

RCOM 2019/2020

Trab. Lab 1



**Trabalho Realizado por:** Luís Pedro Viana Ramos up201706253

Martim Pinto da Silva up201705205

Rúben Filipe Seabra de Almeida up201704618

**Unidade Curricular:** RCOM

**Ano Letivo:** 2019/2020

**Professor Responsável:** Prof.Rui Campos

**Regente:** Prof.Manuel Alberto Pereira Ricardo

**Sumário**

O primeiro trabalho laboratorial de RCOM, insere-se no contexto da introdução às comunicações em rede do MIEIC. Neste relatório, encontrar-se-á, a concretização do primeiro protocolo de comunicação de dados arquitetado aplicado a uma aplicação de transferência de ficheiros, recorrendo, para isso, à programação da histórica porta RS-232 como suporte de hardware.

Este trabalho permitiu-nos perceber melhor o mecanismo *Stop&Wait*, dos requisitos que estes apresentam, bem como as suas limitações.

**Introdução**

O primeiro trabalho laboratorial têm como principais objetivos:

* Desenho de uma aplicação de partilha de ficheiros através da rede-Pag 2
* Introdução à programação de redes em camadas independentes baseadas no modelo TCP/IP-Pag 2
* Elaboração de um protocolo de dados entre duas máquinas-Pag 3
* Necessidade de mecanismos de retransmissão. Pag 4 e 5.
* Programação da Porta RS-232 em ambiente UNIX-Pag 4 a 6
* Contacto com a necessidade de Framing em comunicação assíncrona:
  + Técnicas de Byte-Stuffing- Pag 5
* Estabelecer mecanismos de deteção e resposta a erros de comunicação:
  + Parity check methods(Block Check Character)- Pag 5
  + Mecanismos Stop and Wait. Pag 9
* Conceitos Teóricos:
  + Definição probabilística do erro na comunicação em rede- Pag 7 e 8
  + Eficiência do protocolo- Pag 7 a 9

**Arquitetura**

No trabalho apresentado, segue-se uma filosofia de ocultar os detalhes de baixo nível e mecanismos de controlo do utilizador, por essa razão, o main encontra-se ao nível mais alto de abstração que este trabalho laboratorial contempla, a application layer (application.c).

Application é, assim, o ponto de partido da execução. Este módulo para além de servir-se dos serviços da camada imediatamente mais abaixo, no modelo implementado o data link layer, serve-se das funcionalidades da biblioteca files que agrupa as funcionalidades necessárias à manipulação de ficheiros. Tais como:

* Abertura do ficheiro
* Leitura/Escrita,
* Determinação do tamanho do ficheiro.

De referir a implementação de um arquitetura onde existe **independência de camadas**. Ou seja, que a camada “mais alto nível” a única forma de contacto que possui com níveis mais baixos de implementação é através da API definida pela camada **imediatamente** **subjacente.**

**Estrutura do Código**

A implementação do protocolo de ligação e a aplicação de teste estão fisicamente separadas em ficheiros diferentes, nos ficheiros dedicados ao protocolo encontram-se a definição das técnicas de verificação de erro por paridade, framing por byte stuffing, framing Stop and Wait com recurso a numeração das tramas.

Esta camada constrói ela própria uma API de quatro funções que são, de acordo com a filosofia de independência de camadas, as únicas formas da camada sobrejacente têm à sua disposição para usufruir dos serviços da camada, a saber: estabelecer/desconectar a ligação lógica, usufruir de um serviço de envio/receção de mensagens, resiliente a erros de transmissão. A API definida pela data link layer é a seguinte:

* llopen llclose - Estabelecer e Desconectar(respetivamente) a ligação entre duas máquinas.
* llread, llwrite - Ler e Escrever (respetivamente) informação para outra máquina.

Por sua vez, application, baseando-se nas garantias que a API data lynk layer lhe concede, estabelece, também ela uma API própria orientada ao propósito da aplicação de teste do protocolo, a transferência de ficheiros simples. Contabiliza-se as seguintes funções:

* sendDataBlock receiveDataBlock - Ler e receber bloco de informação de um ficheiro
* sendControlBlock receiveDataBlock - Enviar e receber um bloco de controlo.

A comunicação protocolada, faz-se através de estruturas de dados, que para além dos dados, incorporam outras informações que permitem, no caso das frames que circulam ao nível do data link layer, redundância ligada à paridade, permitindo realizar testes de erro, e flags que delimitam as mensagens com de acordo com a filosofia da técnica de Stop and Wait. Ao nível da Aplicação, os pacotes de controlo são do tipo Type, Length, Value (TLV), enquanto que os pacotes de dados possuem a sua estrutura de dados própria de acordo com as especificações(Figura.2).

**Frames:** (Consultar Figura 1). As frames de dados possuem:

* O byte flag que marca o início e o fim de transferência.
* Field D e fieldBCC2 são apontadores para arrays de tamanhos que podem variar
* bcc2StufSize e dataStufSize guardam o tamanho dos arrays supra mencionados

**Pacotes:** (Consultar Figura 2 e 3). Os pacotes dividem-se em pacotes de controlo-TLV, e pacotes de dados :

* Pacotes Controlo e TLV: Enviam informações de controlo.

ControlField:Flag delimitadora. type: Tipo de mensagem de controlo que queremos enviar.

value:Informação. Array de tamanho variável. length:tamanho do array enviado.

filesize/fileName:Pacote TLV onde é transferido o tamanho e nome do pacote(respetivamente).

* Pacotes de dados: Enviam pacotes do ficheiro.

FieldN- Número de sequência do pacote de dados.

fieldL1 e fieldL2: Transmitem o tamanho do bloco de dados.

Field P: Bloco de dados

**Casos de uso principais**

Para iniciar a aplicação, o utilizador deve seguir a sintaxe seguinte:

* ./reader “número da porta”.
* ./writer “número da porta” “nome ficheiro”

### Transferência de ficheiro através da porta de série

A transferência do ficheiro é feita através da chamada a várias funções e segue o percurso abaixo descrito:

* Chamada à função **sendFile**(emissor) / **receiveFile**(recetor)**.**
* Dentro dessa função é chamada a função **llopen** onde se faz a configuração da ligação entre os 2 computadores e estabelecimento da mesma.
* É feito depois a chamada à função **sendControlBlock / receiveControlBlock** sendo enviado/recebido o pacote inicial de controlo do protocolo da aplicação.
* O ficheiro é transferido recorrendo ao módulo **sendDataBlock(**emissor) e **receiveDataBlock**(recetor)**,**
* Por último, o ficheiro é fechado com uma chamada a sendControlBlock, envia-se o pacote final de controlo do protocolo da aplicação e fecha-se a ligação com a porta de séria com a chamada à função **llclose.** O descritor do ficheiro é fechado.

**Protocolo da ligação lógica**

O protocolo que gere a informação que circula na camada data\_lynk é composto por uma API que engloba as seguintes funções:

**int llopen(int port\_number, int flag):**

Função com a tarefa de efetuar a ligação lógica entre emissor e recetor.

Do lado do emissor é enviado uma mensagem SET que é recebida, sendo procedido de uma resposta UA.

A função é protegida com um alarme de timeout no envio da mensagem SET, ou seja se o recetor não receber a mensagem dentro do tempo estabelecido (no caso 3 segundos), a mensagem SET é reenviada até um máximo de 3 vezes. Caso as três retransmissões forem atingidas, o programa desencadeará os mecanismos de saída, no caso, a função llopen, descrita abaixo.

**int llwrite(int fd, unsigned char \*buffer, int length)**

Função principal do programa emissor, responsável pelo envio de pacotes de dados e pelo mecanismo de stuffing destes pacotes. É aqui que é construída a mensagem a ser enviada para o recetor.

**int llread(int fd, unsigned char \*buffer)**

Função responsável pela leitura dos pacotes e pela técnica de “destuffing” dos mesmos. É feita uma máquina de estados (*Figura 4*), que verifica em que estado está a máquina e qual será o próximo passo na leitura dos pacotes.

Inicialmente é verificado o cabeçalho do pacote recebido. Só a seguir (e dependendo de uma verificação positiva) é feita o processamento dos dados.

Por fim, é feito o “destuffing” e a confirmação de que o BCC2 depois dessa tarefa está correto. É também enviado uma mensagem RR(i) sendo i o número do próximo pacote (0 ou 1).

**int llclose(int fd, int flag)**

Função com a tarefa de terminar a ligação do canal lógico entre emissor e recetor.

Do lado do emissor é enviado uma mensagem DISC que é recebida no lado do recetor e envia novamente uma mensagem DISC como resposta. Por último o emissor envia uma mensagem UA.

Se houver um problema do lado do recetor ou emissor para receber uma mensagem DISC, é feita um número limite de tentativas de retransmissão destas mensagens.

Se houver um problema do lado do recetor para receber uma mensagem UA o recetor faz a retransmissão de DISC para o emissor, que reenvia a mensagem UA.

### Uso da redundância para controlo do erro

No trabalho laboratorial 1, implementa-se de acordo com as especificações um valor de redundância comum a todas as tramas enviadas, que pretende proteger erros no cabeçalho, BCC1. Nas tramas de informação, para além de BCC1, é introduzido uma proteção para o bloco de dados. Ambos valores de BCC são calculados pelo ou-exclusivo dos bytes que pretendem proteger.

### Mecanismo de Framing: Byte Stuffing

Visto tratar-se de um protocolo para ser implementado num hardware de características assíncronas, é necessário delimitar as mensagens com flags de controle, no caso, recorre-se ao byte 0x7E. O uso deste caracter sinalizador, levanta a necessidade de resolver eventuais ambiguidades de flags que ocorram noutros locais distintos da função delimitadora do mesmo. Para tal, recorre-se a um mecanismo de byte stuffing, em que os dados que compõe a mensagem e o valor de BCC2 é submetido a um pré processamento antes de enviar.

No lado do recetor, a primeira operação a ser realizada é a operação inversa, o destuffying. Só depois de realizada o destuffying, é que os dados podem ser processados.

**Protocolo da aplicação**

int sendDataBlock(int fd, uint sequenceNumber, uchar \*buffer, uint length)

* É preenchida a estrutura AppDataStruct com os campos definidos no protocolo da aplicação. A estrutura AppDataStruct é composta por quatro campos do tipo “unsigned char” que contém os campos N, L1, L2 e P e um campo C que é representado através de um “enum” que pode ter os valores de Data, Start ou End.
* O campo N representa o número da sequência (módulo de 255). Os campos L1 e L2 representam o número de octetos do campo de dados que é igual a 256 \* L2 + L1. O campo P representa o campo de dados do pacote.

int receiveDataBlock(int fd, uint sequenceNumber, uchar \*buffer, uint length)

* É lido o pacote de dados através da chamada à função llread.
* De seguida, é verificado se o campo de controlo é Data. Se a verificação for positiva, é atualizado o número da sequência e os dados guardados num “buffer”.

int sendControlBlock(int fd, uint sequenceNumber, uchar \*buffer, uint length)

* É preenchida a estrutura AppControlStruct com os campos definidos no protocolo da aplicação. A estrutura AppControlStruct é composta por dois campos do tipo TLV que contém o tamanho do ficheiro e o nome do ficheiro e um campo C que é representado através de um “enum” que pode ter os valores de Data, Start ou End.

int receiveControlBlock(int fd, uint sequenceNumber, uchar \*buffer, uint length)

* É lido o pacote de controlo através da chamada à função llread.
* De seguida, é verificado se o campo de controlo é Start ou End. Se a verificação for positiva, é guardado o valor do nome do ficheiro e o tamanho do ficheiro.

