Relatório de Redes de Computadores

Trabalho laboratorial 1 - Protocolo de ligação de dados

Realizado por Nuno Gonçalves (up201706864@fe.up.pt)

Sumário

No âmbito da disciplina de Redes de Computadores da Licenciatura de Engenharia Informática e Computação da Faculdade de Engenharia do Porto, realizei um trabalho laboratorial com o objetivo de criar um protocolo de ligação de dados que consiga transferir ficheiros entre dois computadores ligados por um cabo de série.

No final deste trabalho, infelizmente não posso afirmar que o trabalho tenha sido realizado com sucesso, pois a transmissão de ficheiros é demasiado inconsistente e errónea com este protocolo, não obedecendo às especificações do programa nem tendo boa resistência a erros induzidos.

Introdução

Como referido anteriormente, neste trabalho realizou-se um protocolo de ligação de dados para transferir ficheiros entre computadores. Com este protocolo, um emissor envia um ficheiro dividido em pacotes (com tamanho a escolher como argumento na invocação) para um receptor depois de estabelecer comunicação com esta máquina. Pretende-se que este protocolo seja fiável e resistente a erros, embora não se tenha conseguido concretizar esse objetivo devido a falhas inesperadas.

Seguidamente neste relatório irá encontrar várias secções explicitando diferentes partes do trabalho:

- **Arquitetura:** Nesta secção encontrará informação sobre os vários blocos funcionais e interfaces que constituem o programa.
- **Estrutura do código:** Aqui será relatada a informação sobre as APIs, principais estruturas de dados e funções, e estas serão relacionadas com a arquitetura apresentada anteriormente.
- **Casos de uso principais:** Aqui estão descritos não só os casos de uso deste programa como também a sequência de chamadas feitas nestes casos de uso.
- **Protocolo de ligação lógica:** Nesta secção serão identificados os principais aspetos funcionais da ligação lógica, e descrita a estratégia de implementação destes aspetos com excertos de código como exemplo.
- **Protocolo de aplicação:** Esta secção será idêntica à anterior, mas referente a aspetos funcionais da aplicação em vez da ligação lógica.
- **Validação:** Aqui serão descritos os testes utilizados para testar o protocolo e descritos os respetivos resultados.
- **Eficiência do protocolo de ligação:** Nesta secção será caracterizada a eficiência do protocolo, efetuada recorrendo a medidas sobre o código desenvolvido e comparando-a com um protocolo de stop and wait.
- **Conclusões:** Finalmente, será apresentada uma síntese da informação do relatório, refletindo sobre os objetivos do trabalho e em que medida estes foram alcançados.

No final destes capítulos serão apresentados anexos apresentando o código fonte e outra informação que pode ser considerada importante.

Arquitetura

Para este protocolo dividiu-se o código em duas camadas: a camada de aplicação e a camada de ligação lógica. Ambas as camadas são responsáveis por realizar funções diferentes do código, estando a camada de aplicação construída em *app_layer.c* e a de ligação lógica em *link_layer.c* e *link_layer.h*.

A camada de aplicação é responsável pelo processamento de argumentos da chamada do programa, pela construção e desempacotamento de pacotes de dados e pela leitura do ficheiro a transferir no emissor e escrita deste no receptor. No entanto, para a transferência dos pacotes precisa de chamar funções definidas na interface da camada de ligação lógica.

Na camada de ligação lógica, são definidas as funções que se usa para a comunicação dos dois computadores, como a abertura e o fecho da porta de comunicação como também a leitura e escrita de dados na porta. No entanto, a distinção entre pacotes não é feita nesta camada, sendo feito o processamento destes só quando chegam à camada de aplicação.

Também há um ficheiro de definições em *defs.h* em que são definidas várias constantes importantes para o programa.

Na interface define-se que porta se quer usar, em que modo (emissor ou recetor) se quer correr o programa e em caso de ser emissor, que ficheiro se quer transferir e que tamanho cada pacote deverá ter.

Estrutura do código

Neste trabalho decidi não usar estruturas de dados para representar cada camada, no entanto cada camada tem variáveis globais que cumprem essa função, armazenando informação importante a toda a camada.

Camada de aplicação

Na camada de aplicação, as variáveis globais contém o descritor que identifica a porta aberta pela ligação lógica, o modo em que o programa está a correr (0 se emissor ou 1 se recetor), um número para saber que porta abrir, o caminho (path) do ficheiro, o tamanho de cada pacote a transferir, o identificador do ficheiro, o tamanho total do ficheiro e o número total de pacotes necessários para transferir o ficheiro todo.

```
int fd;
unsigned int mode;
int portNr;
char* path;
unsigned int packet_size;
FILE file;
long file_size;
int file_packets;
```

As principais funções desta camada são:

```
int main(int argc, char* argv[]);
int buildControlPacket(uint8_t** packet);
```

```
int buildDataPacket(uint8_t** packet, int packet_nr, uint8_t* data, int data_size);
void unpackControlPacket(unit8_t* packet, char** filename);
```

Camada de ligação lógica

Na camada de ligação lógica há variáveis globais contendo várias informações que precisam de ser armazenadas, como o identificador da porta a abrir, o baudrate de transmissão, o número de sequência atual, o número de segundos que causa um timeout, o número máximo de timeouts que podem ocorrer antes de parar a execução do programa, o modo em que o programa está a correr, a trama a ser lida ou escrita e as definições da porta anteriores. Também há outras duas variáveis globais para representar o estado em que certas funções se encontram, uma delas sendo uma contagem de timeouts consecutivos que ocorreram no envio da trama atual, e outra uma flag para armazenar o estado em ciclos de leitura e escrita.

```
char* port_name;
int baudrate;
unsigned int sequence_number;
unsigned int timeout;
unsigned int max_timeouts;
unsigned int mode;
uint8_t frame[MAX_PACKET_SIZE];
struct termios oldtio;
int timeout_count;
volatile int flag;
```

As principais funções desta camada são:

```
int llopen(int port, int mode);
int llwrite(int fd, uint8_t* buffer, int length);
int llread(int fd, uint8_t* buffer);
int llclose(int fd);
```

Também há bastantes funções auxiliares a estas, incluindo funções para enviar e receber mensagens pela porta, fazer stuffing e destuffing a tramas, verificar e mudar valores e tratar de timeouts.

