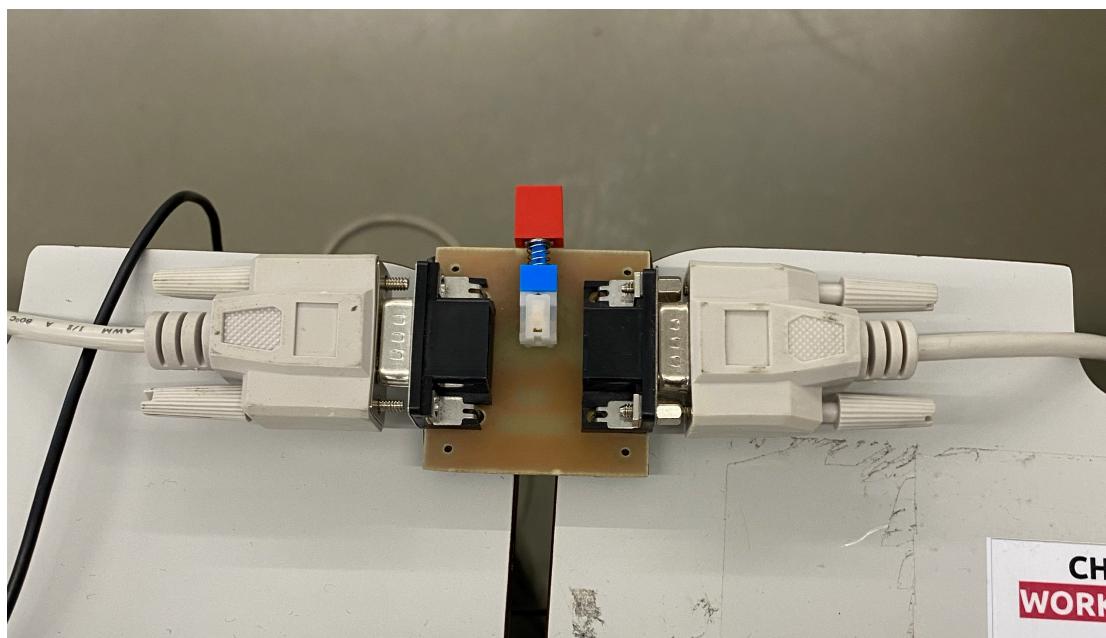




1º Trabalho Laboratorial

Protocolo de Ligação de Dados



Trabalho realizado por:
José Miguel Moreira Isidro
up202006485@fe.up.pt
Turma 12 - Grupo 8

Índice

Sumário	3
Introdução	3
Arquitetura	3
Estrutura do código	4
<i>Funções principais da camada de aplicação</i>	4
<i>Funções auxiliares da camada de aplicação</i>	4
<i>Funções principais da camada de ligação de dados</i>	4
<i>Funções auxiliares da camada de ligação de dados</i>	4
<i>Estruturas de dados</i>	5
<i>Variáveis globais</i>	5
<i>Macros pertinentes</i>	5
Casos de uso principais	6
Interface	6
Transmissão de Dados	6
Protocolos utilizados	7
Protocolo de ligação lógica	7
<i>llopen</i>	7
<i>llwrite</i>	8
<i>llread</i>	9
<i>llclose</i>	9
<i>Funções auxiliares</i>	10
Protocolo de aplicação	10
<i>applicationLayer</i>	10
<i>sendFile</i>	11
<i>receiveFile</i>	11
<i>Funções auxiliares</i>	12
Protocolo Stop & Wait	12
Validação	13
Testes efetuados	13
Resultados obtidos	13
Eficiência do protocolo de ligação de dados	13
Análise dos testes efetuados	13
Variação do FER	14
Variação do tempo de propagação (T_prop)	14
Variação do baudrate (C – capacidade de ligação)	15
Variação do tamanho das tramas de informação	15
Conclusão	16
Anexos	17
Código Fonte	17
Medições da Eficiência	52

Sumário

Este trabalho foi realizado no âmbito da unidade curricular Redes de Computadores (RCOM) do 3º ano da Licenciatura em Engenharia Informática e Computação. O relatório incide sobre o primeiro trabalho laboratorial, cujo foco é a transferência de dados através de uma aplicação. A transferência de dados é feita através da implementação de um protocolo de comunicação entre duas máquinas.

O trabalho prático foi concluído com sucesso, sendo todos os objetivos definidos inicialmente atingidos.

Introdução

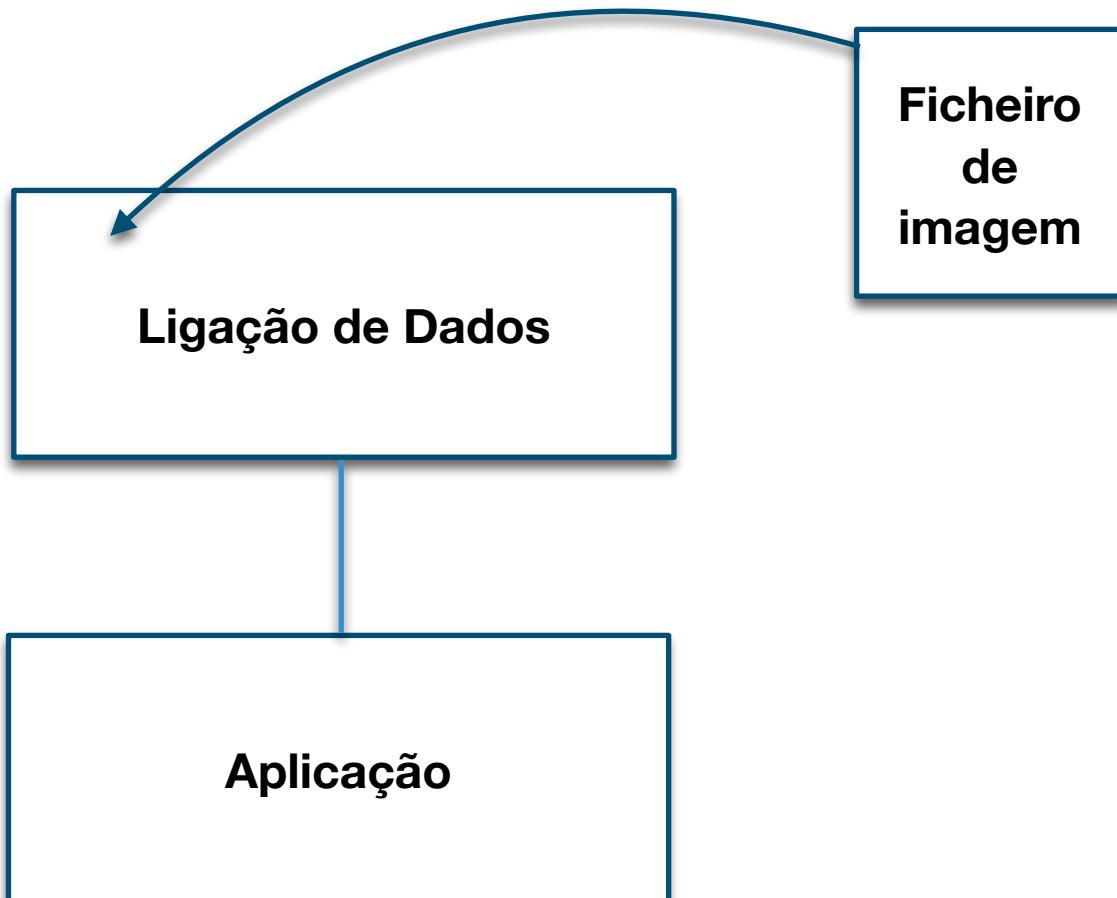
Este trabalho tem dois objetivos principais, o objetivo relativo a camada do protocolo de ligação de dados e o objetivo relativo a camada da aplicação.

O objetivo do protocolo de ligação de dados é fornecer um serviço de comunicação de dados fiável entre dois computadores ligados por um meio (canal) de transmissão – neste caso, um cabo série.

O objetivo ao nível da aplicação é implementar uma aplicação muito simples para a transferência de um ficheiro, usando o serviço oferecido pelo protocolo de ligação de dados.

Arquitetura

O projeto está assente num bloco principal ao nível da aplicação que tem dois modos de funcionamento: *transmitter* (transmissor) e *receiver* (recetor). Em ambos os modos, o bloco chama funções presentes na camada de aplicação e na camada de ligação de dados, contudo há uma clara distinção de quem está a utilizar estas camadas, e por isso, embora ambos utilizem o `llopen` e `llclose` (funções da camada de ligação de dados), a implementação destas funções garante que haja independência entre o transmissor e o recetor.



Estrutura do código

O código encontra-se distribuído por vários ficheiros de modo a ficar mais organizado e legível. No ficheiro principal, `main.c`, é chamada a função **applicationLayer** que se encontra no ficheiro `application_layer.c`. Este segundo ficheiro chama funções da camada da aplicação e estas por sua vez chamam funções da camada de ligação de dados. Indico abaixo as principais funções implementadas, pelo que estas são explicadas em mais detalhe na secção dos “Protocolos utilizados”.

Funções principais da camada de aplicação

- **applicationLayer()** - recebe a porta, o papel desempenhado, a baudrate, o número de tentativas, o timeout e o nome do ficheiro a receber; guarda todas as informações relativas à ligação de dados na variável global `ll` e por fim chama a função **sendFile** ou **receiveFile** dependendo do papel desempenhado
- **sendFile()** - abre o ficheiro, divide-o em pacotes, envia-os dentro de tramas para o recetor e fecha o ficheiro
- **receiveFile()** - recebe o ficheiro, recebe as tramas e guarda os pacotes no novo ficheiro

Funções auxiliares da camada de aplicação

- **buildDataPacket()** - constrói um pacote de dados do ficheiro, para ser enviado numa *I-frame*
- **parseDataPacket()** - extrai a informação de um pacote de dados do ficheiro, proveniente de uma *I-frame*
- **buildControlPacket()** - constrói um pacote de controlo, para ser enviado numa *I-frame*
- **parseControlPacket()** - extrai a informação de um pacote de controlo, proveniente de uma *I-frame*
- **getFileSize()** - retorna o tamanho de um ficheiro

Funções principais da camada de ligação de dados

- **llopen()** - envia uma trama de supervisão *SET* e recebe uma trama *UA*
- **llwrite()** - efetua *byte stuffing* nas *I-frames* e envia-as para o recetor
- **llread()** - recebe as *I-frames*, lê-as, e efetua *byte destuffing*
- **llclose()** - envia uma trama de supervisão *DISC*, recebe uma trama *DISC* e envia uma trama *UA*

Funções auxiliares da camada de ligação de dados

- **byteStuffing()** - efetua *byte stuffing* numa *I-frame*
- **byteDestuffing()** - reverte a operação de *byte stuffing* numa *I-frame*
- **createSupervisionFrame()** - cria uma trama de supervisão
- **createInformationFrame()** - cria uma trama de informação (*I-frame*)
- **readSupervisionFrame()** - lê uma trama de supervisão
- **readInformationFrame()** - lê uma trama de informação (*I-frame*)
- **sendFrame()** - envia uma trama

Estruturas de dados

```

struct ApplicationLayer {
    int fileDescriptor; /* Descriptor correspondente à porta série */
};

typedef enum {
    LITx, /* Transmitter */
    LIRx, /* Receiver */
}; LinkLayerRole

struct linkLayer {
    char serialPort[50]; /* Device /dev/ttySx, x = 0, 1 */
    LinkLayerRole role; /* Role played in transfer: Transmitter ou Receiver */
    int baudRate; /* Rate at which information is transferred in the channel*/
    unsigned int nRetransmissions /* Number of retries in case of error */
    unsigned int timeout; /* Value of the timer in seconds */
    unsigned char frame[MAX_SIZE_FRAME]; /* Frame */
    unsigned int frame_length; /* Current frame size */
    unsigned int sequenceNumber /* Frame sequence number: 0, 1 */
};

```

Variáveis globais

- **ApplicationLayer al;**
- **LinkLayer ll;**
- **struct termios oldtio;**

Macros pertinentes

- **MAX_SIZE_PACK**
- **MAX_SIZE_DATA**
- **MAX_SIZE_FRAME**
- **ADD_SEND**
- **ADD_REC**
- **I_0**
- **I_1**
- **DATA_START**
- **CTRL_START**
- **CTRL_DATA**
- **CTRL_END**
- **TYPE_FILESIZE**
- **TYPE_FILENAME**

Casos de uso principais

Interface

A interface permite ao utilizador escolher o seu papel desempenhado na transferência, assim como escolher o ficheiro a enviar e a porta de série a ser utilizada.