A divisão do ficheiro em fragmentos é feita através da chamada à função fread, que tem um parâmetro “count” que indica o número de caracteres irá ler e esse número de caracteres será posteriormente escrito através da função sendDataBlock.

Através da chamada à função sendDataBlock que faz uma chamada a função llwrite, os fragmentos do ficheiros são encapsulados com o protocolo de dados.

**Validação**

A validação da aplicação foi realizada através de vários testes nos quais foram usados diferentes ficheiros, nomeadamente:

* **pinguim.gif** - Tamanho: 10,968 bytes;
* **Ficheiros de tamanhos médios** - Tamanho: desde 100,000 a 300,000 bytes;
* **Ficheiros de tamanhos grandes** - Tamanho: acima de 500,000 bytes.

Os testes realizados seguiram um conjunto de casos predefinidos:

* **Caso de uso principal com diferentes ficheiros**: foram transferidos de um computador para ficheiros com diferentes tamanhos;
* **Porta de série é fechada e reaberta**: à medida que os ficheiros estão a ser transferidos é desligada a porta de série durante um curto período de tempo, entre 1 a 5 segundos;
* **São introduzidos bits aleatórios na porta série**: à medida que os ficheiros estão a ser transferidos são, manualmente, inseridos bits aleatórios através de um fio ligado à porta série;
* **Junção de diferentes casos de erro**: a aplicação é submetida a diferentes tipos de erro à medida que está a decorrer a transferência sendo que estes não ocorrem em simultâneo pois sendo que a porta de série é desligada é impossível introduzir bits aleatórios.

**Eficiência do protocolo de ligação de dados**

O cálculo da eficiência do protocolo foi feito com base na alteração de vários parâmetros da aplicação definidos através de macros e também através da introdução de erros com probabilidades predefinidas.

A eficiência do protocolo foi calculada através de testes realizados em máquinas virtuais podendo portanto ser um pouco diferente do valor real.

### Eficiência do protocolo com diferentes tamanhos da trama I

A eficiência do protocolo aumenta com o aumento do tamanho da trama até um certo nível, isto deve-se ao facto de com tramas muito pequenas ser necessário o envio adicional do cabeçalho de trama definido no protocolo, sendo portanto bastante dispendioso em termos de tempo. Por outro lado, com tramas demasiado grandes estamos propensos a um maior número de erros apesar de durante os testes realizados nunca terem sido detetados erros levando a deduzir que BER é muito pequeno.

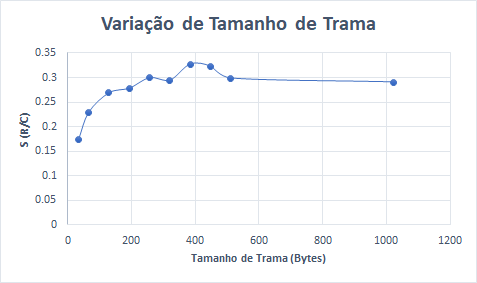


Figura 6: Gráfico do cálculo de eficiência com variação de tamanho de trama

### Eficiência do protocolo com diferentes FER

A eficiência do protocolo com diferentes FER pode ser um pouco subjetiva visto que de acordo com o tipo de erro, sendo em BCC1 ou em BCC2, pode ter consequências diferentes sendo que com o erro sendo em BCC2 é imediatamente enviada uma resposta de erro o que faz com que o emissor reenvie a mensagem e com o erro sendo em BCC1 o recetor fica simplesmente aguardando que o emissor reenvie a mensagem quando ocorrer um *timeout*.

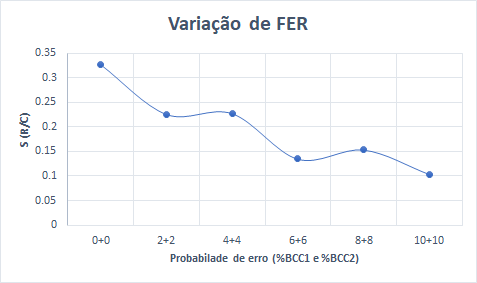
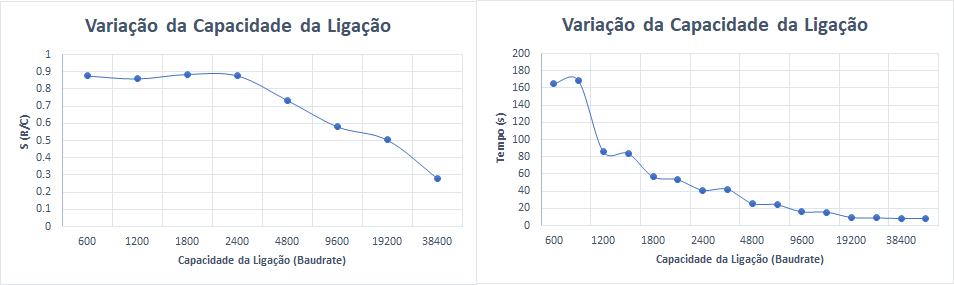


Figura 9: Gráfico do cálculo de eficiência com variação de probabilidade de erro de trama

### Eficiência do protocolo com diferentes capacidades da ligação

A eficiência do protocolo é maior quanto menor for a capacidade da ligação, porém o tempo de transferência é também maior, levando a que gráfico da eficiência seja um enganador quando não acompanhado do gráfico do tempo.

Figura 12: Gráficos do cálculo de eficiência e do tempo de envio de acordo com variação da capacidade de ligação

### Eficiência do protocolo com diferentes tempos de propagação

A eficiência neste caso é maior quanto menor for o tempo de propagação, isto deve-se ao facto de o recetor ter que esperar entre cada byte recebido, levando a um tempo de espera considerável caso o atraso seja considerável, podendo até impossibilitar a comunicação entre os dois devido a provocar um *timeout* devido ao tempo propagação.

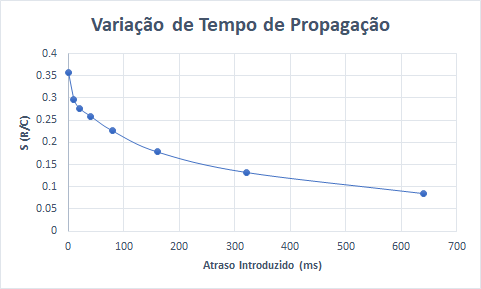


Figura 15: Gráfico do cálculo de eficiência com variação do tempo de propagação

Em relação ao protocolo Stop & Wait a lógica é bastante simples:

* Emissor:
  + Envia pacote com número de sequência que varia apenas entre 0 e 1;
  + Espera resposta do recetor com diferentes números de sequência de acordo com o que foi lido por este.
* Recetor:
  + Lê e verifica a integridade do pacote com número de sequência;
  + Envia resposta de acordo com a integridade do pacote, enviando RR com o próximo número de sequência ou REJ com o atual número de sequência.

Sendo que o nosso protocolo é baseado no protocolo Stop & Wait.

|  |  |  |
| --- | --- | --- |
|  | Resposta do Recetor | |
| Trama enviada pelo Emissor | **Sem erros** | **Com erros** |
| Ns = 0 | RR(Nr = 1) | REJ(Nr = 0) |
| Ns = 1 | RR(Nr = 0) | REJ(Nr = 1) |

**Conclusões**

Conseguimos alcançar o objetivo pretendido, que é produzir um protocolo de ligação de dados robusto, capaz de reagir em caso de erros e de interrupção súbita do canal de comunicação, tendo assim uma melhor perceção do mecanismo de comunicação *Stop&Wait*. Também nos permitiu perceber melhor as limitações deste mecanismo, nomeadamente, ao nível do desempenho do protocolo em casos de erro.

**Anexos:**

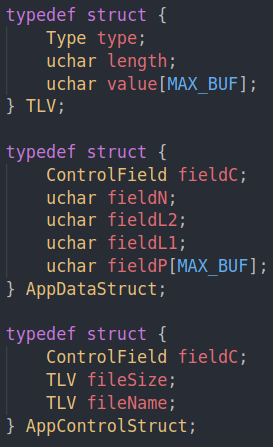
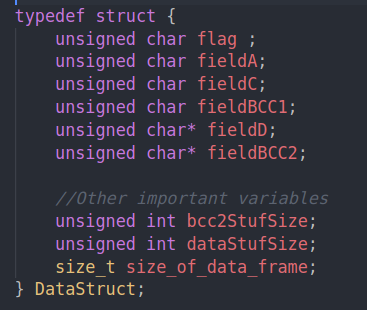


Figura 1:Frames Figura 2:Estruturas de dados da aplicação

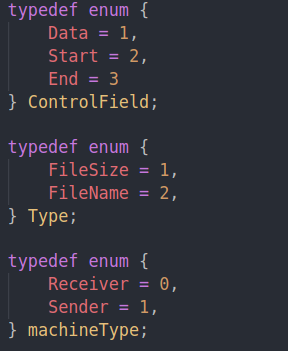


Figura 3:Blocos auxiliares das estruturas de dados

da aplicação

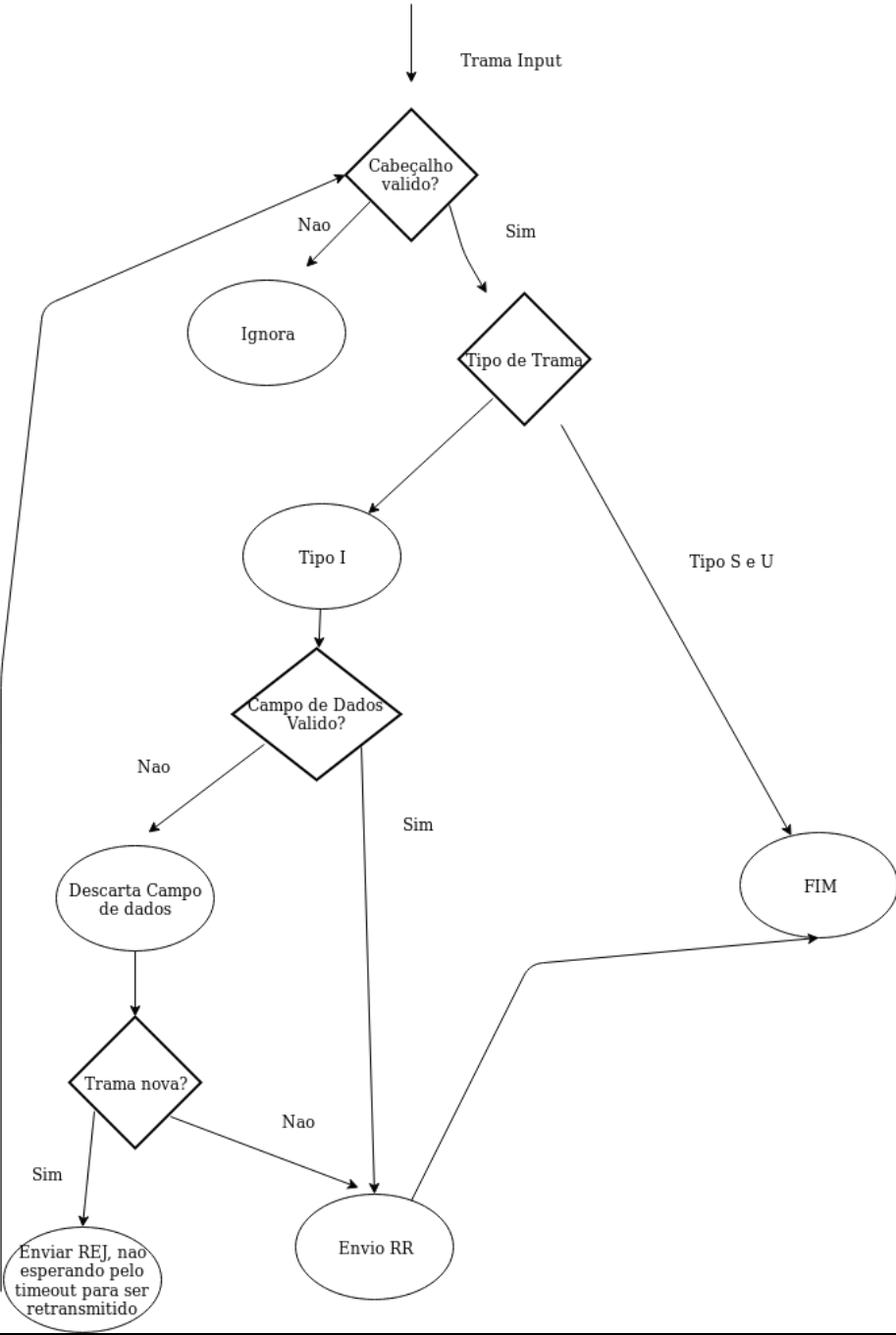
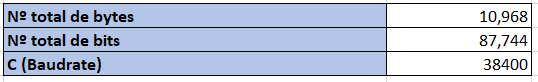
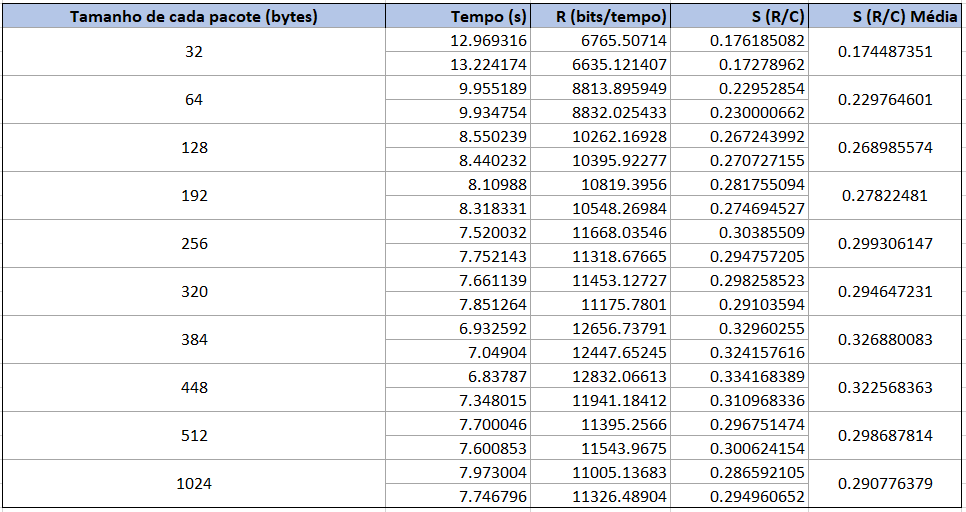


