U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Informática e Computação

Rede de Computadores

Ano Letivo 2014/2015 – 1º Semestre

Protocolo de Ligação de Dados

(1º Trabalho Laboratorial)

SUMÁRIO

O presente relatório serve de apoio ao projeto "Protocolo de Ligação de dados" que consiste em transferir informação entre dois computadores diferentes, através do uso da porta série. Para tal, foi necessária a implementação de programas para ler e escrever a informação transferida entre computadores.

O projeto foi concluído totalmente, sendo que os processos respeitam a sequência de dados enviados e recebidos, previne e corrige erros ao longo da transmissão e restabelece a transmissão aquando esses erros.

Grupo:
Leonardo Pascoal Faria – ei12072@fe.up.pt
Maria João Marques – ei12104@fe.up.pt
Paulo Jorge Costa – ei12099@fe.up.pt
Sofia Oliveira Reis – ei12041@fe.up.pt

Docente:
Manuel Pereira Ricardo – mricardo@fe.up.pt

12 de novembro de 2014

Índice

1. Introdução

O projeto foi desenvolvido para a unidade curricular Redes de Computadores (RCOM) do 3º ano do Mestrado Integrado em Engenharia Informática e Computação (MIEIC) da Faculdade de Engenharia da Universidade do Porto (FEUP).

Os objetivos principais são a implementação de um protocolo de ligação de dados e o desenvolvimento e teste de uma aplicação responsável pela transferência de dados entre computadores. Esta aplicação é composta por dois programas, usados para enviar um ficheiro de um computador para o outro, através da porta série. Para a transferência ser segura, isto é, não ocorrerem erros que prejudiquem a transferência, como a falha na ligação, envio de dados duplicado, entre outros, foi necessário utilizar mecanismos para detetar e corrigir esses erros.

O relatório estás dividido em dez partes. Uma primeira denominada sumário, onde fazemos uma breve introdução e conclusão do relatório. Depois uma Introdução mais prolongada, seguida da descrição da arquitetura da aplicação, a estruturação do código implementado, a apresentação dos casos de utilização, o protocolo de ligação lógica, o protocolo de aplicação, a descrição dos testes efetuados os elementos de valorização feitos, as conclusões e ainda um anexo com o código fonte realizado.

2. Arquitetura

A aplicação desenvolvida está dividida em duas camadas. Uma primeira onde foi implementado o protocolo de ligação de dados (abordado no ponto 5) que usa o mecanismo Stop&Wait para prevenir erros e retransmitir tramas, e outra, a camada de aplicação, que a nível do emissor, é responsável pela separação do ficheiro em tramas e envio dessas tramas através da ligação lógica, e a nível do recetor é responsável por criar um novo ficheiro exatamente igual ao enviado e anexar, a cada iteração, a informação correspondente ao ficheiro, contida na trama recebida.

Quanto às interfaces de interação com o utilizador, é de destacar que no recetor são lançadas várias mensagens de sucesso/erro no receção da trama, o restabelecimento da transmissão caso ocorra algo que interrompa a ligação. É de salientar, também, as mensagens referentes ao tamanho das tramas recebidas e o seu número. O emissor tem uma arquitetura semelhante no que respeita a mensagens para o utilizador.

O utilizador também tem a possibilidade de escolher alguns elementos passados como argumentos na linha de comandos que lhe permite, por exemplo, enviar um ficheiro diferente do padrão.

3. Estrutura do código

No projeto foi utilizada a driver da porta de série (ttyS0 ou ttyS4), que permitia o acesso para leitura e escrita através das funções read() e write() da API do Linux.

As funções mais importantes no protocolo de ligação lógica seguem uma estrutura lógica, isto é, primeiro é efetuado o llopen(AppLayer apl) e só após este ser verificado, é que se introduz um ciclo de llwrite(int fd, char * buffer, int length) por parte do emissor e outro ciclo de llread(int fd, unsigned char** buffer) por parte do recetor, até ser chamado o llclose(AppLayer apl) que assinala o fim da transmissão e que é responsável pelo fim dos processos e pelo fecho da porta série.

Em relação ao protocolo de aplicação, o recetor caracteriza-se por um ciclo que recebe a cada iteração uma trama que de acordo com o seu estado, define o início da transmissão, o envio da informação e o fim da transferência de dados para o ficheiro. No emissor, o processo é iniciado com a abertura do ficheiro e a recolha de informação dividindo-o em vários pacotes. Em primeiro, é criado um pacote de controlo que indica o início da ligação e é enviado após a introdução dos dados necessários para a inicialização da transmissão, como nome do ficheiro e o seu tamanho. De seguida, os pacotes com a informação são passados à camada de ligação que procede ao tratamento da trama e posterior envio. A transmissão termina quando é enviado um pacote com informações de término (para confirmar a integridade do tamanho do ficheiro e nome do mesmo). As tramas de informação estão intercaladas aleatoriamente com tramas com erros, que necessitam de ser assinaladas e tratadas.