Casos de uso principais

Como já referido anteriormente, o programa tem dois casos de uso principais: executar como emissor, enviando um ficheiro, ou como receptor, recebendo o ficheiro enviado pelo emissor.

Emissor

Quando o programa executa como emissor, primeiro o programa começa por chamar **llopen()** após serem processados os argumentos, que abre a porta de comunicação e recebendo o modo como um dos argumentos, executa **setupTransmiter()** vendo que o modo é de emissor, função que envia uma mensagem SET e espera por uma mensagem UA para ver se a comunicação foi corretamente estabelecida. Seguidamente, a função **llopen()** retorna, e se a conexão for estabelecida corretamente, prossegue-se usando

buildControlPacket() para construir o pacote de início, e envia-o com **llwrite()**, em que o pacote é colocado numa trama de informação que é enviada com **sendInfoFrame()** após ser feito o stuffing com **byteStuffing()**. A seguir a isto se o pacote foi enviado com sucesso, entra-se num loop em que o programa vai lendo do ficheiro a enviar, são construídos pacotes de dados com **buildDataPacket()** e estes são enviados para o receptor com **llwrite()**. Este loop acaba ou quando o ficheiro é enviado completamente ou quando há um timeout, executando **llwrite()** uma última vez para enviar o pacote de fim. Finalmente, o ficheiro é fechado, o espaço usado por variáveis é libertado e a porta é fechada com **llclose()**, que usa a função **sendDisconectMessage()** para enviar uma mensagem DISC ao receptor e esperar pela respetiva resposta, ao fim da qual envia uma mensagem UA e a porta é fechada, acabando a execução do programa.

Receptor

Quando o programa é executado como recetor, **llopen()** é aberto semelhantemente ao emissor, mas em vez de executar **setupTransmiter(),** executa **setupReceiver()** que espera pela mensagem SET e envia UA quando recebida. Após **llopen()** retornar, entra-se no loop de leitura em que se vai chamando **llread()** até este receber uma mensagem de desconexão ou há um erro na conexão com a porta. Em **llread()** é chamada **readPacket()** em que se lê o pacote recebido e armazena-se este num buffer, que após **llread()** retornar, se o pacote lido for o pacote de início armazena-se informação do ficheiro com **unpackControlPacket()** e é aberto um ficheiro vazio para se armazenar o ficheiro a receber, e se for um pacote de dados estes dados são escritos no ficheiro. Quando **llread()** recebe uma mensagem de desconexão (DISC), chama-se **sendDisconectAnswer()** para enviar DISC de volta para o emissor e espera-se por UA, continuando se acontecer um timeout. Finalmente, é efetuado um processo semelhante ao fim do modo de emissor, exceto pelo facto de **sendDisconectMessage()** não ser chamado dentro de **llclose().**

Protocolo de ligação lógica

A camada de ligação lógica é onde se realiza a comunicação entre as duas máquinas a partir da porta de série, podendo abri-la, fechá-la, enviar mensagens e recebê-las. Também é nesta camada que se cria tramas de supervisão e não numeradas e faz-se o stuffing e destuffing de pacotes vindos da camada de aplicação.

Nesta camada, as funções mais importantes são:

- int llopen(int port, int mode);

Nesta função abre-se a porta série identificada por *port* para comunicação e envia-se as mensagens necessárias para saber se a porta foi aberta com sucesso. Para isso, após abrir a porta chamando **setupPort()** corre-se **setupTransmiter()** se o modo for transmissor e **setupReceiver()** se o modo for receptor. Nessas funções, o emissor envia SET e espera por UA vindo do receptor, enquanto que o receptor espera por SET e envia UA quando este é recebido, terminando assim o processo de abertura quando todas estas mensagens forem enviadas e recebidas.

- int llwrite(int fd, uint8_t* buffer, int length);

Nesta função escreve-se para a porta de série uma trama de informação. Recebe-se como argumento um pacote da camada de aplicação em *buffer* com um número de bytes *length*, e coloca-se este dentro de uma trama de informação, gerando BCC2 e fazendo-se o byte stuffing da trama. Seguidamente, a mensagem é enviada para o recetor com **sendInfoFrame()** e espera-se por uma resposta do cliente, terminando a função retornando o número de bytes escritos após recebê-la, ou -2 em caso de timeout.

- int llread(int fd, uint8_t* buffer);

Nesta função é logo chamada a função **readFrame()** após verificar-se que a porta indicada está aberta, em que se espera por uma mensagem do emissor e armazena-se esta em *buffer* ao ser lida. Depois faz-se o byte destuffing da mensagem se for uma trama de informação e envia-se uma mensagem REJ se houver erros em BCC2 ou uma mensagem RR se não houver, retornando no final o número de bytes lidos. Caso seja recebida uma mensagem DISC, o receptor executa **sendDisconectAnswer()** em que envia DISC de volta ao emissor, e espera-se por UA até ocorrer um timeout ou esta mensagem ser recebida. Aí envia-se um valor especial diferente dos bytes lidos para sinalizar à app_layer o final do loop de leitura.

- int llclose(int fd);

Nesta função a porta de série é restaurada às definições anteriores e fechada posteriormente. No caso do programa executar no modo de emissor, é executado **sendDisconectMessage()** antes disto, em que a mensagem DISC é enviada, espera-se pela resposta DISC e envia-se UA no final para sinalizar ao receptor para fechar a porta e parar a execução.

