Protocolo de Ligação de Dados

(1º Trabalho Laboratorial)

Redes de Computadores

Licenciatura em Engenharia Informática e Computação

Rúben Tiago Oliveira Silva - up202005334@fe.up.pt

André Gabriel Correia Vieira - up202004159@fe.up.pt

Índice

Sumário

Este projeto foi realizado no âmbito da unidade curricular de Redes de Computadores e consistiu no desenvolvimento de uma aplicação simples para transferência de ficheiros, juntamente com a implementação do protocolo de nível de ligação de dados correspondente, de acordo com as indicações específicas.

O trabalho foi concluído com sucesso, uma vez que os objetivos estabelecidos foram alcançados, resultando numa aplicação funcional que permite a transferência de ficheiros entre dois sistemas ligados por um cabo série.

Introdução

O objetivo deste trabalho é criar um protocolo de nível de ligação de dados que possibilite a transferência de ficheiros entre dois computadores interligados por um cabo série, bem como o desenvolvimento de uma aplicação simples que permite ler ficheiros de um disco no computador emissor e gravá-los no disco do computador receptor, utilizando o protocolo desenvolvido. O presente relatório serve como complemento ao trabalho, explicando o processo e raciocínio associados, e documentando o trabalho realizado, apresentando a seguinte estrutura:

- **Introdução**: Inicialmente, são esclarecidos os objetivos tanto do trabalho quanto deste relatório, além de descrever a lógica seguida no documento, indicando o tipo de informação que cada secção subsequente contém.
- **Arquitetura**: Esta secção descreve os blocos funcionais do sistema, assim como as interfaces que os interligam, detalhando o papel de cada componente na arquitetura global.
- **Estrutura do Código:** São discutidas as APIs utilizadas, as principais estruturas de dados, bem como as funções chave, estabelecendo a conexão entre estas e a arquitetura do sistema.
- **Casos de Uso Principais:** Identificação dos cenários de uso típicos, acompanhados das sequências de chamada de funções correspondentes.
- **Protocolo de Ligação Lógica:** Aqui, são identificados os aspectos funcionais principais do protocolo, com uma descrição detalhada da estratégia de implementação e apresentação de trechos de código relevantes.
- **Protocolo de Aplicação:** Similar à secção anterior, mas focada no protocolo de aplicação, abordando a implementação dos seus principais aspectos funcionais e ilustrando com exemplos de código.
- **Validação:** Descrição dos testes realizados para validar o sistema, com uma apresentação quantitativa dos resultados obtidos.
- **Eficiência do Protocolo de Ligação de Dados:** Análise estatística da eficiência do protocolo.
- **Conclusões:** Síntese das informações das secções anteriores e considerações finais sobre o cumprimento dos objetivos de aprendizagem propostos pelo trabalho.

Arquitetura

O projeto foi desenvolvido seguindo uma estrutura em camadas, o que garante independência entre as mesmas. Na camada de ligação de dados, está implementado o protocolo responsável pela conexão e desconexão, assegurando uma transmissão de dados síncrona e com gestão de erros, utilizando, por exemplo, técnicas de 'byte stuffing'. Na camada de aplicação, procede-se à leitura e escrita de ficheiros e ao envio e receção de tramas, estando esta camada intrinsecamente ligada à camada de ligação de dados. No que toca à interface com o utilizador, no início do uso do programa, é possível ao utilizador definir os parâmetros de acordo com as suas preferências.

Estrutura do código

A camada de ligação é constituída pelas seguintes funções:

- void alarmManager(int signal);
- void stateDetermine(stateMachine *state, char byte, int user);
- int llopen(LinkLayer connectionParameters);
- void receiveACK(stateMachine *state, unsigned char byte, unsigned char *ack, int sequenceNum);
- int llwrite(const unsigned char *buf, int bufSize);
- int receiveData(unsigned char *packet, int sequenceNum, size_t *size_read);
- int llread(unsigned char *packet);
- void DISCStateDetermine(stateMachine *state, char byte);
- int llclose(int statistics);

As funções definidas no código formam uma cadeia de operações essenciais para a comunicação através de uma porta série. A função alarmManager gerencia timeouts e retransmissões, essencial para a robustez da comunicação. **stateDetermine()** atualiza a máquina de estados conforme os bytes recebidos, mantendo o sincronismo entre emissor e receptor. llopen inicializa a conexão, estabelecendo parâmetros para a transmissão de dados.

Durante a transmissão, **llwrite()** envia dados e **receiveACK()** processa as confirmações do receptor, enquanto **receiveData()** e **llread()** gerenciam a receção de dados. **DISCStateDetermine()** cuida da desconexão ordenada ao interpretar sinais de término de ligação, e **llclose ()**conclui a sessão, libertando recursos e compila estatísticas da transmissão. Juntas, estas funções asseguram uma troca de dados eficaz e confiável, articulando-se em um protocolo coeso de comunicação.

Já na camada de aplicação temos a estrutura de dados apresentada em baixo para guardar informação do ficheiro:

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

E as seguintes funções:

- int sendCPacket(int fd, unsigned char packetType, const char *filename);
- int sendDPacket(int fd, const char *filename);
- int sendPacket(int fd, unsigned char packetType, const char *filename);
- int receivePacket(int fd, const char *filename);
- void applicationLayer(const char *port, const char *role, int baudRate, int retries, int timeout, const char *filename);

No contexto do protocolo de camada de aplicação, as funções implementadas no código oferecem um mecanismo completo para a transmissão e receção de ficheiros através de uma porta série. A função **sendCPacket()** é responsável por construir e enviar pacotes de controlo que indicam o início ou o fim de uma transmissão de ficheiro, incluindo informações como o tamanho do ficheiro e o nome. A função **sendDPacket()** lida com o envio de dados do ficheiro em si, segmentando o conteúdo em pacotes e gerenciando a sequência de envios.

A função **sendPacket()** encapsula a lógica de decisão sobre qual tipo de pacote deve ser enviado, enquanto que **receivePacket()** gerencia a receção dos pacotes, reconstruindo o ficheiro original, no sistema receptor, a partir dos fragmentos recebidos. Finalmente, a função **applicationLayer()** configura a camada de aplicação, definindo o papel como emissor ou receptor, e orquestra o processo de envio e receção de acordo com esse papel, invocando as funções adequadas para a transferência do ficheiro. A interação entre essas funções assegura uma comunicação eficiente e a integridade dos dados transmitidos.

Casos de uso principais

A interface, que permite ao utilizador iniciar a aplicação de transferência de dados, e a transmissão em si do ficheiro entre dois computadores, são os dois principais casos de uso deste trabalho laboratorial.

