Faculdade de Engenharia da Universidade do Porto
Faculdade de Ciências da Universidade do Porto

Licenciatura em Engenharia Informática e Computação

Redes de Computadores - L.EIC025

novembro de 2023

Primeiro Trabalho Laboratorial

Relatório

Turma 3LEIC05

Pedro de Almeida Lima - up202108806@up.pt
Pedro Simão Januário Vieira - up202108768@up.pt

Índice

Sumário

O trabalho a que o presente relatório diz respeito foi realizado no âmbito da Unidade Curricular "Redes de Computadores" (L.EIC025), do 1º semestre do 3º ano curricular da Licenciatura em Engenharia Informática e Computação, da Universidade do Porto. O referido trabalho consubstanciou a aplicação prática do que tem vindo a ser abordado nas aulas teóricas da Unidade Curricular.

1. Introdução

O presente relatório versa sobre o primeiro trabalho prático da referida Unidade Curricular, que teve como objetivo o desenvolvimento de um programa para transmissão de um ficheiro entre uma máquina emissora e uma recetora, recorrendo a uma porta-série. Este documento detalha a implementação do processo e a sua validação.

O relatório encontra-se dividido nas seguintes secções, nas quais se pode encontrar:

- Arquitetura - blocos funcionais e interfaces do programa;
- Estrutura do código - estruturas de dados, funções e sua relação com a arquitetura;
- Casos de uso principais - sequência de chamada de funções do ponto de vista do emissor e do recetor;
- Protocolo de ligação lógica - aspetos funcionais e estratégia da sua implementação;
- Protocolo de aplicação - aspetos funcionais e estratégia da sua implementação;
- Validação - testagem da eficácia da abordagem escolhida;
- Eficiência do protocolo de ligação de dados - caracterização estatística do protocolo;
- Conclusões - aprendizagens e ilações retiradas sobre o trabalho desenvolvido.

2. Arquitetura

O programa está organizado em dois módulos diferentes.

O módulo de mais alto nível, o de aplicação, encarrega-se do tratamento do ficheiro a transmitir, seja a sua divisão em pacotes e preparação dos mesmos, do lado do emissor, ou o processamento dos pacotes e escrita dos mesmos no ficheiro recebido, do lado do recetor. Possui apenas um método, que constitui a interface com a função primária de execução de um programa na linguagem C (`int main()`). Dela recebe os parâmetros relevantes para levar a cabo a transmissão. A este módulo dizem respeito os ficheiros `include/application_layer.h` e `src/application_layer.c`.

Por outro lado, o módulo de ligação de dados, de mais baixo nível, é responsável pela efetiva comunicação entre as máquinas, desde a configuração da ligação entre as mesmas, por porta-série, passando pelo envio e receção da informação solicitada pelo módulo acima, até ao fecho da ligação entre o emissor e o recetor. Nele são implementados mecanismos de controlo que, ao longo de toda a ação do módulo, asseguram a transmissão de informação sem erros. A interface com o módulo de aplicação reside em quatro funções: `llopen()`, invocada em ambas as máquinas para estabelecerem ligação entre si; `llwrite()`, chamada pelo módulo de aplicação do emissor para enviar conteúdo do ficheiro a transmitir; `llread()`, usada, analogamente, no recetor para rececionar conteúdo enviado; e `llclose()`, que, tal como a primeira função, é invocada em ambas as máquinas para encerrar a ligação entre elas. A este módulo dizem respeito os ficheiros `include/link_layer.h` e `src/link_layer.c`.

A interface com o utilizador é feita através do `Makefile` no diretório de raiz do projeto. Ajustando-o, é possível definir a porta-série a usar em cada máquina e o ficheiro a transmitir, bem como o nome que o recetor lhe dará.

É de notar que o ficheiro cujo nome é introduzido tem de estar presente no diretório raiz do projeto. Para executar o programa, é necessário definir o *working directory* de ambas as máquinas para o diretório raiz do projeto. De seguida, há que executar `make run_rx` no recetor e `make run_tx` no transmissor (por esta ordem) para dar início à comunicação. O programa fornece continuamente *feedback* textual sobre o progresso da transmissão.

3. Estrutura do código

Camada de ligação de dados

Fez-se uso de cinco estruturas de dados adicionais, sendo as duas primeiras disponibilizadas no *template* proposto no Moodle e as restantes criadas por nós:

```C/C++
// Ilustra o papel que a máquina desepenha na transmissão (emissor/recetor)
typedef enum {LlTx,LlRx} LinkLayerRole;
// Contém parâmetros fundamentais sobre a transmissão
typedef struct {char serialPort[50];LinkLayerRole role;int baudRate;int nRetransmissions;int timeout;} LinkLayer;
// Reflete o tipo de trama processada: de informação, estabelecimento de ligação, encerramento de ligação e de respota (controlo de erros)
typedef enum {INFO, SET, DISC,UA} type_of_frame;
// Encapsula informações sobre uma dada trama
typedef struct {type_of_frame type;unsigned char number; unsigned int size;} frame_info;
// Estados percorridos por llopen(), llwrite() e llclose(), ao validar as tramas de resposta, e por llread() ao validar as tramas de informação
typedef enum {START,FLAG_RCV,A_RCV,C_RCV,C_SET_RCV,C_UA_RCV,C_DISC_RCV,C_INFO_RCV,READING_DATA,ESC_OK,BCC2_OK,ESC_BCC2,FLAG_ESC_BCC2,BCC_OK,STOP} ReceiveState;
```

