Primeiro trabalho laboratorial

Ligação de dados

Mestrado Integrado em Engenharia Informática e Computação

Redes de Computadores

Grupo 8 Turma 2
João Romão - up201806779
Tiago Alves - up201603820

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

10 de novembro de 2020

Conteúdo

1 Sumário

Este relatório foi desenvolvido no âmbito do primeiro trabalho prático da unidade curricular de Redes de Computadores. O tema principal do trabalho é ligação de dados, tendo como objetivo principal o desenvolvimento de uma aplicação capaz de transferir ficheiros de um computador para outro através de uma ligação por cabo entre as portas de série de cada máquina.

O trabalho foi concluido com sucesso, tendo sido desenvolvida uma aplicação capaz de transferir ficheiros sem perda de dados e capaz de recuperar de possíveis erros que possam ocorrer durante a transmissão de informação.

2 Introdução

O objetivo deste trabalho consiste na implementação de um protocolo de ligação de dados que siga as indicações descritas no guião fornecido, permitindo a tranferência de dados e recuperação da ligação perante eventuais erros que possam ocorrer durante a transmissão.

Outro objetivo do trabalho consistia em testar o protocolo com uma aplicação de transferência de ficheiros.

O relatório tem como objetivo principal clarificar o modo de desenvolvimento do software e explicar a componente teórica por detrás do trabalho realizado, possuindo a seguinte estrutura:

- **Arquitetura**

  Explicação dos blocos funcionais e interfaces presentes.

- **Estrutura do código**

  Referência às API's, estruturas de dados, funções implementadas e relação com a arquitetura.

- **Casos de uso principais**

  Identificação e abordagem das sequências de chamadas de funções.

- **Protocolo de ligação lógica**

  Descrição da estratégia de implementação adotada.

- **Protocolo de aplicação**

  Descrição da implementação dos elementos constituintes da camada da aplicação.

- **Validação**

  Descrição dos testes efetuados e e apresentação de resultados.

- **Eficiência do protocolo de ligação de dados**

  Divulgação de estatísticas relativas à eficiência do protocolo de ligação de dados.

- **Conclusão**

  Observações finais relativas ao desenvolvimento do projeto.

3 Arquitetura

O trabalho está organizado em duas camadas principais: a camada do protocolo de ligação de dados e a camada da aplicação.

A **camada de ligação de dados** é responsável por implementar as diversas funções que constituem o protocolo, garantindo implementação de uma estrutura correta das tramas de supervisão e informação e os diversos comportamentos que o receptor e o emissor devem tomar perante o atual estado do programa.

A **camada de aplicação** faz uso da camada de ligação de dados, sendo responsável pela transferência dos ficheiros, através do envio e receção de pacotes de controlo e de dados.

Existe uma total independência entre as camadas, na camada de ligação, não é feito qualquer processamento que incida sobre o cabeçalho dos pacotes a transportar em tramas de Informação e a camada de aplicação não conhece detalhes do protocolo de ligação de dados, apenas a forma como acede ao serviço.

4 Estrutura do código

4.1 Aplicação

O código relativo à implementação da camada de aplicação encontra-se presente nos ficheiros application.c, application.h e main.c.

Principais funções presentes em application:

- **sendFile** - função que realiza o envio do ficheiro do emissor para o recetor.
- **retrieveFile** - função responsável por receber o ficheiro transmitido.
- **sendControlPacket** - constrói e envia um pacote de controlo.
- **parseCtrlPacket** - interpreta o conteúdo do pacote de controlo recebido.
- **sendDataPacket** - constrói e envia um pacote de dados.
- **parseCtrlPacket** - interpreta o conteúdo do pacote de dados recebido.

O código presente em **main.c** é responsável por interpretar os argumentos da linha de comandos e invocar **sendFile** ou **retrieveFile** caso estejamos perante o emissor ou o receptor.

4.2 Protocolo de ligação de dados

O código relativo ao protocolo de ligação de dados encontra-se distribuido pelos seguintes ficheiros: protocol.c, message.c, state_machine.c e data_stuffing.c.

Em **protocol.c** encontram-se implementadas as funções realtivas à API da porta série: **llopen**, **llwrite**, **llread** e **llclose**.

Em **message.c** econtra-se código relativo à contrução, envio e receção de tramas, destacando-se as seguintes funções:

- **write_supervision_message / write_info_message** - contrução e envio de tramas de supervisão e informação, respetivamente.
- **write_supervision_message_retry / write_info_message_retry** - envio de tramas de supervisão e informação, respetivamente, com recurso às funções anteriormente referidas, permitindo retransmissão de informação com recurso a um sinal de alarme.
- **readMessage** - leitura de informação byte a byte, com recurso a uma máquina de estados.

Em **state_machine.c** encontra-se a implementação da máquina de estados utilizada para leitura de informação, destacando-se a função **handleState** que interpreta o estado atual da máquina, chamando a função adequada para a atualizar consuante o caractér recebido.

Em **data_stuffing.c** econtram-se as funções **stuffdata** e **unstuffData**, responsáveis por garantir o mecanismo de byte suffing.

5 Casos de uso principais

Sendo o objetivo principal do trabalho a criação de uma aplicação que transfira ficheiros entre duas máquinas através da porta série o utilizador terá que escolher certos parâmetros para poder utilizar o programa.

Execução do programa

Do lado do **emissor** terá que escolher o ficheiro a enviar, o número que representa a porta série em uso e o tamanho de cada pacote de dados. Deste modo o programa poderá ser executado do seguinte modo:

- ./main.out <número da porta série> emissor <caminho relativo do ficheiro a enviar> <tamanho de cada pacote de dados>
- (exemplo) ./main.out 0 emissor pinguim.gif 255

Do lado do **receptor** apenas terá que escolher o número que representa a porta série em uso. Deste modo o programa poderá ser executado do seguinte modo:

- ./main.out <número da porta série> receptor
- (exemplo) ./main.out 0 receptor

Programa executado como emissor

Uma vez inseridos os argumentos na linha de comandos o programa executará a função **sendFile** responsável pelas ações do emissor, desencadeando a seguinte sequência de chamadas:

- Estabelecimeto da ligação com auxílio de **llopen**
- Envio do pacote de controlo START com **sendControlPacket**
- Leituras sucessivas do ficheiro a transferir, conteúdo que fará parte de um pacote de dados.
- Envios sucessivos dos pacotes de dados correspondentes ao conteúdo lido com **sendDataPacket**
- Envio do pacote de controlo END com **sendControlPacket**
- Encerramento da ligação com a chamada de **llclose**

No envio de cada pacote de dados é chamada a função **llwrite**.

Programa executado como receptor

Uma vez inseridos os argumentos na linha de comandos o programa executará a função **retrieveFile** responsável pelas ações do receptor, desencadeando a seguinte sequência de chamadas:

- Estabelecimeto da ligação com auxílio de **llopen**
- Chamadas sucessivas de **llread**
- Por cada chamada de llread execução de **parsePackets**, identificando os pacotes de controlo START, END e pacotes de dados.
- Encerramento da ligação com a chamada de **llclose**

6 Protocolo de ligação lógica

6.1 llopen

```
int llopen(char* port, conn_type connection_type);
```

Função responsável por estabelecer uma ligação através da porta série.

Quando a função é invocada pelo **emissor** é enviado o comando **SET** através da porta série e é aguardada uma resposta por parte do recetor, comando **UA**.

Caso o tempo de expera pela resposta exceda o **time out** definido, o comando **SET** é reenviado ficando o emissor novamente à espera de nova resposta.

Este mecanismo é conseguido graças à implementação de um **alarme** e caso o número de tentativas de envio do comando **SET** excedam um limite definido o emissor encerra.

Caso a função seja invocada pelo **receptor**, este espera pela receção do comando **SET** e de seguida envia o comando **UA**.

Para garantir este funcionamento são chamadas as funções **write_supervion_message** e **write_supervision_message_retry**.

6.2 llwrite

```
int llwrite(int fd, char* buffer, int length);
```

Função invocada pelo **emissor** e é responsável pela construção e envio de tramas de informação, receção de respostas de reconhecimento por parte do receptor e garante o cumprimento do protocolo de ligação de dados.

O *framing* do buffer recebido é realizado pela função **write_info_message** que porteriormente também envia a mensagem. Durante o processo de *framing* é efetuado o cálculo do BCC1 e do BCC2, o último através da invocação de **buildBCC2**, e o *stuffing* dos dados recebidos, através da invocação de **stuffData**.

O envio da trama de informação possui o mesmo mecanismo de **time out** referido na função **llopen**. Neste caso o **alarme** é acionado quando a mensagem é enviada até à receção de uma resposta **RR** ou **REJ**.

Caso o comando recebido seja do tipo **REJ**, a trama de informação anteriormente enviada é reenviada, garantindo que não há perdas de informação.