No caso da interface, para iniciar a aplicação, deve introduzir um conjunto de argumentos específicos:

- A porta série a ser utilizada (ex: **/dev/ttyS0)**;
- A função de cada computador específico (**tx** no caso de ser emissor, e **rx** se for receptor);
- O nome do ficheiro a ser enviado, no caso do emissor, ou que vai ser criado, no caso do receptor.

Em relação à transferência de ficheiros entre dois computadores, ocorre a seguinte sequência de chamadas de funções:

- Configurar interface
- Estabelecimento da ligação
- Sinalizar começo de transferência com pacote que contém informação sobre o ficheiro
- Leitura do ficheiro a enviar e envio dos dados (pelo emissor)
- Leitura dos dados recebidos e escrita no novo ficheiro (pelo receptor)
- Sinalizar término de transferência com pacote específico
- Fecho da ligação

Protocolo de ligação lógica

O protocolo de ligação lógica, implementado no módulo **link_layer**, pretende fornecer um serviço de comunicação de dados fiável entre dois sistemas ligados por um meio (canal) de transmissão que, neste caso, é um cabo série.

llopen() `int llopen(LinkLayer connectionParameters);`

Uma das funcionalidades genéricas deste protocolo é o estabelecimento da ligação entre os dois sistemas, e a **llopen()** é a função responsável por essa parte. Nesta função, começa-se por estabelecer as definições da porta série, utilizando informação passada pela estrutura `connectionParameters`, e guardando as definições originais (para poderem ser reestabelecidas no final). De seguida, o emissor envia uma mensagem **SET,** e espera receber, durante um tempo predeterminado pelo utilizador, e com auxílio do alarme, uma mensagem **UA** por parte do receptor. No caso de o tempo definido expirar sem a receção da resposta **UA**, o emissor reitera o envio da trama **SET**, num ciclo que se repete até ao limite definido (pelo utilizador) de tentativas. Se a mensagem for recebida, a ligação fica estabelecida, e o programa segue corretamente. A concretização da comunicação é sinalizada pela receção bem-sucedida da trama **UA** dentro do número de tentativas predefinido, o que permite que a ligação seja efetivamente estabelecida e o programa siga de forma adequada.

llwrite() `int llwrite(const unsigned char *buf, int bufSize)`

Esta função é responsável por enviar os dados ao recetor. Recebe como parâmetros um buffer (e respetivo tamanho), e constrói a trama correspondente para enviar. Esta construção consiste em várias etapas:

- Criação do cabeçalho, comum em todas as tramas, e que delimita o começo de uma trama, com uma **FLAG**, numera a trama, e confere proteção com um código detetor de erros

```
message[0] = FLAG;
message[1] = A;
message[2] = sequenceNum << 7;
message[3] = BCC(A, sequenceNum << 7);
```

- Inserção dos dados presentes no buffer e *stuffing de bytes*, para evitar que sequências de controle (**FLAG** ou **ESC**) apareçam no meio dos dados

```
for (int j = 0; j < bufSize; j++)
    {
        switch (buf[j])
        {
        case ESC:
            message[i + j + 4] = ESC;
            message[i + j + 5] = ESC ^0x20;
            BCC2 = BCC(BCC2, ESC);
            i++;
            break;

        case FLAG:
            message[i + j + 4] = ESC;
```

```
            message[i + j + 5] = FLAG ^0x20;
            BCC2 = BCC(BCC2, FLAG);
            i++;
            break;

        default:
            message[i + j + 4] = buf[j];
            BCC2 = BCC(BCC2, buf[j]);
        }
    }
```

- Aumento da proteção contra erros com outro código detetor BCC2 e delimitação do final da trama com outra **FLAG**

```
message[i + bufSize + 4] = BCC2; // Frame footer with BCC
and FLAG
    message[i + bufSize + 5] = FLAG;
```

De seguida, **llwrite()** tenta enviar a trama criada, e aguarda confirmação da receção correta, utilizando uma máquina de estados, na forma de uma mensagem **ACK** (acknowledgment) com numeração correspondente. O ciclo de envio é parecido ao que ocorre na **llopen(),** aguarda a resposta e tenta retransmitir um número predefinido de vezes se ocorrer algum erro (quando recebe **NACK/RREJ,** negative acknowledgment).

llread() `int llread(unsigned char *packet)`

Em contrapartida à **llwrite()**, a função **llread()** está encarregue de ler os dados enviados pelo emissor e interpretá-los, devolvendo os dados a serem escritos no ficheiro recebido dentro da variável **packet** à camada de aplicação. Esta função utiliza uma função auxiliar, **receiveData(),** para ler a informação, fazer **byte destuffing**, e verificar se os dados estão corretos. Se a resposta for positiva, **llread()** responde ao emissor com a mensagem **ACK** esperada. No caso de se verificar algum erro ou uma mensagem repetida, os dados não são passados para a **packet**, e é enviada ou uma resposta **NACK** ou **ACK**, respetivamente.

```
while ((readStatus = receiveData(packet, sequenceNum, &size_read)) !=
TRUE)
    {
        // If the reply indicates an error (e.g., checksum mismatch)
        if (readStatus == 0)
        {
            printf("Sending RRej or NACK\n");

            // Send a NACK (negative acknowledgment)
            unsigned char NACK_C = NACK(1 - sequenceNum);
            unsigned char buf[] = {FLAG, A, NACK_C, BCC(A, NACK_C), F};
            write(fd, buf, 5);
        }
        // If the received message is a duplicate (e.g.,
retransmission)
        else
```

```
        {
            printf("Repeated message, sending ACK\n");

            // Send an ACK to prevent further retransmissions
            unsigned char ACK_C = ACK(sequenceNum);
            unsigned char buf[] = {FLAG, A, ACK_C, BCC(A, ACK_C), F};
            write(fd, buf, 5);
        }
    }

    // Once data is received correctly, send an ACK
    printf("Everything in order, sending ACK\n");
    sequenceNum = 1 - sequenceNum; // Toggle the sequence number
    unsigned char ACK_C = ACK(sequenceNum);
    unsigned char buf[] = {FLAG, A, ACK_C, BCC(A, ACK_C), F};
    write(fd, buf, 5);
```

llclose() `int llclose(int statistics)`

Finalmente, as funcionalidades de assegurar o término da ligação entre os dois sistemas e a reposição das definições originais da porta série são garantidas pela função **llclose().**

No caso do emissor (**LITx**):

1. Envia uma trama **DISC** para iniciar a desconexão.
2. Aguarda pela receção de uma trama **DISC** como confirmação.
3. Posteriormente, envia uma trama **UA** para concluir o procedimento.

No caso do recetor (**LIRx**):

1. Espera pela receção de uma trama **DISC** por parte do emissor.
2. Responde com outra trama **DISC** para confirmar a desconexão.
3. Aguarda por uma trama **UA**, sinalizando o término final da ligação por parte do emissor.