O utilizador, utilizando a consola, correrá o programa, dando um conjunto de argumentos. Do lado do transmissor, deve ser inserida a porta de série a ser utilizada (ex. `/dev/ttyS0`), o papel desempenhado ‘tx’ e o nome do ficheiro a ser enviado (ex. `pinguim.gif`). Do lado do recetor, aplica-se o mesmo que ao transmissor, deve ser inserida a porta de série a ser utilizada (ex. `/dev/ttyS1`), o papel desempenhado será ‘rx’ e o nome do ficheiro a ser recebido (ex. `pinguim-received.gif`).

Para efeitos de teste, pode ser utilizado também o makefile incluído com o código fonte, no qual é possível alterar estas variáveis facilmente.

Transmissão de Dados

A transmissão de dados ocorre via porta de série, entre dois computadores, e ocorre a seguinte sequência de eventos:

- É configurada uma ligação não canónica entre os dois computadores;
- A ligação é estabelecida;
- O transmissor envia os dados, trama a trama;
- Simultaneamente, o recetor recebe os dados, trama a trama;
- À medida que recebe os dados, o recetor guarda-os num ficheiro com nome previamente inserido pelo utilizador;
- A ligação é terminada.

Protocolos utilizados

Protocolo de ligação lógica

No protocolo de ligação, foram implementadas as funções **llopen**, **llwrite**, **llread** e **llclose**. São as principais funções deste protocolo e servem como interface para o protocolo da aplicação.

Estas funções fazem, respetivamente:

- O estabelecimento da ligação entre o transmissor e o recetor;
- O envio de uma trama de informação;
- A receção de uma trama de informação;
- O término da ligação entre o transmissor e o recetor.

As leituras das trama são feitas através de uma **máquina de estados**, que vai recebendo byte a byte a mensagem e executando mudanças de estado, de modo a que apenas se pode chegar ao estado final se a trama recebida tiver um formato válido.

llopen

```
/***
 * Open a connection using the "port" parameters defined in struct LinkLayer (global variable).
 * @return Positive value when sucess; negative value when error
 */
int llopen();
```

Figura 1 - llopen

Esta função começa por abrir a ligação à porta de série e criar o descriptor de ficheiro, e por instalar o alarm handler de modo a poder utilizar o alarme criado. De seguida, tendo em conta o valor do parâmetro **role** guardado em **LL** (variável global), chama uma das duas funções auxiliares, **llOpenTransmitter** ou **llOpenReceiver**.

```
/***
 * Opens the connection for the transmitter
 * @param fd File descriptor for the serial port
 * @return File descriptor; -1 in case of error
 */
int llOpenTransmitter(int fd);
```

Figura 2 - Função llOpenTransmitter

```
/***
 * Opens the connection for the receiver
 * @param fd File descriptor for the serial port
 * @return File descriptor; -1 in case of error
 */
int llOpenReceiver(int fd);
```

Figura 3 - Função llOpenReceiver

Estas funções desempenham o resto das funcionalidades relacionadas com o envio e receção de tramas para estabelecer a ligação. O transmissor envia um trama de supervisão **SET** e fica à espera de ler uma trama **UA**, ocorrendo o timeout e/ou a saída do programa se o número máximo de retransmissões for ultrapassado. O recetor lê uma trama **SET**, e depois de o receber envia uma trama **UA**. O envio de tramas é feito com a função **sendFrame**, e a receção com **readSupervisionFrame**. As tramas são geradas através da função **createSupervisionFrame**. A receção das tramas é feita byte a byte, e o valor de cada um é processado através de uma máquina de estados, onde a função **event_handler** é usada para verificar e atualizar os estados.

llwrite

```
/***
 * Transfer data stored in packet.
 * @param fd File descriptor for the serial port
 * @param packet Packet (data) to be written
 * @param length Size of the packet
 * @return Number of chars written; negative value when error
 */
int llwrite(int fd, unsigned char *packet, int length);
```

Figura 4 - Função llwrite

Esta função é apenas chamada pelo transmissor, de modo a enviar tramas de informação ao receptor e é composta pelas seguintes fases:

- Criação da trama de informação, utilizando os dados da aplicação passados como argumentos (**createInformationFrame**);
- Introdução de *byte stuffing* na trama gerada (**byteStuffing**);
- Envio da trama, com o número de sequência correto (*I_0* ou *I_1*) (**sendFrame**);
- Leitura da resposta (*RR* ou *REJ*), com possibilidade de timeout (**readSupervisionFrame**);
- Saída da função, se foi recebido um *RR*, ou reenvio da trama, se for recebido um *REJ*, as vezes que forem necessárias até receber uma confirmação positiva.

O timeout das tramas *I*, *SET* e *DISC* é feito da seguinte maneira. **finish** e **resendFrame** são flags modificadas pelo alarm handler (implementado em *ll_aux.c*), quando ocorre um timeout, que indicam se o programa deve acabar ou se a trama deve ser reenviada, respetivamente.

```
while (finish != TRUE) {
    // read_value contains the index the expectedByte found by the state machine if it succeeds, else -1
    read_value = readSupervisionFrame(responseBuffer, fd, expectedBytes, 2, ADD_SEND);

    if (resendFrame) {
        sendFrame(ll.frame, fd, ll.frame_length);
        resendFrame = FALSE;
    }

    if (read_value >= 0) {
        // Cancels alarm
        alarm(0);
        finish = TRUE;
    }
}
```

Figura 5 - Ciclo onde é feito o reenvio da trama

llread

```
/*
 * Receive data in packet.
 * @param fd File descriptor for the serial port
 * @param packet Packet to store the data read in
 * @return Number of chars read; negative value when error
 */
int llread(int fd, unsigned char *packet);
```

Figura 6 - Função *llread*

Esta função é apenas chamada pelo receptor, de modo a receber tramas de informação provenientes do transmissor e é composta pelas seguintes fases:

- Leitura da trama de informação enviada pelo transmissor (**readInformationFrame**);
- Reversão do *byte stuffing* na trama recebida, previamente feito pelo transmissor (**byteDes-tuffing**);
- Verificação do BCC2 (**createBCC2**, e comparação com o BCC2 recebido na trama);
- Determinação da resposta que deve ser dada ao transmissor, tendo em conta o resultado dos passos anteriores (*RR* ou *REJ*);
- Criação e envio da trama de supervisão correta ao transmissor (*RR* ou *REJ*, com o número de sequência de trama correto) (**createSupervisionFrame** e **sendFrame**);
- Saída da função, se tudo funcionou corretamente e o packet passado como argumento foi preenchido, ou tentativa de nova leitura de uma trama de informação, se algum erro ocorreu.

llclose

```
/*
 * Close previously opened connection.
 * @param fd File descriptor for the serial port
 * @return Positive value when sucess; negative value when error
 */
int llclose(int fd);
```

Figura 7 - Função *llclose*

Tal como o *llopen*, esta função chama uma das duas funções auxiliares, **lICloseTransmitter** ou **lICloseReceiver**, dependendo do valor do parâmetro **role** guardado em **ll** (variável global).

```
/**  
 * Closes the connection for the transmitter  
 * @param fd File descriptor for the serial port  
 * @return Positive value when sucess; negative value when error  
 */  
int llCloseTransmitter(int fd);
```

Figura 8 - Função llCloseTransmitter

```
/**  
 * Closes the connection for the receiver  
 * @param fd File descriptor for the serial port  
 * @return Positive value when sucess; negative value when error  
 */  
int llCloseReceiver(int fd);
```

Figura 10 - Função llCloseReceiver

O transmissor envia um *DISC*, recebe um *DISC* do recetor e envia um *UA*, fechando a ligação através da porta de série. O recetor recebe o *DISC* do transmissor, envia um *DISC*, e recebe um *UA*, fechando também a ligação posteriormente se não houver erros. Foi também implementada a ocorrência de timeouts e reenvio de tramas caso surjam erros.

Funções auxiliares

Para além das funções descritas acima, várias outras funções auxiliares foram implementadas, como **createSupervisionFrame**, **createInformationFrame**, **readSupervisionFrame** e **readInformationFrame**, que podem ser observadas melhor no ficheiro *ll_aux.c*. As últimas duas referidas fazem a leitura da trama utilizando uma máquina de estados, que pode ser melhor observada no ficheiro *state_machine.c*, nos anexos.

Protocolo de aplicação

O protocolo da aplicação tem como funções principais as funções **sendFile** e **receiveFile**, que são chamadas nas função **applicationLayer**. Estas funções são responsáveis por todas as tarefas que cada máquina deve executar.

applicationLayer

```
/**  
 * Application layer main function.  
 * @param serialPort: Serial port name (e.g., /dev/ttyS0).  
 * @param role: Application role {"tx", "rx"}.  
 * @param baudrate: Baudrate of the serial port.  
 * @param nTries: Application role {"tx", "rx"}.  
 * @param timeout: Maximum number of frame retries.  
 * @param 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);
```

Figura 11 - Função applicationLayer

Esta função recebe os parâmetros introduzidos pelo utilizador em *main.c*, guarda os mesmos na variável global **ll** e de seguida chama uma das funções **sendFile** ou **receiveFile** dependendo do **role** recebido no parâmetro da função.

sendFile

```
/***
 * Function to send a file, using the serial port
 * @param filename Name of the file to be sent through the serial port
 * @return 0 if it was sucessful; negative value otherwise
 */
int.sendFile(const char *filename);
```

Figura 12 - Função *sendFile*

Esta função tem como principais funcionalidades:

- Abertura do ficheiro a ser enviado, com o nome recebido no argumento **filename**;
- Estabelecimento da ligação com o recetor, utilizando a função **llopen**;
- Envio do pacote de controlo *START*, usando **llwrite** (contém o nome e tamanho do ficheiro a enviar);
- Fragmentação do ficheiro em pequenos pacotes, cujo tamanho é dependente do tamanho máximo especificado para os pacotes de dados, e posteriormente o envio de cada pacote, por ordem, ao recetor, utilizando **llwrite**;
- Envio do pacote de controlo *END*, usando **llwrite** (contém o nome e tamanho do ficheiro a enviar);
- Término da ligação com o recetor, usando **llclose**.

receiveFile

```
/***
 * Function to receive a file, that was sent using the serial port
 * @param filename Name of the file to be saved, received through the serial port
 * @return 0 if it was sucessful; negative value otherwise
 */
int.receiveFile(const char *filename);
```

Figura 13 - Função *receiveFile*

Esta função tem como principais funcionalidades:

- Estabelecimento da ligação com o transmissor, utilizando a função **llopen**;
- Receção do pacote de controlo *START*, usando **llread** (contém o nome e tamanho do ficheiro a enviar);

- Abertura de um novo ficheiro com o nome indicado no argumento **filename**;
- Receção, pacote a pacote, com **llread**, dos fragmentos de ficheiro enviados pelo transmissor (a ordem é confirmada através do número de sequência). Cada fragmento é escrito no ficheiro aberto, de modo a juntar os fragmentos recebidos para recriar o ficheiro;
- Receção do pacote de controlo *END*, usando **llread**, (contém o nome e tamanho do ficheiro a enviar);
- Comparação do tamanho do ficheiro recriado com o tamanho de ficheiro passado nos pacotes de controlo;
- Comparação das informações passadas nos pacotes de controlo *START* e *END*, para detectar quaisquer possíveis erros no envio por parte do transmissor;
- Término da ligação com o transmissor, usando **llclose**.