4. Casos de uso principais

Os casos de usos principais usados em ambos os processos (Emissor e Recetor) encontram-se definidos em baixo, por ordem temporalmente ordenada.

- Estabelece ligação e inicia o processo – **Recetor e Emissor**
- Fornece dados de transferência – **Emissor**
- Cria pacotes de dados **- Emissor**
- Envia tramas – **Emissor**
- Espera pela confirmação de receção **- Emissor**
- Recebe tramas – **Recetor**

- Envia confirmação de receção - **Recetor**
- Volta a enviar (**Emissor**) e volta a receber (**Recetor**) até acabarem as tramas
- Finaliza ligação – **Emissor e Recetor**

Em caso de interrupção do envio/receção de tramas, o processo reinicia, por defeito, um máximo de 3 vezes (valor esse que pode ser alterado pelo utilizador se este o entender). Caso este valor se esgote, o programa termina indicando erro.

5. Protocolo de ligação lógica

No protocolo de ligação lógica são usadas 4 funções chaves que recebem o descritor de ficheiro sendo que uns diretamente, outros indiretamente a partir da *struct AppLayer*. Este descritor é responsável por identificar a porta de série.

No emissor, o llopen() faz a inicialização do alarme e trata do envio da trama SET instanciada. Após isto, o programa espera pela receção da trama UA enviada pelo recetor, em resposta à trama SET recebida pelo recetor. Caso não aconteça, o alarme é ativado e o processo reinicia tantas vezes quantas as definidas pelo utilizador ou 3, por defeito. Caso esse número de vezes chegue ao fim, o programa termina indicando erro. Se a trama recebida pelo recetor corresponder a uma trama UA o processo continua.

No llwrite() começa-se por definir os campos da trama a enviar, nomeadamente, a *flag* 0x7E, o campo de endereço, o campo de controlo e os campos responsáveis pela proteção BCC1 e BBC2. O último campo é o ou-exclusivo do BBC1 e BCC2, e apesar de ser inserido depois do campo de dados, tem de ser calculado antes da aplicação da transparência nos mesmos, o que é feito logo de seguida a todos os valores de *flag* ou *esc* encontrados, sendo finalizado pela introdução da *flag* no final da trama.

Em seguida, é enviado o pacote ao recetor, o alarme é ativado e o programa fica à espera de uma resposta. Se essa reposta não acontecer, o alarme é novamente ativado e reinicia até atingir o número máximo de falhas, terminando no fim em caso de erro. Quando a resposta é positiva, o pacote é analisado, verificando a *flag* e o BCC1. Após a sua validação verifica-se se a trama contém RR ou REJ.

Caso o valor REJ seja recebido, o programa deve reenviar a mesma trama tantas vezes quantas definidas pelo utilizador, ao fim das quais deve terminar com indicação de erro. Caso seja RR, os valores de NS e NR devem ser verificados. Se estes forem iguais, a trama é reenviada, tantas vezes quantas as definidas pelo utilizador. Se os valores forem diferentes, o valor de NS é atualizado e o llwrite() retorna sucesso.

No llclose(), após a definição da trama DISC e do seu envio, indicando assim o fim da ligação, o programa espera pela receção de uma trama igual. Caso esta seja

recebida, é enviada uma trama UA que sinaliza o término do programa. Caso a trama recebida não seja DISC, o programa termina com erro.

Passando ao recetor, o llopen() deste estabelece a ligação com a porta série da mesma forma que o emissor. A função inicialmente verifica se a trama enviada pelo emissor é uma trama SET. A deteção de uma *flag* 0x7E marca o início da trama. Recebendo essa *flag*, o programa continua a ler da porta série até encontrar novamente esta flag. Após confirmada a trama recebida (SET), o recetor prepara uma trama de controlo designda UA. Esta é então enviada para ser processada pelo emissor. Se este envio for concluído com sucesso então a função retorna com sucesso.

O llread() funciona com base num ciclo infinito. O teu término ocorrerá aquando da receção da *flag*. O ciclo de leitura da porta série tem portanto a seguinte ideologia: enquanto não ocorrer um alarme (previamente instalado) o ciclo continua a ler. Como inicialmente recebe uma *flag*, fica a espera de ler uma nova para terminar não importando se a trama ou a *flag* está correta.

Terminado o ciclo passamos a parte de verificação da trama. É removido o processo de transparência que foi colocado na trama por parte do emissor e em caso de erro volta ao início do ciclo tentando ler novamente a mesma trama. Se esta remoção for efetuada com sucesso, passamos a parte de verificação dos campos BCC e pelo *parsing* do pacote de dados contido na trama. É verificado o primeiro BCC, se correto é iniciado o processo de leitura do pacote de dados. Ao mesmo tempo é calculado o BBC2. Se correto a função responsável pelo processo retorna o tamanho do pacotes de dados. Em caso de erro, voltamos ao início do ciclo para voltar a ler a mesma trama.

