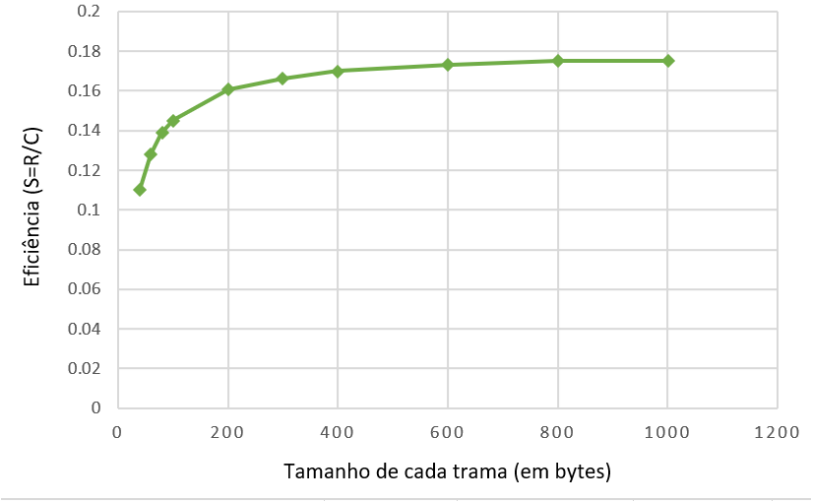

Figura 1: Variação do Tamanho das Tramas I

Variação do tempo de propagação (T_prop)

Podemos confirmar, com o seguinte gráfico, que o aumento do tempo de propagação de cada trama de informação, se traduz numa aplicação menos eficiente. Isto deve-se ao facto de a aplicação passar mais tempo sem enviar tramas uma vez que é simulado que o envio/receção de uma trama demora mais. Nestes testes, utilizou-se tramas de 1000 bytes e *baudrate* de 9600. Para este teste, utilizamos a função usleep() para simular o aumento no tempo de propagação.

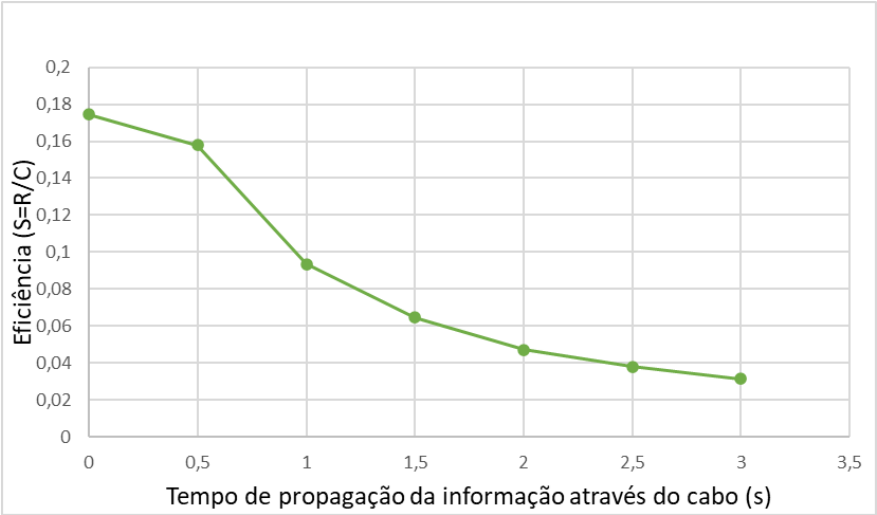

Figura 3: Variação do tempo de propagação (T_prop)

Sobre o protocolo de Stop & Wait, sabemos que é sempre necessária uma confirmação positiva por parte do receptor a cada transmissão de informação por parte do emissor para

prosseguir a transmissão. Esta confirmação é dada em formato de *acknowledgment*, **ACK** a cada trama enviada. Caso contrário, é enviado um *negative acknowledgment*, **NACK**.

Neste trabalho, utilizamos um protocolo baseado no protocolo Stop & Wait para controlo de erros. A cada envio de tramas por parte do receptor é esperado uma resposta. Se o receptor receber os dados sem erros, essa resposta é **RR** ou, se receber dados com erros, **REJ**.

