RCOM 2020/2021

Relatório - 1º Trabalho laboratorial

Eduardo Brito, up201806271
Pedro Ferreira, up201806506
Turma 1, Grupo 11

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Sumário

Este relatório documenta a jornada de trabalho, na unidade curricular de RCOM, em torno do desenvolvimento de um protocolo de ligação de dados e o fornecimento de um serviço de comunicação de dados fiável entre dois sistemas ligados por um meio (canal) de transmissão – neste caso, um cabo série. O ambiente de desenvolvimento estabeleceu-se com o sistema operativo LINUX, a linguagem de programação C e as portas série RS-232 (de comunicação assíncrona), assim como respetivos Drivers e funções da API disponibilizadas pelo sistema operativo.

Como conclusões, retiradas da concretização deste trabalho, são de destacar a consolidação efetiva da aprendizagem no campo dos protocolos de transferência de informação, a implementação bem sucedida do mecanismo por detrás dos mesmos, incidindo em conceitos como *Stop & Wait, framing, byte stuffing, destuffing,* entre outros, e o amadurecimento no que toca à organização do trabalho, à estruturação do código, à validação teórica das formulações e de tudo o resto que envolveu a implementação deste projeto.

Introdução

Os principais objetivos do trabalho prendem-se com a implementação de um protocolo de ligação de dados capaz de fornecer, às camadas lógicas que de si dependem, um serviço de comunicação resiliente a falhas que conceba a ligação necessária à transmissão de dados entre dois sistemas. O protocolo integrou uma aplicação simples de transferência de ficheiros, a qual permitiu testar a sua fiabilidade e eficiência, recorrendo a funções de uma API por este disponibilizada. Genericamente, de protocolos de ligação de dados como o implementado, esperam-se funções de organização e sincronismo de tramas, de estabelecimento e terminação das ligações, de numeração das tramas, confirmação de receção, rejeição de tramas, controlo de erros e de fluxo. Neste trabalho, todas essas noções foram implementadas com sucesso e este relatório explicará o seu processo.

As secções seguintes documentam as várias componentes do projeto, desde a Arquitetura idealizada, as Estruturas de dados, os Casos de uso principais e os dois Protocolos, terminando, por fim, com a Validação e Eficiência do Protocolo de ligação de dados e principais Conclusões do trabalho. As primeiras secções contém informação sobre os blocos funcionais e interfaces criadas, sobre as estruturas de dados concebidas, as funções disponibilizadas e as relações, e organização sequencial, entre os diversos componentes do código. No final, fazer-se-á uma breve descrição dos testes de validação efetuados sobre o protocolo implementado e uma caracterização estatística da sua eficiência.

Arquitetura e Estrutura do Código

O nosso projeto segue uma estrutura de organização baseada em módulos. São claramente distinguidas as interfaces e as respetivas implementações. As primeiras correspondem aos *header files*, que permitem criar uma camada mais elevada de abstração e fornecer aos seus utilizadores apenas os blocos do código necessários, os quais se encontram nos *source files*, num modelo interior que funciona como uma *black box*.

A camada de ligação de dados engloba a maioria dos módulos disponibilizados, sendo eles:

- O módulo *datalink.h,* que contém as funções principais enunciadas - *llopen, llwrite*, *llread* e *llclose*.
- Os módulos *sframe.h* e *iframe.h*, que representam as máquinas de estado associadas à avaliação das tramas com formato de supervisão e informação, respetivamente.
- O módulo *utils.h*, que engloba algumas funções auxiliares, constantes de configuração e as principais estruturas de dados usadas pelos outros módulos.

A camada da aplicação engloba os restantes módulos, que se servem das funções disponibilizadas pelos módulos anteriores, sendo eles:

- O módulo *sender.h*, que, além de disponibilizar a interface do utilizador para envio dos dados, contém também as funções de criação e envio dos pacotes de controlo e dos pacotes de dados.
- O módulo *receiver.h*, que, além de disponibilizar a interface do utilizador para receção dos dados, contém também as funções de receção e confirmação dos pacotes de controlo e dos pacotes de dados.

O seguinte diagrama explicita o que foi acima enunciado, relativamente à organização modular do nosso código e aos vários componentes de cada camada de abstração.

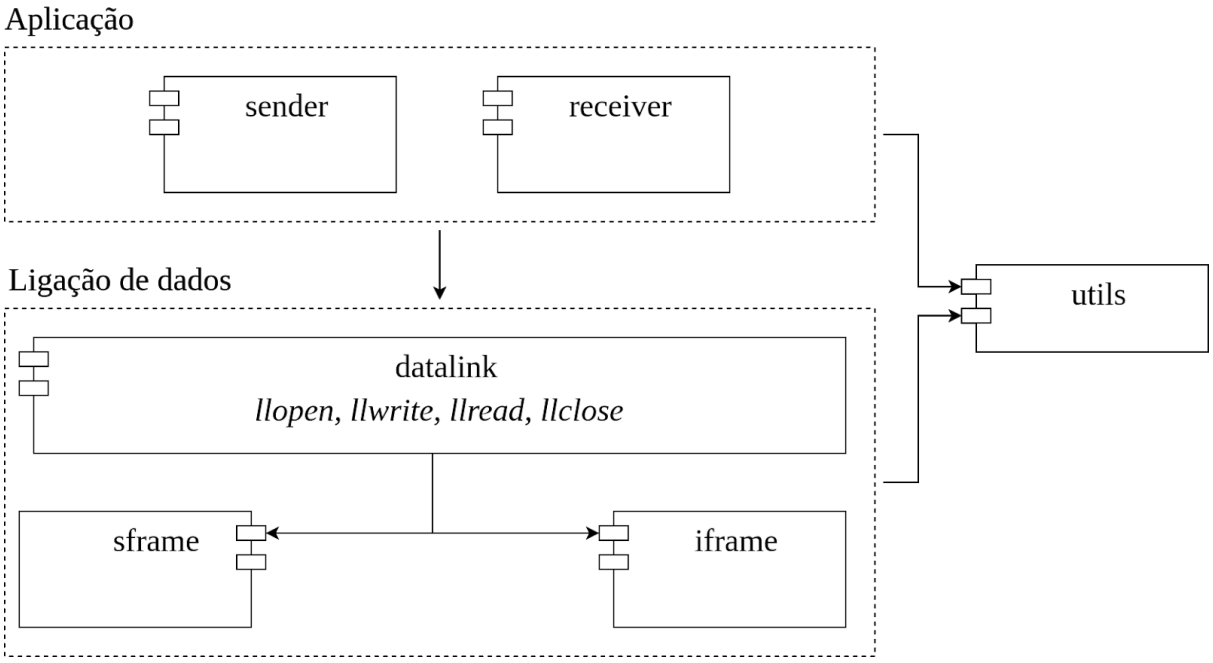


Figura 1. - Visão geral sobre a Arquitetura do projeto.