Funções auxiliares

Para além das funções descritas acima, foram feitas algumas funções auxiliares, mais precisamente para a construção dos pacotes de controlo e de dados (**buildDataPacket** e **buildControlPacket**), e para o parse destes pacotes, de modo a retirar as informações dos mesmos (**parseControlPacket** e **parseDataPacket**). Foi feita também uma função auxiliar para descobrir o tamanho de um ficheiro (**getFileSize**). A implementação destas funções pode ser encontrada em `app_aux.c`, nos anexos.

Protocolo Stop & Wait

Foi utilizado o protocolo *Stop & Wait* para o controlo de erros. Este protocolo requer que após a transmissão de uma trama de informação, o transmissor espere por uma confirmação positiva por parte do recetor, denominada por *acknowledgment*, ACK. Deste modo, quando o recetor recebe a trama, se esta não tiver erros, confirma com ACK; caso contrário, é enviado um NACK (*negative acknowledgment*). Do lado do transmissor, ao receber um ACK, este enviará a trama seguinte (se existir), contudo se receber um NACK irá reenviar a mesma trama. Com o intuito de prevenir o mau funcionamento do programa, caso a trama I ou as respostas ACK ou NACK não sejam recebidas, o programa deve implementar um mecanismo de timeout, fazendo com que a trama I seja reenviada.

De maneira a que o recetor consiga identificar se uma trama é nova ou se é duplicada, e também para o transmissor saber qual a trama enviada a que um ACK se refere, tanto as tramas I como os ACKs devem ser numerados, com um 0 ou 1, sendo que estes valores são utilizados alternadamente (por exemplo, ACK(1) significa que o recetor está à espera de uma trama I(1)).

Nesta aplicação, as tramas de informação são identificadas com um 0 ou 1, sendo que a resposta do recetor pode ser exclusivamente RR (receive ready), equivalente a ACK, ou REJ (reject), equivalente a NACK. Estes últimos dois também se encontram devidamente identificados com o seu número, 0 ou 1, dependendo do número da trama de informação recebida. Abaixo deixo um exemplo de uma trama enviada pelo transmissor e a resposta equivalente dada pelo recetor, respetivamente:

- $N_s = 0$, Sem erros \rightarrow RR ($N_r = 1$); Com erros \rightarrow REJ ($N_r = 0$)
- $N_s = 1$, Sem erros \rightarrow RR ($N_r = 0$); Com erros \rightarrow REJ ($N_r = 1$)

Legenda: N_s é o número de sequência e N_r é o número de resposta

Validação

Testes efetuados

De modo a verificar se programa se comporta como esperado, foram efetuados os seguintes testes:

- Interrupção da ligação por cabo entre as portas de série
- Geração de ruído na ligação entre as portas de série
- Envio de vários ficheiros, com diferentes tamanhos
- Envio de ficheiros com diferentes taxas de simulação de erros (**FER**)
- Envio de ficheiros com a simulação de diferentes distâncias entre as máquinas e daí diferentes tempos de propagação de *I-frames* (**T_prop**)
- Envio de ficheiros com diferentes valores de **Baudrate** (**C** - capacidade de ligação)
- Envio de ficheiros com diferentes tamanhos nas *I-frames* (**size of I-frame**)

Resultados obtidos

Todos os testes que foram efetuados foram concluídos com sucesso, verificando-se o comportamento esperado.

Eficiência do protocolo de ligação de dados

A eficiência **S** foi calculada dividindo o débito recebido **R** (received bitrate) pela capacidade da ligação **C** (baudrate): $S = R / C$. Por sua vez, **R** resulta da divisão do número de bits recebidos pelo tempo da transferência.

Análise dos testes efetuados

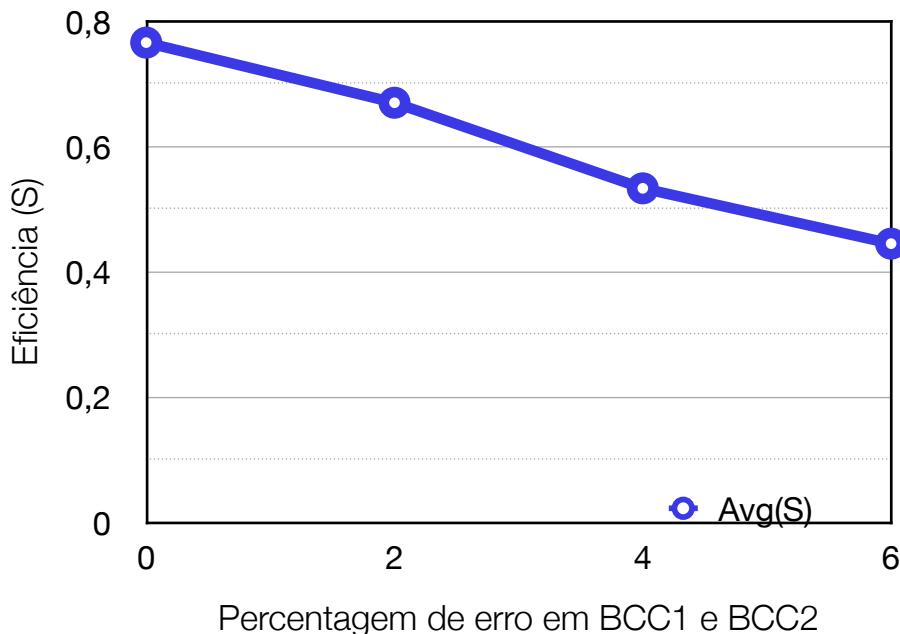
Cada valor de eficiência (**S**) obtido resulta da média de 4 ensaios.

Os testes foram executados com um ficheiro de imagem com 147KB (feup.jpg), de modo a que o grande número de tramas tornasse o efeito das variações testadas homogéneo em todos os ensaios. Ao testar a variação do tamanho da imagem, o tamanho máximo de pacote de informação utilizado foi de 1024B.

As tabelas que contém os valores com os quais os gráficos foram gerados encontram-se no fim do relatório, nos anexos.

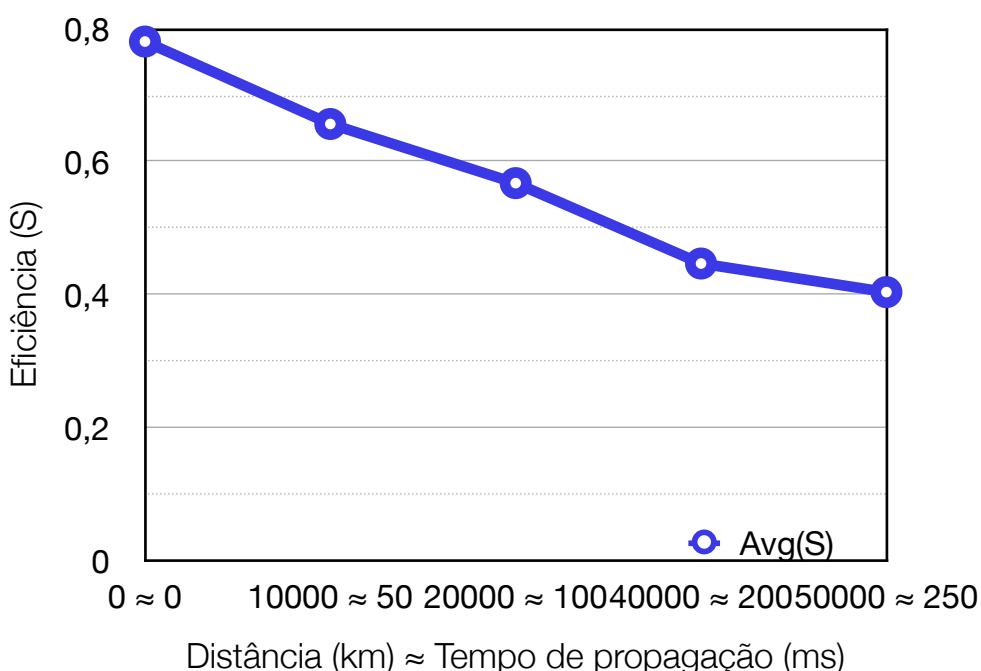
Variação do FER

Com o gráfico a seguir, do valor da eficiência S em função da percentagem de erros simulados, podemos concluir que o FER tem um impacto significativo na eficiência do programa. Isto deve-se, primariamente, ao facto de que, quando é gerado um erro no *BCC1*, irá ocorrer um *timeout*, que resultará na ausência de resposta por parte do receptor, por um número previamente definido de segundos (**3 segundos**), o que afeta negativamente o tempo de execução. Já os erros no *BCC2* não têm um impacto tão grande, pois apenas causam o reenvio da trama, que é imediato.



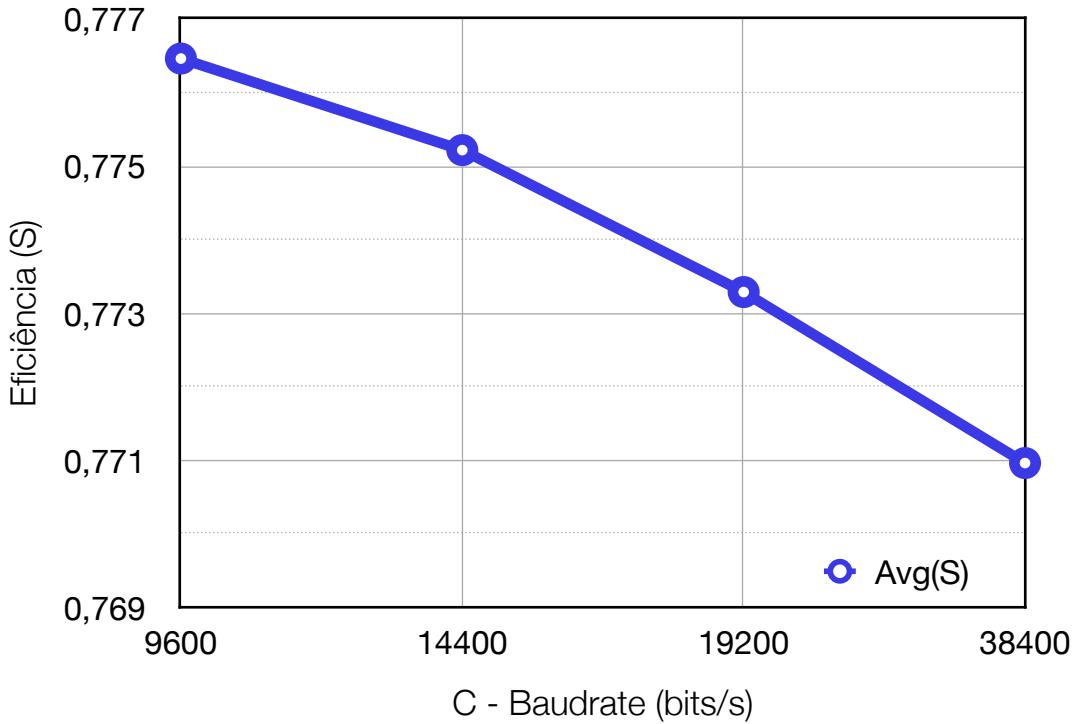
Variação do tempo de propagação (T_prop)

Como seria de esperar, com o aumento do tempo de propagação de cada trama de informação, o programa será menos eficiente. Isto deve-se naturalmente ao facto de a aplicação passar mais tempo sem enviar tramas uma vez que é simulado que o envio/recepção de uma trama demora mais. Utilizando a função `usleep()`, foi simulado um aumento no tempo de propagação, que vai aumentando desde 0 ms até 250 ms, que correspondem aproximadamente a valores de distâncias reais que vão desde 0 km até 50000 km. Estes valores foram calculados e posteriormente arredondados, assumindo que a velocidade no cabo é de cerca de dois terços da velocidade da luz.



