U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Redes de Computadores

Relatório do 1.º Trabalho Prático Laboratorial

Mestrado Integrado em Engenharia Informática e Computação 2020/2021

Autores:

Diogo Oliveira Reis, up201405015

João Pereira da Silva Matos, up201703884

Sumário

Este relatório foi realizado no âmbito da unidade curricular de Redes de Computadores e serve de complemento ao primeiro trabalho prático laboratorial, cuja essência é a transferência de dados usando uma porta de série, seguindo uma série de protocolos.

O trabalho foi realizado com sucesso, todos os protocolos foram implementados e é possível transferir ficheiros de grandes dimensões e com velocidades variáveis sem perda de informação.

Introdução

O objetivo do primeiro trabalho prático no âmbito da unidade curricular de Redes e Computadores foi a realização de uma aplicação capaz de transferir ficheiros de um computador para o outro através de uma porta de série.

O objetivo deste relatório é expor e explicar a componente teórica que serviu de base à realização da aplicação. A estrutura do relatório é a seguinte:

- Arquitetura: apresentação dos blocos de início de execução, tram, link_layer, app_layer e state_machine e das funções usadas para a interação entre eles.
- Estrutura do código: explicação sobre as structs criadas e a sua utilidade para os programas e apresentação detalhada de funções relevantes.
- Casos de uso principais: explicação da utilidade da aplicação e descrição do fluxo de execução do programa.
- Protocolo de ligação lógica: descrição sobre o modo como está implementado o protocolo de ligação da dados.
- Protocolo de aplicação: descrição sobre o modo como está implementado o protocolo da aplicação.
- Validação: Descrição dos testes realizados, variação do tamanho das tramas de informação e taxa de transmissão de dados.
- Eficiência do protocolo de ligação de dados: análise da eficiência do protocolo.
- Conclusão: resumo de toda a informação apresentada e considerações finais.
- Anexo 1: Código fonte.
- Anexo 2: Código fonte usado para medição da eficiência dos protocolos.
- Anexo 3: Gráficos que mostram a variação do tempo de transferência com os parâmetros pedidos

Arquitetura

O trabalho está dividido em dois programas, um para o emissor (a execução deste programa inicia-se em **writenoncanonical.c**) e outro para o recetor (aqui a execução começa no ficheiro **noncanonical.c**). Estes programas interagem com dois blocos/interfaces:

- Funções ll exigidas pelo enunciado e as suas funções auxiliares (ficheiros **link_layer.c** e **link_layer.h**)
    - Interface: llopen, llwrite (apenas usada pelo emissor), llread (apenas usada pelo receptor) e llclose.
- Funções para processamento do ficheiro e pacotes do nível da aplicação (ficheiros **app_layer.c** e **app_layer.h**)
    - Interface: generate_data_packet, generate_control_packet, extract_size_name, extract_seq_size_data, readFile, restoreFile, processFile.

Relativamente a estes blocos, o segundo não faz uso de código de outros blocos, sendo as suas funções usadas apenas pelo bloco onde se inicia a execução.

O bloco das funções ll faz uso de funções auxiliares que pertencem a esse mesmo bloco. Para além disso, usa os seguintes blocos:

- Geração e processamento de tramas (ficheiros **tram.c** e **tram.h**)
    - Interface: setup_initial_values, generate_info_tram, generate_su_tram, parse_info_tram, process_info_tram_received, byte_stuff, byte_unstuff e parse_and_process_su_tram.
- Recepção de tramas (ficheiros **state_machine.c** e **state_machine.h**)
    - Interface: receive_tram e receive_info_tram.

Estes blocos só são chamados pelo conjunto das funções ll e, ou só interagem com outras funções no seu bloco, ou nem isso fazem. No entanto, as variáveis globais são declaradas no ficheiro **tram.h** e são usadas pelos vários blocos que constituem os programas.

Estrutura do código

Estruturas de dados

Em situações que requerem o agrupamento de dados são maioritariamente usados arrays. No entanto, também são utilizadas algumas estruturas personalizadas (structs) nos seguintes casos:

- Quanto uma trama de dados (do nível da ligação) é recebida e processada, a informação resultante deste processamento é guardada numa struct parse_results e posteriormente usada para desencadear a resposta correta em função do estado da trama recebida.
- As informações necessárias para inicializar a porta de série são guardadas numa struct link_layer, como sugerido no enunciado.

Principais funções

- **Writer**
    - Funções principais da camada de ligação: **llopen**, **llwrite** e **llclose**.

- Funções principais da camada de aplicação: **readFile**, **splitFileData** e **savePackets**. Este conjunto de funções têm por objetivo ler o ficheiro que o utilizador seleccionou, dividi-lo em vários pacotes de acordo com o tamanho máximo de pacotes estabelecido e guardar esses pacotes na memória interna do emissor, para serem posteriormente enviados.

- Principais variáveis globais: **file_size**, **baudRate**, **packet_size**.

- **Reader**
  - Funções principais da camada de ligação: **llopen**, **llread** e **llclose**.

  - Funções principais da camada de aplicação: **restoreFile**. Esta função reúne todos os pacotes que recebeu do emissor e constrói um ficheiro com essa informação.

  - Principais variáveis globais: **baudRate**, **packet_size**.

Casos de uso principais

O principal caso de uso desta aplicação é a seleção de um ficheiro que se pretende enviar e a transferência deste ficheiro, via porta de série, entre dois computadores, o emissor e receptor.

De forma a iniciar a aplicação é necessário inserir um conjunto de argumentos. Do lado do recetor é necessário fornecer o número de porta de série que se deseja ler, a taxa de transmissão de dados e o tamanho máximo dos pacotes a receber. Do lado do emissor, a escolha do ficheiro a enviar, a escolha da porta de série, escolha da taxa de transmissão de dados, e escolha do tamanho máximo dos pacotes de dados.

A sequência de execução da aplicação é a seguinte:

1. Recetor é inicializado, no início da função **main** do ficheiro **noncanonical.c**.
2. Emissor é inicializado escolhendo o ficheiro que quer transmitir, primeiras linhas da função **main** do ficheiro **writenoncanonical.c**.
3. Estabelecimento da ligação entre emissor e recetor, função **llopen**.
4. Emissor envia pacotes de dados através da função **llwrite**.
5. Recetor recebe pacotes de dados, recorrendo à função **llread**.
6. Recetor cria um novo ficheiro com os dados recebidos, recorrendo à função **restoreFile**.
7. Terminação da ligação, usando a função **llclose**.

Protocolo de ligação lógica

llopen

Esta função deve inicializar a porta série e realizar os procedimentos necessários para iniciar a comunicação entre os programas como é explicado no enunciado. Para a inicialização e abertura da porta recorremos a um conjunto de funções auxiliares: ll_init que inicializa a struct link_layer e ll_open_serial_port que abre a porta de série

Para a troca de tramas que estabelece o início da comunicação usamos outro conjunto de funções é utilizado: generate_su_tram gera as tramas que serão trocadas, receive_tram efetua a receção das tramas de resposta e parse_and_process_su_tram analisa a última trama recebida e desencadeia a resposta necessária automaticamente

Para assegurar que o emissor efetua o reenvio no caso de timeout ou erro é utilizado um ciclo while que verifica se já ocorreu um envio com sucesso ou se já se atingiu o número máximo de tentativas.

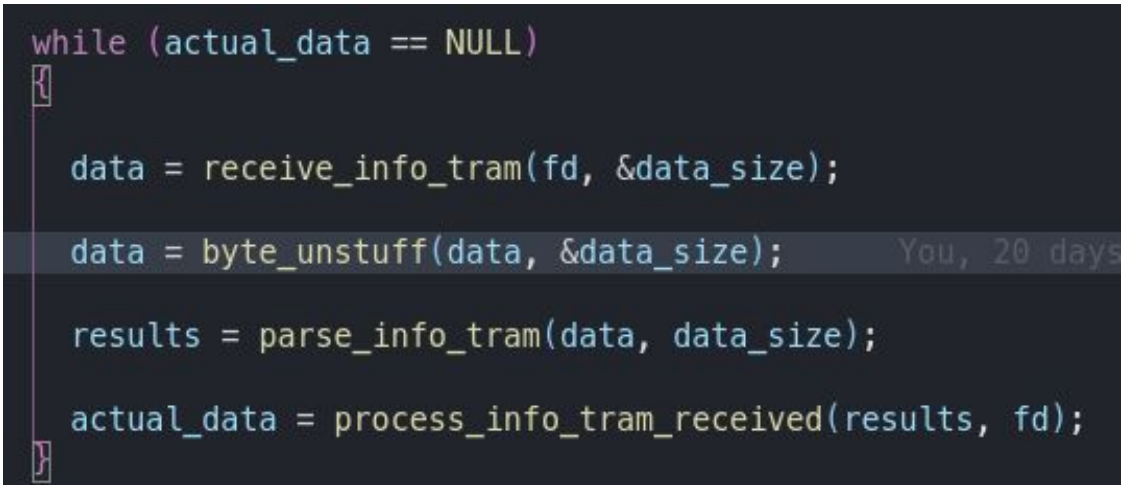
```
while (!sent_success && attempts < TIMEOUT_ATTEMPTS)
{
  res = write(fd, first_message, NON_INFO_TRAM_SIZE);
  if (res != NON_INFO_TRAM_SIZE)
  {
    fprintf(stderr, "Failed to write in llopen!\n");
    free(first_message);
    return -1;
  }
  if (baudRate <= B1800 && baudRate != B0) sleep(1 + 5 * (B1800 - baudRate) * (B1800 - baudRate));
  alarm(timeout);
  reached_timeout = 0;
  response = receive_tram(fd);
  result = parse_and_process_su_tram(response, fd);
  if (response != NULL) free(response);
  if (result == TIMED_OUT) attempts++;
  else if (result == SEND_NEW_DATA)
  {
    alarm(0);
    sent_success = 1;
  }
  else
  {
    fprintf(stderr, "Wrong result in llopen!\n");

    free(first_message);
    return -1;
  }

}
free(first_message);
```

llwrite

A função llwrite deve assegurar que os dados passados como parâmetro a esta função chegam ao receptor. Para gerar tramas info usa-se a função generate_info_tram e posteriormente a função byte_stuff para realizar o stuffing. À semelhança de llopen utiliza-se a função receive_tram para a receção de respostas, assim como um ciclo muito semelhante para os reenvios em caso de erro e timeout.

llread

Esta função deve permitir ao receptor receber dados. Desta forma, é utilizada a função receive_info_tram que trata da recepção de uma trama info, seguida da função byte_unstuff que realiza o unstuffing da trama. Posteriormente, a função parse_info_tram analisa a trama recebida, verificando se existem erros e retorna uma struct com os resultados da análise, a qual será usada pela process_info_tram para desencadear uma resposta adequada. Todo este código está num ciclo para assegurar que a informação é recebida.



llclose

Esta função deve terminar a comunicação na porta série realizando todos os procedimentos necessários. Isto requer novamente o uso das funções generate_su_tram, receive_tram e parse_and_process_su_tram que desempenham funções semelhantes às descritas anteriormente, sendo que para o emissor também é utilizado um ciclo while semelhante ao que é usado pelas funções llopen e llwrite, de forma a realizar o reenvio em caso de timeout ou erros. Por fim, a função ll_close_serial_port fecha a porta série.

