Redes de Computadores

1º Trabalho Laboratorial - Protocolo de Ligação de Dados

FEUP

Licenciatura em Engenharia Informática e Computação

Joana Rita Batista Marques - up202103346
Joao Tomas Matos Fernandes Garcia Padrao – 202108766

Porto, 5 de Novembro de 2023

Sumário

No âmbito da unidade curricular de Redes de Computadores, foi desenvolvido um trabalho laboratorial com o objectivo de criar um protocolo de ligação de dados para transmissão e recepção de um ficheiro, de acordo com as especificações fornecidas. Esta transmissão é realizada entre dois computadores ligados por um cabo de série.

O presente relatório visa fazer uma exposição e análise da implementação desenvolvida ao longo do trabalho, assim como apresentar as principais conclusões obtidas ao longo do mesmo.

Introdução

Como mencionado anteriomente, o objetivo principal deste primeiro trabalho laboratorial é o desenvolvimento de um protocolo do nível de ligação de dados e uma aplicação de transferência de ficheiros. Em seguida, ao longo das várias secções deste relatório iremos descrever a implementação do trabalho e lógica subjacente ao mesmo. De uma forma breve, a estrutura dos tópicos presentes neste relatório é a seguinte:

- Arquitetura - Blocos funcionais e interfaces aplicados.
- Estrutura do código – Descrição das APIs, principais estruturas de dados, principais funções e a sua relação com a arquitetura.
- Casos de uso principais – Identificação dos casos de uso principais e sequências de chamada de funções.
- Protocolo de ligação lógica - Identificação dos principais aspetos funcionais, e ainda, descrição da estratégia de implementação destes aspetos com apresentação de extratos de código.
- Protocolo de aplicação - Identificação dos principais aspetos funcionais e descrição da estratégia de implementação destes aspetos com apresentação de extratos de código.
- Validação - Descrição dos testes efetuados com apresentação dos resultados.
- Conclusões - Síntese da informação apresentada nas secções anteriores e reflexão sobre os objetivos de aprendizagem alcançados.

Arquitetura

A arquitectura do nosso trabalho está assente em dois blocos funcionias, são eles a camada de aplicação (Application Layer) e a camada de ligação de dados (Link Layer).

Na Link Layer estão as funções principais no que diz respeito á implementação do protocolo de ligação de dados, sendo que determina o estabelecimento e terminação da ligação entre as entidades Receptor e Emissor, escrita e leitura de tramas de informação e supervisão, e ainda, o controlo e detecção de erros nos dados recebidos.

A Application Layer utiliza o serviço fornecido pelo protocolo de ligação de dados, através da invocação das funções do Link Layer, para transferência e recepção das tramas do ficheiro.

Estrutura do código

O código desenvolvido foi dividido entre dois ficheiros que representam os blocos funcionais mencionados anteriormente. Assim, a parte de código relativa á camada de aplicação encontra-se no ficheiro application_layer.c e o código relativo á camada de ligação de dados está disponivel no ficheiro intitulado link_layer.c.

No link_layer.c foram desenvolvidas as funções ncessárias para efectuar a transmissão de dados entre o Emissor e o Receptor. Estas funções fazem uso de Máquinas de Estado que tem como propósito validar, ou não, o correcto envio e recepção das diferentes tramas. Adicionalmente, é também verificiado se o tempo máximo de resposta foi excedido através de funções de Alarme.

O application_layer.c inclui funções para envio e recepção de tramas de controlo e pacotes de dados. É neste ficheiro que são criadas as tramas de informação e as tramas de controlo. Dependendo de qual a entidade que está a utilizar o application_layer.c, isto é, se o Emissor ou o Receptor, diferentes funções do link_layer.c serão invocadas pela aplicação.

Casos de uso principais

Após a compilação do programa através do usual comando “make”, para executar o programa é necessário determinar se irá o utilizador irá correr o programa em modo Emissor ou Receptor, para que sejam efectuadas as devidas configurações a nível da aplicação.

Primeiramente deve-se executar o programa em modo receptor, que ficará a aguardar que o emissor dê inicio á ligação. Em seguida, executa-se o programa em modo de emissor. Uma vez estabelecida a ligação, o emissor dá início ao envio dos dados do ficheiro e o receptor procede a recebe-los byte a byte. Dependendo de se o envio e recepção dos dados ocorre com ou sem sucesso, diferentes mensagens surgem no terminal, de forma a indicar o que está a acontecer ao longo das diferentes etapas de envio e recepção de tramas de informação e controle entre o emissor e o receptor.

Se o programa estiver a correr em modo Transmissor (Tx), é criado um pacote de dados ou de controlo, e é invocada a função llwrite do link_layer.c. Caso seja em modo Emissor (Rx), é invocada a função llread.

Protocolo de ligação lógica

No protocolo de ligação lógica (link_layer.c) são efectuadas as seguintes acções:

- Configuração da porta série;
- Estabelecimento da ligação entre os dois computadores pela porta série RS-232;
- Escrita e Leitura das tramas de informação e de controlo;
- Verificação e Tratamento de Erros nos dados recebidos;

Em seguida apresentamos as funções desenvolvidas que permitem efectuar as acções descritas anteriormente.

