Trabalho Prático 1
Ligação de Dados

Feito por

Ana Sofia Teixeira

Pedro Lima

Unidade Curricular de Redes de Computadores

November 1, 2022

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Departamento de Engenharia Eletrotécnica e de Computadores

Sumário

Este trabalho foi desenvolvido com o objetivo de implementar e analisar um protocolo de ligação de dados que consiste na transferência de dados entre computadores através de uma porta série.
O trabalho foi concluído com sucesso uma vez que a aplicação desenvolvida estabelece uma ligação entre dois computadores de forma eficiente.

1 Introdução

O presente relatório foi elaborado como parte integrante da unidade curricular de Redes de Computadores, lecionada no âmbito da Licenciatura em Engenharia Informática e Computação e supervisionada pelo Departamento de Engenharia Eletrotécnica e de Computadores.

Pretende-se que as partes integrantes do projeto tenham conhecimento e domínio sobre redes de comunicações, nomeadamente canais de comunicação e controlo da ligação de dados. Com este projeto pretende-se implementar um protocolo de ligação de dados capaz de transferir ficheiros através de uma porta série. O objetivo do Protocolo de Ligação de Dados é fornecer um serviço de comunicação de dados fiável entre dois sistemas ligados por um meio de transmissão.

Com o objetivo de detalhar a componente teórica do projeto, este relatório aborda os seguintes temas:

- Arquitetura do projeto.
- Etrutura do código, APIs utilizadas, principais estruturas de dados, principais funções e sua relação com a arquitetura.
- Casos de uso principais, sequência de chamada de funções.
- Protocolo de Ligação Lógica.
- Protocolo de Aplicação.
- Validação, descrição dos testes efetuados.
- Eficiência do protocolo de ligação de dados.
- Conclusões.

2 Arquitetura

O *software* que implementa o protocolo de ligação de dados está dividido em dois módulos independentes entre si: o módulo que estabelece a ligação de dados e o módulo da aplicação.

O módulo da ligação de dados trata de toda a comunicação com a porta série, isto é, a sua abertura e fecho, bem como a sua leitura e escrita. Além disso, este módulo é também responsável pela criação e processamento de cada trama, tratando da sua delimitação, transparência, proteção e retransmissão em caso de erro.
O módulo da aplicação utiliza o módulo de ligação de dados e é responsável pelo envio e receção de pacotes, quer de controlo quer de informação. Cada pacote é ainda estruturado por este módulo, efetuando o tratamento do seu cabeçalho e definindo a sua numeração.

A independência entre módulos é assegurada pelos seguintes factos:

- No módulo de ligação de dados não é efetuada qualquer tipo de distinção entre pacotes de controlo e de dados, nem é tida em conta a numeração dos pacotes;
- No módulo da aplicação não há conhecimento acerca do método de ligação de dados, ou seja, este módulo desconhece a estruturação das tramas, a sua criação, e o *stuffing/destuffing* a que estão sujeitas. No entanto, tem acesso às funções do módulo de ligação de dados, para o envio e receção de informação.

3 Estrutura do código

A implementação do código foi feita a partir do *template* dado pelo professor, estando assim divido em vários ficheiros. Foram criados ficheiros novos para além dos que foram dados para melhorar a leitura do código e a sua organização.

Nesta secção está presente uma descrição dos diferentes ficheiros e da relação entre eles.

3.1 Headers

O código contém 4 ficheiros *header*.

- ***application_layer.h*** - declara as funções implementadas pelo ficheiro executável *application_layer.c*.
- ***link_layer.h*** - declara as estruturas de dados e funções implementadas pelo ficheiro executável *link_layer.c*.
- ***statemachine.h*** - declara as estruturas de dados e funções implementadas pelo ficheiro executável *statemachine.h*.
- ***macros.h*** - declara as macros usadas pelos diversos ficheiros executáveis. Foi criado principalmente para otimizar a análise visual do código.

3.2 Executáveis

Para além dos 3 ficheiros executáveis dados pelo *template* do professor (*main.c*, *application_layer.c* e *link_layer.c*), foi criado o ficheiro *statemachine.c*.

3.2.1 main.c

Este ficheiro trata do início da execução do programa. Recebe os argumentos necessários para identificar a porta série, tipo de utilizador e informação sobre o ficheiro a ser transmitido/recebido. Chama a *Application Layer* e passa-lhe como argumentos a porta série, o tipo de utilizador (emissor/recetor), a *baudrate* a ser utilizada, o número de tentativas de retransmissão, o tempo para ocorrer um *TIMEOUT* e o nome do ficheiro.

3.2.2 application_layer.c

Este é o segundo ficheiro a ser executado. Recebe como argumentos os parâmetros que o ficheiro *main.c* lhe enviou. Este ficheiro trata da transmissão de dados entre os dois computadores, trata do começo e do fecho da ligação de dados, trata da leitura do ficheiro em pacotes de dados (no caso do emissor) e trata da construção do ficheiro através dos pacotes de dados (no caso do recetor).

3.2.3 link_layer.c

Este ficheiro é o terceiro a ser executado e as suas funções são chamadas através do ficheiro *application_layer.c*. É o ficheiro responsável pelo protocolo de ligação de dados. As suas principais funções (*llopen*, *llwrite*, *llread* e *llclose*) são chamadas através da *Application Layer* e são fundamentais para o bom funcionamento do programa uma vez que fazem o controlo de todo o processo de transmissão. Asseguram que a ligação entre os dois computadores foi bem sucedida com comunicação bidirecional (*llopen*), asseguram a leitura/envio de dados com respostas para saber o estado da transmissão (*llread* e *llwrite*), e asseguram o término da ligação entre os dois computadores também com comunicação bidirecional (*llclose*).

3.2.4 statemachine.c

Este ficheiro contém a *state machine* utilizada ao longo do trabalho. A *state machine* é usada para definir o estado em que a transmissão se encontra. Este ficheiro foi criado com o intuito de melhorar a análise visual do código uma vez que se trata de uma função muito comprida.

3.3 Estruturas de Dados

Para a realização deste trabalho foram usadas 3 estruturas de dados.