Protocolo de aplicação

O protocolo de aplicação implementado têm como principais objetivos:

O envio dos pacotes de controlo que contêm o nome e o tamanho do ficheiro a ser enviado. O utilizador fornece nos argumentos do emissor o nome do ficheiro que quer enviar ao recetor. A função **readFile** permite ler o ficheiro em questão e armazená-lo na memória interna do emissor.

A divisão do ficheiro em pacotes de dados quando se trata do emissor e a concatenação dos pacotes de dados recebidos, quando se trata do recetor. A função **splitFileData** vai separar a informação do ficheiro em múltiplos pacotes de dados consoante o tamanho máximo de tramas fornecido pelo utilizador. Depois disso a função **savePackets** junta todos estes pacotes numa array para facilitar o envio das várias tramas de informação.

Encapsular cada pacote de dados com um header contendo o número de sequência do pacote e o tamanho do pacote, do lado de emissor. Este procedimento é feito antes de enviar a trama de informação através do **llwrite**, na função **generate_data_packet**. No caso do recetor, **llread** seguido de **extract_seq_size_data** assegura a interpretação da trama de informação e armazena o conteúdo do pacote de dados num array.

Criação do ficheiro, quando se trata do recetor. Esta funcionalidade é assegurada na função **restoreFile** do recetor, que reúne todos os pacotes de dados duma array e cria o ficheiro.

Validação

De forma a estudar a aplicação desenvolvida, foram efetuados os seguintes testes:

- Envio de ficheiros de vários tamanhos.
- Envio do um ficheiro com variação da taxa de transmissão de dados.
- Envio do um ficheiro com variação do tamanho de pacotes.
- Interrupção da ligação por alguns segundos enquanto se envia um ficheiro.

Todos estes testes foram concluídos com sucesso. Para além dos testes realizados na apresentação, também foram realizados outros testes com a intenção de analisar a eficiência do programa, estes testes são explicados no anexo 3. Ao realizar estes testes foi possível verificar que o programa pode não terminar corretamente quando ocorrem demasiados erros consecutivos. Por fim, com taxas de transmissão muito baixas (B1800 ou abaixo) foi necessária a utilização da função sleep com parâmetros grandes para dar tempo ao recetor de responder aos envios do emissor.

Eficiência do protocolo de ligação de dados

No anexo 3 encontram-se os gráficos que mostram os tempos de transferência medidos com os diferentes conjunto de argumentos que foram fornecidos ao programa. Relativamente à caracterização estatística de eficiência, nos testes em que o t_prop não era o foco do teste, o t_prop emulado era nulo, portanto nesses casos a eficiência aproxima-se de 100%.

Quando simulamos um valor de t_prop não nulo, obtivemos os seguintes resultados:
t_prop = 1: a = 1/(1000/(10968/17.588)) ~= 0.624; S ~= 44.5%
t_prop = 2: a = 2/(1000/(10968/31.037)) ~= 0.707; S ~= 41.4%
t_prop = 3: a = 3/(1000/(10968/44.037)) ~= 0.747; S ~= 40.1%
t_prop = 4: a = 4/(1000/(10968/57.037)) ~= 0.769; S ~= 39.4%
t_prop = 5: a = 5/(1000/(10968/70.039)) ~= 0.783; S ~= 39.0%

Como podemos ver, a eficiência vai diminuindo com valores de t_prop maiores como se esperava. Os tempos usados para o cálculo de R foram obtidos com o recetor, no anexo 3 estão disponíveis mais informações sobre a obtenção destes valores. O cálculo do valor de R foi feito dividindo o tamanho do ficheiro pinguim.gif pelo tempo usado para a sua transferência.

Conclusão

Em suma, conseguimos implementar um protocolo de ligação de dados que cumpre todos os requisitos funcionais e passou quase todos os testes a que foi submetido. Isto é feito através do uso de diversos blocos funcionais que cumprem várias funções diferentes, desde a geração de tramas ao nível da aplicação ou ligação de dados até à receção de diferentes tipos de trama. Relativamente à eficiência, o programa poderia ser melhor mas comporta-se do modo esperado. A realização deste trabalho permitiu o aprofundamento de conhecimentos relativamente ao funcionamento da camada de ligação de dados e a criação de um serviço de comunicação de dados fiável entre dois computadores através da porta de série.

Anexo 1

noncanonical.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "tram.h"
#include "state_machine.h"
#include "link_layer.h"
#include "app_layer.h"
#define FALSE 0
volatile int STOP = FALSE;
int main(int argc, char **argv)
{
    int fd = 0;
    ll = NULL;
    packet_size = atoi(argv[3]);
    max_packet_size = packet_size;
    max_array_size = max_packet_size * 2;
    if (argc < 2)
    {
        printf("Usage:\tnserial SerialPort\n\tex: nserial /dev/ttyS11\n");
        exit(1);
    }
    baudRate = 0;
    if (strcmp(argv[2], "B0") == 0)
        baudRate = B0;
    else if (strcmp(argv[2], "B50") == 0)
        baudRate = B50;
    else if (strcmp(argv[2], "B75") == 0)
        baudRate = B75;
    else if (strcmp(argv[2], "B110") == 0)
        baudRate = B110;
    else if (strcmp(argv[2], "B134") == 0)
        baudRate = B134;
    else if (strcmp(argv[2], "B150") == 0)
        baudRate = B150;
    else if (strcmp(argv[2], "B200") == 0)
        baudRate = B200;
    else if (strcmp(argv[2], "B300") == 0)
        baudRate = B300;
    else if (strcmp(argv[2], "B600") == 0)
        baudRate = B600;
    else if (strcmp(argv[2], "B1200") == 0)
        baudRate = B1200;
    else if (strcmp(argv[2], "B1800") == 0)
        baudRate = B1800;
    else if (strcmp(argv[2], "B2400") == 0)
        baudRate = B2400;
    else if (strcmp(argv[2], "B4800") == 0)
```

```
        baudRate = B4800;
    else if (strcmp(argv[2], "B9600") == 0)
        baudRate = B9600;
    else if (strcmp(argv[2], "B19200") == 0)
        baudRate = B19200;
    else if (strcmp(argv[2], "B38400") == 0)
        baudRate = B38400;
    else if (strcmp(argv[2], "B57600") == 0)
        baudRate = B57600;
    else if (strcmp(argv[2], "B115200") == 0)
        baudRate = B115200;
    else
        fprintf(stderr, "Invalid baudrate provided!\n");
    fd = llopen(atoi(argv[1]), RECEIVER);
    if (fd < 0)
    {
        fprintf(stderr, "llopen failed!\n");
        return -1;
    }
    clock_t begin = clock();
    //First Control Packet
    unsigned char *control_packet_received = (unsigned char *)calloc(max_array_size,
sizeof(unsigned char));
    llread(fd, (char *)control_packet_received);
    unsigned char *size = (unsigned char *)calloc(8, sizeof(unsigned char));
    unsigned char *name = (unsigned char *)calloc(MAX_STR_SIZE, sizeof(unsigned char));
    extract_size_name(control_packet_received, size, name);
    unsigned char *expected_final_control = control_packet_received;
    expected_final_control[0] = END;
    long received_size = *((long *)size);
    max_packet_elems = ((int)received_size / max_packet_size) + 1;
    //Initialize packet
    packet = (unsigned char **)calloc(max_packet_elems, sizeof(unsigned char *));
    for (int i = 0; i < max_packet_elems; i++)
    {
        packet[i] = (unsigned char *)calloc(max_array_size, sizeof(unsigned char));
    }
    free(size);
    //Read File Packets Based On Received_Size
    int packet_num;
    if (received_size % packet_size != 0)
        packet_num = received_size / packet_size + 1;
    else
        packet_num = received_size / packet_size;
    unsigned char *tram = (unsigned char *)calloc(max_array_size, sizeof(unsigned char));
    int stored_packet_size, seq;
    //Main reception loop
    int i = 0;
    while (strcmp((char *)tram, (char *)expected_final_control) != 0)
    {
        stored_packet_size = llread(fd, (char *)tram);
        if (tram[0] != 1)
            break;
        extract_seq_size_data(tram, &seq, &stored_packet_size, packet[i]);
        i++;
        printf("Packet Restore = %d%%\n", (100 * i / packet_num));
    }
```