- **Llopen**

Esta função é usada para estabelecer a ligação á porta série através da implementação de máquinas de estado que dependem de se é invocada pelo Emissor ou Receptor. Recebe um LinkLayer struct como input, que contêm os parâmetros de configuração necessários para estabelecer a ligação. A função inicializa variáveis e configura a porta serie com base nos parâmetros fornecidos e define as configurações do terminal usando as funções tcgetattr e tcsetattr.

```
// llopen function
int llopen(LinkLayer connectionParameters){

    unsigned char read_buffer[5] = {0};
    maxNRetransmissions = connectionParameters.nRetransmissions;
    timeout = connectionParameters.timeout;
    states state = START;
    role = connectionParameters.role;
    fd = open(connectionParameters.serialPort, O_RDWR | O_NOCTTY);

    if (fd < 0)···

    // Save current port settings
    if (tcgetattr(fd, &oldtio) == -1)···

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

    newtio.c_cflag = connectionParameters.baudRate | CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;
    newtio.c_lflag = 0;
    newtio.c_cc[VTIME] = 0;
    newtio.c_cc[VMIN] = 0;

    tcflush(fd, TCIOFLUSH);

    // Set new port settings
    if (tcsetattr(fd, TCSANOW, &newtio) == -1)···

    //State Machine
    switch (connectionParameters.role){
```

Em seguida, inicia a máquina de estados de acordo com um parâmetro que determina se se trata do Emissor ou do Receptor. Caso seja o Emissor (LlTx), é enviada uma trama SET ao receptor e aguarda por uma trama UA como resposta. Verifica a trama byte a byte e verifica se os valores recebidos são os que eram expectaveis. Se não for recebida uma resposta dentro do tempo estipulado, volta a tentar enviar a trama. Caso seja o Receptor (LlRx), espera para receber uma trama SET do Emissor, valida os bytes recebidos usando a máquina de estados e envia uma trama UA de volta para confirmar a ligação bem sucedida.

Se os parâmetros de ligação estiverem incorretos ou ocorrer um erro durante a comunicação, a função retorna -1. Caso contrário, retorna o descritor de ficheiro da porta serie.

- **Llwrite**

A função llwrite é usada unicamente pelo emissor e serve para escrever tramas de dados na porta serie. Calcula o tamanho da trama com base no tamanho do buffer de entrada (buf), aloca memória para a trama e constrói a trama definindo os bytes apropriados. Calcula o BCC2 e faz o byte stuffing necessário para certos bytes (FLAG e ESC) para evitar conflitos com o formato da trama. Também acrescenta o cabeçalho da trama, que inclui o flag, o campo de endereço, o campo de controlo e o BCC1. Ao enviar a trama, a função llwrite ativa um temporizador e aguarda por uma resposta do receptor. A resposta pode ser do tipo RR (reconhecimento positivo) ou REJ (reconhecimento negativo). Se não receber resposta dentro do intervalo de tempo especificado, a trama é reenviada. Este processo repete-se até o número máximo de retransmissões ser atingido.

Se a resposta for do tipo REJ, a trama é reenviada sem alterações. Se a resposta for do tipo RR, a trama é considerada aceita e o processo de transmissão é concluído.

```
// Continue creating the frame
frame[frameIndex++] = BCC2;
frame[frameIndex] = FLAG; // End of frame
int transmissionsCounter = 0;
int accepted = FALSE;
alarmEnabled = FALSE;

while((transmissionsCounter <= maxNRetransmissions)){
    printf("Transmissions counter: %d\n", transmissionsCounter);
    if(alarmEnabled == FALSE){
        // Send frame
        printf("Sending frame\n");
        write(fd,frame, frameSize);
        alarm(timeout);
        sleep(1); // Sleep for 1 second to avoid sending the next frame before the receiver sends the RR
        alarmEnabled = TRUE;
        transmissionsCounter++;
    }
    // Get response
    unsigned char response = getResponse();
    if(response == CTRL_RR0 || response == CTRL_RR1){
        printf("Receive RR\n");
        accepted = TRUE;
        alarm(0);
        alarmEnabled = FALSE;
        frame_tx = (frame_tx+1) % 2; // Increment frame_tx to change the control field of the next frame
    }

    if(accepted) break;
}
if(accepted){
    printf("It was accepted (Received RR)\n");
    return frameSize; // Return the number of bytes written
}
else{
    //llclose(0);
    return -1;
```

Excerto da função llwrite.

- **Llread**

A função llread lê e processa pacotes de dados byte a byte, e através da implementação de máquina de estados, verifica a integridade dos dados. A função verifica o cabeçalho e o campo de dados, e envia uma resposta REJ caso sejam inválidos. Verifica também se se trata de uma trama repetida, e se for, o campo de dados é descartado e é feita confirmação da trama com envio de resposta RR. Se a trama recebida não for repetida, a função transita de estado e armazena o campo de controlo. Ao processar o campo de dados recebido, a função realiza destuffing caso encontre um ESC byte e verifica o BCC2, enviando uma resposta RR se este for válido ou REJ caso contrário.

- **Llclose**

A função llclose é responsável por terminar a ligação da porta série e utiliza também uma máqunina de estados. Para terminar a ligação, caso seja o emissor, é enviada uma trama DISC ao receptor. Em seguida, é ativado um alarme para aguardar a resposta do receptor, que deve ser igualmente uma trama DISC.