Figura 4: Máquina de estados



Figuras 5 e 6: Tabelas de valores do cálculo de eficiência com variação do tamanho de trama

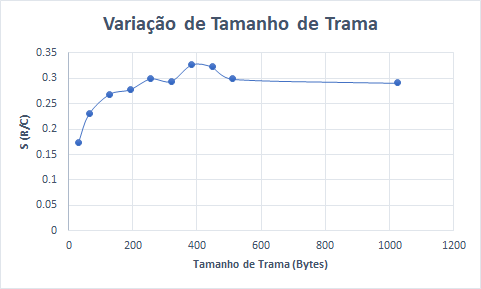
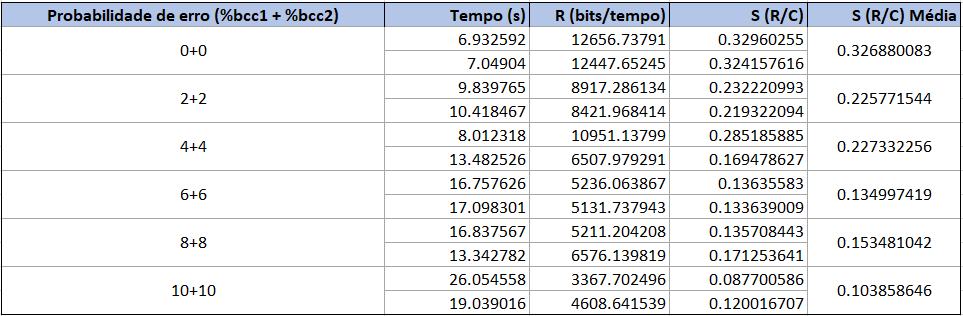


Figura 6: Gráfico do cálculo de eficiência com variação e tamanho de trama



Figuras 7 e 8: Tabelas de valores do cálculo de eficiência com variação de probabilidade de erro

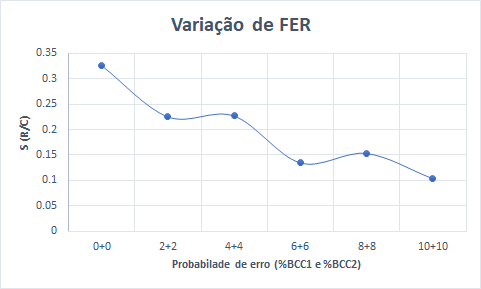
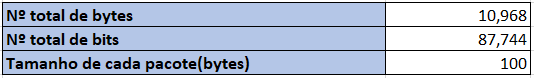
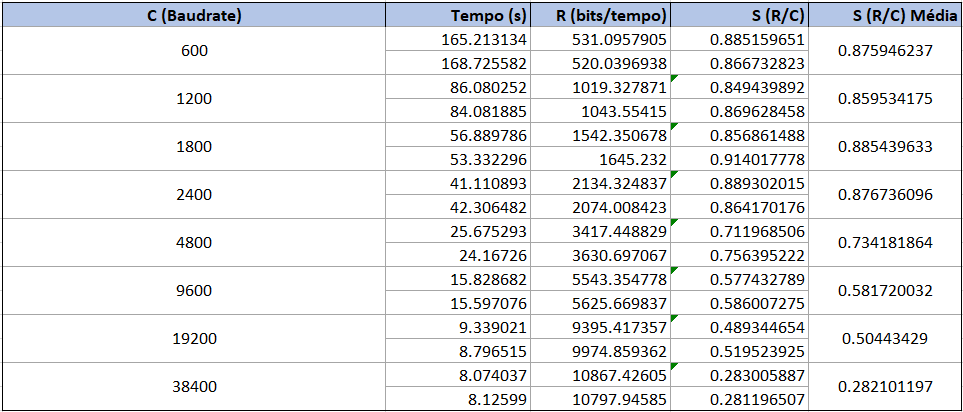


Figura 9: Gráfico do cálculo de eficiência com variação de tamanho de trama





Figuras 10 e 11: Tabelas de valores do cálculo de eficiência com variação de capacidade de ligação

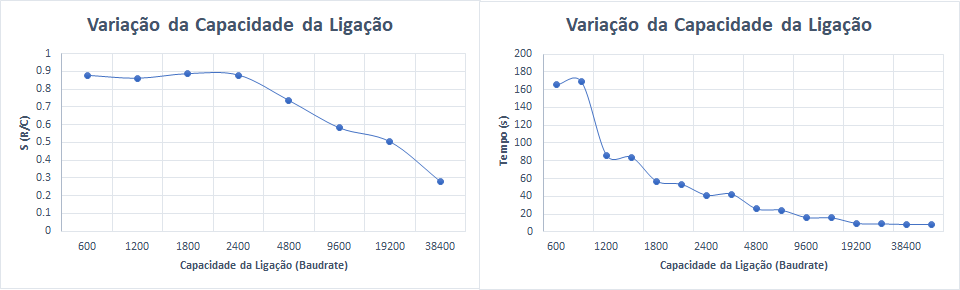
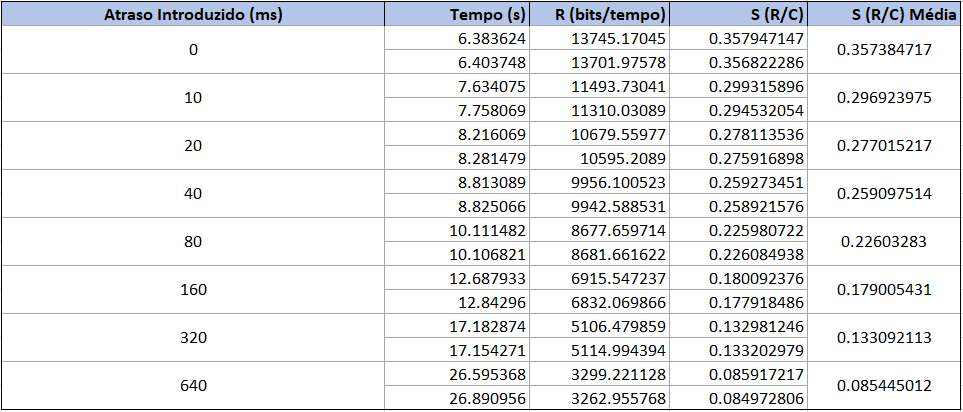
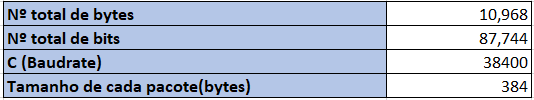


Figura 12: Gráficos do cálculo de eficiência e do tempo de envio de acordo com variação da capacidade de ligação



Figuras 13 e 14: Tabelas de valores do cálculo de eficiência com variação de tempo de propagação

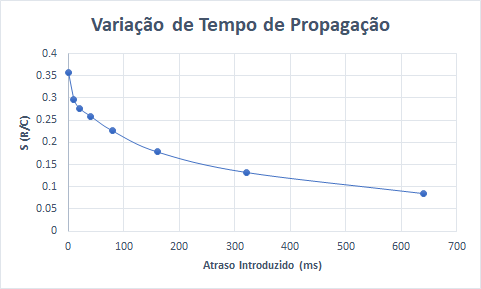


Figura 15: Gráfico do cálculo de eficiência com variação do tempo de propagação

**Anexos de código**

**Application.h:**

#pragma once

typedef unsigned int uint;

typedef unsigned char uchar;

typedef enum {

    Data = 1,

    Start = 2,

    End = 3

} ControlField;

typedef enum {

    FileSize = 1,

    FileName = 2,

} Type;

typedef enum {

    Receiver = 0,

    Sender = 1,

} machineType;

typedef struct {

    Type type;

    uchar length;

    uchar value[MAX\_BUF];

} TLV;

typedef struct {

    ControlField fieldC;

    uchar fieldN;

    uchar fieldL2;

    uchar fieldL1;

    uchar fieldP[MAX\_BUF];

} AppDataStruct;

typedef struct {

    ControlField fieldC;

    TLV fileSize;

    TLV fileName;

} AppControlStruct;

/\*\*

\* Assemble and send data block

\* @param {file descriptor} fd

\* @param {sequence number (%255)} sequenceNumber

\* @param {data} buffer

\* @param {data length} length

\* @return {...} 0 on success, -1 otherwise

\*/

int sendDataBlock(int fd, uint sequenceNumber, uchar \*buffer, uint length);

/\*\*

\* Receives data block and processes it

\* @param {file descriptor} fd

\* @param {returns sequence number (%255)} sequenceNumber

\* @param {returns data block received} buffer

\* @return {...} length of block received on success, -1 otherwise

\*/

int receiveDataBlock(int fd, uint \*sequenceNumber, uchar \*buffer);

/\*\*

\* Assemble and send control block

\* @param {file descriptor} fd

\* @param {control field(START/END)} fieldC

\* @param {TLV first message} fileSize

\* @param {TLV second message} fileName

\* @return {...} 0 on success, -1 otherwise

\*/

int sendControlBlock(int fd, int fieldC, uint fileSize, char \*fileName);