3.3.1 *Link Layer Role*

Esta estrutura de dados já se encontrava no *template* do professor que foi usado para este projeto. É usada para definir o tipo de utilizador, isto é, se se trata do emissor ($LlTx$) ou se se trata do recetor ($LlRx$).

3.3.2 *Link Layer*

Esta estrutura de dados também já se encontrava no *template* do professor que foi usado para este projeto. É usada para guardar toda a informação necessária para a execução do protocolo de ligação de dados. Guarda a porta série, o tipo de utilizador (através da estrutura de dados *Link Layer Role*), o valor da *baudrate*, o número de retransmissões e o valor de *timeout*.

3.3.3 *State*

Esta estrutura de dados foi criada para guardar os estados da *state machine*. É usada nos ficheiros *statemachine.c* e *link_layer.c*. Guarda o estado da *state machine*, o campo de endereço, o campo de controlo, o *bcc*, um campo de dados e o tamanho desse campo de dados.

4 Casos de uso principais

Após a compilação do programa, o utilizador deverá passar como argumentos:

- **Porta série** - no formato */dev/ttySxx*;
- **Identificação do utilizador** - identifica se o utilizador que está a usar o programa se trata do emissor ou recetor.
- **Nome do ficehiro** - nome do ficheiro que vai ser enviado ou nome para dar ao ficheiro que vai ser recebido.

De seguida, o programa toma diferentes rumos dependedo do tipo de utilizador.

4.1 Emissor

O programa começa por chamar a *Application Layer* (AL) que chama a função *llopen* da *Link Layer* para dar início à transmissão de dados entre os dois computadores. Dentro da AL é aberto o ficheiro que vai ser transmitido e é descoberto o seu tamanho, assim é possível construir o primeiro pacote de controlo para iniciar a função *llwrite*. De seguida começa o processo de ler o ficheiro com o número de *bytes* escolhido para o tamanho do pacote de dados e enviar para o recetor através de várias chamadas ao *llwrite* até chegar ao final do ficheiro. Ainda no *llwrite*, são construídas tramas de informação através da função *buildDataFrame* que serão enviadas ao recetor. Na função *buildDataFrame* é

chamada também a função *stuff* que faz o *stuffing* dos dados. A cada envio de uma trama de informação, o *llwrite* espera a confirmação (tramas de supervisão) de que o recetor recebeu um pacote de dados correto (respostas *RR*). Caso o emissor receba um campo de controlo *REJ* no *llwrite*, é iniciada uma retransmissão da mesma trama. Quando este processo termina, a AL chama a função *llclose* para terminar a ligação entre os dois computadores.
Para cada chamada de uma função do emissor que precise de uma resposta do recetor, há um limite de 4 segundos de espera que, quando excedido, dá origem a um *timeout*. Ao fim dos 4 segundos, o emissor volta a enviar a mesma trama até obter uma resposta. Este processo pode repetir-se até 3 vezes. Quando ultrapassado esse limite, o emissor termina a execução dessa função e passa para a função *llclose* para tentar terminar a ligação. Se esta não for bem sucedida, o programa termina.

4.2 Recetor

O programa começa por chamar a *Application Layer* (AL) que chama a função *llopen* para dar início à transmissão de dados entre os dois computadores tal como o emissor. Dentro da AL o recetor recebe um pacote de controlo com informação sobre o ficheiro que irá receber. Neste caso, o recetor apenas recebe o tamanho do ficheiro através do campo *TLV*. Ainda na AL, é criado o ficheiro com o nome recebido no argumento no momento da execução do programa e é começada a leitura do ficheiro através da função *llread*. Na função *llread*, o recetor recebe as tramas de informação enviadas pelo emissor e faz *destuffing* através da *statemachine*. A cada receção de uma trama de informação o recetor verifica o seu conteúdo e envia uma resposta ao emissor (*RR* caso esteja tudo bem, *REJ* caso algum erro tenha sido detetado). Caso o recetor, durante a execução do *llread*, receba um campo de controlo *DISC*, a função *llread* termina e a AL chama a função *llclose*. Quando o processo de receção de dados termina, a AL chama a função *llclose* para terminar a ligação entre os dois computadores.

5 Protocolo de ligação lógica

5.1 *llopen*

Esta função é responsável pelo estabelecimento da ligação entre os dois computadores envolvidos. Assim sendo, independentemente se se trata do emissor ou recetor, esta função é chamada em *application_layer.c* para efetuar a abertura da porta série pela qual será transmitida a informação do ficheiro.

Emissor

- **Envio de trama de controlo *SET*** - É enviada uma trama de controlo *SET* que indica ao recetor que será iniciada a transferência de dados.

- **Receção de trama de controlo *UA*** - O emissor recebe uma trama de controlo *UA* que confirma que o recetor recebeu a primeira trama de controlo. Se esta confirmação não chegar ao fim de 4 segundos, ocorre um *timeout* e é reenviada a primeira trama de controlo que contém o *SET* repetindo o processo.

Recetor

O recetor recebe a trama de controlo *SET* e envia a confirmação de que a transferência de dados pode começar através de uma trama de controlo que contém *UA*.

5.2 *llwrite*

Esta função é responsável pelo envio de tramas.

- É criada um cabeçalho da trama a ser enviada.
- É feito o *stuffing* da mensagem a ser enviada assim como o cálculo do *bcc2*. Após este processo, ainda é efetuado o *stuffing* do *bcc2*.
- É enviada uma trama de informação para a porta série e o emissor espera por uma resposta do recetor.
- Caso a resposta recebida seja negativa (*REJ*) ou occora um *timeout*, a função reenvia a trama de informação. Caso contrário, a função termina corretamente uma vez que recebeu uma resposta positiva (*RR*).

5.3 *llread*

Esta função é responsável pela receção de tramas.

- É efetuada a leitura *byte* a *byte* da porta série.
- É efetuado o *destuffing* do campo de dados da trama de informação recebida.
- Os campos de proteção *bcc1* e *bcc2* são analisados para determinar a resposta a ser enviada para o emissor.
- Caso se detete algum erro, é enviado um *REJ*. Caso não seja detetado nenhum erro, é enviado um *RR* e o conteúdo do campo de dados é guardado num *buffer* passado como parâmetro da função.