Se a resposta for recebida corretamente, o emissor envia uma trama UA ao receptor para confirmar o encerramento da ligação. Em seguida, o emissor repõe as configurações anteriores da porta serie e termina o programa.

O receptor, por sua vez, espera até receber uma trama DISC. Quando a recebe, responde com uma trama do mesmo tipo. Em seguida, o receptor aguarda pela trama UA do emissor e caso esta seja recebida corretamente, o programa é terminado.

Protocolo de aplicação

No protocolo de aplicação (application_layer.c) são efetuadas as seguintes ações:

- A transferência de pacotes de dados e de controlo.
- O envio e a leitura do ficheiro a transmitir.

Em seguida apresentamos as funções desenvolvidas que permitem efectuar as acções descritas anteriormente:

- **sendControlPacket e sendDataPacket**

Estas funções são as responsáveis pela criação dos pacotes de controlo e de dados. A função “sendControlPacket” cria um pacote de controlo, que contém a informação codificada, como referido no guião, em TLVs (Type, Length, Value), isto é, para cada parâmetro a passar nesse pacote, é necessário passar o tipo do parâmetro (tamanho ou nome do ficheiro) (T), o seu tamanho (L) e finalmente o valor do parâmetro (V). De seguida, após a criação do pacote de controlo, este vai ser enviado.

```c
int sendControlPacket(const char* filename,const int control_packet,long int fileLength){

    // Get file length in bytes
    size_t fileLengthL1 = 0;
    long int tempLength = fileLength;
    while (tempLength > 0) {
      fileLengthL1++;
      tempLength >>= 8;
    }
    long int fileNameLengthL2 = strlen(filename)+1;

    // Create control packet
    long int packet_size = 5 + fileLengthL1 + fileNameLengthL2;
    unsigned char* controlPacket = (unsigned char*)malloc(packet_size);

    // Fill control packet
    int idx = 0;
    controlPacket[idx++] = control_packet;

    //File size
    controlPacket[idx++] = 0x00; // T1 (file size)
    memcpy(controlPacket+idx, &fileLengthL1, sizeof(size_t));
    idx += sizeof(size_t);

    //File name
    controlPacket[idx++] = 0x01; // T2 (file name)
    memcpy(controlPacket + idx, filename, fileNameLengthL2);

    // Send control packet
    if(llwrite(controlPacket,packet_size) < 0){
        printf("Error sending control packet in llwrite().\n");
        free(controlPacket);
        return -1;
    }
    free(controlPacket);
```

A função “sendDataPacket” por sua vez procede à construção do pacote de dados, para isso cria o pacote com o campo de controlo “DATA_PACKET” (0x01 para enviar dados), número de octetos (K) do campo de dados sabendo que “K = 256 * L2 + L1” e finalmente os dados recebidos como argumento. Após criado o pacote de dados este é enviado.

```c
int sendDataPacket(int dataSize,unsigned char* data){

    // Create data packet
    long int packet_size = 1 + 1 + 1 + dataSize;
    unsigned char* dataPacket = (unsigned char*)malloc(packet_size);

    // Fill data packet
    int idx = 0;
    dataPacket[idx++] = DATA_PACKET; // T1 (data)
    // K = 256 * L2 + L1
    dataPacket[idx++] = dataSize / 256; // L2 (data)
    dataPacket[idx++] = dataSize % 256; // L1 (data)
    for (int i = 0; i < dataSize; i++) {
        dataPacket[idx++] = data[i]; // P (data)
    }

    // Send data packet
    if(llwrite(dataPacket, packet_size) < 0){
        printf("Error sending data packet in llwrite().\n");
        return -1;
    }
    return 0;
}
```

- **receiveFile e sendFile**

Estas são as funções responsáveis por todo o processo de leitura e escrita do ficheiro. A aplicação primeiramente vai invocar a função “llopen” para estabelecer a conexão. Caso seja um emissor, a aplicação vai invocar a função “sendFile”. Esta por sua vez abre o ficheiro, envia o pacote de controlo “start” chamando a função “sendControlPacket” e seguidamente vai ler e enviar em pacotes de dados, com ajuda da função “sendDataPacket”, todo o ficheiro. Após terminado (quando tiver lido e enviado o ficheiro todo), a função vai enviar um pacote de controlo “end” que sinaliza o final da transmissão e seguidamente invoca a função “llclose” para terminar a ligação. Caso seja um recetor, a aplicação vai invocar a função “receiveFile” que por sua vez invoca a função do link_layer “llread” lendo assim os pacotes enviados pelo emissor e fazendo a interpretação adequada dos mesmos. Se o pacote recebido for um pacote de controlo inicial "start", a função "readControlPacket" será invocada para realizar a leitura apropriada desse pacote. Caso seja um pacote de dados, o conteúdo do pacote será registado num ficheiro. Por fim, se o pacote recebido for um pacote de controlo final "end", o documento será fechado e chama a função "llclose" para terminar a ligação.

```c
int sendFile(const char* filename){

    // Open file
    FILE* file = fopen(filename, "rb");
    if (file == NULL) {
        printf("Error opening file.\n");
        exit(-1);
    }

    // Get file size (Mudar)
    int previousFile = ftell(file); // save previous position
    fseek(file, 0L, SEEK_END); // file pointer at end of file
    long int fileSize = ftell(file) - previousFile; // get file length
    fseek(file, previousFile, SEEK_SET); // go back to previous position

    // Send start control packet
    if(sendControlPacket(filename, START_PACKET, fileSize) < 0){
        printf("Error sending start control packet.\n");
        return -1;
    }

    // Send file
    unsigned char* data = (unsigned char*)malloc(MAX_PAYLOAD_SIZE-3); // -3 because of the header (C, L2, L1)
    int chunkDataSize;
    while((chunkDataSize = fread(data, 1, MAX_PAYLOAD_SIZE-3, file)) > 0){
        if(sendDataPacket(chunkDataSize,data) < 0){
            printf("Error sending data packet.\n");
            return -1;
        }
    }
    free(data);
    fclose(file);

    // Send end control packet
    if(sendControlPacket(filename, END_PACKET, fileSize) < 0){
        printf("Error sending end control packet.\n");
        return -1;
```