/\*\*

\* Receives control block and processes it

\* @param {file descriptor} fd

\* @param {returns control field(START/END)} type

\* @param {returns fileName} fileName

\* @return {...} FileSize on success, -1 otherwise

\*/

int receiveControlBlock(int fd, uint \*type, char \*fileName);

**Application.c:**

#include "protocolo.h"

#include "application.h"

#include "files.h"

int sendDataBlock(int fd, uint sequenceNumber, uchar \*buffer, uint length) {

    AppDataStruct data;

    // Create AppDataStruct

    data.fieldC = Data;

    data.fieldN = sequenceNumber;

    data.fieldL2 = length / 256;

    data.fieldL1 = length % 256;

    memcpy(data.fieldP, buffer, length);

    // Write AppDataStruct to Data Link layer

    if(llwrite(fd, (uchar\*)&data, sizeof(AppDataStruct)) < 0) {

        printf("Error while writing data block to Data Link layer!\n");

        return -1;

    }

    return 0;

}

int receiveDataBlock(int fd, uint \*sequenceNumber, uchar \*buffer) {

    AppDataStruct data;

    uint length;

    if(buffer == NULL) {

        printf("Received buffer is NULL!\n");

        return -1;

    }

    // Read block from Data Link layer

    if(llread(fd, (uchar\*)&data) < 0) {

        printf("Error while reading data block from Data Link layer!\n");

        return -1;

    }

    // Checks if block is a data block

    if (data.fieldC != Data) {

        printf("Error: Received block is not a data block!\n");

        return -1;

    }

    \*sequenceNumber = data.fieldN;

    length = data.fieldL2 \* 256 + data.fieldL1;

    // Does a copy of data from block received

    memcpy(buffer, data.fieldP, length);

    return length;

}

int sendControlBlock(int fd, int fieldC, uint fileSize, char \*fileName) {

    AppControlStruct control;

    // Converts fileSize to char\* to send via TLV struct

    char fileSizeString[64];

    sprintf(fileSizeString, "%d", fileSize);

    // Create AppControlStruct

    control.fieldC = fieldC;

    //FileSize

    control.fileSize.type = FileSize;

    control.fileSize.length = strlen(fileSizeString);

    for (uint i = 0; i < strlen(fileSizeString); i++)

        control.fileSize.value[i] = fileSizeString[i];

    //FileName

    control.fileName.type = FileName;

    control.fileName.length = strlen(fileName);

    for (uint i = 0; i < strlen(fileName); i++)

        control.fileName.value[i] = fileName[i];

    control.fileName.value[strlen(fileName)] = '\0';

    control.fileName.length++;

   // Write AppControlStruct to Data Link layer

    if(llwrite(fd, (uchar\*)&control, sizeof(AppControlStruct)) < 0) {

        printf("Error while writing fileSize to Data Link layer!\n");

        return -1;

    }

    return 0;

}

int receiveControlBlock(int fd, uint \*type , char \*fileName) {

    AppControlStruct control;

    if(fileName == NULL) {

        printf("Received fileName is NULL\n");

        return -1;

    }

    if(llread(fd, (uchar\*)&control) < 0) {

        printf("Error while reading control block from Data Link layer!\n");

        return -1;

    }

    // Control block type

    \*type = control.fieldC;

    // Checks if block is a control block

    if (control.fieldC != Start && control.fieldC != End) {

        printf("Error: Received block is not a control block!\n");

        return -1;

    }

    // FileName

    memcpy(fileName, control.fileName.value, control.fileName.length);

    // Returns FileSize

    return atoi((char\*)control.fileSize.value);

}

int main(int argc, char\* argv[]) {

    int portNumb = 0;

    if(strcmp(argv[0],"./reader") == 0){

        if(argc != 2){

            printf("Invalid call to reader\n");

            printf("Usage:\t./reader SerialPort\n\tex: ./reader 0\n");

            return -1;

        }

        portNumb = atoi(argv[1]);

        if(portNumb != 0 && portNumb != 1&&portNumb!=2){

            printf("Error: Serial port must be either 0 or 1");

            return -1;

        }

        if(receiveFile(portNumb) != 0) {

            printf("Error in receiveFile!\n");

            return -1;

        }

    }

    else if(strcmp(argv[0], "./writer") == 0){

        if(argc != 3){

            printf("Invalid call to reader\n");

            printf("Usage:\t./reader SerialPort FileName\n\tex: ./reader 0 file.txt\n");

            return -1;

        }

        portNumb = atoi(argv[1]);

        if(portNumb != 0 && portNumb != 1&&portNumb!=2){

            printf("Error: Serial port must be either 0 or 1");

            return -1;

        }

        char\* filename = argv[2];

        if(filename == NULL){

            printf("File Name invalid\n");

            return -1;

        }

        if(sendFile(portNumb, filename) < 0) {

            printf("Error in sendFile!\n");

            return -1;

        }

    }

    return 0;

}

**Files.h**

#pragma once

/\*\*

\* Sends file via application layer

\* @param {name of file do send} fileName

\* @return {...} number of bytes written

\*/

int sendFile(int portNumb, char \*fileName);

/\*\*

\* Receives file via application layer

\* @return {...} 0 on success, -1 otherwise

\*/

int receiveFile(int portNumber);

/\*\*

\* Returns number of bytes of a file

\* @param {filepointer} fp

\* @return {...} number of bytes on success, -1 otherwise

\*/

uint fileSize(FILE \*fp);

**Files.c**

#include "protocolo.h"

#include "application.h"

#include "files.h"

#include "time.h"

#include <sys/time.h>

int sendFile(int portNumber, char \*fileName) {

    uint size\_file;

    FILE\* file = fopen(fileName, "rb");

    if(file == NULL) {

        printf("Could not open %s!\n", fileName);

        return -1;

    }

    int fd = llopen(portNumber, FLAG\_LL\_OPEN\_TRANSMITTER);

    if(fd < 0) {

        printf("Error in llopen!\n");

        return -1;

    }

    size\_file = fileSize(file);

    // Send control package - START

    if(sendControlBlock(fd, Start, size\_file, fileName) < 0) {

        printf("Error in sendControlBlock!\n");

        return -1;

    }

    uint length, nBytes = 0, sequenceNumber = 0;

    uchar buffer[MAX\_BUF];

    struct timeval start, end;

    gettimeofday(&start,NULL);

    // While reads file sendDataPackage

    while(nBytes != size\_file) {

        length = fread(buffer, sizeof(char), MAX\_BUF, file);

        if(length < 0){

            perror("Failled to read from file");

            return -1;

        }

        if(sendDataBlock(fd, sequenceNumber%255, buffer, length) == EOF) {

            printf("Error in sendDataPackage!\n");

            return -1;

        }

        sequenceNumber++;

        nBytes += length;

    }

    gettimeofday(&end,NULL);

    printf("Send Time: %lf\n", end.tv\_sec + (double)end.tv\_usec/1000000 - (start.tv\_sec + (double)start.tv\_usec/1000000));

    if(fclose(file) != 0) {

        printf("Error while closing file!\n");

        return -1;

    }

    // Send control package - END

    if(sendControlBlock(fd, End, size\_file, fileName) < 0) {

        printf("Error in sendControlBlock!\n");

        return -1;

    }

    if(llclose(fd, FLAG\_LL\_CLOSE\_TRANSMITTER\_DISC) != LL\_CLOSE\_SUCESS) {

        printf("Error in llclose!\n");

        return -1;

    }

    return nBytes;

}

int receiveFile(int portNumber) {

    int fd = llopen(portNumber, FLAG\_LL\_OPEN\_RECEIVER);

    if(fd < 0) {

        printf("Error in llopen!\n");

        return -1;

    }

    char fileName[MAX\_BUF];

    uint fileSize = 0, controlType;

    // Receive control block - START

    if((fileSize = receiveControlBlock(fd, &controlType, fileName)) < 0) {

        printf("Error in sendControlBlock!\n");

        return -1;

    }

    if(controlType != Start) {

        printf("controlType value is not START\n");

        return -1;

    }

    //w: Create if does not exist / erase if exists

    FILE\* file = fopen(fileName, "w");

    if(file == NULL) {

        printf("Could not open/create %s!\n", fileName);

        return -1;

    }

    uint sequenceNumber, length = 0, totalLength = 0;

    uchar buffer[MAX\_BUF];

    while(totalLength != fileSize) {

        // Receive data block

        if((length = receiveDataBlock(fd, &sequenceNumber, buffer)) <= 0) {

            printf("Error in receiveDataBlock!\n");

            return -1;

        }

        totalLength += length;

        // Writes data block

        if(fwrite(buffer, sizeof(char), length, file) != length) {

            printf("Error while writing to output file!\n");

            return -1;

        }

    }

    if(fclose(file) != 0) {

        printf("Error while closing file!\n");

        return -1;

    }

    // Receive control block - END

    if(receiveControlBlock(fd, &controlType, fileName) < 0) {

        printf("Error in receiveControlBlock!\n");

        return -1;

    }

    if(controlType != End) {

        printf("controlType value is not END\n");

        return -1;

    }

    if(llclose(fd, FLAG\_LL\_CLOSE\_RECEIVER\_DISC) != LL\_CLOSE\_SUCESS) {

        printf("Error in llclose!\n");

        return -1;

    }

    return 0;

}

uint fileSize(FILE \*fp) {

    uint counter = 0;

    if(fp == NULL) {

        printf("File pointer is NULL!\n");

        return -1;

    }

    if(fseek(fp, 0, SEEK\_END) < 0) {

        printf("Error in fseek!\n");

        return -1;

    }

    if((counter = ftell(fp)) < 0) {

        printf("Error in ftell!\n");

        return -1;

    }

    if(fseek(fp, 0, SEEK\_SET) < 0) {

        printf("Error in fseek!\n");

        return -1;

    }

    return counter;

}

**Flags.h**

#pragma once

//Port settings

#define BAUDRATE B38400

#define MODEMDEVICE\_0 "/dev/ttyS0"

#define MODEMDEVICE\_1 "/dev/ttyS1"

#define MODEMDEVICE\_2 "/dev/ttyS2"

#define \_POSIX\_SOURCE 1

#define FALSE 0

#define TRUE 1

#define UNKNOWN\_PORT -1

#define OTHER\_ERROR -2

#define LER\_BYTE\_A\_BYTE 1

//LLOPEN

#define FLAG\_LL\_OPEN\_TRANSMITTER 1

#define FLAG\_LL\_OPEN\_RECEIVER 2

//LLCLOSE

#define FLAG\_LL\_CLOSE\_TRANSMITTER\_DISC 3

#define FLAG\_LL\_CLOSE\_TRANSMITTER\_UA 4

#define FLAG\_LL\_CLOSE\_RECEIVER\_DISC 5

#define FLAG\_LL\_CLOSE\_RECEIVER\_UA 6

#define FLAG\_HANDLER\_CALL 7

#define FLAG\_LL\_CLOSE\_TRANSMITTER 8

#define FLAG\_LL\_DATA\_SEND 9

#define FLAG\_DATA\_SEEKING\_ANSWER\_WITH1 10

#define FLAG\_DATA\_SEEKING\_ANSWER\_WITH0 11

#define FLAG\_DATA\_SENDING\_ANSWER\_RR\_WITH1 12

#define FLAG\_DATA\_SENDING\_ANSWER\_RR\_WITH0 13

#define FLAG\_DATA\_SENDING\_ANSWER\_REJ\_WITH1 14

#define FLAG\_DATA\_SENDING\_ANSWER\_REJ\_WITH0 15

#define LL\_CLOSE\_SUCESS 1

#define LL\_CLOSE\_FAIL -1

//5 Byte message

#define FLAG\_INDEX\_BEGIN 0

#define A\_INDEX 1

#define C\_INDEX 2

#define BCC\_INDEX 3

#define FLAG\_INDEX\_END 4

#define START\_INDEX -1

//Flags values

#define ESC 0x7d

#define ESC\_ESC 0x5d

#define ESC\_FLAG 0x5e

#define FLAG 0x7e

#define A\_CE\_AR 0x03

#define A\_CR\_AE 0x01

#define C(s) ((s == 1)? 0x40: 0x00)