5.4 *llclose*

Esta função é responsável por terminar a ligação entre os dois computadores envolvidos.

Emissor

- **Envio de trama de controlo *DISC*** - É enviada uma trama de controlo que contém *DISC* que indica que ligação de dados será terminada.
- **Receção de trama de controlo *DISC*** - O recetor envia uma trama de controlo *DISC*. Se esta trama não for recebida pelo emissor ao fim de 4 segundos, ocorre um *timeout* e o processo repete-se.
- **Envio de trama de controlo *UA*** - Quando é recebida a trama de controlo *DISC*, é enviada uma trama de controlo *UA* que termina a ligação de dados.

Recetor

O recetor rebece uma trama de controlo *DISC* e de seguida envia uma trama semelhante à trama de controlo recebida. Por fim, recebe uma trama de controlo *UA* e a ligação de dados é terminada.

6 Protocolo de aplicação

Este protocolo serve como uma aplicação de teste para conseguirmos testar o Protocolo de Ligação de Dados (tema principal do trabalho). Assim, o protocolo de aplicação é bastante simples sendo constituído apenas por uma função principal e uma auxiliar.
Mais uma vez, o programa toma diferentes rumos dependendo do tipo de utilizador que o está a usar.

6.1 Emissor

O programa começa por criar uma estrutura de dados *Link Layer* para guardar os argumentos recebidos pelo ficheiro *main.c*. De seguida chama a função *llopen* da *Link Layer* para dar início à ligação. Abre o ficheiro que vai ser enviado e guarda o seu tamanho numa variável *file_size*. Com estes dados consegue construir o primeiro pacote de controlo da *Application Layer* com o campo de controlo *start*, um identificador de que o que vai ser enviado no campo seguinte é o tamnho do ficheiro e, por fim, a variável *file_size*. Este pacote de controlo é então enviado ao recetor através da função *llwrite*. A partir daqui começa uma sequência de iterações em que o ficheiro é lido um certo número de *bytes* de cada vez e construído um pacote de dados que é enviado ao recetor também através da função *llwrite*. Quando o processo da transmissão do ficheiro termina, a *Application Layer* executa a função *llwrite* para terminar a ligação.

6.2 Recetor

O ínicio da execução do programa é semelhante para o emissor e recetor. É também criada uma estrutura de dados *Link Layer* para guardar os mesmos argumentos e de

seguida chamada a função *llopen* para dar ínicio à ligação. Agora, a *Application Layer* chama a função *llread* para ler o primeiro pacote de controlo que dará início à transmissão de dados. Atráves deste pacote, a AL já sabe o tamanho esperado do ficheiro. É criado o ficheiro que vai ser recebido com o nome fornecido pelos argumentos enviados pelo ficheiro *main.c*. A partir daqui começa uma sequência de iterações em que é chamada a função *llread* para ler os pacotes de dados que estão a ser recebidos. Aqui há um controlo caso seja recebido um pacote de controlo *end* antes de o número de *bytes* recebidos ser igual ao número de *bytes* esperado do ficheiro. Quando o processo da transmissão do ficheiro termina, o resto da execução da *Application Layer* é igual ao emissor.

7 Validação e Eficiência do protocolo de ligação de dados

Para testar o funcionamento correto do programa, este foi sujeito aos seguintes testes concluindo-os com sucesso:
- Envio de ficheiros de diversos tamanhos.
- Envio de um mesmo ficheiro com pacotes de tamanhos diferentes.
- Envio de um mesmo ficheiro com várias *baudrates*.
- Interrupção da ligação da porta série durante o envio do ficheiro.
- Introdução de ruído na porta série durante o envio do ficheiro.

Uma vez que a definição de *baudrate* é o número de *bits* transmitidos por segundo, é possível assim calcular o número de *bits* real transmitido pelo programa por segundo. Usando uma *baudrate* de 9600, significa que é suposto o programa transmitir até um máximo de 1200 *bytes* por segundo. Usando um tamanho de pacote de dados igual à *baudrate* (1200 *bytes*) e fazendo a média do tempo demorado em todas as iterações em que é enviado um pacote de dados, quando se divide esse resultado pela *baudrate* obtém-se a enficiência do programa.
Infelizmente este cálculo de eficiência não foi implementado no nosso trabalho uma vez que houve sobrecarga de trabalho que nos impossibilitou de ir ao laboratório realizar testes à eficiência.

8 Conclusões

Em suma, a criação do protocolo de ligação de dados que permite a transferência de informação entre dois sistemas, foi bem sucedida, cumprindo todos os objetivos propostos no guião e passando nos testes submetidos. Com o desenvolvimento do trabalho, foram adquiridos conhecimentos teórico-práticos em relação ao tema abordado.

Anexo I - Código fonte

main.c

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

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

#include "application_layer.h"

#define BAUDRATE 9600
#define N_TRIES 3
#define TIMEOUT 4

// Arguments:
//   $1: /dev/ttySxx
//   $2: tx | rx
//   $3: filename
int main(int argc, char *argv[])
{
    if (argc < 4)
    {
        printf("Usage: %s /dev/ttySxx tx|rx filename\n", argv[0]);
        exit(1);
    }

    const char *serialPort = argv[1];
    const char *role = argv[2];
    const char *filename = argv[3];

    printf("Starting link-layer protocol application\n"
           "  - Serial port: %s\n"
           "  - Role: %s\n"
           "  - Baudrate: %d\n"
           "  - Number of tries: %d\n"
           "  - Timeout: %d\n"
           "  - Filename: %s\n",
           serialPort,
           role,
           BAUDRATE,
           N_TRIES,
           TIMEOUT,
           filename);

    applicationLayer(serialPort, role, BAUDRATE, N_TRIES, TIMEOUT, filename
    );

    return 0;
```

```
}
```

application_layer.h

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

#ifndef _APPLICATION_LAYER_H_
#define _APPLICATION_LAYER_H_

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

#endif // _APPLICATION_LAYER_H_
```

application_layer.c

```
// Application layer protocol implementation
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <time.h>
#include "application_layer.h"
#include "link_layer.h"
#include "macros.h"

unsigned char appbuf[PACKET_SIZE+30];

int read_next_TLV(unsigned char *addr, unsigned char* t, unsigned char* l,
    unsigned char** v){
    *t=addr[0];
    *l=addr[1];
    *v=addr+2;
    return 2 + *l; //returns size of the TLV.
}


void applicationLayer(const char *serialPort, const char *role, int
    baudRate, int nTries, int timeout, const char *filename)
{
    clock_t begin = clock();