Excerto da função sendFile.

Validação

Para testar a nossa aplicação, foram efectuados os seguintes testes:

● Envio de um ficheiro de um computador para o outro (penguin.gif);

● Interrupção da ligação a meio do procedimento de envio;

● Interrupção da ligação seguida de nova conexão enquanto o ficheiro era enviado;

● Envio de tramas duplicadas;

● Geração de ruído em simultâneo do processo de envio do ficheiro;

Todos os testes mencionados foram efectuados com sucesso, obtendo os resultados esperados, com excepção do último. Na realização do teste de ruído, embora fosse detectado erro no BCC2, seguido de nova tentativa de envio, no final o ficheiro embora tivesse sido efectivamente enviado como esperado, o tamanho era ligeiramente superior ao original, situação para a qual não conseguimos encontrar solução ou origem do erro.

Conclusões

Após a realização deste projeto, entendemos que o mesmo teve um impacto positivo na compreensão e aprofundamento das nossas competências ao nível da implementação de protocolos de ligação de dados tanto a nível teórico como prático.

No entanto, sendo este um tema novo para nós e devido à sobrecarga de projetos propostos ao longo deste semestre, tivemos alguma dificuldade para desenvolver com sucesso todos os objetivos propostos, e acreditamos que com mais tempo, teria sido possível alcançar todos os objectivos, nomeadamente medição da eficiência do Protocolo de Ligação de Dados.