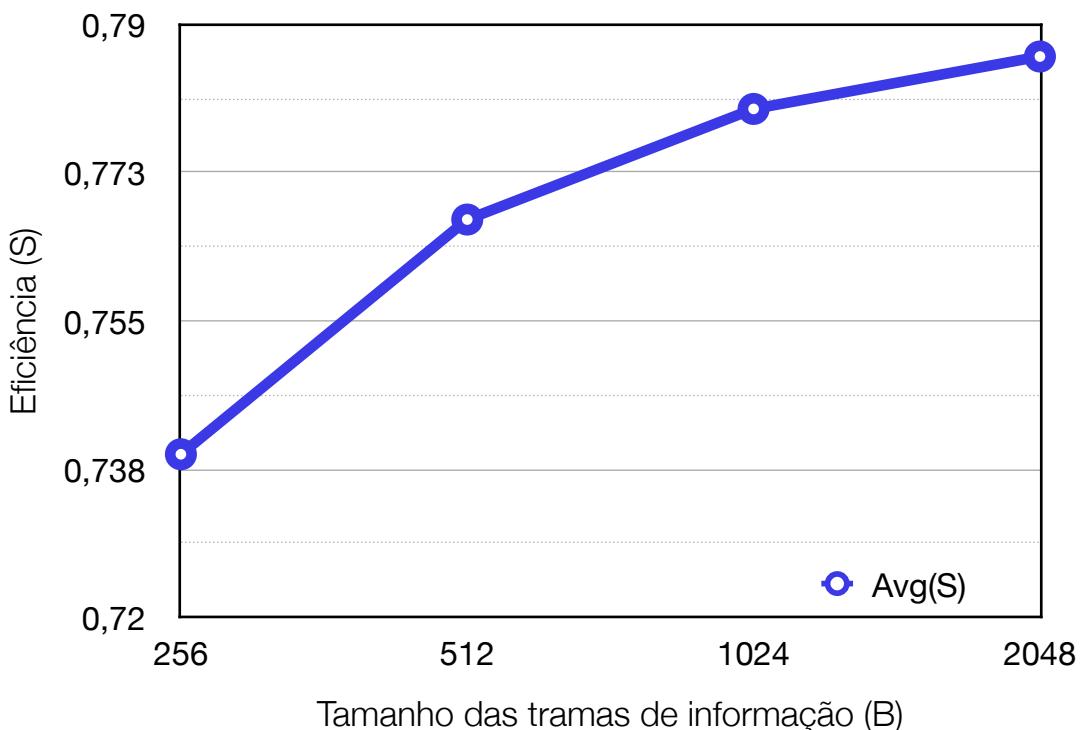
Variação do baudrate (C – capacidade de ligação)

Com este gráfico, podemos concluir que, com o aumento da capacidade de ligação, diminui a eficiência.



Variação do tamanho das tramas de informação

Por último, temos o gráfico correspondente à variação de S, com base no tamanho das armas de informação. Este evidencia que, quanto maior o tamanho das tramas de informação, mais eficiente será o programa. Isto verifica-se porque cada envio contém mais informação, o que reduz o número de tramas a enviar. Isto também leva a que o *timeout* ocorra menos vezes, em caso de erro.



Conclusão

Este trabalho permitiu-me compreender o **protocolo de ligação de dados**, incidindo sobre a estrutura das tramas e o processo de encapsulamento, envio e receção da informação.

Adicionalmente, destaco a importância da **independência entre camadas**, sendo que ambas as camadas do programa respeitam esse conceito. A camada da ligação de dados não recorre, de todo, a qualquer processamento desenvolvido na camada da aplicação. Já a camada da aplicação não conhece quaisquer detalhes da implementação da camada da ligação de dados, apenas sabendo como utilizar as suas funcionalidades.

Numa nota sobre o número de páginas, reconheço que não consegui cumprir com o que foi pedido, contudo penso que o relatório tenha ficado mais organizado e legível desta maneira. Peço a compreensão do professor neste assunto.

Anexos

Código Fonte

app_aux.c

```
#include "app_aux.h"

int buildDataPacket(unsigned char *packetBuffer, int sequenceNumber, unsigned char *dataBuffer, int dataLength) {
    packetBuffer[0] = CTRL_DATA;

    packetBuffer[1] = (unsigned char)sequenceNumber;

    int l1, l2;
    // number of octets (K = 256 * L2 + L1) in the data field
    l1 = dataLength % 256;
    l2 = dataLength / 256;

    packetBuffer[2] = (unsigned char)l2;
    packetBuffer[3] = (unsigned char)l1;

    for (int i = 0; i < dataLength; i++)
        packetBuffer[i + 4] = dataBuffer[i];

    return dataLength + 4;
}

int parseDataPacket(unsigned char *packetBuffer, unsigned char *data, int *sequenceNumber) {
    // checks if the control field is correct
    if (packetBuffer[0] != CTRL_DATA)
        return -1;

    *sequenceNumber = (int)packetBuffer[1];

    // number of octets (K = 256 * L2 + L1) in the data field
    int dataLength = 256 * (int)packetBuffer[2] + (int)packetBuffer[3];

    for (int i = 0; i < dataLength; i++)
    {
        data[i] = packetBuffer[i + 4];
    }

    return 0;
}
```

```

int buildControlPacket(unsigned char *packetBuffer, unsigned char controlByte, int fileSize, const char *fileName) {

    packetBuffer[0] = controlByte;

    packetBuffer[1] = TYPE_FILESIZE; // T1

    int length = 0, currentFileSize = fileSize;

    // cycle to separate file size in bytes (V1)
    while (currentFileSize > 0){
        int rest = currentFileSize % 256;
        int div = currentFileSize / 256;
        length++;

        // shifts all bytes to the right, to make space for the new byte
        for (unsigned int i = 2 + length; i > 3; i--)
            packetBuffer[i] = packetBuffer[i - 1];

        packetBuffer[3] = (unsigned char)rest;

        currentFileSize = div;
    }

    packetBuffer[2] = (unsigned char)length; // L1

    packetBuffer[3 + length] = TYPE_FILENAME; // T2

    packetBuffer[4 + length] = (unsigned char)(strlen(fileName) + 1); // file name length (including '\0') (L2)

    int fileNameStart = 5 + length; // beginning of V2

    for (unsigned int j = 0; j < (strlen(fileName) + 1); j++) // strlen(fileName) + 1 in order to add the '\0' char
        packetBuffer[fileNameStart + j] = fileName[j];

    return 3 + length + 2 + strlen(fileName) + 1; // total length of the packet
}

```

```

int parseControlPacket(unsigned char *packetBuffer, int *fileSize, char *fileName) {

    // checks if the control field is correct
    if (packetBuffer[0] != CTRL_START && packetBuffer[0] != CTRL_END)
        return -1;

    int length1, length2;

    if (packetBuffer[1] == TYPE_FILESIZE) { // T1

        *fileSize = 0;
        length1 = (int)packetBuffer[2]; // L1

        for (int i = 0; i < length1; i++) // V1
            *fileSize = *fileSize * 256 + (int)packetBuffer[3 + i];
    }
    else
        return -1;

    int fileNameStart = 5 + length1;

    if (packetBuffer[fileNameStart - 2] == TYPE_FILENAME) { // T2

        length2 = (int)packetBuffer[fileNameStart - 1]; // L2

        for (int i = 0; i < length2; i++) // V2
            fileName[i] = packetBuffer[fileNameStart + i];
    }
    else
        return -1;

    return 0;
}

```

```

int getFileSize(FILE *fp) {

    int lsize;

    fseek(fp, 0, SEEK_END);
    lsize = (int)f.tell(fp);
    rewind(fp);

    return lsize;
}

```

app_aux.h

```

/***
 * Function that builds an application data packet
 * @param packetBuffer Buffer that will have the final contents of the packet
 * @param sequenceNumber Sequence number of the packet
 * @param dataBuffer Buffer with the data to fill the packet
 * @param dataLength Length of the data in the buffer
 * @return Length of the packet buffer
 */
int buildDataPacket(unsigned char *packetBuffer, int sequenceNumber, unsigned char *dataBuffer, int dataLength);

/***
 * Function that parses the data packets
 * @param packetBuffer Buffer with the data packet
 * @param data Pointer to the file data packet extracted, to be returned by the function
 * @param sequenceNumber Pointer to the sequence number of the packet, to be returned by the function
 * @return 0 if it was sucessful; negative value otherwise
 */
int parseDataPacket(unsigned char *packetBuffer, unsigned char *data, int *sequenceNumber);

/***
 * Function that builds a control packet
 * @param packetBuffer Buffer that will have the final contents of the packet
 * @param controlByte Can be CTRL_START or CTRL_END, to show if the control packet indicates the beginning or end of the file
 * @param fileSize Size of the file, in bytes
 * @param fileName Name of the file
 * @return Length of the packet buffer
 */
int buildControlPacket(unsigned char *packetBuffer, unsigned char controlByte, int fileSize, const char *fileName);

/***
 * Function that parses the control packets
 * @param packetBuffer Buffer with the control packet
 * @param fileSize Pointer to the size of the file, to be returned by the function
 * @param fileName Pointer to the name of the file, to be returned by the function
 * @return 0 if it was sucessful; negative value otherwise
 */
int parseControlPacket(unsigned char *packetBuffer, int *fileSize, char *fileName);

/***
 * Auxiliary function to obtain the size of a file, from its file pointer
 * @param fp File pointer to the file
 * @return Size of the file in question
 */
int getFileSize(FILE *fp);

```

application_layer.c

```
// Application layer protocol implementation

#include "application_layer.h"

void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename)
{
    // store linklayer info in ll struct to be used in the link layer protocol
    strcpy(ll.serialPort, serialPort);
    if (strcmp("tx", role) == 0)
        ll.role = LlTx;
    else if (strcmp("rx", role) == 0)
        ll.role = LlRx;
    else {
        perror("Invalid role");
        return;
    }
    ll.baudRate = baudRate;
    ll.nRetransmissions = nTries;
    ll.timeout = timeout;

    // File Transfer
    if (ll.role == LlTx) {
        if (sendFile(filename) < 0) {
            printf("\nxxxxxx sendFile failed xxxxxx\n\n");
            return;
        }
    }
    else {
        if (receiveFile(filename) < 0) {
            printf("\nxxxxxx receiveFile failed xxxxxx\n\n");
            return;
        }
    }
}
```

```

int receiveFile(const char *filename)
{
    struct timeval start, end;

    // use current time as seed for random generator
    //srand(time(0));

    // store file descriptor in ApplicationLayer struct
    al.fileDescriptor = llopen();
    if (al.fileDescriptor == -1) {
        printf("\nxxxxxx llopen failed xxxxx\n\n");
        return -1;
    }
    else
        printf("\n----- llopen complete -----\\n\\n");

    int packetSize;
    unsigned char packetBuffer[MAX_SIZE_PACK], data[MAX_SIZE_DATA];

    // register start time
    gettimeofday(&start, NULL);

    //usleep(50000);
    packetSize = llread(al.fileDescriptor, packetBuffer);
    if (packetSize < 0)
        return -1;

    int numBitsReceived = 0;

    numBitsReceived += packetSize * 8;

    int packet_filesize_start, packet_filesize_end; // used to compare the filesize stored in the packet
    char packet_filename_start[255], packet_filename_end[255]; // used to compare the filename stored in the packet

    // received START control packet
    if (packetBuffer[0] == CTRL_START) {
        if (parseControlPacket(packetBuffer, &packet_filesize_start, packet_filename_start) < 0)
            return -1;
    }
    else
        return -1;
}