Após a troca bem-sucedida de tramas de controlo, as configurações do terminal são restauradas para o seu estado original, o descritor de ficheiro associado à comunicação série é fechado, e a função termina ao retornar 1 em caso de sucesso ou -1 se ocorrer algum erro durante a restauração das configurações do terminal ou no fechamento do ficheiro.

Protocolo de aplicação

No módulo **application_layer** pretendeu-se implementar um protocolo de aplicação muito simples para transferência de um ficheiro, usando o serviço fiável oferecido pelo protocolo de ligação lógica. Os principais aspetos funcionais e a estratégia de implementação são os seguintes:

applicationLayer()

```
void applicationLayer(const char *port, const char *role, int baudRate, int retries, int timeout, const char *filename)
```

A **applicationLayer()** é a função principal que gerencia a transferência de ficheiros, atuando como emissor ou receptor com base no papel atribuído. Inicializa as estruturas necessárias e chama **llopen()**, da camada lógica, para começar a comunicação, **sendPacket()** ou **receivePacket()** para transferência de dados, e **llclose()** para terminar a ligação.

Estrutura da Camada de Aplicação `struct applicationLayer`**:**

Esta estrutura mantém o descritor da porta série e o estado, indicando o modo de operação (emissor ou receptor).

Envio e Receção de Pacotes

```
int sendPacket(int fd, unsigned char packetType, const char *filename)
int receivePacket(int fd, const char *filename)
```

sendPacket() envia pacotes de controlo ou dados dependendo do tipo do pacote, enquanto **receivePacket()** lida com a receção de pacotes e grava os dados num ficheiro.

Pacotes de Controlo e de Dados

```
int sendCPacket(int fd, unsigned char packetType, const char *filename)
int sendDPacket(int fd, const char *filename)
```

Os pacotes de controlo, enviados pela **sendCPacket()**, contêm metadados sobre a transferência, como o tamanho e o nome do ficheiro, enquanto a função **sendDPacket()** utiliza um ciclo para ler um tamanho máximo de bytes predefinido do ficheiro a enviar, cria os pacotes de dados com essa informação e adiciona-lhes um cabeçalho que identifica a sequência e o tamanho do segmento.

Validação

Testes:

1. Transmissão do ficheiro dado para o trabalho (penguin.gif).
2. Transmissão do ficheiro dado com começo de transferência com receptor inicialmente não ligado.
3. Transmissão do ficheiro dado com interrupções de ligação.
4. Transmissão do ficheiro dado com ruído na ligação.
5. Transmissão do ficheiro dado com variação de baud rate.
6. Transmissão do ficheiro dado com variação de frame size.
7. Transmissão do ficheiro dado com variação de tempo de propagação (sleeps).
8. Transmissão do ficheiro dado com variação de frame error rate.

Resultados:

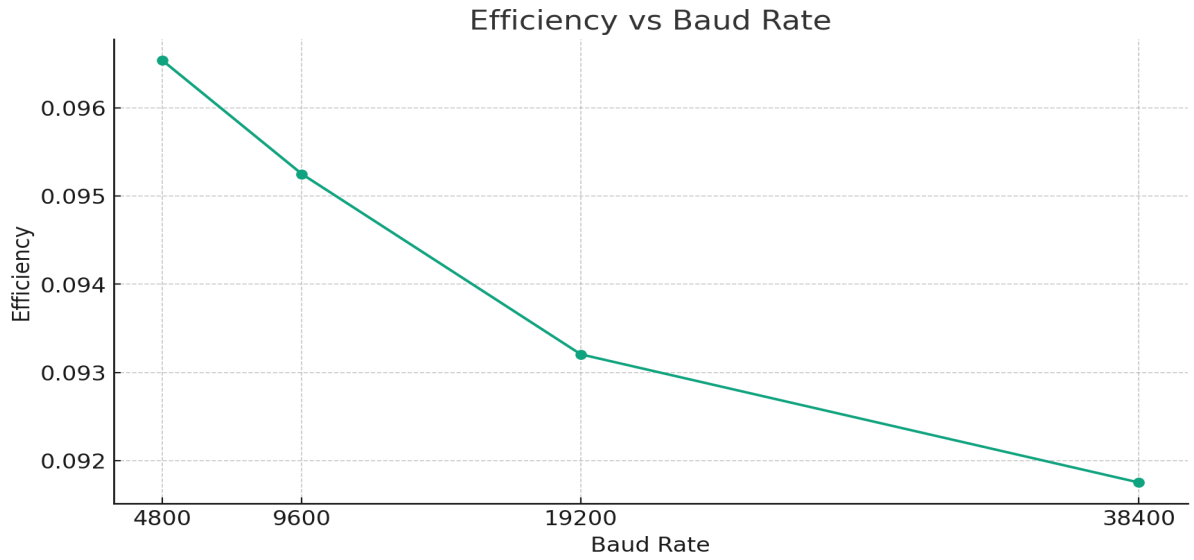
Todos os testes foram bem sucedidos, no entanto o caso número 6 falhou com *frame sizes* inferiores a 600.

Eficiência do protocolo de ligação de dados

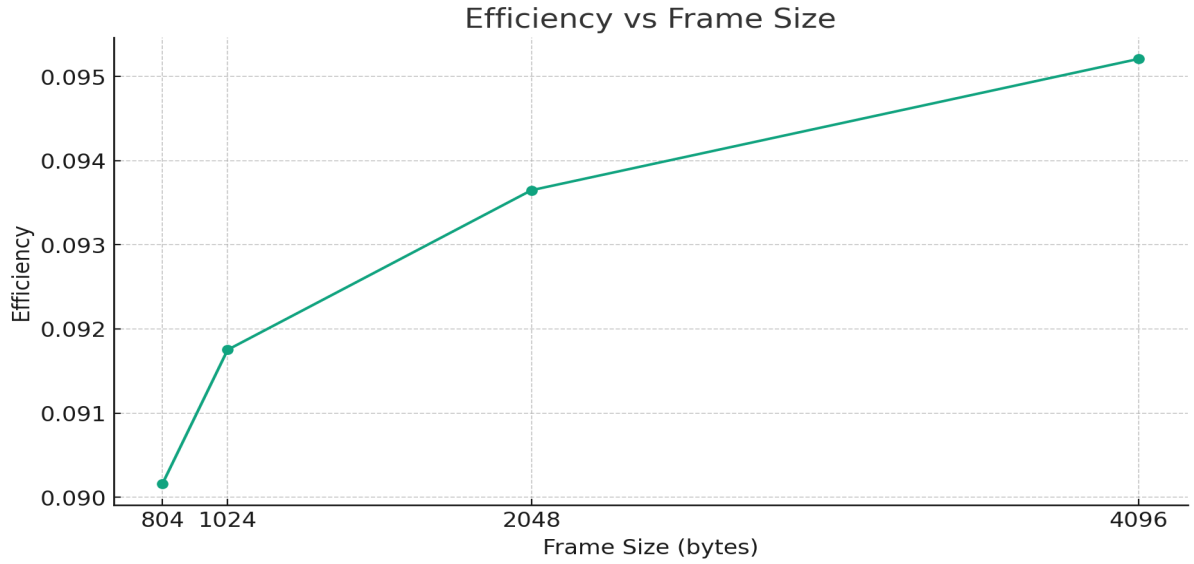
Com o objetivo de medir a eficiência da nossa aplicação fizemos vários testes, cronometrados do lado do recetor. Para cada variação de valores (baud rate, frame size, etc.) produzimos mais do que um teste, com o objetivo de calcularos a média entre os mesmos de forma a melhorar a sua precisão e confiabilidade. Todos estes testes foram efetuados com o ficheiro dado de 10968 bytes.