```
    //Last Control Packet
    unsigned char *last_size = (unsigned char *)calloc(8, sizeof(unsigned char));
    unsigned char *last_name = (unsigned char *)calloc(MAX_STR_SIZE, sizeof(unsigned
char));
    extract_size_name(tram, last_size, name);
    long final_received_size = *((long *)last_size);
    free(last_size);
    if (received_size == final_received_size && strcmp((char *)name, (char *)last_name) ==
0 && tram[0] == END)
    {
        printf("Last Control Packet Checked!\n");
    }
    clock_t end = clock();
    double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
    printf("Execution Time = %f Seconds\n", time_spent);
    restoreFile((char *)name, packet, packet_size, packet_num, final_received_size);
    llclose(fd);
    free(name);
    free(last_name);
    free(tram);
    free(control_packet_received);
    for (int i = 0; i < max_packet_elems; i++)
    {
        free(packet[i]);
    }
    free(packet);
    return 0;
}
```

writenoncanonical.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <unistd.h>
#include <strings.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <time.h>
#include "tram.h"
#include "state_machine.h"
#include "app_layer.h"
#include "link_layer.h"
void sigalrm_handler(int signo)
{
    if (signo != SIGALRM)
        fprintf(stderr, "This signal handler shouldn't have been called. signo: %d\n",
signo);
    reached_timeout = 1;
}
void set_sigaction()
{
    struct sigaction action;
    action.sa_handler = sigalrm_handler;
```

```
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;
    if (sigaction(SIGALRM, &action, NULL) < 0)
        fprintf(stderr, "Couldn't install signal handler for SIGALRM.\n");
}
int main(int argc, char **argv)
{
    set_sigaction();
    struct stat file_data;
    if (stat(argv[2], &file_data) < 0)
    {
        fprintf(stderr, "Couldn't get file data!\n");
    }
    file_size = file_data.st_size;
    char file_name[MAX_STR_SIZE + 6];
    char argv_copy[MAX_STR_SIZE];
    strcpy(argv_copy, argv[2]);
    char *actual_file_name = strtok(argv_copy, ".");
    sprintf(file_name, "%s_clone.%s", actual_file_name, strtok(NULL, "."));
    max_packet_size = atoi(argv[4]);
    packet_size = max_packet_size;
    max_array_size = max_packet_size * 2;
    int fd = 0;
    timeout = 10;
    max_packet_elems = ((int)file_size / max_packet_size) + 1;
    //Initialize packet
    packet = (unsigned char **)calloc(max_packet_elems, sizeof(unsigned char *));
    ll = NULL;
    if (argc < 2)
    {
        printf("Usage:\tnserial SerialPort\n\tex: nserial /dev/ttyS11\n");
        exit(1);
    }
    unsigned char *fileData = readFile((unsigned char *)argv[2]);
    processFile(fileData);
    free(fileData);
    clock_t begin = clock();
    baudRate = 0;
    if (strcmp(argv[3], "B0") == 0)
        baudRate = B0;
    else if (strcmp(argv[3], "B50") == 0)
        baudRate = B50;
    else if (strcmp(argv[3], "B75") == 0)
        baudRate = B75;
    else if (strcmp(argv[3], "B110") == 0)
        baudRate = B110;
    else if (strcmp(argv[3], "B134") == 0)
        baudRate = B134;
    else if (strcmp(argv[3], "B150") == 0)
        baudRate = B150;
    else if (strcmp(argv[3], "B200") == 0)
        baudRate = B200;
    else if (strcmp(argv[3], "B300") == 0)
        baudRate = B300;
    else if (strcmp(argv[3], "B600") == 0)
        baudRate = B600;
    else if (strcmp(argv[3], "B1200") == 0)
```

```
        baudRate = B1200;
    else if (strcmp(argv[3], "B1800") == 0)
        baudRate = B1800;
    else if (strcmp(argv[3], "B2400") == 0)
        baudRate = B2400;
    else if (strcmp(argv[3], "B4800") == 0)
        baudRate = B4800;
    else if (strcmp(argv[3], "B9600") == 0)
        baudRate = B9600;
    else if (strcmp(argv[3], "B19200") == 0)
        baudRate = B19200;
    else if (strcmp(argv[3], "B38400") == 0)
        baudRate = B38400;
    else if (strcmp(argv[3], "B57600") == 0)
        baudRate = B57600;
    else if (strcmp(argv[3], "B115200") == 0)
        baudRate = B115200;
    else
        fprintf(stderr, "Invalid baudrate provided!\n");
    fd = llopen(atoi(argv[1]), TRANSMITTER);
    if (fd < 0)
    {
        fprintf(stderr, "llopen failed!\n");
        free(packet);
        return -1;
    }
    int *t_values = calloc(2, sizeof(int));
    t_values[0] = FILE_SIZE;
    t_values[1] = FILE_NAME;
    int *l_values = calloc(2, sizeof(int));
    l_values[0] = sizeof(file_size);
    l_values[1] = strlen(file_name);
    unsigned char **values = (unsigned char **)calloc(2, sizeof(unsigned char *));
    values[0] = (unsigned char *)&file_size;
    values[1] = (unsigned char *)file_name;
    unsigned char *control_packet = generate_control_packet(START, 2, t_values, l_values,
values);
    long control_packet_size = 1 + l_values[0] + l_values[1] + 4;
    free(t_values);
    free(l_values);
    free(values);
    //First Control Packet
    if (llwrite(fd, (char *)control_packet, control_packet_size) < 0)
    {
        fprintf(stderr, "llwrite failed!\n");
        free(control_packet);
        for (int i = 0; i < max_packet_elems; i++)
        {
            free(packet[i]);
        }
        free(packet);
        return -1;
    }
    //File Packets
    for (int i = 0; i < packet_num; i++)
    {
        unsigned char *data_packet = generate_data_packet(i, packet_size, packet[i]);
```

```
        if (llwrite(fd, (char *)data_packet, packet_size + 4) < 0)
        {
            fprintf(stderr, "llwrite failed!\n");
            free(data_packet);
            free(control_packet);
            for (int i = 0; i < max_packet_elems; i++)
            {
                free(packet[i]);
            }
            free(packet);
            return -1;
        }
        free(data_packet);
    }
    //Last Control Packet
    control_packet[0] = END;
    if (llwrite(fd, (char *)control_packet, control_packet_size) < 0)
    {
        fprintf(stderr, "llwrite failed!\n");
        free(control_packet);
        for (int i = 0; i < max_packet_elems; i++)
        {
            free(packet[i]);
        }
        free(packet);
    }
    clock_t end = clock();
    double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
    printf("Execution Time = %f Seconds\n", time_spent);
    free(control_packet);
    llclose(fd);
    for (int i = 0; i < max_packet_elems; i++)
    {
        free(packet[i]);
    }
    free(packet);
    return 0;
}
```

app_layer.h

```
#pragma once
//Control field values
#define DATA 1
#define START 2
#define END 3
//Type field values
#define FILE_SIZE 0
#define FILE_NAME 1
#include "tram.h"
unsigned char *readFile(unsigned char *fileName);
unsigned char *splitFileData(unsigned char *fileData, int x, int packet_size);
void savePackets(unsigned char *packet[], unsigned char *fileData);
void restoreFile(char *fileName, unsigned char *packet[], int packet_size, int packet_num,
long int file_size);
void restoreSimpleFile(char *fileName, unsigned char *fileData, long int file_size);
void deleteFile(char *fileName);
```

```
void processFile(unsigned char *fileData);
unsigned char *generate_data_packet(int seq_num, int byte_num, const unsigned char *data);
unsigned char *generate_control_packet(unsigned char control_field, int param_num, int
*t_values, int *l_values, unsigned char **values);
void extract_size_name(unsigned char *tram, unsigned char *size, unsigned char *name);
void extract_seq_size_data(unsigned char *tram, int *seq, int *size, unsigned char *data);
```

app_layer.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <unistd.h>
#include <strings.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include "app_layer.h"
unsigned char *readFile(unsigned char *fileName)
{
    printf("Reading File <%s>\n", fileName);
    FILE *f;
    struct stat metadata;
    unsigned char *fileData;
    if ((f = fopen((char *)fileName, "rb")) == NULL)
    {
        perror("Error Read File!");
    }
    stat((char *)fileName, &metadata);
    file_size = metadata.st_size;
    printf("File Size =  %ld Bytes\n", file_size);
    fileData = (unsigned char *)calloc(file_size, 1);
    fread(fileData, sizeof(unsigned char), file_size, f);
    fclose(f);
    return fileData;
}
unsigned char *splitFileData(unsigned char *fileData, int x, int packet_size)
{
    unsigned char *packet_temp = (unsigned char *)calloc(max_array_size, sizeof(unsigned
char));
    int i;
    int j;
    for (j = 0, i = x; i < packet_size; i++, j++)
    {
        if (i >= file_size)
            break;
        packet_temp[j] = fileData[i];
    }
    return packet_temp;
}
void savePackets(unsigned char *packet[], unsigned char *fileData)
{
    for (int i = 0; i < packet_num; i++)
    {
```

```
        packet[i] = splitFileData(fileData, packet_size * i, i * packet_size +
packet_size);
        printf("Packet[%d] Saved!\n", i);
    }
}
void restoreFile(char *fileName, unsigned char *packet[], int packet_size, int packet_num,
long int file_size)
{
    printf("Restoring File...\n");
    FILE *f = fopen((char *)fileName, "wb+");
    for (int i = 0; i < packet_num; i++)
    {
        //Last Packet
        if (i == (packet_num - 1))
        {
            fwrite((void *)packet[i], 1, (file_size % packet_size), f);
        }
        else
            fwrite((void *)packet[i], 1, packet_size, f);
    }
    printf("File Restored!\n");
    fclose(f);
}
void restoreSimpleFile(char *fileName, unsigned char *fileData, long int file_size)
{
    printf("Restoring Simple File...\n");
    FILE *f = fopen((char *)fileName, "ab+");
    fwrite((void *)fileData, 1, file_size, f);
    printf("New File Created!\n");
    fclose(f);
}
void deleteFile(char *fileName)
{
    FILE *output = fopen(fileName, "w");
    fclose(output);
}
void processFile(unsigned char *fileData)
{
    if (file_size % packet_size != 0)
    {
        packet_num = file_size / packet_size + 1;
    }
    else
    {
        packet_num = file_size / packet_size;
    }
    printf("Created %d Packets...\n", packet_num);
    savePackets(packet, fileData);
    printf("Packets Ready To Be Sent!\n");
}
unsigned char *generate_data_packet(int seq_num, int byte_num, const unsigned char *data)
{
    unsigned char *result = calloc(byte_num + 4, sizeof(unsigned char));
    result[0] = DATA;
    result[1] = seq_num;
    int l2 = (int)byte_num / 256;
    int l1 = byte_num % 256;
```

