# Protocolo de Ligação de Dados

## Relatório Final


Mestrado Integrado em Engenharia Informática e Computação

Redes de Computadores

**Turma 2 / Grupo 1:**
António Pedro Araújo Fraga – 201303095
Luís Miguel da Costa Oliveira – 201304515
Miguel Guilherme Perestrelo Sampaio Pereira – 201305998

31 de outubro de 2015

# Sumário

Este relatório tem o propósito de cimentar o trabalho realizado desde o início do ano. Serve como material de apoio ao projeto "Protocolo de Ligação de Dados" que se trata de efetuar uma transferência de ficheiros entre dois computadores, usando para esse efeito uma ligação por cabo pelas portas de série de ambos os computadores.

O trabalho foi terminado com sucesso. A transferência de ficheiros entre os dois computadores é possível e é realizada sem erros ou perdas.

# Índice

# 1. Introdução

O trabalho foi elaborado no âmbito da UC de Redes de Computadores do 3º ano do MIEIC.

As metas fundamentais deste trabalho consistem na implementação do protocolo de ligação de dados e o seu teste com recurso a uma aplicação encarregada das transferências dos dados entre dois computadores. O envio de um computador para o segundo é feito recorrendo à porta de série. De maneira a que a transferência pudesse ocorrer de forma correta, foram implementados mecanismos de deteção e correção de erros por forma a evitar que pudessem, por exemplo, ocorrer falhas na conexão entre os dois computadores, transmissão de pacotes de dados em duplicado ou demais erros.

O relatório, como mencionado no sumário, serve como material de apoio ao trabalho propriamente feito e segue a estrutura recomendada sendo ela:

- **Introdução**, secção em que se refere quais os objetivos do trabalho e do respetivo relatório;
- **Arquitetura**, em que se especifica o funcionamento da aplicação e da interface do utilizador;
- **Estrutura do código**, em que se aborda as estruturas por nós definidas e as funções com maior relevância;
- **Casos de uso principais**;
- **Protocolo de ligação lógica**, em que se explica a forma como esta foi implementada;
- **Protocolo de aplicação**, especificando, tal como no ponto anterior, a linha de pensamento seguida nesta fase do trabalho;
- **Validação**, parte para mencionar os testes realizados e resultados obtidos;
- Nos **elementos de valorização** mencionam-se adições não obrigatórias mas que achamos por bem estarem presentes no trabalho;
- Na **conclusão** estarão as nossas considerações finais acerca do trabalho que realizámos.

# 2. Arquitetura

A aplicação está organizada em duas camadas. Uma onde está implementado o protocolo de ligação de dados, com as funções que estabelecem e terminam a ligação e que garantem o sincronismo e a numeração de tramas, e onde é feito o controlo de erros, recorrendo ao mecanismo *Stop-and-Wait*. Outra, a camada de aplicação, que é responsável pela leitura e escrita do ficheiro e pela emissão e receção de tramas.

No que diz respeito à interface do utilizador, durante a transmissão dos dados é apresentada uma barra de progresso com a respetiva percentagem e os *bytes* já transmitidos em relação ao total. São também apresentadas mensagens de erro, no caso do envio de uma trama falhar e de restabelecimento da ligação, na eventualidade de esta falhar por qualquer motivo. O utilizador pode ainda personalizar as definições da aplicação, tais como o *baud rate*, o tamanho máximo do campo de informação das tramas I, o número máximo de retransmissões e o intervalo de *time-out*.

# 3. Estrutura do código

A camada de ligação de dados é representada por uma estrutura onde é guardada a porta de série, o *baud rate,* o número de sequência esperado, o intervalo de *time-out*, o número máximo de retransmissões, o tamanho do campo de informação das tramas I, as estruturas do *termios* anterior e do atual e a estrutura das estatísticas da aplicação.

```
typedef struct {
        char port[20];
        int baudRate;
        unsigned int sn;
        unsigned int timeout;
        unsigned int numRetries;
        unsigned int pkgSize;
        struct termios oldtio, newtio;
        Statistics statistics;
} LinkLayer;
```

Figura 1 - Estrutura da Link Layer

O protótipo das principais funções desta camada apresenta-se na imagem seguinte.

```
int llopen();
int llwrite(unsigned char * buf, int bufSize);
int llread(unsigned char ** message);
int llclose();
```

Figura 2 - Principais funções da Link Layer

A função *llopen()* estabelece a ligação entre o emissor e o recetor, garantindo que está tudo pronto para se iniciar a transferência dos dados. O emissor chama *llwrite()* que envia os dados que lhe são passados como argumento até confirmação de receção bem sucedida. No recetor, *llread()* fica à espera de receber uma trama enviando, depois, a resposta adequada, caso a trama tenha sido recebida com sucesso ou não. Por fim, *llclose()* termina a ligação, fechando a porta de série.

A camada de aplicação é representada por uma estrutura onde é guardado o descritor correspondente à porta de série, o modo de ligação (transmissor ou recetor) e o descritor do ficheiro.

```
typedef struct {
        int fd;
        int status;
        FILE * file;
} ApplicationLayer;
```

Figura 3 - Estrutura da Application Layer

O protótipo das principais funções é apresentado na imagem seguinte.

```
int sendData(char * filePath, int fileSize);
int receiveData(char * filePath);
```

Figura 4 - Principais da Application Layer

A função *sendData()* é responsável por dividir o ficheiro em pacotes de dados que são, depois, passados à camada de ligação de dados para serem enviados. O início da transferência é indicado com o envio de um pacote de controlo que contém o nome e o tamanho do ficheiro a ser enviado. A função *receiveData()* recebe, interpreta e junta os pacotes de dados num único ficheiro. O fim da transferência é indicado pela receção de um pacote de controlo, de maneira a garantir que o ficheiro foi bem recebido.