```

```
    LinkLayer ll;
    strcpy(ll.serialPort,serialPort);

    if(strcmp(role,"tx")==0) {
        ll.role = LlTx;
    } else if(strcmp(role, "rx")==0) {
        ll.role = LlRx;
    } else {
        printf("ERROR: Not a defined role.\n");
        return;
    }

    ll.baudRate = baudRate;
    ll.nRetransmissions = nTries;
    ll.timeout = timeout;

    printf("\nExecuting llopen:\n\n");

    if(llopen(ll)<0) {
        printf("ERROR: An error occurred while opening the connection.\n");
        llclose(0);
        return;
    }

    if(ll.role == LlTx) {
        FILE* file = fopen(filename, "r");

        if(!file) {
            printf("ERROR: Could not open file to be sent.\n");
            return;
        } else {
            printf("-> Successfully opened file to be sent.\n");
        }

        fseek(file, 0L, SEEK_END);
        long int file_size = ftell(file);
        fseek(file,0,SEEK_SET);
        printf("File Size: %li bytes\n", file_size);
        appbuf[0]=CONTROL_START;
        appbuf[1]=TYPE_FILESIZE;
        appbuf[2]=sizeof(long);

        printf("\nExecuting llwrite:\n\n");

        *((long*)(appbuf+3))=file_size;
        unsigned char failure=0;
        if(-1==llwrite(appbuf,10)){
            printf("ERROR: llwrite failure.\n");
            failure=1;
        }
```

```
74          unsigned long bytes_sent =0;
75          for(unsigned char i=0;bytes_sent<file_size && 0==failure;++i){
76              unsigned long file_bytes = fread(appbuf+4, 1, (file_size-
    bytes_sent<PACKET_SIZE? file_size-bytes_sent : PACKET_SIZE), file);
77
78              if(file_bytes!=(file_size-bytes_sent<PACKET_SIZE? file_size-
    bytes_sent:PACKET_SIZE)){
79                  printf("ERROR: File read failure. file_bytes:%lu, file_size
    :%lu, bytes_sent:%lu\n",file_bytes,file_size,bytes_sent);
80                  failure=1;
81                  break;
82              }
83              appbuf[0]=CONTROL_DATA;
84              appbuf[1]=i;
85              appbuf[2]=file_bytes>>8;
86              appbuf[3]=file_bytes%256;
87              if(-1==llwrite(appbuf,file_bytes+4)){
88                  printf("ERROR: llwrite failure.\n");
89                  failure=1;
90                  break;
91              }
92              printf("Sent packet %i.\n",i);
93              bytes_sent+=file_bytes;
94          }
95          if(!failure){
96              appbuf[0]=CONTROL_END;
97              if(-1==llwrite(appbuf,1)){
98                  printf("ERROR: llwrite on end packet failure.\n");
99              }else{
100                 printf("-> Finished sending.\n");
101             }
102         }
103         fclose(file);
104
105     } else if(ll.role == LlRx) {
106         unsigned long filesize=0,size_received=0;
107         printf("\nExecuting llread:\n\n");
108         int bytes_read = llread(appbuf);
109         unsigned char t,l,*v;
110         if(appbuf[0] == CONTROL_START){
111             int offset=1;
112             for(;offset<bytes_read;){
113                 offset+=read_next_TLV(appbuf+offset,&t,&l,&v);
114                 if(t==TYPE_FILESIZE){
115                     filesize=*((unsigned long*)v);
116                     printf("File Size:%li\n",filesize);
117                 }
118             }
119             FILE* file = fopen(filename, "w");
120             if(!file) {
```

```
121                 printf("ERROR: Could not open file to write in.\n");
122                 return;
123             } else {
124                 printf("-> Received control packet and created file to
    write.\n");
125             }
126             unsigned char early_end =0, last_sequence_number=0;
127             for(;size_received<filesize;){
128                 int numbytes=llread(appbuf);
129                 if(numbytes<1){
130                     if(numbytes==-1){
131                         printf("ERROR: llread failure.\n");
132                         break;
133                     }else
134                         printf("ERROR: Received a packet that is too small.
     (%i bytes)\n",numbytes);
135                 }
136                 if(appbuf[0]==CONTROL_END){
137                     printf("ERROR: Disconnected before end of file.\n");
138                     early_end=1;
139                     break;
140                 }
141                 if(appbuf[0]==CONTROL_DATA){
142                     if(numbytes<5)
143                         printf("ERROR: Received a data packet that is too
    small. (%i bytes)\n",numbytes);
144                     if(appbuf[1]!=last_sequence_number){
145                         printf("ERROR: Received packet with wrong sequence
    number. Was %i, expected:%i\n",appbuf[1],last_sequence_number-1);
146                     }else{
147                         unsigned long size=appbuf[3]+appbuf[2]*256;
148                         if(size!=numbytes-4)
149                             printf("ERROR: Packet length didn't match
    header. Was %i, expected %lu.\n",numbytes-4,size);
150                         fwrite(appbuf+4,1,size,file);
151                         size_received+=size;
152                         printf("Received packet numbered %i.\n",
    last_sequence_number++);
153                     }
154                 }
155             }
156             if(!early_end){
157                 int numbytes=llread(appbuf);
158                 if(numbytes<1){
159                     if(numbytes==-1)
160                         printf("ERROR: llread failure.\n");
161                     else
162                         printf("ERROR: Received a packet that is too small.
     (%i bytes)\n",numbytes);
163                 }
```

```
                if(appbuf[0]!=CONTROL_END){
                    printf("ERROR: Received an unexpected packet. Expected end control packet.\n");
                }else{
                    printf("-> Received end packet. Disconnecting\n");
                }
            }
            fclose(file);
        }else{
            printf("ERROR: Transmission didn't start with a start packet.\n");
            for(unsigned int i=0;i<10;++i)
                printf("%i ",appbuf[i]);
        }
    }

    printf("\nExecuting llclose:\n\n");
    llclose(1);
    sleep(1);
    clock_t end = clock();
    double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
    printf("Time: %f\n", time_spent);
}
```

