**Faculdade de Engenharia da Universidade do Porto**



Protocolo de Ligação de Dados

RCOM: 1º Trabalho Laboratorial

**Turma 2**

Autores:

Miguel Carreira Neves [up201608657@fe.up.pt](mailto:up201608657@fe.up.pt)

José Miguel Maçães [up201806622@fe.up.pt](mailto:up201806622@fe.up.pt)

Sumário

Este trabalho consiste no desenvolvimento de um protocolo de ligação de dados baseado na matéria lecionada. Será concretizado na prática através do envio de um ficheiro entre duas máquinas utilizando uma porta série.

Os objetivos relativos à realização do trabalho foram cumpridos, tendo sido desenvolvida uma aplicação que permite o envio de um ficheiro de uma máquina para outra, pelo que o trabalho foi concluído com sucesso.

Introdução

O principal objetivo do trabalho é desenvolver e implementar um protocolo de ligação de dados, seguindo o guião disponibilizado pelos docentes, bem como testar o mesmo protocolo com recurso a uma aplicação simples que permite transferir ficheiros. O relatório servirá o propósito de complemento e apoio teórico à componente prática desenvolvida em laboratório e segue uma estrutura bem definida:

1. **Introdução:** Exposição dos objetivos do trabalho realizado;
2. **Arquitetura:** Indicação dos blocos funcionais e da interface;
3. **Estrutura do Código:** API’s, principais estruturas de dados, principais funções e a sua relação com a arquitetura;
4. **Casos de Uso Principais:** Identificação dos Casos de Uso Principais e das sequências de chamada de funções;
5. **Protocolo de Ligação Lógica:** Descrição dos principais aspetos funcionais e da estratégia de implementação dos mesmos, com recurso a extratos de código;
6. **Protocolo de Aplicação:** Semelhante ao ponto **5**, neste caso relativamente à aplicação;
7. **Validação:** Descrição dos testes efetuados e apresentação dos seus resultados
8. **Eficiência do Protocolo de Ligação de Dados:** Caracterização estatística da eficiência do protocolo, recorrendo a medidas sobre o código desenvolvido;
9. **Conclusões:** Síntese final e reflexão sobre os objetivos de aprendizagem.

Arquitetura

Neste trabalho estão presentes dois blocos funcionais, um dedicado ao emissor de dados e outro ao recetor. Em cada um dos blocos é feita uma divisão por camadas. A camada de ligação de dados é responsável pelo protocolo de ligação de dados, estabelecendo a ligação, transmitindo os dados com verificação de erros recorrendo ao reenvio de tramas com erros e ao *stuffing* e *destuffing* para essa verificação e, por fim, terminando a ligação. A camada da aplicação lerá de e escreverá no ficheiro. A nível da interface, o utilizador indica os parâmetros na chamada da aplicação.

Estrutura do Código

Relativamente ao código, este encontra-se dividido em dois ficheiros: sender.c e reciever.c, dedicados ao envio e receção, respetivamente.

No ficheiro sender.c, encontram-se as 3 funções da camada da aplicação: **llopen**, que estabelece a ligação através do envio da trama SET e da espera pela receção da trama UA, **llwrite**, que envia as tramas de controlo START e END, que indicam o início e o fim da transmissão de dados além de também enviarem informações relevantes sobre o ficheiro tais como o nome e o tamanho, bem como as tramas de dados do ficheiro a transmitir, esperando sempre confirmação de sucesso, reenviado as tramas se não for esse o caso e **llclose** que trata do envio da trama DISC, espera por uma trama DISC e, por fim, envia uma trama UA, finalizando a desconexão.

No ficheiro reciever.c, repetem-se as funções **llopen**, que neste caso espera a receção de SET e envia como resposta UA, estabelecendo a ligação e **llclose**, que espera receber DISC, envia DISC de volta e espera receber UA para terminar a ligação. Para além destas funções está também presente **llread**, que irá esperar receber as tramas de controlo, para iniciar e terminar a receção de dados e as tramas de dados em concreto, sendo que para cada uma enviará RR e irá guardar os dados nesta mesma trama para o ficheiro com o nome do ficheiro recebido antecedido de “received”, se for recebida corretamente, ou, caso contrário, REJ.

# Casos de Uso Principais

Esta aplicação terá como principais casos de uso (1) a transferência de um ficheiro (com qualquer formato, desde .gif até png, jpeg, txt, ect.) de uma máquina para outra através de uma porta série e (2) a interface da aplicação, que permite ao utilizador especificar com que parâmetros se fará a transmissão, bem como visualizar informação sobre a transferência (quais as transmissões de tramas que formam bem ou mal sucedidas).