```

```

// creates and opens file to write on
FILE* fp;
fp = fopen(filename, "w");

if (fp == NULL) {
    perror(filename);
    return -1;
}

int sequenceNumber, expectedSequenceNumber = 0, dataLength;

printf("\n----- Receiving file ... -----\\n\\n");

// read through received data packets (file data) until receiving the END packet
while (TRUE)
{
    //usleep(50000);
    packetSize = llread(al.fileDescriptor, packetBuffer);
    if (packetSize < 0)
        return -1;

    numBitsReceived += packetSize * 8;

    if (packetBuffer[0] == CTRL_DATA) { // received data packet

        if (parseDataPacket(packetBuffer, data, &sequenceNumber) < 0)
            return -1;

        if (expectedSequenceNumber != sequenceNumber) {
            printf("receiveFile ERROR: Sequence number does not match\\n");
            return -1;
        }

        expectedSequenceNumber = (expectedSequenceNumber + 1) % 256;

        dataLength = packetSize - 4;

        // writes to the file the content read from the serial port
        if (fwrite(data, sizeof(unsigned char), dataLength, fp) != dataLength) {
            return -1;
        }
    }
    // received END packet, indicating the end of the file transfer
    else if (packetBuffer[0] == CTRL_END)
        break;
}

printf("\n----- Finished receiving file! -----\\n\\n");

```

```

gettimeofday(&end, NULL);
// get the difference in seconds, multiply by a million
double transferTime = (end.tv_sec - start.tv_sec) * 1e6;
// then add the difference in microseconds, finally divide by a million to convert result to seconds
transferTime = (transferTime + (end.tv_usec - start.tv_usec)) * 1e-6;

printf("\n----- Protocol Efficiency ----- \n\n");

printf("Number of bits received = %d\n", numBitsReceived);
printf("File transfer time (seconds) = %lf\n", transferTime);
double R = numBitsReceived / transferTime;
double baudRate = 38400.0;
double S = R / baudRate;

printf("\n\n Received bitrate R = %lf", R);
printf("\n\n Link capacity (Baudrate) C = %lf", baudRate);
printf("\n\n Efficiency statistic S (S = R / C) = %lf\n\n", S);

if (getFileSize(fp) != packet_filesize_start) {
    printf("receiveFile ERROR: file size specified in start packet does not match received file size\n");
    return -1;
}

if (parseControlPacket(packetBuffer, &packet_filesize_end, packet_filename_end) < 0)
    return -1;

// checks if filesize and filename are the same in START and END packets
if((packet_filesize_start != packet_filesize_end) || (strcmp(packet_filename_start, packet_filename_end) != 0)){
    printf("receiveFile ERROR: Specified info (filename and/or filesize) in START and END packets does not match");
    return -1;
}

if (llclose(al.fileDescriptor) == -1) {
    printf("\nxxxxxx llclose failed xxxxxx\n\n");
    return -1;
}
else
    printf("\n----- llclose complete ----- \n\n");

if (fclose(fp) != 0)
    return -1;

return 0;
}

```

```
int sendFile(const char *filename)
{
    // opens file to read from
    FILE* fp;
    fp = fopen(filename, "r");

    if (fp == NULL) {
        perror(filename);
        return -1;
    }

    // store file descriptor in ApplicationLayer struct
    al.fileDescriptor = llopen();
    if (al.fileDescriptor == -1) {
        printf("\nxxxxxx llopen failed xxxxxx\n\n");
        return -1;
    }
    else
        printf("\n----- llopen complete -----\\n\\n");

    unsigned char packetBuffer[MAX_SIZE_PACK];
    int fileSize = getFileSize(fp);

    int packetSize = buildControlPacket(packetBuffer, CTRL_START, fileSize, filename);

    // sends control START packet, to indicate the start of the file transfer
    if (llwrite(al.fileDescriptor, packetBuffer, packetSize) < 0) {
        fclose(fp);
        return -1;
    }

    unsigned char data[MAX_SIZE_DATA];
    int read_length;
    int sequenceNumber = 0;
```

```

printf("\n----- Sending file ... -----\\n\\n");

// read through the file to be sent
while (TRUE) {

    read_length = fread(data, sizeof(unsigned char), MAX_SIZE_DATA, fp);

    if (read_length != MAX_SIZE_DATA) {
        if (feof(fp)) { // reached end of file

            packetSize = buildDataPacket(packetBuffer, sequenceNumber, data, read_length);
            sequenceNumber = (sequenceNumber + 1) % 256;

            // sends the last data frame
            if (llwrite(al.fileDescriptor, packetBuffer, packetSize) < 0) {
                fclose(fp);
                return -1;
            }

            break;
        }
        else {
            perror("ERROR while reading file data");
            return -1;
        }
    }

    packetSize = buildDataPacket(packetBuffer, sequenceNumber, data, read_length);
    sequenceNumber = (sequenceNumber + 1) % 256;

    // sends a data frame
    if (llwrite(al.fileDescriptor, packetBuffer, packetSize) < 0) {
        fclose(fp);
        return -1;
    }
}

packetSize = buildControlPacket(packetBuffer, CTRL_END, fileSize, filename);

// sends control END packet, indicating the end of the file transfer
if (llwrite(al.fileDescriptor, packetBuffer, packetSize) < 0) {
    fclose(fp);
    return -1;
}

printf("\n----- File has been sent! -----\\n\\n");

if (llclose(al.fileDescriptor) == -1) {
    printf("\\nxxxxxx llclose failed xxxxxx\\n\\n");
    return -1;
}
else
    printf("\\n----- llclose complete -----\\n\\n");

if (fclose(fp) != 0)
    return -1;

return 0;
}

```

application_layer.h

```

typedef struct {
    int fileDescriptor; /* File Descriptor correspondent to the serial port */
} ApplicationLayer;

// global variables
ApplicationLayer al;

/***
 * Application layer main function.
 * @param serialPort: Serial port name (e.g., /dev/ttyS0).
 * @param role: Application role {"tx", "rx"}.
 * @param baudrate: Baudrate of the serial port.
 * @param nTries: Application role {"tx", "rx"}.
 * @param timeout: Maximum number of frame retries.
 * @param 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);

/***
 * Function to receive a file, that was sent using the serial port
 * @param filename Name of the file to be saved, received through the serial port
 * @return 0 if it was sucessful; negative value otherwise
 */
int receiveFile(const char *filename);

/***
 * Function to send a file,using the serial port
 * @param filename Name of the file to be sent through the serial port
 * @return 0 if it was sucessful; negative value otherwise
 */
int sendFile(const char *filename);

```

link_layer.c

```

// Link layer protocol implementation

#include "link_layer.h"

///////////////////////////////
// LLOPEN
/////////////////////////////
int llopen()
{
    int fd, returnFd;

    printf("llopen: Opening connection...\n");
    // Open non canonical connection
    if ( (fd = openNonCanonical(VTIME_VALUE, VMIN_VALUE)) == -1)
        return -1;

    // installs alarm handler
    alarmHandlerInstaller();

    if (ll.role == LlTx) // Transmitter
    {
        returnFd = llOpenTransmitter(fd);
        if (returnFd == -1) {
            closeNonCanonical(oldtio, fd);
            return -1;
        }
        else
            return returnFd;
    }
    else if (ll.role == LlRx) // Receiver
    {
        returnFd = llOpenReceiver(fd);
        if (returnFd == -1) {
            closeNonCanonical(oldtio, fd);
            return -1;
        }
        else
            return returnFd;
    }

    perror("Invalid role");
    closeNonCanonical(oldtio, fd);
    return -1;
}

```

```
//////////  
// LLOPEN – Receiver  
//////////  
int llOpenReceiver(int fd)  
{  
    unsigned char expectedByte[1];  
    ll.frame_length = BUF_SIZE_SUP;  
    expectedByte[0] = SET;  
  
    if (readSupervisionFrame(ll.frame, fd, expectedByte, 1, ADD_SEND) == -1)  
        return -1;  
  
    printf("llopen: Received SET frame\n");  
  
    if (createSupervisionFrame(ll.frame, UA, LlRx) != 0)  
        return -1;  
  
    // send SET frame to receiver  
    if (sendFrame(ll.frame, fd, ll.frame_length) == -1)  
        return -1;  
  
    printf("llopen: Sent UA frame\n");  
  
    return fd;  
}
```

```

// LLOPEN - Transmitter
///////////////////////////////
int llOpenTransmitter(int fd)
{
    unsigned char responseBuffer[BUF_SIZE_SUP];
    ll.frame_length = BUF_SIZE_SUP;

    // creates SET frame
    if (createSupervisionFrame(ll.frame, SET, LlTx) != 0)
        return -1;

    // send SET frame to receiver
    if (sendFrame(ll.frame, fd, ll.frame_length) == -1)
        return -1;

    printf("llopen: Sent SET frame\n");

    int read_value = -1;
    finish = FALSE;
    num_retr = 0;
    resendFrame = FALSE;

    // Sets alarm
    alarm(ll.timeout);

    unsigned char expectedByte[1];
    expectedByte[0] = UA;

    while (finish != TRUE) {
        // read_value contains the index the expectedByte found by the state machine if it succeeds, else -1
        read_value = readSupervisionFrame(responseBuffer, fd, expectedByte, 1, ADD_SEND);
        if (resendFrame) {
            sendFrame(ll.frame, fd, ll.frame_length);
            resendFrame = FALSE;
        }

        if (read_value >= 0) {
            // Cancels alarm
            alarm(0);
            finish = TRUE;
        }
    }

    if (read_value == -1) {
        printf("llopen ERROR: Closing file descriptor\n");
        return -1;
    }

    printf("llopen: Received UA frame\n");
}

return fd;
}

```

```

///////////////////////////////
// LLWRITE
/////////////////////////////
int llwrite(int fd, unsigned char *packet, int length)
{
    unsigned char responseBuffer[BUF_SIZE_SUP]; // buffer to receive the response
    unsigned char controlByte; // controlByte --> information frame number

    if (ll.sequenceNumber == 0)
        controlByte = I_0;
    else
        controlByte = I_1;

    if (createInformationFrame(ll.frame, controlByte, packet, length) != 0) {
        closeNonCanonical(olddtio, fd);
        return -1;
    }

    int fullLength; // frame length after stuffing

    if ((fullLength = byteStuffing(ll.frame, length)) < 0) {
        closeNonCanonical(olddtio, fd);
        return -1;
    }

    ll.frame_length = fullLength;
    int numBytesWritten; // number of bytes written

    int dataSent = FALSE; // indicates whether the data has been sent

    while (!dataSent)
    {
        if ((numBytesWritten = sendFrame(ll.frame, fd, ll.frame_length)) == -1) {
            closeNonCanonical(olddtio, fd);
            return -1;
        }
    }

    printf("llwrite: Sent I frame\n");

    int read_value = -1;
    finish = FALSE;
    num_retr = 0;
    resendFrame = FALSE;

    alarm(ll.timeout);
}