6.3 llread

```
int llread(int fd, char* buffer);
```

Função invocada pelo **receptor** e é responsável por receber e interpretar as tramas de informação através da porta série, enviando uma resposta adequada ao emissor.

É realizado *destuffing* da mensagem recebida, com a invocação da função **unstuffData**.
Uma vez que tramas com erros no cabeçalho são automaticamente descartadas, devido à implementação da máquina de estados, a função apenas verifica se o valor do **BCC2** se encontra correto, enviando **REJ** caso não se verifique.
Também é realizada uma verificação ao nível do **número de sequência** da trama recebida, de modo a perceber se se trata de uma trama de informação repetida. Caso seja, os dados são automaticamente descartados e é enviado o comando **RR**.
O campo de dados da trama de informação é retornado através do paraâmetro **char* buffer** para futura interpretação por parte da camada da aplicação.

6.4 llclose

```
int llclose(int fd);
```

Função responsável por encerrar a ligação através da porta série, tendo um comportamento semelhante à função **llopen** com a mesma implementação de alarme, diferindo nos comandos enviados.

Quando a função é invocada pelo **emissor** é enviado o comando **DISC** através da porta série e é aguardada uma resposta por parte do recetor, comando **DISC**.
Finalmente é enviado o comando **UA**.

Caso a função seja invocada pelo **receptor**, este espera pela receção do comando **DISC** e de seguida envia o comando **DISC**.
Finalmente espera pela receção do comando **UA**, sendo a ligação através da porta série efetivamente terminada de seguida.

7 Protocolo de aplicação

Responsável pela transmissão do ficheiro, que se econtra repartida pela leitura e escrita do ficheiro assim como pelo envio e receção de pacotes de controlo e de dados.

7.1 Pacotes ao nível da aplicação

A aplicação possui pacotes de dados e pacotes de controlo, podendo estes últimos ser ainda classificados como inicial e final, sendo todos eles distinguidos pelo **campo de controlo** que pode tomar os seguintes valores, definidos no código por macros apropriadas:

- 1 (PACKET_CTRL_DATA) - pacote de dados.
- 2 (PACKET_CTRL_START) - pacote de controlo inicial.
- 3 (PACKET_CTRL_END) - pacote de controlo final.

7.2 Envio de pacotes de aplicação

O **envio de pacotes de controlo** é efetuado pela função **sendControlPacket**, a qual contrói o pacote e envia-o para a porta série com recurso à função **llwrite**.

```
int sendControlPacket(int fd,char * filename,int fileSize,int ctrl);
```

O parâmetro ctrl pode tomar os valores PACKET_CTRL_START ou PACKET_CTRL_END.

Para além dos paraâmetros referidos no enunciado, foi **acrescentado um terceiro parâmetro** representativo do **tamanho de cada pacote de dados**, facilitando a execução do programa, evitando que ao executar o programa receptor se introduza um número diferente do que foi introduzido no programa emissor.

O **envio de pacotes de dados** é efetuado pela função **sendDataPacket**, a qual contrói o pacote e envia-o para a porta série com recurso à função **llwrite**.

```
int sendDataPacket(int fd, char * data, short dataSize, int nseq);
```

7.3 Receção e interpretação de pacotes de aplicação

A receção de pacotes de aplicação é realizada com o auxílio da função **llread**, recebendo o pacote de dados a interpretar.

A interpretação dos pacotes da aplicação fica posteriormente a cabo da função **parsePackets**, que lê o valor do **campo de controlo** do pacote recebido. Consoante esse valor ou função **parseCtrlPacket** ou a função **parseDataPacket** será chamada.

Caso o campo de controlo indique que se trata de um pacote de controlo final, a função sinaliza que a leitura de pacotes de dados deve ser terminada através do valor de retorno.

```
int parsePackets(char* buffer,int buffer_size,char* filename,char* ctrl_packet)
```

A função **parseCtrlPacket** retira o tamanho e nome do ficheiro assim como também o tamanho de cada bloco de dados a ser lido.

```
int parseCtrlPacket(char * buffer, char* filename);
```

A função **parseDataPacket** retira o campo de dados presente no pacote recebido e escreve o conteúdo para o ficheiro cujo nome é recebido como paraâmetro.

```
int parseDataPacket(char* buffer, int nseq, char* filename);
```

8 Validação

De modo a testar o protocolo de ligação de dados e a camada de aplicação desenvolvida foram realizados diversos testes, obtendo resultados positivos para todos eles:

1. Envio do ficheiro fornecido (pinguim.gif).
2. Envio de ficheiros com diferentes tamanhos, até 50MB.
3. Envio de vários tipos de ficheiros (txt, gif, jpg, mp4)
4. Variação do tamanho dos pacotes de dados enviados.
5. Variação da capacidade de ligação (baudrate).
6. Cortar a ligação durante alguns segundos durante o envio do ficheiro.
7. Geração de ruído durante a transmissão do ficheiro.
8. Variação da percentagem de erros simulados.

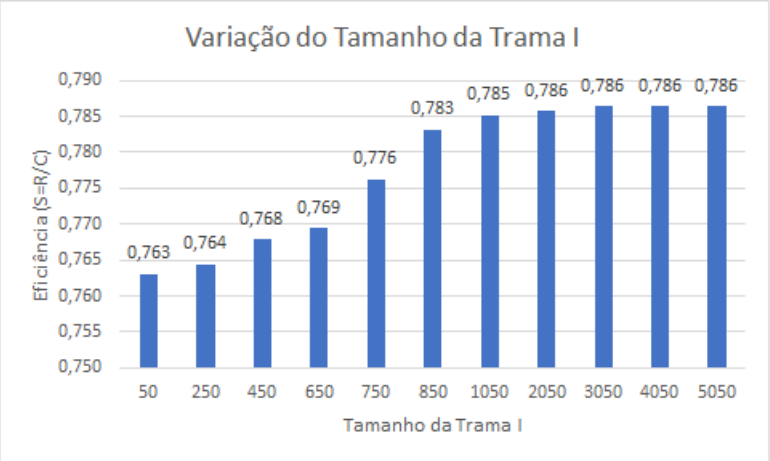
9 Eficiência do protocolo de ligação de dados

De forma a avaliar a eficiência do protocolo desenvolvido, foram realizados os três testes que se encontram de seguida.

Os testes foram todos realizados com um ficheiro gif de 10968 bytes, sendo que foram feitos sempre dois testes, de modo a diminuir o desvio de dados. Todas as tabelas estão presentes no anexo II.

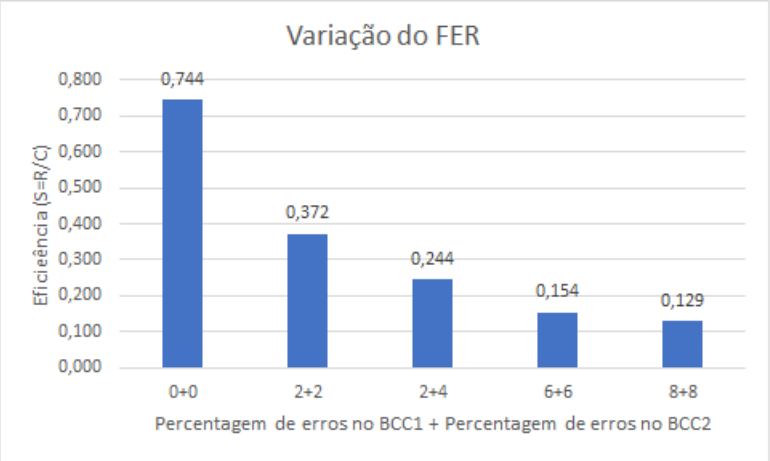
9.1 Variação do tamanho das tramas I

Nesta secção são apresentados os resultados obtidos variando o tamanho da trama I. Com este gráfico podemos concluir que a eficiência da aplicação aumenta com o tamanho da trama, convergindo para uma eficiência de 0.786 em tamanhos mais elevados. Isto verifica-se visto que cada envio contém mais informação, reduzindo o número de tramas.



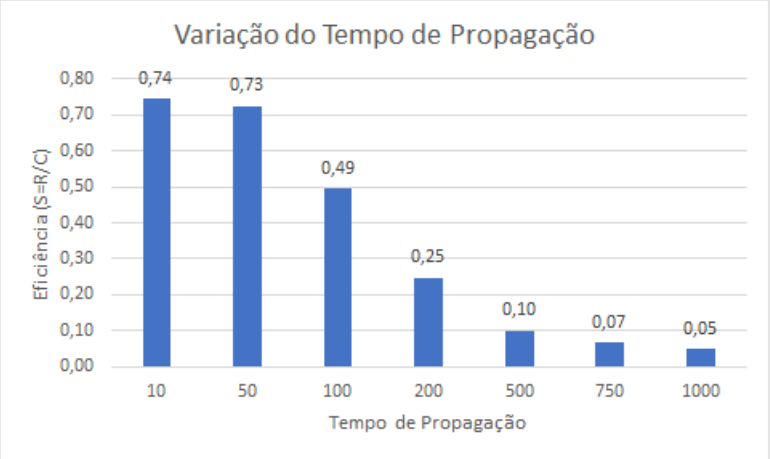
9.2 Variação do FER

Nesta subsecção apresenta-se os resultados obtidos variando a percentagem de erros simulados, para um tamanho de trama e baudrate constantes. Após analisar o gráfico pode-se concluir que o FER tem um impacto significativo no programa. Isto deve-se primeiramente ao facto de que, quando é gerado um erro no BCC1, irá ocorrer timeout, o que consequentemente resultará na ausência de resposta por parte do receptor por três segundos, afetando o tempo de execução. Os erros ocorridos no BCC2 não têm um impacto tão grande visto que apenas implica o reenvio da trama, sendo este imediato.