Relativamente à transferência do ficheiro, é feita a seguinte sequência de chamadas de funções do lado do **emissor** (chamada das funções deve ser lida da esquerda para a direita visto estarem ordenadas):

- main -> llopen | llwrite | llclose

- llopen -> setConnection

- setConnection -> sendSetWithAlarm | readUA

- sendSetWithAlarm -> *write*

- readUA -> *read*

- llwrite -> sendStartOrEnd | sendFileData | sendStartOrEnd

- sendStartOrEnd -> transferData

- sendFileData -> transferData

- transferData -> sendDataWithAlarm | readDataResponse

- sendDataWithAlarm -> *write*

- readDataResponse -> *read*

- llclose -> disconnect

- disconnect -> sendDisconnectWithAlarm | readDisc | sendUA

- sendDisconnectWithAlarm -> *write*

- readDisc -> *read*

- sendUA -> *write*

Do lado do **recetor**:

-main -> llopen | llread | llclose

- llopen -> setProtocol

- setProtocol -> readSet | sendUA

- readSet -> *read*

- sendUA -> *write*

- llread -> receiveStart | dataProtocol | writeToFile

- receiveStart -> dataProtocol

- dataProtocol -> readData | sendREJ | sendRR | saveFileData

- readData -> *read*

- sendREJ -> *write*

- sendRR -> *write*

- saveFileData -> receiveEnd

- llclose -> disconnectProtocol

- disconnectProtocol -> readDisc | sendDiscWithAlarm

- readDisc -> *read*

- sendDiscWithAlarm -> *write*

Na interface da aplicação, o utilizador, na chamada do programa emissor, escolhe a porta série por onde efetuar a transferência, o ficheiro a transferir, o tamanho de cada trama e o rácio de erros em tramas (*Frame Error Ratio*). Poderá ver informação sobre os dados que está a transmitir. Na chamada do programa recetor, apenas é indicada a porta série e é possível ver informação sobre os dados a serem recebidos.

# Protocolo de Ligação Lógica

O Protocolo de Ligação Lógica tem como objetivo principal providenciar um serviço de de comunicação de dados entre dois sistemas ligados por uma porta série. Este serviço deve ser fiável, procurando também cumprir objetivos de **estabelecimento e terminação da ligação**, **confirmação positiva**, se for recebida uma trama sequencialmente correta (sendo este numero de sequencia verificado para verificar se é um duplicado) e sem erros (tanto no header do pacote como nos dados em si), **controlo de fluxo** e **controlo de erros**. Este último será atingido recorrendo a um mecanismo Stop-and-Wait, em que o emissor só envia a trama seguinte de dados após a receção de uma confirmação de receção correta da trama anterior. Neste departamento, pode surgir o caso de o emissor não receber esta confirmação durante um período de tempo e decidir reenviar a trama de dados, sendo isto implementado com recurso a temporizadores, ou o caso de o emissor receber uma confirmação negativa, sendo então reenviada a trama de dados. Estes dois casos podem resultar em transmissões de duplicados, que são detetados e eliminados.

Este protocolo é caracterizado pelo **sincronismo de trama**, organizando a transmissão em tramas, que podem ser de Informação, de Supervisão ou Não Numeradas. Estas tramas têm um cabeçalho com um formato comum, sendo que as de Informação têm um campo para transporte de dados, a ser preenchido por um pacote gerado pela aplicação, bem como um campo que permite a **numeração de trama**. É garantida a transmissão de dados independentemente das tramas estarem codificadas, ou seja, existe transparência, assegurada pela técnica de byte stuffing. Além disso, todas as tramas estão protegidas com um campo de proteção, o BCC, que é verificado em todas as transmissões. Todas as tramas têm tanto no início como no fim uma flag que as delimita sendo o início precedido do header do pacote que além de indicar o address (se é do recetor ou do emissor) contém também o control que indica que tipo de pacote é este e como deve ser interpretado.

# Protocolo de Aplicação

Os principais aspetos funcionais que caracterizam o protocolo de aplicação, do lado do emissor, são o envio do pacote de controlo START indicativo do início da transmissão de dados, a leitura do ficheiro a transmitir, a transmissão de dados em si e o envio, no fim da transmissão, do pacote de controlo END. Ambos estes pacotes START e END têm o seu conteúdo comparado e apenas no caso de ser igual a transferência de ficheiro é dada por finalizada .A transmissão dos dados envolve a divisão do ficheiro em partes mais pequenas (tramas) com tamanho definido pelo utilizador.