# 4. Casos de uso principais

Os principais casos de uso incluem não só aqueles que são responsáveis pela transferência de ficheiros mas também aqueles que são responsáveis pela inserção de dados por parte do utilizador:

- Configuração da ligação e escolha de ficheiro a enviar (ou nome do ficheiro recebido) por parte do utilizador;
- Estabelece uma ligação;
- O emissor envia os dados do ficheiro a enviar;
- O recetor recebe os dados e escreve-os no ficheiro de output;
- Termina a ligação;
- Fecha os ficheiros abertos anteriormente.

Se por alguma razão o emissor não conseguir enviar os dados, então este volta a tentar N vezes com um determinado intervalo de tempo (variáveis que são definidas inicialmente pelo utilizador).

# 5. Protocolo de ligação lógica

O protocolo de ligação lógica implementa as quatro principais funções utilizadas pela camada de aplicação que tratam da transferência/receção propriamente dita dos dados.

Ao ser invocada pelo emissor, a função *llopen()* trata de enviar o comando SET, ficando à espera da resposta do recetor, a trama UA. Caso, ao fim do tempo especificado pelo utilizador, não for recebida qualquer resposta (ou se for recebida a resposta errada), o comando SET é enviado de novo. Este processo é repetido tantas vezes quantas as especificadas pelo utilizador e é implementado recorrendo a um alarme. Se se exceder o número máximo de tentativas, o programa termina, retornando erro. A receção da trama UA indica que a ligação foi bem estabelecida e o programa continua a execução. O recetor fica à espera da trama SET e, assim que a receber de forma correta, envia a trama UA.

A função *llwrite()* recebe como argumento o *buffer* a transmitir, que é usado para a construção da trama de dados, tal como especificado no guião. De seguida, envia-se a trama ao recetor e o programa fica à espera de uma resposta. Se não for recebida uma resposta no intervalo de tempo especificado, a trama é enviada novamente,

processo que é repetido até se atingir o número máximo de tentativas, caso em que a função termina, retornando erro. Quando a resposta recebida é a trama RR, então o recetor recebeu a trama de dados enviada sem erros e a função termina com sucesso, atualizando antes o número de sequência, caso se tratar de uma nova trama e não de uma trama repetida. Se a resposta for REJ, a trama não foi transmitida corretamente e deve ser enviada de novo.

A função *llread()* fica em ciclo à espera de receber uma trama. A trama é verificada com recurso a uma máquina de estados que volta ao estado inicial sempre que recebe um *byte* não esperado. No caso da trama recebida ser uma trama de dados, é feito o *destuffing* da trama e é verificado o campo BCC2. Se todos os campos forem validados com sucesso, a função envia como resposta a trama RR com o número de sequência seguinte e guarda os dados recebidos. Caso contrário, é enviada a trama REJ com número de sequência da trama de dados recebida. Se for recebido o comando DISC, a ligação deve ser terminada.

Quando o emissor invoca a função *llclose()*, é enviado o comando DISC que indica o fim da ligação e o programa fica à espera de receber uma trama igual. Assim que DISC for recebido, a função envia a trama UA e termina com sucesso. Caso contrário, termina com erro. O recetor fica a aguardar a receção da trama DISC. Assim que a recebe, a trama é reenviada, ficando o programa à espera da trama UA. Caso esta trama seja recebida, o programa termina com sucesso. À semelhança do que já foi referido para as funções anteriores, também esta função recorre ao uso de um alarme para gerir os *time-outs*.

# 6. Protocolo de aplicação

O programa inicia e são chamadas todas as funções de recolha de dados introduzidas pelo utilizador (interface).

É inicializada a *Application Layer* (camada de aplicação), com a função *initAppLayer()*. Nesta função é inicializada a estrutura respetiva recebendo o *file descriptor* da porta de série através da função *openSerialPort()*, o estado (recetor ou emissor) e o ficheiro através da função *openFile()*.

Depois disso é inicializada a camada de ligação de dados com a função *initLinkLayer()* e tenta-se estabelecer uma ligação entre os dois computadores recorrendo à função *llopen()*.

Quando os três passos anteriores estiverem completos, é feita a distinção entre o recetor e o emissor com os dados guardados anteriormente na estrutura e é chamada a função *sendData()* no caso do emissor ou a função *receiveData()* no caso do recetor. Dentro da função *sendData()*, é executado a função *sendCtrlPkt()* que envia uma trama com o campo de controlo inicial, com o tamanho e com o nome do ficheiro. No caso do recetor é executada a função *rcvCtrlPkt()* que recebe a trama enviada pelo emissor e guarda as informações sobre o ficheiro a ser recebido.

Após este procedimento estar completo o emissor vai lendo dados com o tamanho especificado inicialmente e envia-os através do método *sendDataPkt()* criando uma trama com o campo de controlo que identifica um pacote de dados, com o tamanho e dados a serem enviados e executa a função *llwrite()* definida na camada de ligação de dados. O recetor, à medida que vai escrevendo os dados no ficheiro criado, recebe todos esses pacotes com a função *llread()* que é executada dentro da função

*rcvDataPkt()* fazendo todas as verificações de erro necessárias, incluindo a verificação do número de sequência e do campo de controlo.

Quando todos os dados forem recebidos, é executada a função *sendCtrlPkt()* novamente, mas envia uma trama com um campo de controlo que identifica o fim do envio dos dados. Essa trama é recebida pela função que recebe pacotes de controlo por parte do recetor, *rcvCtrlPkt()*, chegando ao fim da execução das funções *sendDataPkt()* e *rcvDataPkt()*.

Depois do envio/receção de dados, é executada a função *llclose()* que termina a ligação, seguida do fecho da porta de série a ser utilizada com *closeSerialPort()* e do ficheiro de escrita/leitura.

Termina-se a execução da camada de aplicação depois do programa imprimir todas as estatísticas de envio/receção de mensagens e respostas.