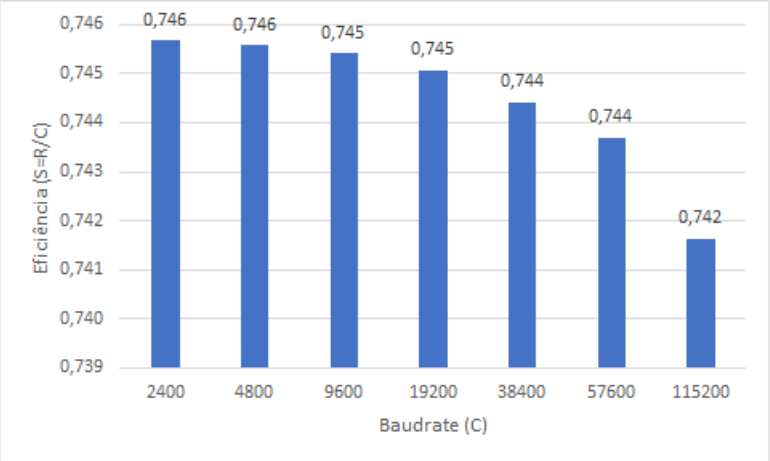
9.3 Variação do tempo de propagação

Os seguintes resultados são fruto da variação do tempo de propagação de cada trama de informação. Após analisar os dados, conclui-se naturalmente que com o aumento deste tempo a eficiência do programa decresce. Isto deve-se ao facto de se estar a aumentar o tempo necessário para enviar/receber uma trama. Estes resultados foram obtidos usando a função *usleep()*, de modo a introduzir o atraso.



9.4 Variação do Baudrate

Para um tamanho de trama constante de 255 bytes, os resultados obtidos variando o Baudrate foram os apresentados de seguida. Como podemos verificar a eficiência diminui com a capacidade de ligação.



9.5 Caracterização teórica de Stop&Wait

Falando agora de um ponto de vista mais teórico sobre o protocolo Stop Wait, após a transmissão de uma trama de informação, o emissor espera por uma confirmação positiva por parte do recetor, denominada por acknowledgment, ACK. Depois de ter recebido esta mensagem, o emissor sabe que a trama foi enviado com sucesso, podendo assim enviar uma nova ao recetor. No caso contrário, em que existe um erro no mensagem, é enviado um NACK, negative acknowledgment, por parte do recetor, de modo a sinalizar ao emissor que tem de reenviar a trama.

O mecanismo usado no nosso programa foi fortemente baseado neste protocolo Stop Wait, sendo que quando o emissor manda uma trama, espera sempre uma resposta. Caso o recetor receba os dados sem erros, é enviada uma mensagem positiva RR, o que sinaliza ao emissor que pode enviar uma nova trama. Caso contrário, é enviada uma mensagem negativa, REJ, que sinaliza o emissor que tem de reenviar a trama. O Nr destas tramas de resposta varia conforme o emissor tenha enviado uma trama de Ns 0 ou 1, de modo a este saber que trama deve mandar e de modo a ajudar no tratamento de duplicados. Se o Ns enviado pelo emissor for 0, deve receber uma resposta RR (Nr=1) quando não há erros e REJ (Nr=0) quando há. Contrariamente, se o Ns enviado pelo emissor for Ns=1, então deverá receber uma resposta RR (Nr=0) no caso de não haver erros, ou REJ (Nr=1) caso haja.

10 Conclusões

Durante as últimas semanas o grupo teve em mãos o desenvolvimento de uma aplicação capaz de transferir ficheiros através de uma porta de série, tendo sempre em conta robustez contra erros e boas práticas de programação. A aplicação cumpriu todos os requisitos, tendo conseguido enviar vários tipos de ficheiros de diferentes tamanhos sem problema, sendo ainda resistente a erros e capaz de retomar o envio da mensagem em caso de erros. Como já foi referido, as duas camadas são independentes entre si, sendo que na camada de ligação de dados não é feito qualquer processamento que incida sobre os pacotes, e sendo que a camada da aplicação não conhece os detalhes do protocolo de dados, apenas acedendo a elas como serviço. Em conclusão, apesar das dificuldades que surgiram, devido à incapacidade de testar o nosso trabalho no laboratório, o objetivo foi cumprido e a sua elaboração contribuiu positivamente para o nosso conhecimento neste tema.