No caso do recetor, este recebe os pacotes de controlo e os dados, juntando estes últimos e escreve-os para o ficheiro que criou para o efeito. À medida que vai recebendo uma trama e a declara como aceite (não tendo erros nem sendo duplicada) esta é guardada sendo escrita para o fim do ficheiro com o nome recebido antecedido por “received”.

É de notar que existe uma clara segregação entre o protocolo de Aplicação e o de Ligação Lógica, tanto no emissor como no recetor, apesar de estarem no mesmo ficheiro ambas as funcoes de protocolos diferentes apenas por conveniência. A Aplicação não depende do processo nem precisa de saber como é feito todo o “heavy lifting” da Ligação Lógica, apenas ordenando através de chamadas de funcções que operações “high level” sejam realizadas (i.e. sendFileData, dataProtocol, writeToFile, setConnection, ect.).

# Validação

Foram realizados vários testes sobre a aplicação desenvolvida:

* Transmissão de diferentes ficheiros, variantes no tamanho e tipo (i.e. txt, gif, png);
* Transmissão de um ficheiro com diferentes tamanhos de tramas, desde tamanho com dois dígitos até tamanhos superiores ao do ficheiro a ser enviado;
* Transmissão de um ficheiro com diferentes taxas de erros, provocados para efeito do teste, desde 0% até valores acima dos 90%;
* Transmissão de um ficheiro com diferentes baudrates, nomeadamente os recomendados entre outros;
* Desligar a porta série durante a transmissão de um ficheiro de forma a causar a ativação do mecanismo STOP and WAIT e a retransmissão de tramas atendendo ao tempo concedido máximo para o fazer. E de seguida voltar a ligar a porta verificando que a aplicação consegue proceder como desejado.
* Simulação de um tempo de propagação desde valores quase nulos até alguns poucos segundos.

# Eficiência do Protocolo de Ligação de Dados

As várias execuções da aplicação feitas para tirar as medidas necessárias para testar a eficiência do Protocolo de Ligação de Dados foram realizadas utilizando uma porta série virtual. Mediu-se o tempo de execução do programa fazendo variar alguns parâmetros e usou-se esse tempo para calcular R = bits/tempo. Com R calculou-se S = R/C, sendo C a capacidade de ligação. Concretamente, um valor S maior corresponderá a uma transmissão mais eficiente e um valor S menor a uma transmissão menos eficiente. Foram realizadas duas execuções para obter 2 medidas de tempo, pelo que é apresentado nos gráficos a média dos dois S calculados. O tamanho base de trama foi de 100 bits.

Para melhor apresentação dos resultados, alguns dos gráficos foram desenhados com escala logarítmica e as imagens correspondentes aos gráficos encontram-se no anexo II.

## Diferentes Frame Error Ratio

Conclui-se que, como esperado, com o aumento da probabilidade de erro a transmissão será menos eficiente. Com o FER bastante alto verifica-se que a eficiência do protocolo reduz drasticamente.

## Diferentes Tempos de Propagação

Pode-se concluir que um aumento no tempo de propagação piorará exponencialmente a eficiência, ou seja, um aumento pequeno no atraso pode gerar grande perda de eficiência. Daí a extrema importância e foco na atualidade em obter tecnologias que permitam um tempo de propagação reduzido.

## Diferentes Tamanho das Tramas

Como é possível verificar no gráfico, uma trama muito pequena é pouco eficiente, mas uma trama muito grande também o será, pelo que há uma gama de tamanhos ideal.

## Diferentes Capacidades de Ligação

Com o aumento do baudrate, S diminui, pelo que se conclui que um menor baudrate é mais eficiente.

# Conclusões

O Protocolo de Ligação de Dados procura disponibilizar um serviço para a transmissão integral e fiável de dados entre dois computadores ligados por um cabo série. Foi implementado recorrendo ao uso de duas camadas, idealmente independentes uma da outra, a da ligação de dados e a da aplicação. Assim, as camadas não necessitam de estabelecer relações entre elas para esta implementação funcionar e uma alteração numa das camadas não seria sentida na outra.

Concluindo, os objetivos foram alcançados, tendo sido registado sucesso na realização prática do trabalho, bem como foi engrandecido o conhecimento teórico do tema. Consideramos que uma das maiores mais valias deste trabalho é a análise feita posterior ao código fazendo variar diferentes parâmetros como especificado na secção anterior, o que sentimos que ajuda a pôr a matéria numa perspetiva mais quantitativa sendo mais fácil verificar os diferentes impactos destes parâmetros bem como a sua importância para todo o conceito de redes de computadores.