# 7. Validação

Para testar a aplicação, efetuou-se a transferência de vários ficheiros, entre eles a imagem *pinguim.gif* fornecida e até mesmo um ficheiro de vídeo .mp4. Todas as transferências foram realizadas com sucesso, mesmo quando se desligavam os cabos de série e se interferia com o envio dos dados, inserindo uma chave nos pinos do cabo.


Figura 5 - Transferência de uma imagem

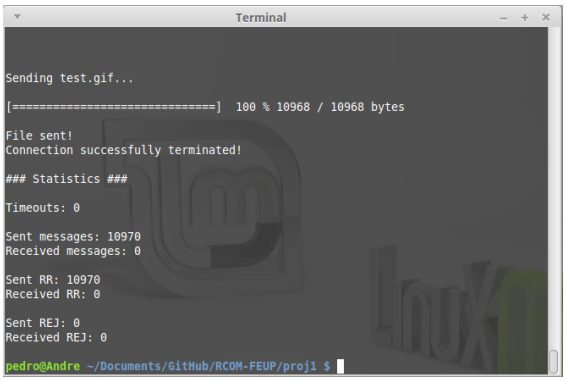


Figura 6 - Fim da transferência

# 8. Elementos de valorização

Foram implementados os seguintes elementos de valorização:

- **Seleção de parâmetros pelo utilizador** – quando o programa é iniciado, é mostrada uma interface que permite ao utilizador definir vários parâmetros de configuração tais como o *baud rate*, o tamanho máximo do campo de informação das tramas I, o número máximo de retransmissões e o intervalo de *time-out*;
- **Implementação de REJ** – caso a mensagem recebida tenha um erro no campo BCC2, o recetor envia a trama REJ ao emissor, pedindo a retransmissão da trama de dados;
- **Verificação da integridade dos dados pela aplicação** – é verificado o tamanho do ficheiro recebido (em comparação com o recebido no pacote de controlo) e se é perdido algum pacote de dados (ou se é recebido um em duplicado), utilizando o campo de numeração do pacote;

- **Registo de ocorrências** – são registadas várias informações durante a transmissão, tais como o número de tramas I transmitidas/recebidas, o número de ocorrências de *time-out* e o número de REJ e RR enviados/recebidos.

# 9. Conclusões

Os objetivos propostos foram cumpridos, tendo sido possível implementar um protocolo de ligação de dados, de acordo com as especificações dadas.

A aplicação está divida em duas camadas, como já foi referido, que são independentes entre si. Na camada de ligação de dados não é feito qualquer processamento que incida sobre o cabeçalho dos pacotes a transportar em tramas de informação, sendo que esta informação é inacessível ao protocolo de ligação de dados. Não existe, também, qualquer distinção entre pacotes de controlo e pacotes de dados.

A camada de aplicação não conhece os detalhes do protocolo de ligação de dados, mas apenas a forma como acede ao serviço – o protocolo de aplicação desconhece a estrutura das tramas e o respetivo mecanismo de delineação, a existência de *stuffing*, o mecanismo de proteção de tramas e eventuais retransmissões de tramas de informação.

Em conclusão, o objetivo foi cumprido, fornecendo um serviço de comunicação de dados fiável entre dois sistemas ligados por um cabo de série, uma vez que foram passados todos os testes realizados pela docente na aula.

# 10. Anexos

## Alarm.h

```
#pragma once

extern int alarmFired;

void alarmHandler(int signal);

void setAlarm();

void stopAlarm();
```

## Alarm.c

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

#include "DataLinkLayer.h"

int alarmFired = 0;

void handler(int signal) {
    if (signal != SIGALRM)
        return;

    alarmFired = 1;

    ll->statistics.timeout++;

    printf("Timeout! Retrying...\n");
}

void setAlarm() {
    struct sigaction action;
    action.sa_handler = handler;
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;

    sigaction(SIGALRM, &action, NULL);

    alarmFired = 0;

    alarm(ll->timeout);
}

void stopAlarm() {
    struct sigaction action;
    action.sa_handler = NULL;
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;

    sigaction(SIGALRM, &action, NULL);

    alarm(0);
}
```

## ApplicationLayer.h

```
#pragma once

typedef struct {
    int fd;
    int status;
    FILE * file;
} ApplicationLayer;

extern ApplicationLayer* al;

typedef enum {
    PARAM_SIZE = 0, PARAM_NAME = 1
} CtrlPckgParam;

int initAppLayer(char * port, int status, char * filePath, int
timeout, int retries, int pktSize, int baudrate);

FILE * openFile(char * filePath);

int sendData(char * filePath, int fileSize);
int receiveData(char * filePath);

int sendCtrlPkt(int ctrlField, char * filePath, int fileSize);
int rcvCtrlPkt(int controlField, int * fileSize, char ** filePath);

int sendDataPkt(char * buffer, int bytesRead, int i);
int rcvDataPkt(unsigned char ** buffer,int i);