#define C\_SET 0x03

#define C\_UA 0x07

#define C\_DISC 0x0B

#define C\_RR(r) ((r == 1)? 0x85: 0x05)

#define C\_REJ(r) ((r == 1)? 0x81: 0x01)

//Other variables

#define BUF\_SIZE 5

#define MAX\_BUF 384

#define DATA\_START\_INDEX 4

//Return types

#define INVALID\_PARAMS -1

#define READ\_SUCCESS 0

#define READ\_RR\_SUCESS 4

#define READ\_REJ\_SUCESS 5

#define READ\_FAIL -1

#define WRITE\_SUCCESS 0

#define WRITE\_FAIL 1

#define NO\_MEM -2

//State machine flags

#define ST\_START 0

#define ST\_FLAG\_RCV 1

#define ST\_A\_RCV 2

#define ST\_C\_RCV 3

#define ST\_BCC\_OK 4

#define ST\_D 4

#define ST\_ESC\_RCV 5

#define ST\_STOP 6

#define ST\_C\_RCV\_RR 7

#define ST\_C\_RCV\_REJ 8

#define ST\_BCC\_OK\_RR 9

#define ST\_BCC\_OK\_REJ 10

#define HANDLING\_CLOSE\_EMISSOR 0

#define HANDLING\_CLOSE\_RECETOR 1

#define HANDLING\_UNDEFINED -1;

**Protocolo.h:**

#pragma once

#include <sys/types.h>

#include <sys/stat.h>

#include <termios.h>

#include <string.h>

#include <fcntl.h>

#include <stdlib.h>

#include <stdio.h>

#include <stdbool.h>

#include <unistd.h>

#include <signal.h>

#include <errno.h>

#include "flags.h"

typedef struct {

    unsigned char flag ;

    unsigned char fieldA;

    unsigned char fieldC;

    unsigned char fieldBCC1;

    unsigned char\* fieldD;

    unsigned char\* fieldBCC2;

    //Other important variables

    unsigned int bcc2StufSize;

    unsigned int dataStufSize;

    size\_t size\_of\_data\_frame;

} DataStruct;

/\*\*

\* Function responsible for establishing logical connection between two points

Functionality diagram:

Sender                          Receiver

------>Send SET MESSAGE

                            <-----Receive SET MESSAGE

                            <--------Send a UA confirmation

Receive UA <-------

CONNECTION ESTABLISHED

This function is protected by timeout alarm

\* @param {...} port\_number

\* @param {TRANSMITTER/RECEIVER} flag

\* @return {...} file descriptor on sucess, -1 otherwise

\*/

int llopen(int port\_number, int flag);

/\*\*

\*

LL WRITE IS THE MAIN API FUNCTION USED BY THE UPPER LAYER TO SEND THEIR MESSAGES TO THE OTHER MACHINE

LL WRITE IS ALSO PROTECTED BY A TIME OUT ALARM

1) MESSAGE IS SENT

2) CONFIRMATION WAITED

3) RR OR REJ COULD BE THE ANSWER.

IF RR SEND NEXT FRAME, IF REJ, ANTICIPATE TIMEOUT AND DO A RETRANSMISSION

\* @param {file descriptor} fd

\* @param {...} buffer

\* @param {...} length

\* @return {...} number of bytes written on success, 0 otherwise

\*/

int llwrite(int fd, unsigned char \* buffer, int length);

/\*\*

 \* LL READ IS THE API FUNCTION PROVIDED TO RECEIVE A MESSAGE FROM THE OTHER MACHINE

 \*

 \* LL READ IS INDEED CONSTRUCTED IN ORDER TO TEST BCC VALUES AND TO DISCARD HEADERS IF NEEDED.

 \*

 \*

 \*

\*

\* @param {file descriptor} fd

\* @param {...} buffer

\* @return {...} buffer size on success, 0 therwise

\*/

int llread(int fd, unsigned char \* buffer);

/\*\*

\* LLCLOSE IS THE API FUNTION PROVIDED TO DISCONECT THE LOGICAL CHANNEL ESTABLISHED BY THE LLOPEN.

    AFTER LLCLOSE BOTH MACHINES ARE READY TO SHUTDOWN SAFELY THEIR COMUNICATION BRIDGE

\*

\*

\* @param {file descriptor} fd

\* @param {TRANSMITTER/RECEIVER} flag

\* @return {...} 1 on sucsess, negativo value otherwise

\*/

int llclose(int fd, int flag);

**Protocolo.c**

#include "protocolo.h"

#include "time.h"

//LOCAL MACROS

#define MAX\_RETR 3

#define TIMEOUT 3

#define DELAY (640\*1000)

//GLOBAL VARS

DataStruct \*pointer\_to\_data=NULL;

static int n\_tries = MAX\_RETR;

static struct termios oldtio;

static int type\_handling=HANDLING\_UNDEFINED;

//LOCAL FUNCTIONS

void alarm\_handler\_set\_signal(int signo);

static int fd\_for\_handler;

int sendBlock( int flag,  int fd);

int readBlock( int flag,  int fd);

DataStruct createMessage(unsigned int sequenceNumber, unsigned char \*buffer, int length);

unsigned int BCC2Stufying(unsigned char \*BCC2);

unsigned int dataStuffing(unsigned char \*data, int length, unsigned char \*fieldD);

//

//FUNCTION RESPONSIBLE TO CONVERT SERIAL PORT NUMBER TO A FD IN THE FD TABLE LIST

int openNonCanonical(int port\_number)

{

  struct termios newtio;

  int fd\_port = 0;

  switch (port\_number)

  {

  case 0:

    fd\_port = open(MODEMDEVICE\_0, O\_RDWR | O\_NOCTTY);

    break;

  case 1:

    fd\_port = open(MODEMDEVICE\_1, O\_RDWR | O\_NOCTTY);

    break;

  case 2:

    fd\_port = open(MODEMDEVICE\_2, O\_RDWR | O\_NOCTTY);

  break;

  default:

    return UNKNOWN\_PORT;

  }

  if (fd\_port < 0)

  {

    perror("Error");

    exit(OTHER\_ERROR);

  }

  if (tcgetattr(fd\_port, &oldtio) == -1)

  { /\* save current port settings \*/

    perror("tcgetattr");

    exit(OTHER\_ERROR);

  }

  bzero(&newtio, sizeof(newtio));

  newtio.c\_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;

  newtio.c\_iflag = IGNPAR;

  newtio.c\_oflag = 0;

  /\* set input mode (non-canonical, no echo,...) \*/

  newtio.c\_lflag = 0;

  newtio.c\_cc[VTIME] = 0;              /\* inter-character timer unused \*/

  newtio.c\_cc[VMIN] = LER\_BYTE\_A\_BYTE; /\* blocking read until 1 char received \*/

  /\*

    VTIME e VMIN devem ser alterados de forma a proteger com um temporizador a

    leitura do(s) pr�ximo(s) caracter(es)

  \*/

  tcflush(fd\_port, TCIOFLUSH);

  if (tcsetattr(fd\_port, TCSANOW, &newtio) == -1)

  {

    perror("tcsetattr");

    exit(OTHER\_ERROR);

  }

  return fd\_port;

}

//HANDLER FOR THE LLOPEN SET PROTECTION

void alarm\_handler\_set\_signal(int signo)

{

  if (signo != SIGALRM)

  {

    printf("Handler nao executado\n");

    return;

  }

  if (n\_tries > 0)

  {

    if (sendBlock(FLAG\_LL\_OPEN\_TRANSMITTER, fd\_for\_handler) != WRITE\_SUCCESS)

    {

      printf("Error in sendSet function no handler\n");

    }

    alarm(TIMEOUT);

    n\_tries--;

  }

  else

  {

    llclose(fd\_for\_handler, FLAG\_HANDLER\_CALL);

    exit(-1);

  }

  return;

}

//HANDLER FOR THE LLCLOSE DISC PROTECTION

void alarm\_handler\_disc\_signal(int signo)

{

  if (signo != SIGALRM)

  {

    printf("Handler nao executado\n");

    return;

  }

  if (n\_tries > 0)

  {

    if(type\_handling==HANDLING\_CLOSE\_EMISSOR){

      if (sendBlock(FLAG\_LL\_CLOSE\_TRANSMITTER\_DISC,fd\_for\_handler) != WRITE\_SUCCESS)

      {

        printf("Error in sendSet function no handler\n");

      }

    }else if(type\_handling==HANDLING\_CLOSE\_RECETOR){

      if (sendBlock(FLAG\_LL\_CLOSE\_RECEIVER\_DISC,fd\_for\_handler) != WRITE\_SUCCESS)

      {

        printf("Error in sendSet function no handler\n");

      }

    }else{

      printf("Handling undefined. STOP\n");

      exit(-1);

    }

    alarm(TIMEOUT);

    n\_tries--;

  }

  else

  {

    printf("No LLCLOSE fui incapaz de desligar corretamente\nMatei o processo\n");

    exit(-1);

  }

  return;

}

//HANDLER FOR THE LLWRITE PROTECTION

void alarm\_handler\_data(int signo)

{

  if (signo != SIGALRM)

  {

    printf("Handler nao executado\n");

    return;

  }

  if (n\_tries > 0)

  {

       printf("Fiz retransmissao\n");

    sendBlock(FLAG\_LL\_DATA\_SEND,fd\_for\_handler);

    alarm(TIMEOUT);

    n\_tries--;

  }

  else

  {

    printf("No LLCLOSE fui incapaz de desligar corretamente\nMatei o processo\n");

    exit(-1);

  }

  return;

}

//FUNCTION TO TESTE THE BBC2 VALUE OF THE DATA SEGMENT OF A FRAME. IF FAIL REJ SHOULD BE SENT

int checkBCC2(unsigned char \*buffer, int size)

{

  unsigned char bcc = buffer[size - 1];

  unsigned char bcc\_check = 0;

  if(size<=1){

    printf("Size mudou abrupotamente\n");

    return -1;

  }

  //Ate size n-1, por que o data vai estar em buffer entre [0,size-1]

  for (int i = 0; i < (size - 1); i++)

  {

    bcc\_check ^= buffer[i];

  }

  return bcc == bcc\_check;

}

//DESTUFFING ROUTINE. CALLED BY THE READER SIDE TO START THE BYTE INTEGRATY ANALISYS

int byteDeStuffing(unsigned char \*buf,  int size\_orig)

{

  unsigned int size\_dest = 0;

  if (buf == NULL)

  {

    perror("Failled to allocate memory");

    exit(NO\_MEM);

  }

  if(size\_orig<=1){

    printf("O SIZE MUDOU ABRUPTAMENTE\n");

    return -1;

  }

  for (int i = DATA\_START\_INDEX; i < (size\_orig - 1); i++)

  {

    if (buf[i] == ESC)

    {

      i++;

      if (buf[i] == ESC\_FLAG)

        buf[size\_dest] = FLAG;

      else

        buf[size\_dest] = ESC;

    }

    else if(buf[i]!=FLAG)

      buf[size\_dest] = buf[i];

    else if(buf[i]==FLAG){

      printf("FLAG no sitio ERRADO\n REJ\n");

      return 4;