Posto isto, o emissor deverá processar a resposta e enviar uma nova trama ou reenviar a mesma trama. O **Nr** destas tramas de resposta vai variar conforme o emissor tenha enviado uma trama de **Ns** 0 ou 1, para assim, quando voltar a enviar uma trama saber se pode mandar uma trama nova ou reenviar a mesma. Este mecanismo é, também, importante no tratamento de tramas duplicadas.

Exemplo:

Trama Enviada → Ns = 0
Resposta do Receptor → Sem erros: RR (Nr = 1) | Com erros: REJ (Nr=0).

Trama Enviada → Ns = 1
Resposta do Receptor → Sem erros: RR (Nr = 0) | Com erros: REJ (Nr=1).

10. Conclusões

Com este projeto, tivemos oportunidade de melhorar as nossas competências ao nível de compreensão, programação e implementação de protocolos de ligação de dados.

Consideramos que este trabalho contribuiu positivamente para nosso entendimento e aprofundamento, tanto teórico quanto prático do tema em questão, sendo considerado um sucesso pois alcançamos todos os objetivos propostos.

11. Anexo I – Código Fonte

application_layer.c

```
// Application layer protocol implementation


#include "../include/application_layer.h"


int createControlPacket(char* filename, int fileSize, int start,
unsigned char* packet){


    //L1 é so um byte ent V (o filename) não pode passar de 255
    if(strlen(filename) > 255) {
        printf("size of filename can't fit in 1 byte\n");
        return -1;
    }
    unsigned char size_string[20];
    sprintf(size_string, "%02lx", fileSize);
    int sizeBytes = strlen(size_string) / 2;


    if(start) packet[0] = 0x02; //start
    else packet[0] = 0x03; // end


    packet[1] = 0; //tamanho do ficheiro
    packet[2] = sizeBytes;
```

```
    int i = 4;

    for(int j = (sizeBytes - 1); j > -1; j--){

        packet[i] = fileSize >> (j*8);

        i++;

    }


    packet[i] = 1; //nome do ficheiro

    i++;


    int filename_size = strlen(filename);


    packet[i] = filename_size;

    i++;

    for(int j = 0; j < filename_size; j++){

        packet[i] = filename[i];

        i++;

    }


    return i;

}


int createDataPacket(unsigned char* packet, unsigned int nBytes,
int index){


    unsigned char buf[500] = {0};
```

```
    buf[0] = 0x01; //C
    buf[1] = index%255; //numero de sequencia
    buf[2] = nBytes/256;//L2
    buf[3] = nBytes%256;//L1

    for(int i = 4; i < nBytes; i++){
        buf[i] = packet[i];
    }

    for (int j = 0; j < (nBytes+4); j++) {
        packet[j] = buf[j];
    }

    return (nBytes+4); //tamanho do data packet

}

void applicationLayer(const char *serialPort, const char *role, int
baudRate,
                      int nTries, int timeout, const char
*filename) {
    printf("in the application layer\n");

    LinkLayer ll;
    strcpy(ll.serialPort, serialPort);
```

```
    ll.baudRate = baudRate;

    if(strcmp(role, "rx") == 0){

        ll.role = LlRx;

    }

    else if(strcmp(role,"tx") == 0){

        ll.role = LlTx;

    }

    ll.nRetransmissions = nTries;

    ll.timeout = timeout;



    if (llopen(ll) == -1) {

        printf("\nCouldn't estabilish the connection\n");

        return;

    } else {

        printf("\nConnection estabilished\n");

    }




    if(ll.role == LlTx){

        unsigned char packet[500];

        struct stat st;

        stat(filename, &st);

        int file = open(filename, O_RDONLY);



        if(!file){
```

```
            printf("Could not open file\n");
            return;
        }
        else {
            printf("Open file successfully\n");
        }

        unsigned int sizePac = createControlPacket(filename,
st.st_size, 1, &packet);
        if(llwrite(packet, sizePac) < 0){
            llclose(0, ll);
            return;
        }
        unsigned int bytes;
        unsigned int index = 0;
        int count = 0;
        while ((bytes = read(file, packet, 500-4)) > 0) {
            index++;
            count += bytes;
            bytes = createDataPacket(&packet, bytes, index);
            if (llwrite(packet, bytes) < 0) {
                printf("Failed to send information frame\n");
                llclose(0, ll);
                return;
            }
```

```
            printf("Sending: %d/%d (%d%%)\n", count, st.st_size,
(int) (((double)count / (double)st.st_size) *100));

        }

        bytes = createControlPacket(filename, st.st_size, 0,
&packet); //end



        if (llwrite(packet, bytes) < 0) {

            printf("Failed to send information frame\n");

        }

        close(file);



    }

    else if (ll.role == LlRx){

        int *file = -1;

        int STOP = FALSE;

        while(!STOP){

            unsigned char packet[600] = {0};

            int sizePacket = 0;

            int response = llread(&packet, &sizePacket);

            if(response < 0){

                continue;

            }

            if(packet[0] == 0x02){ //start control

                printf("\nStart control\n");

                file = fopen(filename, "wb");

            }
```

```
            else if(packet[0]==0x03){ //end control
                printf("\nEnd control\n");
                fclose(file);
                STOP = TRUE;
            }
            else{ //data
                for(int i=4; i<sizePacket; i++){
                    fputc(packet[i], file);
                }
            }
        }

    }

    if(llclose(1, ll)< 0){
        printf("Couldn't close\n");
    }

}
```