```
    result[2] = l2;
    result[3] = l1;
    for (int i = 4; i < (byte_num + 4); i++)
    {
        result[i] = data[i - 4];
    }
    return result;
}
unsigned char *generate_control_packet(unsigned char control_field, int param_num, int
*t_values, int *l_values, unsigned char **values)
{
    int total_byte_num = 1 + 2 * param_num;
    for (int i = 0; i < param_num; i++)
    {
        total_byte_num += l_values[i];
    }
    unsigned char *result = calloc(total_byte_num, sizeof(unsigned char));
    result[0] = control_field;
    int nextIndex = 1;
    for (int i = 0; i < param_num; i++)
    {
        result[nextIndex++] = t_values[i];
        result[nextIndex++] = l_values[i];
        int currentDataSize = l_values[i];
        for (int j = 0; j < currentDataSize; j++)
        {
            result[nextIndex++] = values[i][j];
        }
    }
    return result;
}
void extract_size_name(unsigned char *tram, unsigned char *size, unsigned char *name)
{
    if (tram[0] != END && tram[0] != START) return;
    int middle_of_data = 0, extracted_name = 0, extracted_size = 0;
    int i = 1;
    int data_index = 0;
    unsigned char type, length;
    while (!extracted_size || !extracted_name)
    {
        if (!middle_of_data)
        {
            type = tram[i];
            length = tram[++i];
            middle_of_data = 1;
            data_index = 0;
        }
        else
        {
            if (type == FILE_SIZE)
            {
                if (length == 0)
                {
                    extracted_size = 1;
                    middle_of_data = 0;
                    continue;
                }
```

```
                size[data_index++] = tram[i];
            }
            else if (type == FILE_NAME)
            {
                if (length == 0)
                {
                    extracted_name = 1;
                    middle_of_data = 0;
                    continue;
                }
                name[data_index++] = tram[i];
            }
            length--;
        }
        i++;
    }
}
void extract_seq_size_data(unsigned char *tram, int *seq, int *size, unsigned char *data)
{
    (*seq) = tram[1];
    (*size) = tram[2] * 256 + tram[3];
    for (int i = 0; i < (*size); i++)
    {
        data[i] = tram[i + 4];
    }
}
```

link_layer.h

```
#pragma once
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <unistd.h>
#include <strings.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include "tram.h"
#include "state_machine.h"
#define MAX_TRAM_SIZE 2
// Flag values
#define TRANSMITTER 0
#define RECEIVER 1
typedef struct
{
    char *port;
    int baudRate;
    unsigned int sequenceNumber;
    unsigned int timeout;
    unsigned int numTransmissions;
    char frame[MAX_TRAM_SIZE];
} link_layer;
static link_layer *ll;
int ll_init(char *port, int baudRate, unsigned int timeout, unsigned int numTransmissions);
```

```c
int ll_open_serial_port(int fd, int baudRate);
int llopen(int port, int flag);
int llwrite(int fd, char *buffer, int length);
int llread(int fd, char *buffer);
void ll_close_serial_port(int fd);
int llclose(int fd);
```

link_layer.c

```c
#include "link_layer.h"
#include "app_layer.h"
#include <time.h>
struct termios oldtio, newtio;
int ll_init(char *port, int baudRate, unsigned int timeout, unsigned int numTransmissions)
{
    if (ll == NULL)
    {
        ll = calloc(1, sizeof(link_layer));
    }
    else
        printf("Link Layer Already Initialized\n");
    ll->port = port;
    ll->baudRate = baudRate;
    ll->timeout = timeout;
    ll->numTransmissions = numTransmissions;
    ll->sequenceNumber = 0;
    printf("Link Layer Initialized!\n");
    return 0;
}
int ll_open_serial_port(int fd, int baudRate)
{
    fd = open(ll->port, O_RDWR | O_NOCTTY);
    if (fd < 0)
    {
        perror(ll->port);
        exit(-1);
    }
    if (tcgetattr(fd, &oldtio) == -1)
    { /* save current port settings */
        perror("tcgetattr");
        exit(-1);
    }
    bzero(&newtio, sizeof(newtio));
    newtio.c_cflag = baudRate | CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;
    newtio.c_lflag = 0;
    newtio.c_cc[VTIME] = ll->timeout * 100; /* inter-character timer unused */
    newtio.c_cc[VMIN] = 5;                  /* blocking read until 5 chars received */
    tcflush(fd, TCIOFLUSH);
    if (tcsetattr(fd, TCSANOW, &newtio) == -1)
    {
        perror("tcsetattr");
        exit(-1);
    }
    printf("New termios structure set\n");
    return fd;
```

```
}
int llopen(int port, int flag)
{
    char *actual_port = calloc(12, sizeof(char));
    sprintf(actual_port, "/dev/ttyS%d", port);
    ll_init(actual_port, baudRate, timeout, 1);
    int fd = 0;
    if (flag == TRANSMITTER)
    {
        sender = 1;
        setup_initial_values();
        int sent_success = 0;
        fd = ll_open_serial_port(fd, baudRate);
        unsigned char *first_message = generate_su_tram(COMM_SEND_REP_REC, SET, 0);
        int res, result;
        int attempts = 0;
        unsigned char *response;
        while (!sent_success && attempts < TIMEOUT_ATTEMPTS)
        {
            res = write(fd, first_message, NON_INFO_TRAM_SIZE);
            if (res != NON_INFO_TRAM_SIZE)
            {
                fprintf(stderr, "Failed to write in llopen!\n");
                free(first_message);
                return -1;
            }
            if (baudRate <= B1800 && baudRate != B0)
                sleep(1 + 5 * (B1800 - baudRate) * (B1800 - baudRate));
            alarm(timeout);
            reached_timeout = 0;
            response = receive_tram(fd);
            result = parse_and_process_su_tram(response, fd);
            if (response != NULL)
                free(response);
            if (result == TIMED_OUT)
                attempts++;
            else if (result == SEND_NEW_DATA)
            {
                alarm(0);
                sent_success = 1;
            }
            else
            {
                fprintf(stderr, "Wrong result in llopen!\n");
                free(first_message);
                return -1;
            }
        }
        free(first_message);
        if (attempts == TIMEOUT_ATTEMPTS)
            return -1;
    }
    else
    {
        sender = 0;
        setup_initial_values();
        fd = ll_open_serial_port(fd, baudRate);
```

```
        unsigned char *first_request = receive_tram(fd);
        int result = parse_and_process_su_tram(first_request, fd);
        if (result != DO_NOTHING)
        {
            fprintf(stderr, "Wrong result in llopen!\n");
            return -1;
        }
        free(first_request);
    }
    free(actual_port);
    free(ll);
    printf("Finished the start process!\n");
    return fd;
}
int llwrite(int fd, char *buffer, int length)
{
    int tram_length = length + 6;
    unsigned char *data_tram = generate_info_tram(buffer, COMM_SEND_REP_REC, length);
    data_tram = byte_stuff(data_tram, &tram_length);
    int res = -1, parse_result;
    int data_sent_success = 0;
    int attempts = 0;
    unsigned char *response;
    while (!data_sent_success && attempts < TIMEOUT_ATTEMPTS)
    {
        res = write(fd, data_tram, tram_length);
        if (res != tram_length)
        {
            fprintf(stderr, "Failed to write in llwrite!\n");
            return -1;
        }
        if (baudRate <= B1800 && baudRate != B0)
            sleep(1 + 5 * (B1800 - baudRate) * (B1800 - baudRate));
        alarm(timeout);
        reached_timeout = 0;
        response = receive_tram(fd);
        parse_result = parse_and_process_su_tram(response, fd);
        free(response);
        if (parse_result == SEND_NEW_DATA)
        {
            data_sent_success = 1;
            alarm(0);
        }
        else if (parse_result == DO_NOTHING)
        {
            fprintf(stderr, "S/U tram processing failed in llwrite!\n");
            return -1;
        }
        else if (parse_result == TIMED_OUT)
            attempts++;
    }
    free(data_tram);
    if (attempts == TIMEOUT_ATTEMPTS)
    {
        fprintf(stderr, "Reached max number of attempts!\n");
        return -1;
    }
```

```
    return res;
}
int llread(int fd, char *buffer)
{
    char *actual_data = NULL;
    int data_size;
    unsigned char *data;
    struct parse_results *results;
    while (actual_data == NULL)
    {
        data = receive_info_tram(fd, &data_size);
        data = byte_unstuff(data, &data_size);
        results = parse_info_tram(data, data_size);
        actual_data = process_info_tram_received(results, fd);
    }
    free(data);
    free(results->received_data);
    free(results);
    memcpy(buffer, actual_data, max_array_size);
    free(actual_data);
    data_trams_received = data_trams_received + 1;
    return (data_size - 4);
}
void ll_close_serial_port(int fd)
{
    if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
    {
        perror("tcsetattr");
        exit(-1);
    }
    close(fd);
}
int llclose(int fd)
{
    if (sender)
    {
        unsigned char *new_tram = generate_su_tram(COMM_SEND_REP_REC, DISC, 0);
        int size = NON_INFO_TRAM_SIZE;
        int res, result;
        int sent_success = 0, attempts = 0;
        while (!sent_success && attempts < TIMEOUT_ATTEMPTS)
        {
            res = write(fd, new_tram, size);
            if (res != NON_INFO_TRAM_SIZE)
            {
                fprintf(stderr, "Failed to write on llclose!\n");
                return -1;
            }
            if (baudRate <= B1800 && baudRate != B0)
                sleep(1 + 5 * (B1800 - baudRate) * (B1800 - baudRate));
            alarm(timeout);
            reached_timeout = 0;
            unsigned char *response = receive_tram(fd);
            result = parse_and_process_su_tram(response, fd);
            free(response);
            if (result == TIMED_OUT)
                attempts++;
```