# Anexo I - Código Fonte

## sender.c

/\*Non-Canonical Input Processing\*/

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <termios.h>

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <string.h>

#define BAUDRATE B38400

#define MODEMDEVICE "/dev/ttyS1"

#define \_POSIX\_SOURCE 1 /\* POSIX compliant source \*/

#define FALSE 0

#define TRUE 1

#define FLAG 0x7e

#define ESC 0x7d

#define STUFF 0x20

#define BYTE\_TO\_BINARY\_PATTERN "%c%c%c%c%c%c%c%c"

#define BYTE\_TO\_BINARY(byte) \

(byte & 0x80 ? '1' : '0'), \

(byte & 0x40 ? '1' : '0'), \

(byte & 0x20 ? '1' : '0'), \

(byte & 0x10 ? '1' : '0'), \

(byte & 0x08 ? '1' : '0'), \

(byte & 0x04 ? '1' : '0'), \

(byte & 0x02 ? '1' : '0'), \

(byte & 0x01 ? '1' : '0')

volatile int STOP = FALSE;

int nAlarm = 0;

int fd;

int setConnection();

void sendSetWithAlarm();

int readUA();

int frameMaxSize = 256;

int frameErrorRate = 0; //To begin error from 1 -> 100

char\* globalData;

int currentDataSize = 0;

int dataFrameNum = 0;

int resend = 0;

int sendDataWithAlarm();

int readDataResponse();

int disconnect();

int readDisc();

int sendDisconnectWithAlarm();

int sendUA();

char openBuf[26];

int llopen(int porta, int flag);

int llwrite(char\* filename);

int sendStartOrEnd(char\* filename, int start);

int numOfFrame = 0;

int sendFileData(char\* filename);

int llclose(int fd);

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

{

int c, res;

struct termios oldtio, newtio;

int i, sum = 0, speed = 0;

if ((argc < 5) ||

((strcmp("/dev/ttyS10", argv[1]) != 0) &&

(strcmp("/dev/ttyS11", argv[1]) != 0) &&

(strcmp("/dev/ttyS0", argv[1]) != 0) &&

(strcmp("/dev/ttyS1", argv[1]) != 0) &&

(strcmp("/dev/ttyS5", argv[1]) != 0)))

{

printf("Usage:\tnserial SerialPort FileName FrameMaxSize FrameErrorRate<0-100>\n\tex: nserial /dev/ttyS10 pinguim.gif 256 10\n");

exit(1);

}

char\* nameOfFile = argv[2];

frameMaxSize = atoi(argv[3]);

frameErrorRate = atoi(argv[4]);

/\*

Open serial port device for reading and writing and not as controlling tty

because we don't want to get killed if linenoise sends CTRL-C.

\*/

fd = open(argv[1], O\_RDWR | O\_NOCTTY);

if (fd < 0)

{

perror(argv[1]);

exit(-1);

}

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

{ /\* save current port settings \*/

perror("tcgetattr");

exit(-1);

}

bzero(&newtio, sizeof(newtio));

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

newtio.c\_iflag = IGNPAR;

newtio.c\_oflag = 0;

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

newtio.c\_lflag = 0;

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

newtio.c\_cc[VMIN] = 5; /\* blocking read until 5 chars 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, TCIOFLUSH);

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

{

perror("tcsetattr");

exit(-1);

}

globalData = malloc(frameMaxSize\*2);

llopen(10,0);

llwrite(nameOfFile);

llclose(fd);

printf("Closing\n");

// sleep(1);

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

{

perror("tcsetattr");

exit(-1);

}

close(fd);

return 0;

}

#pragma region ////////Connect

int setConnection()

{

nAlarm = 0;

STOP = FALSE;

printf("\nConnecting...\n");

(void)signal(SIGALRM, sendSetWithAlarm);

sendSetWithAlarm();

//Leitura da mensagem do receptor UA

while (STOP == FALSE && nAlarm < 3)

{ /\* loop for input \*/

if (readUA() == 0)

STOP = TRUE;

}

return 0;

}

int readUA()

{

char buf[1];

printf("\n%s\n", "Waiting for UA...");

int stop = 0;

int flag = 0;

int res = 0;

while (stop == 0)

{ //state machine

if(flag == 0)

res = read(fd, buf, 1);

if (buf[0] == 0b01111110)

{ //flag

int res = read(fd, buf, 1);

if (buf[0] == 0b00000001)