void printStatistics();
```

## ApplicationLayer.c

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

#include "DataLinkLayer.h"
#include "Utilities.h"
#include "ApplicationLayer.h"
#include "Cli.h"

ApplicationLayer* al;

int initAppLayer(char* port, int status, char * filePath,int timeout,
int retries, int pktSize, int baudrate) {
    al = (ApplicationLayer*) malloc(sizeof(ApplicationLayer));

    al->fd = openSerialPort(port);

    if (al->fd < 0) {
        printf("ERROR in initAppLayer(): could not open serial
port\n");
        return ERROR;
```

```
    }
    al->status = status;

    al->file = openFile(filePath);

    int fileSize;

    if (al->status == TRANSMITTER) {
        struct stat st;
        if (stat(filePath, &st) == 0)
            fileSize = st.st_size;
        else {
            printf("ERROR getting file size!\n");
            return ERROR;
        }
    }

    if (al->file == NULL )
        return ERROR;

    if (initLinkLayer(port, baudrate, pktSize, timeout, retries) < 0)
{
        printf("ERROR in initAppLayer(): could not initialize link
layer\n");
        return ERROR;
    }

    printWaiting(al->status);

    if (llopen() == ERROR)
        return ERROR;

    if (al->status == TRANSMITTER)
        sendData(filePath, fileSize);
    else if (al->status == RECEIVER)
        receiveData(filePath);

    llclose();
    closeSerialPort();

    printStatistics();

    return 0;

}

FILE * openFile(char * filePath) {

    FILE * file;

    if(al->status == TRANSMITTER) file = fopen(filePath, "rb");
    else file = fopen(filePath, "wb");

    if(file == NULL) {
        printf("ERROR in openFile(): error opening file with path
<%s>\n", filePath);
        return NULL;
    }

    return file;
}
```

```
int sendData(char * filePath, int fileSize) {

    if (sendCtrlPkt(CTRL_PKT_START, filePath, fileSize) < 0)
        return ERROR;

    ll->statistics.msgSent++;

    int bytesRead = 0, i = 0, bytesAcumulator = 0;;
    char * buffer = malloc(ll->pktSize * sizeof(char));

    while((bytesRead = fread(buffer, sizeof(char), ll->pktSize, al-
>file)) > 0){
        if(sendDataPkt(buffer, bytesRead, i) < 0)
            return ERROR;

        ll->statistics.msgSent++;
        i++;
        if (i > 207)
            i = 0;
        bytesAcumulator += bytesRead;
        printProgressBar(filePath, bytesAcumulator, fileSize, 0);
    }

    if (fclose(al->file) < 0) {
        printf("ERROR in sendData(): error closing file!\n");
        return ERROR;
    }

    if (sendCtrlPkt(CTRL_PKT_END, filePath, fileSize) < 0)
        return ERROR;

    ll->statistics.msgSent++;

    printf("File sent!\n");

    return 0;
}



int receiveData(char * filePath) {
    int fileSize;

    if(rcvCtrlPkt(CTRL_PKT_START, &fileSize, &filePath) < 0)
        return ERROR;

    ll->statistics.msgRcvd++;

    int bytesRead, bytesAcumulator = 0, i = 0;
    unsigned char * buffer = malloc(ll->pktSize * sizeof(char));

    while (bytesAcumulator < fileSize){
        bytesRead = rcvDataPkt(&buffer, i);
        printf("%d\n", bytesRead);
        if(bytesRead < 0)
            return ERROR;
        ll->statistics.msgRcvd++;
        bytesAcumulator += bytesRead;
        fwrite(buffer, sizeof(char), bytesRead, al->file);
        i++;
```

```
            if (i > 207)
                i = 0;
            printProgressBar(filePath, bytesAcumulator, fileSize, 1);
        }

        if (fclose(al->file) < 0) {
            printf("ERROR in receiveData(): error closing file!\n");
            return ERROR;
        }

        if (rcvCtrlPkt(CTRL_PKT_END, &fileSize, &filePath) < 0)
            return ERROR;

        ll->statistics.msgRcvd++;

        printf("File received!\n");

        return 0;
}


int sendCtrlPkt(int ctrlField, char * filePath, int fileSize) {

        char sizeString[16];
        sprintf(sizeString, "%d", fileSize);

        int size = 5 + strlen(sizeString) + strlen(filePath);

        unsigned char ctrlPckg[size];

        ctrlPckg[0] = ctrlField + '0';
        ctrlPckg[1] = PARAM_SIZE + '0';
        ctrlPckg[2] = strlen(sizeString) + '0';


        int i, acumulator = 3;
        for(i = 0; i < strlen(sizeString); i++) {
            ctrlPckg[acumulator] = sizeString[i];
            acumulator++;;
        }

        ctrlPckg[acumulator] = PARAM_NAME + '0';
        acumulator++;
        ctrlPckg[acumulator] = strlen(filePath) + '0';
        acumulator++;

        for(i = 0; i < strlen(filePath); i++) {
            ctrlPckg[acumulator] = filePath[i];
            acumulator++;;
        }

        if (llwrite(ctrlPckg, size) < 0) {
            printf("ERROR in sendCtrlPkt(): llwrite() function error!\n");
            return ERROR;
        }

        return 0;
}


int rcvCtrlPkt(int controlField, int * fileSize, char ** filePath) {
```

```
    unsigned char * info;

    if (llread(&info) < 0) {
        printf("ERROR in rcvCtrlPkt(): \n");
        return ERROR;
    }

    if ((info[0] - '0') != controlField) {
        printf("ERROR in rcvCtrlPkt(): unexpected control field!\n");
        return ERROR;
    }

    if ((info[1] - '0') != PARAM_SIZE) {
        printf("ERROR in rcvCtrlPkt(): unexpected size param!\n");
        return ERROR;
    }

    int i, fileSizeLength = (info[2] - '0'), acumulator = 3;

    char fileSizeStr[MAX_STR_SIZE];

    for(i = 0; i < fileSizeLength; i++) {
        fileSizeStr[i] = info[acumulator];
        acumulator++;
    }

    fileSizeStr[acumulator - 3] = '\0';

    (*fileSize) = atoi(fileSizeStr);

    if((info[acumulator] - '0') != PARAM_NAME) {
        printf("ERROR in rcvCtrlPkt(): unexpected name param!\n");
        return ERROR;
    }

    acumulator++;

    int pathLength = (info[acumulator] - '0');
    acumulator++;

    char pathStr[MAX_STR_SIZE];

    for(i = 0; i < pathLength; i++) {
        pathStr[i] = info[acumulator];
        acumulator++;
    }

    pathStr[i] = '\0';
    strcpy((*filePath), pathStr);

    return 0;
}


int sendDataPkt(char * buffer, int bytesRead, int i) {

    int size = bytesRead + 4;
    unsigned char dataPckg[size];

    dataPckg[0] = CTRL_PKT_DATA + '0';
```

```
    dataPckg[1] = i + '0';

    dataPckg[2] = bytesRead / 256;
    dataPckg[3] = bytesRead % 256;
    memcpy(&dataPckg[4], buffer, bytesRead);

    if (llwrite(dataPckg, size) < 0) {
        printf("ERROR in sendDataPkt(): llwrite() function error!\n");
        return ERROR;
    }

    return 0;
}

int rcvDataPkt(unsigned char ** buffer,int i) {

    unsigned char * info = NULL;
    int bytes = 0;

    if (llread(&info) < 0) {
        printf("ERROR in rcvDataPkt(): llread() function error!\n");
        return ERROR;
    }

    if (info == NULL)
        return 0;

    int C = info[0] - '0';
    int N = info[1] - '0';

    if (C != CTRL_PKT_DATA) {
        printf("ERROR in rcvDataPkt(): control field it's different
from CTRL_PKT_DATA!\n");
        return ERROR;
    }

    if (N != i) {
        printf("ERROR in rcvDataPkt(): sequence number it's
wrong!\n");
        return ERROR;
    }

    int L2 = info[2], L1 = info[3];
    bytes = 256 * L2 + L1;

    memcpy((*buffer), &info[4], bytes);

    free(info);

    return bytes;
}

void printStatistics() {
    printf("\n");
    printf("### Statistics ###\n\n");
    printf("Timeouts: %d\n\n", ll->statistics.timeout);
    printf("Sent messages: %d\n", ll->statistics.msgSent);
    printf("Received messages: %d\n\n", ll->statistics.msgRcvd);
    printf("Sent RR: %d\n", ll->statistics.rrSent);
    printf("Received RR: %d\n\n", ll->statistics.rrRcvd);
    printf("Sent REJ: %d\n", ll->statistics.rejSent);
```

```
    printf("Received REJ: %d\n\n", ll->statistics.rejRcvd);
}
```