No caso do *Baud Rate* fizemos os testes com Frame size a 1024, T_Prop a 0 e FER a 0. Analisando os gráficos, rapidamente concluímos que a nossa aplicação perde eficiência com o aumento do Baud Rate, o que seria de esperar analisando a fórmula do cálculo de eficiência.

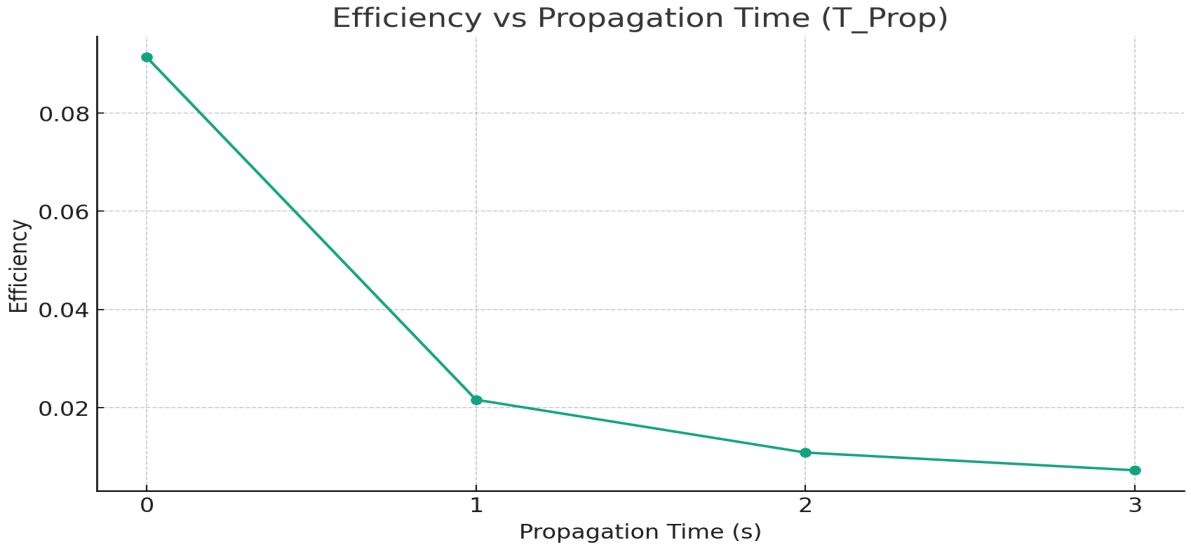
$$S = \frac{T_f}{T_{prop} + T_f + T_{prop}} = \frac{1}{1+2a}$$



Em relação ao *Frame Size* fizemos os testes com Baud Rate a 38400, *T_Prop* a 0 e FER a 0. Com uma observação rápida do gráfico, conseguimos concluir que a nossa aplicação ganha eficiência com o aumento do Frame Size, o que seria de esperar visto que diminui o número de tramas e por sua vez headers necessários para enviar o ficheiro.



Finalmente, no caso do *T_Prop,* fizemos os testes com frame size a 1024, Baud Rate a 38400 e FER a 0. Olhando brevemente para o gráfico, chegamos à conclusão que a nossa aplicação perde eficiência com o aumento do tempo de propagação, o que seria de esperar visto que aumenta drasticamente o tempo para o envio/receção de cada trama.



Conclusões

Este projeto foi fundamental para consolidar o nosso conhecimento em redes de computadores, ligando a teoria e a prática. Através da implementação do protocolo de ligação de dados e da camada de aplicação, reforçamos competências essenciais como o controlo de erros, a confiabilidade na transmissão de dados e a coordenação entre camadas de software.

A codificação, envio e receção de ficheiros destacaram a importância de um design de software estruturado, onde cada função tem um propósito claro, que trabalham em conjunto para assegurar uma boa transferência de dados. O tratamento de erros e falhas promoveu o desenvolvimento do nosso pensamento crítico e habilidades de resolução de problemas.

Analisando agora o nosso trabalho pessoal, acreditamos que, numa nova oportunidade, conseguiríamos construir algo mais eficiente, mas estamos contentes com os resultados alcançados.

Em suma, este projeto permitiu a aplicação de teorias de redes de computadores em situações práticas, reforçando o entendimento de que a teoria subjacente é essencial para o desenvolvimento de soluções eficientes e confiáveis. Foi também uma oportunidade para apreciar a complexidade e o engenho por detrás de alguns dos sistemas de comunicações que usamos diariamente.

Anexo I - Código Fonte

main.c

```
// Main file of the serial port project.
// NOTE: This file must not be changed.

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <termios.h>

#include "application_layer.h"

#define BAUDRATE B38400
#define N_TRIES 3
#define TIMEOUT 4

// 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;
}
```