link_layer.h

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

#ifndef _LINK_LAYER_H_
#define _LINK_LAYER_H_

typedef enum
{
    LlTx,
    LlRx,
} LinkLayerRole;

typedef struct
{
    char serialPort[50];
    LinkLayerRole role;
    int baudRate;
    int nRetransmissions;
    int timeout;
} LinkLayer;

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

```

```
26 // MISC
27 #define FALSE 0
28 #define TRUE 1
29
30 // Open a connection using the "port" parameters defined in struct
       linkLayer.
31 // Return "1" on success or "-1" on error.
32 int llopen(LinkLayer connectionParameters);
33
34 // Send data in buf with size bufSize.
35 // Return number of chars written, or "-1" on error.
36 int llwrite(const unsigned char *buf, int bufSize);
37
38 // Receive data in packet.
39 // Return number of chars read, or "-1" on error.
40 int llread(unsigned char *packet);
41
42 // Close previously opened connection.
43 // if showStatistics == TRUE, link layer should print statistics in the
       console on close.
44 // Return "1" on success or "-1" on error.
45 int llclose(int showStatistics);
46
47 #endif // _LINK_LAYER_H_
```

link_layer.c

```
1  // Link layer protocol implementation
2
3  #include <termios.h>
4  #include <stdio.h>
5  #include <unistd.h>
6  #include <fcntl.h>
7  #include <signal.h>
8  #include <string.h>
9  #include <stdlib.h>
10 #include "link_layer.h"
11 #include "macros.h"
12 #include "statemachine.h"
13
14 // MISC
15 #define _POSIX_SOURCE 1 // POSIX compliant source
16
17 LinkLayer connection;
18 struct termios oldtio;
19 struct termios newtio;
20 int fd;
21 int receivedDISC = 0;
22 unsigned char alarmEnabled=0, tries=0;
23 unsigned char buf[128], *bigBuf =NULL;
24 unsigned long bigBufsize=0;
```

```
25 unsigned char data_s_flag = 0;
26
27 int nTimeouts = 0;
28 int nRetransmissions = 0;
29 float effiency = 0;
30
31 int stuff(const unsigned char *buffer, int bufSize, unsigned char* dest,
       unsigned char *bcc){
32     int size=0;
33     for(unsigned int i=0; i<bufSize ; ++i){
34         if(bcc!=NULL)
35             *bcc^=buffer[i];
36         if(buffer[i]==FLAG){
37             dest[size++]=ESCAPE;
38             dest[size++]=ESCAPE_FLAG;
39             break;
40         }
41         if(buffer[i]==ESCAPE){
42             dest[size++]=ESCAPE;
43             dest[size++]=ESCAPE_ESCAPE;
44             break;
45         }
46         dest[size++]=buffer[i];
47     }
48     return size;
49 }
50 int buildCommandFrame(unsigned char* buffer, unsigned char address,
       unsigned char control){
51     buffer[0]= FLAG;
52     buffer[1]= address;
53     buffer[2]= control;
54     buffer[3]= address ^ control;
55     //is command packet.
56     buffer[4]= FLAG;
57     return 5;
58 }
59 int buildDataFrame(unsigned char* framebuf, const unsigned char* data,
       unsigned int data_size, unsigned char address, unsigned char control){
60     framebuf[0]= FLAG;
61     framebuf[1]= address;
62     framebuf[2]= control;
63     framebuf[3]= address ^ control;
64     int offset=0;
65     unsigned char bcc=0;
66     for(unsigned int i=0;i<data_size;++i){
67         offset+=stuff(data+i,1,framebuf+offset+4,&bcc);
68     }
69     offset+=stuff(&bcc,1,framebuf+offset+4,NULL);
70     framebuf[4+offset]=FLAG;
71     return 5+offset;
```

```
72 }
73
74
75 void alarmHandler(int signal){
76     ++tries;
77     alarmEnabled=0;
78 }
79
80 ////////////////////////////////////////////////
81 // LLOPEN
82 ////////////////////////////////////////////////
83 int llopen(LinkLayer connectionParameters)
84 {
85     connection=connectionParameters;
86     // Open serial port device for reading and writing and not as
    controlling tty
87     // because we don't want to get killed if linenoise sends CTRL-C.
88     fd = open(connection.serialPort, O_RDWR | O_NOCTTY);
89     if (fd < 0)
90     {
91         perror(connection.serialPort);
92         exit(-1);
93     }
94     // Save current port settings
95     if (tcgetattr(fd, &oldtio) == -1)
96     {
97         perror("tcgetattr");
98         exit(-1);
99     }
100
101     // Clear struct for new port settings
102     memset(&newtio, 0, sizeof(newtio));
103
104     newtio.c_cflag = connection.baudRate | CS8 | CLOCAL | CREAD;
105     newtio.c_iflag = IGNPAR;
106     newtio.c_oflag = 0;
107
108     // Set input mode (non-canonical, no echo,...)
109     newtio.c_lflag = 0;
110     newtio.c_cc[VTIME] = 0; // Inter-character timer unused
111     newtio.c_cc[VMIN] = 0;  // Blocking read until 1 char received
112
113     // VTIME e VMIN should be changed in order to protect with a
114     // timeout the reception of the following character(s)
115
116     // Now clean the line and activate the settings for the port
117     // tcflush() discards data written to the object referred to
118     // by fd but not transmitted, or data received but not read,
119     // depending on the value of queue_selector:
120     //   TCIFLUSH - flushes data received but not read.
```