```

```

unsigned char expectedBytes[2];

if (controlByte == I_0) {
    expectedBytes[0] = RR_1;
    expectedBytes[1] = REJ_0;
}
else if (controlByte == I_1) {
    expectedBytes[0] = RR_0;
    expectedBytes[1] = REJ_1;
}

while (finish != TRUE) {
    // read_value contains the index the expectedByte found by the state machine if it succeeds, else -1
    read_value = readSupervisionFrame(responseBuffer, fd, expectedBytes, 2, ADD_SEND);

    if (resendFrame) {
        sendFrame(ll.frame, fd, ll.frame_length);
        resendFrame = FALSE;
    }

    if (read_value >= 0) {
        // Cancels alarm
        alarm(0);
        finish = TRUE;
    }
}

if (read_value == -1) {
    printf("llwrite ERROR: Closing file descriptor\n");
    closeNonCanonical(oldtio, fd);
    return -1;
}

if (read_value == 0) // read a RR
    dataSent = TRUE;
else // read a REJ
    dataSent = FALSE;

printf("llwrite: Received response frame, C = %x\n", responseBuffer[2]);
}

if (ll.sequenceNumber == 0)
    ll.sequenceNumber = 1;
else if (ll.sequenceNumber == 1)
    ll.sequenceNumber = 0;
else
    return -1;

return (numBytesWritten - 6); // length of the data packet sent to the receiver
}

```

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

int llread(int fd, unsigned char *packet)
{
    // use current time as seed for random generator
    srand(time(0));

    int numBytesRead; // number of bytes read
    unsigned char expectedBytes[2];
    expectedBytes[0] = I_0;
    expectedBytes[1] = I_1;

    int read_value;

    int isBufferFull = FALSE;

    while (!isBufferFull) {

        read_value = readInformationFrame(ll.frame, fd, expectedBytes, 2, ADD_SEND);

        printf("llread: Received Info frame\n");

        if ((numBytesRead = byteDestuffing(ll.frame, read_value)) < 0) {
            closeNonCanonical(olddtio, fd);
            return -1;
        }

        int controlByteRead;
        if (ll.frame[2] == I_0)
            controlByteRead = 0;
        else if (ll.frame[2] == I_1)
            controlByteRead = 1;

        unsigned char responseByte;
```

```

if (ll.frame[numBytesRead - 2] == createBCC_2(&ll.frame[DATA_START], numBytesRead - 6)) { // checks if bcc2 is correct

    if (controlByteRead == ll.sequenceNumber) { // Expected frame (sequence number matches the number in control byte)
        // transfers information to the packet
        for (int i = 0; i < numBytesRead - 6; i++)
            packet[i] = ll.frame[DATA_START + i];

        isBufferFull = TRUE;
    }
    // updates the response and sequence number whether it was the Expected frame or a Duplicated frame
    // (in which case, we discard the rest of the information)
    // Expected frame: store the information and request the next frame
    // Duplicated frame: request the next frame
    if (controlByteRead == 0) {
        responseByte = RR_1;
        ll.sequenceNumber = 1;
    }
    else {
        responseByte = RR_0;
        ll.sequenceNumber = 0;
    }
}
else { // if bcc2 is not correct

    if (controlByteRead != ll.sequenceNumber) { // duplicated frame; discards information and requests the next frame

        if (controlByteRead == 0) {
            responseByte = RR_1;
            ll.sequenceNumber = 1;
        }
        else {
            responseByte = RR_0;
            ll.sequenceNumber = 0;
        }
    }
    else { // ignores frame data (because of error) and rejects this frame

        if (controlByteRead == 0) {
            responseByte = REJ_0;
            ll.sequenceNumber = 0;
        }
        else {
            responseByte = REJ_1;
            ll.sequenceNumber = 1;
        }
    }
}
}

```

```

if (createSupervisionFrame(ll.frame, responseByte, LlRx) != 0) {
    closeNonCanonical(oldtio, fd);
    return -1;
}

ll.frame_length = BUF_SIZE_SUP;

// send RR/REJ frame to receiver
if (sendFrame(ll.frame, fd, ll.frame_length) == -1) {
    closeNonCanonical(oldtio, fd);
    return -1;
}

printf("llread: Sent response frame, C = %x\n", ll.frame[2]);

}

return (numBytesRead - 6); // number of bytes of the data packet read
}

```

```
//////////  
// LLCLOSE  
//////////  
int llclose(int fd)  
{  
    if (ll.role == LlTx) {  
        if (llCloseTransmitter(fd) == -1) {  
            closeNonCanonical(olddio, fd);  
            return -1;  
        }  
    }  
    else if (ll.role == LlRx) {  
        if (llCloseReceiver(fd) == -1) {  
            closeNonCanonical(olddio, fd);  
            return -1;  
        }  
    }  
    else {  
        perror("Invalid role");  
        return -1;  
    }  
  
    // Close non canonical connection  
    if (closeNonCanonical(olddio, fd) == -1)  
        return -1;  
  
    return 1;  
}
```

```

int llCloseReceiver(int fd)
{
    unsigned char responseBuffer[BUF_SIZE_SUP], expectedByte[1];
    ll.frame_length = BUF_SIZE_SUP;
    expectedByte[0] = DISC;

    if (readSupervisionFrame(ll.frame, fd, expectedByte, 1, ADD_SEND) == -1)
        return -1;

    printf("llclose: Received DISC frame\n");

    // creates DISC frame
    if (createSupervisionFrame(ll.frame, DISC, LlRx) != 0)
        return -1;

    // send DISC frame to receiver
    if (sendFrame(ll.frame, fd, ll.frame_length) == -1)
        return -1;

    printf("llclose: Sent DISC frame\n");

    int read_value = -1;
    finish = FALSE;
    num_retr = 0;
    resendFrame = FALSE;

    alarm(ll.timeout);

    expectedByte[0] = UA;

    while (finish != TRUE) {

        // read_value contains the index the expectedByte found by the state machine if it succeeds, else -1
        read_value = readSupervisionFrame(responseBuffer, fd, expectedByte, 1, ADD_REC);

        if (resendFrame)
        {
            sendFrame(ll.frame, fd, ll.frame_length);
            resendFrame = FALSE;
        }

        if (read_value >= 0)
        {
            // Cancels alarm
            alarm(0);
            finish = TRUE;
        }
    }

    if (read_value == -1) {
        printf("llclose ERROR: Closing file descriptor\n");
        return -1;
    }

    printf("llclose: Received UA frame\n");

    return 0;
}

```

```

int llCloseTransmitter(int fd)
{
    unsigned char responseBuffer[BUF_SIZE_SUP], expectedByte[1];
    ll.frame_length = BUF_SIZE_SUP;

    // creates DISC frame
    if (createSupervisionFrame(ll.frame, DISC, LlTx) != 0)
        return -1;

    // send DISC frame to receiver
    if (sendFrame(ll.frame, fd, ll.frame_length) == -1)
        return -1;

    printf("llclose: Sent DISC frame\n");

    int read_value = -1;
    finish = FALSE;
    num_retr = 0;
    resendFrame = FALSE;

    // Sets alarm
    alarm(ll.timeout);

    expectedByte[0] = DISC;
}

```

```

while (finish != TRUE) {
    // read_value contains the index the expectedByte found by the state machine if it succeeds, else -1
    read_value = readSupervisionFrame(responseBuffer, fd, expectedByte, 1, ADD_REC);

    if (resendFrame) {
        sendFrame(ll.frame, fd, ll.frame_length);
        resendFrame = FALSE;
    }

    if (read_value >= 0) {
        // Cancels alarm
        alarm(0);
        finish = TRUE;
    }
}

if (read_value == -1) {
    printf("llclose ERROR: Closing file descriptor\n");
    return -1;
}

printf("llclose: Received DISC frame\n");

// creates UA frame
if (createSupervisionFrame(ll.frame, UA, LlTx) != 0)
    return -1;

// send DISC frame to receiver
if (sendFrame(ll.frame, fd, ll.frame_length) == -1)
    return -1;

printf("llclose: Sent UA frame\n");

return 0;
}

```

link_layer.h

```

typedef enum
{
    LlTx, /* Transmitter */
    LlRx, /* Receiver */
} LinkLayerRole;

typedef struct
{
    char serialPort[50]; /* Device /dev/ttySx, x = 0, 1 */
    LinkLayerRole role; /* Role played in transfer: Transmitter ou Receiver */
    int baudRate; /* Rate at which information is transferred in the channel*/
    unsigned int nRetransmissions; /* Number of retries in case of error */
    unsigned int timeout; /* Value of the timer in seconds */
    unsigned char frame[MAX_SIZE_FRAME]; /* Frame */
    unsigned int frame_length; /* Current frame size */
    unsigned int sequenceNumber; /* Frame sequence number: 0, 1 */
} LinkLayer;

// global variables
LinkLayer ll;
struct termios oldtio;

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


/***
 * Open a connection using the "port" parameters defined in struct LinkLayer (global variable).
 * @return File descriptor; negative value otherwise
 */
int llopen();

/***
 * Opens the connection for the receiver
 * @param fd File descriptor for the serial port
 * @return File descriptor; negative value otherwise
 */
int llOpenReceiver(int fd);

```

```
/**  
 * Opens the connection for the transmitter  
 * @param fd File descriptor for the serial port  
 * @return File descriptor; negative value otherwise  
 */  
int llOpenTransmitter(int fd);  
  
/**  
 * Transfer data stored in packet.  
 * @param fd File descriptor for the serial port  
 * @param packet Packet (data) to be written  
 * @param length Size of the packet  
 * @return Number of chars written; negative value otherwise  
 */  
int llwrite(int fd, unsigned char *packet, int length);  
  
/**  
 * Receive data in packet.  
 * @param fd File descriptor for the serial port  
 * @param packet Packet to store the data read in  
 * @return Number of chars read; negative value otherwise  
 */  
int llread(int fd, unsigned char *packet);  
  
/**  
 * Close previously opened connection.  
 * @param fd File descriptor for the serial port  
 * @return Positive value if it was sucessful; negative value otherwise  
 */  
int llclose(int fd);  
  
/**  
 * Closes the connection for the receiver  
 * @param fd File descriptor for the serial port  
 * @return Positive value if it was sucessful; negative value otherwise  
 */  
int llCloseReceiver(int fd);  
  
/**  
 * Closes the connection for the transmitter  
 * @param fd File descriptor for the serial port  
 * @return Positive value if it was sucessful; negative value otherwise  
 */  
int llCloseTransmitter(int fd);
```

```

int openNonCanonical(int vtime, int vmin) {
    // Program usage: Uses either COM1 or COM2
    // Open serial port device for reading and writing, and not as controlling tty
    // because we don't want to get killed if linenoise sends CTRL-C.
    int fd = open(ll.serialPort, O_RDWR | O_NOCTTY);

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

    struct termios newtio;

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

    // Clear struct for new port settings
    memset(&newtio, 0, sizeof(newtio));

    newtio.c_cflag = ll.baudRate | CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;

    // Set input mode (non-canonical, no echo,...)
    newtio.c_lflag = 0;
    newtio.c_cc[VTIME] = vtime;
    newtio.c_cc[VMIN] = vmin;

    // VTIME e VMIN should be changed in order to protect with a
    // timeout the reception of the following character(s)

    // Now clean the line and activate the settings for the port
    // tcflush() discards data written to the object referred to
    // by fd but not transmitted, or data received but not read,
    // depending on the value of queue_selector:
    // TCIFLUSH – flushes data received but not read.
    tcflush(fd, TCI0FLUSH);