Protocolo de aplicação

A camada de aplicação, ao contrário da de ligação lógica, não comunica com a porta série e está encarregue de ler do ficheiro a enviar e escrever o ficheiro recebido, criação e processamento de pacotes de dados e de controlo e interpretação dos argumentos da chamada da linha de comandos.

Embora esta camada tenha algumas funções auxiliares para a construção e desempacotamento de pacotes, a maioria da sua funcionalidade localiza-se na função **main()**. Esta função começa por atribuir os parâmetros da chamada da função a variáveis, fazendo cálculos para saber o número de pacotes em que se divide o ficheiro no caso de executar no modo de emissor, e **llopen()** é chamado para abrir a porta. A seguir a isto, tanto o emissor e o transmissor entram em partes diferentes da função em que os seus loops principais são executados.

No modo de emissor cria-se o pacote de início com **buildControlPacket()** e envia-se este com **llwrite()**, e se for enviado com sucesso, entra-se no loop de escrita do ficheiro. Aqui vão sendo criados pacotes de dados com os dados lidos do ficheiro a enviar, e estes são enviados com **llwrite()**, executando este ciclo até ser enviada a totalidade do ficheiro ou haver um timeout. No final deste ciclo, se não tiver acontecido um timeout, é enviado um final pacote de controlo a sinalizar o fim do loop.

No modo de receptor, entra-se no loop de leitura em que se são recebidos os pacotes enviados pelo emissor com **llread()**, desempacotando o pacote de início quando o recebe para saber os critérios do ficheiro a receber e seguidamente recebendo os pacotes de dados e

escrevendo-os para um ficheiro com o mesmo nome do ficheiro recebido. Quando um valor de desconexão é retornado por **llwrite()**, pára-se o loop.

No final da execução do programa, ambos os modos fecham o ficheiro, libertam espaço ocupado por buffers e executam **llclose()** para fechar a porta série.

Validação

Neste trabalho laboratorial era suposto haver testes que testassem as várias funcionalidades do programa, no entanto, houve inconvenientes e erros inesperados que dificultaram esta implementação, o que deixou os testes realizados bastante aquém do esperado.

No entanto, foram realizados alguns testes enviando vários ficheiros com este protocolo. Para isto, enviou-se vários ficheiros, para além de *pinguim.gif* (10,7 KB) de vários tamanhos e alguns tipos diferentes para testar como a transferência ocorria neles: *test.txt* (598 B), *GrassPatch.png* (1,45 KB), *snowymountainsbackground.png* (117 KB) e *Periodic_table_large.png* (2,24 MB). Também se testou transferências com vários tamanhos de pacotes para ver as diferenças na variação deste critério na transmissão.

Eficiência do protocolo de ligação de dados

Este protocolo usa um mecanismo de *Stop and Wait* em que após o emissor enviar uma mensagem, espera por uma resposta do receptor até continuar a execução, e a mensagem inicial é reenviada em caso de não receber uma mensagem do receptor em certo intervalo de tempo (timeout). Para saber se a resposta recebida é da mensagem atual e não uma retransmissão, faz-se o uso de números de sequência, que alternam entre tramas.

Embora o protocolo implementado funcione a certo nível, as retransmissões quando há erros de BCC2 não estão a funcionar. Como este erro só foi detetado tardiamente no desenvolvimento do trabalho, e a tentativa de resolução acabou por quebrar mais o código que estava antes, acabou por não se implementar o *Stop and Wait* totalmente, deixando as retransmissões de BCC2 de fora.

Conclusões

O protocolo desenvolvido tem como objetivo transferir ficheiros entre máquinas, e enquanto consiga fazer isso nalguns casos, tem demasiados erros para cumprir os objetivos enunciados no trabalho. Entre estes encontram-se a falta de retransmissões em erros de BCC2, que causam com que os ficheiros tenham erros na versão transferida a partir da trama em que o erro ocorreu. Este número de erros é proporcional ao número de pacotes usados para a transferência, tendo maior probabilidade de acontecer em quanto mais pacotes o ficheiro seja dividido.

Outro erro bizarro ocorrido foi que quando executado nos computadores do laboratório, em **llread()** o computador lê um número erróneo de bytes muito maior do que devia, levando a que a execução do protocolo não seja possível. Enquanto tenha tentado resolver este erro, a dificuldade em ligar-me a ambos os computadores por ligação ssh em casa pois em grande parte das vezes um dos computadores não encontrava a rota para o outro (“No route to host”), fez com que não conseguisse testar porque este erro estava a ocorrer, deixando-o por corrigir.

Com isto posso dizer que os objetivos do trabalho não foram cumpridos. No entanto, mesmo com estas falhas todas, posso dizer que depois da realização deste trabalho fiquei a perceber mais sobre a comunicação entre camadas de um programa e a comunicação entre computadores.

Anexo - Código fonte

app_layer.c

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

#include "defs.h"
#include "link_layer.h"

int fd;
unsigned int mode;
int portNr;

char* path;
unsigned int packet_size;
FILE* file;
long file_size;
int file_packets;

int buildControlPacket(uint8_t** packet){
    unsigned int packet_size;
    char* filename;

    char* marker = strrchr(path, '/');
    if(marker != NULL){
        filename = &(marker[1]);
    }

    int fs_oct = 0;
```

```
    long fs_temp = file_size;

    while(fs_temp > 0){
        fs_temp >>= 8;
        fs_oct++;
    }

    int fn_length = strlen(filename);
    packet_size = 6 + fs_oct + fn_length;
    *packet = (uint8_t*) malloc(packet_size);

    (*packet)[0] = START_PACKET;
    (*packet)[1] = SIZE_PARAM;
    (*packet)[2] = fs_oct;

    for(int i = 0; i < fs_oct; i++){
        int size_byte = file_size >> 8 * (fs_oct - i - 1);
        (*packet)[3 + i] = size_byte;
    }