11 Anexo I - Código fonte

11.1 application.c

```
#include "protocol.h"
#include "message.h"
#include "file_handler.h"
#include "application.h"

int data_block_size;

int sendFile(char * port_num,char * filename,int data_size){
    data_block_size = data_size;
    setBlockSize(data_block_size);

    install_alarm();
    int fd = llopen(port_num,EMISSOR);
    if (fd == -1) return -1;

    FILE* file = fopen(filename, "rb");
    if (file == NULL){
        perror("Error reading file");
        return -1;
    }

    int fileSize = findSize(filename);

    if(sendControlPacket(fd,filename,fileSize,PACKET_CTRL_START) != 0){
        printf("Error sending controll packet\n");
        return -1;
    }

    int ret = 0;
    char* buffer = calloc(data_block_size - 4, sizeof(char));
    int i = 0;

    while (TRUE){
        ret = fread(buffer , sizeof(char), data_block_size - 4, file);
        if (ret <= 0) break;

        if(sendDataPacket(fd, buffer, ret, i) < 0){
            printf("\nError sending data packet\n");
            return -1;
        }

        i++;
        progressBar(EMISSOR,(1.0*i*(data_block_size-4)/fileSize) *100);
    }
    fclose(file);

    if(sendControlPacket(fd,filename,fileSize,PACKET_CTRL_END) != 0){
        printf("Error sending controll packet\n");
        return -1;
    }

    if(llclose(fd)<0){
        printf("Error closing connection\n");
    } else printf("Connection closed successfully\n");

    return 0;
}

int retrieveFile(char * port_num){

    char* buffer = malloc(CTRL_PACKET_SIZE(FILENAME_MAX));
    int fd = llopen(port_num, RECEPTOR);
    if (fd == -1) return -1;

    char filename[FILENAME_MAX];
    memset(filename, 0, FILENAME_MAX);

    char ctrl_packet[CTRL_PACKET_SIZE(FILENAME_MAX)];
    memset(ctrl_packet, 0, CTRL_PACKET_SIZE(FILENAME_MAX));
```

```
    printf("Receiving file...\n");
    int first_loop = TRUE;

    while(TRUE){
        int buffer_size = llread(fd, buffer);
        if (buffer_size == -1){
            printf("LLREAD failure\n");
            return -1;
        }
        else if (buffer_size == -2 || buffer_size == -3){
            continue;
        }

        int r = parsePackets(buffer, buffer_size,filename,ctrl_packet);
        if (r == -1){
            printf("Error parsing packets\n");
            return -1;
        }
        else if (r == -2){
            printf("File received successfully\n");
            break;
        }

        if (first_loop){
            buffer = realloc(buffer, data_block_size);
            setBlockSize(data_block_size);
            first_loop = FALSE;
        }

        memset(buffer, 0, data_block_size);
    }

    if(llclose(fd)<0){
        printf("Error closing connection\n");
    } else printf("Connection closed successfully\n");

    return 0;
}

int sendControlPacket(int fd,char * filename,int fileSize,int ctrl){
    u_int file_name_size = strlen(filename);
    u_int file_size_size = sizeof(u_int);
    u_int block_size = sizeof(u_int);
    if(file_name_size > 255) {
        printf("Error, filename cannot be greater that 255 characters\n");
    }

    int controlPacketSize = CTRL_PACKET_SIZE(file_name_size);
    char * controlPacket = malloc(controlPacketSize);

    controlPacket[PACKET_CTRL_IDX] = ctrl;

    //FILESIZE
    controlPacket[CTRL_SIZE_T_IDX] = CTRL_SIZE_OCTET;
    controlPacket[CTRL_SIZE_L_IDX] = file_size_size;
    for (int i = 0; i < file_size_size; i++)
        controlPacket[CTRL_SIZE_V_IDX+i] = (u_int8_t) (fileSize >> (8 * i));

    //FILENAME
    controlPacket[CTRL_NAME_T_IDX] = CTRL_NAME_OCTET;
    controlPacket[CTRL_NAME_L_IDX] = file_name_size;
    for (int i = 0; i < file_name_size; i++)
        controlPacket[CTRL_NAME_V_IDX+i] = filename[i];

    //DATABLOCKSIZE
    controlPacket[CTRL_NAME_V_IDX + file_name_size] = 2;
    controlPacket[CTRL_NAME_V_IDX + file_name_size + 1] = block_size;
    for (int i = 0; i < block_size; i++)
        controlPacket[CTRL_NAME_V_IDX + file_name_size + 2 + i] = (u_int8_t) (
            data_block_size >> (8 * i));

    int ret = llwrite(fd, controlPacket, controlPacketSize);
```

```
    free(controlPacket);

    return ret;
}
int parseCtrlPacket(char * buffer,char * filename){

    if ((buffer[PACKET_CTRL_IDX] != PACKET_CTRL_START) && (buffer[
        PACKET_CTRL_IDX] != PACKET_CTRL_END ) ){
        printf("Error recieving control packet\n");
        return -1;
    }
    if (buffer[CTRL_SIZE_T_IDX] != CTRL_SIZE_OCTET){
        printf("Error recieving size of file\n");
        return -1;
    }
    if (buffer[CTRL_NAME_T_IDX] != CTRL_NAME_OCTET){
        printf("Error recieving name of file\n");
        return -1;
    }

    //PARSE FILE SIZE
    u_int fileSize = 0;
    memcpy(&fileSize, buffer + CTRL_SIZE_V_IDX, buffer[CTRL_SIZE_L_IDX]);

    //PARSE FILE NAME
    memset(filename + buffer[CTRL_NAME_L_IDX], 0, 1);
    memcpy(filename, buffer + CTRL_NAME_V_IDX, buffer[CTRL_NAME_L_IDX]);

    if ((buffer[PACKET_CTRL_IDX] != PACKET_CTRL_START))
        printf("filename: %s\n", filename);

    //PARSE DATA BLOCK SIZE
    int file_name_size = strlen(filename);
    memcpy(&data_block_size, buffer + CTRL_NAME_V_IDX + file_name_size + 2,
        buffer[CTRL_NAME_V_IDX + file_name_size + 1]);

    //APAGAR TODO
    char *tmp = strdup(filename);
    strcpy(filename, "Images/"); //Put str2 or annother string that you want at
         the begining
    strcat(filename, tmp);  //concatenate previous str1

    return 0;
}

int sendDataPacket(int fd,char * data,short dataSize,int nseq){
    int dataPacketSize = DATA_PACKET_SIZE(dataSize);
    char * dataPacket = malloc(dataPacketSize);

    dataPacket[DATA_CTRL_IDX] = DATA_CTRL_VALUE;
    dataPacket[DATA_N_SEQ_IDX] = nseq % 255;
    dataPacket[DATA_L1_IDX] = (u_int8_t) (dataSize);
    dataPacket[DATA_L2_IDX] = (u_int8_t) (dataSize >> 8);

    for (int i = 0; i < dataSize; i++){
        dataPacket[i + DATA_P_INITIAL_IDX] = data[i];
    }

    int ret = llwrite(fd,dataPacket,dataPacketSize);
    free(dataPacket);
    return ret;
}

int parseDataPacket(char * buffer, int nseq,char * filename){
    int dataSize = (u_int8_t) buffer[DATA_L1_IDX] + (u_int8_t) buffer[
        DATA_L2_IDX] * 256;
    char * file_send = calloc(dataSize, sizeof(char));
    memcpy(file_send, buffer + DATA_INF_BYTE, dataSize);
    write_file(filename, file_send, dataSize);
    return 1;
}
```

```c
int parsePackets(char * buffer, int buffer_size,char * filename,char *
    ctrl_packet){

    char cc = buffer[0];
    int nseq = 0;

    if(cc == PACKET_CTRL_END){
        parseCtrlPacket(buffer,filename);

        if(compareCtrlPackets(buffer,ctrl_packet) < 0){
            printf("Error: Control packets are not compatible!\n");
            return -1;
        }
        return -2;
    }
    else if (cc == PACKET_CTRL_START){
        memcpy(ctrl_packet,buffer,buffer_size);
        parseCtrlPacket(buffer,filename);
        return 0;
    }
    else if(cc != PACKET_CTRL_DATA){
        printf("Error recieving packet\n");
        for (int i = 0; i < buffer_size; i++){
            printf("%02x ", (unsigned char) buffer[i]);
        }
        printf("\n\n");
        printf("Error parsing packets cc: %02x\n",cc);
        return -1;
    }
    else{
        nseq = (int)buffer[DATA_N_SEQ_IDX];
        if(!parseDataPacket(buffer,nseq,filename)){
            printf("Error parsing data packet: %d",nseq);
            return -1;
        }
    }
    return 0;

}

void progressBar(conn_type type, int progress) {

    char * msg;
    switch (type) {
        case RECEPTOR:
            msg = "Receiving file |";
            break;
        case EMISSOR:
            msg = "Sending file |";
            break;
        default:
            msg = "";
    }
    if (progress < 100){
        printf("\r%s", msg);
        for (int i = 0; i < PROGRESS_BAR_SIZE; i++){
            if ((int)(progress*0.01*PROGRESS_BAR_SIZE) < i)
                printf(" ");
            else
                printf("=");
        }
        printf("| %d%%", progress);
        fflush(stdout);
    }

    if (progress >= 100){
        printf("\r%s", msg);
        for (int i = 0; i < PROGRESS_BAR_SIZE; i++)
            printf("=");
        printf("| %d%%\n", 100);
        fflush(stdout);
    }
}
```

```
int compareCtrlPackets(char * ctrl1,char * ctrl2){

    u_int fileSize = 0;
    memcpy(&fileSize, ctrl1 + CTRL_SIZE_V_IDX, ctrl1[CTRL_SIZE_L_IDX]);
    //printf("File Size: %u\n",fileSize);

    u_int fileSize2 = 0;
    memcpy(&fileSize2, ctrl2 + CTRL_SIZE_V_IDX, ctrl2[CTRL_SIZE_L_IDX]);
    if(fileSize != fileSize2){
        return -1;
    }

    char filename[FILENAME_MAX];
    memset(filename,0,FILENAME_MAX);

    //char filename[buffer[CTRL_NAME_L_IDX]];
    memcpy(filename, ctrl1 + CTRL_NAME_V_IDX, ctrl1[CTRL_NAME_L_IDX]);

    char filename2[FILENAME_MAX];
    memset(filename2,0,FILENAME_MAX);

    //char filename[buffer[CTRL_NAME_L_IDX]];
    memcpy(filename2, ctrl2 + CTRL_NAME_V_IDX, ctrl2[CTRL_NAME_L_IDX]);
    //printf("%s   %s",filename,filename2);
    if(strcmp(filename,filename2) != 0){
        return -1;
    }

    return 0;
}
```

11.2 connection.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <strings.h>
#include "connection.h"
#include "protocol.h"

static struct termios oldtio;

int open_connection(link_layer layer) {
    int fd;
        struct termios newtio;

    fd = open(layer.port, O_RDWR | O_NOCTTY );
        if (fd <0) {
        perror(layer.port);
        exit(-1);
    }

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

        bzero(&newtio, sizeof(newtio));
        newtio.c_cflag = layer.baud_rate | 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]    = (RESEND_DELAY-1)*10;   /* inter-character timer
             unused */
```

```
        newtio.c_cc[VMIN]     = 0;   /* blocking read until 5 chars received */

    tcflush(fd, TCIOFLUSH);

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

        printf("Connection opened\n");

    return fd;
}

int close_connection(int fd){

        sleep(1);

        if ( tcsetattr(fd,TCSANOW,&oldtio) == -1) {
                perror("tcsetattr");
                exit(-1);
        }

        return close(fd);
}
```

11.3 connection.h

```
#define BAUDRATE B38400
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define NUM_TRANSMISSIONS 3
#define LAYER_TIMEOUT 3

typedef struct{
  char port[20];                /*Dispositivo /dev/ttySx, x = 0, 1*/
  int baud_rate;                 /*Velocidade de transmiss o*/
  unsigned int timeout;         /*Valor do temporizador: 1 s*/
  unsigned int num_transmissions; /*N mero de tentativas em caso defalha*/
} link_layer;

/**
 * @brief Opens serial port connection.
 *
 * @param layer Struct containing necessary information to open connection.
 * @return int Serial port file descriptor.
 */
int open_connection(link_layer layer);

/**
 * @brief Closes serial port connection.
 *
 * @param fd Serial port file descriptor.
 * @return int Return value of function close(fd).
 */
int close_connection(int fd);
```

11.4 data_stuffing.c