Foram implementadas as seguintes funções:

```C/C++
// Invocada em ambos os lados pelo módulo da camada de aplicação para estabelecer ligação entre as máquinas
int llopen(LinkLayer connectionParameters);
// Aplicação do emissor usa para enviar um pacote
int llwrite(const unsigned char *buf, int bufSize);
// Aplicação do recetor usa para receber um pacote
int llread(unsigned char *packet);
// Ambos lados chamam para terminar a ligação entre as máquinas
int llclose(int showStatistics);
// Usada por llread() para gerir a receção de uma trama de informação e por llclose() para as tramas envolvidas no fecho da ligação, retornando informação sobre a sua validez
frame_info check_frame(int fd, unsigned char* buffer, unsigned char last_received_frame);
// Lida com os disparos do alarme do sistema
void alarmHandler(int signal);
```

Camada de aplicação

A única função do módulo recorre à estrutura de dados `LinkLayer`, definida pelo módulo homónimo, para encapsular os parâmetros da ligação a estabelecer, e concretiza todo o propósito da camada de aplicação.

```C/C++
void applicationLayer(const char *serialPort, const char *role, int baudRate,
int nTries, int timeout, const char *filename);
```

4. Casos de uso principais

Pressupõe-se a execução paralela do programa em ambas as máquinas. É importante salientar que é a função única do módulo de aplicação quem controla o fluxo da transmissão, i.e. chama as funções de interface do módulo de ligação de dados à medida das necessidades de cada fase da transmissão. Numa transmissão completa, as sequências de chamada de funções, em cada máquina, são as seguintes:

Transmissor

1. `llopen()` envia para o recetor uma trama de controlo para estabelecimento da ligação e espera por confirmação;
2. `llwrite()` envia trama contendo um primeiro pacote com informações sobre o ficheiro e é ciclicamente envia os pacotes com os conteúdos do ficheiro e novamente trama de pacote de controlo com informações sobre o ficheiro, esperando sempre por confirmação para controlo de erros;
3. `llclose()` envia trama de encerramento de ligação;
4. Resposta é validada por `check_frame()` e programa termina.

Recetor

1. `llopen()` recebe e responde a pedido de abertura de ligação do transmissor;
2. `llread()` é ciclicamente chamada para receber tramas de informação e responder às mesmas, invocando `check_frame()` para validar as tramas recebidas;
3. A camada de aplicação recebe e escreve no ficheiro de destino a informação contida nos pacotes extraídos das tramas recebidas por `llread()`;
4. Ao detetar pacote que indica o fim do ficheiro, `applicationLayer()` invoca `llclose()`, que chama `check_frame()` para validar a trama envolvida.

5. Protocolo de ligação lógica

A camada de ligação de dados configura a ligação entre as duas máquinas, transmite a informação solicitada pela camada de aplicação, interagindo diretamente com a porta-série (camada física), prevenindo e lidando com erros na transmissão, recorrendo ao protocolo *Stop & Wait* para o efeito.

Genericamente, o mecanismo *Stop & Wait* pode ser entendido como um protocolo de pergunta e resposta: para cada trama enviada pelo emissor, espera-se uma confirmação, da parte do recetor, de que recebeu a trama sem erros (ou um pedido de retransmissão). No caso de não obter uma resposta dentro de uma janela temporal, o emissor volta a tentar transmitir a trama, até um número limite de tentativas.

Ainda, um aspeto geral a considerar é que a escrita (envio) de tramas na porta-série é feito "de uma vez", i.e. escrevendo toda a trama com uma só chamada a `write()`, através do *file descriptor* referente à porta-série. Tal procedimento acontece nas duas máquinas:

```
C/C++
if (write(fd, set_frame, 5) < 0) return -1;
```

(Exemplo de envio de uma trama `set_frame`, com 5 bytes.)

Já a leitura de informação da porta-série é feita byte a byte, para que as máquinas de estados das várias funções do módulo (explicadas em detalhe mais adiante) processem as tramas passo a passo para controlar erros.

`llopen()` configura a ligação entre as duas máquinas e realiza um *handshake* inicial para garantir a correção da ligação. Para isso, o emissor envia uma trama de supervisão (trama SET) e espera por uma confirmação (UA). Para processar a resposta, uma máquina de estados percorre a trama UA byte a byte, verificando a correção e consistência do conteúdo dos cabeçalhos da trama. No caso de não obter qualquer resposta num tempo predefinido, recorrendo ao alarme do sistema, o emissor volta a enviar a trama SET, decrementando um número predeterminado de tentativas até desistir. Já do lado do recetor, a trama SET recebida é processada do mesmo modo, ato após o qual é enviada a trama UA.

`llwrite()` é invocada exclusivamente no lado do emissor. Recebe da camada de aplicação o pacote a transmitir e constrói os cabeçalhos que acompanharão o conteúdo na trama, procedendo, depois, ao *byte stuffing* da mesma, "mascarando" ocorrências da *flag* de início e término da trama no campo de informação e no campo BCC2. De seguida, procede ao envio da trama para o recetor, esperando por uma resposta do mesmo, do modo explicitado no parágrafo anterior, para `llopen()`. No caso de uma resposta negativa (na situação de o recetor encontrar erros na trama), repete-se o envio, não sendo esta nova tentativa contabilizada para o limite predefinido.

```
C/C++
while (tries > 0) {
       if (read(fd, &byte, 1)) {
       switch (state) {
       case START:
              if (byte == F) state = FLAG_RCV; break;
       case FLAG_RCV:
              if (byte == A_Tx) state = A_RCV;
              else if (byte == F) continue;
              else state = START; break;
                                    ...
       if (state == STOP) break;
       else if (alarmTriggered == TRUE) {
              tries--;   printf("llwrite():   Alarm   triggered,   %i   tries
remaining.\n", tries);
              alarmTriggered = FALSE;
              (void) signal(SIGALRM, alarmHandler);
              alarm(timeout);
              if (write(fd, newFrame, newFrameSize) < 0) return -1;}
```

(Exemplo: em cima, excerto da máquina de estados que processa byte a byte a trama de resposta do recetor. Em baixo, procedimento no caso de não se ter obtido resposta dentro do tempo-limite.)