    (*packet)[3 + fs_oct] = NAME_PARAM;
    (*packet)[4 + fs_oct] = fn_length;

    for(int i = 0; i < fn_length; i++){
        (*packet)[5 + fs_oct + i] = filename[i];
    }

    return packet_size;
}

int buildDataPacket(uint8_t** packet, int packet_nr, uint8_t* data, int
data_size){
    unsigned int packet_size = data_size + 5;
    *packet = malloc(packet_size);

    (*packet)[0] = DATA_PACKET;
    (*packet)[1] = packet_nr;
    (*packet)[2] = data_size / 256;
    (*packet)[3] = data_size % 256;

    for(int i = 0; i < data_size; i++){
        (*packet)[i + 4] = data[i];
    }
```

```
    printf("Packet size: %i\n", packet_size);
    return packet_size;
}

void unpackControlPacket(uint8_t* packet, char** filename){
    //printf("Unpack entered\n");
    int pointer = 2;
    int fs_oct = packet[pointer];
    pointer++;

    file_size = 0;
    for(int i = 0; i < fs_oct; i++){
        file_size += (packet[pointer] << 8 *(fs_oct - i - 1));
        pointer++;
    }

    pointer++;

    int fn_length = packet[pointer];
    pointer++;
    *filename = malloc(fn_length + 1);

    for(int i = 0; i < fn_length; i++){
        (*filename)[i] = packet[pointer];
        pointer++;
    }

    (*filename)[fn_length] = '\0';
    printf("Filename: %s\n", *filename);
}

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

    //Process args for transmitter and receiver modes
    if(argc < 3){
            printf("Usage: file_transfer <port 0|1|2|10|11> <mode 0|1>
<file path> <packet size>\n");
        printf("port: /dev/ttySn, where n is the number entered\n");
        printf("mode: 0 for transmiter and 1 for receiver\n");
          printf("file path and packet size should only be entered if
mode is 0 (transmiter)\n");
```

```
        return -1;
    }

    portNr = strtol(argv[1], NULL, 10);
    mode = strtol(argv[2], NULL, 10);
    uint8_t* buffer;

    if(mode != TRANSMITER && mode != RECEIVER){
           printf("Invalid mode; mode can only be 0 (transmiter) or 1
(receiver)\n");
        return -1;
    }

    if(mode == TRANSMITER){
        if(argc < 5){
             printf("Usage: file_transfer <port 0|1|2|10|11> <mode 0|1>
<file path> <packet size>\n");
                   printf("port: /dev/ttySn, where n is the number
entered\n");
            printf("mode: 0 for transmiter and 1 for receiver\n");
             printf("file path and packet size should only be entered if
mode is 0 (transmiter)\n");
            return -1;
        }

        path = argv[3];
        file = fopen(path, "rb");
        //file_fd = fileno(file);
        printf("Opened file\n");

        if(file == NULL){
            perror("Couldn't open the file indicated in <file path>");
            return -1;
        }

        fseek(file, 0, SEEK_END);
        file_size = ftell(file);
        fseek(file, 0, SEEK_SET);

        packet_size = strtol(argv[4], NULL, 10);
        buffer = malloc(packet_size);

        file_packets = file_size / packet_size;
```

```
        if(file_size % packet_size > 0){
            file_packets++;
        }

        if(packet_size > (MAX_PACKET_SIZE - FRAME_INFO_SIZE)){
            perror("Packet size too big");
            return -1;
        }
    }

    fd = llopen(portNr, mode);
    printf("Comunication port opened\n");

    if(fd < 0){
        return -1;
    }

    uint8_t* control_packet = NULL;
    uint8_t* data_packet = NULL;
    int file_op_return;

    //Read loop if receiver
    if(mode == RECEIVER){
        int bytes_read = 0;
        char* filename;
        buffer = malloc(256);

        while(1){
            bytes_read = llread(fd, buffer);
            printf("llread finished with a value of %i\n", bytes_read);

             if(bytes_read == DISC_RETURN || bytes_read == DISC_ERROR ||
bytes_read < 0){
                break;
            }

            if(buffer[0] == START_PACKET){
                unpackControlPacket(buffer, &filename);

                if(file_size > 256){
                    buffer = realloc(buffer, file_size);
                }
```

```
                file = fopen(filename, "wb");
                printf("Opened file\n");
                //file_fd = fileno(file);
                //printf("Fileno\n");
            }

                else if(buffer[0] == DATA_PACKET && bytes_read !=
BCC2_ERROR){

                file_op_return = fwrite(&buffer[4], 1, bytes_read - 10,
file);
                printf("Wrote %i bytes\n", bytes_read - 9);

                if(file_op_return < 0 || feof(file) > 0){
                    perror("Error while writing in file");
                    continue;
                }
            }

            else if(buffer[0] == END_PACKET){
                printf("End packet received\n");
            }
        }

        if(bytes_read < 0){
            perror("Error on reading from port; aborting");
        }
    }

    //Write loop if transmitter
    if(mode == TRANSMITER){
        int cp_length = buildControlPacket(&control_packet);
        printf("Control packet built\n");
        int bytes_writen = llwrite(fd, control_packet, cp_length);

        if(bytes_writen > 0){
            printf("Control packet sent, sending data\n");
            int data_packets_writen = 0;
            int bytes_to_read = packet_size;

            while(data_packets_writen < file_packets){
                if((file_size / packet_size) == data_packets_writen){
                    bytes_to_read = file_size % packet_size;
```

```
                }

                file_op_return = fread(buffer, 1, bytes_to_read, file);

                if(file_op_return != 1 || feof(file) > 0){
                    if(ferror(file)){
                        perror("Error while reading file");
                        continue;
                    }
                }