## Cli.h

```
#pragma once

int getMode();

void clrscr();

char * getPort();

char * getFileName(int status);

int getRetries();

int getTimeout();

int getBaudrate();

void printProgressBar(char * fileName, int bytes, int size, int
status);

void printWaiting(int status);

int getPktSize();
```

## Cli.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "Cli.h"
#include "Utilities.h"
#include "DataLinkLayer.h"


int getMode() {
    int mode = ERROR;
    while (mode != 1 && mode != 2) {
        clrscr();
        printf("# Do you want to receive or send data?\n\n");
        printf("\t1. Send\n");
        printf("\t2. Receive\n\n> ");
        mode = getchar() - '0';
    }
    return mode - 1;
}


char * getPort() {
    int port = ERROR;
    while (port != 1 && port != 2) {
        clrscr();
        printf("# What port do you want to use?\n\n");
        printf("\t1. /dev/ttyS0\n");
        printf("\t2. /dev/ttyS4\n\n> ");
        port = getchar() - '0';
    }

    if (port == 1)
```

```
        return "/dev/ttyS0";
    else
        return "/dev/ttyS4";
}


char * getFileName(int mode) {
    char * fileName = malloc(150*sizeof(char));;
    clrscr();
    if (mode == 0)
        printf("# Type the name of file to be sent: \n\n> ");
    else
        printf("# Type the name of the output file: \n\n> ");

    scanf("%s", fileName);

    return fileName;
}

int getRetries() {
    int retries = ERROR;
    while (retries <= 0) {
        clrscr();
        printf("# What is the maximum number of retries to send a
packet?\n\n> ");

        scanf("%d", &retries);
    }

    return retries;
}

int getTimeout() {
    int timeout = ERROR;
    while (timeout <= 0) {
        clrscr();
        printf("# What is the timeout waiting time in seconds?\n\n>
");

        scanf("%d", &timeout);
    }

    return timeout;
}

int getPktSize() {
    int pktSize = ERROR;
    while (pktSize <= 0 || pktSize > 512) {
        clrscr();
        printf("# What is the maximum packet size?\n\n> ");

        scanf("%d", &pktSize);
    }

    return pktSize;
}

int getBaudrate() {
  int choice = ERROR;
  while (getBaudrateChoice(choice) < 0) {
    clrscr();
```

```
    printf("# What is the baudrate value?\n\n");
    printf("[ 0, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400,
4800, 9600, 19200, 38400, 57600, 115200, 230400, 460800]\n\n> ");

    scanf("%d", &choice);
  }
  return getBaudrateChoice(choice);
}

void printProgressBar(char * fileName, int bytes, int size, int mode)
{
    clrscr();
    if (mode == 0)
        printf("Sending %s...\n\n", fileName);
    else if (mode == 1)
        printf("Receiving %s...\n\n", fileName);

    printf("[");
    int i, barSize = 30;
    for (i = 0; i < barSize; i++) {
        if(((float)bytes / (float)size ) > ( (float)i / barSize))
            printf("=");
        else
            printf(" ");
    }
    printf("]");
    printf("  %d %%\t%d / %d bytes\n\n", (int)((float)bytes /
(float)size * 100), bytes, size);
}

void printWaiting(int mode) {
    clrscr();
    if (mode == 0)
        printf("Waiting for receiver...\n\n");
    else
        printf("Waiting for transmitter...\n\n");
}

void clrscr() {
    printf("\033[2J");
}
```

DataLinkLayer.h

```
#pragma once

#include <termios.h>

#include "Utilities.h"

#define C_SET 0x03
#define C_UA 0x07
#define C_RR 0x05
#define C_REJ 0x01
#define C_DISC 0x0B