Estruturas de dados

Neste trabalho, para guardar as informações necessárias de forma simples e compreensível foram utilizadas algumas estruturas de dados que achamos convenientes. Todas elas se encontram no ficheiro utils.h. As estruturas utilizadas foram:

- Uma *enum* **user**, de forma a saber se nos encontramos na presença do emissor, ou do recetor. Esta estrutura é utilizada pelos vários módulos, definindo, durante a execução, o papel do processo na transmissão dos dados, permitindo chamar determinadas funções, ou induzir determinado comportamento, no decorrer do programa.

```
typedef enum // User Type
{
    SENDER,
    RECEIVER,
} user;
```

Figura 2. Estrutura **user**

- A *enum* **fstate**, que engloba os possíveis estados utilizados nas máquinas de estado, definindo o estado atual do **user** aquando da receção e avaliação das tramas, tanto de informação como de supervisão. Estes estados permitem determinar a condição atual dos bytes recebidos e avançar, retroceder, descartar, ou fazer corresponder determinado comportamento, à medida que as tramas vão chegando à entidade que as recebe.

```
typedef enum // Frame States
{
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    RR_DUP,
    BCC1_OK,
    DATA_RCV,
    ESC_RCV,
    BCC2_REJ,
    STOP
} fstate;
```

```
ate iframe_getState(unsigned char input, pframe *t)
{
    switch (t->state)
    {
    case START:
        return iframe_startState(input, t);
    case FLAG_RCV:
        return iframe_flagState(input, t);
    case A_RCV:
        return iframe_aState(input, t);
    case C_RCV:
        return iframe_cState(input, t);
    case BCC1_OK:
        return iframe_dataState(input, t);
    case DATA_RCV:
        return iframe_dataState(input, t);
    case ESC_RCV:
        return iframe_escState(input, t);
    case STOP:
        return STOP;
    default:
        return iframe_startState(input, t);
    }
}
```

```
fstate sframe_getState(unsigned char input, pframe *t)
{
    switch (t->state)
    {
    case START:
        return sframe_startState(input, t);
    case FLAG_RCV:
        return sframe_flagState(input, t);
    case A_RCV:
        return sframe_aState(input, t);
    case C_RCV:
        return sframe_cState(input, t);
    case BCC1_OK:
        return sframe_bccState(input, t);
    case STOP:
        return STOP;
    default:
        return sframe_startState(input, t);
    }
}
```

Figura 3. Estrutura **fstate** e os motores principais das máquinas de estados.

- Uma *struct* **pframe** que armazena, em runtime, informações sobre o **user**, a trama e seus valores expectáveis (flag, a, c, bcc e bcc2), o estado atual da máquina de estados, a informação sobre a porta, o número de tentativas restantes e o número de sequência atual. Armazena, ainda, uma *array de bytes*, com os dados da trama atual, e o tamanho desses dados, e, por fim, uma *struct*, que contém as configurações da porta série, usada nas funções *llopen e llclose*. Esta *struct* **pframe** pode ser usada tanto pelo emissor, como pelo recetor, fazendo, cada processo, a sua própria gestão dos recursos e dos campos desta estrutura comum.

```
typedef struct // Protocol Frame Struct
{
    user u;
    unsigned char flag1;
    unsigned char a;
    unsigned char expected_a;
    unsigned char c;
    unsigned char expected_c;
    unsigned char bcc;
    unsigned char bcc2;
    unsigned char flag2;
    fstate state;
    int port;
    unsigned int num_retr;
    unsigned int seqnumber;
    unsigned char *buffer;
    unsigned int length;
    struct termios *oldtio;
} pframe;
```

Figura 4. Estrutura **pframe**

Casos de uso

O fornecimento de um serviço de comunicação de dados fiável entre dois sistemas ligados por um meio (canal) de transmissão – neste caso, o cabo série - é garantido através da ponte feita entre os dois módulos criados. Usando-o, o utilizador consegue transferir dados entre as duas máquinas, isto é, o processo emissor, iniciado com o comando `./sender.o -p <port> <filename>`, e o recetor, iniciado com o comando `./receiver.o -p <port>`. O que acontece, de seguida, é o estabelecimento da ligação, a efetiva transmissão e receção dos dados e, por fim, a terminação e conclusão da ligação, tudo isto contemplado num conjunto de funções que são enumeradas nos próximos tópicos.

Protocolo de ligação de dados

Toda a implementação deste módulo *datalink.h* se baseia num mesmo mecanismo de escrita e leitura da porta série, com a criação das tramas, o seu envio, e a sua posterior receção e confirmação através de máquinas de estados implementadas nos módulos correspondentes *sframe.h* e *iframe.h*. As tramas são criadas e enviadas, e feito o *byte stuffing* nas tramas de informação, através das funções:

```
int send_sframe(int fd, unsigned char A, unsigned char C)
int send_iframe(int fd, int ns, unsigned char *buffer, int length)
```

- As funções principais *llopen, llread, llwrite e llclose* são responsáveis, respetivamente:
  - pela abertura e configuração da porta: `int llopen(int port, user u)`
  - pela receção, leitura e validação das tramas de informação (exclusivamente por parte do recetor): `int llread(int port, unsigned char *buffer)`
  - pelo envio das tramas de informação e validação da receção comunicada pelo recetor (exclusivamente por parte do emissor): `int llwrite(int port, unsigned char *buffer, int length)`
  - pela comunicação final e envio das tramas DISC, correspondendo ao final da transmissão de dados e encerramento do canal: `int llclose(int port)`

As tramas são recebidas byte a byte para serem confirmadas nas máquinas de estado, sendo o processo, em todas as funções *llopen*, *llwrite*, *llread*, *llclose*, bastante semelhante ao loop da Figura 5. A máquina de estados avalia cada input e retorna o novo estado, fazendo também o *destuffing no caso das tramas de informação*, através das funções:

```
fstate sframe_getState(unsigned char input, pframe *t)
fstate iframe_getState(unsigned char input, pframe *t)
```

```
while (t->state != STOP)
{
    unsigned char input;
    int res = read(t->port, &input, 1);

    if (res < 0)
    {
        logpf(printf("DTL #### Could not read from serial port.\n"));
        perror("Error: ");
        return -1;
    }
    else if (res == 0 && t->u == SENDER)
    {
        if (t->num_retr > 0)
        {
            if (send_sframe(t->port, A1, SET) == -1) // SENDER sends SET message to RECEIVER again
                return -1;
            t->num_retr--;
        }
        else if (t->num_retr <= 0)
        {
            logpf(printf("DTL ##### No answer received. Ending port connection.\n"));
            return -1;
        }
    }
    else
    {
        t->num_retr = MAX_RETR;
    }

    t->state = sframe_getState(input, t);
}
```

Figura 5. loop de recepção das tramas.

Destacamos, também, o mecanismo de timeout que é implementado com recurso às configurações da porta série, nomeadamente, os campos VTIME e VMIN, fazendo todas chamadas read retornar com 0 (VMIN) apenas no caso de terem passado VTIME * 0.1 segundos. Estas configurações encontram-se no módulo *utils.h* e são estabelecidas na função *llopen*.

Protocolo de aplicação

A camada de aplicação é representada por dois ficheiros distintos, correspondendo um ao emissor e outro ao recetor. Falando, primeiramente, do ficheiro correspondente ao emissor (*sender.c*), encontraremos três funções:

- A função *main*, correspondendo à função principal do ficheiro, responsável pela análise dos argumentos passados, aquando da inicialização do processo, bem como da transmissão dos dados para a camada inferior (ligação de dados). Esta transferência é feita com a ajuda das duas seguintes funções.
- A função

  ```
  int send_ctrl_packet(int ctrl_type, int fd, long int filesize, char *filename)
  ```

  que está encarregue de criar e escrever na porta de série um pacote de controlo. Esta função é chamada uma vez, no princípio da emissão, e outra no final, correspondendo, respetivamente, ao pacote *start* e ao pacote *end*.
- A função

  ```
  int send_data_packet(int fd, int nr, unsigned char *data, int length)
  ```

  que, tal como o nome indica, se encarrega de criar e escrever na porta de série os pacote de dados, sendo chamada até não haver mais dados do ficheiro a ser transferido.