ANEXOS

Link_layer.c

```
// Link layer protocol implementation

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


// MISC
#define _POSIX_SOURCE 1 // POSIX compliant source

struct termios oldtio;
struct termios newtio;
int alarmEnabled = FALSE;
int alarmCount = 0;

int maxNRetransmissions = 0;
int timeout = 0;
int fd;
int role;

unsigned char frame_tx = 0;
unsigned char frame_rx = 0;

// Alarm function handler
void alarmHandler(int signal){
    alarmEnabled = FALSE;
    alarmCount++;

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

// Send a supervision frame
int sendTrama(int fd, unsigned char Address, unsigned char Control){
    unsigned char BUFFER[5] = {FLAG, Address, Control, Address ^ Control, FLAG};
    return write(fd, BUFFER, 5);
}
```

```
int getResponse(){
    unsigned char response_buffer;
    unsigned char ctrl_field;
    states state = START;
    int b1;

    printf("in getResponse\n");
    while(state != STATE_STOP){
        b1 = read(fd,&response_buffer,1);
        if(b1 > 0){ //If read was successful
            switch (state){
                case START:
                    if(response_buffer == FLAG)
                        state = FLAG_RCV;
                    break;
                case FLAG_RCV:
                    printf("FLAG_RCV\n");
                    if(response_buffer == FLAG){
                        state = FLAG_RCV;
                    }
                    else if (response_buffer == ADDR_Rx){
                        state = A_RCV;
                    }
                    else{
                        state = START;
                    }
                    break;
                case A_RCV:
                    printf("A_RCV\n");
                    printf("response_buffer = 0x%02X\n", response_buffer);
                    if(response_buffer == FLAG){
                        state = FLAG_RCV;
                    }
                        if(response_buffer == CTRL_REJ0 || response_buffer == CTRL_REJ1 || response_buffer ==
CTRL_RR0 || response_buffer == CTRL_RR1 || response_buffer == CTRL_DISC){
                        state = C_RCV;
                        ctrl_field = response_buffer;
                    }
                    else{
                        state = START;
                    }
                    break;
                case C_RCV:
                    printf("C_RCV\n");
                    if(response_buffer == FLAG){
                        state = FLAG_RCV;
                    }
                    else if (response_buffer == (ADDR_Rx ^ ctrl_field)){
                        state =  BCC_RCV;
                    }
                    else{
```

```
                    state = START;
                }
                break;
            case BCC_RCV:
                if(response_buffer == FLAG){
                    state = STATE_STOP;
                }
                else{
                    state = START;
                }
                break;
            default:
                break;
            }
        }
        else if(b1 == 0){
            printf("Timeout -> NOTHING TO READ \n");
            return 0;
        }
        else{
            printf("Error reading from serial port\n");
            return -1;
        }
    }
    return ctrl_field;
}


// llopen function
int llopen(LinkLayer connectionParameters){

    unsigned char read_buffer[5] = {0};
    maxNRetransmissions = connectionParameters.nRetransmissions;
    timeout = connectionParameters.timeout;
    states state = START;
    role = connectionParameters.role;

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

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



    // 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 = connectionParameters.baudRate | CS8 | CLOCAL | CREAD;
newtio.c_iflag = IGNPAR;
newtio.c_oflag = 0;

newtio.c_lflag = 0;
newtio.c_cc[VTIME] = 0;
newtio.c_cc[VMIN] = 0;

tcflush(fd, TCIOFLUSH);

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

//State Machine
switch (connectionParameters.role){
    case LITx:{ //Transmitter
        (void) signal(SIGALRM, alarmHandler);
        alarmEnabled = FALSE;
        while(alarmCount < maxNRetransmissions && state != STATE_STOP){
            if(alarmEnabled == FALSE){
                //Send SET
                printf("Sending SET\n");
                sendTrama(fd, ADDR_Tx, CTRL_SET);
                alarm(timeout); // Activates alarm during timeout seconds
                alarmEnabled = TRUE;
            }
            //Read the UA response
            while(state != STATE_STOP && alarmCount < 3){
                int b1 = read(fd,read_buffer,1); //Reads one byte
                if(b1 > 0){
                    switch(state){
                        case START:
                            if(read_buffer[0] == FLAG){
                                state = FLAG_RCV;
                            }
                            break;
                        case FLAG_RCV:
                            if(read_buffer[0] == FLAG){
                                state = FLAG_RCV;
                            }
                            else if (read_buffer[0] == ADDR_Rx){
```

```
                    state = A_RCV;
                }
                else{
                    state = START;
                }
                break;
            case A_RCV:
                if(read_buffer[0] == FLAG){
                    state = FLAG_RCV;
                }
                else if(read_buffer[0] == CTRL_UA){
                    state = C_RCV;
                }
                else{
                    state = START;
                }
                break;
            case C_RCV:
                if(read_buffer[0] == FLAG){
                    state = FLAG_RCV;
                }
                else if (read_buffer[0] == (ADDR_Rx ^ CTRL_UA)){
                    state =  BCC_RCV;
                }
                else{
                    state = START;
                }
                break;
            case BCC_RCV:
                if(read_buffer[0] == FLAG){
                    state = STATE_STOP;
                    alarm(0);
                    alarmEnabled = 0;
                }
                else{
                    state = START;
                }
                break;
            default:
                printf("%d",state);
                break;
            }
        }
    }
    if (state != STATE_STOP) return -1;
    break;
}