    // Set new port settings
    if (tcsetattr(fd, TCSANOW, &newtio) == -1)
    {
        perror("tcsetattr");
        exit(-1);
    }

    printf("openNonCanonical: new termios structure set\n");

    return fd;
}

```

ll_aux.c

```
// Link layer protocol auxiliary funtions

#include "ll_aux.h"

void alarmHandler(int signal) {

    if (num_retr < ll.nRetransmissions) {
        resendFrame = TRUE;
        printf("Alarm: Timeout, Sending frame again... (Number of retransmissions: %d)\n", num_retr+1);
        alarm(ll.timeout);
        num_retr++;
    }
    else {
        printf("Alarm: Number of retransmissions exceeded\n");
        finish = TRUE;
    }
}

void alarmHandlerInstaller() {
    // Set alarm function handler
    (void)signal(SIGALRM, alarmHandler);
}
```

```

int closeNonCanonical(struct termios oldtio, int fd) {

    // Wait until all bytes have been written to the serial port
    sleep(1);

    // Restore the old port settings
    if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
    {
        perror("tcsetattr");
        exit(-1);
    }

    close(fd);

    return 0;
}

unsigned char createBCC(unsigned char a, unsigned char c) {
    return a ^ c;
}

unsigned char createBCC_2(unsigned char* frame, int length) {

    unsigned char bcc2 = frame[0];

    for(int i = 1; i < length; i++){
        bcc2 = bcc2 ^ frame[i];
    }

    return bcc2;
}

```

```

int byteStuffing(unsigned char* frame, int length) {

    // allocates space for aux buffer (length of the packet + 6 bytes for the frame header and tail)
    unsigned char aux[length + 6];

    // transfers info from the frame to aux
    for(int i = 0; i < (length + 6); i++){
        aux[i] = frame[i];
    }

    int finalLength = DATA_START;
    // parses aux buffer, and fills in correctly the frame buffer
    for(int i = DATA_START; i < (length + 6); i++){
        // If the octet 0x7E (FLAG) occurs inside the frame, the octet is replaced by the sequence: 0x7D 0x5E (ESCAPE_BYTE BYTE_STUFFING_FLAG)
        if(aux[i] == FLAG && i != (length + 5)) {
            frame[finalLength] = ESCAPE_BYTE;
            frame[finalLength+1] = BYTE_STUFFING_FLAG;
            finalLength = finalLength + 2;
        }
        // If the octet 0x7D (ESCAPE_BYTE) occurs inside the frame, the octet is replaced by the sequence: 0x7D 0x5D (ESCAPE_BYTE BYTE_STUFFING_ESCAPE)
        else if(aux[i] == ESCAPE_BYTE && i != (length + 5)) {
            frame[finalLength] = ESCAPE_BYTE;
            frame[finalLength+1] = BYTE_STUFFING_ESCAPE;
            finalLength = finalLength + 2;
        }
        else{
            frame[finalLength] = aux[i];
            finalLength++;
        }
    }

    return finalLength;
}

```

```

int byteDestuffing(unsigned char* frame, int length) {

    // allocates space for the max possible frame length read (length of the data packet + bcc2, already with stuffing, plus the other 5 bytes in the frame)
    unsigned char aux[length + 5];

    // copies the content of the frame (with stuffing) to the aux frame
    for(int i = 0; i < (length + 5); i++) {
        aux[i] = frame[i];
    }

    int finalLength = DATA_START;

    // iterates through the aux buffer, and fills the frame buffer with destuffed content
    for(int i = DATA_START; i < (length + 5); i++) {

        if(aux[i] == ESCAPE_BYTE){
            // If the octet 0x5D (BYTE_STUFFING_ESCAPE) occurs inside the frame, the octet is replaced by the escape byte: 0x7D
            if (aux[i+1] == BYTE_STUFFING_ESCAPE) {
                frame[finalLength] = ESCAPE_BYTE;
            }
            // If the octet 0xE5 (BYTE_STUFFING_FLAG) occurs inside the frame, the octet is replaced by the flag: 0x7E
            else if(aux[i+1] == BYTE_STUFFING_FLAG) {
                frame[finalLength] = FLAG;
            }
            i++;
            finalLength++;
        }
        else{
            frame[finalLength] = aux[i];
            finalLength++;
        }
    }

    return finalLength;
}

```

```

int createSupervisionFrame(unsigned char* frame, unsigned char controlField, int role) {

    frame[0] = FLAG;

    /* ADDRESS FIELD */
    if(role == LlTx) {
        // If the frame is sent by the Sender, the Address field will be ADD_SEND (0x03)
        if(controlField == SET || controlField == DISC) {
            frame[1] = ADD_SEND;
        }
        // If the frame is an answer from the Sender, the Address field will be ADD_REC (0x01)
        else if(controlField == UA || controlField == RR_0 || controlField == REJ_0 || controlField == RR_1 || controlField == REJ_1 ) {
            frame[1] = ADD_REC;
        }
        else return -1;
    }
    else if(role == LlRx) {
        // If the frame is sent by the Receiver, the Address field will be ADD_REC (0x01)
        if(controlField == SET || controlField == DISC) {
            frame[1] = ADD_REC;
        }
        // If the frame is an answer from the Receiver, the Address field will be ADD_SEND (0x03)
        else if(controlField == UA || controlField == RR_0 || controlField == REJ_0 || controlField == RR_1 || controlField == REJ_1 ) {
            frame[1] = ADD_SEND;
        }
        else return -1;
    }
    else return -1;

    frame[2] = controlField;

    frame[3] = createBCC(frame[1], frame[2]);

    frame[4] = FLAG;

    return 0;
}

```

```

int createInformationFrame(unsigned char* frame, unsigned char controlField, unsigned char* infoField, int infoFieldLength) {

    frame[0] = FLAG;
    frame[1] = ADD_SEND;
    frame[2] = controlField;
    frame[3] = createBCC(frame[1], frame[2]);

    for(int i = 0; i < infoFieldLength; i++) {
        frame[i + 4] = infoField[i];
    }

    frame[infoFieldLength + 4] = createBCC_2(infoField, infoFieldLength);
    frame[infoFieldLength + 5] = FLAG;

    return 0;
}

int readSupervisionFrame(unsigned char* frame, int fd, unsigned char* expectedBytes, int expectedBytesLength, unsigned char addressByte) {

    state_machine *st = create_state_machine(expectedBytes, expectedBytesLength, addressByte);
    unsigned char byte;

    // run state machine
    while(st->state != STOP && !finish && !resendFrame) {
        if(read(fd, &byte, sizeof(unsigned char)) > 0)
            event_handler(st, byte, frame, SUPERVISION);
    }

    // return index found in state machine
    int ret = st->foundIndex;

    destroy_state_machine(st);

    if(finish || resendFrame)
        return -1;

    return ret;
}

```

```

int readInformationFrame(unsigned char* frame, int fd, unsigned char* expectedBytes, int expectedBytesLength, unsigned char addressByte) {

    state_machine *st = create_state_machine(expectedBytes, expectedBytesLength, addressByte);
    unsigned char byte;

    // run state machine
    while(st->state != STOP) {
        if(read(fd, &byte, sizeof(unsigned char)) > 0)
            event_handler(st, byte, frame, INFORMATION);
    }

    // return dataLength = length of the data packet sent from the application on the transmitter side (includes data packet + bcc2, with stuffing)
    int ret = st->dataLength;

    destroy_state_machine(st);

    return ret;
}

int sendFrame(unsigned char* frame, int fd, int length) {
    int n; // number of bytes written

    if( (n = write(fd, frame, length)) <= 0){
        return -1;
    }

    return n;
}

```

ll_aux.h

```

// global variables
// finish - TRUE when the alarm has completed all retransmissions so we close the connection
// num_retr - current number of retransmissions
// resendFrame - boolean to determine whether to resend a frame or not based on the alarm
int finish, num_retr, resendFrame;

/**
 * Handles the alarm signal
 * @param signal Signal that is received
 */
void alarmHandler(int signal);

/**
 * Function to install the alarm handler, using sigaction
 */
void alarmHandlerInstaller();

/**
 * Function to open the file descriptor through which to execute the serial port communications,
 * in the non-canonical mode, according to the serial port file transfer protocol
 * @param vtime Value to be assigned to the VTIME field of the new settings – time between bytes read
 * @param vmin Value to be assigned to the VMIN field of the new settings – minimum amount of bytes to read
 * @return File descriptor that was opened with the given port; negative value otherwise
 */
int openNonCanonical(int vtime, int vmin);

/**
 * Function to open the file descriptor through which to execute the serial port communications,
 * in the non-canonical mode, according to the serial port file transfer protocol
 * @param fd File descriptor that was opened with the given port
 * @param oldtio termios used to store oldtio to restore the old port settings when closing the connection
 * @return 0 if it was sucessful; negative value otherwise
 */
int closeNonCanonical(struct termios oldtio, int fd);

/**
 * Function to create the Block Check Character relative to the Address and Control fields
 * @param a Address Character of the frame
 * @param c Control Character of the frame
 * @return Expected value for the Block Check Character
 */
unsigned char createBCC(unsigned char a, unsigned char c);

/**
 * Function to create the Block Check Character relative to the Data Characters of the frame
 * @param frame Frame position where the Data starts
 * @param length Number of Data Characters to process
 * @return Expected value for the Block Check Character
 */
unsigned char createBCC_2(unsigned char* frame, int length);

```

```

/**
 * Function to apply byte stuffing to the Data Characters of a frame
 * @param frame Address of the frame
 * @param length Number of Data Characters to process
 * @return Length of the new frame, post byte stuffing
 */
int byteStuffing(unsigned char* frame, int length);

/**
 * Function to reverse the byte stuffing applied to the Data Characters of a frame
 * @param frame Address of the frame
 * @param length Number of Data Characters to process
 * @return Length of the new frame, post byte destuffing
 */
int byteDestuffing(unsigned char* frame, int length);

/**
 * Function to create a supervision frame for the serial port file transfer protocol
 * @param frame Address where the frame will be stored
 * @param controlField Control field of the supervision frame
 * @param role Role for which to create the frame, marking the difference between the Transmitter and the Receiver
 * @return 0 if successful; negative if an error occurs
 */
int createSupervisionFrame(unsigned char* frame, unsigned char controlField, int role);

/**
 * Function to create an information frame for the serial port file transfer protocol
 * @param frame Address where the frame will be stored
 * @param controlField Control field of the supervision frame
 * @param infoField Start address of the information to be inserted into the information frame
 * @param infoFieldLength Number of data characters to be inserted into the information frame
 * @return Returns 0, as there is no place at which an error can occur
 */
int createInformationFrame(unsigned char* frame, unsigned char controlField, unsigned char* infoField, int infoFieldLength);

/**
 * Function to read a supervision frame, sent according to the serial port file transfer protocol
 * @param frame Address where the frame is being stored
 * @param fd File descriptor from which to read the frame
 * @param expectedBytes Array containing the possible expected control bytes of the frame
 * @param expectedBytesLength Number of possible expected control bytes of the frame
 * @param addressByte Address from which a frame is expected
 * @return Index of the expected byte found, in the expectedBytes array
 */
int readSupervisionFrame(unsigned char* frame, int fd, unsigned char* expectedBytes, int expectedBytesLength, unsigned char addressByte);

```

```

/**
 * Function to read an information frame, sent according to the serial port file transfer protocol
 * @param frame Address where the frame is being stored
 * @param fd File descriptor from which to read the frame
 * @param expectedBytes Array containing the possible expected control bytes of the frame
 * @param expectedBytesLength Number of possible expected control bytes of the frame
 * @param addressByte Address from which a frame is expected
 * @return Length of the data packet sent, including byte stuffing and BCC2
 */
int readInformationFrame(unsigned char* frame, int fd, unsigned char* expectedBytes, int expectedBytesLength, unsigned char addressByte);