O ficheiro correspondente ao recetor (*receiver.c*), apresenta quatro funções que, ao receber os dados provenientes da camada de ligação de dados, os analisam e guardam num novo ficheiro, cópia do original. Deste modo, as funções presentes nele são:

- Tal como no emissor, uma função *main* que, além de validar os argumentos passados, é responsável pela leitura dos pacotes provenientes da porta série. Para obter as informações enviadas pelo emissor, de forma simplista, são utilizadas as três funções auxiliares de obtenção de dados.
- As funções

  ```
  int get_ctrl_packet_filesize(unsigned char *buffer)
  unsigned char *get_ctrl_packet_filename(unsigned char *buffer, unsigned char *filename)
  int get_data_packet_size(unsigned char * buffer, int nr, int lread)
  ```

  Estas três funções são utilizadas para obtenção da informação correspondente ao tamanho total e ao nome do ficheiro, nos pacotes de controlo, e ao tamanho de cada pacote de dados a receber.

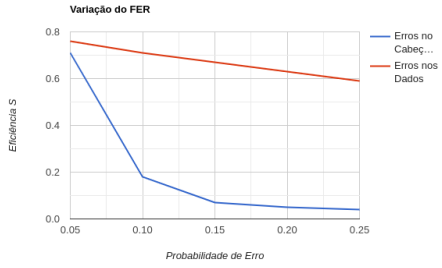
Validação

Foram realizados vários testes de validação dos resultados obtidos. Entre eles, destacam-se os testes ordinários de transferência dos mais variados ficheiros, como imagens, PDFs, vídeos e até ficheiros compostos apenas por bytes do tipo ESC e FLAG, com a finalidade de testar o mecanismo de byte stuffing e destuffing do protocolo. Outros testes mais elaborados foram também conduzidos, nomeadamente, testes para analisar a eficiência, variando diferentes campos do protocolo, e testes para validar o envio de mensagens REJ, ou deteção de duplicados. Para confirmar, detetar e analisar os resultados, um mecanismo de logging foi criado, baseando-se na duplicação de descritores de ficheiros, com o armazenamento das mensagens numa pasta logs, a criar pelo utilizador da nossa aplicação, caso pretenda o relatório das mensagens guardado em ficheiros. Ao longo da transferência, é possível também visualizar a percentagem total de dados enviados e recebidos, até completar o total do ficheiro.

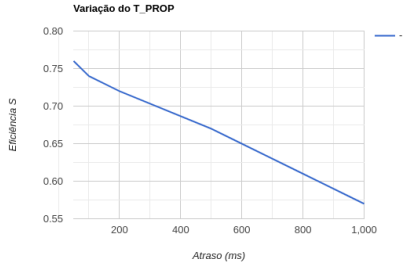
Eficiência do protocolo de ligação de dados

A caracterização estatística da eficiência S (FER, a) foi feita através de medições e testes de variação de determinadas componentes que permitiram validar as fórmulas teóricas estudadas. Seguindo as sugestões do guião deste projeto, apresentam-se, de seguida, os testes de eficiência efetuados. Em anexo, é possível rever as tabelas que originam estes gráficos.

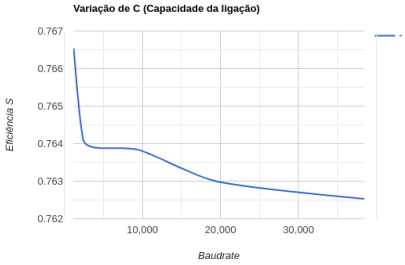
- Figura 6. Variação do FER - através da geração aleatória de erros em tramas de informação, tanto no cabeçalho, como no campo de dados. O seu impacto na eficiência do protocolo é notório, sobretudo pelos erros no cabeçalho, que são ignorados pelo recetor e levam ao *timeout* no emissor. No entanto, a validade também é verificada com os erros no Campo de Dados, onde o tempo de execução é afetado apenas pela probabilidade e não pelo *timeout*. Neste teste, a *baudrate* (38400 bit/s), o tamanho das tramas (256 bytes) e o tamanho do ficheiro (10968 bytes) mantiveram-se constantes, fazendo variar, apenas, a probabilidade de erro.



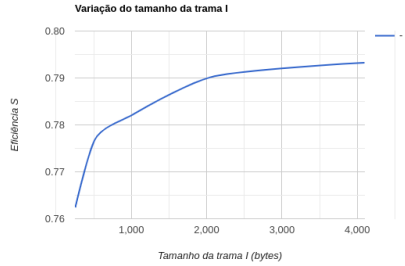
- Figura 7. Variação do T_PROP - através da geração de atraso de propagação simulado. Neste teste, a *baudrate* (38400 bit/s), o tamanho das tramas (256 bytes) e o tamanho do ficheiro (10968 bytes) mantiveram-se constantes, fazendo variar, apenas, o tempo de atraso. Como previsto, o decréscimo foi aproximadamente linear já que o tempo deixa de depender de erros ocasionais e é mais afetado pelo atraso na receção da informação.



- Figura 8. Variação de C (Capacidade da ligação) - Neste teste, o tamanho das tramas (256 bytes) e o tamanho do ficheiro (10968 bytes) mantiveram-se constantes, fazendo variar, apenas, a *baudrate*. Apesar do intervalo de valores acentuar certas diferenças, verifica-se que a Eficiência diminui com o aumento da Capacidade de Ligação, já que o tamanho das tramas não sofre diferença.



- Figura 9. Variação do tamanho da trama I - Neste teste, a *baudrate* (38400 bit/s) e o tamanho do ficheiro (10968 bytes) mantiveram-se constantes, fazendo variar, apenas, o tamanho das tramas. Pela análise dos valores, percebe-se a tendência dos números, à medida que o tamanho da trama aumenta e se aproxima da capacidade da ligação, ganhando na eficiência.



Conclusões

Com a concretização deste trabalho, consolidou-se, efetivamente, a aprendizagem no campo dos protocolos de transferência de informação. Conceitos como *Stop & Wait*, *framing*, *byte stuffing*, *destuffing*, entre outros, foram absorvidos com sucesso e notámos um amadurecimento no que toca à organização dos trabalhos, à estruturação do código, à validação teórica das formulações e de tudo o resto que envolveu a implementação deste projeto. Como verificado anteriormente, existem vários fatores que podem interferir com a eficiência do protocolo de ligação de dados, aspetos comprovados através dos múltiplos testes efetuados, que permitiram a verificação teórica de toda a matéria envolvida neste trabalho.