                        int dp_length = buildDataPacket(&data_packet,
data_packets_writen, buffer, bytes_to_read);
                //printf("Sending data\n");
                bytes_writen = llwrite(fd, data_packet, dp_length);
                //printf("Data packet sent\n");

                if(bytes_writen == -2){
                    break;
                }

                data_packets_writen++;
            }
        }

        if(bytes_writen != -2){
            control_packet[0] = END_PACKET;
            llwrite(fd, control_packet, cp_length);
        }
    }

    fclose(file);

    free(buffer);
    if(control_packet){
        free(control_packet);
    }
    if(data_packet){
        free(data_packet);
    }

    printf("Closing file\n");
    if(llclose(fd) != 1){
```

```
        return -1;
    }

    return 0;
}
```

link_layer.c

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

#include "defs.h"
#include "link_layer.h"

char* port_name;
int baudrate = DEFAULT_BAUD;
unsigned int sequence_number;
unsigned int timeout = 1;
unsigned int max_timeouts = 5;

//0 if transmiter and 1 if receiver
unsigned int mode;

uint8_t frame[MAX_PACKET_SIZE];

struct termios oldtio;

int timeout_count = 0;

//flag controls the timeout mechanism; 0 if waiting, 1 if timeout and 2
if received
volatile int flag = 0;

void timeout_handler()
```

```
{
    printf("timeout n°%d\n", timeout_count);
    flag = 1;
    timeout_count++;
}

void flipSequenceNumber(){
    if(sequence_number == 1){
        sequence_number = 0;
    }
    else if(sequence_number == 0){
        sequence_number = 1;
    }
}

//Opens the port identified by the port number
//Returns fd on sucess and -1 on failure
int setupPort(int port){

    int fd;

    switch (port)
    {
    case 0:
        port_name = "/dev/ttyS0";
        break;

    case 1:
        port_name = "/dev/ttyS1";
        break;

    case 2:
        port_name = "/dev/ttyS2";
        break;

    case 10:
        port_name = "/dev/ttyS10";
        break;

    case 11:
        port_name = "/dev/ttyS11";
        break;
```

```
    default:
        perror("Invalid port number");
        return -1;
    }

    fd = open(port_name, O_RDWR);

    if(fd < 0){
        perror("Failed to open specified port");
        return -1;
    }

     if (tcgetattr(fd, &oldtio) == -1) { /* save current port settings
*/
      perror("Error on tcgetattr");
      return -1;
    }

    struct termios newtio;

    bzero(&newtio, sizeof(newtio));
    newtio.c_cflag = baudrate | CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;

    newtio.c_lflag = 0;

    newtio.c_cc[VTIME] =0.1;
    newtio.c_cc[VMIN] = 0;

    tcflush(fd, TCIOFLUSH);

    if (tcsetattr(fd, TCSANOW, &newtio) == -1) {
      perror("Error on tcsetattr");
      return -1;
    }

    return fd;
}

int checkAcknowledgement(uint8_t ack){

    if(ack == RR_1 || ack == REJ_1){
```

```
        if(sequence_number == 0){
            return 1;
        }
        else{
            perror("Wrong sequence number value in acknowledgement");
            return 0;
        }
    }

    if(ack == RR_0 || ack == REJ_0){
        if(sequence_number == 1){
            return 1;
        }
        else{
            perror("Wrong sequence number value in acknowledgement");
            return 0;
        }
    }

    return 0;
}

int checkControlField(uint8_t c){
     if(c == CTR_I_FRAME_1 || c == CTR_I_FRAME_0 || c == SET || c ==
DISC){
        return 1;
    }

    return 0;
}

int checkBcc2(uint8_t bcc2, int msg_length){
    uint8_t test_bcc2;

    for (int i = 0; i < msg_length; i++)
    {
        if(i == 0){
            test_bcc2 = frame[i];
        }
        else{
            test_bcc2 ^= frame[i];
        }
    }
```

```
    if(test_bcc2 == bcc2){
        return 1;
    }

    return 0;
}

void byteStuffing(int *length){
    int extra_length = 0;
    int prev_length = *length;

    for(int i = INITIAL_FRAME_BITS; i < prev_length - 1; i++){
        if(frame[i] == FLAG || frame[i] == ESCAPE){
            extra_length++;
        }
    }

    *length += extra_length;
    uint8_t temp_frame[*length];
    int j = 0;

    for(int i = INITIAL_FRAME_BITS; i < prev_length - 1; i++){
        if(frame[i] == FLAG){
            temp_frame[j] = ESCAPE;
            temp_frame[j+1] = FLAG ^ XOR_BYTE;
            j += 2;
        }
        else if(frame[i] == ESCAPE){
            temp_frame[j] = ESCAPE;
            temp_frame[j+1] = ESCAPE ^ XOR_BYTE;
            j += 2;
        }
        else{
            temp_frame[j] = frame[i];
            j++;
        }
    }

    for(int i = 0; i < *length - FRAME_INFO_SIZE + 1; i++){
        frame[i + INITIAL_FRAME_BITS] = temp_frame[i];
    }
```

```
    frame[*length - 1] = FLAG;
}

void byteDestuffing(int *length){
    int extra_length = 0;
    int prev_length = *length;

    for(int i = INITIAL_FRAME_BITS; i < prev_length - 1; i++){
         if(frame[i] == ESCAPE && (frame[i + 1] == (FLAG ^ XOR_BYTE) ||
frame[i + 1] == (ESCAPE ^ XOR_BYTE))){
            extra_length++;
        }
    }

    *length -= extra_length;
    uint8_t temp_frame[*length];
    int j = 0;

    for(int i = INITIAL_FRAME_BITS; i < prev_length - 1; i++){
        if(frame[i] == ESCAPE){
            if(frame[i + 1] == (FLAG ^ XOR_BYTE)){
                temp_frame[j] = FLAG;
            }
            else if(frame[i + 1] == (ESCAPE ^ XOR_BYTE)){
                temp_frame[j] = ESCAPE;
            }
            i++;
        }
        else{
            temp_frame[j] = frame[i];
        }
        j++;
    }

    for(int i = 0; i < *length - FRAME_INFO_SIZE + 1; i++){
        frame[i + INITIAL_FRAME_BITS] = temp_frame[i];
    }

    frame[*length - 1] = FLAG;
}

int setupTransmiter(int fd){
    sequence_number = 0;
```

```
    mode = 0;

    uint8_t set[5] = {FLAG, A_TR, SET, A_TR ^ SET, FLAG};

    uint8_t buffer[255];
    int state;
    timeout_count = 0;
    flag = 0;

    (void) signal(SIGALRM, timeout_handler);

    while(timeout_count < max_timeouts && flag < 2){
        flag = 0;
        alarm(timeout);

        write(fd, set, 5);

        state = OTHER_RCV;

        while (flag == 0) {

            read(fd, buffer, 1);

            switch(state){
                case OTHER_RCV:
                    if(buffer[0] == FLAG) { state = FLAG_RCV; }
                    break;

                case FLAG_RCV:
                    if(buffer[0] == FLAG) { state = FLAG_RCV; }
                    else if(buffer[0] == A_RE) { state = A_RCV; }
                    else { state = OTHER_RCV; }
                    break;

                case A_RCV:
                    if(buffer[0] == FLAG) { state = FLAG_RCV; }
                    else if(buffer[0] == UA) { state = C_RCV; }
                    else { state = OTHER_RCV; }
                    break;

                case C_RCV:
                    if(buffer[0] == FLAG) { state = FLAG_RCV; }
```

```
                        else if(buffer[0] == (A_RE ^ UA)) { state =
BCC_RCV; }
                    else { state = OTHER_RCV; }
                    break;

                case BCC_RCV:
                    if(buffer[0] == FLAG) {
                        alarm(0);
                        flag = 2;
                    }
                    else { state = OTHER_RCV; }
                    break;
            }

        }
    }

    if(timeout_count >= max_timeouts){
        perror("Too many timeouts ocurred, couldn't acnowledge port");
        return -1;
    }

    return 0;
}

int setupReceiver(int fd){
    sequence_number = 1;
    mode = 1;

    uint8_t ua[5] = {FLAG, A_RE, UA, A_RE ^ UA, FLAG};

    uint8_t buffer[255];
    int state = OTHER_RCV;
    int stop = FALSE;

    while (stop == FALSE) {
      read(fd, &buffer[0], 1);

      switch(state){
        case OTHER_RCV:
            if(buffer[0] == FLAG) { state = FLAG_RCV; }
            break;
```

```
        case FLAG_RCV:
            if(buffer[0] == FLAG) { state = FLAG_RCV; }
            else if(buffer[0] == A_TR) { state = A_RCV; }
            else { state = OTHER_RCV; }
            break;

        case A_RCV:
            if(buffer[0] == FLAG) { state = FLAG_RCV; }
            else if(buffer[0] == SET) { state = C_RCV; }
            else { state = OTHER_RCV; }
            break;

        case C_RCV:
            if(buffer[0] == FLAG) { state = FLAG_RCV; }
            else if(buffer[0] == (A_TR ^ SET)) { state = BCC_RCV; }
            else { state = OTHER_RCV; }
            break;

        case BCC_RCV:
            if(buffer[0] == FLAG) { stop = TRUE; }
            else { state = OTHER_RCV; }
            break;
      }

    }

    write(fd, ua, 5);

    return 0;
}

int sendDisconectMessage(int fd){
    uint8_t disc[5] = {FLAG, A_TR, DISC, A_TR ^ DISC, FLAG};
    uint8_t ua[5] = {FLAG, A_TR, UA, A_TR ^ UA, FLAG};

    uint8_t buffer[255];
    int state;
    timeout_count = 0;
    flag = 0;

    (void) signal(SIGALRM, timeout_handler);

    while(timeout_count < max_timeouts && flag < 2){
```

```
        flag = 0;
        alarm(timeout);

        write(fd, disc, 5);

        state = OTHER_RCV;

        while (flag == 0) {

            read(fd, buffer, 1);

            switch(state){
                case OTHER_RCV:
                    if(buffer[0] == FLAG) { state = FLAG_RCV; }
                    break;

                case FLAG_RCV:
                    if(buffer[0] == FLAG) { state = FLAG_RCV; }
                    else if(buffer[0] == A_RE) { state = A_RCV; }
                    else { state = OTHER_RCV; }
                    break;

                case A_RCV:
                    if(buffer[0] == FLAG) { state = FLAG_RCV; }
                    else if(buffer[0] == DISC) { state = C_RCV; }
                    else { state = OTHER_RCV; }
                    break;

                case C_RCV:
                    if(buffer[0] == FLAG) { state = FLAG_RCV; }
                         else if(buffer[0] == (A_RE ^ DISC)) { state =
BCC_RCV; }
                    else { state = OTHER_RCV; }
                    break;

                case BCC_RCV:
                    if(buffer[0] == FLAG) {
                        alarm(0);
                        flag = 2;
                    }
                    else { state = OTHER_RCV; }
                    break;
            }
```

```
        }
    }

    if(flag == 2){
        write(fd, ua, 5);
    }

    else{
        perror("Too many timeouts ocurred, couldn't send acnowledgement
message on close");
        return -1;
    }

    return 0;
}

int sendDisconectAnswer(int fd){
    uint8_t disc[5] = {FLAG, A_RE, DISC, A_RE ^ DISC, FLAG};

    uint8_t buffer[255];
    int state = OTHER_RCV;

    write(fd, disc, 5);
    flag = 0;

    (void) signal(SIGALRM, timeout_handler);
    alarm(timeout);

    while(flag == 0){
        read(fd, &buffer[0], 1);

        switch(state){
            case OTHER_RCV:
                if(buffer[0] == FLAG) { state = FLAG_RCV; }
                break;

            case FLAG_RCV:
                if(buffer[0] == FLAG) { state = FLAG_RCV; }
                else if(buffer[0] == A_TR) { state = A_RCV; }
                else { state = OTHER_RCV; }
                break;

            case A_RCV:
```

```
                if(buffer[0] == FLAG) { state = FLAG_RCV; }
                else if(buffer[0] == UA) { state = C_RCV; }
                else { state = OTHER_RCV; }
                break;

            case C_RCV:
                if(buffer[0] == FLAG) { state = FLAG_RCV; }
                else if(buffer[0] == (A_TR ^ UA)) { state = BCC_RCV; }
                else { state = OTHER_RCV; }
                break;

            case BCC_RCV:
                if(buffer[0] == FLAG) {
                    alarm(0);
                    flag = 2;
                }
                else { state = OTHER_RCV; }
                break;
        }
    }

    if(flag == 1){
        perror("UA acnowledgement not received");
        return -1;
    }

    return 0;
}

int sendInfoFrame(int fd, int frame_length, uint8_t* buffer){

    int bytes_writen;
    int state;
    uint8_t ack;
    timeout_count = 0;
    flag = 0;

    (void) signal(SIGALRM, timeout_handler);

    while(timeout_count < max_timeouts && flag < 2){
        flag = 0;
        alarm(timeout);
```

```
        bytes_writen = write(fd, frame, frame_length);

        state = OTHER_RCV;

        while (flag == 0)
        {
            read(fd, buffer, 1);

            switch(state){
                case OTHER_RCV:
                    if(buffer[0] == FLAG) {
                        state = FLAG_RCV;
                        }
                    break;

                case FLAG_RCV:
                    if(buffer[0] == FLAG) { state = FLAG_RCV; }
                    else if(buffer[0] == A_RE) {
                        state = A_RCV;
                    }
                    else { state = OTHER_RCV; }
                    break;

                case A_RCV:
                    if(buffer[0] == FLAG) { state = FLAG_RCV; }
                    else if(checkAcknowledgement(buffer[0]) == 1) {
                        ack = buffer[0];
                        state = C_RCV;
                    }
                    else { state = OTHER_RCV; }