```
121     tcflush(fd, TCIOFLUSH);
122
123     // Set new port settings
124     if (tcsetattr(fd, TCSANOW, &newtio) == -1)
125     {
126         perror("tcsetattr");
127         exit(-1);
128     }
129
130     signal(SIGALRM,alarmHandler);
131
132     if(connection.role==LlTx){ //Transmitter
133         int receivedUA=0;
134         state.state=SMSTART;
135         tries=0;
136         while(tries<connection.nRetransmissions && !receivedUA){
137             alarm(connection.timeout);
138             alarmEnabled=1;
139             if(tries>0) {
140                 printf("Timed out.\n");
141                 nTimeouts++;
142             }
143             int size = buildCommandFrame(buf,ADR_TX,CTRL_SET);
144             printf("-> Sent SET.\n");
145             write(fd,buf,size);
146             while(alarmEnabled && !receivedUA){
147                 int bytes_read = read(fd,buf,PACKET_SIZE_LIMIT);
148                 if(bytes_read<0)
149                     return -1;
150                 for(unsigned int i=0;i<bytes_read && !receivedUA;++i){
151                     state_machine(buf[i],&state);
152                     if(state.state==SMEND && state.adr==ADR_TX && state.
    ctrl == CTRL_UA)
153                         receivedUA=1;
154                 }
155             }
156         }
157         if(receivedUA)
158             printf("-> Received UA.\n");
159         else
160             return -1;
161         return 1;
162     }
163     else{ // Receiver
164         tries=0;
165         state.state=SMSTART;
166         int receivedSET=0;
167         while(!receivedSET){
168             int bytes_read = read(fd,buf,PACKET_SIZE_LIMIT);
169             if(bytes_read<0)
```

```
170                 return -1;
171             for(unsigned int i=0;i<bytes_read && !receivedSET;++i){
172                 state_machine(buf[i],&state);
173                 if(state.state==SMEND && state.adr==ADR_TX && state.ctrl ==
     CTRL_SET)
174                     receivedSET=1;
175                 if(state.state==SMEND && state.adr==ADR_TX && state.ctrl==
    CTRL_DISC) {
176       receivedDISC = 1;
177       printf("-> Received DISC.\n");
178       return -1;
179     }
180             }
181         }
182         if(receivedSET) printf("-> Received Set.\n");
183         int frame_size=buildCommandFrame(buf,ADR_TX,CTRL_UA);
184         //sleep(9);
185         write(fd,buf,frame_size); //sends UA reply.
186         printf("-> Sent UA.\n");
187         return 1;
188     }
189     return -1;
190 }
191
192 ////////////////////////////////////////////////
193 // LLWRITE
194 ////////////////////////////////////////////////
195 int llwrite(const unsigned char *buffer, int bufferSize)
196 {
197     if(bigBufsize < bufferSize*2+10){
198         if(bigBufsize==0)
199             bigBuf=malloc(bufferSize*2+10);
200         else
201             bigBuf=realloc(bigBuf,bufferSize*2+10);
202     }
203     int frame_size=buildDataFrame(bigBuf,buffer,bufferSize,ADR_TX,CTRL_DATA
    (data_s_flag));
204     for(unsigned int sent=0;sent<frame_size;){ //In case write doesnt write
     all bytes from the first call.
205         int ret=write(fd,bigBuf+sent,frame_size-sent);
206         if(ret==-1)
207             return -1;
208         sent+=ret;
209     }
210     int receivedPacket=0, resend=0, retransmissions=0;
211     state.data=NULL; //State machine writes to packet buffer directly.
212
213     alarmEnabled=1;
214     alarm(connection.timeout);
215     while(!receivedPacket){
```

```
216         if(!alarmEnabled){
217             resend=1;
218             alarmEnabled=1;
219             alarm(connection.timeout);
220         }
221         if(resend){
222             if(retransmissions>0) {
223                 printf("Timed out.\n");
224                 nTimeouts++;
225             }
226             if(retransmissions == connection.nRetransmissions){
227                 printf("Exceeded retransmission limit.\n");
228                 return -1;
229             }
230 
231             for(unsigned int sent=0;sent<frame_size;){ //In case write
    doesnt write all bytes from the first call.
232                 int ret=write(fd,bigBuf+sent,frame_size-sent);
233                 if(ret==-1)
234                     return -1;
235                 sent+=ret;
236             }
237             resend = 0;
238             retransmissions++;
239         }
240         int bytes_read = read(fd,buf,PACKET_SIZE_LIMIT);
241         if(bytes_read<0)
242             return -1;
243         for(unsigned int i=0;i<bytes_read && !receivedPacket &&
    alarmEnabled;++i){ //TODO avoid discarding reads after valid packet.
244             state_machine(buf[i],&state);
245             if(state.state==SMEND){
246                 if(state.adr==ADR_TX && (state.ctrl == CTRL_RR(0) || state.
    ctrl == CTRL_RR(1))){ //Receiver Ready for next.
247                     receivedPacket = 1;
248                     if(state.ctrl == CTRL_RR(data_s_flag))//Requesting next
     packet.
249                         printf("Requesting next packet.\n");
250                     resend = 0;
251                 }
252                 if(state.adr==ADR_TX && state.ctrl == CTRL_REJ(data_s_flag)
    ){//Requesting retransmission.
253                     printf("Requesting retransmission.\n");
254                 retransmissions=0;
255                     nRetransmissions++;
256                 }
257             }
258         }
259     }
260     data_s_flag= data_s_flag?0:1;
```