Anexos

utils.h

```
#include <termios.h>

#ifndef UTILS_H
#define UTILS_H

#define BAUDRATE B38400
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FALSE 0
#define TRUE 1

#define SVTIME 30
#define SVMIN 0
#define MAX_RETR 3

#define FLAG 0x7E
#define A1 0x03
#define A2 0x01
#define SET 0x03
#define UA 0x07
#define DISC 0x0B
#define CI(n) (n << 6)
#define RR(n) (n << 7 | 0b101)
#define REJ(n) (n << 7 | 0b1)
#define ESC 0x7d
#define EFLAG 0x5e
#define EESC 0x5d

#define DATAP 1
#define STARTP 2
#define ENDP 3

#define FILE_SIZEP 0
#define FILE_NAMEP 1

#define MAX_SIZE 256 // Needs to be greater than 4
```

```
#define MAX_SIZEP MAX_SIZE - 4

typedef enum // User Type
{
   SENDER,
   RECEIVER,
} user;

typedef enum // Frame States
{
   START,
   FLAG_RCV,
   A_RCV,
   C_RCV,
   RR_DUP,
   BCC1_OK,
   DATA_RCV,
   ESC_RCV,
   BCC2_REJ,
   STOP
} fstate;

typedef struct // Protocol Frame Struct
{
   user u;
   unsigned char flag1;
   unsigned char a;
   unsigned char expected_a;
   unsigned char c;
   unsigned char expected_c;
   unsigned char bcc;
   unsigned char bcc2;
   unsigned char flag2;
   fstate state;
   int port;
   unsigned int num_retr;
   unsigned int seqnumber;
   unsigned char *buffer;
   unsigned int length;
   struct termios *oldtio;
} pframe;
```

```
/**
* Parses the user console args
* @param argc
* @param argv
* @param port to be initialized with the port number given
* @param filename to be initialized with the filename given
* @return port number, or -1 if an error occurred
*/
int parse_args(int argc, char **argv, int *port, char *filename);

/**
* Shows a user message in the screen
* @param stdout file descriptor
* @param filename the name of the file being uploaded/downloaded
* @param total current number of bytes uploaded/downloaded
* @param size file size
* @param action "Uploading" or "Downloading" expected string
* @return 0 if success, an error otherwise
*/
int send_user_message(int stdout, char *filename, int total, int
size, char *action);

/**
* Flushes stdout pending output
* @param printf wrapper for printf calls
* @return 0 if success, an error otherwise
*/
int logpf(int printf);

/**
* Simulates an error based on some probability
* @param p probability
* @return 1 if error, 0 otherwise
*/
int prob(double p);

#endif
```

utils.c

```
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <unistd.h>
#include <string.h>
#include "utils.h"

int parse_args(int argc, char **argv, int *port, char *filename)
{
   char *p = NULL;
   int index, c;

   opterr = 0;

   while ((c = getopt(argc, argv, "p:")) != -1)
       switch (c)
       {
       case 'p':
           p = optarg;
           break;
       case '?':
           if (optopt == 'p')
                   fprintf(stderr, "APP ##### Option -%c requires an
argument.\n", optopt);
           else if (isprint(optopt))
                fprintf(stderr, "APP ##### Unknown option `-%c'.\n",
optopt);
           else
               fprintf(stderr,
                                "APP ##### Unknown option character
`\\x%x'.\n",
                       optopt);
           fflush(stdout);
           return -1;
       default:
           return -1;
       }
```

```
    if (p != NULL)
        *port = atoi(p);
    else
        return -1;

    if (filename != NULL)
        strcpy(filename, argv[optind]);

    return atoi(p);
}

int send_user_message(int stdout, char *filename, int total, int
size, char *action)
{
    char str[256];
    float percentage = (float)(1.0 * total / size) * 100.0;
     int l = sprintf(str, "\r%s %s     |     Please Wait... %.0f%%",
action, filename, percentage);

    if (total == size)
    {
         l = sprintf(str, "\r%s %s     |     Complete %.0f%%      \n",
action, filename, percentage);
    }
    write(stdout, str, l);

    return 0;
}

int logpf(int printf)
{
    fflush(stdout);
    return 0;
}

int prob(double p){
    double prob = (double) rand() / RAND_MAX;
    if(prob < p) return 1;
    return 0;
}
```

sframe.h

```
#include <termios.h>
#include "utils.h"

#ifndef SFRAME_H
#define SFRAME_H

/**
* Inits the State Machine
* @param port port file descriptor
* @param u user Type
* @param t pframe struct to be initialized
* @return pframe pointer to struct
*/
pframe *sframe_init_stm(int port, user u, pframe *t);

/**
* Start State for STM
* @param input read from port
* @param t frame struct
* @return current fstate
*/
fstate sframe_startState(unsigned char input, pframe *t);

/**
* Flag State for STM
* @param input read from port
* @param t frame struct
* @return current fstate
*/
fstate sframe_flagState(unsigned char input, pframe *t);

/**
* A State for STM
* @param input read from port
* @param t frame struct
* @return current fstate
*/
fstate sframe_aState(unsigned char input, pframe *t);
```

```
/**
* C State for STM
* @param input read from port
* @param t frame struct
* @return current fstate
*/
fstate sframe_cState(unsigned char input, pframe *t);

/**
* BCC State for STM
* @param input read from port
* @param t frame struct
* @return current fstate
*/
fstate sframe_bccState(unsigned char input, pframe *t);

/**
* Gets current STM state
* @param input read from port
* @param t frame struct
* @return current fstate
*/
fstate sframe_getState(unsigned char input, pframe *t);

#endif
```

sframe.c

```
#include <stdlib.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <termios.h>
#include <string.h>
#include "sframe.h"

pframe *sframe_init_stm(int port, user u, pframe *t)
{
   if (t == NULL)
   {
       t = malloc(sizeof(pframe));
       t->oldtio = malloc(sizeof(struct termios));
       t->seqnumber = 0;
   }
   t->state = START;
   t->u = u;
   t->port = port;
   t->num_retr = MAX_RETR;
   return t;
}

fstate sframe_startState(unsigned char input, pframe *t)
{
   if (input == FLAG)
   {
       t->flag1 = input;
       return FLAG_RCV;
   }
   return START;
}

fstate sframe_flagState(unsigned char input, pframe *t)
{
   if (input == t->expected_a)
   {
       t->a = input;
```

```
        return A_RCV;
    }
    else if (input == FLAG)
        return FLAG_RCV;
    return START;
}

fstate sframe_aState(unsigned char input, pframe *t)
{
    if (input == t->expected_c)
    {
        t->c = input;
        return C_RCV;
    }
    else if (input == RR(t->seqnumber))
        return RR_DUP;
    else if (input == REJ(t->seqnumber))
        return BCC2_REJ;
    else if (input == FLAG)
        return FLAG_RCV;
    return START;
}

fstate sframe_cState(unsigned char input, pframe *t)
{
    if (input == (t->a ^ t->c))
    {
        t->bcc = input;
        return BCC1_OK;
    }
    else if (input == FLAG)
        return FLAG_RCV;
    return START;
}

fstate sframe_bccState(unsigned char input, pframe *t)
{
    if (input == FLAG)
    {
        t->flag2 = input;
        return STOP;
```

```
    }
    return START;
}

fstate sframe_getState(unsigned char input, pframe *t)
{
    switch (t->state)
    {
    case START:
        return sframe_startState(input, t);
    case FLAG_RCV:
        return sframe_flagState(input, t);
    case A_RCV:
        return sframe_aState(input, t);
    case C_RCV:
        return sframe_cState(input, t);
    case BCC1_OK:
        return sframe_bccState(input, t);
    case STOP:
        return STOP;
    default:
        return sframe_startState(input, t);
    }
}
```