case LIRx:{
    //Read SET
    while(state != STATE_STOP){
```

```
int b1 = read(fd,read_buffer,1);
    if(b1 > 0){
    switch(state){
        case START:
            if(read_buffer[0] == FLAG){
                state = FLAG_RCV;
            }
            break;
        case FLAG_RCV:
            if(read_buffer[0] == FLAG){
                state = FLAG_RCV;
            }
            else if (read_buffer[0] == ADDR_Tx){
                state = A_RCV;
            }
            else{
                state = START;
            }
            break;
        case A_RCV:
            if(read_buffer[0] == FLAG){
                state = FLAG_RCV;
            }
            else if(read_buffer[0] == CTRL_SET){
                state = C_RCV;
            }
            else{
                state = START;
            }
            break;
        case C_RCV:
            if(read_buffer[0] == FLAG){
                state = FLAG_RCV;
            }
            else if (read_buffer[0] == (ADDR_Tx ^ CTRL_SET)){
                state =  BCC_RCV;
            }
            else{
                state = START;
            }
            break;
        case BCC_RCV:
            if(read_buffer[0] == FLAG){
                state = STATE_STOP;
            }
            else{
                state = START;
            }
            break;
        default:
            printf("%d",state);
```

```
                    break;
                }
            }
        }
        //Send UA
        printf("Sending UA\n");
        sendTrama(fd, ADDR_Rx, CTRL_UA);
        break;
    }
    default: //Error
        printf("Error: Invalid role\n");
        return -1;
    }
  return fd;
}


// llwrite function
int llwrite(const unsigned char *buf, int bufSize){

  printf("Entered llwrite\n");
  int frameSize = 6 + bufSize; // 6 bytes are the fixed size of the Information frame

  (void) signal(SIGALRM, alarmHandler);

  // Create frame
  unsigned char* frame = malloc(frameSize);
  frame[0] = FLAG;
  frame[1] = ADDR_Tx;
  // To distinguish between frames (Control field) we use the frame_tx variable
  if(frame_tx % 2 == 0) frame[2] = C_I0;
  else frame[2] = C_I1;
  frame[3] = frame[1] ^ frame[2]; // BCC1 = A ^ C


  unsigned char BCC2 = 0;

  // Calculate BCC2
  for (int i = 0; i < bufSize; i++) BCC2 ^= buf[i];

  // Stuffing
  int frameIndex = 4; // Start at 4 because we already have 4 bytes
  for(int i = 0; i < bufSize; i++){
    if(buf[i] == FLAG || buf[i] == ESC){ // If we find a FLAG or ESC byte we need to stuff
      frame = realloc(frame, ++frameSize); // Increase frame size by 1 because we need to add the ESC byte
      frame[frameIndex++] = ESC;
      frame[frameIndex++] = buf[i] ^ XOR_STUFFING; // We add the ESC and after XOR between the byte and
0x20
    }
    else { // If we don't find a FLAG or ESC byte we just add the byte to the frame
      frame[frameIndex++] = buf[i];
```

```
        }
    }

    // Continue creating the frame
    frame[frameIndex++] = BCC2;
    frame[frameIndex] = FLAG; // End of frame


    int transmissionsCounter = 0;
    int accepted = FALSE;
    alarmEnabled = FALSE;

    while((transmissionsCounter <= maxNRetransmissions)){
        printf("Transmissions counter: %d\n", transmissionsCounter);
        if(alarmEnabled == FALSE){
            // Send frame
            printf("Sending frame\n");
            write(fd,frame, frameSize);
            alarm(timeout);
            sleep(1); // Sleep for 1 second to avoid sending the next frame before the receiver sends the RR
            alarmEnabled = TRUE;
            transmissionsCounter++;
        }
        // Get response
        unsigned char response = getResponse();
        if(response == CTRL_RR0 || response == CTRL_RR1){
            printf("Receive RR\n");
            accepted = TRUE;
            alarm(0);
            alarmEnabled = FALSE;
            frame_tx = (frame_tx+1) % 2; // Increment frame_tx to change the control field of the next frame
        }

        if(accepted) break;
    }
    if(accepted){
        printf("It was accepted (Received RR)\n");
        return frameSize; // Return the number of bytes written
    }
    else{
        //llclose(0);
        return -1;

    }
}


// llread function
int llread(unsigned char *packet){
    int index = 0;
    unsigned char b_read;
```

```
unsigned char control_field;
states state = START;
printf("Initiating reading process\n");

while(state != STATE_STOP){

    if(read(fd, &b_read, 1) > 0){

        switch (state){

        case START:
            if(b_read == FLAG)
                state = FLAG_RCV;
            break;

        case FLAG_RCV:
            if(b_read == ADDR_Tx)
                state = A_RCV;
            else if(b_read != FLAG)
                state = START;
            break;

        case A_RCV:
            //Confirm that frames are NOT repeated
            if((b_read == C_I0 && (frame_rx%2==0)) || (b_read == C_I1 && (frame_rx%2==1))){
                state = C_RCV;
                control_field = b_read;
            }
            else if ((b_read == C_I0 && (frame_rx%2!=0)) || (b_read == C_I1 && (frame_rx%2!=1))){
                //If it's a duplicate, the data field is discarded, but we must CONFIRM the frame with RR
                sendTrama(fd, ADDR_Tx, (R((frame_rx + 1)%2) | 0x05));  //Send RR0 or RR1
                return 0;
            }
            else if (b_read == CTRL_DISC) {
                sendTrama(fd, ADDR_Tx, CTRL_DISC);
                return 0;
            }
            else if (b_read == FLAG){
                state = FLAG_RCV; // We can have more than one flag
            }
            else
                state = START;
            break;

        case C_RCV:
            if(b_read == (control_field ^ ADDR_Tx))
                state = DATA_FIELD;
            else if(b_read == FLAG)
                state = FLAG_RCV;
            else
                state = START;
```

```
            break;

        case DATA_FIELD:
            if (b_read == ESC)
                state = DESTUFFING;

            else if (b_read == FLAG){
                index--;
                unsigned char BBC2 = packet[index];
                packet[index] = '\0';
                unsigned char data_check = packet[0];

                for (unsigned int k = 1; k < index; k++){
                    data_check ^= packet[k];
                }

                if (BBC2 == data_check){
                    state = STATE_STOP;