Em caso de sucesso da leitura é necessário enviar uma confirmação ao emissor para este saber que pode passar à próxima trama. Para tal é criada a trama de controlo RR tendo em conta o valor de NS e NR. Se a escrita desta for efetuada com sucesso, o llwrite() retorna o tamanho do pacote de dados lidos. Em caso de erro retorna um valor negativo.

Por fim o llclose() praticamente da mesma forma que o emissor. Este fica a espera de receber DISC, em caso de sucesso na leitura envia a confirmação ao emissor para se desligar. De seguida espera pela confirmação para se desligar. Esta confirmação corresponde a espera da receção da trama de controlo UA. Em caso de sucesso o programa retorna e o recetor desliga-se. É importante referir que é instalado um alarme para, da mesma forma já referida noutras funções, terminar se esgotar as falhas possíveis.

6. Protocolo de aplicação

A aplicação é iniciada através da função main(), que é responsável por chamar as funções relativas ao nível da aplicação tendo em conta qual o modo do programa (emissor ou recetor). Inicialmente, é chamada a função llopen() responsável por abrir a porta série para a transferência de ficheiros.

Estando no modo de recetor, após a inicialização da porta de série é chamada a função alread(). Dentro desta é feita então a leitura (através da função especifica definida na camada de ligação de dados) das tramas enviadas pelo emissor. Inicialmente, o programa espera ler o pacote inicial que contém e define o tamanho do ficheiro e o seu nome. Após a leitura são verificados os dados (através dos BCC). Se os dados estiverem corretos são guardados o nome e tamanho do ficheiro em variáveis definidas para o efeito, parando o ciclo. Em caso de erro nalgum destes passos o programa interrompe a sua execução.

Se o programa continuar, é criado um novo ficheiro com o nome que recebeu e começa-se a ler as tramas que contém realmente os dados do ficheiro. Tendo em conta o tamanho do ficheiro e o tamanho dos pacotes de dados é definido o número de pacotes que vai ser necessário ler.

Desta forma, num ciclo, é lido todos os pacotes, sendo que em cada iteração é lido um. Após a receção, verifica-se se o pacote manteve a sua integridade através do BBC1 e BBC2. Se esta última verificação estiver correta, os dados são lidos para um *buffer* e posteriormente escritos no ficheiro.

Por fim é repetido o mesmo processo de ler o pacote de início, só que neste caso é feita a leitura do pacote de fim (igual ao de inicio). O objetivo desta trama é confirmar se o pacote de início está correto. Para tal basta comparar o nome e tamanho de ficheiro. Se tivermos uma correspondência o ciclo termina e a função retorna com sucesso.

No modo emissor é invocada a função alwrite(). Esta está responsável pela escrita na porta série de todos os pacotes de dados necessários para transmitir o ficheiro.

Inicialmente, a função abre o ficheiro que pretendemos enviar. Utilizando funções específicas do Linux conseguimos obter o tamanho do ficheiro, necessário para podermos definir a quantidade de pacotes de dados.

Com este valor pode-se passar a leitura gradual do ficheiro e escrita do mesmo na porta série. Em cada iteração é criado um pacote de dados que é passado como parâmetro à função de llwrite(). Na primeira e última iteração do ciclo é feita a escrita do pacote de início e fim, respetivamente. Entre estes dois é lido do ficheiro, em cada iteração, a quantidade de dados definida pelo utilizador, criado um pacote com esses dados e posteriormente escrito na porta série. Em caso de sucesso o ciclo continua, senão o programa aborta a sua execução.

Todos estes pacotes referidos no alwrite() são criados por funções definidas para o efeito. Estas adicionam todos os campos de controlo necessários para a posterior verificação por parte do recetor.

Após a execução do alread() e alwrite(), em caso de ambas terminarem com sucesso, é invocado a função responsável pelo fecho da porta série, o llclose(). Se tivermos um retorno positivo, tudo correu como esperado e é certo que o ficheiro foi transmitido sem qualquer problema.

7. Validação

Os testes efetuados ao projeto foram feitos através de ficheiros de imagem (.gif) e texto (.txt) para ver se o ficheiro era enviado corretamente. E também foram realizados printfs das informações dos pacotes para verificar se estavam a ser mandados corretamente.

8. Elementos de valorização

Dos elementos de valorização mencionados no guião de trabalho, foram realizados os seguintes:

- Seleção de parâmetros pelo utilizador: é possível selecionar o tamanho máximo do campo de informação das tramas I (sem *stuffing*), o número máximo de retransmissões, o intervalo de time-out e ainda alterar o nome do ficheiro;
- O REJ foi implementado do lado recetor e não do emissor. Está minimamente feito, mas não concluído.

9. Conclusão

O objetivo principal do projeto foi perceber como acontece a transmissão de tramas de informação de um computador que funciona como emissor, para outro, que funciona como recetor, permitindo, assim que o ficheiro enviado e chegue sem erros ao destino. Esse objetivo foi alcançado com a implementação das 4 funções chaves já mencionadas no corpo do relatório e de funções auxiliares.