iframe.h

```
#include <termios.h>
#include "utils.h"

#ifndef IFRAME_H
#define IFRAME_H

/**
* Inits the Information State Machine
* @param port port file descriptor
* @param u user Type
* @param t pframe struct to be initialized
* @return pframe pointer to struct
*/
pframe *iframe_init_stm(int port, user u, pframe *t);

/**
* Start State for STM
* @param input read from port
* @param t frame struct
* @return current fstate
*/
fstate iframe_startState(unsigned char input, pframe *t);

/**
* Flag State for STM
* @param input read from port
* @param t frame struct
* @return current fstate
*/
fstate iframe_flagState(unsigned char input, pframe *t);

/**
* A State for STM
* @param input read from port
* @param t frame struct
* @return current fstate
*/
fstate iframe_aState(unsigned char input, pframe *t);
```

```
/**
* C State for STM
* @param input read from port
* @param t frame struct
* @return current fstate
*/
fstate iframe_cState(unsigned char input, pframe *t);

/**
* DATA State for STM
* @param input read from port
* @param t frame struct
* @return current fstate
*/
fstate iframe_dataState(unsigned char input, pframe *t);

/**
* ESC State for STM
* @param input read from port
* @param t frame struct
* @return current fstate
*/
fstate iframe_escState(unsigned char input, pframe *t);

/**
* Gets current STM state
* @param input read from port
* @param t frame struct
* @return current fstate
*/
fstate iframe_getState(unsigned char input, pframe *t);

#endif
```

iframe.c

```
#include <stdlib.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <termios.h>
#include <string.h>
#include "iframe.h"

pframe *iframe_init_stm(int port, user u, pframe *t)
{
   if (t == NULL)
   {
       t = malloc(sizeof(pframe));
       t->seqnumber = 0;
   }
   if (t->buffer != NULL)
       free(t->buffer);
    t->buffer = malloc((MAX_SIZE + 1) * 2 * sizeof(unsigned char));
// Double for all-ESC case and + 1 for BCC2
   t->length = 0;
   t->bcc2 = 0;

   t->state = START;
   t->u = u;
   t->port = port;
   t->num_retr = MAX_RETR;
   return t;
}

fstate iframe_startState(unsigned char input, pframe *t)
{
   t->length = 0;
   if (input == FLAG)
   {
       t->flag1 = input;
       return FLAG_RCV;
   }
   return START;
```

```
}

fstate iframe_flagState(unsigned char input, pframe *t)
{
   if (input == A1)
   {
       t->a = input;
       return A_RCV;
   }
   else if (input == FLAG)
       return FLAG_RCV;
   return START;
}

fstate iframe_aState(unsigned char input, pframe *t)
{
   if (input == t->expected_c)
   {
       t->c = input;
       return C_RCV;
   }
   else if (input == CI(!t->seqnumber))
       return RR_DUP;
   else if (input == FLAG)
       return FLAG_RCV;
   return START;
}

fstate iframe_cState(unsigned char input, pframe *t)
{
   if (input == (t->a ^ t->c))
   {
       t->bcc = input;
       return BCC1_OK;
   }
   else if (input == FLAG)
       return FLAG_RCV;
   return START;
}

fstate iframe_dataState(unsigned char input, pframe *t)
```

```
{
   if (input == ESC)
   {
       return ESC_RCV;
   }
   else if (input == FLAG)
   {
       unsigned char bcc2 = t->buffer[t->length - 1];
         t->bcc2 ^= bcc2; // Reverting the last XOR that was bcc2
itself
       t->flag2 = input;
       t->length--;

       if (bcc2 != t->bcc2)
       {
           t->bcc2 = 0;
           return BCC2_REJ;
       }

       return STOP;
   }

   t->buffer[t->length] = input;
   t->bcc2 ^= input;
   t->length++;

   return DATA_RCV;
}

fstate iframe_escState(unsigned char input, pframe *t)
{
   if (input == EFLAG)
   {
       t->buffer[t->length] = FLAG;
       t->bcc2 ^= FLAG;
   }
   else if (input == EESC)
   {
       t->buffer[t->length] = ESC;
       t->bcc2 ^= ESC;
   }
```

```
    t->length++;
    return DATA_RCV;
}

fstate iframe_getState(unsigned char input, pframe *t)
{
    switch (t->state)
    {
    case START:
        return iframe_startState(input, t);
    case FLAG_RCV:
        return iframe_flagState(input, t);
    case A_RCV:
        return iframe_aState(input, t);
    case C_RCV:
        return iframe_cState(input, t);
    case BCC1_OK:
        return iframe_dataState(input, t);
    case DATA_RCV:
        return iframe_dataState(input, t);
    case ESC_RCV:
        return iframe_escState(input, t);
    case STOP:
        return STOP;
    default:
        return iframe_startState(input, t);
    }
}
```

datalink.h

```
#include "utils.h"

#ifndef APP_H
#define APP_H

/**
* Opens the transmission at port for the user type u
* @param port to use for the transmission
* @param u user type
* @return -1 if an error occurred, or port file descriptor
*/
int llopen(int port, user u);

/**
* Writes length chars from buffer to port
* @param port to use for the transmission
* @param buffer to be sent to port
* @param length size of buffer
* @return -1 if an error occurred, or total bytes written
*/
int llwrite(int port, unsigned char * buffer, int length);

/**
* Reads from port to buffer
* @param port to use for the transmission
* @param buffer to be filled with bytes read
* @return -1 if an error occurred, or total bytes read
*/
int llread(int port, unsigned char *buffer);

/**
* Closes the transmission at port
* @param port where was established the transmission
* @return -1 if an error occurred, success otherwise
*/
int llclose(int port);
#endif
```