{ //address

int res = read(fd, buf, 1);

if (buf[0] == 0b00000111)

{ //control

int res = read(fd, buf, 1);

if (buf[0] == (0b00000001 ^ 0b00000111))

{ //bcc

int res = read(fd, buf, 1);

if (buf[0] == 0b01111110)

{ //final flag

stop = 1;

}

else

printf("Not the correct final flag\n");

}

else

printf("Not the correct bcc\n");

}

else

printf("Not the correct control\n");

}

else

printf("Not the correct address\n");

}

flag = 0;

if(buf[0] == 0b01111110) //if its a flag

flag = 1;

}

printf("UA received sucessfuly!\n");

return 0;

}

void sendSetWithAlarm() // atende alarme

{

char buf[255];

if (nAlarm < 3)

{

char flag = 0b01111110; //todas as flags teem este valor, slide 10

char address = 0b00000011; //header do emissor, slide 10

char control = 0b00000011; //SET ,slide 7

char bcc = (address ^ control); //XOR dos bytes anteriores ao mesmo

buf[4] = flag;

buf[3] = bcc;

buf[2] = control;

buf[1] = address;

buf[0] = flag;

printf("%s\n", "Sending SET...");

/\* COMMENT

for (size\_t i = 0; i < 5; i++)

{

printf(" " BYTE\_TO\_BINARY\_PATTERN, BYTE\_TO\_BINARY(buf[i]));

} \*/

write(fd, buf, 5);

printf("\nalarme # %d\n", nAlarm + 1);

nAlarm++;

}

else

{

printf("\nCan't connect\n");

exit(1);

}

alarm(3);

}

#pragma endregion

#pragma region ////////Transfer Data

int transferData(){

nAlarm = 0;

STOP = FALSE;

printf("\nTransfering Data...\n");

(void)signal(SIGALRM, sendDataWithAlarm);

sendDataWithAlarm();

//Leitura da mensagem do receptor UA

while (STOP == FALSE && nAlarm < 3)

{ /\* loop for input \*/

int res = readDataResponse();

if (res == 0) //Obteve resposta RR cm o dataFrameNum correto

STOP = TRUE;

else if (res == 1){ //Obtever REJ, resend

resend = 1;

STOP = TRUE;

}

}

if(resend == 1){

printf("Resending Data...\n");

transferData();

return 0;

}

dataFrameNum = 1- dataFrameNum;

printf("\nData Transfered with success!\n");

return 0;

}

int sendDataWithAlarm(){

char buf[frameMaxSize\*2];

if (nAlarm < 3)

{

char flag = 0b01111110; //todas as flags teem este valor, slide 10

char address = 0b00000011; //header do emissor, slide 10

char control;

if(dataFrameNum == 0) //S e N(s), slide 7

control = 0b00000000;

else

control = 0b01000000;

char bcc1 = (address ^ control); //XOR dos bytes anteriores ao mesmo

buf[3] = bcc1;

buf[2] = control;

buf[1] = address;

buf[0] = flag;

int i, n = 4;

char bcc2 = 0; //XOR dos bytes de Data

for(i = 0; i < currentDataSize; i++){

if (globalData[i] == 0b01111110 || globalData[i] == 0b01111101)

{

buf[n] = ESC;

buf[++n] = globalData[i] ^ STUFF;

n++;

}

else

{

buf[n] = globalData[i];

n++;

}

bcc2 ^= globalData[i];

}

if (bcc2 == 0b01111110 || bcc2 == 0b01111101)

{

buf[n] = ESC;

buf[++n] = bcc2 ^ STUFF;

}

else

buf[n] = bcc2;

buf[++n] = flag;

printf("%s", "Sending Data...");

int size = ++n;

/\* COMMENT

printf("\n%s%d ->", "Size: ", size);

for (size\_t i = 0; i < size; i++)

{

printf(" " BYTE\_TO\_BINARY\_PATTERN, BYTE\_TO\_BINARY(buf[i]));

} \*/

//FER - frame error rate

/\* for( int i = 4; i < size-2; i++){

int random = rand() % 10000 + 1; // 1 -> 10000

if(random <= frameErrorRate)

buf[i] = buf[i] ^ 0x0F;

} \*/

int random = rand() % 100 + 1; // 1 -> 100

if(random <= frameErrorRate)

buf[size-2] = buf[size-2] ^ 0x0F;

write(fd, buf, size);

printf("\nalarme # %d\n", nAlarm + 1);

nAlarm++;