Tivemos alguns problemas em arrancar e começar o projeto. A nossa maior dificuldade ao longo do desenvolvimento do projeto foi lidar com alguns erros provocados por apontadores e alocação de memória.

Concluindo, o objetivo do projeto foi cumprido, sendo que o projeto passou em todos os testes feitos pelo professor e possui alguns elementos de valorização feitos.

10. Anexo

DATALINK.H

```
#include <linux/types.h>
#include <linux/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <time.h>
#include <signal.h>
#include <stdint.h>
#include <stdlib.h>
#include <strings.h>
#include <errno.h>
#include <string.h>

#define SET 0x03
#define UA  0x07
#define DISC    0x0b

#define RR1 0x85
#define REJ1    0x81
#define RR0 0x05
#define REJ0    0x01

#define A   0x03
#define F   0x7e

#define ESCAPE 0x7D
#define XOR_ESCAPE 0x5D
#define XOR_FLAG 0x5E
#define STF_XOR_VALUE 0x20

#define BAUDRATE B38400
#define _POSIX_SOURCE 1

#define TRANSMITER 0
#define RECEIVER 1

struct termios oldtio, newtio;

int PACK_SIZE;
int DATA_SIZE;
int ns;

int bytesWritedReaded;

int timeout;
int falhas;
int textMode;
int numRetransmissoes;
int timeoutTime;

int timeout;
int falhas;

struct applicationLayer {
    int fileDescriptor; /*Descritor correspondente à porta série*/
    int mode; /*TRANSMITTER | RECEIVER*/
```

```
}typedef AppLayer;

int llwrite(int fd, unsigned char * buffer, int length);
int llread(int fd, unsigned char** buffer);
int llopen(AppLayer apl);
int llclose(AppLayer apl);

int stuffing(unsigned char* buf, int length, unsigned char**
stufBuf);
int destuffing(unsigned char* buf, int length, unsigned char**
unstBuf);

int create_control_frame(unsigned char control, unsigned char**
frame);
int create_info_frame(int ns, unsigned char *packages, int
packages_size,
        unsigned char** frame);

void alarmhandler(int signo);

int controlStateMachine(int fd, unsigned char trama[5]);
int infoStateMachine(unsigned char *frame, int length, unsigned char
**package);
```