    }

    size\_dest++;

  }

  return size\_dest;

}

//SINCE HANDLERS CANT IN CASE O RETRANSMISSION CALL LLWRITE OR LLOPEN OR LLCLOSE AS WHOLE SINCE THIS ROUTINES MAKE OTHER FUNCTIONALITIES FURTHER THAN JUST SENT INFORMATION. API MUST BE SUPPORTED BY A ENCAPSALUTED SEGMENT WHOSE REPONSABILITY IS JUST THE INFORMATION SEND. THAT SEGMENT IS SENDBLOCK

int sendBlock(int flag, int fd)

{

  unsigned char buf[BUF\_SIZE + 1];

  //A Flag inicial e comum a qualquer trama

  if (flag == FLAG\_LL\_OPEN\_RECEIVER || flag == FLAG\_LL\_OPEN\_TRANSMITTER)

  {

    //E se usassemos esta funcao para depois enviarmos qualquer informacao que nao so LLOPEN?

    buf[FLAG\_INDEX\_BEGIN] = FLAG;

    buf[A\_INDEX] = A\_CE\_AR;

    if (flag == FLAG\_LL\_OPEN\_TRANSMITTER)

    {

      buf[C\_INDEX] = C\_SET;

    }

    else

    {

      buf[C\_INDEX] = C\_UA;

    }

    buf[BCC\_INDEX] = buf[A\_INDEX] ^ buf[C\_INDEX];

    buf[FLAG\_INDEX\_END] = FLAG;

    int bytes\_send = write(fd, buf, BUF\_SIZE);

    if (bytes\_send != BUF\_SIZE)

    {

      perror("Error writing in llopen:");

      return WRITE\_FAIL;

    }

  }

  else if (flag == FLAG\_LL\_CLOSE\_TRANSMITTER\_DISC || flag == FLAG\_LL\_CLOSE\_TRANSMITTER\_UA)

  {

    buf[FLAG\_INDEX\_BEGIN] = FLAG;

    buf[A\_INDEX] = A\_CE\_AR;

    if (flag==FLAG\_LL\_CLOSE\_TRANSMITTER\_DISC)

    {

      buf[C\_INDEX] = C\_DISC;

    }

    else if (flag==FLAG\_LL\_CLOSE\_TRANSMITTER\_UA)

    {

      buf[C\_INDEX] = C\_UA;

    }

    buf[BCC\_INDEX] = buf[A\_INDEX] ^ buf[C\_INDEX];

    buf[FLAG\_INDEX\_END] = FLAG;

    if (write(fd, buf, BUF\_SIZE) != BUF\_SIZE)

    {

      perror("Erro no write do FLAG\_LL\_CLOSE\_TRANSMITTER\_DISC:");

      return WRITE\_FAIL;

    }

  }

  else if (flag == FLAG\_LL\_CLOSE\_RECEIVER\_DISC)

  {

    buf[FLAG\_INDEX\_BEGIN] = FLAG;

    buf[A\_INDEX] = A\_CR\_AE;

    if (FLAG\_LL\_CLOSE\_RECEIVER\_DISC)

    {

      buf[C\_INDEX] = C\_DISC;

    }

    buf[BCC\_INDEX] = buf[A\_INDEX] ^ buf[C\_INDEX];

    buf[FLAG\_INDEX\_END] = FLAG;

    if (write(fd, buf, BUF\_SIZE) != BUF\_SIZE)

    {

      perror("Erro no write do FLAG\_LL\_CLOSE\_TRANSMITTER\_DISC:");

      return WRITE\_FAIL;

    }

  }

  else if(flag==FLAG\_LL\_DATA\_SEND)

  {

    int n\_bytes=0;

    n\_bytes += write(fd, (char\*)&pointer\_to\_data->flag, sizeof(pointer\_to\_data->flag));

    n\_bytes += write(fd,(char \*)&pointer\_to\_data->fieldA, sizeof(pointer\_to\_data->fieldA));

    n\_bytes += write(fd, (char \*)&pointer\_to\_data->fieldC, sizeof(pointer\_to\_data->fieldC));

    n\_bytes += write(fd, (char\*) &pointer\_to\_data->fieldBCC1, sizeof(pointer\_to\_data->fieldBCC1));

    n\_bytes += write(fd, pointer\_to\_data->fieldD, pointer\_to\_data->dataStufSize);

    n\_bytes += write(fd, pointer\_to\_data->fieldBCC2, pointer\_to\_data->bcc2StufSize);

    n\_bytes += write(fd, (char \*)&pointer\_to\_data->flag, sizeof(pointer\_to\_data->flag));

    if(n\_bytes != (pointer\_to\_data->bcc2StufSize + pointer\_to\_data->dataStufSize + 5)){

      printf("Nao escrevi o bloco de data todo");

      return WRITE\_FAIL;

    }else{

      return n\_bytes;

    }

  }

  else if (flag == FLAG\_DATA\_SENDING\_ANSWER\_RR\_WITH1 || flag == FLAG\_DATA\_SENDING\_ANSWER\_RR\_WITH0)

  {

    buf[FLAG\_INDEX\_BEGIN] = FLAG;

    buf[A\_INDEX] = A\_CE\_AR;

    if(flag == FLAG\_DATA\_SENDING\_ANSWER\_RR\_WITH1)

      buf[C\_INDEX] = C\_RR(1);

    else

      buf[C\_INDEX] = C\_RR(0);

    buf[BCC\_INDEX] = buf[A\_INDEX] ^ buf[C\_INDEX];

    buf[FLAG\_INDEX\_END] = FLAG;

    if (write(fd, buf, BUF\_SIZE) != BUF\_SIZE)

    {

      perror("Erro no write do FLAG\_DATA\_SENDING\_ANSWER\_WITH1:");

      return WRITE\_FAIL;

    }

  }

  else if (flag == FLAG\_DATA\_SENDING\_ANSWER\_REJ\_WITH1 || flag == FLAG\_DATA\_SENDING\_ANSWER\_REJ\_WITH0)

  {

    buf[FLAG\_INDEX\_BEGIN] = FLAG;

    buf[A\_INDEX] = A\_CE\_AR;

    if(flag == FLAG\_DATA\_SENDING\_ANSWER\_REJ\_WITH1)

      buf[C\_INDEX] = C\_REJ(1);

    else

      buf[C\_INDEX] = C\_REJ(0);

    buf[BCC\_INDEX] = buf[A\_INDEX] ^ buf[C\_INDEX];

    buf[FLAG\_INDEX\_END] = FLAG;

    if (write(fd, buf, BUF\_SIZE) != BUF\_SIZE)

    {

      perror("Erro no write do FLAG\_DATA\_SENDING\_ANSWER\_WITH1:");

      return WRITE\_FAIL;

    }

  }

  else

  {

    return WRITE\_FAIL;

  }

  return WRITE\_SUCCESS;

}

int readBlock(int flag, int fd)

{

  unsigned char leitura;

  unsigned int size = 0, state = ST\_START;

  if (flag == FLAG\_LL\_OPEN\_RECEIVER || flag == FLAG\_LL\_OPEN\_TRANSMITTER  || flag == FLAG\_LL\_CLOSE\_TRANSMITTER\_UA || flag == FLAG\_LL\_CLOSE\_RECEIVER\_DISC)

  {

    for (size = 0; state != ST\_STOP && size < MAX\_BUF; size++)

    {

      //A mensagem vai ser lida byte a byte para garantir que nao há falha de informacao

      if ((read(fd, &leitura, 1) != 0))

      {

      }

      switch (state)

      {

      case ST\_START:

      {

        //check FLAG byte

        if (leitura == FLAG)

        {

          state = ST\_FLAG\_RCV;

        }

      }

      break;

      case ST\_FLAG\_RCV:

        switch (leitura)

        {

        case A\_CE\_AR:

        {

          //Recebi uma mensagem do emissor ou um resposta do recetor

          state = ST\_A\_RCV;

          break;

        }

        case FLAG:

          //Same state

          break;

        default:

          state = ST\_START;

        }

        break;

      case ST\_A\_RCV:

        switch (leitura)

        {

        case C\_UA:

          if (flag == FLAG\_LL\_OPEN\_TRANSMITTER || flag == FLAG\_LL\_CLOSE\_RECEIVER\_UA)

          {

            //received C\_SET and is Transmitter, go to state C received

            //Or its llclose receiver a receiver

            state = ST\_C\_RCV;

            break;

          }

          else

            //received C\_SET and is Receiver, go to state start

            state = ST\_START;

          break;

        case C\_SET:

          if (flag == FLAG\_LL\_OPEN\_RECEIVER)

          {

            //received C\_SET and is Receiver, go to state C received

            state = ST\_C\_RCV;

          }

          else

            //received C\_SET and is Transmitter, go to state start

            state = ST\_START;

          break;

        case C\_DISC:

          if (flag == FLAG\_LL\_CLOSE\_RECEIVER\_DISC || flag == FLAG\_LL\_CLOSE\_TRANSMITTER\_DISC)

          {

            //received UA on LL\_CLOSE

            state = ST\_C\_RCV;

          }

          else

            //received C\_SET and is Transmitter, go to state start

            state = ST\_START;

          break;

        case FLAG:

          //received FLAG, go to flag state

          state = ST\_FLAG\_RCV;

          break;

        default:

          //received other, go to start

          state = ST\_START;

          break;

        }

        break;

      case ST\_C\_RCV:

      {

        //received BCC, check BCC

        if (leitura == (A\_CE\_AR ^ C\_SET) && flag == FLAG\_LL\_OPEN\_RECEIVER)

        {

          //BCC correct

          state = ST\_BCC\_OK;

        }

        else if (leitura == (A\_CE\_AR ^ C\_UA) && flag == FLAG\_LL\_OPEN\_TRANSMITTER)

        {

          state = ST\_BCC\_OK;

        }

        else if (leitura == (A\_CE\_AR ^ C\_DISC) && flag == FLAG\_LL\_CLOSE\_RECEIVER\_DISC)

        {

          state = ST\_BCC\_OK;

        }

        else if (leitura == (A\_CR\_AE ^ C\_DISC) && flag == FLAG\_LL\_CLOSE\_TRANSMITTER\_DISC)

        {

          state = ST\_BCC\_OK;

        }

        else if (leitura == FLAG)

        {

          //Received FLAG

          state = ST\_FLAG\_RCV;

        }

        else

        {

          //Received other

          state = ST\_START;

        }

        break;

      }

      case ST\_BCC\_OK:

        //check FLAG byte

        if (leitura == FLAG)

        {

          //received all, stop cycle

          return READ\_SUCCESS;

        }

        else

          //received other, go to start

          state = ST\_START;

        break;

      default:

        state = ST\_START;

        break;

      }

    }

    return READ\_FAIL;

  }

  else if(flag==FLAG\_LL\_CLOSE\_TRANSMITTER){

    for (size = 0; state != ST\_STOP && size < MAX\_BUF; size++)

    {

      //A mensagem vai ser lida byte a byte para garantir que nao há falha de informacao

      if ((read(fd, &leitura, 1) != 0))

      {

      }

      switch (state)

      {

      case ST\_START:

      {

        //check FLAG byte

        if (leitura == FLAG)

        {

          state = ST\_FLAG\_RCV;

        }

      }

      break;

      case ST\_FLAG\_RCV:

        switch (leitura)

        {

        case A\_CR\_AE:

        {

          //Recebi uma mensagem do emissor ou um resposta do recetor

          state = ST\_A\_RCV;

          break;

        }

        case FLAG:

          //Same state

          break;

        default:

          state = ST\_START;

        }

        break;

      case ST\_A\_RCV:

        switch (leitura)

