Redes de Computadores

Protocolo de Ligação de Dados

Flávio Couto, João Gouveia e Pedro Afonso Castro

3 de Novembro de 2015

Sumário

No âmbito da unidade curricular de Redes de Computadores, foi-nos proposta a elaboração de uma aplicação que permitia a transferência de dados entre dois computadores através da porta de série. Para tal, recorremos à implementação de duas camadas: a camada de ligação de dados e a camada da aplicação.

O nosso grupo soube compreender bem os objetivos propostos, tendo implementado com sucesso não só estes mesmos objetivos, como alguns elementos de valorização propostos pelos docentes, servindo este relatório para mostrar isso mesmo.

Conteúdo

1 Introdução

O objetivo deste trabalho é implementar, num ambiente Linux, um protocolo de ligação de dados, recorrendo à porta de série. Assim, dois computadores poderão enviar dados um para o outro através deste protocolo, desde que estejam ligados por uma porta de série e tenham a capacidade de interpretar chamadas POSIX.

No que toca ao relatório, este tem como objetivo apresentar uma descrição detalhada deste trabalho, mostrando a sua arquitetura, estrutura do código, explicando os protocolos utilizados e mostrando resultados de testes efetuados.

2 Arquitetura

Para o projeto, foram usados dois PCs com sistema operativo Linux, ligados através de uma porta de série em modo não canónico e assíncrono. Utilizamos também uma estruturação em camadas para o código, em que cada uma das camadas é independente das outras. O nosso projeto está dividido em duas camadas: ligação de dados e aplicação. A camada de ligação de dados trata dos detalhes relativos ao protocolo de ligação de dados, bem como do *byte stuffing* e *byte destuffing*, abstraindo-se da informação contida nos pacotes que por si passam. Já a camada da aplicação efetua a divisão da informação em pacotes e respetivo processamento, enviando cada um à camada de ligação de dados, através de uma API disponibilizada por esta.

3 Estrutura do código

3.1 Camada de aplicação

A implementação da camada de aplicação encontra-se no ficheiro *applayer.c*. A camada contém uma estrutura de dados referente ao ficheiro a ser enviado/recebido, onde guardamos o descritor do ficheiro, o seu nome e o seu tamanho.

```
typedef struct {
        int fd;
        char name[MAX_FILENAME_SIZE];
```

As principais funções da camada de aplicação são as seguintes:

```
int app_transmitter(int argc, char **argv);
int app_receiver(int argc, char **argv);
int get_file_info(File_info_t* file_info, char* filePath);
```

3.2 Camada de ligação de dados

A implementação da camada de ligação de dados encontra-se nos ficheiros *linklayer.c* e *linklayer.h*. A camada contém uma estrutura de dados onde é guardado o descritor da porta série, o baudrate, o tempo de espera após falha de tranmissão, antes de ocorrer nova tentativa, o número de tentativas, o tamanho máximo das tramas I, uma flag contendo a informação de se a instância representa um transmissor ou recetor, uma estrutura de dados contendo as configurações anteriores à utilização do programa, uma variável que nos diz se a próxima trama a enviar/receber é do tipo I0 ou I1, um buffer utilizado para transmissão de informação e por último, informação acerca dos REJ recebidos e tramas I enviadas/recebidas (para efeitos de estatística).

```
struct LinkLayer_t {
        int fd;
        unsigned int baudrate;
        unsigned int timeout;
        unsigned int max_tries;
        unsigned int max_frame_size;
        int flag; // TRANSMITTER or RECEIVER
        struct termios oldtio;
```

```
        unsigned int sequence_number;
        char* buffer;
        int rejNo;
        int iNo;
};
```

As principais funções da camada de ligação de dados são as seguintes:

```
LinkLayer llinit(int port, int flag, unsigned int baudrate, unsigned
    int max_tries, unsigned int timeout, unsigned int max_frame_size);
int llopen(LinkLayer link_layer);
int llclose(LinkLayer link_layer;
int llread(LinkLayer link_layer, uint8_t *buf);
int llwrite(LinkLayer link_layer, uint8_t* buf, int length);
void lldelete(LinkLayer link_layer);
```

Temos também um ficheiro utils.h, que contém constantes simbólicas utilizadas nos vários ficheiros, e uma Makefile, utilizada para compilar mais facilmente o projeto.

4 Casos de uso principais

A aplicação resultante do trabalho contém dois casos de uso: o caso em que o programa funciona como emissor e o caso em que funciona como recetor.

No caso em que o programa funciona como emissor, temos a seguinte sequência de chamada de funções:

- O main começa por chamar a função **app_transmitter**, que faz o parse das flags inseridas pelo utilizador, cria uma instância de LinkLayer recorrendo à função **llinit**.
- Depois dessa instância ter sido criada com sucesso, tenta estabelecer uma conexão com o recetor através da função **llopen**.
- Estabelecida a conexão, envia o primeiro pacote de controlo, os pacotes de informação e o segundo pacote de controlo, através da função **llwrite**.
- Seguidamente, após a informação ter sido enviada, a conexão é fechada recorrendo à função **llclose**.
- Depois de fechada a conexão, o programa imprime as estatísticas recorrendo à função **lllog**.
- Por último, destrói a instância de LinkLayer recorrendo à função **llclose**.

É importante referir também que as funções **llopen** e **llclose** recorrem às funções auxiliares

llopen_transmitter e **llclose_transmitter**, respetivamente, para fazer as tarefas de transmissão. Estas funções ainda invocam funções auxiliares, mas a explicação da função dessas funções será dada mais à frente na secção 5.

No caso em que o programa funciona como recetor, temos a seguinte sequência de chamada de funções:

- O main começa por chamar a função **app_receiver**, que faz o parse das flags inseridas pelo utilizador, cria uma instância de LinkLayer recorrendo à função **llinit**.
- Depois dessa instância ter sido criada com sucesso, tenta estabelecer uma conexão com o transmissor através da função **llopen**.
- Estabelecida a conexão, recebe o primeiro pacote de controlo, os pacotes de informação e o segundo pacote de controlo, através da função **llread**.
- Seguidamente, após a informação ter sido recebida, a conexão é fechada recorrendo à função **llclose**.
- Depois de fechada a conexão, o programa imprime as estatísticas recorrendo à função **lllog**.
- Por último, destrói a instância de LinkLayer recorrendo à função **llclose**.

Tal como no emissor, as funções **llopen** e **llclose** recorrem a funções auxiliares para fazer as tarefas de receção de informação. Estas chamam-se **llopen_receiver** e **llclose_receiver**, respetivamente. As funções auxiliares que estas usam também serão explicadas na secção 5.