`llread()` é invocada do lado do recetor. Está encarregue, através de `check_frame()`, de avaliar a correção e consistência das tramas de informação recebidas e de, consoante esses aspetos, dar respostas ao emissor, respondendo RR se a trama for recebida corretamente ou se for duplicada ou REJ se for detetado algum erro na trama. Além disso, procede ao *byte destuffing* e à remoção de cabeçalhos que apenas interessam à camada de ligação de dados e devolve à camada de aplicação, através de um dos argumentos com que foi chamada, o pacote recebido.

`llclose()`, após a transmissão dos conteúdos do ficheiro, é invocada por ambas as máquinas com o propósito de efetuar um *handshake* final e repor a configuração das portas-série. Do lado do transmissor, envia a trama DISC para a porta-série e, no caso particular de fecho de ligação, espera não por uma resposta de confirmação, mas sim por uma trama DISC, validada recorrendo a `check_frame()`, à qual responde com uma trama UA própria. Já no recetor, espera pela trama DISC do transmissor, valida-a com a função auxiliar habitual e, replicando o mecanismo *Stop & Wait*, transmite uma trama DISC sua e espera pela resposta do transmissor; esperando um tempo predefinido por uma resposta, findo o qual reenvia DISC e volta a esperar por uma resposta, até um número limite de tentativas também predeterminado.

Uma importante função auxiliar usada pelo módulo é a tão referida `check_frame()`, que implementa uma máquina de estados preparada para processar e validar tramas de supervisão (SET, DISC e UA) e de informação (INFO).

6. Protocolo de aplicação

O módulo de aplicação do constitui a camada de mais alto nível do nosso programa. Controla o fluxo da transmissão e receção dos conteúdos do ficheiro, invocando continuamente as funções necessárias da camada inferior. Como já referido, assenta num único método homónimo, `applicationLayer()`, que distingue o trabalho a realizar consoante o papel da máquina em que está a ser executado:

```C/C++
void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename) {
    if (strcmp(role, "tx") == 0) {...}
    else if (strcmp(role, "rx" == 0)) {...} ... }
```

(No excerto, pode verificar-se que o papel que a máquina desempenhará é especificado no argumento `role` e que se distingue o ramo a seguir recorrendo a `strcmp()`.)

Existem 3 tipos de pacotes na camada: de começo, de informação e de fim, distinguidos por um cabeçalho inicial de um byte.

Em ambos os casos, começa por se abrir o ficheiro a ler/a escrever e, no caso do transmissor, segue-se a obtenção do tamanho do ficheiro, que será transmitido no primeiro pacote a enviar. Depois, em ambas as máquinas, constrói-se uma estrutura `LinkLayer`, que encapsula os parâmetros da ligação e que é passada a `llopen()`.

Os pacotes de informação trocados (de modo abstrato) entre as camadas de aplicação de ambos os lados têm até um número máximo de bytes (`#define MAX_PAYLOAD_SIZE XXX`, em `include/link_layer.h`). Desses bytes, o primeiro indica que se trata de um pacote de informação; os dois seguintes, o número de bytes de dados a ser enviados; os restantes, a porção de conteúdo do ficheiro. Então, após invocar `llwrite()` para enviar um primeiro pacote de início, contendo o tamanho total do ficheiro, o emissor, ciclicamente, obtém uma porção de dados do ficheiro a enviar, tendo em conta o tamanho máximo permitido, encapsula-a num pacote de informação e

envia-o por `llwrite()`, culminando no envio de um pacote de fim, que contém também o tamanho do ficheiro. Já o recetor invoca ciclicamente `llread()` e, tendo em conta o especificado no segundo e terceiro bytes dos pacotes de informação, escreve as porções do ficheiro recebidas no destino e vai acumulando o número de bytes do ficheiro recebido. Assim procede até detetar a receção de um pacote de fim, fase em que confronta o tamanho especificado neste pacote com o tamanho acumulado ao longo da receção. Por último, ambas as máquinas libertam a memória dinamicamente alocada, fecham os ficheiros que abriram e invocam `llclose()`.

7. Validação

A validação do produto desenvolvido residiu, em primeiro lugar, na passagem da sua execução para o ambiente dos computadores de laboratório, ligados por uma porta-série real, em vez de se fazer transmitir os dados por uma ligação virtual simulada. De seguida, experimentámos a solução para transmitir ficheiros com nome, conteúdo e tamanho diferentes dos da imagem de exemplo. Introduzimos ruído na ligação física entre as máquinas, por curto-circuito na mesma e desligamos os cabos-série intermitentemente. Por último, também para caracterização estatística, introduzimos erros aleatórios em tramas recebidas pelo recetor (virtual), para que houvesse rejeição de tramas e consequente pedido de retransmissão.

Em todos os cenários de perturbação testados, o protocolo comportou-se como esperado. É importante ressalvar que o dito comportamento esperado, nomeadamente nos casos de interrupção da ligação física e de introdução de ruído por curto-circuito, passa por, em casos extremos, aborto da transmissão por parte do transmissor, por obra do tempo e número limite de tentativas de retransmissão de tramas em caso de ausência de resposta.

8. Eficiência do protocolo de ligação de dados

Para testar a eficiência do nosso protocolo de ligação de dados, variou-se artificialmente o *bit error rate* (BER), o tempo de propagação (T_prop), a capacidade de ligação (C) e o tamanho das tramas (I). Os tempos apresentados são o resultado da média de 3 medições.