#define FLAG 0x7E
#define A03 0x03
#define A01 0x01
#define ESCAPE 0x7D

typedef enum {
```

```
    SET, UA, RR, REJ, DISC, NONE
} Command;

typedef enum {
    INVALID, DATA, COMMAND
} FrameType;

typedef struct {
    int timeout;

    int msgSent;
    int msgRcvd;

    int rrSent;
    int rrRcvd;

    int rejSent;
    int rejRcvd;

} Statistics;

typedef struct {
    char port[20];
    int baudRate;
    unsigned int sn;
    unsigned int timeout;
    unsigned int numRetries;
    unsigned int pktSize;
    struct termios oldtio, newtio;
    Statistics statistics;
} LinkLayer;

typedef struct {
    unsigned char frame[MAX_FRAME_SIZE];
    unsigned int size;
    unsigned int sn;
    FrameType type;
    Command answer;
} Frame;


extern LinkLayer* ll;

int initLinkLayer(char* port, int baudRate, int pktSize, unsigned int
timeout, unsigned int numTransmissions);

int getBaudrateChoice(int choice);

int openSerialPort(char* port);

int closeSerialPort();

int setNewTermios();

int llopen();
int llwrite(unsigned char * buf, int bufSize);
int llread(unsigned char ** message);
int llclose();

unsigned char getBCC2(unsigned char* data, unsigned int size);
```

```
int sendDataFrame(int fd, unsigned char* data, unsigned int size);

int sendCommand(int fd, Command cmd);

int isCommand(Frame frm, Command cmd);

unsigned char getAFromCmd();

unsigned char getAFromRspn();

Frame receiveFrame(int fd);

Frame stuff(Frame df);