}

else

{

printf("\nCan't transfer the data\n");

exit(1);

}

alarm(3);

}

int readDataResponse(){

char buf[1];

printf("\n%s\n", "Waiting for Response...");

int stop = 0;

int flag = 0;

int res = 0;

resend = 0;

int tempDataFrameNum = 0;

while (stop == 0)

{ //state machine

if(flag == 0)

res = read(fd, buf, 1);

if (buf[0] == 0b01111110)

{ //flag

int res = read(fd, buf, 1);

if (buf[0] == 0b00000001)

{ //address

int res = read(fd, buf, 1);

char controlRR;

char controlREJ;

tempDataFrameNum = 1-dataFrameNum;

if(tempDataFrameNum == 0){ //RR e R = dataFrameNum ,slide 7

controlRR = 0b00000101;

controlREJ = 0b10000001;

}

else{

controlRR = 0b10000101;

controlREJ = 0b00000001;

}

if (buf[0] == controlRR)

{ //control

int res = read(fd, buf, 1);

if (buf[0] == (0b00000001 ^ controlRR))

{ //bcc

int res = read(fd, buf, 1);

if (buf[0] == 0b01111110)

{ //final flag

stop = 1;

}

else

printf("Not the correct final flag\n");

}

else

printf("Not the correct bcc\n");

}

else if(buf[0] == controlREJ){

printf("Received REJ!\n");

return 1;

}

else

printf("Not the correct control\n");

}

else

printf("Not the correct address\n");

}

flag = 0;

if(buf[0] == 0b01111110) //if its a flag

flag = 1;

}

printf("Response received sucessfuly!\n");

return 0;

}

#pragma endregion

#pragma region ////////Disconnect

int disconnect()

{

//Send Disc

nAlarm = 0;

STOP = FALSE;

printf("\nDisconnecting...\n");

(void)signal(SIGALRM, sendDisconnectWithAlarm);

sendDisconnectWithAlarm();

//Leitura da mensagem do receptor Disc

while (STOP == FALSE && nAlarm < 3)

{ /\* loop for input \*/

if (readDisc() == 0)

STOP = TRUE;

}

//Send UA

sendUA();

printf("\nDisconnected sucessfully!\n");

return 0;

}

int sendDisconnectWithAlarm()

{

char buf[255];

if (nAlarm < 3)

{

char flag = 0b01111110; //todas as flags teem este valor, slide 10

char address = 0b00000011; //header do emissor, slide 10

char control = 0b00001011; //DISC ,slide 7

char bcc = (address ^ control); //XOR dos bytes anteriores ao mesmo

buf[4] = flag;

buf[3] = bcc;

buf[2] = control;

buf[1] = address;

buf[0] = flag;

printf("%s\n", "Sending DISC...");

/\* COMMENT

for (size\_t i = 0; i < 5; i++)

{

printf(" " BYTE\_TO\_BINARY\_PATTERN, BYTE\_TO\_BINARY(buf[i]));

} \*/

write(fd, buf, 5);

printf("\nalarme # %d\n", nAlarm + 1);

nAlarm++;

}

else

{

printf("\nCan't disconnect\n");

exit(1);

}

alarm(3);

}

int readDisc()

{

char buf[1];

printf("\n%s\n", "Waiting for Disc...");

int stop = 0;

int flag = 0;

int res = 0;

while (stop == 0)

{ //state machine

if(flag == 0)

res = read(fd, buf, 1);

if (buf[0] == 0b01111110)

{ //flag

int res = read(fd, buf, 1);

if (buf[0] == 0b00000001)

{ //address

int res = read(fd, buf, 1);

if (buf[0] == 0b00001011)

{ //control

int res = read(fd, buf, 1);

if (buf[0] == (0b00000001 ^ 0b00001011))

{ //bcc

int res = read(fd, buf, 1);

if (buf[0] == 0b01111110)

{ //final flag

stop = 1;

}

else

printf("Not the correct final flag\n");

}

else

printf("Not the correct bcc\n");

}

else

printf("Not the correct control\n");

}

else

printf("Not the correct address\n");

}

flag = 0;

if(buf[0] == 0b01111110) //if its a flag

flag = 1;

}

printf("Disc received sucessfuly!\n");

return 0;

}

int sendUA()

{

//Send UA back

char buf[255];

int res;

printf("\n%s\n", "Sending UA back...");

char flagUA = 0b01111110; //todas as flags teem este valor, slide 10

char addressUA = 0b00000011; //header do emissor, slide 10

char controlUA = 0b00000111; //UA ,slide 7

char bccUA = (addressUA ^ controlUA); //XOR dos bytes anteriores ao mesmo

buf[4] = flagUA;

buf[3] = bccUA;

buf[2] = controlUA;

buf[1] = addressUA;

buf[0] = flagUA;