                    if(frame_rx % 2 == 0){
                    printf("BCC2 CORRECT: Sending confirmation of frame 0\n");
                    sendTrama(fd, ADDR_Rx, CTRL_RR0);
                    }
                    else if(frame_rx % 2 == 1){
                    printf("BCC2 CORRECT: Sending confirmation of frame 1\n");
                    sendTrama(fd, ADDR_Rx, CTRL_RR1);
                    }

                    frame_rx = (frame_rx + 1)%2;
                    return index;
                }
                else if(BBC2 != data_check){

                    state = START;
                    if(frame_rx % 2 == 0){
                    printf("ERROR! BCC2 INCORRECT: Frame 0 Rejected\n");
                    sendTrama(fd, ADDR_Rx, CTRL_REJ0);
                    }
                    else if(frame_rx % 2 == 1){
                    printf("ERROR! BCC2 INCORRECT: Frame 1 Rejected\n");
                    sendTrama(fd, ADDR_Rx, CTRL_REJ1);
                    }

                    return 1;
                };

            }
            else{
                packet[index++] = b_read;
            }
            break;
```

```
            case DESTUFFING:
                state = DATA_FIELD;
                packet[index++] = b_read^ XOR_STUFFING;
                break;

            default:
                break;
            }
        }
    }
    return 1;
}

//llclose function
int llclose(int showStatistics){
    states state = START;
    unsigned char read_buffer[5] = {0};
    (void) signal(SIGALRM, alarmHandler);

    switch(role){
        case LITx:{
            while(state != STATE_STOP){
                //Send first DISC to receiver
                printf("Sending DISC to Receiver\n");
                sendTrama(fd, ADDR_Tx, CTRL_DISC);
                alarm(timeout); // Activates alarm during timeout seconds
                alarmEnabled = TRUE;

                //Read the DISC that was sent back
                while(alarmEnabled == TRUE && state != STATE_STOP){
                    int b1 = read(fd,read_buffer,1); //Reads one byte
                    if(b1 > 0){
                        switch(state){
                        case START:
                            printf("START CLOSE\n");
                            if(read_buffer[0] == FLAG){
                                state = FLAG_RCV;
                            }
                            break;
                        case FLAG_RCV:
                            printf("FLAG_RCV CLOSE\n");
                            if(read_buffer[0] == FLAG){
                                state = FLAG_RCV;
                            }
                            else if (read_buffer[0] == ADDR_Rx){
                                state = A_RCV;
                            }
                            else{
                                state = START;
                            }
```

```
                    break;
                case A_RCV:
                    printf("A_RCV CLOSE\n");
                    if(read_buffer[0] == FLAG){
                        state = FLAG_RCV;
                    }
                    else if(read_buffer[0] == CTRL_DISC){
                        state = C_RCV;
                    }
                    else{
                        state = START;
                    }
                    break;
                case C_RCV:
                    printf("C_RCV CLOSE\n");
                    if(read_buffer[0] == FLAG){
                        state = FLAG_RCV;
                    }
                    else if (read_buffer[0] == (ADDR_Rx ^ CTRL_DISC)){
                        state =  BCC_RCV;
                    }
                    else{
                        state = START;
                    }
                    break;
                case BCC_RCV:
                    if(read_buffer[0] == FLAG){
                        state = STATE_STOP;
                        alarm(0);
                        alarmEnabled = FALSE;
                    }
                    else{
                        state = START;
                    }
                    break;
                default:
                    break;
                }
            }
        }
        if(state != STATE_STOP) return -1;
    }
}
case LlRx:{
    //Read DISC from transmitter
    while(state != STATE_STOP){
            int b1 = read(fd,read_buffer,1);
            if(b1 > 0){
                switch(state){
                case START:
                    printf("START CLOSE\n");
```

```
                if(read_buffer[0] == FLAG){
                    state = FLAG_RCV;
                }
                break;
            case FLAG_RCV:
                printf("FLAG_RCV CLOSE\n");
                if(read_buffer[0] == FLAG){
                    state = FLAG_RCV;
                }
                else if (read_buffer[0] == ADDR_Tx){
                    state = A_RCV;
                }
                else{
                    state = START;
                }
                break;
            case A_RCV:
                printf("A_RCV CLOSE\n");
                if(read_buffer[0] == FLAG){
                    state = FLAG_RCV;
                }
                else if(read_buffer[0] == CTRL_DISC){
                    state = C_RCV;
                }
                else{
                    state = START;
                }
                break;
            case C_RCV:
                printf("C_RCV CLOSE\n");
                if(read_buffer[0] == FLAG){
                    state = FLAG_RCV;
                }
                else if (read_buffer[0] == (ADDR_Tx ^ CTRL_DISC)){
                    state =  BCC_RCV;
                }
                else{
                    state = START;
                }
                break;
            case BCC_RCV:
                if(read_buffer[0] == FLAG){
                    state = STATE_STOP;
                }
                else
                    state = START;
                break;
            default:
                break;
            }
        }
```

```
            }
        // Send DISC to transmitter
        printf("Sending DISC to Transmitter\n");
        sendTrama(fd, ADDR_Rx, CTRL_DISC);
    }
    default:
        break;
    }

    //Send UA to close connection
    if(role == LITx) {
        printf("Sending UA to close connection\n");
        sendTrama(fd, ADDR_Tx, CTRL_UA);
    }


    if(close(fd) < 0)
        return -1;

    return 1;
}
```