application_layer.h

```
// Application layer protocol header.
// NOTE: This file must not be changed.
```

```c
#ifndef _APPLICATION_LAYER_H_
#define _APPLICATION_LAYER_H_
#include "link_layer.h"


#include <sys/stat.h>
// Application layer main function.
// Arguments:
//   serialPort: Serial port name (e.g., /dev/ttyS0).
//   role: Application role {"tx", "rx"}.
//   baudrate: Baudrate of the serial port.
//   nTries: Maximum number of frame retries.
//   timeout: Frame timeout.
//   filename: Name of the file to send / receive.
void applicationLayer(const char *serialPort, const char *role, int
baudRate,
                      int nTries, int timeout, const char
*filename);


#endif // _APPLICATION_LAYER_H_
```

link_layer.c

```c
// Link layer protocol implementation


#include "../include/link_layer.h"


#include "../include/macros.h"


#define FALSE 0
```

```c
#define TRUE 1


volatile int STOP = FALSE;


int alarmEnabled = FALSE;

int alarmCount = 0;




#define BUF_SIZE 256


int fd;

int timeout, tries, previousNumber = 1;


struct termios oldtio;

struct termios newtio;


int infoFlag = 0;

clock_t start;


// Alarm function handler

void alarmHandler(int signal)

{

    alarmEnabled = FALSE;

    alarmCount++;
```

```
    printf("Alarm #%d\n", alarmCount);
}


enum setState stateMachineUA (unsigned char b, enum setState
state){


    switch (state)
    {
    case START_STATE:
        // se encontrar FLAG_RCV passa pra o proximo state
        if(b == FLAG) state = FLAG_RCV;
        break;


    case FLAG_RCV:
        // se encontrar A_RCV parra pro proximo state
        if(b == A) state = A_RCV;
        else if(b == A_RX) state = A_RCV;
        // se encontrar a mesma flag, FLAG_RCV, fica no mesmo
estado
        else if (b == FLAG) state = FLAG_RCV;


        // se encontrar qualquer outra flag, volta para o estado
START_STATE
        else state = START_STATE;


        break;
```

```
    case A_RCV:
        // Para o read em vez de c_ua é c_set
        if(b == C_UA) state = C_RCV;
        else if (b == FLAG) state = FLAG_RCV;
        else state = START_STATE;
        break;
    case C_RCV:
        // Para o read em vez de bcc_ua é bcc_set
        if(b == BCC_UA) state = BCC;
        else if (b == FLAG) state = FLAG_RCV;
        else state = START_STATE;
        break;


    case BCC:
        if (b == FLAG){
            state = STOP_STATE;
            STOP = TRUE;
            }
        break;


    case STOP_STATE:
        break;


    default:
        break;
```

```
    }
    return state;
}

enum setState stateMachineSET(unsigned char b, enum setState
state){
    switch (state)
    {
    case START_STATE:
        // se encontrar FLAG_RCV passa pra o proximo state
        if(b == FLAG) state = FLAG_RCV;
        break;

    case FLAG_RCV:
        // se encontrar A_RCV parra pro proximo state
        if(b == A) state = A_RCV;

        // se encontrar a mesma flag, FLAG_RCV, fica no mesmo
estado
        else if (b == FLAG) state = FLAG_RCV;

        // se encontrar qualquer outra flag, volta para o estado
START_STATE
        else state = START_STATE;
```

```
        break;


    case A_RCV:
        // Para o read em vez de c_ua é c_set
        if(b == C_SET) state = C_RCV;
        else if (b == FLAG) state = FLAG_RCV;
        else state = START_STATE;
        break;


    case C_RCV:
        // Para o read em vez de bcc_ua é bcc_set
        if(b == BCC_SET){
            state = BCC;


        }
        else if (b == FLAG) state = FLAG_RCV;
        else state = START_STATE;
        break;
    case BCC:
        if (b == FLAG){
            state = STOP_STATE;
            STOP = TRUE;
        }
        else state = START_STATE;
        break;
```

```
    case STOP_STATE:
        STOP = TRUE;
        break;

    default:
        break;
    }

    return state;
}

enum setState stateMachineDISC(unsigned char b, enum setState
state){
    switch (state)