Variação do Frame Error Rate (FER) e do tempo de propagação (T_prop)

Variou-se o bit error rate, BER, (variando assim também o Frame Error Rate, FER) e o tempo de propagação, T_prop, (variando assim também o a = T_prop/T_f) de modo a obter S(FER, a).

Para variar o Bit Error Rate, definiu-se uma macro `BER`, e após a leitura de cada byte, chama-se uma função `introduce_error()` que tem uma chance de `BER` de introduzir um erro em cada bit lido:

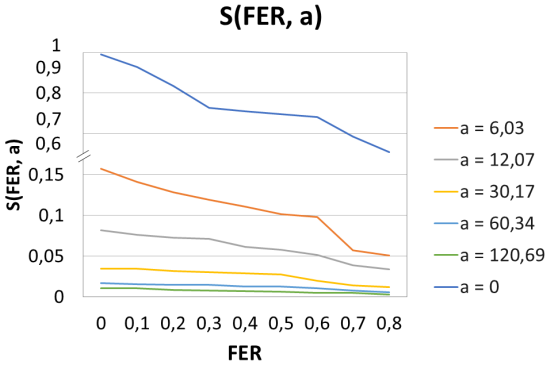
```C/C++
#define BER 0
#define INT_BER RAND_MAX * BER

unsigned char introduce_error(unsigned char input){
  if(BER == 0) return input;
  unsigned char error_mask = 0;
  for(int i = 0; i < 8; i++){
    unsigned int r = rand();
    if(r < INT_BER)  error_mask |= BIT(i); }
  return input ^ error_mask; }
```

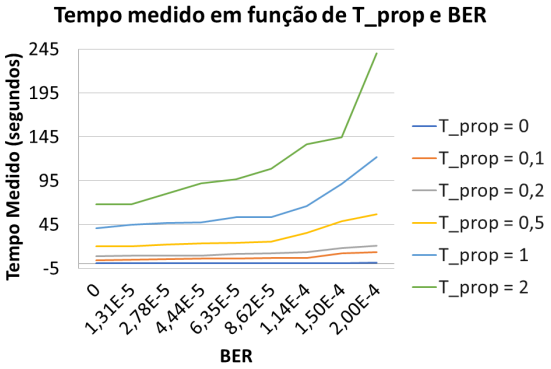
Para variar o tempo de propagação, definiu-se uma macro `T_prop`. Antes do envio de cada trama, chama-se a função `mysleep(T_prop)`, que retarda o programa por `T_prop` segundos.

```C/C++
void my_sleep(double sec){
    struct timespec ts;
    ts.tv_sec = (int) sec;
    ts.tv_nsec = (sec - ((int) sec)) * 1000000000;
    nanosleep(&ts,NULL); }
```

S(FER,a)	0	6,03	12,07	30,17	60,34	120,69
0	0,99	0,15	0,082	0,034	0,017	0,0102
10%	0,94	0,14	0,076	0,035	0,016	0,0102
20%	0,86	0,12	0,073	0,031	0,0149	0,0087
30%	0,77	0,11	0,071	0,030	0,015	0,0076
40%	0,75	0,11	0,061	0,029	0,013	0,0072
50%	0,74	0,10	0,057	0,027	0,013	0,0064
60%	0,73	0,09	0,051	0,020	0,011	0,0051
70%	0,66	0,06	0,038	0,014	0,008	0,0048
80%	0,59	0,05	0,033	0,012	0,006	0,0029

Tempo Medido (s)	0	0,1	0,2	0,5	1	2
0	0,701	4,487	8,506	20,23	40,58	68,07
1,31E-05	0,741	4,937	9,169	20,07	44,52	68,14
2,77E-05	0,807	5,420	9,583	22,11	46,67	80,18
4,44E-05	0,901	5,860	9,779	23,15	47,67	92,24
6,35E-05	0,917	6,273	11,33	24,28	53,49	96,30
8,62E-05	0,933	6,848	12,02	25,63	53,62	108,4
1,14E-04	0,948	7,115	13,58	35,35	65,77	136,5
1,50E-04	1,06	12,17	17,99	48,79	91,18	144,3
2,00E-04	1,18	13,75	20,51	56,85	122,1	240,5





Como era de esperar, concluímos que a eficiência do protocolo é máxima quando o *Frame Error Rate* e o tempo de propagação são mínimos (e consequentemente quando o *Bit Error Rate* e o tempo de propagação também são mínimos, neste caso 0). No projeto, vimos uma influência muito grande do tempo de propagação na eficiência, uma vez que o mecanismo ARQ utilizado é o *Stop & Wait*, que obriga a que o transmissor receba uma resposta do recetor, antes de enviar outra trama. Com tempos de propagação muito grandes, tal torna-se problemático, fazendo com que a eficiência decresça bastante.

Variação da capacidade de ligação(C)

Para variar a capacidade de ligação, variou-se a macro `BAUDRATE`, definida em `main.c`, que é passada como argumento em `applicationLayer()` e em `llopen()`, através dos parâmetros de conexão. Ao correr `llopen()`, este argumento é usado para definir o `newtio.c_cflag`, assim definindo a capacidade de ligação do nosso canal.

A variação da capacidade de ligação não resultou em nenhuma mudança significativa na eficiência do protocolo. Isto deve-se ao facto de a eficiência ser obtida dividindo o débito recebido pela capacidade de ligação. Como o débito recebido é proporcional à capacidade, este valor deve manter-se constante.