5 Protocolo de ligação lógica

A camada de ligação de dados é responsável pelas seguintes funcionalidades:

- Estabelecer a ligação entre o emissor e o recetor.
- Enviar e receber pacotes de informação, contendo *flags* e mensagens.
- Fazer *byte stuffing* e *byte destuffing* da informação recebida, consoante a necessidade.
- Validar as tramas recebidas.

5.1 API disponibilizada à camada de aplicação - LinkLayer

A camada de ligação de dados disponibiliza uma API para a camada de aplicação utilizar como achar necessário. Nesta API estão contidas 8 funções.

5.1.1 llinit

A função **llinit** cria uma instância da estrutura de dados LinkLayer, funcionando como um "construtor". Esta função inicializa os valores que a camada de ligação de dados precisa com os valores fornecidos pelo utilizador, ou, no caso de este não fornecer algum valor, com valores por predefinição.

5.1.2 llopen

A função **llopen** é responsável por estabelecer uma ligação através da porta de série.

Quando invocada pelo emissor, esta função envia uma *flag* SET, aguardando por uma resposta do recetor sob a forma de uma *flag* UA. Se esta resposta não chegar, o emissor espera um predeterminado número de segundos (definido na variável *timeout*) e tenta reenviar o SET. Se após um número de tentativas estipulado (na variável *max_ tries* o emissor continuar sem receber o UA, a tentativa de ligação é abortada. Em caso contrário, isso significa que a ligação foi corretamente estabelecida.

Quando invocada pelo recetor, esta função espera por uma *flag* SET, e, quando a recebe, envia uma *flag* UA. Se não receber o SET ao fim de *max_ tries* x *timeout*, a tentativa de ligação é abortada. Caso contrário, significa que a ligação foi corretamente estabelecida.

5.1.3 llwrite

A função **llwrite** é utilizada pelo emissor como forma de enviar informação para a porta de série. Recebe um buffer, bem como o seu tamanho, que contêm o conteúdo a enviar.

Recorre à função **write_frame** para enviar esse conteúdo. Esta função prepara as flags, juntando-as ao buffer, e envia a informação pela porta de série.

Depois de terminado o processo de envio, a função llwrite espera por uma resposta do recetor, que pode ser uma *flag* RR se a mensagem tiver sido transmitida corretamente, ou uma *flag* REJ se a mensagem não tiver sido transmitida corretamente, sendo esta retransmitida até ser corretamente aceite ou houver um *timeout*.

5.1.4 llread

A função **llread** é utilizada pelo recetor para receber informação da porta de série. Recebe um buffer onde é colocada a informação que vem no pacote.

Caso receba informação inválida, envia a *flag* REJ. Caso contrário, envia a *flag* RR. Recorre à função **read_frame** para receber o conteúdo. Esta função espera *max_ tries* x *timeout* segundos antes de abortar a ligação.

5.1.5 llclose

A função **llclose** é responsável por terminar a ligação. Se o invocador da função for o emissor, esta começará por enviar uma *flag* DISC, e aguardará pela recepção da mesma flag. Quando a receber, envia a *flag* UA e termina. No caso do recetor, espera por uma *flag* DISC, reenviando-a, para por último receber uma flag *flag* UA e terminar a ligação.

5.1.6 lldelete

A função **lldelete** destrói uma instância de LinkLayer, libertando todo o espaço de memória associado a esta, funcionando como um "destrutor"da estrutura de dados.

5.1.7 lllog

A função **lllog** imprime no ecrã estatísticas após o término da ligação entre o emissor e o recetor.

5.1.8 get_max_message_size

A função **get_max_message_size** retorna o tamanho máximo que pode ter uma trama do tipo I sem as flags necessárias.

6 Protocolo de aplicação

A camada da aplicação é responsável pelas seguintes funcionalidades:

- Envio de pacotes de controlo.
- Envio do ficheiro em questão, sob a forma de pacotes de dados.

6.1 Pacotes do nível de aplicação

A aplicação envia dois tipos de pacotes: **pacotes de controlo** e **pacotes de dados**. Os pacotes de controlo podem ser classificados como inicial ou final, distinguidos através do primeiro byte.

6.2 Envio e recepção dos pacotes

Os pacotes de controlo marcam o início e o fim da transmissão de um ficheiro, enquanto que os pacotes de dados contêm a informação do ficheiro propriamente dita. O envio/receção é efeito pelas funções abaixo:

```
int app_transmitter(int argc, char **argv);
int app_receiver(int argc, char **argv);
```

A função de envio coloca no pacote inicial a informação referente ao ficheiro (nome, tamanho), e envia-o através da função llwrite. A função de recepção recebe este pacote e processa-o. Este processo é repetido para o pacote final. Quanto ao pacote de dados, o processo é semelhante, só que a informação colocada pelo emissor é o tipo de pacote e o seu tamanho.

7 Validação

Para avaliar a fiabilidade do protocolo implementado no que ao envio e recepção de ficheiros toca, foram realizados vários testes.

Foi removido o cabo durante a transferência de informação, sendo novamente reposto ao fim de poucos segundos. A aplicação soube continuar a transferência após a retoma da conexão.

Também experimentámos separar os cabos, esfregar a porta de série com um objeto metálico e ligá-los outra vez. A aplicação soube descartar os pacotes inválidos e retomar a transferência após os cabos serem ligados.

Também experimentámos enviar outro tipo de ficheiros - ficheiros de texto, outras imagens e uma música. Foram todos enviados sem erros.

8 Elementos de valorização

8.1 Selecção de parâmetros pelo utilizador

O utilizador pode, se assim o entender, alterar as definições que entender, recorrendo a *flags*. Abaixo mostramos a mensagem de erro mostrada pela aplicação que contém esta informação:

Usage: ./nserial TRANSMITTER /dev/ttyS<portNo> <filepath> <flags>

Flags: -b [BAUDRATE] - Sets the baudrate.

-t [TIMEOUT] - Sets the timeout before attempting to retransmit (default: 3).

-m [MAXTRIES] - Sets the maximum mumber of tries to transmit a message (default: 3).

-i [MAX_I_FRAME_SIZE] - Sets the maximum I frame size (before stuffing) (default: 255).

8.2 Geração aleatória de erros em tramas I

A aplicação, para cada trama I correntamente recebida simula no receptor a ocorrência de um erro 20% das vezes, e procede como se de um erro real se tratasse.

8.3 Implementação de REJ

Quando ocorre um erro na *flag* **BCC2** na função llread, a *flag* REJ é automaticamente enviada para que o emissor envie a mensagem que não chegou ao recetor corretamente.

8.4 Verificação da integridade dos dados pela aplicação

A aplicação verifica que o tamanho do ficheiro recebido é igual ao tamanho do ficheiro enviado. Também verifica se o pacote recebido pelo recetor era o esperado, terminando a ligação imediatamente caso não seja.