    {
    case START_STATE:
        // se encontrar FLAG_RCV passa pra o proximo state
        if(b == FLAG) state = FLAG_RCV;
        break;

    case FLAG_RCV:
        // se encontrar A_RCV parra pro proximo state
        if(b == A) state = A_RCV;
        else if(b == A_RX) state = A_RCV;
```

```
        // se encontrar a mesma flag, FLAG_RCV, fica no mesmo
estado
        else if (b == FLAG) state = FLAG_RCV;


        // se encontrar qualquer outra flag, volta para o estado
START_STATE
        else state = START_STATE;


        break;


    case A_RCV:
        // Para o read em vez de c_ua é c_set
        if(b == C_DISC) state = C_RCV;
        else if (b == FLAG) state = FLAG_RCV;
        else state = START_STATE;
        break;
    case C_RCV:
        // Para o read em vez de bcc_ua é bcc_set
        if(b == A^C_DISC){
            state = BCC;
        }
        else if (b == FLAG) state = FLAG_RCV;
        else state = START_STATE;
        break;
    case BCC:
        if (b == FLAG){
```

```
            state = STOP_STATE;

            STOP = TRUE;

        }

        else state = START_STATE;

        break;

    default:

        break;

    }


    return state;

}


// MISC

#define _POSIX_SOURCE 1 // POSIX compliant source


////////////////////////////////////////////////

// LLOPEN

////////////////////////////////////////////////

int llopen(LinkLayer connectionParameters)


{

    start = clock();

    alarmCount = 0;

    (void)signal(SIGALRM, alarmHandler);
```

```
    fd = open(connectionParameters.serialPort, O_RDWR | O_NOCTTY |
O_NONBLOCK);

    if (fd < 0) {

        perror(connectionParameters.serialPort);

        return -1;

    }



    struct termios oldtio;

    struct termios newtio;



    // Save current port settings

    if (tcgetattr(fd, &oldtio) == -1)

    {

        perror("tcgetattr");

        return -1;

    }



    // Clear struct for new port settings

    memset(&newtio, 0, sizeof(newtio));



    newtio.c_cflag = connectionParameters.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] = 1; // Inter-character timer unused

    newtio.c_cc[VMIN] = 0;  // Blocking read until 5 chars received

    tcflush(fd, TCIOFLUSH);


    if (tcsetattr(fd, TCSANOW, &newtio) == -1)

    {

        perror("tcsetattr");

        return -1;

    }


    timeout = connectionParameters.timeout;

    tries = connectionParameters.nRetransmissions;

    printf("ROLE: %d\n", connectionParameters.role);


    if (connectionParameters.role == LlTx){

        alarmCount = 0;

        printf("I am the Emissor\n");


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

        enum setState state;

        unsigned char b;

        unsigned char SET[5];

        SET[0] = FLAG;

        SET[1] = A;
```

```
        SET[2] = C_SET;

        SET[3] = BCC_SET;

        SET[4] = FLAG;



        while(alarmCount < tries){

            state = START_STATE;

            int bytes;

            if(alarmEnabled == FALSE){

                bytes = write(fd, SET, 5);

                alarm(timeout); // 3s para escrever

                alarmEnabled = TRUE;

                if (bytes < 0){

                    printf("Failed to send SET\n");

                }

                else{

                    printf("Sending: %x,%x,%x,%x,%x\n", SET[0],
SET[1], SET[2], SET[3], SET[4]);



                    printf("Sent SET FRAME\n");



                }

            }



            while (state != STOP_STATE)

            {
```

```
                int b_rcv = read(fd, &b, 1);
                if(b_rcv <= 0){
                    break;
                }
                state = stateMachineUA(b, state);
            }

            if (state == STOP_STATE){
                break;
            }

        }

        if (alarmCount >= connectionParameters.nRetransmissions){
            printf("Error UA\n");
            return -1;
        }