Frame destuff(Frame df);
```

## DataLinkLayer.c

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

#include "DataLinkLayer.h"
#include "ApplicationLayer.h"
#include "Alarm.h"

LinkLayer* ll;

int initLinkLayer(char* port, int baudRate, int pktSize, unsigned int
timeout, unsigned int numRetries) {
    ll = (LinkLayer*) malloc(sizeof(LinkLayer));

    strcpy(ll->port, port);
    ll->baudRate = baudRate;
    ll->sn = 0;
    ll->timeout = timeout;
    ll->numRetries = numRetries;
    ll->pktSize = pktSize;

    ll->statistics.timeout = 0;
    ll->statistics.msgSent = 0;
    ll->statistics.msgRcvd = 0;
    ll->statistics.rrSent = 0;
    ll->statistics.rrRcvd = 0;
    ll->statistics.rejSent = 0;
    ll->statistics.rejRcvd = 0;

    if (setNewTermios() < 0)
        return ERROR;

    return 0;
}

int getBaudrateChoice(int choice) {
    switch (choice) {
```

```
    case 0:
        return B0;
    case 50:
        return B50;
    case 75:
        return B75;
    case 110:
        return B110;
    case 134:
        return B134;
    case 150:
        return B150;
    case 200:
        return B200;
    case 300:
        return B300;
    case 600:
        return B600;
    case 1200:
        return B1200;
    case 1800:
        return B1800;
    case 2400:
        return B2400;
    case 4800:
        return B4800;
    case 9600:
        return B9600;
    case 19200:
        return B19200;
    case 38400:
        return B38400;
    case 57600:
        return B57600;
    case 115200:
        return B115200;
    case 230400:
        return B230400;
    case 460800:
        return B460800;
    default:
        return -1;
    }
}

int openSerialPort(char* port) {
    return open(port, O_RDWR | O_NOCTTY);
}

int closeSerialPort() {
    // set old settings
    if (tcsetattr(al->fd, TCSANOW, &ll->oldtio) < 0) {
        printf("ERROR in closeSerialPort(): could not set old
termios\n");
        return ERROR;
    }

    close(al->fd);

    return 0;
}
```

```
int setNewTermios() {
    // save current port settings
    if (tcgetattr(al->fd, &ll->oldtio) < 0) {
        printf("ERROR in setNewTermios(): could not get old
termios\n");
        return ERROR;
    }

    // set new termios
    bzero(&ll->newtio, sizeof(ll->newtio));
    ll->newtio.c_cflag = ll->baudRate | CS8 | CLOCAL | CREAD;
    ll->newtio.c_iflag = IGNPAR;
    ll->newtio.c_oflag = 0;


    ll->newtio.c_lflag = 0;

    ll->newtio.c_cc[VTIME] = 0;
    ll->newtio.c_cc[VMIN] = 1;


    if (tcflush(al->fd, TCIOFLUSH) < 0) {
        printf("ERROR in setNewTermios(): could not flush non-
transmitted output data\n");
        return ERROR;
    }

    if (tcsetattr(al->fd, TCSANOW, &ll->newtio) < -1) {
        printf("ERROR in setNewTermios(): could not set new
termios\n");
        return ERROR;
    }
    return 0;
}

int llopen() {
    int counter = 0;

    switch(al->status){
        case TRANSMITTER:
            while(counter < ll->numRetries) {
                if (counter == 0 || alarmFired) {
                    alarmFired = 0;
                    sendCommand(al->fd, SET);
                    counter++;

                    setAlarm();
                }

                if (isCommand(receiveFrame(al->fd), UA)) {
                    counter--;
                    break;
                }
            }

            stopAlarm();
            if (counter < ll->numRetries)
                printf("Connection successfully established!\n");
            else {
```

```
                printf("Could not establish a connection: maximum
number of retries reached\n");
                return ERROR;
            }
            break;
        case RECEIVER:
            if (isCommand(receiveFrame(al->fd), SET)) {
                sendCommand(al->fd, UA);
                printf("Connection successfully established!\n");
            }
            break;
        default:
            break;
    }

    return 0;
}

int llwrite(unsigned char* buf, int bufSize) {
    int counter = 0;
    Frame receivedFrame;

    while(counter < ll->numRetries) {
        if (counter == 0 || alarmFired) {
            alarmFired = 0;
            sendDataFrame(al->fd, buf, bufSize);
            counter++;

            setAlarm();
        }

        receivedFrame = receiveFrame(al->fd);

        if (isCommand(receivedFrame, RR)) {
            ll->statistics.rrSent++;

            if(ll->sn != receivedFrame.sn)
                ll->sn = receivedFrame.sn;

            stopAlarm();
            counter--;
            break;

        } else if (isCommand(receivedFrame, REJ)) {
            ll->statistics.rejSent++;
            counter = 0;
            stopAlarm();
        }

    }

    if (counter >= ll->numRetries) {
        printf("Could not send frame: maximum number of retries
reached\n");
        stopAlarm();
        return ERROR;
    }

    return 0;
}
```

```
int llread(unsigned char ** message) {
    int disc = 0, dataSize;
    Frame frm;

    while (!disc) {
        frm = receiveFrame(al->fd);

        switch (frm.type) {
            case COMMAND:
                if (isCommand(frm, DISC))
                    disc = 1;

                break;
            case DATA:
                if (frm.answer == RR && ll->sn == frm.sn) {
                    ll->statistics.rrRcvd++;
                    ll->sn = !frm.sn;
                    dataSize = frm.size - DATA_FRAME_SIZE;
                    *message = malloc(dataSize);
                    memcpy(*message, &frm.frame[4], dataSize);
                    disc = 1;
                }
                else if (frm.answer == REJ) {
                    ll->statistics.rejRcvd++;
                    ll->sn = frm.sn;
                }

                if (frm.answer != NONE)
                    sendCommand(al->fd, frm.answer);
                break;
            case INVALID:
                break;
            default:
                return ERROR;
        }

    }

    return 0;
}

int llclose() {
    int counter = 0;

    switch(al->status) {
        case TRANSMITTER:
            while(counter < ll->numRetries) {
                if (counter == 0 || alarmFired) {
                    alarmFired = 0;
                    sendCommand(al->fd, DISC);
                    counter++;
                    setAlarm();
                    if (isCommand(receiveFrame(al->fd), DISC)) {
                        sendCommand(al->fd, UA);
                        sleep(1);
                        break;
                    }
                }
            }

            stopAlarm();
```

```
            if (counter < ll->numRetries)
                printf("Connection successfully terminated!\n");
            else {
                printf("Could not disconect: maximum number of retries
reached\n");
                return ERROR;
            }
            break;
        case RECEIVER:
            while (!isCommand(receiveFrame(al->fd), DISC))
                continue;

            while(counter < ll->numRetries) {
                if (counter == 0 || alarmFired) {
                    alarmFired = 0;
                    counter++;
                    setAlarm();
                    sendCommand(al->fd, DISC);
                    if (isCommand(receiveFrame(al->fd), UA))
                        break;
                }
            }
            stopAlarm();
            if (counter < ll->numRetries)
                printf("Connection successfully terminated!\n");
            else {
                printf("Could not disconect: maximum number of retries
reached\n");
                return ERROR;
            }
            break;
        default:
            break;
    }

    return 0;
}

int isCommand(Frame frm, Command cmd) {
    if (frm.type == INVALID)
        return 0;

    switch (frm.frame[2] & 0x0F) {
    case C_SET:
        if (cmd == SET)
            return 1;
        else
            return 0;
    case C_UA:
        if (cmd == UA)
            return 1;
        else
            return 0;
    case C_RR:
        if (cmd == RR)
            return 1;
        else
            return 0;
    case C_REJ:
        if (cmd == REJ)
            return 1;
```

```
        else
            return 0;
    case C_DISC:
        if (cmd == DISC)
            return 1;
        else
            return 0;
    default:
        return 0;
    }

    return 0;
}

unsigned char getBCC2(unsigned char* data, unsigned int size) {
    unsigned char BCC = 0;

    int i;
    for (i = 0; i < size; i++)
        BCC ^= data[i];

    return BCC;
}

int sendDataFrame(int fd, unsigned char* data, unsigned int size) {
    Frame df;
    df.size =  size + DATA_FRAME_SIZE;

    df.frame[0] = FLAG;
    df.frame[1] = A03;
    df.frame[2] = ll->sn << 5;
    df.frame[3] = df.frame[1] ^ df.frame[2];
    memcpy(&df.frame[4], data, size);
    df.frame[4 + size] = getBCC2(data, size);
    df.frame[5 + size] = FLAG;

    df = stuff(df);

    if (write(fd, df.frame, df.size) != df.size) {
        printf("ERROR in sendDataFrame(): could not send frame\n");
        return ERROR;
    }

    return 0;
}

int sendCommand(int fd, Command cmd) {
    unsigned char frame[FRAME_SIZE];

    frame[0] = FLAG;
    frame[4] = FLAG;

    switch(cmd) {
        case SET:
            frame[1] = getAFromCmd();
            frame[2] = C_SET;
            frame[3] = frame[1] ^ frame[2];
            break;
        case UA:
            frame[1] = getAFromRspn();
            frame[2] = C_UA;
```

```
            frame[3] = frame[1] ^ frame[2];
            break;
        case DISC:
            frame[1] = getAFromCmd();
            frame[2] = C_DISC;
            frame[3] = frame[1] ^ frame[2];
            break;
        case RR:
            frame[1] = getAFromRspn();
            frame[2] = C_RR;
            frame[2] |= (ll->sn << 5);
            frame[3] = frame[1] ^ frame[2];
            break;
        case REJ:
            frame[1] = getAFromRspn();
            frame[2] = C_REJ;
            frame[2] |= (ll->sn << 5);
            frame[3] = frame[1] ^ frame[2];
            break;
        default:
            printf("ERROR in sendFrame(): unexpected frame\n");
            break;
    }

    if(frame[1] == 0) {
        printf("ERROR in sendFrame(): unnexpected status\n");
        return ERROR;
    }

    if (write(fd, frame, FRAME_SIZE) != FRAME_SIZE) {
        printf("ERROR in sendFrame(): could not send frame\n");
        return ERROR;
    }

    return 0;
}

unsigned char getAFromCmd() {
    switch(al->status){
        case TRANSMITTER:
            return A03;
        case RECEIVER:
            return A01;
    }
    return 0;
}

unsigned char getAFromRspn() {
    switch(al->status){
        case TRANSMITTER:
            return A01;
        case RECEIVER:
            return A03;
    }
    return 0;
}

Frame receiveFrame(int fd) {
    unsigned char c;
    int res, receiving = 1, state = 0, dataFrame = 0, i = 0;
    Frame frm;
```

```
frm.type = INVALID;

while(receiving) {

    res = read(fd, &c, 1);

    if (res < 1)
        return frm;

    switch(state) {
    case 0:
        if (c == FLAG) {
            frm.frame[i] = c;
            i++;
            state++;
        }
        break;
    case 1:
        if (c == A01 || c == A03) {
            frm.frame[i] = c;
            i++;
            state++;
        }
        else if (c != FLAG) {
            state = 0;
            i = 0;
        }
        break;
    case 2:
        if (c != FLAG) {
            frm.frame[i] = c;
            i++;
            state++;
        }
        else if (c == FLAG) {
            state = 1;
            i = 1;
        }
        else {
            state = 0;
            i = 0;
        }
        break;
    case 3:
        if (c == (frm.frame[1]^frm.frame[2])) {
            frm.frame[i] = c;
            i++;
            state++;
        }
        else if (c == FLAG) {
            state = 1;
            i = 1;
        }
        else {
            state = 0;
            i = 0;
        }
        break;
    case 4:
        if (c == FLAG) {
            frm.frame[i] = c;
            i++;
```

```
                frm.frame[i] = 0;
                receiving = 0;

                if (i > 5)
                    dataFrame = 1;
            }
            else {
                frm.frame[i] = c;
                i++;
            }
            break;
        default:
            break;
        }
    }

    if (dataFrame) {
        frm.size = i;
        frm.answer = NONE;
        frm.type = DATA;
        frm = destuff(frm);

        // check BCC1
        if (frm.frame[3] != (frm.frame[1] ^ frm.frame[2])) {
            printf("ERROR in receiveFrame(): BCC1 error\n");
            return frm;
        }

        // check BCC2
        int dataSize = frm.size - DATA_FRAME_SIZE;
        unsigned char BCC2 = getBCC2(&frm.frame[4], dataSize);
        if (frm.frame[4 + dataSize] != BCC2) {
            printf("ERROR in receiveFrame(): BCC2 error\n");
            frm.answer = REJ;
        }

        if (frm.answer == NONE)
            frm.answer = RR;

        frm.sn = (frm.frame[2] >> 5) & BIT(0);
    }
    else {
        frm.type = COMMAND;
        if (isCommand(frm, RR) || isCommand(frm, REJ))
            frm.sn = (frm.frame[2] >> 5) & BIT(0);
    }
    return frm;
}

Frame stuff(Frame df) {
    Frame stuffedFrame;
    unsigned int newSize = df.size;

    int i;
    for (i = 1; i < (df.size - 1); i++) {
        if (df.frame[i] == FLAG || df.frame[i] == ESCAPE)
            newSize++;
    }

    stuffedFrame.frame[0] = df.frame[0];
    int j = 1;
```

```
    for (i = 1; i < (df.size - 1); i++) {
        if (df.frame[i] == FLAG || df.frame[i] == ESCAPE) {
            stuffedFrame.frame[j] = ESCAPE;
            stuffedFrame.frame[++j] = df.frame[i] ^ 0x20;
        }
        else
            stuffedFrame.frame[j] = df.frame[i];
        j++;
    }

    stuffedFrame.frame[j] = df.frame[i];
    stuffedFrame.size = newSize;

    return stuffedFrame;
}

Frame destuff(Frame df) {
    Frame destuffedFrame;
    destuffedFrame.sn = df.sn;
    destuffedFrame.type = df.type;
    destuffedFrame.answer = df.answer;
    int j = 0;

    int i;
    for (i = 0; i < df.size; i++) {
        if (df.frame[i] == ESCAPE)
            destuffedFrame.frame[j] = df.frame[++i] ^ 0x20;
        else
            destuffedFrame.frame[j] = df.frame[i];
        j++;
    }

    destuffedFrame.size = j;

    return destuffedFrame;
}
```