DATALINK.C

```
#include "dataLink.h"

int llwrite(int fd, unsigned char * buffer, int length) {
    unsigned char* stuffed;
    unsigned char* frame;

    int frame_size = create_info_frame(ns, buffer, length, &frame);

    stuffed = malloc(length * 2);
    int stuff_size = stuffing(frame, frame_size, &stuffed);
    falhas = 0;
    while (falhas < numRetransmissoes) {
        timeout = 0;
        int missing = stuff_size;
        int num = 0;
        while (missing > 0) {
            num = write(fd, stuffed, missing);
            if (textMode)
                printf("Wrote %d bytes\n", num);
            stuffed += num;
            missing -= num;
        }
        stuffed -= stuff_size;
        unsigned char* rr;
        if (ns == 0)
            create_control_frame(RR1, &rr);
        else
            create_control_frame(RR0, &rr);
        alarm(timeoutTime);
        if (controlStateMachine(fd, rr) == 0)
            break;
    }
    if (falhas >= numRetransmissoes)
        return -1;
    if (ns == 0)
```

```
        ns = 1;
    else
        ns = 0;
    return frame_size;
}

void send_reject(int fd) {
    if (textMode)
        printf("Writing REJ\n");
    unsigned char* rej;
    int missing;
    if (ns == 0)
        missing = create_control_frame(REJ1, &rej);
    else
        missing = create_control_frame(REJ0, &rej);

    int num = 0;

    while (missing > 0) {
        num = write(fd, rej, missing);
        rej += num;
        missing -= num;
    }
}

int llread(int fd, unsigned char** buffer) {
    int stuff_size;
    int pack_size;
    falhas = 0;
    while (1) {
        unsigned char *stuffed = malloc((PACK_SIZE + 4) * 2 + 2);
        alarm(timeoutTime);
        timeout = 0;
        stuff_size = 0;
        int start = 0;
        int stop = 0;

        while (timeout != 1 && !stop) {
            int r = read(fd, &stuffed[stuff_size], 1);
            if (r > 0) {
                if (stuff_size != 0 && stuffed[stuff_size] == F)
                    stop = 1;
                else if (stuff_size == 0 && stuffed[0] == F)
                    start = 1;
                if (start)
                    stuff_size += r;
            }
        }
        if (timeout == 1) {
            timeout = 0;
            continue;
        }
        falhas = 0;
        alarm(0);
        unsigned char* frame;

        int frame_size = destuffing(stuffed, stuff_size, &frame);

        if (frame_size < 0) {
            //send_reject(fd);
            continue;
        }
```

```
        pack_size = infoStateMachine(frame, frame_size, buffer);

        if (pack_size < 0) {
            //send_reject(fd);
            continue;
        }

        if (pack_size >= 0) {
            unsigned char* rr;
            int missing;

            if (ns == 0)
                missing = create_control_frame(RR1, &rr);
            else
                missing = create_control_frame(RR0, &rr);

            int num = 0;

            while (missing > 0) {
                num = write(fd, rr, missing);
                rr += num;
                missing -= num;
            }
            break;
        }
    }
    if (ns == 0)
        ns = 1;
    else
        ns = 0;
    if (falhas < numRetransmissoes)
        return pack_size;
    return -1;
}

int stuffing(unsigned char* buf, int length, unsigned char**
stufBuf) {
    unsigned i;
    unsigned j = 0;
    (*stufBuf) = malloc(length * 2 + 2);
    (*stufBuf)[j++] = F;

    for (i = 0; i < length; i++) {
        if (buf[i] == F || buf[i] == ESCAPE) {
            (*stufBuf)[j++] = ESCAPE;
            if (buf[i] == F)
                (*stufBuf)[j++] = XOR_FLAG;
            else
                (*stufBuf)[j++] = XOR_ESCAPE;
        } else
            (*stufBuf)[j++] = buf[i];
    }

    (*stufBuf)[j++] = F;
    return j;
}

int destuffing(unsigned char* buf, int length, unsigned char**
unstBuf) {
    unsigned i = 0;
    unsigned j = 0;
```

```
    i = 1;
    (*unstBuf) = malloc(length - 2);
    while (i < length - 1) {
        if (buf[i] == ESCAPE) {
            if (buf[i + 1] == XOR_FLAG) {
                (*unstBuf)[j++] = F;
                i += 2;
            } else if (buf[i + 1] == XOR_ESCAPE) {
                (*unstBuf)[j++] = ESCAPE;
                i += 2;
            } else
                return -1;

        } else
            (*unstBuf)[j++] = buf[i++];
    }
    return j;
}

int create_control_frame(unsigned char control, unsigned char**
frame) {
    (*frame) = malloc(5);
    (*frame)[0] = F;
    (*frame)[1] = A;
    (*frame)[2] = control;
    (*frame)[3] = (*frame)[1] ^ (*frame)[2];
    (*frame)[4] = F;
    return 5;
}

int create_info_frame(int ns, unsigned char *packages, int
packages_size,
        unsigned char** frame) {
    (*frame) = malloc(packages_size + 4);
    (*frame)[0] = A;
    if (ns == 0)
        (*frame)[1] = 0;
    else
        (*frame)[1] = 0x40;
    (*frame)[2] = (*frame)[1] ^ (*frame)[0];
    int i;
    unsigned char bcc = 0;
    for (i = 0; i < packages_size; ++i) {
        (*frame)[i + 3] = packages[i];
        bcc ^= packages[i];
    }
    (*frame)[i + 3] = bcc;

    return packages_size + 4;
}

int llopen(AppLayer apl) {

    if (tcgetattr(apl.fileDescriptor, &oldtio) == -1) {
        perror("tcgetattr");
        exit(-1);
    }

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

```
    newtio.c_oflag = 0;
    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(apl.fileDescriptor, TCIOFLUSH);

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

    falhas = 0;
    timeout = 0;

    if (apl.mode == 0) { //EMISSOR
        while (falhas < numRetransmissoes) {
            timeout = 0;
            unsigned char *set;
            int size = create_control_frame(SET, &set);

            int num = 0;
            int missing = size;
            if (textMode)
                printf("Sender is writing SET.\n");
            while (missing > 0) {
                num = write(apl.fileDescriptor, set, missing);
                set += num;
                missing -= num;
            }
            unsigned char *ua;
            create_control_frame(UA, &ua);
            alarm(timeoutTime);
            if (controlStateMachine(apl.fileDescriptor, ua) == 0) {
                if (textMode)
                    printf("Read  UA.\n");
                break;
            }
        }
    } else { //RECETOR
        while (falhas < numRetransmissoes) {
            unsigned char *set;
            create_control_frame(SET, &set);

            timeout = 0;
            alarm(timeoutTime);

            if (textMode)
                printf("Reading SET\n");
            if (controlStateMachine(apl.fileDescriptor, set) == 0) {
                if (textMode)
                    printf("SET received correctly. Receiver is
writing UA.\n");

                unsigned char *ua;
                int size = create_control_frame(UA, &ua);
                int num = 0;
                int missing = size;

                while (missing > 0) {
```

```
                    num = write(apl.fileDescriptor, ua, missing);
                    ua += num;
                    missing -= num;
                }
                break;
            }
        }
    }
    if (falhas >= numRetransmissoes)
        return -1; //erro
    else
        return 0; //sucesso
}

int llclose(AppLayer apl) {
    int num;
    falhas = 0;
    timeout = 0;
    if (apl.mode == 0) {
        while (falhas < numRetransmissoes) {
            timeout = 0;
            if (textMode)
                printf("Writing DISC\n");
            unsigned char *disc;
            int size = create_control_frame(DISC, &disc);
            num = 0;
            int missing = size;