        else printf("Received UA successfully\n");
    }

    else if (connectionParameters.role == LlRx)
    {
        printf("I am the Receptor\n");

        enum setState stateR = START_STATE;
```

```
        unsigned char c;


        while (stateR != STOP_STATE)
        {
            int b_rcv = read(fd, &c, 1);
            if (b_rcv > 0)
            {
                printf("Receiving: %x\n", c);


                stateR = stateMachineSET(c, stateR);
            }
        }


        printf("Received SET FRAME\n");


        unsigned char UA[5];
        UA[0] = FLAG;
        UA[1] = A_RX;
        UA[2] = C_UA;
        UA[3] = BCC_UA;
        UA[4] = FLAG;
        int bytesReceptor = write(fd, UA, 5);
        if (bytesReceptor < 0){
            printf("Failed to send UA\n");
        }
```

```
        else {
            printf("Sending: %x,%x,%x,%x,%x\n", UA[0], UA[1],
UA[2], UA[3], UA[4]);
            printf("Sent UA FRAME\n");
        }

    }
    else {
        printf("Unknown role!\n");
        exit(1);
    }


    return 1;
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(const unsigned char *buf, int bufSize)
{
    alarmCount = 0;
    (void)signal(SIGALRM, alarmHandler);

    char bcc2 = 0x00;
```

```
    for (int i = 0; i < bufSize; i++) {
        bcc2 = bcc2 ^ buf[i];
    }
    unsigned char infoFrame[1000] = {0};

    infoFrame[0] = FLAG;
    infoFrame[1] = A;
    infoFrame[2] = (infoFlag << 6); // control
    infoFrame[3] = A ^ (infoFlag << 6);

    //byte stuffing
    int index = 4;
    for(int i = 0; i < bufSize; i++) {
        if (buf[i] == 0x7E) {
            infoFrame[index] = 0x7D;
            index++;
            infoFrame[index] = 0x5E;
            index++;
        }
        else if (buf[i] == 0x7D) {
            infoFrame[index] = 0x7D;
            index++;
            infoFrame[index] = 0x5D;
            index++;
        }
```

```
        else {
            infoFrame[index] = buf[i];
            index++;
        }
    }

    // Byte stuffing of the bcc2
    if (bcc2 == 0x7E) {
        infoFrame[index] = 0x7D;
        index++;
        infoFrame[index] = 0x5E;
        index++;
    }
    else if (bcc2 == 0x7D) {
        infoFrame[index] = 0x7D;
        index++;
        infoFrame[index] = 0x5D;
        index++;
    }
    else {
        infoFrame[index] = bcc2;
        index++;
    }
    infoFrame[index] = FLAG;
    index++;
```

```
int STOP = FALSE;
unsigned char rcv[5];
alarmCount = 0;
alarmEnabled = FALSE;

while(alarmCount < tries){
    if(alarmEnabled == FALSE){
        write(fd, infoFrame, index);
        //usleep(50*1000);
        printf("\nInfo Frame sent Ns = %d\n", infoFlag);
        alarm(timeout);
        alarmEnabled = TRUE;
    }

    int response = read(fd, rcv, 5);

    if(response > 0){
        if(rcv[2] != (!infoFlag << 7 | 0x05)){
            printf("\nREJ Received\n");
            alarmEnabled = FALSE;
            continue;
        }
        else if(rcv[3] != (rcv[1]^rcv[2])){
```

```
                printf("\nRR not correct\n");
                alarmEnabled = FALSE;
                continue;
            }
            else{
                printf("\nRR correctly received\n");
                break;
            }
        }
    }
    if(alarmCount >= tries){
        printf("TIME-OUT\n");
        return -1;
    }
    previousNumber = infoFlag;
    if(infoFlag) infoFlag = 0;
    else infoFlag = 1;
    //se a resposta for RR mudo o infoflag

    return 0;

}

////////////////////////////////////////////////
// LLREAD
```

```
////////////////////////////////////////////////


int llread(unsigned char *packet, int *sizePacket)
{
    unsigned char c;
    unsigned char infoFrame[1000];
    int sizeInfoFrame = 0;
    enum statePacket statePac = packet_START;

    while (statePac != packet_STOP) {
        int bytes = read(fd, &c, 1);

        if(bytes < 0){
            continue;
        }

        //state machine pra cabeçalho