/**
 * Function to send a frame to the designated file descriptor
 * @param frame Start address of the frame to be sent
 * @param fd File descriptor to which to write the information
 * @param length Size of the frame to be sent (size of information to be written)
 * @return Number of bytes written if successful; negative if an error occurs
 */
int sendFrame(unsigned char* frame, int fd, int length);

```

state_machine.c

```
int is_expected(unsigned char byte, state_machine* sm) {
    for (int i = 0; i < sm->expectedBytesLength; i++) {
        if (sm->expectedBytes[i] == byte)
            return i;
    }

    return -1;
}

state_machine* create_state_machine(unsigned char* expectedBytes, int expectedBytesLength, unsigned char addressByte) {
    state_machine* sm = malloc(sizeof(state_machine));
    sm->state = START;
    sm->expectedBytes = expectedBytes;
    sm->expectedBytesLength = expectedBytesLength;
    sm->addressByte = addressByte;
    sm->dataLength = 0;
    return sm;
}
```

```

void event_handler(state_machine* sm, unsigned char byte, unsigned char* frame, int mode) {

    static int i = 0;

    if(mode == SUPERVISION){
        switch(sm->state) {

            case START:
                if (byte == FLAG) {
                    sm->state = FLAG_RCV;
                    frame[0] = byte;
                }
                break;

            case FLAG_RCV:
                if (byte == FLAG)
                    break;
                else if (byte == sm->addressByte) {
                    sm->state = A_RCV;
                    frame[1] = byte;
                }
                else
                    sm->state = START;
                break;
        }
    }
}

```

```

case A_RCV:
    if (byte == FLAG)
        sm->state = FLAG_RCV;
    else {
        int n;
        if ((n = is_expected(byte, sm))>=0){
            sm->state = C_RCV;
            sm->foundIndex = n;
            frame[2] = byte;
        }
        else
            sm->state = START;
    }
    break;

case C_RCV:
    if (byte == createBCC(frame[1], frame[2])){
        sm->state = BCC_OK;
        frame[3] = byte;
    }

    else if (byte == FLAG)
        sm->state = FLAG_RCV;
    else
        sm->state = START;
    break;

case BCC_OK:
    if (byte == FLAG){
        sm->state = STOP;
        frame[4] = byte;
    }
    else
        sm->state = START;
    break;

default:
    break;
}
}

```

```

else if(mode == INFORMATION){
    switch(sm->state) {
        case START:
            i = 0;
            if (byte == FLAG) {
                sm->state = FLAG_RCV;
                frame[i++] = byte;
            }
            break;

        case FLAG_RCV:
            if (byte == FLAG)
                break;
            else if (byte == sm->addressByte) {
                sm->state = A_RCV;
                frame[i++] = byte;
            }
            else {
                sm->state = START;
                i = (int) sm->state;
            }
            break;

        case A_RCV:
            if (byte == FLAG) {
                sm->state = FLAG_RCV;
                i = (int) sm->state;
            }
            else {
                if (is_expected(byte, sm) >= 0){
                    sm->state = C_RCV;
                    frame[i++] = byte;
                }
                else {
                    sm->state = START;
                    i = (int) sm->state;
                }
            }
            break;
    }
}

```

```

case C_RCV:
    if (byte == createBCC(frame[1], frame[2])){
        sm->state = BCC_OK;
        frame[i++] = byte;
    }
    else {
        if (byte == FLAG)
            sm->state = FLAG_RCV;
        else
            sm->state = START;

        i = (int) sm->state;
    }
    break;

case BCC_OK:
    if(byte == FLAG){
        frame[i] = byte;
        sm->state = STOP;
        sm->dataLength = i-4;
    }
    else{
        frame[i++] = byte;
    }
    break;

default:
    break;
}

void destroy_state_machine(state_machine* sm) {
    free(sm);
}

```

state_machine.h

```

typedef enum{
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    BCC_OK,
    STOP,
} State;

typedef struct {
    State state; /* Current state */
    unsigned char* expectedBytes; /* Array of possible bytes that are expected in the frame */
    int expectedBytesLength; /* Number of possible bytes that are expected in the frame */
    unsigned char addressByte; /* Address from which the frame is expected */
    int foundIndex; /* Index of the array where the byte was found; negative value otherwise */
    int dataLength; /* Length of the data packet sent from the application on the transmitter side (includes data packet + bcc2, with stuffing) */
} state_machine;

/***
 * Function to check if a byte is contained in the state machine's expectedBytes field, indicating it is expected
 * @param byte Byte to be checked
 * @param sm State machine for which to check
 * @return Index of the array where the byte was found; negative value otherwise
 */
int is_expected(unsigned char byte, state_machine* sm);

```

```

/***
 * Function to create a state machine, with the given attributes
 * @param expectedBytes Possible bytes that are expected in the frame
 * @param expectedBytesLength Number of possible bytes that are expected in the frame
 * @param addressByte Address from which the frame is expected
 * @return Pointer to the new state machine object
 */
state_machine* create_state_machine(unsigned char* expectedBytes, int expectedBytesLength, unsigned char addressByte);

/***
 * Function to update the state machine according to the bytes read
 * @param sm State machine to be updated
 * @param byte Last byte to have been read of the frame
 * @param frame Address where the frame is being stored
 * @param mode Type of frame (Supervision or Information)
 */
void event_handler(state_machine* sm, unsigned char byte, unsigned char* frame, int mode);

/***
 * Function to free the memory allocated to a state machine object
 * @param sm State machine to be destroyed
 */
void destroy_state_machine(state_machine* sm);

```

macros.h

```

// MISC
#define FALSE 0
#define TRUE 1

// ---- macros for both layers ----

#define MAX_SIZE_DATA 1024 // max size of a data packet
#define MAX_SIZE_PACK (MAX_SIZE_DATA + 4) // max size of a data packet + 4 bytes for packet head
#define MAX_SIZE (MAX_SIZE_PACK + 6) // max size of data in a frame + 4 bytes for packet head, + 6 bytes for frame header and tail
#define MAX_SIZE_FRAME (((MAX_SIZE_PACK + 1) * 2) + 5) // max size of a frame, with byte stuffing, is ((1029 * 2) + 5)
// 1029 -> all bytes that can suffer byte stuffing (and therefore be "duplicated"), which are the packet and the BCC2
// 5 -> the bytes that certainly won't suffer any byte stuffing (flags, bcc1, address and control bytes)

#define BUF_SIZE_SUP 5 // size of a supervision frame

#define BAUDRATE 38400 // 38400 is the normal value
#define _POSIX_SOURCE 1 // POSIX compliant source

#define N_TRIES 3 // Number of tries before the alarm stops
#define TIMEOUT 4 // Timeout for the alarm

// ---- macros for data link layer ----

#define SUPERVISION 0 // Supervision frame
#define INFORMATION 1 // Information frame

#define FLAG 0x7E // Synchronisation: start or end of frame
#define ADD_SEND 0x03 // Address field in frames that are commands sent by the Sender or replies sent by the Receiver
#define ADD_REC 0x01 // Address field in frames that are commands sent by the Receiver or replies sent by the Sender
#define I_0 0x00 // Information frame number 0
#define I_1 0x40 // Information frame number 1
#define SET 0x03 // Set Up --> sent by the transmitter to initiate a connection
#define DISC 0x0B // Disconnect --> indicate the termination of a connection
#define UA 0x07 // Unnumbered Acknowledgment --> confirmation to the reception of a valid supervision frame
#define RR_0 0X05 // Receive Ready for number 0 --> indication sent by the Receiver that it is ready to receive an information frame number 0
#define RR_1 0X85 // Receive Ready for number 1 --> indication sent by the Receiver that it is ready to receive an information frame number 1
#define REJ_0 0x01 // Reject --> indication sent by the Receiver that it rejects an information frame number 0
#define REJ_1 0x81 // Reject --> indication sent by the Receiver that it rejects an information frame number 1
#define VTIME_VALUE 0
#define VMIN_VALUE 0

```

```

#define BYTE_STUFFING_ESCAPE 0x5D // If the octet 0x7D (ESCAPE_BYTE) occurs inside the frame, the octet is replaced by the sequence: 0x7D 0x5D (ESCAPE_BYTE BYTE_STUFFING_ESCAPE)
#define BYTE_STUFFING_FLAG 0x5E // If the octet 0x7E (FLAG) occurs inside the frame, the octet is replaced by the sequence: 0x7D 0x5E (ESCAPE_BYTE BYTE_STUFFING_FLAG)
#define ESCAPE_BYTE 0x7D // escape octet

#define DATA_START 4 // start of the data field in an I-frame

// ---- macros for application layer ----

#define CTRL_DATA 0x01 // value for DATA in control field (C) meaning its a data packet
#define CTRL_START 0x02 // value for START in control field (C) meaning its the START control packet
#define CTRL_END 0x03 // value for END in control field (C) meaning its the END control packet

#define TYPE_FILESIZE 0x00 // value for FILESIZE in the type field (T), in the control packet
#define TYPE_FILENAME 0x01 // value for FILENAME the type field (T), in the control packet

```

main.c

```
// Arguments:  
//   $1: /dev/ttySxx  
//   $2: tx | rx  
//   $3: filename  
int main(int argc, char *argv[]){  
    if (argc < 4){  
        printf("Usage: %s /dev/ttySxx tx|rx filename\n", argv[0]);  
        exit(1);  
    }  
  
    const char *serialPort = argv[1];  
    const char *role = argv[2];  
    const char *filename = argv[3];  
  
    printf("Starting link-layer protocol application\n"  
           "  - Serial port: %s\n"  
           "  - Role: %s\n"  
           "  - Baudrate: %d\n"  
           "  - Number of tries: %d\n"  
           "  - Timeout: %d\n"  
           "  - Filename: %s\n",  
           serialPort,  
           role,  
           BAUDRATE,  
           N_TRIES,  
           TIMEOUT,  
           filename);  
  
    applicationLayer(serialPort, role, BAUDRATE, N_TRIES, TIMEOUT, filename);  
  
    return 0;  
}
```

Medições da Eficiência

Frame Error Ratio Table

FER (%)	0	2	4	6
	0,76545	0,65559	0,54112	0,45252
	0,76543	0,68738	0,53352	0,43173
	0,76621	0,67234	0,52192	0,44851
	0,76686	0,66431	0,53612	0,44645
Avg(S)	0,7659875	0,669905	0,53317	0,4448025

T_prop Table

Distance (km) ≈ T_prop (ms)	0 ≈ 0	10000 ≈ 50	20000 ≈ 100	40000 ≈ 200	50000 ≈ 250
	0,78082	0,65672	0,56772	0,44671	0,40370
	0,78086	0,65677	0,56774	0,44673	0,40369
	0,78080	0,65675	0,56771	0,44673	0,40368
	0,78083	0,65674	0,56778	0,44672	0,40371
Avg(S)	0,7808275	0,656745	0,5677375	0,4467225	0,403695

C (baudrate) Table

C (baudrate)	9600	14400	19200	38400
	0,77545	0,77479	0,77352	0,77015
	0,77561	0,77657	0,77358	0,77198
	0,77664	0,77398	0,77289	0,77120
	0,77814	0,77552	0,77315	0,77051
Avg(S)	0,77646	0,775215	0,773285	0,77096

Packet Size Table

I-Frame Size (B)	256	512	1024	2048
	0,73896	0,76768	0,78003	0,78621
	0,73967	0,76739	0,78001	0,78620
	0,73881	0,76638	0,78003	0,78622
	0,73964	0,76642	0,78002	0,78620
Avg(S)	0,73927	0,7669675	0,7800225	0,7862075