```
            else if (result == DO_NOTHING)
            {
                alarm(0);
                sent_success = 1;
            }
            else
            {
                fprintf(stderr, "Processing failed in llclose on sender! Result was: %d\n",
result);
                return -1;
            }
        }
        sleep(2);
        free(new_tram);
        if (attempts == TIMEOUT_ATTEMPTS)
            return -1;
    }
    else
    {
        unsigned char *end_request = receive_tram(fd);
        int result = parse_and_process_su_tram(end_request, fd);
        if (result != DO_NOTHING)
        {
            fprintf(stderr, "Processing failed in llclose for the receiver!\n");
            return -1;
        }
        free(end_request);
        unsigned char *acknowledgment = receive_tram(fd);
        result = parse_and_process_su_tram(acknowledgment, fd);
        if (result != DO_NOTHING)
        {
            fprintf(stderr, "Processing failed in llclose for the receiver!\n");
            return -1;
        }
        free(acknowledgment);
    }
    ll_close_serial_port(fd);
    return 1;
}
```

state_machine.h

```
#include "tram.h"
#pragma once
int timeout;
int reached_timeout;
enum reception_state
{
    start,
    flag_rcv,
    a_rcv,
    c_rcv,
    bcc_ok
};
enum reception_info_state
{
    start_info,
```

```
    flag_rcv_info,
    a_rcv_info,
    c_rcv_info,
    receiving_data_info
};
unsigned char *receive_tram(int fd);
unsigned char *receive_info_tram(int fd, int *data_size);
```

state_machine.c

```
#include "state_machine.h"
#include <errno.h>
unsigned char *receive_tram(int fd)
{
    unsigned char *result = calloc(3, sizeof(unsigned char));
    enum reception_state state = start;
    unsigned char currentByte = 0x00;
    int res, continue_loop = 1;
    while (continue_loop && !reached_timeout)
    {
        res = read(fd, &currentByte, 1);
        if (res != 1)
            fprintf(stderr, "Failed to read in receive_tram!\n");
        switch (state)
        {
        case start:
        {
            if (currentByte == FLAG)
                state = flag_rcv;
            break;
        }
        case flag_rcv:
        {
            if (currentByte == COMM_SEND_REP_REC || currentByte == COMM_REC_REP_SEND)
            {
                state = a_rcv;
                result[0] = currentByte;
            }
            else if (currentByte != FLAG)
                state = start;
            break;
        }
        case a_rcv:
        {
            if (currentByte == UA || currentByte == DISC || currentByte == SET ||
currentByte == REJ || currentByte == (REJ | R_MASK) || currentByte == RR || currentByte ==
(RR | R_MASK))
            {
                state = c_rcv;
                result[1] = currentByte;
            }
            else if (currentByte == FLAG)
                state = flag_rcv;
            else
                state = start;
            break;
        }
```

```
        case c_rcv:
        {
            if (currentByte == (result[0] ^ result[1]))
            {
                state = bcc_ok;
                result[2] = currentByte;
            }
            else if (currentByte == FLAG)
                state = flag_rcv;
            else
                state = start;
            break;
        }
        case bcc_ok:
        {
            if (currentByte == FLAG)
            {
                continue_loop = 0;
            }
            else
                state = start;
            break;
        }
        default:
        {
            fprintf(stderr, "Invalid reception state!\n");
            break;
        }
        }
    }
    if (reached_timeout)
    {
        return NULL;
    }
    return result;
}
unsigned char *receive_info_tram(int fd, int *data_size)
{
    unsigned char *result = calloc(max_array_size, sizeof(unsigned char));
    enum reception_info_state state = start_info;
    unsigned char currentByte = 0x00;
    int res, continue_loop = 1, currentIndex = 0;
    while (continue_loop)
    {
        res = read(fd, &currentByte, 1);
        if (res != 1)
            fprintf(stderr, "Failed to read in receive_info_tram!\n");
        switch (state)
        {
        case start_info:
        {
            if (currentByte == FLAG)
                state = flag_rcv_info;
            break;
        }
        case flag_rcv_info:
        {
```

```
            if (currentByte == COMM_SEND_REP_REC || currentByte == COMM_REC_REP_SEND)
            {
                state = a_rcv_info;
                result[0] = currentByte;
            }
            else if (currentByte != FLAG)
                state = start_info;
            break;
        }
        case a_rcv_info:
        {
            if (currentByte == INFO_CTRL || currentByte == (INFO_CTRL | S_MASK))
            {
                state = c_rcv_info;
                result[1] = currentByte;
            }
            else if (currentByte == FLAG)
                state = flag_rcv_info;
            else
                state = start_info;
            break;
        }
        case c_rcv_info:
        {
            if (currentByte == (result[0] ^ result[1]))
            {
                state = receiving_data_info;
                result[2] = currentByte;
                currentIndex = 3;
            }
            else if (currentByte == FLAG)
                state = flag_rcv_info;
            else
                state = start_info;
            break;
        }
        case receiving_data_info:
        {
            if (currentByte == FLAG)
            {
                continue_loop = 0;
            }
            else
            {
                result[currentIndex++] = currentByte;
                continue;
            }
            break;
        }
        default:
        {
            fprintf(stderr, "Invalid reception state!\n");
            break;
        }
        }
    }
    (*data_size) = currentIndex;
```

```
    return result;
}
```

tram.h

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#pragma once
// Flag
#define FLAG 0x7e
// Control Field
#define INFO_CTRL 0x00
#define SET 0x03
#define DISC 0x0b
#define UA 0x07
#define RR 0x05
#define REJ 0x01
#define R_MASK 0x80
#define S_MASK 0x40
// Address Field
#define COMM_SEND_REP_REC 0x03
#define COMM_REC_REP_SEND 0x01
//Escape sequences
#define ESC_BYTE_1 0x7d
#define ESC_BYTE_2 0x5e
#define ESC_BYTE_3 0x5d
//Sizes
#define NON_INFO_TRAM_SIZE 5
#define MAX_STR_SIZE 100
//Number of attempts
#define TIMEOUT_ATTEMPTS 3
//Process SU tram results
#define DO_NOTHING 0
#define SEND_NEW_DATA 1
#define RESEND_DATA 2
#define TIMED_OUT 3
//Last sequential number received/sent
int last_seq;
//Data received
unsigned char **packet;
//Parse results
struct parse_results
{
    unsigned char *received_data; //NULL if it's not info tram
    int tram_size;                //size of the tram in bytes
    int duplicate;                //boolean to indicate if the tram received is a duplicate
    int data_integrity;           //boolean to indicate if bcc2 checks out
    int control_bit;              //value of the bit in the control_field, either 0 or 1
    int header_validity;          //boolean to indicate if the header is valid
};
int r, s;
int last_s, last_r;
long int data_trams_received;
int sender; //boolean that indicates whether the program running is the sender or the
receiver
```

```
unsigned char *last_tram_sent;
int last_tram_sent_size;
int last_packet_index;
int packet_size, packet_num;
long int max_packet_size;
long int max_packet_elems;
long int file_size;
int baudRate;
long int max_array_size;
void setup_initial_values();
unsigned char *generate_info_tram(char *data, unsigned char address, int array_size);
unsigned char *generate_su_tram(unsigned char address, unsigned char control, int dup);
struct parse_results *parse_info_tram(unsigned char *tram, int tram_size);
char *process_info_tram_received(struct parse_results *results, int port);
unsigned char *translate_array(unsigned char *array, int offset, int array_size, int
starting_point);
unsigned char *byte_stuff(unsigned char *tram, int *tram_size);
unsigned char *byte_unstuff(unsigned char *tram, int *tram_size);
int parse_and_process_su_tram(unsigned char *tram, int fd);
```