8.5 Registo de ocorrências

A aplicação vai registando ao longo do tempo de execução as ocorrências de *flags* RR, REJ e o número de tramas I recebidas/enviadas, sendo estas estatísticas exibidas após o fim da transmissão de dados.

9 Conclusões

O grupo compreendeu bem todos os pontos pedidos no guião e soube resolver todos os problemas que lhe foram propostos, tendo inclusivé implementado todos os elementos de valorização sugeridos.

O projeto foi dividido em duas camadas: camada de ligação de dados e camada da aplicação. A camada da aplicação é uma camada de alto nível, enquanto que a camada de ligação de dados é uma camada de baixo nível. A camada da aplicação trata de enviar os pacotes de dados e de controlo, enquanto que a camada de ligação de dados trata de enviar as *flags* SET, UA, I, RR, REJ e DISC e de fazer o *byte stuffing* e *byte destuffing*.

Gostaríamos de agradecer à nossa docente pelo apoio prestado nas aulas, sem o qual nos teria sido muito mais díficil e demorado fazer este projeto, nomeadamente no que à compreensão dos vários tipos de *flags* bem como a informação nelas contida.

A realização deste projeto contribuiu para uma melhor compreensão do funcionamento de um protocolo de transferência de dados e do aprofundamento dos conhecimentos em relação à porta de série, que já tinhamos utilizado previamente na unidade curricular de Laboratório de Computadores.