            while (missing > 0) {
                num = write(apl.fileDescriptor, disc, missing);
                disc += num;
                missing -= num;
            }
            disc -= num;
            alarm(timeoutTime);
            if (controlStateMachine(apl.fileDescriptor, disc) == 0)
{
                if (textMode)
                    printf("DISC received. Sending UA!\n");
                unsigned char *ua;
                int size = create_control_frame(UA, &ua);
                num = 0;
                int missing = size;
                while (missing > 0) {
                    num = write(apl.fileDescriptor, ua, missing);
                    ua += num;
                    missing -= num;
                }
                break;
            }
        }
    } else {
        unsigned char *disc;
        int size = create_control_frame(DISC, &disc);
        while (falhas < numRetransmissoes) {
            timeout = 0;
            alarm(timeoutTime);
            if (controlStateMachine(apl.fileDescriptor, disc) == 0)
{
                if (textMode)
                    printf("DISC received. Sending it again!\n");
```

```
                num = 0;
                int missing = size;

                while (missing > 0) {
                    num = write(apl.fileDescriptor, disc, missing);
                    disc += num;
                    missing -= num;
                }

                unsigned char *ua;
                create_control_frame(UA, &ua);

                if (controlStateMachine(apl.fileDescriptor, ua) ==
0) {
                    if (textMode)
                        printf("UA received!\n");
                    break;
                }
            }
        }
    }

    if (num > 0)
        return 0; //sucesso
    else
        return -1; //erro
}
void alarmhandler(int signo) {
    if (textMode)
        printf("Failed to finish read\n");
    falhas++;
    timeout = 1;
}

int controlStateMachine(int fd, unsigned char trama[5]) {
    unsigned char temp[1];
    int state = 0;
    while (state != 5 && !timeout) {
        int r = read(fd, temp, 1);

        if (r > 0) {
            alarm(0);
            switch (state) {
            case 0:
                if (*temp == trama[0])
                    state = 1;
                break;
            case 1:
                if (*temp == trama[1])
                    state = 2;
                else
                    return -1;
                break;
            case 2:
                if (*temp == trama[0])
                    state = 1;
                else if (*temp == trama[2])
                    state = 3;
                else
                    return -1;
                break;
            case 3:
```

```
                if (*temp == trama[0])
                    state = 1;
                else if (*temp == trama[3])
                    state = 4;
                else
                    return -1;
                break;
            case 4:
                if (*temp == trama[4])
                    state = 5;
                else
                    return -1;
                break;
            }
            alarm(timeoutTime);
        }
    }

    if (timeout)
        return -1;
    else
        return 0;
}

int infoStateMachine(unsigned char *frame, int length, unsigned char
**package) {
    int state = 0;
    int i;
    while (state != 4) {
        switch (state) {
        case 0:
            if (frame[0] == A)
                state++;
            else
                return -1;
            break;
        case 1:
            if ((ns == 0 && frame[1] == 0x00) || (ns == 1 &&
frame[1] == 0x40))
                state++;
            else
                return -2;
            break;
        case 2:
            if (frame[0] ^ frame[1] == frame[2])
                state++;
            else
                return -3;
            break;
        case 3: {
            unsigned char bcc = frame[3];

            for (i = 4; i < length - 1; i++) {
                bcc ^= frame[i];
            }
            if (bcc == frame[length - 1])
                state++;
            else
                return -4;
        }
            break;
```

```
        }
    }
    (*package) = malloc(length - 4);
    for (i = 3; i < length; i++) {
        (*package)[i - 3] = frame[i];
    }
    return length - 4;
}
```

APLICATION.H

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>
#include <time.h>
#include <stdint.h>
#include <stdlib.h>
#include <strings.h>
#include <string.h>
#include <math.h>
#include "dataLink.h"

int create_control_package(unsigned char type, unsigned char* name,
        unsigned int name_size, unsigned int file_size, unsigned
char** package);

int create_data_package(int n, unsigned char* data, int data_size,
        unsigned char** package);

int alread(int fd);

void alwrite(int fd, char* file_name, int name_size);

int controlPackageStateMachine(unsigned char *frame, int control,
        char** fileName, int* fileSize);

int dataPackageStateMachine(unsigned char *frame, int control,
        int sequenceNumber, unsigned char **data);
```