```
#include "data_stuffing.h"
#include "message.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

data_stuff stuffData(char* buffer,int length){
        int stuff_data_index = 0;
        char* stuffed_data_buffer = calloc(2*length, sizeof(char));
        for (int i = 0; i < length; i++)
        {
                if(buffer[i] == FLAG){
                        stuffed_data_buffer[stuff_data_index++] =ESC;
                        stuffed_data_buffer[stuff_data_index++] =
                            FLAG_STUFFING_BYTE;
```

```
                } else if( buffer[i] == ESC){
                        stuffed_data_buffer[stuff_data_index++] =ESC;
                        stuffed_data_buffer[stuff_data_index++] =
                            ESC_STUFFING_BYTE;
                } else{
                        stuffed_data_buffer[stuff_data_index++] = buffer[i];
                }
        }

        data_stuff data;
        data.data = stuffed_data_buffer;
        data.data_size = stuff_data_index;

        return data;

}

data_stuff unstuffData(char * buffer,int length){
    data_stuff data_stuff;

    int i = DATA_INF_BYTE;
    int unstuffed_data_index = 0;
    char* unstuffed_data = calloc(length, sizeof(char));
    for (int j = 0; j < DATA_INF_BYTE; j++){
        unstuffed_data[unstuffed_data_index++] = buffer[j];
    }
    while (i < length-2){
        if (buffer[i] == ESC){
            i++;
            if(buffer[i] == FLAG_STUFFING_BYTE){
                unstuffed_data[unstuffed_data_index++] = FLAG;
            } else if(buffer[i] == ESC_STUFFING_BYTE){
                unstuffed_data[unstuffed_data_index++] = ESC;
            } else printf("ERROR UNSTUFFING DATA\n");
        } else unstuffed_data[unstuffed_data_index++] = buffer[i];
        i++;
    }

    for (; i < length; i++){
        unstuffed_data[unstuffed_data_index++] = buffer[i];
    }

    data_stuff.data = unstuffed_data;
    data_stuff.data_size = unstuffed_data_index;

    return  data_stuff;
}
```

11.5 data_stuffing.h

```
/* MACROS for byte stuffing */
#define FLAG_STUFFING_BYTE 0x5e
#define ESC_STUFFING_BYTE 0x5d

typedef struct {
    char * data;       /** Stuffed Data */
    int data_size;      /** Number of bytes after stuffing data*/
} data_stuff;

/**
 * @brief Applies data stuffing to data received in parameter buffer.
 *
 * @param buffer Data to be stuffed.
 * @param length Size, in bytes, of data to be stuffed.
 * @return data_stuff Struct containing stuffed data and its size.
 */
data_stuff stuffData(char* buffer,int length);

/**
 * @brief Reverts data stuffing previously applied.
 *
 * @param buffer Data to be unstuffed.
 * @param length Size, in bytes, of data to be unstuffed.
```

```
 * @return data_stuff Struct containing original data and its size.
 */
data_stuff unstuffData(char * buffer,int length);
```

11.6 file_handler.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#include "file_handler.h"

int write_file(char* file_path, char* buffer, int data_size) {
    FILE* file = fopen(file_path, "ab");
    if (file == NULL){
        perror("Error reading file");
    }

    if (fwrite(buffer, sizeof(char), data_size, file) < 0){
        printf("Error writing to file!\n");
    }
    fclose(file);
    return 0;
}

int findSize(char file_name[])
{
    // opening the file in read mode
    FILE* fp = fopen(file_name, "r");

    // checking if the file exist or not
    if (fp == NULL) {
        printf("File Not Found!\n");
        return -1;
    }

    fseek(fp, 0L, SEEK_END);

    // calculating the size of the file
    int res = ftell(fp);

    // closing the file
    fclose(fp);

    return res;
}
```

11.7 file_handler.h

```
/**
 * @brief Write content present in buffer, to file with name filepath
 *
 * @param file_path Name of the file where data will be written to.
 * @param buffer Data to be written.
 * @param data_size Size, in bytes, of buffer.
 * @return int
 */
int write_file(char* file_path, char* buffer, int data_size);

/**
 * @brief Finds size, in bytes, of file with name file_name.
 *
 * @param file_name Name of the file.
 * @return int Size of the file.
 */
int findSize(char file_name[]);
```

11.8 main.c

```
#include "protocol.h"
#include "application.h"
```

```c

#include <time.h>
#define BILLION 1000000000.0

volatile int STOP=FALSE;

int main(int argc, char** argv)
{
        arguments args = parse_arguments(argc,argv);
        srand(time(0));
    struct timespec start, end;

    if(args.role == EMISSOR){
        clock_gettime(CLOCK_REALTIME, &start);
        sendFile(args.port_num,args.filename,args.data_block_size);
        clock_gettime(CLOCK_REALTIME, &end);
        printf("Time consumed: %fs\n", end.tv_sec - start.tv_sec + (end.tv_nsec
            - start.tv_nsec) / BILLION - 1);

    }else if(args.role == RECEPTOR){
        retrieveFile(args.port_num);
    }

        return 0;
}
```

11.9 message.c

```c
#include "message.h"
#include "protocol.h"
#include "state_machine.h"
#include "data_stuffing.h"

int flag=1, conta=1;
static int ns = 0;
static int block_size = 0;

void setBlockSize(int value) {
        block_size = value;
}

int write_supervision_message(int fd, char cc_value){
    char trama[SUPERVISION_TRAMA_SIZE];
    trama[FLAGI_POSTION] = FLAG;
        trama[ADRESS_POSITION] = AREC;
        trama[CC_POSITION] = cc_value;
        trama[PC_POSITION] = (AREC) ^ (cc_value);
        trama[FLAGF_POSTION] = FLAG;

    return write(fd,trama,SUPERVISION_TRAMA_SIZE);
}


int write_info_message(int fd, char* data, int data_size, int cc_value){
        if(data_size==0) {
                printf("Recceived empty data buffer\n");
                return -1;
        }

        data_stuff stuffedData = stuffData(data,data_size);

        int trama_size = INFO_SIZE_MSG(stuffedData.data_size);
        char* trama = malloc(trama_size + 1); //space in case bcc2 needs
            stuffing

    trama[FLAGI_POSTION] = FLAG;
        trama[ADRESS_POSITION] = AREC;
        trama[CC_POSITION] = CC_INFO_MSG(cc_value);
        trama[PC_POSITION] = (AREC) ^ CC_INFO_MSG(cc_value);

        char bcc2 = buildBCC2(data, data_size);
        if (bcc2 == FLAG || bcc2 == ESC) {
                trama_size++;
```

```
                trama[trama_size - 3] = ESC;
                trama[trama_size - 2] = bcc2 ^ STUFF;
        }else {
                trama[trama_size - 2] = bcc2;
        }

        trama[trama_size - 1] = FLAG;

        memcpy(trama + DATA_INF_BYTE, stuffedData.data, stuffedData.data_size);

        return write(fd,trama,trama_size);
}

int write_supervision_message_retry(int fd, char cc_value, char cc_compare){
        int success = FALSE, rd;
        char* buffer;

        while(conta <= WRITE_NUM_TRIES && !success){
                if(flag){
                        alarm(RESEND_DELAY);
                        flag=0;

                        /* writing */
                        int n_bytes = write_supervision_message(fd,cc_value);
                        if(n_bytes <= 0){
                                printf("Error writing supervision message\n");
                        }

                        /* read message back */
                        buffer = readMessage(fd, &rd, 0, 1);

                        if (rd != n_bytes || buffer == NULL){
                                success = FALSE;
                        }
                        else if (buffer[CTRL_POS] == cc_compare){
                                success = TRUE;
                                reset_alarm();
                        }
                        else{
                                success = FALSE;
                        }
                }
        }
        if (success == TRUE){
                return 0;
        }
        printf("Finishing attempt after %d tries have received time out of %d
            seconds.\n", WRITE_NUM_TRIES, RESEND_DELAY);
        return -1;
}

int write_inform_message_retry(int fd, char * data, int dataSize){
        int success = FALSE, rd = 0;
        char* buffer;

        while(conta <= WRITE_NUM_TRIES && !success){
                if(flag){
                        alarm(RESEND_DELAY);
                        flag=0;

                        /* writing */
                        int n_bytes = write_info_message(fd, data, dataSize, ns
                            );
                        if(n_bytes <= 0){
                                printf("Error writing Inform message\n");
                        }

                        /* read message back */
                        buffer = readMessage(fd, &rd, 0, 1);

                        if (buffer == NULL){
                                success = FALSE;
                        }
```

```
                        else if (parseREJ(buffer)){
                                success = FALSE;
                                reset_alarm();
                        }
                        else if (parseRR(buffer)){
                                ns++;
                                success = TRUE;
                                reset_alarm();
                        }
                        else{
                                success = FALSE;
                        }
                }
        }

        if (success == TRUE){
                return 0;
        }
        printf("Finishing attempt after %d tries have received time out of %d
            seconds.\n", WRITE_NUM_TRIES, RESEND_DELAY);
        return -1;
}

char* readMessage(int fd, int* size, int i_message, int emissor){
        char r;
        int rd = 0, pos = 0, nulls_count = 0, error = 0;
        char* buffer = malloc(1);

        while (getStateMachine() != STOP){
                rd = read(fd, &r, 1);
                if (rd <= 0){
                        nulls_count++;
                        if (nulls_count == ERR_LIMIT || emissor)
                                return NULL;
                        else
                                continue;
                        update_state(START);
                }

                buffer = realloc(buffer, pos + 2);
                handleState(r, i_message, &error, emissor);
                buffer[pos++] = r;

                if (getStateMachine() == START){
                        free(buffer);
                        buffer = malloc(1);
                        pos = 0;
                        if (r != FLAG) error = 1;
                }
        }

        update_state(START);
        *size = pos;

        return buffer;
}

void atende(int signo){
        //printf("alarme # %d\n", conta);
        flag=1;
        conta++;
}

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

    if (sigaction(SIGALRM,&action,NULL) < 0){
        fprintf(stderr,"Unable to install SIGALRM handler\n");
        exit(1);
    }
```

```
}

void reset_alarm(){
        flag = 1;
        conta = 1;
        alarm(0);
}

char buildBCC2(char * data, int data_size) {
        char xor = data[0] ^ data[1];
        for (int i = 2; i < data_size; i++) {
                xor = xor ^ data[i];
        }
        return xor;
}

int verifyBCC(char * inform, int infMsgSize, char * data, int dataSize){
        char bcc = buildBCC2(data,dataSize);

        if(inform[infMsgSize-2] == bcc) return 0;
        else return -1;
}

void errorsBCC2(char* buffer, int buffer_size){
        int prob = rand() % 100;

        if (prob < BCC2_ERR_PROB){
                int changed_byte_index = rand() % (buffer_size -
                    SUPERVISION_TRAMA_SIZE) + DATA_INF_BYTE;
                char randomletter = 'A' + (rand() % 26);
                buffer[changed_byte_index] = randomletter;
                printf("Generated BCC2 errors!\n");
        }
}

void errorsBCC1(char* addr, char* ctrl){
        int prob = rand() % 100;

        if (prob < BCC1_ERR_PROB){
                int changed_byte = rand() % 2;
                char randomletter = 'A' + (rand() % 26);
        if(changed_byte) *ctrl = randomletter;
        else *addr = randomletter;
        printf("Generated BCC1 errors!\n");
        }
}

int parseREJ(char* buffer) {
        unsigned char arq = (unsigned char) buffer[CTRL_POS];
        if (arq == REJ(0) || arq == REJ(1))
                return 1;
        return 0;
}

int parseRR(char* buffer) {
        unsigned char arq = (unsigned char) buffer[CTRL_POS];
        if (arq == RR(0) || arq == RR(1))
                return 1;
        return 0;
}

int getSequenceNumber(char* buffer) {
        return (int) (buffer[CTRL_POS] >> 6);
}
```