        switch(statePac){
            case packet_START:
                if (c == FLAG) {
                    statePac = packet_FLAG1;
                    infoFrame[sizeInfoFrame] = FLAG;
                    sizeInfoFrame++;
                }
```

```
            break;
        case packet_FLAG1:
            if (c == FLAG){
                statePac = packet_FLAG1;
            }
            else {
                statePac = packet_A;
                infoFrame[sizeInfoFrame] = c;
                sizeInfoFrame++;
            }
            break;
        case packet_A:
            if (c == FLAG) {
                statePac = packet_STOP;
                infoFrame[sizeInfoFrame] = c;
                sizeInfoFrame++;
            }
            else {
                infoFrame[sizeInfoFrame] = c;
                sizeInfoFrame++;
            }
            break;
        default:
            break;
    }
```

```
    }


    unsigned char rFrame[5];

    rFrame[0] = FLAG;

    rFrame[1] = A;

    rFrame[4] = FLAG;


    if(infoFrame[2] != (infoFlag << 6)){ //campo de controlo


        printf("\nInfo Frame not received correctly\nSending
REJ\n");

        rFrame[2] = (infoFlag << 7) | 0x01;

        rFrame[3] = A ^ rFrame[2];

        write(fd, rFrame, 5);


        printf("return on line 540\n");

        return -1;

    }

    else if ((infoFrame[1]^infoFrame[2]) != infoFrame[3]){ //bcc1

        printf("\nError in the protocol\nSending REJ\n");

        rFrame[2] = (infoFlag << 7) | 0x01;

        rFrame[3] = A ^ rFrame[2];

        write(fd, rFrame, 5);


        return -1;
```

```
    }


    //destuffing
    int index = 0;
    for(int i = 0; i < sizeInfoFrame; i++){
        if(infoFrame[i] == 0x7D && infoFrame[i+1]==0x5e){
            packet[index] = FLAG;
            index++;
            i++;
        }

        else if(infoFrame[i] == 0x7D && infoFrame[i+1]==0x5d){
            packet[index] = 0x7D;
            index++;
            i++;
        }

        else {
            packet[index] = infoFrame[i];
            index++;
        }
    }
    unsigned char bcc2 = 0x00;
    int size = 0;
    if(packet[4] == 0x01){ //pacote de dados
```

```
        size = 256*packet[6] + packet[7] + 4 + 5;

    } else{ //pacote de controlo

        size += packet[6] + 3 + 4; //C, T1, L1, FLAG, A, C, BCC

        size += packet[size+2] + 2 +2; //2 para contar com T2 e L2
//+2 para contar com BCC2 e FLAG

    }

    for(int i = 4; i < size-1; i++){

        bcc2 = bcc2 ^ packet[i];

    }



    //confirmar bcc2

    if(packet[size-1] == bcc2){

        if(packet[4]==0x01){ // se for dados

            if(infoFrame[5] == previousNumber){  // conferir numero
de sequencia

                printf("\nDuplicate Frame. Sending RR.\n");

                rFrame[2] = (!infoFlag << 7) | 0x05;

                rFrame[3] = rFrame[1] ^ rFrame[2];

                write(fd, rFrame, 5);

                if(infoFlag) infoFlag = 0;

                else infoFlag = 1;

                return -1;

            }

            else{

                previousNumber = infoFrame[5];

            }
```

```
        }

        printf("\nReceived InfoFrame! Sending RR\n");

        rFrame[2] = (!infoFlag << 7) | 0x05;

        rFrame[3] = rFrame[1] ^ rFrame[2];

        write(fd, rFrame, 5);

    }


    else {//erro no bcc2

        printf("\nError in the data. Sending REJ.\n");

        rFrame[2] = (infoFlag << 7) | 0x01;

        rFrame[3] = rFrame[1] ^ rFrame[2];

        write(fd, rFrame, 5);


        return -1;

    }



    index = 0;

    unsigned char packetAux[400];


    for(int i = 4; i < size-1; i++){

        packetAux[index] = packet[i];

        index++;

    }
```

```
    (*sizePacket) = size - 5;

    //esvaziar packet

    for(int i=0; i < (*sizePacket); i++){

        packet[i] = 0;

    }

    //guardar dados

    for(int i=0; i<(*sizePacket); i++){

        packet[i] = packetAux[i];