Variação do tamanho das tramas de Informação(I)

Ao desenvolver o módulo de Aplicação, utilizou-se como tamanho indicativo de cada pacote o valor na macro `MAX_PAYLOAD_SIZE`, já declarada em `src/link_layer.h` . Assim, como o Link Layer adiciona um header e um footer ao pacote recebido, o tamanho real da trama será `MAX_PAYLOAD_SIZE+5`. Sabendo isto, para alterar o tamanho da trama basta alterar o valor da macro `MAX_PAYLOAD_SIZE`. Para realizar os testes variando o tamanho das tramas de informação, utilizou-se um tempo de propagação de 0,1 segundos e um FER de 20%, pois consideramos que assim se teria uma melhor noção da real influência do tamanho das tramas de informação na eficiência.

MAX_PAYLOAD_SIZE	I (bytes)	Tempo medido (s)	S
10	15	376,349	0,00185036
45	50	62,406	0,01115888
195	200	13,911	0,05005973
995	1000	4,921	0,14151208
4995	5000	3,512	0,19828615
9995	10000	2,323	0,29977656

Com isto concluímos que quanto menores as tramas de informação, pior será a eficiência do nosso protocolo, uma vez que serão necessárias mais viagens para transmitir o mesmo ficheiro, o que resulta num maior tempo total de emissão. Este problema é agravado pelo facto do mecanismo ARQ implementado ser o *Stop & Wait*, que obriga a que para cada trama transmitida seja enviado uma resposta pelo recetor, antes de enviar a trama seguinte, o que aumenta ainda mais o tempo necessário para enviar o ficheiro, porque, como as tramas são mais pequenas e temos mais tramas para enviar, mais tempo será desperdiçado a enviar e receber respostas.

9. Conclusões

O Primeiro Trabalho Laboratorial constituiu um importante momento de aplicação prática dos conceitos abordados nas aulas teóricas da UC, o que, sem dúvida, permitiu interiorizar melhor os mesmos. Tivemos, ainda, oportunidade de conhecer os mecanismos dos sistemas operativos Unix e da linguagem de programação C para interagir com portas-série e as particularidades e limitações dos mesmos.

Anexo I - Código-fonte

Foi seguida a estrutura proposta no Moodle. Assim, os únicos ficheiros por nós intervencionados foram `src/application_layer.c` e `src/link_layer.c`.

applicationlayer.c

```C/C++
// Application layer protocol implementation

#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>

#include "../include/application_layer.h"
#include "../include/link_layer.h"

#define C_DATA 1
#define C_START 2
#define C_END 3

void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename)
{
        if (strcmp(role, "tx") == 0) {
        int fd = open(filename, O_RDONLY);
        if (fd == -1) {
        printf("Error opening \"%s\".\n", filename);
        return;
        }

        struct stat st;
        stat(filename, &st);
        unsigned long size = (unsigned long) st.st_size;

        unsigned char *control_packet = (unsigned char*) malloc (11);
        control_packet[0] = C_START;
        control_packet[1] = 0;
        control_packet[2] = 8;

        unsigned long size_aux = size;
        for (int i = 7; i >= 0; i--) {
        control_packet[i + 3] = (unsigned char) (0xff & size_aux);
        size_aux >>= 8;
        }

        LinkLayer connectionParameters;

        LinkLayerRole role = LlTx;
        strcpy(connectionParameters.serialPort, serialPort);
        connectionParameters.role = role;
        connectionParameters.baudRate = baudRate;
```

```
connectionParameters.nRetransmissions = nTries;
connectionParameters.timeout = timeout;

if (llopen(connectionParameters) == -1) {
printf("Error setting connection.\n");
return;
}
printf("\nTransmition Started\n\n");

if (llwrite(control_packet, 11) == -1) {
printf("Error transmitting information.\n");
return;
}

unsigned long left = size;

unsigned char* data_packet = (unsigned char*) malloc (MAX_PAYLOAD_SIZE);
unsigned int line_size = MAX_PAYLOAD_SIZE - 3;

while (left > 0) {
if (left <= line_size) line_size = left;

data_packet[0] = C_DATA;
data_packet[2] = (unsigned char) (0xff & (line_size));
data_packet[1] = (unsigned char) (0xff & ((line_size) >> 8));

read(fd, data_packet + 3, line_size);

if (llwrite(data_packet, line_size + 3) != line_size + 3) {
printf("Error transmitting information.\n");
return;
}
printf("Answer Received.\n");

left -= line_size;
}

free(data_packet);

control_packet[0] = C_END;
if (llwrite(control_packet, 11) == -1) {
printf("Error transmitting information.\n");
return;
}
free(control_packet);

while (llclose(0) == -1) {
printf("Error closing connection.\n");
return;
}

close(fd);

} else if (strcmp(role, "rx") == 0) {
int fd;
int fd_temp = open(filename, O_WRONLY | O_TRUNC);
if(fd_temp == -1)
```

```
fd = open(filename, O_APPEND | O_CREAT | O_WRONLY);
else {
write(fd_temp, "", 0);
close(fd_temp);
fd = open(filename, O_APPEND | O_WRONLY);
}
if (fd == -1) {
printf("Error opening \"%s\".\n", filename);
return;
}
LinkLayer connectionParameters;

LinkLayerRole role = LlRx;
strcpy(connectionParameters.serialPort, serialPort);
connectionParameters.role = role;
connectionParameters.baudRate = baudRate;
connectionParameters.nRetransmissions = nTries;
connectionParameters.timeout = timeout;

if (llopen(connectionParameters) == -1) {
printf("Error setting connection.\n");
return;
}
printf("\nTransmition Started\n\n");

unsigned char* buffer = (unsigned char*) malloc (MAX_PAYLOAD_SIZE);
if(llread(buffer) == -1) {
printf("Error receiving information.\n");
return;
}
printf("Packet Received\n");

unsigned long size = 0;
for (int i = 0; i < 8; i++) {
size <<= 8;
size += buffer[i + 3];
}

while(1) {
if(llread(buffer) == -1) {
continue;
}
if(buffer[0] != C_DATA) break;
unsigned int current_size = buffer[1];
current_size <<= 8;
current_size += buffer[2];

write(fd, buffer + 3, current_size);
printf("Packet Received \n");
}

if(buffer[0] != C_END) {
printf("Error receiving information(C_END).\n");
return;
}
```

```
        unsigned long new_size = 0;
        for(int i = 0; i < 8; i++){
        new_size <<= 8;
        new_size += buffer[i + 3];
        }
        if(new_size != size) {
        printf("Error receiving information(size).\n");
        return;
        }

        free(buffer);

        while(llclose(0) == -1);
        close(fd);

        }

        printf("\nTransmition Ended Sucessfully\n");
}
```