11.10 message.h

```
/* SET & DISC are commands, the rest are answers */
#define SET 0x03
#define DISC 0x0b
#define UA 0x07
#define RR(s) ((((s + 1) % 2) << 7) | 0x03)
#define REJ(s) ((((s + 1) % 2) << 7) | 0x01)
```

```c
// Trama array positions
#define FLAGI_POSTION 0
#define ADRESS_POSITION 1
#define CC_POSITION 2 //Control camp position
#define PC_POSITION 3 //Protection camp position
#define FLAGF_POSTION 4

/* Trama delimiting flag */
#define FLAG 0x7e

/* A field for commands sent by receptor(REC) and emissor(EM) */
#define AREC 0x03 //Commands sent by emissor and responses by receptor
#define AEM 0x01  //Commands sent by receptor and responses by emissor

/* Escape byte*/
#define ESC 0x7d

/* Stuffing */
#define STUFF 0x20

//Size of the information message
#define INFO_SIZE_MSG(data_size)    ((data_size) + 6)

//Initial position of data blocks in information frame
#define DATA_INF_BYTE 4

//Supervision frame size
#define SUPERVISION_TRAMA_SIZE 5

//Info Message
#define CC_INFO_MSG(s)           (((s) % 2) << 6)

//Control fiel frame position
#define CTRL_POS 2

//Number of messages without answer
#define ERR_LIMIT 5

//BCC error generation probability
#define BCC1_ERR_PROB 0
#define BCC2_ERR_PROB 0
typedef struct {
    int stuffed_data;      /** Number of data bytes that were processed and
        stuffed */
    int stuffed_data_size;     /** Number of bytes that were occupied in the
        stuffing buffer */
} data_stuffing_t;

/**
 * @brief Alarm siganl handler.
 *
 * @param signo
 */
void atende(int signo);

/**
 * @brief Intalls alarm signal.
 *
 */
void install_alarm();

/**
 * @brief Resets alarm signal and associated variables.
 *
 */
void reset_alarm();

/**
 * @brief Reads message from serial port, byte per byte, updating associated
    state machine when a byte is received.
 *
 * @param fd Serial port file decriptor.
```

```
 * @param size Size of the message read.
 * @param i_message 1 if a information frame is being received, 0 if it is a
     supervision message.
 * @param emissor 1 if message is being read by emissor, 0 otherwise.
 * @return char* Message read.
 */
char* readMessage(int fd, int* size, int i_message, int emissor);

/**
 * @brief Builds and writes supervision frame message.
 *
 * @param fd Serial port file descriptor.
 * @param cc_value Control field of supervision frame.
 * @return int Number of bytes written (return value of write function).
 */
int write_supervision_message(int fd, char cc_value);

/**
 * @brief Writes supervision frame message and waits for receptor, ending if a
     valid supervision frame is received or if limit of resending tries is
     reached.
 *
 * @param fd Serial port file descriptor.
 * @param cc_value Control field of supervision frame.
 * @param cc_compare Value expected from control field of supervision frame
     message received from receptor.
 * @return int 0 in case of success, -1 otherwise.
 */
int write_supervision_message_retry(int fd, char cc_value, char cc_compare);

/**
 * @brief Builds and writes information frame message.
 *
 * @param fd Serial port file descriptor.
 * @param data Data to be inserted into information frame.
 * @param data_size Size, in bytes, of field data.
 * @param cc_value Control field of information frame.
 * @return int
 */
int write_info_message(int fd, char * data, int data_size, int cc_value);

/**
 * @brief Writes information frame message and waits for receptor, ending if a
     valid supervision frame is received or if limit of resending tries is
     reached.
 *
 * @param fd Serial port file descriptor.
 * @param buffer Data to be inserted into information frame.
 * @param dataSize Size, in bytes, of field data.
 * @return int Number of bytes written (return value of write function).
 */
int write_inform_message_retry(int fd, char * buffer, int dataSize);

/**
 * @brief Calculates bcc2 of data.
 *
 * @param data
 * @param data_size Size of data, in bytes.
 * @return char Bcc2 value.
 */
char buildBCC2(char * data, int data_size);

/**
 * @brief Verifies if there are bcc2 errors in the information frame received
     as parameter.
 *
 * @param inform Information frame received and unstuffed.
 * @param infMsgSize Information frame size.
 * @param data Data fields in information frame received to calculate bcc2 from
     .
 * @param dataSize Data size.
 * @return int 0 if bcc2 is successfully verified, -1 otherwise.
 */
```

```c
int verifyBCC(char * inform, int infMsgSize, char * data, int dataSize);

/**
 * @brief Verifies if supervision frame received as parameter is a REJ message.
 *
 * @param buffer Supervision frame.
 * @return int 1 if buffer is a REJ message, 0 otherwise.
 */
int parseREJ(char* buffer);

/**
 * @brief Verifies if supervision frame received as parameter is a RR message.
 *
 * @param buffer Supervision frame.
 * @return int 1 if buffer is a RR message, 0 otherwise.
 */
int parseRR(char* buffer);

/**
 * @brief Get the Sequence Number from information frame received.
 *
 * @param buffer Information received.
 * @return int Sequence number (0 or 1).
 */
int getSequenceNumber(char* buffer);

/**
 * @brief Set the Block Size value
 *
 * @param value Size of data block.
 */
void setBlockSize(int value);

/**
 * @brief Generate errors in BCC2
 *
 * @param buffer
 * @param buffer_size
 */
void errorsBCC2(char* buffer, int buffer_size);

/**
 * @brief Generate errors in BCC1
 *
 * @param addr Frame address field to possibly be changed.
 * @param ctrl Frame control field to possibly be changed.
 */
void errorsBCC1(char* addr, char* ctrl);
```