    }

    previousNumber = infoFlag;

    if(infoFlag) infoFlag = 0;

    else infoFlag = 1;

    return 0;

}



////////////////////////////////////////////////

// LLCLOSE

////////////////////////////////////////////////

int llclose(int showStatistics, LinkLayer connectionParameters)

{

    if(showStatistics){

        printf("\n---STATISTICS---\n");

        double time = ((double) (clock() - start)) / CLOCKS_PER_SEC
* 1000;

        printf("\nExecution time: %f miliseconds\n", time);
```

```
}
sleep(1);
printf("\n----LLCLOSE----\n");
alarmCount = 0;
signal(SIGALRM,alarmHandler);
alarmEnabled = FALSE;
printf("ROLE: %d\n", connectionParameters.role);


if(connectionParameters.role == LlTx){


    unsigned char array[5];
    array[0] = FLAG;
    array[1] = A;
    array[2] = C_DISC;
    array[3] = A^C_DISC;
    array[4] = FLAG;


    while(alarmCount < connectionParameters.nRetransmissions){


        enum setState state = START_STATE;
        unsigned char b;
        int bytes;
        if(alarmEnabled == FALSE){
            bytes = write(fd, array, 5);
```

```
            alarm(timeout);
            alarmEnabled = TRUE;
            if (bytes < 0){
                printf("Emissor: Failed to send DISC\n");
            }
            else{
                printf("Emissor: Sent DISC\n");
            }
        }

        //receber DISC
        while (state != STOP_STATE)
        {
            int bytesR= read(fd, &b, 1);
            if(bytesR <= 0){
                break;
            }
            printf("Reading: %x\n", b);

            state = stateMachineDISC(b, state);
        }
        if (state == STOP_STATE){
            break;
        }
    }
```

```
        if (alarmCount >= connectionParameters.nRetransmissions){
            printf("Didn't receive DISC\n");
            printf("TIME-OUT\n");

            return -1;
        }
        else printf("Emissor: Received DISC\n");

        //mandar UA
        unsigned char UA[5];
        UA[0] = FLAG;
        UA[1] = A;
        UA[2] = C_UA;
        UA[3] = BCC_UA;
        UA[4] = FLAG;
        int bytesUA = write(fd, UA, 5);
        sleep(1);
        if (bytesUA < 0){
            printf("Emissor: Failed to send UA\n");
        }
        else {
            printf("Emissor: Sending UA: %x,%x,%x,%x,%x\n", UA[0],
UA[1], UA[2], UA[3], UA[4]);

            printf("Sent UA FRAME\n");
```

```
        }


    }


    if(connectionParameters.role == LlRx){


        printf("Receiving DISC:\n");


        enum setState stateR = START_STATE;
        unsigned char a;


        while (stateR != STOP_STATE)
        {
            int b_rcv = read(fd, &a, 1);
            if (b_rcv > 0)
            {
                printf("Reading: %x\n", a);


                stateR = stateMachineDISC(a, stateR);


            }


        }
        printf("Receptor: Received DISC!\n");
```

```
unsigned char array[5];
array[0] = FLAG;
array[1] = A_RX;
array[2] = C_DISC;
array[3] = A_RX^C_DISC;
array[4] = FLAG;

alarmCount = 0;
unsigned char ua_rcv[5] = {0};
while(alarmCount < connectionParameters.nRetransmissions){

    enum setState state = START_STATE;
    unsigned char d;
    int bytes;
    if(alarmEnabled == FALSE){
        bytes = write(fd, array, 5);
        alarm(timeout); // 3s para escrever
        alarmEnabled = TRUE;
        if (bytes < 0){
            printf("Receptor: Failed to send DISC\n");
        }
        else{
            printf("Receptor: Sent DISC\n");
        }
```

```
            printf("Receptor: Receiving UA\n");
        }

        while (state != STOP_STATE)
        {
            int bytesR = read(fd, &d, 1);
            if(bytesR <= 0){
                break;
            }
            printf("Reading: %02x\n", d);

            state = stateMachineUA(d, state);

        }
        if(state == STOP_STATE){
            break;
        }
    }


}

if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
{
```