application_layer.h

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

#ifndef _APPLICATION_LAYER_H_
#define _APPLICATION_LAYER_H_

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

#endif // _APPLICATION_LAYER_H_
```

application_layer.c

```
// Application layer protocol implementation

#include "application_layer.h"
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
#include "link_layer.h"
```

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

#define END_PACKET 0x03
#define START_PACKET 0x02
#define DATA_PACKET 0x01

// Function to send control packet with file information
int sendCPacket(int fd, unsigned char packetType, const char *filename)
{
    FILE *f = fopen(filename, "rb");
    fseek(f, 0L, SEEK_END);
    int sizeF = ftell(f);
    fclose(f);
    unsigned char buf[1024];
    unsigned char b3 = sizeF >> 8;
    unsigned char b4 = (unsigned char)sizeF;

    buf[0] = packetType;
    buf[1] = 0x00;
    buf[2] = 0x02;
    buf[3] = b3;
    buf[4] = b4;
    buf[5] = 0x01;
    buf[6] = strlen(filename);

    memcpy(buf + 7, filename, strlen(filename));
    llwrite(buf,strlen(filename) + 7);

    return 0;
}

// Function to send data packets with file content
int sendDPacket(int fd, const char *filename)
{
    FILE *f = fopen(filename, "rb");
    unsigned char buf[MAX_PAYLOAD_SIZE + 4];
    unsigned int bytesRead = 0;
    int packetNumber = 0;

    while ((bytesRead = fread(buf + 4, 1, MAX_PAYLOAD_SIZE, f)) > 0)
```

```
    {
        buf[0] = DATA_PACKET;
        buf[1] = packetNumber;
        buf[2] = bytesRead / 256;
        buf[3] = bytesRead % 256;

        if (llwrite(buf, bytesRead + 4) == -1)
        {
            printf("Maximum tries reached\n");
            exit(-1);
        }
        packetNumber++;
    }

    fclose(f);
    return 0;
}

// Function to direct packet send request to appropriate function based
on packet type
int sendPacket(int fd, unsigned char packetType, const char *filename)
{
    switch (packetType) {
        case START_PACKET:

        case END_PACKET:
            return sendCPacket(fd, packetType, filename);

        case DATA_PACKET:
            return sendDPacket(fd, filename);

        default:
            return -1;
    }
}

// Function to receive packets and write data to file as per packet
type
int receivePacket(int fd, const char *filename)
{
    FILE *f;
    unsigned int addSize, bytesRead;
    unsigned char buf[(MAX_PAYLOAD_SIZE+4)*2];
    int packetNumber = 0;
```

```
    while (1) {
        bytesRead = llread(buf);

        if (buf[0] == START_PACKET)
        {
            f = fopen(filename, "wb");
        } else if (bytesRead > 0 && buf[0] == DATA_PACKET)
        {
            addSize = buf[2] * 256 + buf[3];
            fwrite(buf + 4, 1, addSize, f);
            if (buf[1] == packetNumber)
            {
                packetNumber++;
            }
        } else if (buf[0] == END_PACKET)
        {
            printf("ENDING\n");
            break;
        }
    }

    fclose(f);
    return fd;
}

void applicationLayer(const char *port, const char *role, int baudRate,
                      int retries, int timeout, const char *filename)
{
    struct applicationLayer applicationLayer;

    LinkLayerRole linkRole;
    linkRole = (strcmp(role, "tx") == 0) ? LlTx : LlRx;

    applicationLayer.status = (strcmp(role, "tx") == 0) ? 1 : 0;

    LinkLayer linkLayer;
    linkLayer.role = linkRole;
    linkLayer.baudRate = baudRate;
    linkLayer.nRetransmissions = retries;
    strcpy(linkLayer.serialPort, port);
    linkLayer.timeout = timeout;

    int fd = llopen(linkLayer);

    if (fd == -1)
```

```
        return;

    applicationLayer.fileDescriptor = fd;

    switch (applicationLayer.status)
    {
        case 1:
            printf("Sending file\n");
            sendPacket(fd, START_PACKET, filename);
            sendPacket(fd, DATA_PACKET, filename);
            sendPacket(fd, END_PACKET, filename);
            break;

        case 0:
            printf("Receiving file\n");
            receivePacket(fd, filename);
            break;

        default:
            printf("Invalid role\n");

    }

    printf("END\n");
    llclose(0);
}
```

link_layer.h

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

#ifndef _LINK_LAYER_H_
#define _LINK_LAYER_H_

typedef enum
{
    LlTx,
    LlRx,
} LinkLayerRole;

typedef struct
{
    char serialPort[50];
    LinkLayerRole role;
```

```
    int baudRate;
    int nRetransmissions;
    int timeout;
} LinkLayer;

// SIZE of maximum acceptable payload.
// Maximum number of bytes that application layer should send to link
layer
#define MAX_PAYLOAD_SIZE 1020

// MISC
#define FALSE 0
#define TRUE 1

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

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

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

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

#endif // _LINK_LAYER_H_
```

link_layer.c

```
// Link layer protocol implementation
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
#include <termios.h>
```

```c
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include "link_layer.h"

// Finite state machine states
typedef enum
{START, FLAG_RCV,A_RCV,C_RCV, BCC_NORMAL, BCC_DATA, DONE} stateMachine;

// Various constants and macros
#define C_RECEIVER 0x07
#define FALSE 0
#define TRUE 1
#define SIZE_SET 5
#define RECEIVER 0
#define BCC(n, m) (n ^ m)
#define A 0x03
#define SIZE_UA 5
#define _POSIX_SOURCE 1 // POSIX compliant source
#define F 0x7e
#define NACK(n) ((n) << 7 | 0x01)
#define ACK(n) ((n) << 7 | 0x05)
#define ESC 0x7D
#define TRANSMITER 1
#define FLAG 0x7e
#define C 0x03
#define REPEATED_MSG_CODE 2

stateMachine state;
int fd;     // file descriptor
int sequenceNum = 0;
int hasFailed = 0;
int alarmCount = 0;
int alarmOn;

volatile int STOP = FALSE;
struct termios oldtio; // old terminal IO settings

int nRetransmissions = 0;
LinkLayer linkLayer;

// Manager for alarm signal
void alarmManager(int signal)
{
    printf("<No answer from receiving end>\n");
```

```
    alarmOn = FALSE;
    alarmCount++;
    hasFailed = 1;
}

// Determine the current state of the state machine
void stateDetermine(stateMachine *state, char byte, int user)
{
    unsigned char FLAG_A = 0x03;                // Address flag
constant
    unsigned char FLAG_C = (user) ? 0x07 : 0x03; // Control flag
constant, varies based on user role (transmitter or receiver)

    // Switch through the different states of the state machine
    switch (*state)
    {
    case START:
        // If FLAG is received, transition to the FLAG_RCV state
        if (byte == FLAG)
            *state = FLAG_RCV;
        break;

    case FLAG_RCV:
        // If another FLAG is received, stay in FLAG_RCV state
        if (byte == FLAG)
            *state = FLAG_RCV;
        // If FLAG_A (address byte) is received, transition to the
A_RCV state
        else if (byte == FLAG_A)
            *state = A_RCV;
        // Any other byte, revert back to the START state
        else
            *state = START;
        break;

    case A_RCV:
        // If FLAG is received, transition to the FLAG_RCV state
        if (byte == FLAG)
            *state = FLAG_RCV;
        // If FLAG_C (control byte) is received, transition to the
C_RCV state
        else if (byte == FLAG_C)
            *state = C_RCV;
        // Any other byte, revert back to the START state
        else
```