```
261     return 0;
262 }
263
264 ////////////////////////////////////////////////
265 // LLREAD
266 ////////////////////////////////////////////////
267 int llread(unsigned char *packet)
268 {
269     if(bigBufsize < PACKET_SIZE_LIMIT){
270         if(bigBufsize==0)
271             bigBuf=malloc(PACKET_SIZE_LIMIT);
272         else
273             bigBuf=realloc(bigBuf,PACKET_SIZE_LIMIT);
274     }
275     int receivedPacket=0;
276     state.data=packet; //State machine writes to packet buffer directly.
277     while(!receivedPacket){
278         int bytes_read = read(fd,bigBuf,PACKET_SIZE_LIMIT);
279         if(bytes_read<0)
280             return -1;
281         for(unsigned int i=0;i<bytes_read && !receivedPacket;++i){
282             state_machine(bigBuf[i],&state);
283             if(state.state==SMREJ && state.adr==ADR_TX){
284                 int frame_size=buildCommandFrame(buf,ADR_TX,(state.ctrl==
    CTRL_DATA(0)?CTRL_REJ(0):CTRL_REJ(1)));
285                 write(fd,buf,frame_size); //sends REJ reply.
286                 printf("-> Sent REJ.\n");
287             }
288             if(state.state==SMEND && state.adr==ADR_TX && state.ctrl ==
    CTRL_SET){
289                 int frame_size=buildCommandFrame(buf,ADR_TX,CTRL_UA);
290                 write(fd,buf,frame_size); //sends UA reply.
291                 printf("-> Sent UA.\n");
292             }
293             if(state.state==SMEND && state.adr==ADR_TX){
294                 if(state.ctrl == CTRL_DATA(data_s_flag)){
295                     data_s_flag=data_s_flag?0:1;
296                     int frame_size=buildCommandFrame(buf,ADR_TX,CTRL_RR(
    data_s_flag));
297                     write(fd,buf,frame_size);
298                     printf("-> Sent RR %i.\n",data_s_flag);
299                     return state.data_size;
300                 }
301                 else{
302                     int frame_size=buildCommandFrame(buf,ADR_TX,CTRL_RR(
    data_s_flag));
303                     write(fd,buf,frame_size);
304                     printf("-> Sent RR %i requesting retransmission.\n",
    data_s_flag);
305                 }
```

```
306             }
307             if(state.ctrl==CTRL_DISC) {
308                 receivedDISC = 1;
309                 int frame_size=buildCommandFrame(buf,ADR_TX,(state.ctrl==
    CTRL_DATA(0)?CTRL_REJ(0):CTRL_REJ(1)));
310                 write(fd,buf,frame_size); //sends REJ reply.
311                 printf("-> Received DISC.\n");
312                 return -1;
313                 break;
314             }
315         }
316     }
317     return 0;
318 }
319
320 ////////////////////////////////////////////////
321 // LLCLOSE
322 ////////////////////////////////////////////////
323 int llclose(int showStatistics)
324 {
325     signal(SIGALRM,alarmHandler);
326     if(bigBufsize>0)
327         free(bigBuf);
328
329     if(connection.role==LlTx) { //Transmitter
330
331         int receivedDISC_tx=0;
332         tries=0;
333         while(tries<connection.nRetransmissions && !receivedDISC_tx){
334             alarm(connection.timeout);
335             alarmEnabled=1;
336             if(tries>0) {
337                 printf("Timed out.\n");
338                 nTimeouts++;
339             }
340             int size = buildCommandFrame(buf,ADR_TX,CTRL_DISC);
341             printf("-> Sent DISC.\n");
342             write(fd,buf,size);
343             while(alarmEnabled && !receivedDISC_tx){
344                 int bytes_read = read(fd,buf,PACKET_SIZE_LIMIT);
345                 if(bytes_read<0)
346                     return -1;
347                 for(unsigned int i=0;i<bytes_read && !receivedDISC_tx;++i){
348                     state_machine(buf[i],&state);
349                     if(state.state==SMEND && state.adr==ADR_TX && state.
    ctrl == CTRL_DISC)
350                         receivedDISC_tx=1;
351                     if(state.state==SMEND && state.adr==ADR_TX && (state.
    ctrl == CTRL_RR(0) || state.ctrl == CTRL_RR(1) || state.ctrl ==
    CTRL_REJ(0) || state.ctrl == CTRL_REJ(1))){
```

```
352                         tries=0; //reset tries as receiver was still in
    llread.
353                     }
354                 }
355             }
356         }
357         if(receivedDISC_tx){
358             printf("-> Received DISC.\n");
359             int frame_size=buildCommandFrame(buf,ADR_TX,CTRL_UA);
360             write(fd,buf,frame_size); //sends UA reply.
361             printf("-> Sent UA.\n");
362             sleep(1);
363         }
364 
365     } else { //Receiver
366 
367         while(!receivedDISC){
368             int bytes_read = read(fd,buf,PACKET_SIZE_LIMIT);
369             if(bytes_read<0)
370                 return -1;
371             for(unsigned int i=0;i<bytes_read && !receivedDISC;++i){
372                 state_machine(buf[i],&state);
373                 if(state.state==SMEND && state.adr==ADR_TX && state.ctrl ==
     CTRL_DISC)
374                     receivedDISC=1;
375             }
376         }
377         if(receivedDISC) printf("-> Received DISC.\n");
378         int frame_size=buildCommandFrame(buf,ADR_TX,CTRL_DISC);
379         write(fd,buf,frame_size); //sends DISC reply.
380         printf("-> Sent DISC.\n");
381 
382         int receivedUA=0;
383         while(!receivedUA){
384             int bytes_read = read(fd,buf,PACKET_SIZE_LIMIT);
385             if(bytes_read<0)
386                 return -1;
387             for(unsigned int i=0;i<bytes_read && !receivedUA;++i){
388                 state_machine(buf[i],&state);
389                 if(state.state==SMEND && state.adr==ADR_TX && state.ctrl ==
     CTRL_UA)
390                     receivedUA=1;
391             }
392         }
393         if(receivedUA) printf("-> Received UA.\n");
394     }
395 
396     // Restore the old port settings
397     if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
398     {
```

```
399         perror("tcsetattr");
400         exit(-1);
401     }
402 
403     if(showStatistics && connection.role==LlTx) {
404         printf("\nStatistics:\n\n");
405         printf("- Number of Timeouts: %d\n", nTimeouts);
406         printf("- Number of Retransmissions: %d\n", nRetransmissions);
407 
408     }
409 
410     close(fd);
411     return 1;
412 }
```

statemachine.h