        {

        case C\_DISC:

          state = ST\_C\_RCV;

        break;

        case FLAG:

          //received FLAG, go to flag state

          state = ST\_FLAG\_RCV;

          break;

        default:

          //received other, go to start

          state = ST\_START;

          break;

        }

        break;

      case ST\_C\_RCV:

      {

        //received BCC, check BCC

        if (leitura == (A\_CR\_AE ^ C\_DISC))

        {

          state = ST\_BCC\_OK;

        }

        else if (leitura == FLAG)

        {

          //Received FLAG

          state = ST\_FLAG\_RCV;

        }

        else

        {

          //Received other

          state = ST\_START;

        }

        break;

      }

      case ST\_BCC\_OK:

        //check FLAG byte

        if (leitura == FLAG)

        {

          //received all, stop cycle

          return READ\_SUCCESS;

        }

        else

          //received other, go to start

          state = ST\_START;

        break;

      default:

        state = ST\_START;

        break;

      }

    }

    return READ\_FAIL;

  }

  else if(flag == FLAG\_LL\_CLOSE\_RECEIVER\_UA){

    for (size = 0; state != ST\_STOP && size < MAX\_BUF; size++)

    {

      //A mensagem vai ser lida byte a byte para garantir que nao há falha de informacao

      if ((read(fd, &leitura, 1) != 0))

      {

      }

      switch (state)

      {

      case ST\_START:

      {

        //check FLAG byte

        if (leitura == FLAG)

        {

          state = ST\_FLAG\_RCV;

        }

      }

      break;

      case ST\_FLAG\_RCV:

        switch (leitura)

        {

        case A\_CE\_AR:

        {

          //Recebi uma mensagem do emissor ou um resposta do recetor

          state = ST\_A\_RCV;

          break;

        }

        case FLAG:

          //Same state

          break;

        default:

          state = ST\_START;

        }

        break;

      case ST\_A\_RCV:

        switch (leitura)

        {

        case C\_UA:

          state = ST\_C\_RCV;

        break;

        case FLAG:

          //received FLAG, go to flag state

          state = ST\_FLAG\_RCV;

          break;

        default:

          //received other, go to start

          state = ST\_START;

          break;

        }

        break;

      case ST\_C\_RCV:

      {

        //received BCC, check BCC

        if (leitura == (A\_CE\_AR ^ C\_UA))

        {

          state = ST\_BCC\_OK;

        }

        else if (leitura == FLAG)

        {

          //Received FLAG

          state = ST\_FLAG\_RCV;

        }

        else

        {

          //Received other

          state = ST\_START;

        }

        break;

      }

      case ST\_BCC\_OK:

        //check FLAG byte

        if (leitura == FLAG)

        {

          //received all, stop cycle

          return READ\_SUCCESS;

        }

        else

          //received other, go to start

          state = ST\_START;

        break;

      default:

        state = ST\_START;

        break;

      }

    }

    return READ\_FAIL;

  }

  else if(flag==FLAG\_DATA\_SEEKING\_ANSWER\_WITH0){

    for (size = 0; state != ST\_STOP && size < MAX\_BUF; size++)

    {

      //A mensagem vai ser lida byte a byte para garantir que nao há falha de informacao

      if ((read(fd, &leitura, 1) != 0))

      {

      }

      switch (state)

      {

      case ST\_START:

      {

        //check FLAG byte

        if (leitura == FLAG)

        {

          state = ST\_FLAG\_RCV;

        }

      }

      break;

      case ST\_FLAG\_RCV:

        switch (leitura)

        {

        case A\_CE\_AR:

        {

          //Recebi uma mensagem do emissor ou um resposta do recetor

          state = ST\_A\_RCV;

          break;

        }

        case FLAG:

          //Same state

          break;

        default:

          state = ST\_START;

        }

        break;

      case ST\_A\_RCV:

        switch (leitura)

        {

        case C\_REJ(0):

          state = ST\_C\_RCV\_REJ;

        break;

        case C\_RR(0):

          state=ST\_C\_RCV\_RR;

        break;

        case FLAG:

          //received FLAG, go to flag state

          state = ST\_FLAG\_RCV;

          break;

        default:

          //received other, go to start

          state = ST\_START;

          break;

        }

        break;

      case ST\_C\_RCV\_REJ:

      {

        //received BCC, check BCC

        if (leitura == (A\_CE\_AR ^ C\_REJ(0)))

        {

          state = ST\_BCC\_OK\_REJ;

        }

        else if (leitura == FLAG)

        {

          //Received FLAG

          state = ST\_FLAG\_RCV;

        }

        else

        {

          //Received other

          state = ST\_START;

        }

        break;

      }

      case ST\_C\_RCV\_RR:

      {

        //received BCC, check BCC

        if (leitura == (A\_CE\_AR ^ C\_RR(0)))

        {

          state = ST\_BCC\_OK\_RR;

        }

        else if (leitura == FLAG)

        {

          //Received FLAG

          state = ST\_FLAG\_RCV;

        }

        else

        {

          //Received other

          state = ST\_START;

        }

        break;

      }

      case ST\_BCC\_OK\_REJ:

        //check FLAG byte

        if (leitura == FLAG)

        {

          //received all, stop cycle

          return READ\_REJ\_SUCESS;

        }

        else

          //received other, go to start

          state = ST\_START;

        break;

      case ST\_BCC\_OK\_RR:

        //check FLAG byte

        if (leitura == FLAG)

        {

          //received all, stop cycle

          return READ\_RR\_SUCESS;

        }

        else

          //received other, go to start

          state = ST\_START;

        break;

      default:

        state = ST\_START;

        break;

      }

    }

    return READ\_FAIL;

  }

  else if(flag==FLAG\_DATA\_SEEKING\_ANSWER\_WITH1){

    for (size = 0; state != ST\_STOP && size < MAX\_BUF; size++)

    {

      //A mensagem vai ser lida byte a byte para garantir que nao há falha de informacao

      if ((read(fd, &leitura, 1) != 0))

      {

      }

      switch (state)

      {

      case ST\_START:

      {

        //check FLAG byte

        if (leitura == FLAG)

        {

          state = ST\_FLAG\_RCV;

        }

      }

      break;

      case ST\_FLAG\_RCV:

        switch (leitura)

        {

        case A\_CE\_AR:

        {

          //Recebi uma mensagem do emissor ou um resposta do recetor

          state = ST\_A\_RCV;

          break;

        }

        case FLAG:

          //Same state

          break;

        default:

          state = ST\_START;

        }

        break;

      case ST\_A\_RCV:

        switch (leitura)

        {

        case C\_REJ(1):

          state = ST\_C\_RCV\_REJ;

        break;

        case C\_RR(1):

          state=ST\_C\_RCV\_RR;

        break;

        case FLAG:

          //received FLAG, go to flag state

          state = ST\_FLAG\_RCV;

          break;

        default:

          //received other, go to start

          state = ST\_START;

          break;

        }

        break;

      case ST\_C\_RCV\_REJ:

      {

        //received BCC, check BCC

        if (leitura == (A\_CE\_AR ^ C\_REJ(1)))

        {

          state = ST\_BCC\_OK\_REJ;

        }

        else if (leitura == FLAG)

        {

          //Received FLAG

          state = ST\_FLAG\_RCV;

        }

        else

        {

          //Received other

          state = ST\_START;

        }

        break;

      }

      case ST\_C\_RCV\_RR:

      {

        if (leitura == (A\_CE\_AR ^ C\_RR(1)))

        {

          state = ST\_BCC\_OK\_RR;

          break;

        }

        else if (leitura == FLAG)

        {

          //Received FLAG

          state = ST\_FLAG\_RCV;

        }

        else

        {

          //Received other

          state = ST\_START;

        }

        break;

      }

      case ST\_BCC\_OK\_RR:{

        //check FLAG byte

        if (leitura == FLAG)

        {

          //received all, stop cycle

          return READ\_RR\_SUCESS;

        }

        else

          //received other, go to start

          state = ST\_START;

        break;

      }

      case ST\_BCC\_OK\_REJ:{

        //check FLAG byte

        if (leitura == FLAG)

        {

          //received all, stop cycle

          return READ\_REJ\_SUCESS;

        }

        else

          //received other, go to start

          state = ST\_START;

        break;

      }

      default:

        state = ST\_START;

        break;

      }

    }

    return READ\_FAIL;

  }

  else

  {

    return READ\_FAIL;

  }

}

int llopen(int port\_number, int flag)

{

  int fd;

  fd = openNonCanonical(port\_number);

  fd\_for\_handler=fd;

  int ret\_read\_block = READ\_FAIL;

  n\_tries=MAX\_RETR;

  if (fd == UNKNOWN\_PORT)

  {

    printf("Unknown port, must be either 0 or 1\n");

    exit(UNKNOWN\_PORT);

  }

  //Transmitter

  if (flag == FLAG\_LL\_OPEN\_TRANSMITTER)

  {

    if (signal(SIGALRM, alarm\_handler\_set\_signal) == SIG\_ERR)

    {

      perror("Error instaling SIG ALARM handler\n");

      return -1;

    }

    if (sendBlock(FLAG\_LL\_OPEN\_TRANSMITTER, fd) != WRITE\_SUCCESS)

    {

      printf("Error in sendSet function\n");

    }

    alarm(TIMEOUT);

    while (ret\_read\_block == READ\_FAIL)

    {

      ret\_read\_block = readBlock(FLAG\_LL\_OPEN\_TRANSMITTER, fd);

    }

    if (ret\_read\_block == READ\_FAIL)

    {

      printf("AS tentativas todas deram fail. Retornei erro\n");

      return READ\_FAIL;

    }

    if (signal(SIGALRM, SIG\_IGN) == SIG\_ERR)

    {

      perror("Error in ignoring SIG ALARM handler");

    }

  }

  //Receiver

  else if (flag == FLAG\_LL\_OPEN\_RECEIVER)

  {

    if (readBlock(FLAG\_LL\_OPEN\_RECEIVER, fd) != READ\_SUCCESS)

    {

      perror("Error in reading from llopen:");

      return -1;

    }

    if (sendBlock(FLAG\_LL\_OPEN\_RECEIVER, fd) != WRITE\_SUCCESS)

    {

      perror("Error in writing from llopen:");

      return -1;

    }

  }

  //LL open deve retornar identificador da ligacao de dados

  return fd;

}

/\*LLwrite Enviar uma mensagem do emissor para o recetor

-Tem de preencher o bloco: Com o BCC calc

-Tem de fazer stuffing

-Tem de enviar tudo

-Tem de ficar a espera da resposta: Se for REJ Envio outra vez (vou enivar para ja tudo. Depois vejo como vamos para llread)

//So BCC2 e DADOS sao tidos em conta para stuffing. Os outros codigos estao escolhidos para nao ocorrer mal

\*/

int llwrite(int fd, unsigned char \*buffer, int length)

{

  //Nao esquecer que elas sao sempre cruzados envio 0 recebo 1. Comeco a 0

  static unsigned int sequenceNumber = 0;

  int num\_bytes = 0;

  int ret\_resposta=READ\_FAIL;

  DataStruct data = createMessage(sequenceNumber, buffer, length);

  data.size\_of\_data\_frame=sizeof(data);

  pointer\_to\_data=&data;

  while(ret\_resposta==READ\_FAIL||ret\_resposta==READ\_REJ\_SUCESS){

    if (signal(SIGALRM, alarm\_handler\_data) == SIG\_ERR)

    {

      perror("Error in ignoring SIG ALARM handler");

    }

    // usleep(DELAY);

    num\_bytes=sendBlock(FLAG\_LL\_DATA\_SEND,fd);

    if(num\_bytes==WRITE\_FAIL){

      printf("Erro a enviar o bloco\n");

      return -1;

    }

    n\_tries=MAX\_RETR;

    alarm(TIMEOUT);

    if(sequenceNumber==0){

      ret\_resposta=readBlock(FLAG\_DATA\_SEEKING\_ANSWER\_WITH1,fd);

    }else if(sequenceNumber==1){

      ret\_resposta=readBlock(FLAG\_DATA\_SEEKING\_ANSWER\_WITH0,fd);

    }

    if (signal(SIGALRM, SIG\_IGN) == SIG\_ERR)

    {

        perror("Error in ignoring SIG ALARM handler");

    }

    alarm(0);

  }

  free(data.fieldD);

  free(data.fieldBCC2);

  sequenceNumber = (sequenceNumber + 1) % 2;

  return num\_bytes;

}

/\*

Em LL read:

-Recebemos a mensagem

-Verificamos o cabecalho.