tram.c

```
#include "tram.h"
#include "state_machine.h"
void setup_initial_values()
{
    last_seq = -1;
    reached_timeout = 0;
    if (!sender)
        data_trams_received = 0;
}
unsigned char *generate_info_tram(char *data, unsigned char address, int array_size)
{
    unsigned char *tram = calloc((6 + array_size), sizeof(unsigned char));
    tram[0] = FLAG;
    tram[1] = address;
    unsigned char actual_control = INFO_CTRL;
    if (last_seq == -1 || last_seq == 1)
        last_seq = 0;
    else if (last_seq == 0)
        last_seq = 1;
    if (last_seq == 1)
        actual_control |= S_MASK;
    tram[2] = actual_control;
    tram[3] = address ^ actual_control;
    unsigned char bcc2 = 0x00;
    for (int i = 4; i < (array_size + 4); i++)
    {
        tram[i] = data[i - 4];
        bcc2 ^= tram[i];
    }
    tram[4 + array_size] = bcc2;
    tram[5 + array_size] = FLAG;
    last_tram_sent = tram;
    last_tram_sent_size = array_size + 6;
    return tram;
}
```

```
unsigned char *generate_su_tram(unsigned char address, unsigned char control, int dup)
{
    unsigned char *tram = calloc(5, sizeof(unsigned char));
    tram[0] = FLAG;
    tram[1] = address;
    unsigned char actual_control = control;
    if (control == REJ && last_seq == 1)
        actual_control = control |= R_MASK;
    else if (control == RR)
    {
        if (!dup)
        {
            if (last_seq == 1)
                last_seq--;
            else
            {
                last_seq++;
                actual_control = control |= R_MASK;
            }
        }
        else
        {
            if (last_seq == 1)
                actual_control = control |= R_MASK;
        }
    }
    tram[2] = actual_control;
    tram[3] = address ^ actual_control;
    tram[4] = FLAG;
    return tram;
}
int parse_and_process_su_tram(unsigned char *tram, int fd)
{
    if (tram == NULL)
        return TIMED_OUT;
    unsigned char *response;
    int res;
    int result = DO_NOTHING;
    switch (tram[1])
    {
    case SET:
    {
        printf("Received request to start communication. Acknowledging.\n");
        response = generate_su_tram(COMM_SEND_REP_REC, UA, 0);
        break;
    }
    case UA:
    {
        if (sender)
        {
            printf("Communication start request was acknowledged. Starting to send
data.\n");
            return SEND_NEW_DATA;
        }
        else
        {
            printf("Communication ended.\n");
```

```
            return DO_NOTHING;
        }
        break;
    }
    case DISC:
    {
        if (sender)
        {
            printf("Communication end request was acknowledged. Acknowledging end.\n");
            response = generate_su_tram(COMM_REC_REP_SEND, UA, 0);
        }
        else
        {
            printf("Received request to end communication. Ending communication.\n");
            response = generate_su_tram(COMM_REC_REP_SEND, DISC, 0);
        }
        break;
    }
    case REJ:
    {
        printf("Last info packet sent had issues. Resending.\n");
        return RESEND_DATA;
        break;
    }
    case (REJ | R_MASK):
    {
        printf("Last info packet sent had issues. Resending.\n");
        return RESEND_DATA;
        break;
    }
    case RR:
    {
        printf("Last info packet sent had no issues. Processing.\n");
        return SEND_NEW_DATA;
        break;
    }
    case (RR | R_MASK):
    {
        printf("Last info packet sent had no issues. Processing.\n");
        return SEND_NEW_DATA;
        break;
    }
    default:
        printf("Invalid control byte!\n");
    }
    res = write(fd, response, NON_INFO_TRAM_SIZE);
    printf("%d Bytes Written\n", res);
    free(response);
    return result;
}
struct parse_results *parse_info_tram(unsigned char *tram, int tram_size)
{
    struct parse_results *result = calloc(1, sizeof(struct parse_results));
    result->received_data = calloc(max_array_size, sizeof(unsigned char));
    result->tram_size = tram_size;
    result->duplicate = 0;
    result->data_integrity = 1;
```

```
    result->control_bit = 0;
    result->header_validity = 1;
    if ((tram[0] != COMM_SEND_REP_REC) || (tram[1] != INFO_CTRL && tram[1] != (INFO_CTRL |
S_MASK)))
        result->header_validity = 0;
    unsigned char bcc1 = tram[0] ^ tram[1];
    if (bcc1 != tram[2])
        result->header_validity = 0;
    switch (tram[1])
    {
    case INFO_CTRL:
    {
        if (last_seq == -1)
            last_seq = 0;
        else if (last_seq == 1)
            result->duplicate = 1;
        memcpy(result->received_data, &tram[3], (tram_size - 4));
        printf("Data Tram Received.\n");
        break;
    }
    case (INFO_CTRL | S_MASK):
    {
        if (last_seq == -1)
            last_seq = 1;
        else if (last_seq == 0)
            result->duplicate = 1;
        memcpy(result->received_data, &tram[3], (tram_size - 4));
        printf("Data Tram Received.\n");
        break;
    }
    default:
        result->header_validity = 0;
    }
    unsigned char bcc2 = 0x00;
    for (int i = 3; i < (tram_size - 1); i++)
    {
        bcc2 ^= tram[i];
    }
    if (bcc2 != tram[tram_size - 1])
        result->data_integrity = 0;
    return result;
}
char *process_info_tram_received(struct parse_results *results, int port)
{
    unsigned char *response;
    char *result;
    result = calloc(max_array_size, sizeof(unsigned char));
    int response_size = 0;
    if (!results->header_validity)
        return NULL;
    if (results->header_validity && results->data_integrity)
    {
        if (!results->duplicate)
        {
            memcpy(result, results->received_data, results->tram_size - 4);
        }
        else
```

```
            result = NULL;
        response = generate_su_tram(COMM_SEND_REP_REC, RR, 0);
        response_size = 5;
    }
    if (results->header_validity && !results->data_integrity)
    {
        if (!results->duplicate)
        {
            response = generate_su_tram(COMM_SEND_REP_REC, REJ, 0);
            fprintf(stderr, "Data had errors, responding with REJ.\n");
        }
        else
        {
            response = generate_su_tram(COMM_SEND_REP_REC, RR, 1);
            fprintf(stderr, "Data had errors but was duplicate, responding with RR.\n");
        }
        result = NULL;
        response_size = 5;
    }
    int res = write(port, response, response_size);
    free(response);
    res = res;
    printf("%d Bytes Written\n", res);
    return result;
}
unsigned char *translate_array(unsigned char *array, int offset, int array_size, int
starting_point)
{
    unsigned char *new_array = calloc(array_size + offset, sizeof(unsigned char));
    for (int i = 0; i < (array_size); i++)
    {
        if (i < starting_point)
        {
            new_array[i] = array[i];
        }
        else if (offset > 0)
        {
            new_array[i + offset] = array[i];
        }
        else
        {
            if (i == (array_size + offset))
                break;
            new_array[i] = array[i - offset];
        }
    }
    free(array);
    return new_array;
}
unsigned char *byte_stuff(unsigned char *tram, int *tram_size)
{
    for (int i = 4; i < ((*tram_size) - 1); i++)
    {
        if (tram[i] == FLAG)
        {
            tram = translate_array(tram, 1, (*tram_size), i);
            (*tram_size)++;
```

```
            tram[i] = ESC_BYTE_1;
            tram[++i] = ESC_BYTE_2;
        }
        else if (tram[i] == ESC_BYTE_1)
        {
            tram = translate_array(tram, 1, (*tram_size), i);
            (*tram_size)++;
            tram[i] = ESC_BYTE_1;
            tram[++i] = ESC_BYTE_3;
        }
    }
    return tram;
}
unsigned char *byte_unstuff(unsigned char *tram, int *tram_size)
{
    for (int i = 3; i < (*tram_size); i++)
    {
        if (tram[i] == ESC_BYTE_1 && tram[i + 1] == ESC_BYTE_2)
        {
            tram = translate_array(tram, -1, (*tram_size), i);
            (*tram_size)--;
            tram[i] = FLAG;
        }
        else if (tram[i] == ESC_BYTE_1 && tram[i + 1] == ESC_BYTE_3)
        {
            tram = translate_array(tram, -1, (*tram_size), i);
            (*tram_size)--;
            tram[i] = ESC_BYTE_1;
        }
    }
    return tram;
}
```

Anexo 2

noncanonical.c

```
void sigalrm_handler(int signo)
{
  if (signo != SIGALRM) fprintf(stderr, "This handler shouldn't have been called.\n");
  need_to_wait = 0;
}

void set_sigaction()
{
  struct sigaction action;
  action.sa_handler = sigalrm_handler;
  sigemptyset(&action.sa_mask);
  action.sa_flags = 0;

  if (sigaction(SIGALRM, &action, NULL) < 0) fprintf(stderr, "Couldn't install signal handler for SIGALRM.\n");
}
```

```c
int main(int argc, char **argv)
{
  srand(time(NULL));
  set_sigaction();
```

```c
fer = atoi(argv[4]);
t_prop = atoi(argv[5]);
```

```c
struct timespec start_time;
clock_gettime(CLOCK_REALTIME, &start_time);
fd = llopen(atoi(argv[1]), RECEIVER);
```

```c
llclose(fd);
struct timespec end_time;
clock_gettime(CLOCK_REALTIME, &end_time);
double sTime = start_time.tv_sec + start_time.tv_nsec * 1e-9;
double eTime = end_time.tv_sec + end_time.tv_nsec * 1e-9;
double final_time = eTime - sTime;
//Writing results to csv
FILE* csv = fopen("results_reader.csv","a");
fprintf(csv,"%d,%d,%s,%ld,%f\n",fer,t_prop,argv[2],max_packet_size,final_time);
fclose(csv);
//printf("Execution Time = %.6lf\n", eTime - sTime);

restoreFile((char *)name, packet, packet_size, packet_num, final_received_size);
```

writenoncanonical.c

```c
struct timespec start_time;
clock_gettime(CLOCK_REALTIME, &start_time);
fd = llopen(atoi(argv[1]), TRANSMITTER);
```

```c
fer = atoi(argv[5]);
t_prop = atoi(argv[6]);
```

```c
free(control_packet);
llclose(fd);

struct timespec end_time;
clock_gettime(CLOCK_REALTIME, &end_time);
double sTime = start_time.tv_sec + start_time.tv_nsec * 1e-9;
double eTime = end_time.tv_sec + end_time.tv_nsec * 1e-9;
double final_time = eTime - sTime;
//Writing results to csv
FILE* csv = fopen("results_writer.csv","a");
fprintf(csv,"%d,%d,%s,%ld,%f\n",fer,t_prop,argv[3],max_packet_size,final_time);
fclose(csv);
```

link_layer.c

```
while (actual_data == NULL)
{

  data = receive_info_tram(fd, &data_size);
  //Simulating t_prop
  if (data != NULL && t_prop > 0)
  {
    need_to_wait = 1;
    alarm(t_prop);
    while (need_to_wait);
  }
  //Simulating errors
  if (data != NULL && fer > 0)
  {
    int rand_num = rand() % 100 + 1;
    if (rand_num <= fer)
    {
      int rand_num_header = rand() % 100 + 1;
      int rand_num_data = rand() % 100 + 1;
      if (rand_num_data <= DATA_ERROR_PROB)
      {
        data[20] /= 2;
        data[21] /= 5;
        data[22] /= 7;
      }
      if (rand_num_header <= HEAD_ERROR_PROB) data[0] = 9;
    }
  }
  data = byte_unstuff(data, &data_size);

  results = parse_info_tram(data, data_size);

  actual_data = process_info_tram_received(results, fd);
}
```