Anexos

A Código Fonte

linklayer.c:

```
#include "linklayer.h"
#include "utils.h"

#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <termios.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

static int tries;
static int timeoutNo;

int is_valid_s_u(const char* string);
int is_a_flag(const char a);
int is_c_flag(const char c);
int is_valid_bcc(const char a, const char c, const char bcc);
int is_valid_combination(const char a, const char c);
int is_valid_i(const char* string, int string_length);

int llopen_receiver(LinkLayer link_layer);
int llopen_transmitter(LinkLayer link_layer);
int llclose_transmitter(LinkLayer link_layer);
int llclose_receiver(LinkLayer link_layer);

struct LinkLayer_t {
```

```
        int fd;
        unsigned int baudrate;
        unsigned int timeout;
        unsigned int max_tries;
        unsigned int max_frame_size;
        int flag; // TRANSMITTER or RECEIVER
        struct termios oldtio;
        unsigned int sequence_number;
        char* buffer;
        int rejNo;
        int iNo;
};

int generate_error(){
        return (rand()%5 == 0);
}

/* Validates string */
int is_valid_string(const char* string, const int string_length){
        if(string_length == 3)
                return is_valid_s_u(string);
        else
                return is_valid_i(string, string_length);
}

/* Validates su flag*/
int is_valid_s_u(const char* string){
        if(!is_a_flag(string[A_FLAG_INDEX])){
                return FALSE;
        }

        if(!is_c_flag(string[C_FLAG_INDEX])){
                return FALSE;
        }
```

```
        if(!is_valid_bcc(string[A_FLAG_INDEX],string[C_FLAG_INDEX],
            string[BCC_FLAG_INDEX])){
                return FALSE;
        }

        return is_valid_combination(string[A_FLAG_INDEX], string[
            C_FLAG_INDEX]);
}

/* Validates a flag */
int is_a_flag(const char a){
        switch(a){
                case 0x01:
                case 0x03:
                        return TRUE;
                default:
                        return FALSE;
        }
}

/* Validates c flag */
int is_c_flag(const char c){
        switch(c){
                case SERIAL_C_SET:
                case SERIAL_C_DISC:
                case SERIAL_C_UA:
                        return TRUE;
                default:
                        return (c|C_FLAG_R_VALUE)==SERIAL_C_RR_N1 ||
                                (c|C_FLAG_R_VALUE)==
                                    SERIAL_C_REJ_N1 ||
                                (c|C_FLAG_R_VALUE)==SERIAL_I_C_N1;
        }
```

```
}

/* Validates BCC Flag */
int is_valid_bcc(const char a, const char c, const char bcc){
        return (a^c) == bcc;
}

/* Validates the string */
int is_valid_combination(const char a, const char c){
        return TRUE;
}

/* Validates I String Header */
int is_valid_i(const char* string, int string_length){
        if(string_length < 5) //Pressupoe que e mandado pelo menos 1
            byte de data
                return FALSE;

        if(!is_a_flag(string[A_FLAG_INDEX])){
                return FALSE;
        }

        if(!is_c_flag(string[C_FLAG_INDEX])){
                return FALSE;
        }

        if(!is_valid_bcc(string[A_FLAG_INDEX],string[C_FLAG_INDEX],
            string[BCC_FLAG_INDEX])){
                return FALSE;
        }

        return TRUE;
}
```

```
int i_valid_bcc2(LinkLayer link_layer, char* buffer, int
    string_length){
        int bcc2 = 0;
        int i;

        for(i = 3; i < string_length - 1; ++i)
                bcc2 ^= buffer[i];

        return bcc2 == buffer[string_length-1];
}

int is_expected_i(LinkLayer link_layer, char* buffer){
        if((buffer[C_FLAG_INDEX] == SERIAL_I_C_N0) && (link_layer->
            sequence_number == 0))
                return TRUE;

        if((buffer[C_FLAG_INDEX] == SERIAL_I_C_N1) && (link_layer->
            sequence_number == 1))
                return TRUE;

        return FALSE;
}

int ua_validator(int flag, char* buffer, int length){
    int aFlag;

    if(flag == TRANSMITTER)
        aFlag = SERIAL_A_ANS_RECEIVER;
    else
        aFlag = SERIAL_A_ANS_TRANSMITTER;

    return      (length == 3) &&
                (buffer[A_FLAG_INDEX] == aFlag) &&
                (buffer[C_FLAG_INDEX] == SERIAL_C_UA);
```

```
}

int set_validator (int flag, char* buffer, int length){
        int aFlag;

        if(flag == TRANSMITTER)
        aFlag = SERIAL_A_COM_RECEIVER;
    else
        aFlag = SERIAL_A_COM_TRANSMITTER;

    return      (length == 3) &&
                (buffer[A_FLAG_INDEX] == aFlag) &&
                (buffer[C_FLAG_INDEX] == SERIAL_C_SET);
}

int disc_validator (int flag, char* buffer, int length){
    int aFlag;

        if(flag == TRANSMITTER)
        aFlag = SERIAL_A_COM_RECEIVER;
    else
        aFlag = SERIAL_A_COM_TRANSMITTER;

    return      (length == 3) &&
                (buffer[A_FLAG_INDEX] == aFlag) &&
                (buffer[C_FLAG_INDEX] == SERIAL_C_DISC);
}

int rr_validator (char* buffer, int length){
    return (length == 3) && ((buffer[C_FLAG_INDEX] == SERIAL_C_RR_N0)
        || (buffer[C_FLAG_INDEX] == SERIAL_C_RR_N1));
}

int rej_validator (char* buffer, int length){
```

```
        return  (length == 3) && ((buffer[C_FLAG_INDEX] ==
            SERIAL_C_REJ_N0) || (buffer[C_FLAG_INDEX] ==
            SERIAL_C_REJ_N1));
}

void sig_alarm_handler(int sig) {
    if (sig == SIGALRM) {
        ++tries;
                ++timeoutNo;
    }
}

void reset_alarm() {
        tries = 0;
}

int read_frame(LinkLayer link_layer) {
        int length = 0;
        char c;
        int res;

        do {
                res = read(link_layer->fd, &c, 1);
                if (res < 1)
                        return -1;
        } while(c != SERIAL_FLAG);

        while (length < 3) {
                length = 0;

                do {
                res = read(link_layer->fd, &c, 1);
                        if (res < 1)
                                return -1;
```

```
                if (c == SERIAL_FLAG)
                        break;

                switch(c){
            case SERIAL_ESCAPE:
                res = read(link_layer->fd, &c ,1);
                if (res < 1)
                        return -1;
                if(c == SERIAL_FLAG_REPLACE)
                    link_layer->buffer[length] = SERIAL_FLAG;
                else if(c == SERIAL_ESCAPE_REPLACE)
                    link_layer->buffer[length] =
                       SERIAL_ESCAPE;
                else
                        return -1;
                break;
            default:
                link_layer->buffer[length] = c;
                break;
            }
                ++length;
            } while(length < link_layer->max_frame_size);

            if (length == link_layer->max_frame_size) {
                do {
                        res = read(link_layer->fd, &c, 1);
                        if (res < 1)
                                return -1;
                } while(c != SERIAL_FLAG);
            }
        }

        return length;
}
```

```
int write_frame(LinkLayer link_layer, char* buffer, unsigned int
    length) {
        unsigned int i;
        int ret = 0;
        char c = SERIAL_FLAG;

        if(write(link_layer->fd,&c,1) < 1)
                return -1;

        /* Byte stuffing */

        for(i = 0; i < length; ++i) {
                 switch(buffer[i]) {
            case SERIAL_FLAG:
            c = SERIAL_ESCAPE;
            if(write(link_layer->fd, &c,1) < 1)
                return ret;
            c = SERIAL_FLAG_REPLACE;
            if(write(link_layer->fd, &c, 1) < 1)
                return ret;
            break;
            case SERIAL_ESCAPE:
            c = SERIAL_ESCAPE;
            if(write(link_layer->fd, &c,1) < 1)
                return ret;
            c = SERIAL_ESCAPE_REPLACE;
            if(write(link_layer->fd, &c,1) < 1)
                return ret;
            break;
            default:
            if(write(link_layer->fd, &buffer[i], 1) < 1)
                return ret;
            break;
```

```
        }
        ++ret;
        }

        c = SERIAL_FLAG;
        if(write(link_layer->fd,&c,1) < 1)
                return -1;

        return ret;
}

LinkLayer llinit(int port, int flag, unsigned int baudrate, unsigned
    int max_tries, unsigned int timeout, unsigned int max_frame_size)
    {
        char port_name[20];
        sprintf(port_name, "/dev/ttyS%d", port);
        struct termios oldtio;
        srand(time(NULL));

        int fd = open(port_name, O_RDWR | O_NOCTTY );
        if (fd < 0) {
                perror(port_name);
                return NULL;
        }

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

        struct termios newtio;
        memset(&newtio, 0, sizeof(newtio));
```

```
    /*       Control, input, output flags */
    newtio.c_cflag = baudrate | CS8 | CLOCAL | CREAD; /*  */
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = OPOST;

    /* set input mode (non-canonical, no echo,...) */
    newtio.c_lflag = 0;

newtio.c_cc[VTIME]    = 0;   /* inter-character timer unused */
newtio.c_cc[VMIN]     = 1;   /* blocking read until 5 chars
   received */


    if(tcflush(fd, TCIFLUSH) != 0)
            return NULL;

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

    /* Open the serial port for sending the message */

struct sigaction sa;
sa.sa_flags = 0;
sa.sa_handler = sig_alarm_handler;
if (sigaction(SIGALRM, &sa, NULL) == -1) {
    perror("Error: cannot handle SIGALRM");
    return NULL;
    }

    if(max_frame_size <= 6){
            printf("Error: Maximum frame size too small\n");
            return NULL;
```

```
        }

        if(max_frame_size > 65536){
                return NULL;
        }

        LinkLayer link_layer = malloc(sizeof(struct LinkLayer_t));
        if(link_layer == NULL)
                return NULL;

        link_layer->fd = fd;
        link_layer->flag = flag;