## Utilities.h

```
#pragma once

#define TRANSMITTER 0
#define RECEIVER 1
#define BAUDRATE B38400
#define ERROR -1

#define FRAME_SIZE 5
#define DATA_FRAME_SIZE 6
#define MAX_BUF_SIZE 256
#define MAX_FRAME_SIZE 1024

#define MAX_STR_SIZE 100

#define BIT(n) (0x01 << n)

typedef enum {
    CTRL_PKT_DATA = 0, CTRL_PKT_START = 1, CTRL_PKT_END = 2
} ControlPacketType;
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>

#include "Utilities.h"
#include "ApplicationLayer.h"
#include "Cli.h"


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

    if (argc > 1) {
        clrscr();
        printf("ERROR! This programs takes no arguments\n");
        exit(ERROR);
    }

    int mode = getMode();

    if (mode == ERROR )
        return ERROR;
    else if( mode == RECEIVER || mode == TRANSMITTER)
        ;
    else
        return ERROR;

    char * port = getPort();

    int retries = getRetries();

    int timeout = getTimeout();

    int pktSize = getPktSize();

    int baudrate = getBaudrate();

        char * fileName = getFileName(mode);

        initAppLayer(port, mode, fileName,timeout, retries, pktSize,
baudrate);

    return 0;
}
```