linklayer.c

```C/C++
// Link layer protocol implementation
#include <fcntl.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <termios.h>
#include <time.h>
#include <unistd.h>

#include "../include/link_layer.h"

typedef enum {
	INFO,
	SET,
	DISC,
	UA,
	DUP,
} type_of_frame;

typedef struct{

	type_of_frame type;

	unsigned char number;
	unsigned int size;

} frame_info;

frame_info check_frame(int fd, unsigned char* buffer, unsigned char last_received_frame);

// MISC
#define BAUDRATE B38400
#define _POSIX_SOURCE 1 // POSIX compliant source

// Common frame fields

#define F 0x7e

#define A 0x03
#define A_Tx 0x03
#define A_Rx 0x01

#define C_0 0x00
#define C_1 0x40

#define C_SET 0x03
#define C_UA 0x07
#define C_RR0 0x05
#define C_RR1 0x85
#define C_REJ0 0x01
```

```
#define C_REJ1 0x81
#define C_DISC 0x0b

#define ESC 0x7d

#define BIT(n) (1 << n)

// Message reception state machine
typedef enum {
        START,
        FLAG_RCV,
        A_RCV,
        C_RCV,
        C_SET_RCV,
        C_UA_RCV,
        C_DISC_RCV,
        C_INFO_RCV,
        READING_DATA,
        ESC_OK,
        BCC2_OK,
        ESC_BCC2,
        FLAG_ESC_BCC2,
        BCC_OK,
        STOP
} ReceiveState;

struct termios oldtio;

LinkLayerRole role;
int nRetransmissions;
int timeout;

int fd;

int alarmTriggered = FALSE;

int info_frame_number = 0;

unsigned char set_frame[5] = {F, A_Tx, C_SET, A_Tx ^ C_SET, F};
unsigned char ua_frame[5] = {F, A_Tx, C_UA, A_Tx ^ C_UA, F};
unsigned char disc_frame[5] = {F, A_Tx, C_DISC, A_Tx ^ C_DISC, F};

unsigned char rr_frame_0[5] = {F, A_Tx, 0x05, A_Tx ^ 0x05, F};
unsigned char rr_frame_1[5] = {F, A_Tx, 0x85, A_Tx ^ 0x85, F};

unsigned char rej_frame_0[5] = {F, A_Tx, 0x01, A_Tx ^ 0x01, F};
unsigned char rej_frame_1[5] = {F, A_Tx, 0x81, A_Tx ^ 0x81, F};

unsigned char mask = ~((unsigned char) BIT(6));

void alarmHandler(int signal) {
        alarmTriggered = TRUE;
}

////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
```

```
int llopen(LinkLayer connectionParameters)
{

        role = connectionParameters.role;
        nRetransmissions = connectionParameters.nRetransmissions;
        timeout = connectionParameters.timeout;
        fd = open(connectionParameters.serialPort, O_RDWR | O_NOCTTY);
        if (fd < 0) return -1;
        if (tcgetattr(fd, &oldtio) == -1) return -1;

        struct termios newtio;
        memset(&newtio, 0, sizeof(newtio));
        newtio.c_cflag = connectionParameters.baudRate | CS8 | CLOCAL | CREAD;
        newtio.c_iflag = IGNPAR;
        newtio.c_oflag = 0;
        newtio.c_cc[VTIME] = 0;
        if(connectionParameters.role == LlTx)
         newtio.c_cc[VMIN] = 0;
        else if(connectionParameters.role == LlRx)
         newtio.c_cc[VMIN] = 1;

        tcflush(fd, TCIOFLUSH);
        if (tcsetattr(fd, TCSANOW, &newtio) == -1) return -1;

        ReceiveState state = START;
        int tries = nRetransmissions;
        unsigned char byte;

        if (connectionParameters.role == LlTx) {
        if (write(fd, set_frame, 5) < 0) return -1;

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

        while (tries > 0) {
        if (read(fd, &byte, 1)) {
        switch (state) {
                case START:
                if (byte == F) state = FLAG_RCV;
                break;
                case FLAG_RCV:
                if (byte == A_Tx) state = A_RCV;
                else if (byte == F) continue;
                else state = START;
                break;
                case A_RCV:
                if (byte == C_UA) state = C_RCV;
                else if (byte == F) state = FLAG_RCV;
                else state = START;
                break;
                case C_RCV:
                if (byte == (A_Tx ^ C_UA)) state = BCC_OK;
                else if (byte == F) state = FLAG_RCV;
                else state = START;
                break;
                case BCC_OK:
```

```
        if (byte == F) {
        state = STOP;
        } else state = START;
        break;
        default: break;
}
}
if (state == STOP) break;
else if (alarmTriggered == TRUE) {
tries--;
printf("llopen(): Alarm triggered, %i tries remaining.\n", tries);

alarmTriggered = FALSE;
(void) signal(SIGALRM, alarmHandler);
alarm(timeout);
if (write(fd, set_frame, 5) < 0) return -1;
}
}
if (tries == 0 && state != STOP) return -1;
} else if (connectionParameters.role == LlRx) {
while (1) {
if (read(fd, &byte, 1)) {
switch (state) {
        case START:
        if (byte == F) state = FLAG_RCV;
        break;
        case FLAG_RCV:
        if (byte == A_Tx) state = A_RCV;
        else if (byte == F) continue;
        else state = START;
        break;
        case A_RCV:
        if (byte == C_SET) state = C_RCV;
        else if (byte == F) state = FLAG_RCV;
        else state = START;
        break;
        case C_RCV:
        if (byte == (A_Tx ^ C_SET)) state = BCC_OK;
        else if (byte == F) state = FLAG_RCV;
        else state = START;
        break;
        case BCC_OK:
        if (byte == F) {
        state = STOP;
        } else state = START;
        break;