Application_Layer.c

```
// Application layer protocol implementation

#include "application_layer.h"
#include "link_layer.h"
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdbool.h>
#include <stdlib.h>




int sendControlPacket(const char* filename,const int control_packet,long int fileLength){

    // Get file length in bytes
    size_t fileLengthL1 = 0;
    long int tempLength = fileLength;
    while (tempLength > 0) {
      fileLengthL1++;
```

```
    tempLength >>= 8;
  }
  long int fileNameLengthL2 = strlen(filename)+1;

  // Create control packet
  long int packet_size = 5 + fileLengthL1 + fileNameLengthL2;
  unsigned char* controlPacket = (unsigned char*)malloc(packet_size);

  // Fill control packet
  int idx = 0;
  controlPacket[idx++] = control_packet;

  //File size
  controlPacket[idx++] = 0x00; // T1 (file size)
  memcpy(controlPacket+idx, &fileLengthL1, sizeof(size_t));
  idx += sizeof(size_t);

  //File name
  controlPacket[idx++] = 0x01; // T2 (file name)
  memcpy(controlPacket + idx, filename, fileNameLengthL2);

  // Send control packet
  if(llwrite(controlPacket,packet_size) < 0){
    printf("Error sending control packet in llwrite().\n");
    free(controlPacket);
    return -1;
  }
  free(controlPacket);

  return 1;
}

int readControlPacket(const int control_packet,size_t * fileLength,unsigned char* packet){

  // Check control packet
  int idx = 0;
  if (packet[idx++] != control_packet) {
    printf("Invalid control packet.\n");
    return -1;
  }

  // Check file size
  if (packet[idx++] != 0x00) { // T1 (file size)
    printf("Invalid type file size packet.\n");
    return -1;
  }

  // Get file length
  unsigned char fileSizeBytes = packet[idx]; // L1 (file size)
  unsigned char aux[fileSizeBytes];
  memcpy(aux, packet + idx + 1, fileSizeBytes); // aux = V1 (file size)
```

```
    *fileLength = 0;
    for (int i = fileSizeBytes - 1; i >= 0; i--) {
        *fileLength |= aux[i] << (8 * i); // V1 (file size)
    }

    //Get file name
    unsigned char fileNameLength = packet[fileSizeBytes+4]; // L2 (file name)
    unsigned char* fileName = (unsigned char*)malloc(fileNameLength);
    memcpy(fileName, packet + fileSizeBytes + 5, fileNameLength); // V2 (file name)

    return 1;
}


int sendDataPacket(int dataSize,unsigned char* data){

    // Create data packet
    long int packet_size = 1 + 1 + 1 + dataSize;
    unsigned char* dataPacket = (unsigned char*)malloc(packet_size);

    // Fill data packet
    int idx = 0;
    dataPacket[idx++] = DATA_PACKET; // T1 (data)
    // K = 256 * L2 + L1
    dataPacket[idx++] = dataSize / 256; // L2 (data)
    dataPacket[idx++] = dataSize % 256; // L1 (data)
    for (int i = 0; i < dataSize; i++) {
        dataPacket[idx++] = data[i]; // P (data)
    }

    // Send data packet
    if(llwrite(dataPacket, packet_size) < 0){
        printf("Error sending data packet in llwrite().\n");
        return -1;
    }
    return 0;
}

int sendFile(const char* filename){

     // Open file
    FILE* file = fopen(filename, "rb");
    if (file == NULL) {
        printf("Error opening file.\n");
        exit(-1);
    }

    // Get file size (Mudar)
    int previousFile = ftell(file); // save previous position
    fseek(file, 0L, SEEK_END); // file pointer at end of file
    long int fileSize = ftell(file) - previousFile; // get file length
```

```
    fseek(file, previousFile, SEEK_SET); // go back to previous position

     // Send start control packet
    if(sendControlPacket(filename, START_PACKET, fileSize) < 0){
        printf("Error sending start control packet.\n");
        return -1;
    }

    // Send file
    unsigned char* data = (unsigned char*)malloc(MAX_PAYLOAD_SIZE-3); // -3 because of the header (C, L2,
L1)
    int chunkDataSize;
    while((chunkDataSize = fread(data, 1, MAX_PAYLOAD_SIZE-3, file)) > 0){
        if(sendDataPacket(chunkDataSize,data) < 0){
            printf("Error sending data packet.\n");
            return -1;
        }
    }
    free(data);
    fclose(file);

    // Send end control packet
    if(sendControlPacket(filename, END_PACKET, fileSize) < 0){
        printf("Error sending end control packet.\n");
        return -1;
    }
    printf("Sent end packet.\n");

    // Close connection
    if (llclose(0) < 0) {
        printf("Error send closing connection.\n");
        return -1;
    }

    return 1;
}


int receiveFile(const char* filename){
    size_t packetSize;

    unsigned char* packet = (unsigned char*)malloc(MAX_PAYLOAD_SIZE);
    if(llread(packet) < 0){
        printf("Error receiving start control packet.\n");
        return -1;
    }

    // Read start control packet
    if(readControlPacket(START_PACKET, &packetSize, packet) < 0){
        printf("Error reading start control packet.\n");
        return -1;
```

```
    }


    FILE* receiveFile = fopen((char *) filename, "wb+");

    if(receiveFile == NULL){
        printf("Error opening received file.\n");
        exit(-1);
    }

    // Receive file
    free(packet);
    int dataSize = 0;
     while((dataSize = llread(packet)) >= 0){
        printf("DATASIZE: %d\n",dataSize);
        if(packet[0] == DATA_PACKET){
            printf("Data packet received.\n");
            fwrite(packet + 3, 1, dataSize - 3, receiveFile);
            }
        else if(packet[0] == END_PACKET){
                printf("End packet received.\n");
                break;
            }
        }
    free(packet);
    fclose(receiveFile);
    printf("Before closing connection.\n");
    if(llclose(0) < 0){
        printf("Error receive closing connection.\n");
        return -1;
    }
    return 1;
}


void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename){

    // Create connection parameters
    LinkLayer connectionParameters;
    connectionParameters.baudRate = baudRate;
    connectionParameters.nRetransmissions = nTries;
    connectionParameters.timeout = timeout;
    strcpy(connectionParameters.serialPort, serialPort);
    if (strcmp(role, "tx") == 0) connectionParameters.role = LITx;
    else if (strcmp(role, "rx") == 0) connectionParameters.role = LIRx;
    else printf("Invalid role\n");

    // Establish connection
    int fd = llopen(connectionParameters);
    if (fd <= 0) {
```

```
        printf("Error opening serial port.\n");
        exit(-1);
    }

    // Send / receive file
    switch (connectionParameters.role){
        case LlTx:
            if(sendFile(filename) < 0){
                printf("Error sending file.\n");
                exit(-1);
            }
            break;
        case LlRx:
             if(receiveFile(filename) < 0){
                printf("Error sending file.\n");
                exit(-1);
            }
            break;
        default:
            break;
    }
}
```