datalink.c

```
#include <stdlib.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>
#include <string.h>
#include "sframe.h"
#include "iframe.h"
#include "datalink.h"

pframe *t;

int send_sframe(int fd, unsigned char A, unsigned char C)
{
   unsigned char a[5] = {FLAG, A, C, A ^ C, FLAG};
   int res = write(fd, a, 5);
   if (res <= 0)
   {
       logpf(printf("DTL ##### Could not write to serial port.\n"));
       perror("Error: ");
       return -1;
   }
    logpf(printf("DTL ##### SFRAME (C=%x, BCC1=%x) \tSENT\n", C, A ^
C));

   return res;
}

int send_iframe(int fd, int ns, unsigned char *buffer, int length)
{
   unsigned char C = CI(ns);
   unsigned char BCC2 = 0;

   unsigned char a[4] = {FLAG, A1, C, A1 ^ C};
   int res = write(fd, a, 4);
   if (res <= 0)
   {
```

```
        logpf(printf("DTL ##### Could not write to serial port.\n"));
        perror("Error: ");
        return -1;
    }

    size_t i;
    for (i = 0; i < length; i++)
    {
        BCC2 ^= buffer[i];

        if (buffer[i] == FLAG)
        {
            a[0] = ESC;
            a[1] = EFLAG;
            res = write(fd, a, 2);
        }
        else if (buffer[i] == ESC)
        {
            a[0] = ESC;
            a[1] = EESC;
            res = write(fd, a, 2);
        }
        else
        {
            res = write(fd, &buffer[i], 1);
        }

        if (res <= 0)
        {
                logpf(printf("DTL ##### Could not write to serial
port.\n"));
            perror("Error: ");
            return -1;
        }
    }

    if (BCC2 == FLAG)
    {
        a[0] = ESC;
        a[1] = EFLAG;
        a[2] = FLAG;
```

```
        res = write(fd, a, 3);
    }
    else if (BCC2 == ESC)
    {
        a[0] = ESC;
        a[1] = EESC;
        a[2] = FLAG;
        res = write(fd, a, 3);
    }
    else
    {
        a[0] = BCC2;
        a[1] = FLAG;
        res = write(fd, a, 2);
    }

    if (res <= 0)
    {
        logpf(printf("DTL ##### Could not write to serial port.\n"));
        perror("Error: ");
        return -1;
    }

     logpf(printf("DTL ##### IFRAME (Ns=%d, C=%x, BCC2 %x) \tSENT\n",
t->seqnumber, C, BCC2));

    if (t->buffer != NULL)
        free(t->buffer);
    t->buffer = malloc(length * sizeof(unsigned char));

    for (int i = 0; i < length; i++)
    {
        t->buffer[i] = buffer[i];
    }

    t->length = length;

    return length;
}

int llopen(int port, user u)
```

```
{
   char portname[12];
   sprintf(portname, "/dev/ttyS%d", port);

   int fd = open(portname, O_RDWR | O_NOCTTY);
   if (fd < 0)
   {
       perror(portname);
       return -1;
   }

   t = sframe_init_stm(fd, u, t);
   struct termios newtio;

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

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

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

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

   tcflush(fd, TCIOFLUSH);

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

   logpf(printf("DTL ##### New termios structure set\n"));
```

```
   logpf(printf("DTL ##### Starting port connection...\n"));

   t->expected_a = A1;
   if (t->u == SENDER)
   {
        if (send_sframe(t->port, A1, SET) == -1) // SENDER sends SET
message to RECEIVER
           return -1;

           t->expected_c = UA; // SENDER expects UA message from
RECEIVER
   }
   else if (t->u == RECEIVER)
         t->expected_c = SET; // RECEIVER expects SET message from
SENDER

   while (t->state != STOP)
   {
       unsigned char input;
       int res = read(t->port, &input, 1);

       if (res < 0)
       {
                logpf(printf("DTL ##### Could not read from serial
port.\n"));
           perror("Error: ");
           return -1;
       }
       else if (res == 0 && t->u == SENDER)
       {
           if (t->num_retr > 0)
           {
                 if (send_sframe(t->port, A1, SET) == -1) // SENDER
sends SET message to RECEIVER again
                  return -1;
               t->num_retr--;
           }
           else if (t->num_retr <= 0)
           {
                  logpf(printf("DTL ##### No answer received. Ending
port connection.\n"));
```

```
                return -1;
            }
        }
        else
        {
            t->num_retr = MAX_RETR;
        }

        t->state = sframe_getState(input, t);
    }

     logpf(printf("DTL ##### SFRAME (C=%x, BCC1=%x) \tRECEIVED\n",
t->c, t->bcc));
    if (t->u == RECEIVER)
    {
         if (send_sframe(t->port, A1, UA) == -1) // RECEIVER sends UA
message to SENDER
            return -1;
    }

    return fd;
}

int llwrite(int port, unsigned char *buffer, int length)
{

    t = sframe_init_stm(port, SENDER, t);

     int total = send_iframe(port, t->seqnumber, buffer, length); //
SENDER sends IFRAME to RECEIVER
    t->expected_a = A1;
     t->expected_c = RR(!t->seqnumber); // SENDER expects RR message
from RECEIVER

    if (total < 0)
    {
                logpf(printf("DTL ##### Aborting llwrite after
send_iframe.\n"));
        return -1;
    }
```

```
   while (t->state != STOP)
   {
       unsigned char input;
       int res = read(t->port, &input, 1);

       if (res < 0)
       {
                logpf(printf("DTL ##### Could not read from serial
port.\n"));
           perror("Error: ");
           return -1;
       }
       else if (res == 0)
       {
           if (t->num_retr > 0)
           {
                    total = send_iframe(port, t->seqnumber, buffer,
length); // SENDER sends IFRAME to RECEIVER again

               if (total < 0)
               {
                      logpf(printf("DTL ##### Aborting llwrite after
send_iframe.\n"));
                   return -1;
               }

               t->num_retr--;
           }
           else if (t->num_retr <= 0)
           {
                  logpf(printf("DTL ##### No answer received. Ending
port connection.\n"));
               return -1;
           }
       }

       t->state = sframe_getState(input, t);

       if (t->state == BCC2_REJ)
       {
```

```
                logpf(printf("DTL ##### BCC2 REJECTED (Nr=%d, C=%x)
\tRECEIVED\n", t->seqnumber, REJ(t->seqnumber)));
             total = send_iframe(port, t->seqnumber, buffer, length);
// SENDER sends IFRAME to RECEIVER again

           if (total < 0)
           {
                    logpf(printf("DTL ##### Aborting llwrite after
send_iframe.\n"));
               return -1;
           }

           t->num_retr--;
           if (t->num_retr <= 0)
           {
                  logpf(printf("DTL ##### No correct answer received
after many BCC2_REJ. Ending port connection.\n"));
               return -1;
           }
       }
       else if (t->state == RR_DUP)
       {
                    logpf(printf("DTL ##### DUP RR (Nr=%d, C=%x)
\tRECEIVED\n", t->seqnumber, RR(t->seqnumber)));
           t->state = START;
       }
   }

    logpf(printf("DTL ##### RR (Nr=%d, C=%x, BCC1=%x) \tRECEIVED\n",
!t->seqnumber, t->c, t->bcc));
   t->seqnumber = !t->seqnumber;

   return total;
}

int llread(int port, unsigned char *buffer)
{
   t = iframe_init_stm(port, RECEIVER, t);

   t->expected_a = A1;
```

```
   t->expected_c = CI(t->seqnumber); // RECEIVER expects IFRAME from
SENDER

   logpf(printf("DTL ##### Reading from port.\n"));

   while (t->state != STOP)
   {
       unsigned char input;
       int res = read(t->port, &input, 1);

       if (res < 0)
       {
                logpf(printf("DTL ##### Could not read from serial
port.\n"));
           perror("Error: ");
           return -1;
       }

       t->state = iframe_getState(input, t);

       if (t->state == BCC2_REJ)
       {
            logpf(printf("DTL ##### BCC2 REJECTED (C=%x) \tSENT\n",
REJ(t->seqnumber)));
           if (send_sframe(t->port, A1, REJ(t->seqnumber)) == -1) //
RECEIVER sends REJ message to SENDER
               return -1;
           t->state = START;
       }
       else if (t->state == RR_DUP)
       {
                logpf(printf("DTL ##### DUP RR (C=%x) \tSENT\n",
RR(t->seqnumber)));
            if (send_sframe(t->port, A1, RR(t->seqnumber)) == -1) //
RECEIVER sends DUP message to SENDER
               return -1;
           t->state = START;
       }
   }
```

```
      logpf(printf("DTL ##### BCC2 SUCCESSFULLY (C=%2x, BCC2=%x)
RECEIVED\n", t->c, t->bcc2));
   t->seqnumber = !t->seqnumber;

   if (send_sframe(t->port, A1, RR(t->seqnumber)) == -1) // RECEIVER
sends RR message to SENDER
       return -1;

   memcpy(buffer, t->buffer, t->length);

   return t->length;
}

int llclose(int port)
{
   t = sframe_init_stm(port, t->u, t);
    t->expected_c = DISC; // SENDER/RECEIVER expects DISC message
from RECEIVER/SENDER

   logpf(printf("DTL ##### Closing port connection...\n"));

   if (t->u == SENDER)
   {
         if (send_sframe(t->port, A1, DISC) == -1) // SENDER sends
DISC message to RECEIVER
           return -1;

       t->expected_a = A2;
   }

   while (t->state != STOP)
   {
       unsigned char input;
       int res = read(t->port, &input, 1);

       if (res < 0)
       {
                logpf(printf("DTL ##### Could not read from serial
port.\n"));
           perror("Error: ");
           return -1;
```

```
        }
        else if (res == 0)
        {
            if (t->num_retr > 0 && t->u == SENDER)
            {
                  if (send_sframe(t->port, A1, DISC) == -1) // SENDER
sends DISC message to RECEIVER again
                    return -1;
                t->num_retr--;
            }
            else if (t->num_retr <= 0)
            {
                   logpf(printf("DTL ##### No answer received. Ending
port connection.\n"));
                return -1;
            }
        }
        else
        {
            t->num_retr = MAX_RETR;
        }

        t->state = sframe_getState(input, t);
    }

    if (t->u == RECEIVER)
    {
          logpf(printf("DTL ##### DISC (C=%x, BCC1=%x) \tRECEIVED\n",
t->c, t->bcc));

        t = sframe_init_stm(port, t->u, t);

         if (send_sframe(t->port, A2, DISC) == -1) // RECEIVER sends
DISC message to SENDER
            return -1;

           t->expected_c = UA; // RECEIVER expects UA message from
SENDER
        t->expected_a = A2;

        while (t->state != STOP)
```

```
        {
            unsigned char input;
            int res = read(t->port, &input, 1);

            if (res < 0)
            {
                   logpf(printf("DTL ##### Could not read from serial
port.\n"));
                perror("Error: ");
                return -1;
            }
            else if (res == 0)
            {
                if (t->num_retr > 0)
                {
                         if (send_sframe(t->port, A2, DISC) == -1) //
RECEIVER sends DISC message to SENDER again
                        return -1;
                    t->num_retr--;
                }
                else if (t->num_retr <= 0)
                {
                          logpf(printf("DTL ##### No answer received.
Ending port connection.\n"));
                    return -1;
                }
            }
            else
            {
                t->num_retr = MAX_RETR;
            }

            t->state = sframe_getState(input, t);
        }

          logpf(printf("DTL ##### UA (C=%x, BCC1=%x) \tRECEIVED\n",
t->c, t->bcc));
    }
    else
    {
```

```
        logpf(printf("DTL ##### DISC (C=%x, BCC1=%x) \tRECEIVED\n",
t->c, t->bcc));

        if (send_sframe(t->port, A2, UA) == -1) // SENDER sends UA
message to RECEIVER
           return -1;
   }

   if (tcsetattr(port, TCSANOW, t->oldtio) != 0)
   {
       perror("tcsetattr: ");
       return -1;
   }

   free(t->oldtio);
   free(t);

   logpf(printf("DTL ##### Ending transmittion.\n"));

   if (close(port) != 0)
   {
       perror("close: ");
       return -1;
   }

   return 0;
}
```