        link_layer->baudrate = baudrate;
        link_layer->timeout = timeout;
        link_layer->max_tries = max_tries;
        link_layer->max_frame_size = max_frame_size;
        link_layer->sequence_number = 0;
        link_layer->buffer = malloc((max_frame_size-2)*sizeof(char));
        link_layer->oldtio = oldtio;
        link_layer->rejNo = 0;
        link_layer->iNo = 0;

        if(link_layer->buffer == NULL){
                free(link_layer);
                return NULL;
        }

        timeoutNo = 0;

        return link_layer;
}

int llopen(LinkLayer link_layer){
        if(link_layer->flag == TRANSMITTER)
```

```
                return llopen_transmitter(link_layer);
        else if (link_layer->flag == RECEIVER)
                return llopen_receiver(link_layer);
        else
                return -1;
}

int llopen_transmitter(LinkLayer link_layer) {
        char set[] = {SERIAL_A_COM_TRANSMITTER,
        SERIAL_C_SET,
        SERIAL_A_COM_TRANSMITTER^SERIAL_C_SET};

    int length;
    reset_alarm();
    while (tries < link_layer->max_tries) {
        write_frame(link_layer,set,3);
        alarm(link_layer->timeout);
        while (1) {
                length = read_frame(link_layer);
                if (length == -1)
                break;
                if(is_valid_string(link_layer->buffer,length) &&
                    ua_validator(link_layer->flag, link_layer->buffer,
                     length)) {
                alarm(0);
                break;
                }
        }
        if (length != -1)
                break;
    }
        alarm(0);
    if (tries == link_layer->max_tries)
        return -1;
```

```
    return 0;
}

int llopen_receiver(LinkLayer link_layer) {

    char ua[] = {SERIAL_A_ANS_RECEIVER,
       SERIAL_C_UA,
       SERIAL_A_ANS_RECEIVER^SERIAL_C_UA};

    int length;

        alarm(link_layer->max_tries * link_layer->timeout);
    while (tries == 0) {
        length = read_frame(link_layer);
        if (length <= 0)
            continue;
        if(is_valid_string(link_layer->buffer,length) &&
           set_validator(link_layer->flag, link_layer->buffer, length
           ))
           break;
   }
        alarm(0);
        if (length <= 0)
        return -1;

   write_frame(link_layer, ua, 3);
   return 0;
}

int llclose(LinkLayer link_layer){

        if(link_layer->flag == TRANSMITTER)
                llclose_transmitter(link_layer);
```

```
        else if (link_layer->flag == RECEIVER)
                llclose_receiver(link_layer);
        else
                return -1;

        if (tcsetattr(link_layer->fd,TCSANOW,&(link_layer->oldtio))
           == -1) {
            perror("tcsetattr");
            return -1;
    }
    close(link_layer->fd);

    return 0;
}

int llclose_transmitter(LinkLayer link_layer){
    char disc[] = {SERIAL_A_COM_TRANSMITTER,
        SERIAL_C_DISC,
        SERIAL_A_COM_TRANSMITTER^SERIAL_C_DISC};

    char ua[] = {
        SERIAL_A_ANS_TRANSMITTER,
        SERIAL_C_UA,
        SERIAL_A_ANS_TRANSMITTER^SERIAL_C_UA,
    };

    int length;
    reset_alarm();
    while (tries < link_layer->max_tries) {
                write_frame(link_layer,disc,3);
                alarm(link_layer->timeout);
                while (1) {
                length = read_frame(link_layer);
                if (length <= 0)
```

```
                        break;

                    if(is_valid_string(link_layer->buffer,length) &&
                        disc_validator(link_layer->flag, link_layer->
                        buffer, length)) {
                    alarm(0);
                    break;
                    }
            }
            if (length > 0)
                    break;
        }
            alarm(0);
        if (tries == link_layer->max_tries)
            return -1;

        write_frame(link_layer, ua, 3);
        return 0;
}

void lllog(LinkLayer link_layer){
            printf("Number of timeouts occured: %d\n",timeoutNo);
            printf("Number of REJ frames %s: %d\n", link_layer->flag ==
               TRANSMITTER ? "received": "transmitted", link_layer->rejNo
               );
            printf("Number of I frames %s: %d\n", link_layer->flag ==
               RECEIVER ? "received": "transmitted", link_layer->iNo);
}

int llclose_receiver(LinkLayer link_layer) {
        char disc[] = {SERIAL_A_COM_RECEIVER, SERIAL_C_DISC,
           SERIAL_A_COM_RECEIVER^SERIAL_C_DISC};
        int length;
```

```
    reset_alarm();
    alarm(link_layer->max_tries * link_layer->timeout);
while (tries == 0) {
    length = read_frame(link_layer);
    if (length <= 0)
        continue;
    if(is_valid_string(link_layer->buffer,length) &&
       disc_validator(link_layer->flag, link_layer->buffer,
       length))
       break;
}
    alarm(0);
    if (length <= 0)
            return -1;

reset_alarm();
while (tries < link_layer->max_tries) {
            write_frame(link_layer,disc,3);
            alarm(link_layer->timeout);
            while (1) {
                length = read_frame(link_layer);
                if (length <= 0)
                    break;

                if(is_valid_string(link_layer->buffer,length) &&
                   ua_validator(link_layer->flag, link_layer->
                   buffer, length)) {
                    alarm(0);
                    break;
                }
            }
            if (length > 0)
                    break;
}
```

```
    if (tries == link_layer->max_tries)
        return -1;



    return 0;
}

int llread(LinkLayer link_layer, uint8_t *buf){
        int length;
        char ans[3];

        reset_alarm();
        alarm(link_layer->max_tries * link_layer->timeout);
    while (tries == 0) {
        length = read_frame(link_layer);
        if (length <= 0)
            continue;

        if(is_valid_string(link_layer->buffer,length) == FALSE)
                continue;

                if(length == 3) {
                        if(set_validator(link_layer->flag, link_layer
                           ->buffer, length)){
                                ans[0] = SERIAL_A_ANS_RECEIVER;
                                ans[1] = SERIAL_C_UA;
                                ans[2] = ans[0] ^ ans[1];
                                write_frame(link_layer, ans, 3);
                        }
                        continue;
                }

                ++link_layer->iNo;
```

```
    if(i_valid_bcc2(link_layer, link_layer->buffer,length) &&
        is_expected_i(link_layer, link_layer->buffer) && !
       generate_error())
                break;

        ans[0] = SERIAL_A_ANS_RECEIVER;
if(is_expected_i(link_layer, link_layer->buffer)){
        ans[1] = link_layer->sequence_number == 0 ?
           SERIAL_C_REJ_N0 : SERIAL_C_REJ_N1;
                ++link_layer->rejNo;
        }
else
        ans[1] = link_layer->sequence_number == 0 ?
           SERIAL_C_RR_N0 : SERIAL_C_RR_N1;
ans[2] = ans[0] ^ ans[1];

write_frame(link_layer, ans, 3);

}
alarm(0);
if (length <= 0)
        return -1;

memcpy(buf, &(link_layer->buffer[3]), length-4);

ans[0] = SERIAL_A_ANS_RECEIVER;
ans[1] = link_layer->sequence_number == 0 ? SERIAL_C_RR_N1 :
   SERIAL_C_RR_N0;
ans[2] = ans[0] ^ ans[1];

write_frame(link_layer,ans,3);

link_layer->sequence_number = 1 - link_layer->sequence_number
```

```
            ;

        return length-4;
}

int llwrite(LinkLayer link_layer, uint8_t* buf, int length){
        int iLength = length+4;
        char frame[iLength];

        frame[0] = SERIAL_A_COM_TRANSMITTER;
        frame[1] = link_layer->sequence_number == 0 ? SERIAL_I_C_N0 :
            SERIAL_I_C_N1;
        frame[2] = frame[0] ^ frame[1];

        int bufCounter;
        char bcc2 = 0;
        for(bufCounter = 0; bufCounter < length; bufCounter++){
                frame[3+bufCounter] = buf[bufCounter];
                bcc2^=buf[bufCounter];
        }
        frame[iLength-1] = bcc2;

        int ansLength;
        int resend;
        reset_alarm();
    while (tries < link_layer->max_tries) {
                write_frame(link_layer,frame,iLength);
                ++link_layer->iNo;
                resend = FALSE;
        alarm(3);
        while (1) {
                ansLength = read_frame(link_layer);
                if (ansLength <= 0){
                        resend = TRUE;
```

```
            break;
        }

        if(!is_valid_string(link_layer->buffer,ansLength))
            continue;
        if(ansLength != 3)
            continue;
        if(rr_validator(link_layer->buffer, ansLength)) {
                    break;
            }
        if(rej_validator(link_layer->buffer, ansLength)){
                    ++link_layer->rejNo;
            if(link_layer->sequence_number == 0 &&
               link_layer->buffer[C_FLAG_INDEX] ==
               SERIAL_C_REJ_N0){
                    resend = TRUE;
                    break;
            }
            if(link_layer->sequence_number == 1 &&
               link_layer->buffer[C_FLAG_INDEX] ==
               SERIAL_C_REJ_N1){
                    resend = TRUE;
                    break;
            }
            continue;
        }

        }
        if (resend)
            continue;

        if(link_layer->sequence_number == 0 && link_layer->
           buffer[C_FLAG_INDEX] == SERIAL_C_RR_N1)
        break;
```

```
        if(link_layer->sequence_number == 1 && link_layer->buffer[
            C_FLAG_INDEX] == SERIAL_C_RR_N0)
                break;
        }
        alarm(0);

        if (tries == link_layer->max_tries)
        return -1;

    link_layer->sequence_number = 1 - link_layer->sequence_number;
    return length;
}

void lldelete(LinkLayer link_layer) {
        if (link_layer) {
                if (link_layer->buffer)
                        free(link_layer->buffer);
                free(link_layer);
        }
}

int get_max_message_size(LinkLayer link_layer) {
        return link_layer->max_frame_size - 6;
}
```