Dependendo do valor avancamos ou nao para o processamento dos dados

Nao esta certo: Estado morto. Timeout

Esta certo:

Fazer o distuffing

Testar Bcc2.

Testar se a trama e repetida ou nao

Se repetida -> envio um RR nao passo nada à app

Se nova -> Passo a app e envio a app

----> Tambem se aplica ao caso de cabecalho certo, dados errados pq a seriacao da trama vem no C\_VALUE

\*/

//Buffer vai retornar a app

//Intermedio um buf dinamico porque nao sei qual e o tamanho lido

int llread(int fd, unsigned char \*buffer)

{

  static unsigned int r = 0;

  int size\_buf=0, max\_size=2\*MAX\_BUF;

  unsigned int state = ST\_START;

  bool end = false;

  unsigned char \*buf = NULL;

  int size\_buffer = 0;

  if (buffer == NULL || fd < 0)

  {

    printf("Passei parametros invalidos a llread\n");

    return INVALID\_PARAMS;

  }

  buf = (unsigned char \*)malloc(max\_size \* sizeof(unsigned char));

  if (buf == NULL)

  {

    perror("Failled to allocate memory");

    exit(NO\_MEM);

  }

  //While data is rejected be cause of errors go to state machine

  while (!end)

  {

    //State machine

    for (; state != ST\_STOP; size\_buf++)

    {

      //Check size

      if (size\_buf > max\_size)

      {

        //Se tiver cheio, faco 2 x o buf

        max\_size \*= 2;

        buf = (unsigned char \*)realloc((void \*)buf, max\_size \* sizeof(unsigned char));

        if (buf == NULL)

        {

          perror("Failled to allocate memory");

          exit(NO\_MEM);

        }

      }

      //Read byte para o buf

      if (read(fd, &buf[size\_buf], 1) < 0)

      {

        free(buf);

        perror("Failled to read");

        return READ\_FAIL;

      }

      //Go through state machine

      switch (state)

      {

        case ST\_START:

        {

          if (buf[size\_buf] == FLAG)

            state = ST\_FLAG\_RCV;

          else

            size\_buf = -1;

        }

        break;

        case ST\_FLAG\_RCV:

        {

          if (buf[size\_buf] == A\_CE\_AR)

            state = ST\_A\_RCV;

          else if (buf[size\_buf] != FLAG)

          {

            state = ST\_START;

            size\_buf = START\_INDEX;

          }

          else

            size\_buf = 0;

        }

        break;

        case ST\_A\_RCV:

        {

          if (buf[size\_buf] == C(r))

            state = ST\_C\_RCV;

          else if (buf[size\_buf] == FLAG)

          {

            state = ST\_FLAG\_RCV;

            size\_buf = 0;

          }

          else if(buf[size\_buf] == C((r+1)%2) )

          {

            if(r)

              sendBlock(FLAG\_DATA\_SENDING\_ANSWER\_RR\_WITH1, fd);

            else

              sendBlock(FLAG\_DATA\_SENDING\_ANSWER\_RR\_WITH0, fd);

            state=ST\_START;

            size\_buf=START\_INDEX;

            break;

          }

          else

          {

            state = ST\_START;

            size\_buf = START\_INDEX;

          }

        }

        break;

        case ST\_C\_RCV:

        {

          if (buf[size\_buf] == (A\_CE\_AR ^ C(r)))

            state = ST\_D;

          else if (buf[size\_buf] == FLAG)

          {

            state = ST\_FLAG\_RCV;

            size\_buf = 0;

          }

          else

          {

            state = ST\_START;

            size\_buf = START\_INDEX;

          }

        }

        break;

        case ST\_D:

        {

          if (buf[size\_buf] == FLAG)

            state = ST\_STOP;

        }

        break;

      }

    }

    //Byte destuffing, returns size of buffer

    size\_buffer = byteDeStuffing(buf, size\_buf);

    if(size\_buffer<1){

      end=false;

    }else{

      //Caso em que da sucesso destuffing

      //Check BCC2

      if (checkBCC2(buf, size\_buffer)==1)

      {

        end = true;

      }else{

        end=false;

      }

    }

    r = (r + 1) % 2;

    //Send acknowlegment

    if(end){

      if(r)

        sendBlock(FLAG\_DATA\_SENDING\_ANSWER\_RR\_WITH1, fd);

      else

        sendBlock(FLAG\_DATA\_SENDING\_ANSWER\_RR\_WITH0, fd);

    }

    //Deu erro BCC2 vou enviar um REJ

    else{

      if(r)

        sendBlock(FLAG\_DATA\_SENDING\_ANSWER\_REJ\_WITH1, fd);

      else

        sendBlock(FLAG\_DATA\_SENDING\_ANSWER\_REJ\_WITH0, fd);

      r = (r + 1) % 2;

    }

    size\_buf = 0;

    state = ST\_START;

  }

  memcpy(buffer,buf,size\_buffer-1);

  //Free read buf

  free(buf);

  return size\_buffer - 1;

}

DataStruct createMessage(unsigned int sequenceNumber, unsigned char \*buffer, int length)

{

  DataStruct data;

  data.flag = FLAG;

  data.fieldA=A\_CE\_AR;

  data.fieldC = C(sequenceNumber);

  data.fieldBCC1 = data.fieldA ^ data.fieldC;

  data.fieldBCC2 = (unsigned char \*)malloc(sizeof(unsigned char));

  //Algortimo de calculo de BCC2 e

  //I\_0=D0//I i+1= I i ^data i

  data.fieldBCC2[0] = buffer[0];

  for (int i = 1; i < length; i++)

  {

    data.fieldBCC2[0] ^= buffer[i];

  }

  //Max size is 2 \* length because byte stuffing doubles size

  data.fieldD = (unsigned char \*)malloc(sizeof(unsigned char)\*length\*2);

  data.bcc2StufSize = BCC2Stufying(data.fieldBCC2);

  data.dataStufSize = dataStuffing(buffer, length, data.fieldD);

  return data;

}

unsigned int BCC2Stufying(unsigned char \*BCC2)

{

  unsigned int size = 1;

  if (BCC2[0] == FLAG)

  {

    //Vai ser necessario realocar memoria

    BCC2=realloc(BCC2,2\*sizeof(unsigned char));

    BCC2[0] = ESC;

    BCC2[1] = ESC\_FLAG;

    size = 2;

  }

  else if (BCC2[0] == ESC)

  {

    //Vai ser necessario realocar memoria

    BCC2=realloc(BCC2,2\*sizeof(unsigned char));

    BCC2[0] = ESC;

    BCC2[1] = ESC\_ESC;

    size = 2;

  }

  return size;

}

unsigned int dataStuffing(unsigned char \*data, int length, unsigned char \*fieldD)

{

  unsigned int pos = 0;

  for (int i = 0; i < length; i++)

  {

    if (data[i] == FLAG)

     {

      fieldD[i + pos] = ESC;

      pos++;

      fieldD[i + pos] = ESC\_FLAG;

    }

    else if (data[i] == ESC)

    {

      fieldD[i + pos] = ESC;

      pos++;

      fieldD[i + pos] = ESC\_ESC;

    }

    else{

      fieldD[i+pos]=data[i];

    }

  }

  length += pos;

  return length;

}

int llclose(int fd, int flag)

{

  int read\_bloc\_ret = READ\_FAIL;

  n\_tries = MAX\_RETR;

  //Transmitter side

  if (flag == FLAG\_LL\_CLOSE\_TRANSMITTER\_DISC)

  {

    if (signal(SIGALRM, alarm\_handler\_disc\_signal) < 0)

    {

      perror("Erro a instalar o handler no LL\_CLOSE, no receiver");

    }

    if (sendBlock(flag,fd) != WRITE\_SUCCESS)

    {

      printf("Erro a enviar FLAG\_LL\_CLOSE\_TRANSMITTER\_DISC\n");

      return -1;

    }

    type\_handling=HANDLING\_CLOSE\_EMISSOR;

    alarm(TIMEOUT);

    while (read\_bloc\_ret == READ\_FAIL)

    {

      read\_bloc\_ret = readBlock(FLAG\_LL\_CLOSE\_TRANSMITTER, fd);

    }

    if (read\_bloc\_ret == READ\_FAIL)

    {

      printf("O llclose no transmitter dei timeout sem resposta valida\n");

      return -1;

    }

    if (signal(SIGALRM, SIG\_IGN) == SIG\_ERR)

    {

      perror("Error in ignoring SIG ALARM handler");

    }

    if (sendBlock(FLAG\_LL\_CLOSE\_TRANSMITTER\_UA,fd) != WRITE\_SUCCESS)

    {

      printf("Erro a enviar FLAG\_LL\_CLOSE\_TRANSMITTER\_UA\n");

      return -1;

    }

  }

  //Receiver block

  else if (flag == FLAG\_LL\_CLOSE\_RECEIVER\_DISC)

  {

    if (signal(SIGALRM, alarm\_handler\_disc\_signal) < 0)

    {

      perror("Erro a instalar o handler no LL\_CLOSE, no receiver");

    }

    if (readBlock(flag, fd) != READ\_SUCCESS)

    {

      printf("Erro a receber FLAG\_LL\_CLOSE\_TRANSMITTER\_DISC\n");

      return READ\_FAIL;

    }

    if (sendBlock(flag, fd) != WRITE\_SUCCESS)

    {

      printf("Erro a enviar FLAG\_LL\_CLOSE\_TRANSMITTER\_DISC\n");

      return -1;

    }

    type\_handling=HANDLING\_CLOSE\_RECETOR;

    alarm(TIMEOUT);

    while (read\_bloc\_ret == READ\_FAIL)

    {

      read\_bloc\_ret = readBlock(FLAG\_LL\_CLOSE\_RECEIVER\_UA,fd);

    }

    if (read\_bloc\_ret == READ\_FAIL)

    {

      printf("O llclose no transmitter dei timeout sem resposta valida\n");

      return -1;

    }

  }

  else if(flag==FLAG\_HANDLER\_CALL){

  }

  else

  {

    printf("ERRO EM LLCLOSE");

    return OTHER\_ERROR;

  }

  if (tcsetattr(fd, TCSANOW, &oldtio) == -1)

  {

    perror("tcsetattr");

    exit(-1);

  }

  if (close(fd) != 0)

  {

    perror("Failled to close file");

  }

  return LL\_CLOSE\_SUCESS;

}

**Makefile:**

all:

    gcc  protocolo.c  application.c  files.c  -lrt  -lm  -Wall  -g  -o  writer

    gcc  protocolo.c  application.c  files.c  -lrt  -lm  -Wall  -g  -o  reader/reader

clean:

    rm writer

    rm reader

valgrind:

    valgrind --leak-check=yes -v ./writer

    valgrind --leak-check=yes -v ./reader/reader

rerun:

    make clean all valgrind