                    break;

                case C_RCV:
                    if(buffer[0] == FLAG) { state = FLAG_RCV; }
                    else if(buffer[0] == (A_RE ^ ack)) {
                        state = BCC_RCV;
                    }
                    else { state = OTHER_RCV; }
                    break;

                case BCC_RCV:
                    if(buffer[0] == FLAG) {
                        alarm(0);
```

```
                        flag = 2;
                    }
                    else { state = OTHER_RCV; }
                    break;
            }
        }

        if(flag == 2){
            flipSequenceNumber();
        }

            /* Useless code because it's never going to be entered, however, there's an error with
         / retransmissions that I wasn't able to fix in time, because I only recently realized what was
        / going wrong
        else if(ack == REJ_0 || ack == REJ_1){
             //If a REJ acknowledgement is received, flag is reset to 0 so there's a retry without a timeout
            printf("REJ received\n");
            flag = 0;
        }
        */
    }

    if(timeout_count >= max_timeouts){
              perror("Too many timeouts ocurred, didn't receive an acknowledgement");
        return -2;
    }

    //Here for debug
    else{
        //printf("Received response\n");
    }

    return bytes_writen;
}

int readFrame(int fd, uint8_t* buffer){
    uint8_t control_field;
    int bytes_read;
    int state = OTHER_RCV;
```

```
int stop = FALSE;

while (stop == FALSE) {
  read(fd, &buffer[0], 1);

  switch(state){
    case OTHER_RCV:
        if(buffer[0] == FLAG) {
            bytes_read = 1;
            state = FLAG_RCV;
        }
        break;

    case FLAG_RCV:
        if(buffer[0] == FLAG) { state = FLAG_RCV; }
        else if(buffer[0] == A_TR) {
            bytes_read++;
            state = A_RCV;
        }
        else { state = OTHER_RCV; }
        break;

    case A_RCV:
        if(buffer[0] == FLAG) { state = FLAG_RCV; }
        else if(checkControlField(buffer[0]) == 1) {
            bytes_read++;
            control_field = buffer[0];
            state = C_RCV;
        }
        else { state = OTHER_RCV; }
        break;

    case C_RCV:
        if(buffer[0] == FLAG) { state = FLAG_RCV; }
        else if(buffer[0] == (A_TR ^ control_field)) {
            bytes_read++;
            state = BCC_RCV;
        }
        else { state = OTHER_RCV; }
        break;

    case BCC_RCV:
        if(buffer[0] == FLAG) {
```

```
                stop = TRUE;
                //printf("packet fully received\n");
            }
            else { bytes_read++; }
            break;
        }

        frame[bytes_read - 1] = buffer[0];
    }

          if(control_field   ==   CTR_I_FRAME_1   ||   control_field   ==
CTR_I_FRAME_0){
        //printf("Control entered\n");
        uint8_t answer[5];
        byteDestuffing(&bytes_read);
        //printf("Destuffing done\n");

        uint8_t bcc2 = frame[bytes_read - 2];

        answer[0] = FLAG;
        answer[1] = A_RE;
        answer[4] = FLAG;

        if(checkBcc2(bcc2, bytes_read - FRAME_INFO_SIZE) == 1){
            //printf("Sending REJ\n");
            if(sequence_number == 1){
                answer[2] = REJ_1;
                answer[3] = A_RE ^ REJ_1;
            }
            else if (sequence_number == 0){
                answer[2] = REJ_0;
                answer[3] = A_RE ^ REJ_0;
            }

            write(fd, answer, 5);
            printf("REJ sent\n");

            return BCC2_ERROR;
        }

        else{
            if(sequence_number == 1){
                answer[2] = RR_1;
```

```
                answer[3] = A_RE ^ RR_1;
            }
            else if(sequence_number == 0){
                answer[2] = RR_0;
                answer[3] = A_RE ^ RR_0;
            }

            write(fd, answer, 5);
            //printf("Wrote RR\n");

                memcpy(buffer, &frame[INITIAL_FRAME_BITS], bytes_read -
FRAME_INFO_SIZE);

            flipSequenceNumber();
        }
    }

    else if(control_field == SET){
        uint8_t ua[5] = {FLAG, A_RE, UA, A_RE ^ UA, FLAG};

        write(fd, ua, 5);
    }

    else if(control_field == DISC){

        if(sendDisconectAnswer(fd) < 0){
            bytes_read = DISC_ERROR;
        }
        else{
            bytes_read = DISC_RETURN;
        }
    }

    return bytes_read;
}

int llopen(int port, int mode){
    int fd;

    fd = setupPort(port);

    if(fd < 0){
        return -1;
```

```
    }

    if(mode == TRANSMITER){
        if(setupTransmiter(fd) < 0) { return -1; }
    }

    else if(mode == RECEIVER){
        if(setupReceiver(fd) < 0) { return -1; }
    }

    else{
        perror("Invalid mode");
        return -1;
    }

    //printf("Port sucessfully opened\n");

    return fd;
}

int llwrite(int fd, uint8_t* buffer, int msg_length){
    //printf("llwrite entered\n");

    if(fcntl(fd, F_GETFD) < 0){
        perror("Invalid fd");
        return -1;
    }

    int frame_length = msg_length + FRAME_INFO_SIZE;
    uint8_t bcc2;

    frame[0] = FLAG;
    frame[1] = A_TR;

    if(sequence_number == 1){
        frame[2] = CTR_I_FRAME_1;
        frame[3] = A_TR ^ CTR_I_FRAME_1;
    }

    else if(sequence_number == 0){
        frame[2] = CTR_I_FRAME_0;
        frame[3] = A_TR ^ CTR_I_FRAME_0;
    }
```

```
    else{
        perror("r has an erroneous value, it should only take values of
1 or 0");
    }

    for (int i = 0; i < msg_length; i++)
    {
        frame[i + INITIAL_FRAME_BITS] = buffer[i];
        if(i == 0){
            bcc2 = buffer[i];
        }
        else{
            bcc2 ^= buffer[i];
        }
    }

    frame[frame_length - 2] = bcc2;
    frame[frame_length - 1] = FLAG;

    byteStuffing(&frame_length);

    //printf("Bytes stuffed\n");

    int bytes_writen = sendInfoFrame(fd, frame_length, buffer);

    return bytes_writen;
}

int llread(int fd, uint8_t* buffer){
    int frame_length = 0;

    if(fcntl(fd, F_GETFD) < 0){
        perror("Invalid fd");
        return -1;
    }

    frame_length = readFrame(fd, buffer);

    return frame_length;
}

int llclose(int fd){
```

```
    if(fcntl(fd, F_GETFD) < 0){
        perror("Invalid fd");
        return -1;
    }

    if(mode == 0){
        if(sendDisconectMessage(fd) < 0){
            return -1;
        }
    }

    if(tcsetattr(fd, TCSANOW, &oldtio) == -1) {
        perror("Error while restoring port settings");
        return -1;
    }

    if(close(fd) != 0){
        perror("Error while closing port");
        return -1;
    }

    return 1;
}
```