```
            *state = START;
        break;

    case C_RCV:
        // If FLAG is received, transition to the FLAG_RCV state
        if (byte == FLAG)
            *state = FLAG_RCV;
        // If BCC (checksum of address and control bytes) is received,
transition to the BCC_NORMAL state
        else if (byte == BCC(FLAG_A, FLAG_C))
            *state = BCC_NORMAL;
        // Any other byte, revert back to the START state
        else
            *state = START;
        break;

    case BCC_NORMAL:
        // If FLAG is received, marking the end of the frame,
transition to the DONE state
        if (byte == FLAG)
            *state = DONE;
        // Any other byte, revert back to the START state
        else
            *state = START;
        break;

    default:
        break;
    }
}

// LLOPEN

// Opens the logical link layer communication.
int llopen(LinkLayer connectionParameters)
{
    // Setup for signal handling
    struct sigaction action;
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;
    action.sa_handler = alarmManager;
    sigaction(SIGALRM, &action, NULL);
    struct termios newtio;
    linkLayer = connectionParameters;
```

```
    // Open the serial port with read/write access
    fd = open(connectionParameters.serialPort, O_RDWR | O_NOCTTY);

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

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

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

    // Set parameters for the serial port connection
    newtio.c_iflag = IGNPAR;
    newtio.c_cflag = connectionParameters.baudRate | CS8 | CLOCAL |
CREAD;
    newtio.c_lflag = 0;
    newtio.c_oflag = 0;
    newtio.c_cc[VTIME] = 1; // Inter-character timer is not used
    newtio.c_cc[VMIN] = 0;  // Read blocks until a character arrives
    tcflush(fd, TCIOFLUSH);

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

    printf("Set new TermIOs struct\n");

    // If operating in transmitter mode
    if (connectionParameters.role == LlTx)
    {
        // Prepare the SET message for transmission
        unsigned char buf[256] = {0};
        buf[0] = FLAG;
        buf[1] = A;
```

```
        buf[2] = C;
        buf[3] = BCC(buf[1], buf[2]);
        buf[4] = F;
        hasFailed = 0;
        int bytesNum = 0;

        // Attempt to send the SET message and wait for UA response
        do
        {
            STOP = FALSE;
            bytesNum = write(fd, buf, SIZE_SET);
            printf("Sent SET: ");
            for (int i = 0; i < bytesNum; i++)
            {
                printf("%02X ", buf[i]); // Print each element of
message as a hexadecimal value
            }
            printf("\n");
            alarm(connectionParameters.timeout);
            alarmOn = TRUE;
            printf("Attempt n°%d\n", alarmCount);
            state = START;
            hasFailed = 0;
            unsigned char aux = 0;

            while (STOP == FALSE)
            {
                bytesNum = read(fd, &aux, 1);
                if (bytesNum > 0)
                {
                    stateDetermine(&state, aux, 1);
                }
                if (state == DONE || hasFailed == 1)
                {
                    alarm(0);
                    STOP = TRUE;
                }
            }
        } while (alarmCount < connectionParameters.nRetransmissions &&
state != DONE);

        if (state == DONE)
        {
            printf("Received UA\n");
        }
```

```
        else
        {
            printf("Didn´t receive UA\n");
            return -1;
        }
    }
    else
    {
        // If operating in receiver mode
        unsigned char aux = 0;
        state = START;

        // Wait for a SET message
        while (STOP == FALSE)
        {
            read(fd, &aux, 1);
            stateDetermine(&state, aux, 0);

            if (state == DONE)
                STOP = TRUE;
        }

        printf("Received SET\n");

        // Prepare and send a UA message in response
        unsigned char message[256] = {0};
        message[0] = FLAG;
        message[1] = 0x03;
        message[2] = 0x07;
        message[3] = BCC(0x03, 0x07);
        message[4] = FLAG;

        int bytesNum = write(fd, message, SIZE_UA);
        printf("Sent UA: ");
        for (int i = 0; i < bytesNum; i++)
        {
            printf("%02X ", message[i]); // Print each element of
message as a hexadecimal value
        }
        printf("\n");
    }

    return fd;
}
```

```
////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////

// Process received ACK messages
void receiveACK(stateMachine *state, unsigned char byte, unsigned char
*ack, int sequenceNum)
{
    switch (*state)
    {
    case START:
        // If the byte is a FLAG, transition to the FLAG_RCV state.
        if (byte == FLAG)
        {
            *state = FLAG_RCV;
        }
        break;

    case FLAG_RCV:
        // On receiving a FLAG, remain in the FLAG_RCV state.
        if (byte == FLAG)
        {
            *state = FLAG_RCV;
        }
        // On receiving A, transition to the A_RCV state.
        else if (byte == A)
        {
            *state = A_RCV;
        }
        // Any other byte, revert to the START state.
        else
        {
            *state = START;
        }
        break;

    case A_RCV:
        // If a FLAG is received, return to the FLAG_RCV state.
        if (byte == FLAG)
        {
            *state = FLAG_RCV;
        }
        // If the byte matches the expected ACK, transition to the
C_RCV state.
        else if (byte == ACK(sequenceNum))
```

```
        {
            *state = C_RCV;
            *ack = ACK(sequenceNum);
        }
        // If the byte matches the expected NACK, also transition to
the C_RCV state.
        else if (byte == NACK(sequenceNum))
        {
            *state = C_RCV;
            *ack = NACK(sequenceNum);
        }
        // Any other byte, revert to the START state.
        else
        {
            *state = START;
        }
        break;

    case C_RCV:
        // If the byte matches the expected BCC value, transition to
the BCC_NORMAL state.
        if (byte == BCC(A, *ack))
        {
            *state = BCC_NORMAL;
        }
        // If a FLAG is received, reset the acknowledgment and return
to the FLAG_RCV state.
        else if (byte == FLAG_RCV)
        {
            *ack = FALSE;
            *state = FLAG_RCV;
        }
        // For any other byte, reset acknowledgment and revert to the
START state.
        else
        {
            *state = START;
            *ack = FALSE;
        }
        break;

    case BCC_NORMAL:
        // If a FLAG is received after BCC, transition to the DONE
state, indicating a complete ACK/NACK frame.
        if (byte == FLAG)
```

```
        {
            *state = DONE;
        }
        // Any other byte, reset the acknowledgment and revert to the
START state.
        else
        {
            *ack = FALSE;
            *state = START;
        }
        break;

    default:
        break;
    }
}

// Sends a data buffer over the link layer, handles byte stuffing, and
acknowledges receipt.
int llwrite(const unsigned char *buf, int bufSize)
{
    signal(SIGALRM, alarmManager); // Register the alarm signal manager
    state = START;
    int attemptNum = 0;         // Counter for retry attempts