receiver.c

```
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include "datalink.h"
#include "receiver.h"

int get_ctrl_packet_filesize(unsigned char *buffer)
{
 if (buffer[1] == FILE_SIZEP)
 {
   int l1 = buffer[2]; // FILE SIZE string size in chars
   unsigned char filesize_s[l1];
   for (int i = 0; i < l1; i++)
   {
     filesize_s[l1 - i - 1] = buffer[i + 3];
   }

   return atoi(filesize_s);
 }

 return 0;
}

unsigned char *get_ctrl_packet_filename(unsigned char *buffer,
unsigned char *filename)
{
 int l1 = buffer[2]; // FILE SIZE string size in chars
 if (buffer[3 + l1] == FILE_NAMEP)
 {
   int l2 = buffer[4 + l1]; // FILE NAME string size in chars
   int i;
   for (i = 0; i < l2; i++)
   {
     filename[i] = buffer[5 + l1 + i];
   }
```

```
   filename[i] = '_'; // TEST

   return filename;
 }

 return NULL;
}

int get_data_packet_size(unsigned char * buffer, int nr, int lread){
 unsigned char C = buffer[0];
 unsigned char N = buffer[1];
 unsigned char L2 = buffer[2];
 unsigned char L1 = buffer[3];

 int l2 = (int) L2;
 int l1 = (int) L1;
 int l = 256 * l2 + l1;

 if(C != DATAP || (int) N != nr || l != lread)
   return -1;
  return l;
}

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

 int port;

 if (argc < 2 || parse_args(argc, argv, &port, NULL) < 0)
 {
       logpf(printf("Usage:\t./receiver.o   -p   serialport   \n\tex:
./receiver.o -p 11\n"));
   fflush(stdout);
   return -1;
 }

 int fd = 0, file = 0, logs = 0, old_stdout = dup(STDOUT_FILENO);

 logs = open("logs/r.log", O_WRONLY | O_CREAT | O_TRUNC, 0777);
 dup2(logs, STDOUT_FILENO);
```

```
 if ((fd = llopen(port, RECEIVER)) <= 0)
 {
   logpf(printf("APP ##### Failed at llopen on port %d.\n", port));
   return -1;
 }

 unsigned char ctrl[300] = {0};
 unsigned char filename[256] = {0};
 unsigned char *buffer = malloc(MAX_SIZE * sizeof(unsigned char));
 int l = 0;
 int filesize;

 int ctrl_start = 0, ctrl_end = 0;

 while (!ctrl_start)
 {
   l = llread(fd, ctrl);
   if (l <= 0)
   {
       logpf(printf("APP ##### Failed at llread when receiving ctrl
packet start.\n"));
     free(buffer);
     return -1;
   }

   if (ctrl[0] == STARTP)
   {

     ctrl_start = 1;

     if ((filesize = get_ctrl_packet_filesize(ctrl)) <= 0)
     {
          logpf(printf("APP ##### Failed at get ctrl packet start
filesize %d.\n", filesize));
       ctrl_start = 0;
     }

     if (get_ctrl_packet_filename(ctrl, filename) == NULL)
     {
```

```
          logpf(printf("APP ##### Failed at get ctrl packet start
filename %s.\n", filename));
       ctrl_start = 0;
     }
   }
 }

 if ((file = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0777)) <
0)
 {
   logpf(printf("APP ##### Failed to open file to be written.\n"));
   free(buffer);
   return -1;
 }

 long int total = 0;
 int w = 0, nr = 0;

 while (total < filesize)
 {
   int l = llread(fd, buffer);
   int datasize = l - 4;

    if (l < 0 || get_data_packet_size(buffer, nr % 255, datasize) <
0) // -4 represents the app packet header
   {
       logpf(printf("APP ##### Failed at llread when receiving data
packet.\n"));
     free(buffer);
     return -1;
   }

   unsigned char aux[datasize];

   for (size_t i = 4; i < l; i++)
   {
     aux[i - 4] = buffer[i];
   }

   int w = 0;
```

```
   if ((w = write(file, aux, datasize)) < datasize)
   {
     logpf(printf("APP ##### Error when writing to new file.\n"));
   }

   total += w;
   nr += 1;

   if(logs > 0)
          send_user_message(old_stdout,  filename,  total,  filesize,
"Downloading");
 }

 while (!ctrl_end)
 {
   l = llread(fd, ctrl);
   if (l <= 0)
   {
       logpf(printf("APP ##### Failed at llread when receiving ctrl
packet end.\n"));
     free(buffer);
     return -1;
   }

   if (ctrl[0] == ENDP)
   {

     ctrl_end = 1;

     if (get_ctrl_packet_filesize(ctrl) != filesize)
     {
           logpf(printf("APP  #####  Failed  at  get  ctrl  packet  end
filesize %d.\n", filesize));
       ctrl_end = 0;
     }

     unsigned char filename_aux[256] = {0};
          if  (strncmp(get_ctrl_packet_filename(ctrl,  filename_aux),
filename, strlen(filename)) != 0)
     {
```

```
            logpf(printf("APP ##### Failed at get ctrl packet end
filename %s.\n", filename));
        ctrl_end = 0;
      }

    }
  }

  free(buffer);

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

  fflush(stdout);
  dup2(old_stdout, STDOUT_FILENO);

  return 0;
}
```