tram.h

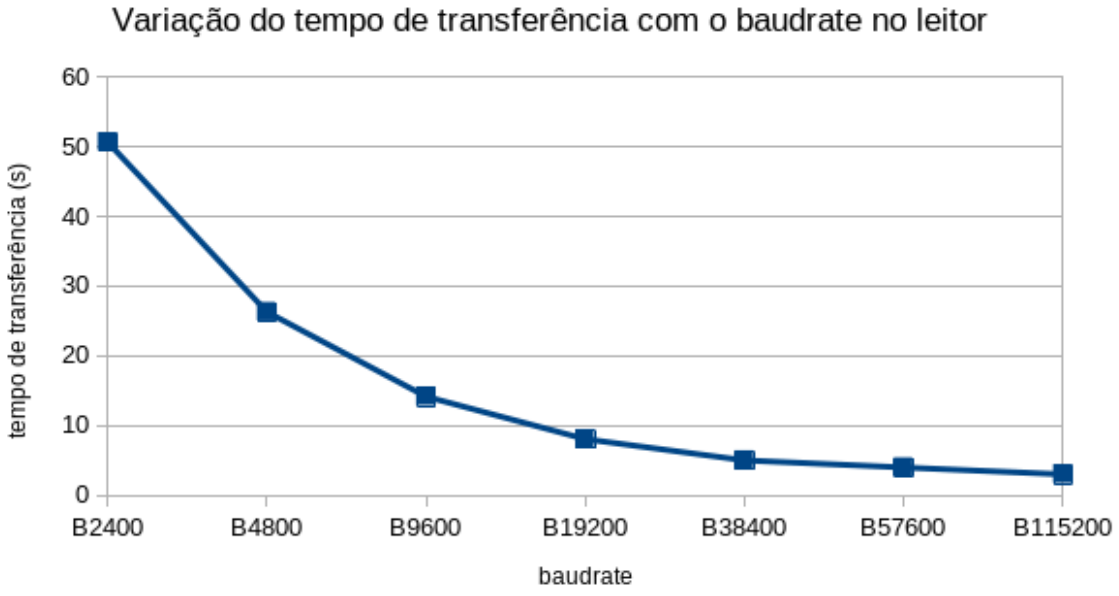
```
//Error generation
#define DATA_ERROR_PROB 30
#define HEAD_ERROR_PROB 30
```

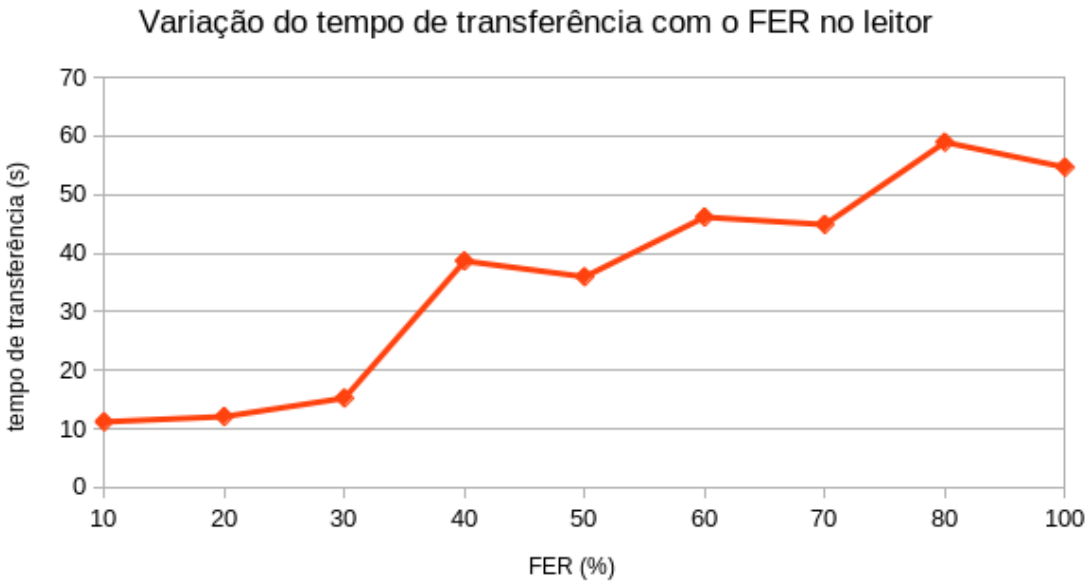
```
+ int fer, t_prop;
+ int need_to_wait;
+
```

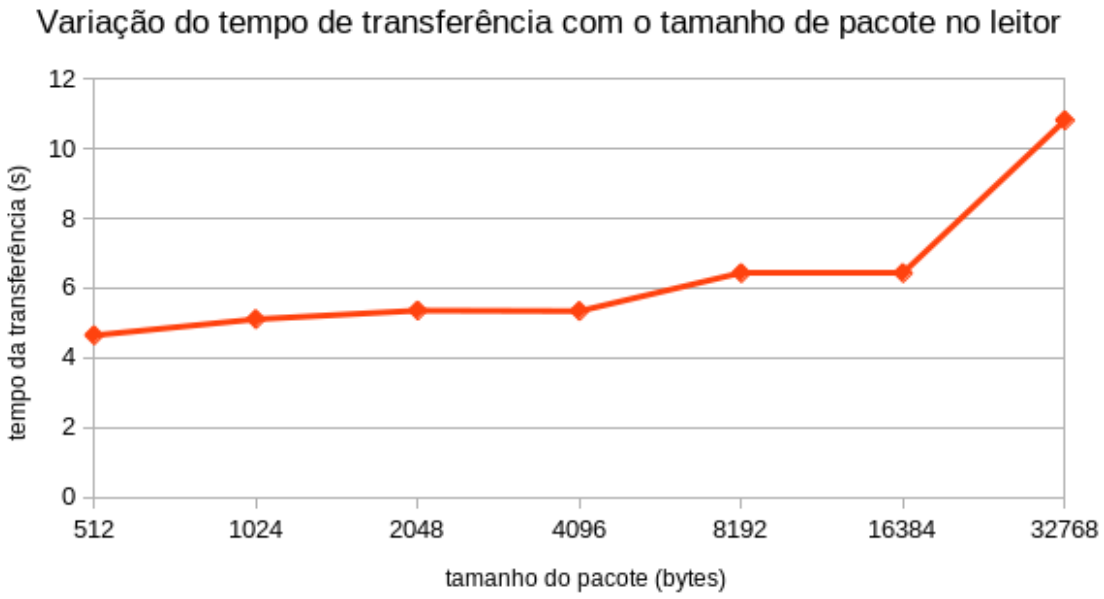
Anexo 3

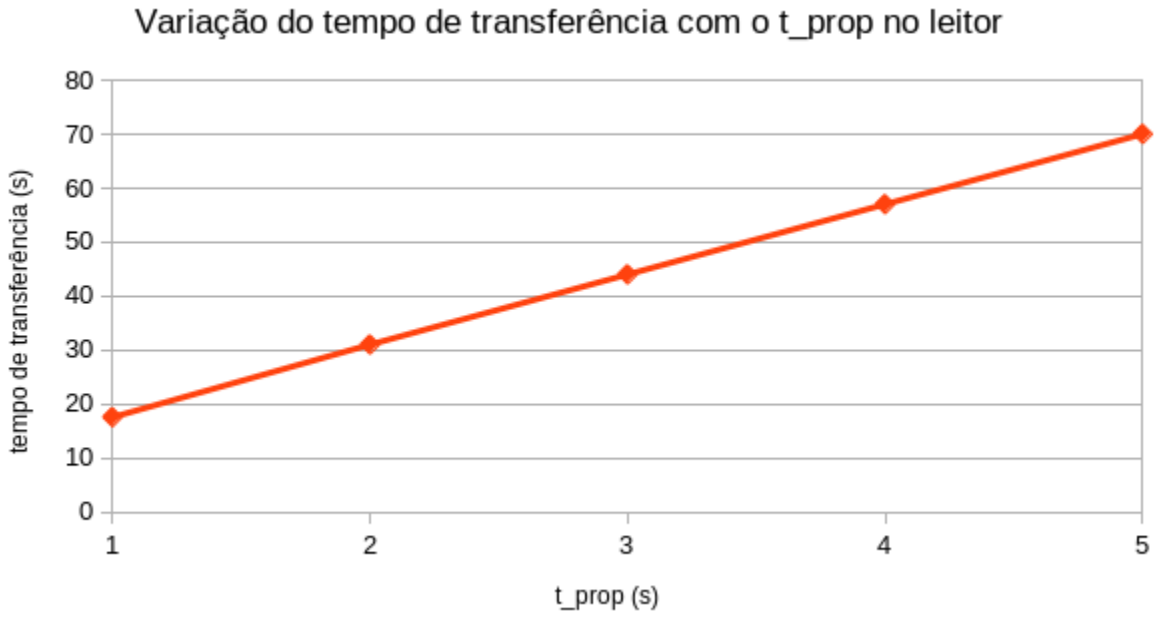
Os valores por defeito usados para quando um determinado parâmetro não está a ser testados são os seguintes: FER = 0, t_prop = 0, baudrate = B38400 e tamanho de pacote = 1000. O programa foi testado com cada conjunto de argumentos três vezes e os tempos apresentados resultam do cálculo da média das três repetições, excetuando os casos em que os programas não terminaram corretamente, nestas situações apenas foram usadas para cálculo da média as execuções que terminaram corretamente. É de salientar que ao testar a variação de FER, dada a aleatoriedade associada à simulação de erros, não foi possível medir nenhum tempo com FER = 90 e é por isso que os gráficos saltam de 80 para 100.