    // Count bytes that need stuffing
    unsigned int counter = 0;
    for (int i = 0; i < bufSize; i++)
    {
        if (buf[i] == FLAG || buf[i] == ESC)
        {
            counter++;
        }
    }

    unsigned int size = bufSize + 6 + counter; // Calculate total size
after stuffing
    unsigned char message[size];
    memset(message, 0, size); // Initialize the message buffer

    // Frame header
    message[0] = FLAG;
    message[1] = A;
    message[2] = sequenceNum << 7;
    message[3] = BCC(A, sequenceNum << 7);
```

```
    unsigned int BCC2 = 0, i = 0;

    // Byte stuffing for the message
    for (int j = 0; j < bufSize; j++)
    {
        switch (buf[j])
        {
        case ESC:
            message[i + j + 4] = ESC;
            message[i + j + 5] = ESC ^0x20;
            BCC2 = BCC(BCC2, ESC);
            i++;
            break;

        case FLAG:
            message[i + j + 4] = ESC;
            message[i + j + 5] = FLAG ^0x20;
            BCC2 = BCC(BCC2, FLAG);
            i++;
            break;

        default:
            message[i + j + 4] = buf[j];
            BCC2 = BCC(BCC2, buf[j]);
        }
    }

    message[i + bufSize + 4] = BCC2; // Frame footer with BCC and FLAG
    message[i + bufSize + 5] = FLAG;

    STOP = FALSE;
    alarmOn = FALSE;

    // Loop until the frame is acknowledged or the maximum number of
attempts is reached
    while (STOP != TRUE && state != DONE)
    {
        unsigned char ack;      // ACK byte
        unsigned char receivedByte; // Byte read from the link

        // Send the frame if the alarm is not already set
        if (alarmOn == FALSE)
        {
            if (attemptNum > linkLayer.nRetransmissions)
```

```
            {
                return -1; // Max attempts reached
            }
            attemptNum++;

            write(fd, message, size);           // Send the message
            signal(SIGALRM, alarmManager); // Set the alarm signal
manager again
            alarm(3);                       // Set an alarm for 3
seconds
            alarmOn = TRUE;
        }

        // Read a byte from the link
        unsigned int bytesNum = read(fd, &receivedByte, 1);

        if (bytesNum > 0)
        {
            receiveACK(&state, receivedByte, &ack, 1 - sequenceNum);

            // Check if the acknowledgment is as expected
            if (state == DONE && ack == ACK(1 - sequenceNum))
            {
                sequenceNum = 1 - sequenceNum;
                alarm(0);     // Cancel the alarm
                STOP = TRUE; // Stop the loop
                printf("RECEIVED ACK aka RR...\n");
            }
            // Resend the frame if NACK received
            else if (state == DONE)
            {
                printf("RECEIVED NACK aka RREJ...\n");
                write(fd, message, size);
                state = START;
            }
        }
    }
    return 0; // Return success
}