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

    close(fd);

    return 1;
}
```

link_layer.h

```
// Link layer header.
// NOTE: This file must not be changed.

#ifndef _LINK_LAYER_H_
#define _LINK_LAYER_H_

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

```c
#include <time.h>


typedef enum
{
    LlTx,
    LlRx,
} LinkLayerRole;


typedef struct
{
    char serialPort[50];
    LinkLayerRole role;
    int baudRate;
    int nRetransmissions;
    int timeout;
} LinkLayer;


// SIZE of maximum acceptable payload.
// Maximum number of bytes that application layer should send to link layer
#define MAX_PAYLOAD_SIZE 1000


// MISC
#define FALSE 0
#define TRUE 1


// Open a connection using the "port" parameters defined in struct linkLayer.
// Return "1" on success or "-1" on error.
int llopen(LinkLayer connectionParameters);
```

```
// Send data in buf with size bufSize.
// Return number of chars written, or "-1" on error.
int llwrite(const unsigned char *buf, int bufSize);

// Receive data in packet.
// Return number of chars read, or "-1" on error.
int llread(unsigned char *packet, int *sizePacket);

// Close previously opened connection.
// if showStatistics == TRUE, link layer should print statistics in
the console on close.
// Return "1" on success or "-1" on error.
int llclose(int showStatistics, LinkLayer connectionParameters);

#endif // _LINK_LAYER_H_
```

macros.h

```
enum statePacket {
    packet_START,
    packet_FLAG1,
    packet_A,
    packet_STOP
};

enum setState{
    START_STATE,
    FLAG_RCV,
```

```
    A_RCV,
    C_RCV,
    BCC,
    STOP_STATE
};

#define FLAG 0x7E
#define A 0x03
#define A_RX 0x01
#define C_SET 0x03
#define C_UA 0x07
#define BCC_SET (A^C_SET)
#define BCC_UA (A^C_UA)
#define C_DISC 0x0B
```