APLICATION.C

```
#include "aplication.h"
int create_data_package(int n, unsigned char* data, int data_size,
        unsigned char** package) {
    (*package) = malloc(data_size + 4);
    (*package)[0] = 0x01;
    (*package)[1] = n;
    (*package)[2] = data_size / DATA_SIZE;
    (*package)[3] = data_size % DATA_SIZE;
    int i;
    for (i = 0; i < data_size; ++i)
        (*package)[i + 4] = data[i];

    return data_size + 4;
```

```
}

int create_control_package(unsigned char type, unsigned char* name,
        unsigned int name_size, unsigned int file_size, unsigned
char** package) {
    unsigned char size_str[100];
    int size_size = 0;

    while (file_size > 0) {
        size_str[size_size++] = file_size % DATA_SIZE;
        file_size = file_size / DATA_SIZE;
    }
    (*package) = malloc(4 + size_size + name_size + 1);
    (*package)[0] = type;
    (*package)[1] = 1;
    (*package)[2] = name_size;

    int i;
    for (i = 0; i < name_size; i++)
        (*package)[3 + i] = name[i];

    (*package)[3 + i++] = 0;
    (*package)[3 + i] = size_size;
    for (i = 0; i < size_size; i++)
        (*package)[5 + name_size + i] = size_str[size_size - i -
1];
    return 5 + size_size + name_size;
}

int alread(int fd) {

    bytesWritedReaded = 0;

    unsigned char* package;
    package = malloc(DATA_SIZE);
    int pack_size;
    char *fileName1, *fileName2;
    int fileSize1, fileSize2;
    while (1) {
        pack_size = llread(fd, &package);

        if (pack_size > 0
                && controlPackageStateMachine(package, 2,
&fileName1,
                        &fileSize1) == 0) {
            if (textMode)
                printf("Start package received!\n");
            break;
        } else {
            if (textMode)
                printf("Error while reading start package...
Exiting!\n");
            exit(-1);
        }
    }

    int i = 0;
    if (textMode)
        printf("Filename: %s\nFilesize: %d\n", fileName1,
fileSize1);
    int file = open(fileName1, O_WRONLY | O_CREAT, (S_IRUSR |
S_IWUSR));
```

```
    int sequenceNumber = 0;
    int numPackets = 1 + fileSize1 / DATA_SIZE;

    while (i < numPackets) {
        pack_size = llread(fd, &package);
        unsigned char *data;
        int n = dataPackageStateMachine(package, 1, sequenceNumber,
&data);
        if (textMode)
            printf("Pack with %d bytes read.\n", pack_size);
        if (n > 0) {
            write(file, data, n);
            sequenceNumber++;
            bytesWritedReaded += n;
            if (sequenceNumber == 256)
                sequenceNumber = 0;
        } else {
            if (textMode)
                printf("Error while reading data... Exiting!\n");
            exit(-1);
        }
        i++;
    }

    if (textMode)
        printf("%d bytes writed to file.\n", bytesWritedReaded);

    while (1) {
        pack_size = llread(fd, &package);

        if (controlPackageStateMachine(package, 3, &fileName2,
&fileSize2)
                == 0) {
            if (fileSize1 == fileSize2 && strcmp(fileName1,
fileName2) == 0) {
                if (textMode)
                    printf("End package received!\n");
                break;
            } else {
                if (textMode)
                    printf(
                            "The start and end package doesn't
match... Exiting\n");
                remove(fileName1);
                exit(1);
            }
        } else {
            if (textMode)
                printf("Error while reading end package...
Exiting!\n");
            exit(-1);
        }
    }
    return 0;
}

int controlPackageStateMachine(unsigned char *frame, int control,
        char** fileName, int* fileSize) {
    int state = 0;
    unsigned i = 0;
    int L1, L2, T1, T2;
    (*fileSize) = 0;
```

```
    int j;
    while (state != 7) {
        switch (state) {
        case 0:
            if (frame[i++] == control)
                state++;
            else
                return -1;
            break;
        case 1:
            T1 = frame[i++];
            state++;
            break;
        case 2:
            L1 = frame[i++];
            state++;
            break;
        case 3:
            if (T1 == 0) { //tamanho do ficheiro
                for (j = 0; j < L1; j++) {
                    (*fileSize) += pow(DATA_SIZE, L1 - j - 1) *
frame[i++];
                }
            } else if (T1 == 1) //nome do ficheiro
                    {
                (*fileName) = malloc(L1);
                for (j = 0; j < L1; j++)
                    (*fileName)[j] = frame[i++];
            } else
                return -1;
            state++;
            break;
        case 4:
            T2 = frame[i++];
            state++;
            break;
        case 5:
            L2 = frame[i++];
            state++;
            break;
        case 6:
            if (T2 == 0) { //tamanho do ficheiro
                for (j = 0; j < L2; j++) {
                    (*fileSize) += pow(DATA_SIZE, L2 - j - 1) *
frame[i++];
                }