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

```
// Function to process received data and handle byte stuffing return
true when it must return ack, and false for nack
int receiveData(unsigned char *packet, int sequenceNum, size_t
*size_read)
{
    state = START;
    unsigned char REPLY_C = (1 - sequenceNum) << 7;
    unsigned char CONTROL_C = sequenceNum << 7;          // Construct
control byte using sequence number
    unsigned int BCC2 = 0, i = 0, stuffing = 0; // Stuffing flag: set
to 1 every time an ESC is encountered

    // Keep processing bytes until the end flag is received
    while (state != DONE)
    {
        unsigned char receivedByte;
        unsigned int bytesNum = read(fd, &receivedByte, 1);

        if (bytesNum > 0)
        {

            switch (state)
            {
            case START:
                if (receivedByte == FLAG)
                {
                    state = FLAG_RCV;
                }
                break;

            case A_RCV:
                if (receivedByte == FLAG)
                {
                    state = FLAG_RCV;
                }
                else if (receivedByte == CONTROL_C)
                {
                    state = C_RCV;
                }
                // Handling case of receiving repeated message and
waiting for the next one
                else if (REPLY_C == receivedByte)
                {
                    return REPEATED_MSG_CODE; // Return code indicating
a repeated message was received
```

```
            }
            else
            {
                state = START;
            }
            break;

        case FLAG_RCV:
            if (receivedByte == FLAG)
            {
                state = FLAG_RCV;
            }
            else if (receivedByte == A)
            {
                state = A_RCV;
            }
            else
            {
                state = START;
            }
            break;

        case C_RCV:
            if (receivedByte == FLAG)
            {
                state = FLAG_RCV;
            }
            else if (receivedByte == BCC(A, CONTROL_C))
            {
                state = BCC_DATA;
            }
            else
            {
                state = START;
            }
            break;

        // Handle byte stuffing and frame de-stuffing
        case BCC_DATA:
            if (!stuffing)
            {
                if (receivedByte == FLAG)
                {
                    state = DONE;
                    BCC2 = BCC(BCC2, packet[i - 1]);
```

```
                        *size_read = i - 1;             // Update the
size of the read data
                        return (BCC2 == packet[i - 1]); // Return TRUE
if BCC matches, otherwise FALSE
                    }
                    else if (receivedByte == ESC)
                    {
                        stuffing = TRUE; // Set the stuffing flag if
ESC is encountered
                    }
                    else
                    {
                        BCC2 = BCC(BCC2, receivedByte);
                        packet[i] = receivedByte;
                        i++;
                    }
                }
                else
                {
                    stuffing = FALSE; // Reset the stuffing flag
                    BCC2 = BCC(BCC2, receivedByte ^ 0x20);
                    packet[i] = receivedByte ^ 0x20;
                    i++;
                }
                break;

            default:
                break;
            }
        }
    }

    return FALSE;
}


// Reads data from the link layer and acknowledges the received data.
int llread(unsigned char *packet)
{
    int readStatus;
    size_t size_read;


    // Continuously try to receive data until successful
```

```
    while ((readStatus = receiveData(packet, sequenceNum, &size_read))
!= TRUE)
    {
        // If the reply indicates an error (e.g., checksum mismatch)
        if (readStatus == 0)
        {
            printf("Sending RRej or NACK\n");

            // Send a NACK (negative acknowledgment)
            unsigned char NACK_C = NACK(1 - sequenceNum);
            unsigned char buf[] = {FLAG, A, NACK_C, BCC(A, NACK_C), F};
            write(fd, buf, 5);
        }
        // If the received message is a duplicate (e.g.,
retransmission)
        else
        {
            printf("Repeated message, sending ACK\n");

            // Send an ACK to prevent further retransmissions
            unsigned char ACK_C = ACK(sequenceNum);
            unsigned char buf[] = {FLAG, A, ACK_C, BCC(A, ACK_C), F};
            write(fd, buf, 5);
        }
    }

    // Once data is received correctly, send an ACK
    printf("Everything in order, sending ACK\n");
    sequenceNum = 1 - sequenceNum; // Toggle the sequence number
    unsigned char ACK_C = ACK(sequenceNum);
    unsigned char buf[] = {FLAG, A, ACK_C, BCC(A, ACK_C), F};
    write(fd, buf, 5);

    return size_read;
}


////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////

// Determine the next state of the state machine based on the received
byte.
void DISCStateDetermine(stateMachine *state, char byte)
{
```

```
// Define the flags for the state transitions
unsigned char FLAG_C = 0x0B;
unsigned char FLAG_A = 0x03;

// Determine the current state and process the incoming byte
switch (*state)
{
// Waiting for the initial FLAG byte
case START:
    if (byte == FLAG)
    {
        *state = FLAG_RCV;
    }
    break;

// A FLAG byte has been received and waiting for the FLAG_A
case FLAG_RCV:
    if (byte == FLAG)
    {
        *state = FLAG_RCV;
    }
    else if (byte == FLAG_A)
    {
        *state = A_RCV;
    }
    else
    {
        *state = START;
    }
    break;

// FLAG_C has been received, waiting for BCC of FLAG_A and FLAG_C
case C_RCV:
    if (byte == FLAG)
    {
        *state = FLAG_RCV;
    }
    else if (byte == BCC(FLAG_A, FLAG_C))
    {
        *state = BCC_NORMAL;
    }
    else
    {
        *state = START;
    }
```

```
        break;

    // FLAG_A has been received, waiting for FLAG_C
    case A_RCV:
        if (byte == FLAG)
        {
            *state = FLAG_RCV;
        }
        else if (byte == FLAG_C)
        {
            *state = C_RCV;
        }
        else
        {
            *state = START;
        }
        break;

    // BCC for FLAG_A and FLAG_C received, waiting for final FLAG
    case BCC_NORMAL:
        if (byte == FLAG)
        {
            *state = DONE;
        }
        else
        {
            *state = START;
        }
        break;

    default:
        break;
    }
}

// Closes the logical link layer communication.
int llclose(int statistics)
{
    int bytesNum = 0;
    unsigned char aux = 0;
    unsigned char message[256] = {0};

    switch (linkLayer.role)
    {
    case LlTx:
```

```
        // Send DISC (Disconnect Frame)
        message[0] = FLAG;
        message[1] = 0x03;
        message[2] = 0x0B;
        message[3] = BCC(0x03, 0x0B);
        message[4] = FLAG;
        bytesNum = write(fd, message, 5);
        printf("Sent DISC: ");
        for (int i = 0; i < bytesNum; i++)
        {
            printf("%02X ", message[i]); // Print each element of
message as a hexadecimal value
        }
        printf("\n");
        state = START;
        STOP = FALSE;

        // Wait for DISC acknowledgment
        while (STOP == FALSE)
        {
            read(fd, &aux, 1);
            DISCStateDetermine(&state, aux);
            if (state == DONE)
                STOP = TRUE;
        }

        // Send UA (Unnumbered Acknowledgment Frame)
        unsigned char ua[256] = {0};
        ua[0] = FLAG;
        ua[1] = 0x03;
        ua[2] = 0x07;
        ua[3] = BCC(0x03, 0x07);
        ua[4] = FLAG;
        bytesNum = write(fd, ua, SIZE_UA);
        printf("UA sent: ");
        for (int i = 0; i < bytesNum; i++)
        {
            printf("%02X ", ua[i]); // Print each element of message as
a hexadecimal value
        }
        printf("\n");
        sleep(1);
        break;

    case LlRx:
```

```
        STOP = FALSE;
        state = START;

        // Wait for DISC from transmitter
        while (STOP == FALSE)
        {
            read(fd, &aux, 1);
            DISCStateDetermine(&state, aux);

            if (state == DONE)
                STOP = TRUE;
        }
        printf("Received DISC\n");

        // Send DISC as acknowledgment
        message[0] = FLAG;
        message[1] = 0x03;
        message[2] = 0x0B;
        message[3] = BCC(0x03, 0x0B);
        message[4] = FLAG;
        bytesNum = write(fd, message, 5);

        printf("Sent DISC to acknowledge: ");

        for (int i = 0; i < bytesNum; i++)
        {
            printf("%02X ", message[i]); // Print each element of
message as a hexadecimal value
        }
        printf("\n");

        // Wait for UA acknowledgment from transmitter
        state = START;
        STOP = FALSE;
        while (STOP == FALSE)
        {
            read(fd, &aux, 1);
            stateDetermine(&state, aux, 1);

            if (state == DONE)
                STOP = TRUE;
        }
        printf("Received UA\n");
        break;
    }
```

```
    // Restore old terminal settings
    if (tcsetattr(fd, TCSANOW, &oldtio) != 0)
    {
        perror("llclose() - Error on tcsetattr()");
        return -1;
    }

    // Close the file descriptor
    if (close(fd) != 0)
    {
        perror("llclose() - Error on close()");
        return -1;
    }
    return 1;
}
```

Makefile

```
# Makefile to build the project
# NOTE: This file must not be changed.

# Parameters
CC = gcc
CFLAGS = -Wall

SRC = src/
INCLUDE = include/
BIN = bin/
CABLE_DIR = cable/

TX_SERIAL_PORT = /dev/ttyS10
RX_SERIAL_PORT = /dev/ttyS11

TX_FILE = penguin.gif
RX_FILE = penguin-received.gif

# Create the bin directory if it doesn't exist
$(shell mkdir -p $(BIN))

# Targets
.PHONY: all
all: $(BIN)/main $(BIN)/cable
```

```
$(BIN)/main: main.c $(SRC)/*.c
    $(CC) $(CFLAGS) -o $@ $^ -I$(INCLUDE)

$(BIN)/cable: $(CABLE_DIR)/cable.c
    $(CC) $(CFLAGS) -o $@ $^

.PHONY: run_tx
run_tx: $(BIN)/main
    ./$(BIN)/main $(TX_SERIAL_PORT) tx $(TX_FILE)

.PHONY: run_rx
run_rx: $(BIN)/main
    ./$(BIN)/main $(RX_SERIAL_PORT) rx $(RX_FILE)

.PHONY: run_cable
run_cable: $(BIN)/cable
    ./$(BIN)/cable

.PHONY: check_files
check_files:
    diff -s $(TX_FILE) $(RX_FILE) || exit 0

.PHONY: clean
clean:
    rm -f $(BIN)/main
    rm -f $(BIN)/cable
    rm -f $(RX_FILE)
```