linklayer.h:

```
#ifndef __LINKLAYER
#define __LINKLAYER

#include <stdint.h>

#define TRANSMITTER 0
#define RECEIVER 1

typedef struct LinkLayer_t * LinkLayer;
```

```
LinkLayer llinit(int port, int flag, unsigned int baudrate, unsigned
    int max_tries, unsigned int timeout, unsigned int max_frame_size);
int llopen(LinkLayer link_layer);
int llwrite(LinkLayer link_layer, uint8_t *buf, int length);
int llread(LinkLayer link_layer, uint8_t *buf);
int llclose(LinkLayer link_layer);
void lldelete(LinkLayer link_layer);
int get_max_message_size(LinkLayer link_layer);
void lllog(LinkLayer link_layer);

#endif // __LINKLAYER
```

applayer.c:

```
#include "linklayer.h"
#include "utils.h"

#include <stdio.h>
#include <fcntl.h>
#include <libgen.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <termios.h>
#include <string.h>
#include <stdint.h>

#define BAUDRATE B38400
#define PACKAGE_DATA 0
#define PACKAGE_START 1
#define PACKAGE_END 2
#define PACKAGE_T_SIZE 0
#define PACKAGE_T_NAME 1
```

```
#define MAX_FILENAME_SIZE 255


typedef struct {
        int fd;
        char name[MAX_FILENAME_SIZE];
        uint32_t size;
} File_info_t;


int get_file_info(File_info_t* file_info, char* filePath){
        int fd = open(filePath,O_RDONLY);

        if(fd == -1)
                return -1;

        struct stat st;
        if(fstat(fd,&st) == -1)
                return -1;

        file_info->fd = fd;
        sprintf(file_info->name, "%s", basename(filePath));
        file_info->size = st.st_size;

        return 0;
}

void print_usage(int flag, char* argv0){
        if(flag == TRANSMITTER)
                printf("Usage: %s TRANSMITTER /dev/ttyS<portNo> <
                    filepath> <flags>\n", argv0);
        else
                printf("Usage: %s RECEIVER /dev/ttyS<portNo> <flags>\
                    n", argv0);
```

```
        printf("Flags: -b [BAUDRATE] - Sets the baudrate.\n");
        printf("       -t [TIMEOUT] - Sets the timeout before
            attempting to retransmit (default: 3).\n");
        printf("       -m [MAXTRIES] - Sets the maximum mumber of
            tries to transmit a message (default: 3). \n");
        printf("       -i [MAX_I_FRAME_SIZE] - Sets the maximum I
            frame size (before stuffing) (default: 255).\n");
}

int parse_baudrate(int baudrate){
        switch(baudrate){
                case 0:
                        return B0;
                case 50:
                        return B50;
                case 75:
                        return B75;
                case 110:
                        return B110;
                case 134:
                        return B134;
                case 150:
                        return B150;
                case 200:
                        return B200;
                case 300:
                        return B300;
                case 600:
                        return B600;
                case 1200:
                        return B1200;
                case 2400:
                        return B2400;
                case 4800:
```

```
                return B4800;
        case 9600:
                return B9600;
        case 19200:
                return B19200;
        case 38400:
                return B38400;
        case 57600:
                return B57600;
        case 115200:
                return B115200;
        case 230400:
                return B230400;
        default:
                return -1;
    }
}

int app_transmitter(int argc, char **argv) {

    int flag = TRANSMITTER;

    if(argc < 4){
        print_usage(flag, argv[0]);
        return 1;
    }

    File_info_t file_info;
    int port = -1;
    int baudrate = BAUDRATE;
    int max_tries = 3;
    int timeout = 3;
    int max_frame_size = 256;
    char* filePath = argv[3];
```

```
sscanf(argv[2],"/dev/ttyS%d",&port);

int arg;
int changeMask[] = {FALSE, FALSE, FALSE, FALSE};
for(arg = 4; arg < argc; ++arg){
        if((arg +1) == argc) {
                printf("Error: Failed to parse flag %s \n",
                   argv[arg]);
                return 1;
        }
        if(strcmp(argv[arg],"-b") == 0)
                        if(changeMask[0] == FALSE){
                                changeMask[0] = TRUE;
                                if(sscanf(argv[++arg], "%d",
                                   &baudrate) != 1){
                                        printf("Error: Unable
                                           to parse baudrate
                                           value \n");
                                        return 1;
                                }
                                baudrate = parse_baudrate(
                                   baudrate);
                                if(baudrate == -1){
                                        printf("Error:
                                           Invalid baudrate
                                           value.\n");
                                        printf("Valid
                                           baudrate values:
                                           0, 50, 75, 110,
                                           134, 150, 200,
                                           300, 600, 1200,
                                           1800, 2400, 4800,
                                           "
```

```
                                                    "
                                                        9600,
                                                        ␣
                                                        19200,
                                                        ␣
                                                        38400,
                                                        ␣
                                                        57600,
                                                        ␣
                                                        115200,
                                                        ␣
                                                        230400\
                                                        n"
                                                        );
                                    return 1;
                                }
                            }
                            else {
                                printf("Error:␣Baudrate␣
                                    defined␣more␣than␣once.␣
                                    First␣defined␣as␣value:␣%d
                                    \n", baudrate);
                                return 1;
                            }
                    else if(strcmp(argv[arg],"-t") == 0)
                            if(changeMask[1] == FALSE){
                                changeMask[1] = TRUE;
                                if(sscanf(argv[++arg], "%d",
                                    &timeout) != 1){
                                    printf("Error␣while␣
                                        parsing␣flag\n");
                                    return 1;
                                }
                            }
```

```
                else {
                        printf("Error: Timeout
                            defined more than once.
                            First defined as value: %d
                            \n", timeout);
                        return 1;
                }
        else if(strcmp(argv[arg],"-m") == 0)
                if(changeMask[2] == FALSE){
                        changeMask[2] = TRUE;
                        if(sscanf(argv[++arg], "%d",
                            &max_tries) != 1){
                                printf("Error while
                                    parsing flag\n");
                                return 1;
                        }
                }
                else {
                        printf("Error: Maximum
                            retransmission tries cap
                            defined more than once.
                            First defined as value %d\
                            n", max_tries);
                        return 1;
                }
        else if(strcmp(argv[arg],"-i") == 0)
                if(changeMask[3] == FALSE){
                        changeMask[3] = TRUE;
                        if(sscanf(argv[++arg], "%d",
                            &max_frame_size) != 1){
                                printf("Error while
                                    parsing flag\n");
                                return 1;
                        }
```

```
                        }
                        else {
                                printf("Error: Maximum I
                                    frame size defined more
                                    than once. First defined
                                    as value %d\n",
                                    max_frame_size);
                                return 1;
                        }

        else {
                printf("Error: Unrecognized flag %s", argv[
                    arg]);
                return 1;
        }

}

if(port == -1){
        printf("Error: Unrecognized port: %s", argv[2]);
        return 1;
}

LinkLayer link_layer = llinit(port, flag, baudrate, max_tries
    , timeout, max_frame_size);

if(link_layer == NULL){
        return 1;
}

if(get_file_info(&file_info, filePath) == -1){
        printf("Error: Can't open file: %s", filePath);
        return 1;
}
```

```
printf("Connecting to receiver... \n");

if (llopen(link_layer) != 0){
        printf("Error: Unable to connect to receiver\n");
        return 1;
}
printf("Connected to receiver\n");

unsigned int segmentSize = get_max_message_size(link_layer) -
    4;
uint8_t segment[segmentSize];

// Build Control package
segment[0] = PACKAGE_START;
segment[1] = PACKAGE_T_NAME;
uint8_t file_name_size = strlen(file_info.name) + 1;
segment[2] = file_name_size;
memcpy(&(segment[3]), file_info.name, file_name_size);
segment[3+file_name_size] = PACKAGE_T_SIZE;
segment[4+file_name_size] = sizeof(uint32_t);
*((uint32_t *)&segment[5+file_name_size]) = file_info.size;

if((9+file_name_size) > segmentSize){
        printf("Error: Maximum frame size too small\n");
        return 1;
}

printf("Sending data...\n");

//Send start
if (llwrite(link_layer, segment, file_name_size + 9) !=
   file_name_size + 9) {