        default: break;
}
}
if (state == STOP) break;
}
if (write(fd, ua_frame, 5) < 0) return -1;
}
tries = connectionParameters.nRetransmissions + 1;
```

```
        return 1;
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(const unsigned char *buf, int bufSize)
{
        int oldFrameSize = bufSize + 6;
        unsigned char *oldFrame = (unsigned char*) malloc (oldFrameSize);

        oldFrame[0] = F;
        oldFrame[1] = A_Tx;
        oldFrame[2] = info_frame_number == 0 ? C_0 : C_1;
        oldFrame[3] = A_Tx ^ (info_frame_number == 0 ? C_0 : C_1);
        int i;
        unsigned char BCC2 = 0;
        for (i = 4; i < bufSize + 4; i++) {
        oldFrame[i] = buf[i - 4];
        BCC2 ^= buf[i - 4];
        }
        oldFrame[i] = BCC2;
        i++;
        oldFrame[i] = F;

        unsigned char *newFrame = (unsigned char*) malloc (oldFrameSize * 2);
        for (int j = 0; j < 4; j++) newFrame[j] = oldFrame[j];
        int newFrameSize = 4;
        for (int j = 4; j < oldFrameSize - 1; j++) {
        if (oldFrame[j] == F) {
        newFrame[newFrameSize] = ESC; newFrameSize++;
        newFrame[newFrameSize] = 0x5e; newFrameSize++;
        } else if (oldFrame[j] == ESC) {
        newFrame[newFrameSize] = ESC; newFrameSize++;
        newFrame[newFrameSize] = 0x5d; newFrameSize++;
        } else {
        newFrame[newFrameSize] = oldFrame[j];
        newFrameSize++;
        }
        }
        newFrameSize++;
        newFrame[newFrameSize - 1] = F;
        newFrame = realloc(newFrame, newFrameSize);

        ReceiveState state = START;
        int tries = nRetransmissions;
        unsigned char byte;

        if (write(fd, newFrame, newFrameSize) < 0) return -1;

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

        while (tries > 0) {
        if (read(fd, &byte, 1)) {
        switch (state) {
```

```
        case START:
                if (byte == F) state = FLAG_RCV;
                break;
        case FLAG_RCV:
                if (byte == A_Tx) state = A_RCV;
                else if (byte == F) continue;
                else state = START;
                break;
        case A_RCV:
                if (byte == F) {state = FLAG_RCV; break;}
                if (info_frame_number == 0) {
                if (byte == C_RR1) state = C_RCV;
                else if (byte == C_REJ0) state = START;
                } else if (info_frame_number == 1) {
                if (byte == C_RR0) state = C_RCV;
                else if (byte == C_REJ1) state = START;
                }
                break;
        case C_RCV:
                if (byte == (A_Tx ^ (info_frame_number == 0 ? C_RR1 : C_RR0))) state =
BCC_OK;
                else if (byte == F) state = FLAG_RCV;
                else state = START;
                break;
        case BCC_OK:
                if (byte == F) {
                state = STOP;
                } else state = START;
                break;
        default: break;
        }
        }
        if (state == STOP) {
        break;
        }
        else if (alarmTriggered == TRUE) {
        tries--;
        printf("llwrite(): Alarm triggered, %i tries remaining.\n", tries);

        alarmTriggered = FALSE;
        (void) signal(SIGALRM, alarmHandler);
        alarm(timeout);

        if (write(fd, newFrame, newFrameSize) < 0) return -1;
        } else if (state == START) {
        alarmTriggered = FALSE;
        (void) signal(SIGALRM, alarmHandler);
        alarm(timeout);

        if (write(fd, newFrame, newFrameSize) < 0) return -1;
        }
        }
        if (tries == 0 && state != STOP) return -1;

        info_frame_number = info_frame_number == 0 ? 1 : 0;
```

```
        return bufSize;
}

unsigned char last_received_frame = 1;

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
int llread(unsigned char *packet){
        frame_info info = check_frame(fd, packet, last_received_frame);
        switch (info.type){
        case SET:
        write(fd, ua_frame, 5);
        break;
        case UA:
        break;

        case INFO:
        last_received_frame = info.number;
        if(info.number == 0) write(fd, rr_frame_1, 5);
        else if(info.number == 1) write(fd, rr_frame_0, 5);
        return info.size;
        break;

        case DUP:
         if(info.number == 0) write(fd, rr_frame_1, 5);
        else if(info.number == 1) write(fd, rr_frame_0, 5);
        return -1;
        break;

        default:
        if(info.number == 0) write(fd, rej_frame_0, 5);
        else if (info.number == 1) write(fd, rej_frame_1, 5);

        return -1;
        }
        return 0;
}

int block_info = FALSE;

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics){
        block_info = TRUE;
        if(role == LlTx) {
        write(fd, disc_frame, 5);
        frame_info info = check_frame(fd, NULL, 0);
        if(info.type == DISC) write(fd, ua_frame, 5);
        else return -1;
        }
        else if(role == LlRx) {
        frame_info info = check_frame(fd, NULL, 0);
        if(info.type == DISC){
        write(fd, disc_frame, 5);
```

```