link_layer.h

```
#ifndef __LINK_LAYER_H__
#define __LINK_LAYER_H__

void timeout_handler();

void flipSequenceNumber();

int setupPort(int port);

int checkAcknowledgement(uint8_t ack);

int checkControlField(uint8_t c);

int checkBcc2(uint8_t bcc2, int msgLength);
```

```
void byteStuffing(int *length);

void byteDestuffing(int *length);

int setupTransmiter(int fd);

int setupReceiver(int fd);

int sendDisconectMessage(int fd);

int sendDisconectAnswer(int fd);

int sendInfoFrame(int fd, int frameLength, uint8_t* buffer);

int readFrame(int fd, uint8_t* buffer);

int llopen(int port, int mode);

int llwrite(int fd, uint8_t* buffer, int length);

int llread(int fd, uint8_t* buffer);

int llclose(int fd);

#endif
```

defs.h

```
#ifndef __DEFS_H__
#define __DEFS_H__

typedef unsigned char uint8_t;

#define FALSE 0
#define TRUE 1

#define DATA_PACKET 1
#define START_PACKET 2
#define END_PACKET 3
#define SIZE_PARAM 0
```

```
#define NAME_PARAM 1

#define CTR_I_FRAME_0 0x00
#define CTR_I_FRAME_1 0x40
#define SET 0x03
#define DISC 0x0B
#define UA 0x07
#define RR_0 0x05
#define RR_1 0x85
#define REJ_0 0x01
#define REJ_1 0x81

#define FLAG 0x7E
#define A_TR 0x03
#define A_RE 0x01

#define OTHER_RCV 0
#define FLAG_RCV 1
#define A_RCV 2
#define C_RCV 3
#define BCC_RCV 4

#define ESCAPE 0x7D
#define XOR_BYTE 0x20

#define FRAME_INFO_SIZE 6
#define INITIAL_FRAME_BITS 4
#define FINAL_FRAME_BITS 2
#define MAX_PACKET_SIZE 64000

#define DISC_RETURN 4
#define DISC_ERROR 3
#define BCC2_ERROR 2

#define TRANSMITER 0
#define RECEIVER 1

#define DEFAULT_BAUD B9600

#endif
```