            } else if (T2 == 1) //nome do ficheiro
                    {
                (*fileName) = malloc(L2);
                for (j = 0; j < L2; j++)
                    (*fileName)[j] = frame[i++];
            } else
                return -1;

            state++;
            break;
        }
    }
    return 0;
}
```

```
int dataPackageStateMachine(unsigned char *frame, int control,
        int sequenceNumber, unsigned char **data) {
    unsigned state = 0;
    unsigned i = 0;
    int L1, L2;
    while (state != 5)
        switch (state) {
        case 0:
            if (frame[i++] == control)
                state++;
            else
                return -1;
            break;
        case 1:
            if (frame[i++] == sequenceNumber)
                state++;
            else
                return -1;
            break;
        case 2:
            L1 = frame[i++];
            state++;
            break;
        case 3:
            L2 = frame[i++];
            state++;
            break;
        case 4:
            (*data) = malloc(DATA_SIZE * L1 + L2);
            int j;
            for (j = 0; j < (DATA_SIZE * L1 + L2); j++)
                (*data)[j] = frame[i++];

            state++;
            break;
        }
    return DATA_SIZE * L1 + L2;
}

void alwrite(int fd, char* file_name, int name_size) {

    bytesWritedReaded = 0;

    int i, pack_size;
    unsigned char* pack;
    struct stat st;

    int file = open(file_name, O_RDONLY);
    stat(file_name, &st);

    int npacks = 2 + ((int) st.st_size) / ((int) DATA_SIZE);
    if (st.st_size % DATA_SIZE != 0)
        ++npacks;

    if (textMode)
        printf("Num of packs: %d\n", npacks);

    for (i = 0; i < npacks; i++) {
        if (textMode)
            printf("PACK %d:\n", i);
        if (i == 0) {
            if (textMode)
```

```
                printf("Sending start packet.\n");
            pack_size = create_control_package(2, (unsigned char*)
file_name,
                    name_size, st.st_size, &pack);
        } else if (i == npacks - 1) {
            if (textMode)
                printf("Sending end packet\n");
            pack_size = create_control_package(3, (unsigned char*)
file_name,
                    name_size, st.st_size, &pack);
        } else {
            unsigned char* r = malloc(DATA_SIZE);
            int re = read(file, r, DATA_SIZE);
            bytesWritedReaded += re;
            if (textMode)
                printf("Sending data packet with %d bytes\n", re);
            pack_size = create_data_package((i - 1) % 256, r, re,
&pack);
        }
        if (llwrite(fd, pack, pack_size) < 0) {
            if (textMode)
                printf("Failed %d times to transmit data pack.
Exiting...\n",
                        numRetransmissoes);
            exit(-1);
        }

    }
    if (textMode)
        printf("%d bytes read from file\n", bytesWritedReaded);
}

int main(int argc, char** argv) {

    DATA_SIZE = 256;
    PACK_SIZE = DATA_SIZE + 4;

    textMode = 0;
    numRetransmissoes = 3;
    timeoutTime = 3;

    char *fileName = "pinguim.gif";

    int fd;

    if ((argc < 3)
            || ((strcmp("/dev/ttyS0", argv[1]) != 0)
                    && (strcmp("/dev/ttyS4", argv[1]) != 0))
            || ((strcmp("0", argv[2]) != 0) && (strcmp("1",
argv[2]) != 0))) {
        printf(
                "Usage:\trcom SerialPort Mode\n\tex: rcom
/dev/ttySX Y \n\tX is the port number and Y defines if transmiter
(0) or receiver(1).\n");
        printf(
                "Flags:\n\t-v Show text: 0 if false, 1 if true\n\t-
r Max atemps to resend\n\t-f File to send\n\t-t Timeout in
seconds\n\t-p Package size in bytes\n");
        exit(-1);
    }

    //process arguments
```

```
unsigned i = 3;
while (i < argc) {
    if (strcmp("-v", argv[i]) == 0) {
        textMode = 1;
        i++;
    }
    else if (strcmp("-r", argv[i]) == 0) {
        numRetransmissoes = atoi(argv[i+1]);
        i+=2;
    }
    else if (strcmp("-f", argv[i]) == 0) {
        fileName = argv[i+1];
        i+=2;
    }
    else if (strcmp("-t", argv[i]) == 0) {
        timeoutTime = atoi(argv[i+1]);
        i+=2;
    }
    else if (strcmp("-p", argv[i]) == 0) {
        DATA_SIZE = atoi(argv[i+1]);
        PACK_SIZE = DATA_SIZE + 4;
        i+=2;
    }
}

AppLayer apl;

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

timeout = 0;
apl.fileDescriptor = fd;
apl.mode = atoi(argv[2]);

struct sigaction act;
act.sa_handler = alarmhandler;
sigemptyset(&act.sa_mask);
act.sa_flags = 0;
sigaction(SIGALRM, &act, NULL);

if (llopen(apl) < 0) {
    if (textMode)
        printf("Failed to write or read SET/UA! Exiting!\n");
    exit(-1);
}
if (apl.mode == 1)
    alread(apl.fileDescriptor);
else
    alwrite(fd, fileName, strlen(fileName));
if (llclose(apl) == 0) {
    tcsetattr(apl.fileDescriptor, TCSANOW, &oldtio);
    close(apl.fileDescriptor);
    printf("Sucess!\n");
    exit(0);
} else {
    if (textMode)
        printf("Failed to write or read DISC/UA! Exiting!\n");
    exit(-1);
}
```

```
}
```