        (void) signal(SIGALRM, alarmHandler);
        alarm(timeout);
        int tries = nRetransmissions;
        while(info.type != UA && tries > 0){
        info = check_frame(fd, NULL, 0);
        if(alarmTriggered){
               write(fd, disc_frame, 5);

               alarmTriggered = FALSE;
               (void) signal(SIGALRM, alarmHandler);
               alarm(timeout);

        }
        }
        if(tries <= 0){
        return -1;
        }
        }
        else return -1;
        }
        if (tcsetattr(fd, TCSANOW, &oldtio) == -1){
        perror("tcsetattr");
        exit(-1);
        }
        close(fd);
        return 1;
}

frame_info   check_frame(int   fd,   unsigned   char*   buffer,   unsigned   char
last_received_frame) {
        ReceiveState state = START;
        frame_info info;
        info.type = -1; info.number = 2; info.size=0;


        unsigned char current_C;

        unsigned char BCC2 = 0x00;

        unsigned char byte_read;
        unsigned int current_buffer_byte = 0;

        unsigned int initial_header_size, info_field_size, final_header_size;

        while(state != STOP){
        if (read(fd, &byte_read, 1) == -1) continue;
        switch(state){
        case START:
        initial_header_size = 0; info_field_size = 0; final_header_size = 0;
        if(byte_read == F) state = FLAG_RCV;
        else state = START;
        break;

        case FLAG_RCV:
        initial_header_size++;
        if(byte_read == A) state = A_RCV;
```

```
else if(byte_read == F) state = FLAG_RCV;
else state = START;
break;

case A_RCV:
initial_header_size++;
if(byte_read == C_SET) state = C_SET_RCV;
else if(byte_read == C_UA) state = C_UA_RCV;
else if(byte_read == C_DISC) state = C_DISC_RCV;
else if(!(byte_read & mask)) { state = C_INFO_RCV; current_C = byte_read; }
else if(byte_read == F) state = FLAG_RCV;
else state = START;
break;

case C_SET_RCV:
initial_header_size++;
info.type = SET;
if(byte_read == (A^C_SET)) state = BCC_OK;
else if(byte_read == F) state = FLAG_RCV;
else state = START;
break;

case C_UA_RCV:
initial_header_size++;
info.type = UA;
if(byte_read == (A^C_UA)) state = BCC_OK;
else if(byte_read == F) state = FLAG_RCV;
else state = START;
break;

case C_DISC_RCV:
initial_header_size++;
info.type = DISC;
if(byte_read == (A^C_DISC)) state = BCC_OK;
else if(byte_read == F) state = FLAG_RCV;
else state = START;
break;

case C_INFO_RCV:
       if(block_info) { info.type = -1; return info; }
initial_header_size++;
info.type = INFO;
info.number = current_C >> 6;
if(last_received_frame == info.number &&
       byte_read == (A^current_C)) { info.type = DUP; return info; }
if(byte_read == (A^current_C)) state = READING_DATA;
else if(byte_read == F) state = FLAG_RCV;
else state = START;
break;

case READING_DATA:
if(byte_read == ESC && (BCC2 == 0x7e || BCC2 == 0x7d)) state = ESC_BCC2;
else if(byte_read == ESC) state = ESC_OK;
else if(byte_read == BCC2) state = BCC2_OK;
else if(byte_read == F){ info.type = -1; return info; }
else {
```

```
                info_field_size++;
                BCC2 ^= byte_read;
                buffer[current_buffer_byte++] = byte_read;
                state = READING_DATA;
        }
        break;

        case ESC_OK:
        if(byte_read == 0x5d) {
                BCC2 ^= 0x7d;
                info_field_size++;
                buffer[current_buffer_byte++] = 0x7d;
                state = READING_DATA;
        } else if(byte_read == 0x5e) {
                BCC2 ^= 0x7e;
                info_field_size++;
                buffer[current_buffer_byte++] = 0x7e;
                state = READING_DATA;
        } else { info.type = -1; return info; }
        break;

        case BCC2_OK:
        final_header_size++;
        if(byte_read == F) state = STOP;
        else {
                buffer[current_buffer_byte++] = BCC2;
                BCC2 ^= BCC2;

                if(byte_read == ESC && (BCC2 == 0x7e || BCC2 == 0x7d)) state =
ESC_BCC2;
                else if(byte_read == ESC) state = ESC_OK;
                else if(byte_read == BCC2) state = BCC2_OK;

                else {
                BCC2 ^= byte_read;
                buffer[current_buffer_byte++] = byte_read;
                state = READING_DATA;
                }

        }
        break;

        case ESC_BCC2:
        final_header_size++;
        if(BCC2 == 0x7d && byte_read == 0x5d) state = FLAG_ESC_BCC2;
        else if(BCC2 == 0x7e && byte_read == 0x5e) state = FLAG_ESC_BCC2;
        else { info.type = -1; return info; }
        break;

        case FLAG_ESC_BCC2:
        final_header_size++;
        if(byte_read == F) state = STOP;
        else {
                buffer[current_buffer_byte++] = BCC2;
                BCC2 ^= BCC2;
                state = READING_DATA;
```

```
        }

        case BCC_OK:
        final_header_size++;
        if(byte_read == F) state = STOP;
        else state = START;
        break;

        case STOP:
        break;
        default:
        break;
        }
        }
        info.size = current_buffer_byte;
        return info;
}
```