/\* COMMENT

for (size\_t i = 0; i < 5; i++)

{

printf(" " BYTE\_TO\_BINARY\_PATTERN, BYTE\_TO\_BINARY(buf[i]));

} \*/

res = write(fd, buf, 5);

printf("%s\n", "\nUA sent!");

return 0;

}

#pragma endregion

#pragma region //////APP

int llopen(int porta, int flag){ //TO DO utilizar o port e a flag

printf("\nllopen\n");

//Establish Logic connection

printf("New termios structure set\n");

setConnection();

printf("\nConnection SET!\n");

}

int llwrite(char\* filename){

sendStartOrEnd(filename,1);

sendFileData(filename);

sendStartOrEnd(filename,0);

}

int sendStartOrEnd(char\* filename, int start){

memset(openBuf, 0, 26);

char c; //Slide 23

if(start == 1){

printf("\nSending Start...\n");

c = 2;

} else {

printf("\nSending End...\n");

c = 3;

}

int sizeName = strlen(filename);

char\* str = "./";

char nameDest[sizeName+2];

strcpy(nameDest,str);

strcat(nameDest,filename);

struct stat st;

if(stat(nameDest,&st) == -1){

printf("\nError reading the file\n");

return -1;

}

off\_t fileSize = st.st\_size;

char t1 = 0b00000000; //tamanho ficheiro

char l1 = 0b00000100; //tamanho v

int v1 = (int)(fileSize);

char t2 = 0b00000001; //nome

char l2 = 0b00001011;

char v2[sizeName];

strcpy(v2,filename);

char t3 = 0b00000010;

char l3 = 0b00000100;

int v3 = frameMaxSize;

openBuf[0] = c;

openBuf[1] = t1;

openBuf[2] = l1;

int x = (v1) & 0xFF;

openBuf[3] = (v1 >> 24) & 0xFF;

openBuf[4] = (v1 >> 16) & 0xFF;

openBuf[5] = (v1 >> 8) & 0xFF;

openBuf[6] = (v1) & 0xFF;

openBuf[7] = t2;

openBuf[8] = l2;

openBuf[9] = v2[0];

openBuf[10] = v2[1];

openBuf[11] = v2[2];

openBuf[12] = v2[3];

openBuf[13] = v2[4];

openBuf[14] = v2[5];

openBuf[15] = v2[6];

openBuf[16] = v2[7];

openBuf[17] = v2[8];

openBuf[18] = v2[9];

openBuf[19] = v2[10];

openBuf[20] = t3;

openBuf[21] = l3;

openBuf[22] = (v3 >> 24) & 0xFF;

openBuf[23] = (v3 >> 16) & 0xFF;

openBuf[24] = (v3 >> 8) & 0xFF;

openBuf[25] = (v3) & 0xFF;

currentDataSize = 26;

memset(globalData, 0, frameMaxSize\*2);

memcpy(globalData, openBuf, currentDataSize);

transferData();

if(start == 1)

printf("\nStarted File Data Communication\n");

else

printf("\nEnded File Data Communication\n");

return 0;

}

int sendFileData(char\* filename){

printf("\n\n\nFILE\n\n\n");

FILE \*fileptr;

char \*buffer;

long filelen;

fileptr = fopen(filename, "rb"); // Open the file in binary mode

fseek(fileptr, 0, SEEK\_END); // Jump to the end of the file

filelen = ftell(fileptr); // Get the current byte offset in the file

rewind(fileptr); // Jump back to the beginning of the file

buffer = (char \*)malloc(filelen \* sizeof(char)); // Enough memory for the file

fread(buffer, filelen, 1, fileptr); // Read in the entire file

fclose(fileptr); // Close the file

printf("File len is : %d\n",filelen);

int numFramesToSend = filelen / frameMaxSize;

int bytesLeft = filelen - (numFramesToSend\*frameMaxSize);

if(bytesLeft == 0)

printf("Num of frames to send is %d\n", numFramesToSend);

else

printf("Num of frames to send is %d\n", numFramesToSend+1);

char c = 1;

char n = 0;

unsigned char l2;

unsigned char l1;

if(frameMaxSize > 256){

l2 = frameMaxSize / 256;

l1 = frameMaxSize - (l2\*256);

} else {

l2 = 0;

l1 = frameMaxSize;