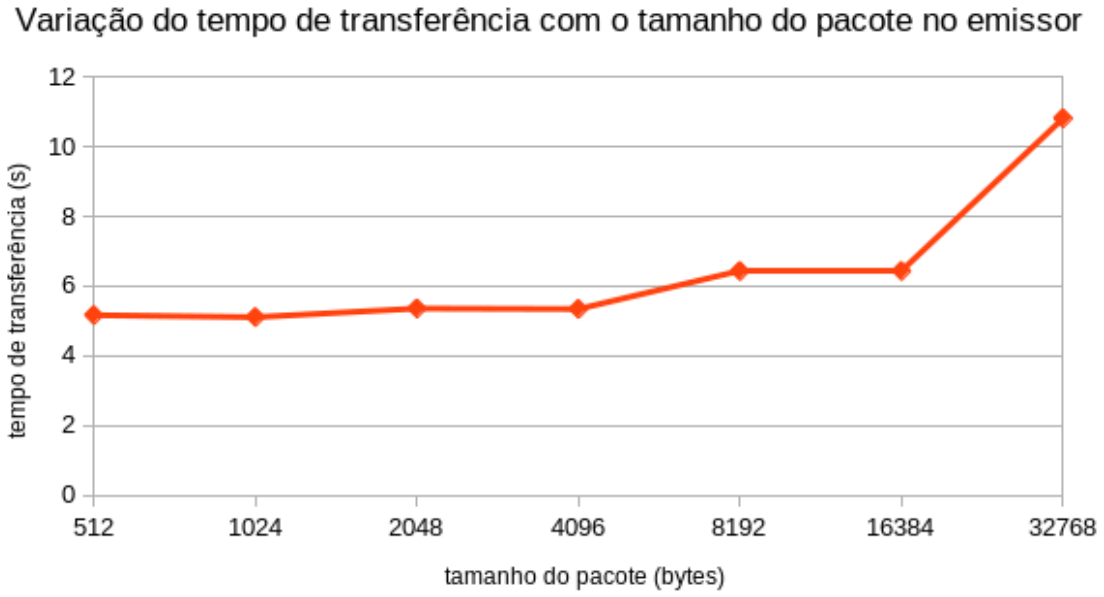
Relativamente à simulação de erros, as probabilidades de simular um erro no cabeçalho ou no campo de dados são de 30% e independente, ou seja, a probabilidade efetiva de simular um erro no campo de dados ou no cabeçalho é de 0.3 a multiplicar pelo FER fornecido aos programas e é possível que seja simulado um erro nos dois ou em nenhum deles, a implementação da emulação de erros é apresentada no anexo 2.

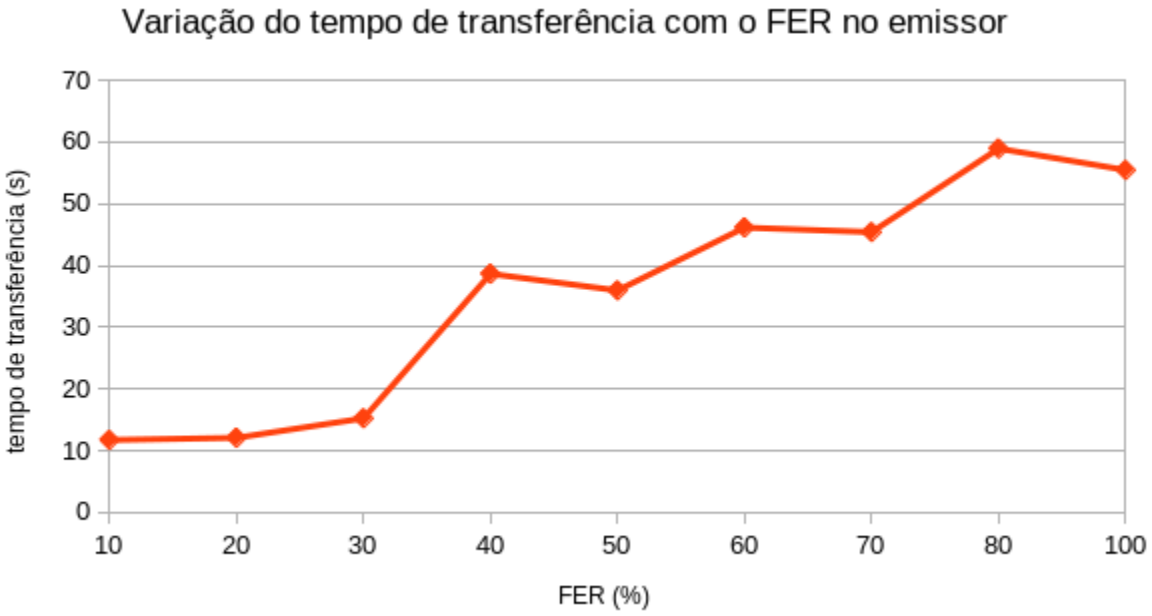


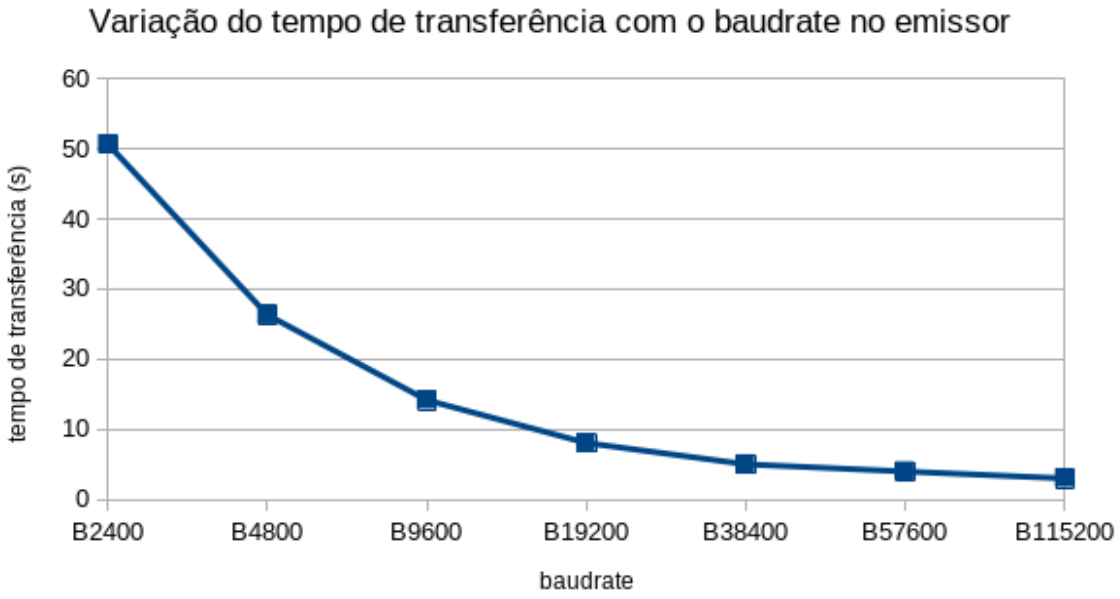

Variação do tempo de transferência com o FER no leitor
tempo de transferência (s)
0
10
20
30
40
50
60
70
10
20
30
40
50
60
70
80
100
FER (%)

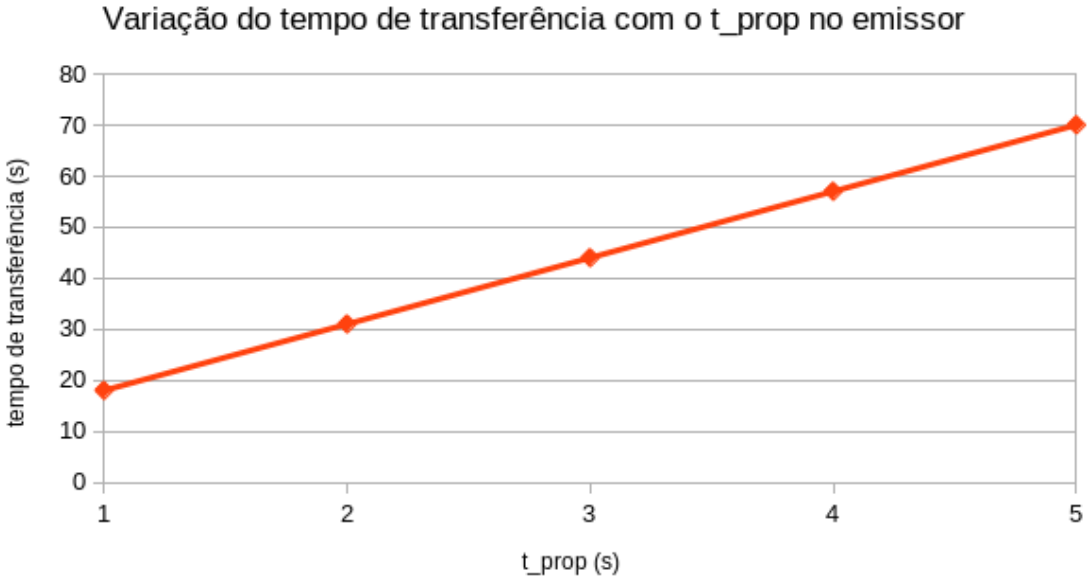

Variação do tempo de transferência com o tamanho de pacote no leitor
tempo da transferência (s)
0
2
4
6
8
10
12
512
1024
2048
4096
8192
16384
32768
tamanho do pacote (bytes)


Variação do tempo de transferência com o t_prop no leitor
tempo de transferência (s)
80
70
60
50
40
30
20
10
0
1
2
3
4
5
t_prop (s)


Variação do tempo de transferência com o tamanho do pacote no emissor
tempo de transferência (s)
12
10
8
6
4
2
0
512
1024
2048
4096
8192
16384
32768
tamanho do pacote (bytes)


Variação do tempo de transferência com o FER no emissor
tempo de transferência (s)
70
60
50
40
30
20
10
0
10
20
30
40
50
60
70
80
100
FER (%)


Variação do tempo de transferência com o baudrate no emissor
tempo de transferência (s)
60
50
40
30
20
10
0
B2400
B4800
B9600
B19200
B38400
B57600
B115200
baudrate


Variação do tempo de transferência com o t_prop no emissor
tempo de transferência (s)
80
70
60
50
40
30
20
10
0
1
2
3
4
5
t_prop (s)