11.11 protocol.c

```c
#include "protocol.h"
#include "connection.h"
#include "message.h"
#include "state_machine.h"
#include "data_stuffing.h"
#include <time.h>
#include <unistd.h>

conn_type connection;
static int nr = 1;

int llopen(char* port, conn_type connection_type) {
        link_layer layer;
        strcpy(layer.port,port);
        layer.baud_rate = BAUDRATE;
        layer.num_transmissions = NUM_TRANSMISSIONS;
        layer.timeout = LAYER_TIMEOUT;
        connection = connection_type;

        int fd = open_connection(layer);
        if(connection_type == EMISSOR){
                if(write_supervision_message_retry(fd,SET,UA) == -1){
```

```
                        printf("Error establishing connection\n");
                        return -1;
                }
        }else if (connection_type == RECEPTOR){
                int trama_size = 0;
                char* trama = readMessage(fd, &trama_size, 0, 0);

                if (trama == NULL || trama == 0){
                        printf("LLOPEN: error reading SET message\n");
                        return -1;
                }

                if (trama[CTRL_POS] != SET) {
                        printf("LLOPEN: expected SET message\n");
                        return -1;
                }

                if(write_supervision_message(fd,UA) == -1){
                        printf("Error writing UA\n");
                        return -1;
                }
        }
        return fd;
}

int llclose(int fd) {
        if(connection == EMISSOR){
                if(write_supervision_message_retry(fd,DISC,DISC) == -1){
                        printf("Error establishing connection\n");
                }
                if(write_supervision_message(fd,UA) == -1){
                        printf("Error writing UA\n");
                }
        }
        else if (connection == RECEPTOR) {
                int buffer_size = 0;
                char* temp = readMessage(fd, &buffer_size, 0, 0);

                if (temp[CTRL_POS] == DISC) {
                        if (write_supervision_message(fd, DISC) == -1){
                                printf("LLCLOSE: error writing DISC message
                                    back\n");
                                return -2;
                        }
                }else{
                        printf("Error receiving DISC message\n");
                }

                free(temp);
                temp = readMessage(fd, &buffer_size, 0, 0);

                if (temp == NULL || buffer_size == 0){
                        printf("LLCLOSE: error reading UA message after sending
                             DISC\n");
                        return -1;
                }

                if (temp[CTRL_POS] != UA) {
                        printf("LLCLOSE: wrong message after sending DISC\n");
                        return -1;
                }
        }

    return close_connection(fd);
}

int llwrite(int fd, char* buffer, int length) {
        return write_inform_message_retry(fd, buffer, length);
}

int llread(int fd, char* buffer) {
        int temp_size = 0;
```

```
        //tprop variation simulation
        //usleep(10*1000);

        char* temp = readMessage(fd, &temp_size, 1, 0);
        if(temp == NULL || temp_size == 0){
                printf("LLREAD: exit after %d failed attempts to read message\n
                    ", ERR_LIMIT);
                return -1;
        }

        if (temp_size <= 5){
                printf("Connection changed ways\n");
                return -2;
        }

        errorsBCC2(temp, temp_size);

        int seq_number = getSequenceNumber(temp);
        if (getSequenceNumber(temp) == nr % 2) {
                printf("Received repeated trama\n");
                if (write_supervision_message(fd, RR(seq_number)) == -1){
                        printf("LLREAD: error writing RR message back\n");
                        return -1;
                }
                return -2;
        }

        data_stuff unstuffedData = unstuffData(temp, temp_size);

        int buffer_size = unstuffedData.data_size - 6;
        memcpy(buffer, unstuffedData.data + DATA_INF_BYTE, buffer_size);

        if(verifyBCC(unstuffedData.data,unstuffedData.data_size,buffer,
            buffer_size) < 0){
                printf("Error verifying BCC2\n");
                int rd = write_supervision_message(fd, REJ(seq_number));
                if (rd == -1){
                        printf("LLREAD: error writing REJ message back\n");
                        return -1;
                }
                return -3;
        }
        if (write_supervision_message(fd, RR(seq_number)) == -1){
                printf("LLREAD: error writing RR message back\n");
                return -1;
        }

        nr++;
        return buffer_size;
}
```

11.12 protocol.h

```
#define FALSE 0
#define TRUE 1

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <strings.h>
#include <signal.h>

#include "utils.h"

//Message configurations
#define RESEND_DELAY 3
#define WRITE_NUM_TRIES 3
```

```c
/**
 * @brief Opens serial port connection. If connecition_type is of type EMISSOR,
      write_supervision_message_retry is called to send SET message and wait
     for UA message. If connection_type is of type RECEPTOR it waits for a SET
     message and sends a UA message after it.
 *
 * @param arg Port identifier.
 * @param connection_type EMISSOR | RECEPTOR
 * @return int Serial port file descriptor.
 */
int llopen(char* arg, conn_type connection_type);

/**
 * @brief Closes connection. If
 *
 * @param fd Serial port file descriptor.
 * @return int Return value of function close_connection(fd)
 */
int llclose(int fd);

/**
 * @brief Function receives data from file and its size, builds information
     frame, sends it and waits for acknowledgement message with alarm signal
     implemented to enable retransmissions.
 *
 * @param fd Serial port file descriptor.
 * @param data Data to be transmitted.
 * @param data_size Size of data, in bytes.
 * @return int Number of bytes written, negative value in case of error.
 */
int llwrite(int fd, char* data, int data_size);

/**
 * @brief Receives information frame, checks if it consists of a repeated
     message and if bcc2 is correct and sends correspondent acknowledgement
     message back to emitter.
 *
 * @param fd Serial port file descriptor.
 * @param buffer Data received.
 * @return int Number of bytes read, negative value in case of error.
 */
int llread(int fd, char* buffer);
```

11.13 state_machine.c

```c
#include "message.h"
#include "state_machine.h"
#include "protocol.h"

state_machine current_state = START;

state_machine getStateMachine(){
        return current_state;
}

char addr, ctrl;
int inf_bytes = 0;

void handleState(char msg, int i_message, int* error, int emissor){
    state_machine state_machine = getStateMachine();

    switch (state_machine){
        case START:
            handleStartState(msg, error);
            break;
        case FLAG_RCV:
            handleFlagReceived(msg);
            addr = msg;
            break;
        case A_RCV:
            handleAddrReceived(msg);
            ctrl = msg;
            break;
```

```
        case C_RCV:
            handleCtrlState(msg,addr,ctrl,emissor);
            break;
        case BCC1_OK:
            handleBcc1State(msg, i_message);
            break;
        case DATA_INF:
            handleDataState(msg);
            break;
        case STOP:
            handleStopState(msg);
            break;

        default:
            break;
    }
}

void handleStartState(char msg, int* error){
    if (msg == FLAG && *error){
        *error = 0;
    }else if (msg == FLAG){
        update_state(FLAG_RCV);
    }
}
void handleFlagReceived(char msg) {
    switch (msg) {
        case FLAG:
            break;
        case AREC: case AEM:
            update_state(A_RCV);
            break;
        default:
            update_state(START);
            break;
    }
}

void handleAddrReceived(unsigned char msg) {
    switch (msg) {
        case FLAG:
            update_state(FLAG_RCV);
            break;
        case DISC: case SET: case UA:
            update_state(C_RCV);
                        break;
        case CC_INFO_MSG(0): case CC_INFO_MSG(1):
            update_state(C_RCV);
            break;
        case RR(0):
            update_state(C_RCV);
            break;
        case REJ(0): case REJ(1):
            update_state(C_RCV);
            break;
        default:
            update_state(START);
            break;
    }
}

void handleCtrlState(char msg, char addr, char ctrl, int emissor){
    switch (msg){
                case FLAG:
            update_state(FLAG_RCV);
            break;
        default:
            if (!emissor && (ctrl == CC_INFO_MSG(0) || ctrl == CC_INFO_MSG(1)))
                errorsBCC1(&addr, &ctrl);
            if(msg == (addr ^ ctrl))
                update_state(BCC1_OK);
            else{
                update_state(START);
```

```
                if (!emissor) printf("BCC1 error\n");
            }
            break;
    }

}

void handleBcc1State(char msg, int i_message) {
    switch (msg) {
        case FLAG:
            update_state(STOP);
            break;
        default:
            if (i_message)
                update_state(DATA_INF);
            else
                update_state(START);
            break;
    }
}

void handleDataState(char msg){
    switch (msg){
        case FLAG:
            update_state(STOP);
            break;
        default:
            break;
    }
}


void handleStopState(char msg) {
    switch (msg) {
        case FLAG:
            update_state(FLAG_RCV);
            break;
        default:
            update_state(START);
            break;
    }
}

void update_state(state_machine state){
    current_state = state;
}
```