sender.c

```
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/stat.h>
#include "datalink.h"
#include "sender.h"

int send_ctrl_packet(int ctrl_type, int fd, long int filesize, char
*filename)
{

 int numbers = 0;
 int filename_size = strlen(filename);
 long int aux = filesize;

 while (aux != 0)
 {
   aux /= 10;
   ++numbers; // number of chars in filesize
 }

 int total = 5 + numbers + filename_size; // ctrl packet total size

 unsigned char *buffer = malloc(total * sizeof(unsigned char));

 buffer[0] = ctrl_type;
 buffer[1] = FILE_SIZEP;
 buffer[2] = numbers;

 int i = 0;
 while (filesize != 0)
 {
   aux = filesize % 10;
```

```
    filesize /= 10;
    buffer[i + 3] = aux + '0'; // saving filesize as string of chars
    i++;
  }

  i += 3;
  buffer[i] = FILE_NAMEP;
  i++;
  buffer[i] = filename_size;
  i++;

  int j;

  for (j = 0; j < filename_size; j++)
  {
    buffer[i + j] = filename[j]; // saving filename as string of
chars
  }

  if (llwrite(fd, buffer, total) <= 0)
  {
    logpf(printf("APP ##### Failed at llwrite when sending ctrl
packet %d.\n", ctrl_type));
    free(buffer);
    return -1;
  }

  free(buffer);

  return total;
}

int send_data_packet(int fd, int nr, unsigned char *data, int
length)
{

  int l1 = length % 256;
  int l2 = length / 256;

  int total = 4 + length; // data packet total size
```

```
 unsigned char *buffer = malloc(total * sizeof(unsigned char));

 buffer[0] = DATAP;
 buffer[1] = nr;
 buffer[2] = l2;
 buffer[3] = l1;

 for (int i = 0; i < length; i++)
 {
   buffer[i + 4] = data[i];
 }

 if (llwrite(fd, buffer, total) <= 0)
 {
   logpf(printf("APP ##### Failed at llwrite when sending data
packet nr %d.\n", nr));
   free(buffer);
   return -1;
 }

 free(buffer);

 return total;
}

int main(int argc, char **argv)
{
 int port;
 char filename[256] = {0};

 if (argc < 3 || parse_args(argc, argv, &port, filename) < 0)
 {
   logpf(printf("Usage:\t./sender.o -p serialport filename \n\tex:
./sender.o -p 10 /tests/p.gif\n"));
   return -1;
 }

 int fd = 0, file = 0, logs = 0, old_stdout = dup(STDOUT_FILENO);

 logs = open("logs/s.log", O_WRONLY | O_CREAT | O_TRUNC, 0777);
 dup2(logs, STDOUT_FILENO);
```

```
 if ((fd = llopen(port, SENDER)) < 0)
 {
   logpf(printf("APP ##### Failed at llopen on port %d.\n", port));
   return -1;
 }

 if ((file = open(filename, O_RDONLY)) < 0)
 {
   logpf(printf("APP ##### Failed to open file to be sent.\n"));
   return -1;
 }

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

 if (send_ctrl_packet(STARTP, fd, size, filename) <= 0)
 {
   logpf(printf("APP ##### Failed to send ctrl packet %d.\n",
STARTP));
   return -1;
 }

 long int total = 0;
 int nr = 0;

 while (total < size)
 {
   unsigned char buffer[MAX_SIZEP];
   int l = MAX_SIZEP > (size - total) ? size - total : MAX_SIZEP;
   int r = 0;
   if ((r = read(file, buffer, l)) < l)
   {
     logpf(printf("APP ##### Error when reading from file to be
sent.\n"));
   }

   total += r;

   if (send_data_packet(fd, nr % 255, buffer, r) <= 0)
```

```
   {
     logpf(printf("APP ##### Failed to send data packet %d.\n", nr %
255));
     return -1;
   }

   nr++;

   if(logs > 0)
     send_user_message(old_stdout, filename, total, size,
"Uploading");
 }

 if (send_ctrl_packet(ENDP, fd, size, filename) <= 0)
 {
   logpf(printf("APP ##### Failed to send ctrl packet %d.\n",
ENDP));
   return -1;
 }

 close(file);
 if (llclose(fd) < 0)
 {
   logpf(printf("APP ##### Error with llclose.\n"));
   return -1;
 }

 fflush(stdout);
 dup2(old_stdout, STDOUT_FILENO);

 return 0;
}
```

Tabelas

- Variação do FER

Probabilidade de Erro	Eficiência S - Erros no Cabeçalho com *timeout de* 3 s	Eficiência S - Erros no Campo de Dados
0.05	3.2 s => S = 0.71	3.01 s => S = 0.76
0.1	13.0 s => S = 0.18	3.20 s => S = 0.71
0.15	35.1 s => S = 0.07	3.43 s => S = 0.67
0.2	42.2 s => S = 0.05	3.61 s => S = 0.63
0.25	46.1 s => S = 0.04	3.82 s => S = 0.59

- Variação do T_PROP

Atraso	T_PROP	Eficiência S
50 ms	2.962 s	0.76
100 ms	3.049 s	0.74
200 ms	3.133 s	0.72
500 ms	3.411 s	0.67
1000 ms	3.997 s	0.57

- Variação de C (Capacidade da ligação)

Baudrate	Tempo decorrido	Eficiência S
1200	95.4 s	0.76654
2400	47.8 s	0.76412
9600	11.9 s	0.76383
19200	5.98 s	0.76301
38400	2.99 s	0.76253

- Variação do tamanho da trama I

Tamanho da trama I	Tempo decorrido	Eficiência S
256 bytes	3.0 s	0.76242
512 bytes	2.94 s	0.77673
1024 bytes	2.92 s	0.78224
2048 bytes	2.89 s	0.79015
4096 bytes	2.88 s	0.79327