```
1  // Statemachine header.
2 
3  #ifndef _STATEMACHINE_H_
4  #define _STATEMACHINE_H_
5 
6  typedef struct {
7      enum state_t { SMSTART,SMFLAG,SMADR,SMCTRL,SMBCC1,SMDATA,SMESC,SMBCC2,
       SMEND,SMREJ} state;
8      unsigned char adr;
9      unsigned char ctrl;
10     unsigned char bcc;
11     unsigned char *data;
12     unsigned int data_size;
13 } State;
14 State state;
15 
16 void state_machine(unsigned char byte, State* state);
17 
18 #endif // _STATEMACHINE_H_
```

statemachine.c

```
1  #include <stdio.h>
2  #include <string.h>
3  #include "macros.h"
4  #include "statemachine.h"
5 
6  void state_machine(unsigned char byte,State* state){
7      switch (state->state){
8          case SMREJ:
9          case SMEND:
10             state->state=SMSTART;
11         case SMSTART:
12             if(byte==FLAG)
```

```
13                  state->state=SMFLAG;
14              break;
15          case SMFLAG:
16              state->data_size=0;
17              if(byte==FLAG){
18                  break;
19              }
20              if(byte==ADR_TX || byte == ADR_RX){
21                  state->state=SMADR;
22                  state->adr=byte;
23                  break;
24              }
25              state->state=SMSTART;
26              break;
27          case SMADR:
28              if(byte==FLAG){
29                  state->state=SMFLAG;
30                  break;
31              }
32              if(byte==CTRL_DISC || byte==CTRL_SET
33              || byte==CTRL_UA || byte == CTRL_REJ(0)
34              || byte==CTRL_RR(0) || byte == CTRL_REJ(1)
35              || byte==CTRL_RR(1) || byte == CTRL_DATA(0)
36              || byte==CTRL_DATA(1)){
37                  state->state = SMCTRL;
38                  state->ctrl = byte;
39                  state->bcc = state->adr ^ state->ctrl;
40                  break;
41              }
42              state->state=SMSTART;
43              break;
44          case SMCTRL:
45              if( byte == state->bcc){
46                  state->state=SMBCC1;
47                  break;
48              }
49              if(byte==FLAG){
50                  state->state = SMFLAG;
51                  break;
52              }
53              state->state=SMSTART;
54              break;
55          case SMBCC1:
56              if(byte==FLAG){
57                  if(state->ctrl==CTRL_DATA(0) || state->ctrl==CTRL_DATA(1)){
58                      //Received flag when expecting data.
59                      state->state=SMFLAG;
60                      break;
61                  }
62                  state->state=SMEND; //received valid frame.
```

```
63                  break;
64              }
65              if((state->ctrl==CTRL_DATA(0) || state->ctrl==CTRL_DATA(1) ) &&
    state->data != NULL){
66                  state->data_size=0;
67                  if(byte==ESCAPE){
68                      state->bcc=0;
69                      state->state=SMESC;
70                      break;
71                  }
72                  state->data[state->data_size++]=byte;
73                  state->bcc=byte;
74                  state->state=SMDATA;
75                  break;
76              }
77              state->state=SMSTART;
78              break;
79          case SMDATA:
80              if(byte==ESCAPE){
81                  state->state=SMESC;
82                  break;
83              }
84              if(byte==FLAG){
85                  state->state=SMREJ;
86                  break;
87              }
88              if(byte==state->bcc){
89                  state->state=SMBCC2;
90                  break;
91              }
92              state->data[state->data_size++]=byte;
93              state->bcc^=byte;
94              break;
95          case SMESC:
96              if(byte==FLAG){
97                  state->state=SMREJ;
98                  break;
99              }
100             if(byte==ESCAPE_FLAG){
101                 if(state->bcc==FLAG){
102                     state->state=SMBCC2;
103                     break;
104                 }
105                 state->bcc^=FLAG;
106                 state->data[state->data_size++]=FLAG;
107                 state->state=SMDATA;
108                 break;
109             }
110             if(byte==ESCAPE_ESCAPE){
111                 if(state->bcc==ESCAPE){
```

```
                    state->state=SMBCC2;
                    break;
                }
                state->bcc^=ESCAPE;
                state->data[state->data_size++]=ESCAPE;
                state->state=SMDATA;
                break;
            }
            state->state=SMSTART;
            break;
        case SMBCC2:
            if(byte==FLAG){
                state->state=SMEND;
                break;
            }
            if(byte==0){
                state->data[state->data_size++]=state->bcc;
                state->bcc=0;
                break;
            }
            if(byte==ESCAPE){
                state->data[state->data_size++]=state->bcc;
                state->bcc=0;
                state->state=SMESC;
                break;
            }
            state->data[state->data_size++]=state->bcc;
            state->data[state->data_size++]=byte;
            state->bcc=byte;
            state->state=SMDATA;
            break;
    }
}
```

macros.h

```
// Macros header.

#ifndef _MACROS_H_
#define _MACROS_H_

//Application Layer
#define PACKET_SIZE (1024)
#define CONTROL_START (2)
#define CONTROL_END (3)
#define CONTROL_DATA (1)
#define TYPE_FILESIZE (0)

//Link Layer
#define FLAG (0b01111110)
#define ESCAPE (0x7d)
```

```
16 #define ESCAPE_FLAG (0x5e)
17 #define ESCAPE_ESCAPE (0x5d)
18 #define ADR_TX (0b00000011)
19 #define ADR_RX (0b00000001)
20 #define CTRL_SET (0b00000011)
21 #define CTRL_DISC (0b00001011)
22 #define CTRL_UA (0b00000111)
23 #define CTRL_RR(R) ((R)%2?0b10000101:0b00000101)
24 #define CTRL_REJ(R) ((R)%2?0b10000001:0b00000001)
25 #define CTRL_DATA(S) ((S)%2?0b01000000:0b00000000)
26 #define PACKET_SIZE_LIMIT (128)
27
28 #endif // _MACROS_H_
```