11.14 state_machine.h

```
typedef enum {
  START = 0,
  FLAG_RCV = 1,
  A_RCV = 2,
  C_RCV = 3,
  BCC1_OK = 4,
  DATA_INF = 5,
  STOP = 6
} state_machine;

/**
 * @brief Recognizes state machine current state and updates it according to
     new char msg received.
 *
 */
void handleState(char msg, int i_message, int* error, int emissor);

/**
 * @brief Handles start state, advances to state FLAG_RCV if a char of value
     FLAG is received and no errors were verified. If there were previous
     errors and a byte of value FLAG was received errors value is updated to 0.
 *
 * @param msg Byte received.
 * @param error Error indicator variable.
```

```
 */
void handleStartState(char msg, int* error);

/**
 * @brief Hadles FLAG_RCV state, advances if a valid adress field value is
     received.
 *
 * @param msg Byte received.
 */
void handleFlagReceived(char msg);

/**
 * @brief Handles A_RCV state, advances if a valid control field is received.
 *
 * @param msg Byte received.
 */
void handleAddrReceived(unsigned char msg);

/**
 * @brief Handles C_RCV state, advances if BCC1 is verified.
 *
 * @param msg Byte received.
 * @param addr Address field to calculate BCC1.
 * @param ctrl Control field to calculate BCC1.
 * @param emissor 1 if state machine is reading messages that are being
     received by the emitter, for error generation reasons.
 */
void handleCtrlState(char msg,char addr,char ctrl,int emissor);

/**
 * @brief Handles BCC1_OK state, if i_message is verified advances to DATA_INF
     state, otherwise if a FLAG is received updates state machine to STOP state
     .
 *
 * @param msg Byte received.
 * @param i_message 1 if byte received belongs to information frame, 0
     otherwise.
 */
void handleBcc1State(char msg, int i_message);

/**
 * @brief Handles DATA_INF state, only advancing if a FLAG is received.
 *
 * @param msg Byte received.
 */
void handleDataState(char msg);

/**
 * @brief Handles STOP state.
 *
 * @param msg
 */
void handleStopState(char msg);

/**
 * @brief Sets new state machine value.
 *
 * @param state New state machine value.
 */
void update_state(state_machine state);

/**
 * @brief Get the current state of the state machine.
 *
 * @return state_machine
 */
state_machine getStateMachine();
```

11.15 utils.c

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

```c
#include <unistd.h>
#include "utils.h"


char* concat(const char *s1, const char *s2){
    char *result = malloc(strlen(s1) + strlen(s2) + 1); // +1 for the null-
        terminator
    strcpy(result, s1);
    strcat(result, s2);
    return result;
}

bool check_arg(int argc, char* argv[]){
  return (argc >= 3) &&(
         ((argc == 5) && (strcmp("emissor", argv[2]) == 0)) ||
         ((argc == 3) && (strcmp("receptor", argv[2]) == 0)));
}

arguments parse_arguments(int argc, char *argv[]) {
    if (!check_arg(argc, argv)){
        printf("Usage Receptor: main <serial port number> receptor\n");
        printf("Usage Emissor: main <serial port number> emitter (filename) (
            data_block_size)\n");
        exit(1);
    }

    char * port = concat(PORT_ARG,argv[1]);
    arguments args;
    if (argc == 3) args.role = RECEPTOR;
    else{
        args.role = EMISSOR;
        args.filename = argv[3];
        args.data_block_size = atoi(argv[4]);
    }
    args.port_num = port;

    return args;
}
```

11.16 utils.h

```c
#include <stdbool.h>

typedef enum{
    EMISSOR,
    RECEPTOR
} conn_type;

typedef struct {
  conn_type role;
  char *filename;
  char * port_num;
  int data_block_size;
} arguments;

#define PORT_ARG "/dev/ttyS"

/**
 * @brief Parses command line arguments int appropiate structr arguments.
 *
 * @param argc
 * @param argv
 * @return arguments
 */
arguments parse_arguments(int argc, char *argv[]);

/**
 * @brief Checks if command line arguments were correctly introduced, returning
      true or false accordingly.
 *
 * @param argc
 * @param argv
 * @return true
```

```
 * @return false
 */
bool check_arg(int argc, char* argv[]);

/**
 * @brief Concatenates two buffers.
 *
 * @param s1
 * @param s2
 * @return char*
 */
char* concat(const char *s1, const char *s2);
```

12 Anexo II - Tabelas

12.1 Variação do Tamanho de Trama I

Os seguintes resultados foram realizados nas seguintes condições:

- Número total de bytes: 10968
- Baudrate: 38400

Os seguintes resultados foram realizados nas seguintes condições:

Tamanho da Trama I	Tempo	R(bits/tempo)	S =R/C	Média S
50	4,2132	20825,98	0,542	0,542
	4,2159	20812,64	0,542	
150	3,3759	25991,29	0,677	0,677
	3,3767	25985,13	0,677	
250	3,1661	27713,59	0,722	0,722
	3,1661	27713,59	0,722	
350	3,1014	28291,74	0,737	0,737
	3,1013	28292,65	0,737	
450	3,0554	28717,68	0,748	0,748
	3,055	28721,44	0,748	
550	3,0324	28935,50	0,754	0,754
	3,0324	28935,50	0,754	
650	3,0072	29177,97	0,760	0,760
	3,0073	29177,00	0,760	
750	2,9948	29298,78	0,763	0,763
	2,9947	29299,76	0,763	
850	2,9834	29410,74	0,766	0,764
	2,9947	29299,76	0,763	
950	2,976	29483,87	0,768	0,768
	2,9759	29484,86	0,768	
1050	2,9698	29545,42	0,769	0,769
	2,9696	29547,41	0,769	

Figura 1: Resultados obtidos variando o Tamanho da trama I

12.2 Variação do Baudrate

- Número total de bytes: 10968
- Tamanho da trama I: 255

Tamanho da Trama I	Tempo	R(bits/tempo)	S =R/C	Média S
50	4,2132	20825,98	0,542	0,542
	4,2159	20812,64	0,542	
150	3,3759	25991,29	0,677	0,677
	3,3767	25985,13	0,677	
250	3,1661	27713,59	0,722	0,722
	3,1661	27713,59	0,722	
350	3,1014	28291,74	0,737	0,737
	3,1013	28292,65	0,737	
450	3,0554	28717,68	0,748	0,748
	3,055	28721,44	0,748	
550	3,0324	28935,50	0,754	0,754
	3,0324	28935,50	0,754	
650	3,0072	29177,97	0,760	0,760
	3,0073	29177,00	0,760	
750	2,9948	29298,78	0,763	0,763
	2,9947	29299,76	0,763	
850	2,9834	29410,74	0,766	0,764
	2,9947	29299,76	0,763	
950	2,976	29483,87	0,768	0,768
	2,9759	29484,86	0,768	
1050	2,9698	29545,42	0,769	0,769
	2,9696	29547,41	0,769	
2050	2,9179	30070,94	0,783	0,783
	2,9178	30071,97	0,783	
3050	2,9104	30148,43	0,785	0,785
	2,9104	30148,43	0,785	
4050	2,9059	30195,12	0,786	0,786
	2,9058	30196,16	0,786	
5050	2,9058	30196,16	0,786	0,786
	2,9058	30196,16	0,786	

Figura 2: Resultados obtidos variando o Baudrate

12.3 Variação do FER

- Número total de bytes: 10968
- Tamanho da trama I: 255
- Baudrate: 38400

Probabilidade de Erro	Tempo	R(bits/tempo)	S=R/C	Média S
0+0	3,0696	28584,8319	0,744397	0,744405
	3,0695	28585,76315	0,744421	
	3,0696	28584,8319	0,744397	
2+2	6,209	14131,74424	0,368014	0,372234
	6,0696	14456,30684	0,376466	
	6,1388	14293,34723	0,372223	
2+4	9,2099	9527,139274	0,248103	0,244462
	9,4172	9317,41919	0,242641	
	9,4171	9317,518132	0,242644	
6+6	18,2087	4818,795411	0,125489	0,153601
	12,3471	7106,446048	0,185064	
	15,208	5769,59495	0,15025	
8+8	21,3821	4103,619383	0,106865	0,129299
	24,6256	3563,121305	0,09279	
	12,1387	7228,451152	0,188241	
10+10	30,4292	2883,546068	0,075092	0,098234
	18,2086	4818,821875	0,12549	
	24,2775	3614,210689	0,09412	

Figura 3: Resultados obtidos variando o FER

12.4 Variação do Tempo de Propagação

- Número total de bytes: 10968
- Tamanho da trama I: 255
- Baudrate: 38400

T de Propagação	Tempo	R(bits/tempo)	S =R/C	Média S
10	3,07	28584,83	0,74	0,74
	3,07	28583,90	0,74	
50	3,15	27878,25	0,73	0,73
	3,15	27875,59	0,73	
100	4,62	18999,20	0,49	0,49
	4,62	18997,96	0,49	
200	9,22	9518,46	0,25	0,25
	9,22	9518,56	0,25	
500	23,02	3811,91	0,10	0,10
	23,02	3811,89	0,10	
750	34,52	2541,95	0,07	0,07
	34,52	2541,96	0,07	
1000	46,02	1906,69	0,05	0,05
	46,02	1906,72	0,05	

Figura 4: Resultados obtidos variando o Tempo de Propagação