        printf("Error: Failed to send start control package\n
```

```
            ");
        return 1;
}

//Send data
int length;
int sentBytes = 0;
unsigned char sequenceNumber = 0;
do {
        length = read(file_info.fd, &(segment[4]),
            segmentSize);

        if (length > 0) {
                segment[0] = PACKAGE_DATA;
                segment[1] = sequenceNumber;
                segment[2] = (length & 0xFF00) >> 8;
                segment[3] = length & 0xFF;

                if(llwrite(link_layer,segment, length+4) !=
                    length + 4){
                        printf("Error: Failed to send data
                            package\n");
                        return 1;
                }
                sentBytes += length;
                printf("Sending data... %.2f%% done\n",
                    sentBytes * 100.0 / file_info.size);
                ++sequenceNumber;
        }
} while (length > 0);

//Send end
segment[0] = PACKAGE_END;
segment[1] = PACKAGE_T_NAME;
```

```
        segment[2] = file_name_size;
        memcpy(&(segment[3]), file_info.name, file_name_size);
        segment[3+file_name_size] = PACKAGE_T_SIZE;
        segment[4+file_name_size] = sizeof(uint32_t);
        *((uint32_t *)&segment[5+file_name_size]) = file_info.size;

        if (llwrite(link_layer, segment, file_name_size + 9) !=
            file_name_size + 9) {
                printf("Error: Failed to send end control package\n")
                    ;
                return 1;
        }

        printf("Data sent\nClosing connection...\n");


        if (llclose(link_layer) != 0){
                printf("Error: Unable to close connection\n");
                return 1;
        }

        printf("Connection closed\n\n");
        lllog(link_layer);
        lldelete(link_layer);

        if (close(file_info.fd) != 0){
                printf("Error: Unable to close transmitted file\n");
                return 1;
        }

        return 0;
}

int app_receiver(int argc, char **argv) {
```

```
int flag = RECEIVER;

if (argc < 3) {
        print_usage(flag, argv[0]);
        return 1;
}

int port = -1;
int baudrate = BAUDRATE;
int max_tries = 3;
int timeout = 3;
int max_frame_size = 65536;

sscanf(argv[2],"/dev/ttyS%d",&port);

int arg;
int changeMask[] = {FALSE, FALSE, FALSE, FALSE};
for(arg = 3; arg < argc; ++arg){
        if((arg +1) == argc) {
                printf("Error: Parsing flag %s \n", argv[arg
                    ]);
                return 1;
        }
        if(strcmp(argv[arg],"-b") == 0){
                if(changeMask[0] == FALSE){
                        changeMask[0] = TRUE;
                        if(sscanf(argv[++arg], "%d", &
                            baudrate) != 1){
                                printf("Error: Unrecognized
                                    value for flag -b\n");
                                return 1;
                        }
                        baudrate = parse_baudrate(baudrate);
```

```
                    if(baudrate == -1){
                            printf("Error: Invalid
                                baudrate value.\n");
                            printf("Valid baudrate values
                                : 0, 50, 75, 110, 134,
                                150, 200, 300, 600, 1200,
                                1800, 2400, 4800, "
                                        "9600, 19200,
                                            38400,
                                            57600,
                                            115200,
                                            230400\n")
                                            ;
                            return 1;
                    }

            }
            else {
                    printf("Error: Baudrate defined more
                        than once. First defined as value:
                        %d\n", baudrate);
                    return 1;
            }
    }
    else if(strcmp(argv[arg],"-t") == 0){
            if(changeMask[1] == FALSE){
                    changeMask[1] = TRUE;
                    if(sscanf(argv[++arg], "%d", &timeout
                        ) != 1){
                            printf("Unrecognized value
                                for flag -t\n");
                            return 1;
                    }
```

```
                }
                else {
                        printf("Error: Timeout defined more
                            than once. First defined as value:
                            %d\n", timeout);
                        return 1;
                }
        }
        else if(strcmp(argv[arg],"-m") == 0){
                if(changeMask[2] == FALSE){
                        changeMask[2] = TRUE;
                        if(sscanf(argv[++arg], "%d", &
                            max_tries) != 1){
                                printf("Unrecognized value
                                    for flag -m\n");
                                return 1;
                        }

                }
                else {
                        printf("Error: Maximum transmission
                            tries cap defined more than once.
                            First defined as value %d\n",
                            max_tries);
                        return 1;
                }
        }
        else {
                printf("Error: Unrecognized flag %s\n", argv[
                    arg]);
                return 1;
        }
}
```

```
if(port == -1){
        printf("Error: Unrecognized port: %s\n", argv[2]);
        return 1;
}

LinkLayer link_layer = llinit(port, flag, baudrate, max_tries
    , timeout, max_frame_size);

if(link_layer == NULL){
        return 1;
}

printf("Connecting to transmitter... \n");

if (llopen(link_layer) != 0){
        printf("Error: Unable to connect to transmitter\n");
        return 1;
}

printf("Receiving data... \n");
// Read start
unsigned int maxSegmentLength = get_max_message_size(
    link_layer);
uint8_t startSegment[maxSegmentLength];
uint8_t segment[maxSegmentLength];
int segmentLength = llread(link_layer, startSegment);
int startSegmentLength = segmentLength;

if (segmentLength <= 0) {
        printf("Error: Failed to read start control package\n
            ");
        return 1;
}
```

```
int i;

File_info_t file_info;

if(startSegment[0] == PACKAGE_START){
        i=1;
        while(i < startSegmentLength){
                char type = startSegment[i];
                unsigned char size = startSegment[i+1];
                switch(type){
                        case PACKAGE_T_SIZE:
                                file_info.size = *((uint32_t
                                    *) &startSegment[i+2]);
                                break;
                        case PACKAGE_T_NAME:
                                memcpy(file_info.name,&
                                    startSegment[i+2],size);
                                break;
                }
                i += 2 + size;
        }
}
else {
        printf("Error: Start package missing.\n");
        return 1;
}

// Create file
file_info.fd = creat(file_info.name, 0666);
uint32_t reported_size = 0;
uint8_t sequenceNumber = 0;
while (1) {
        segmentLength = llread(link_layer, segment);
```

```
if (segmentLength <= 0) {
        printf("Error: Failed to read data package.\n
            ");
        return 1;
}
//check end conditions
if (segment[0] == PACKAGE_END)
        break;

if (segment[0] == PACKAGE_START){
        printf("Error: Received duplicate start
            package\n");
        return 1;
}

if (segment[0] == PACKAGE_DATA) {// copy to file
        if (segmentLength < 4) {
                printf("Error: Wrong size for segment
                    \n");
                return 1;
        }
        if (segment[1] != sequenceNumber) {
                printf("Error: Package out of order\n
                    ");
                return 1;
        }

        uint16_t package_data_size = segment[2] << 8
            | segment[3];
        if (package_data_size != segmentLength - 4) {
                printf("Error: Reported package size
                    and received size differ\n");
                printf("Package data size: %d\n",
                    package_data_size);
```

```
                        printf("Segment Length - 4 %d\n", (
                            segmentLength - 4));
                        return 1;
                    }


                    write(file_info.fd, &(segment[4]),
                        package_data_size);
                    reported_size += package_data_size;
                    printf("Receiving data... %.2f%% done\n",
                        reported_size * 100.0 / file_info.size);
                    ++sequenceNumber;
                }
                else {
                    printf("Error: Received unknown package type\
                        n");
                    return 1;
                }
            }


            if(reported_size != file_info.size){
                printf("Error: Reported size (%d bytes) differs from
                    expected size (%d bytes)\n", reported_size,
                    file_info.size);
            }

            printf("Data received\n");
            printf("Transfered file: %s\n", file_info.name);

            printf("Closing connection...\n");
            if (llclose(link_layer) != 0){
                printf("Error: Unable to close the connection\n");
                return 1;
```

```
        }

        printf("Connection closed\n\n");

        lllog(link_layer);
        lldelete(link_layer);

        if (close(file_info.fd) != 0){
                printf("Error: Unable to close received file\n");
                return 1;
        }

        return 0;
}

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

        if (argc == 1)
        {
                print_usage(TRANSMITTER, argv[0]);
                print_usage(RECEIVER, argv[0]);
                return 1;
        }

        if (strcmp(argv[1], "TRANSMITTER") == 0)
                return app_transmitter(argc, argv);
        else if (strcmp(argv[1], "RECEIVER") == 0)
                return app_receiver(argc, argv);
        else
        {
                print_usage(TRANSMITTER, argv[0]);
                print_usage(RECEIVER, argv[0]);
                return 1;
        }
```

```
}
```

utils.h:

```
#ifndef __UTILS
#define __UTILS

#define SERIAL_FLAG 0x7E
#define SERIAL_ESCAPE 0x7D
#define SERIAL_FLAG_REPLACE 0x5E
#define SERIAL_ESCAPE_REPLACE 0x5D

#define SERIAL_A_COM_TRANSMITTER 0x03
#define SERIAL_A_ANS_RECEIVER 0x03
#define SERIAL_A_COM_RECEIVER 0x01
#define SERIAL_A_ANS_TRANSMITTER 0x01
#define SERIAL_C_SET 0x07
#define SERIAL_C_DISC 0x0B
#define SERIAL_C_UA 0x03
#define SERIAL_C_RR_N1 0x21
#define SERIAL_C_RR_N0 0x01
#define SERIAL_C_REJ_N0 0x05
#define SERIAL_C_REJ_N1 0x25
#define SERIAL_I_C_N0    0x00
#define SERIAL_I_C_N1    0x20

#define SERIAL_SU_STRING_SIZE    5
#define MAX_STRING_SIZE 255

#define A_FLAG_INDEX     0
#define C_FLAG_INDEX     1
#define BCC_FLAG_INDEX   2

#define C_FLAG_R_VALUE   0x20
```

```
#define RR0             0
#define RR1     1
#define I0              2
#define I1              3

#define TRUE 1
#define FALSE 0

#endif /* __UTILS */
```

Makefile:

```
all:
        gcc -Wall -o nserial app_layer.c link_layer.c

clean:
        rm nserial
```