} //TO DO

currentDataSize = frameMaxSize;

int j = 0;

for(int i = 0; i < numFramesToSend; i++){

memset(globalData, 0, frameMaxSize\*2);

memcpy(globalData, buffer + j, currentDataSize);

transferData();

j += currentDataSize;

numOfFrame++;

printf("Num of frame %d\n",numOfFrame);

}

if(bytesLeft>0){

currentDataSize = bytesLeft;

memset(globalData, 0, frameMaxSize\*2);

memcpy(globalData, buffer + j, bytesLeft);

transferData();

numOfFrame++;

printf("Num of frame %d\n",numOfFrame);

}

}

int llclose(int fd){

disconnect();

}

#pragma endregion

## reciever.c

/\*Non-Canonical Input Processing\*/

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <termios.h>

#include <stdio.h>

#include <signal.h>

#define BAUDRATE B38400

#define \_POSIX\_SOURCE 1 /\* POSIX compliant source \*/

#define FALSE 0

#define TRUE 1

#define FLAG 0x7e

#define ESC 0x7d

#define STUFF 0x20

#define BYTE\_TO\_BINARY\_PATTERN "%c%c%c%c%c%c%c%c"

#define BYTE\_TO\_BINARY(byte) \

(byte & 0x80 ? '1' : '0'), \

(byte & 0x40 ? '1' : '0'), \

(byte & 0x20 ? '1' : '0'), \

(byte & 0x10 ? '1' : '0'), \

(byte & 0x08 ? '1' : '0'), \

(byte & 0x04 ? '1' : '0'), \

(byte & 0x02 ? '1' : '0'), \

(byte & 0x01 ? '1' : '0')

volatile int STOP = FALSE;

int setProtocol(int fd);

int readSet(int fd);

int sendUA(int fd);

int maxFrameSize = 256; //default value, only for receiving start

char \*data;

int dataFrameNum = 0;

int duplicate = 0;

int isStart = 0;

int dataProtocol(int fd);

int readData(int fd);

int sendRR(int fd);

int sendREJ(int fd);

int disconnectProtocol(int fd);

int readDisc(int fd);

int sendDiscWithAlarm(int fd);

int readUA(int fd);

int llopen(int porta,int flag);

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

char\* fileData;

int fileSize;

int cntFileBytesRead = 0;

int ignore = 0;

int numOfFrame = 0;

int saveFileData(int fd);

int writeToFile();

int llclose(int porta);

char\* start;

char\* fileName;

int startSize;

int recieveStart(int fd);

int recieveEnd(int fd);

int nAlarm = 0;

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

{

int fd, c, res;

struct termios oldtio, newtio;

if ((argc < 2) ||

((strcmp("/dev/ttyS10", argv[1]) != 0) &&

(strcmp("/dev/ttyS11", argv[1]) != 0) &&

(strcmp("/dev/ttyS0", argv[1]) != 0) &&

(strcmp("/dev/ttyS1", argv[1]) != 0) &&

(strcmp("/dev/ttyS5", argv[1]) != 0)))

{

printf("Usage:\tnserial SerialPort\n\tex: nserial /dev/ttyS11\n");

exit(1);

}

/\*

Open serial port device for reading and writing and not as controlling tty

because we don't want to get killed if linenoise sends CTRL-C.

\*/

fd = open(argv[1], O\_RDWR | O\_NOCTTY);

if (fd < 0)

{

perror(argv[1]);

exit(-1);

}

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

{ /\* save current port settings \*/

perror("tcgetattr");

exit(-1);

}

bzero(&newtio, sizeof(newtio));

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

newtio.c\_iflag = IGNPAR;

newtio.c\_oflag = 0;

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

newtio.c\_lflag = 0;

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

newtio.c\_cc[VMIN] = 5; /\* blocking read until 5 chars 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, TCIOFLUSH);

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

{

perror("tcsetattr");

exit(-1);

}

printf("%s\n", "New Termios Structure set");

llopen(fd,1);

llread(fd,data);

llclose(fd);

//sleep(2);

tcsetattr(fd, TCSANOW, &oldtio);

close(fd);

return 0;

}

#pragma region ///////SET

int setProtocol(int fd)

{

//Read Input SET

STOP = FALSE;

while (STOP == FALSE)

{ /\* loop for input \*/

if (readSet(fd) == 0)

STOP = TRUE;

}

sendUA(fd);

return 0;

}

int sendUA(int fd)

{

//Send UA back

char buf[255];

int res;

printf("\n%s\n", "Sending UA back...");

char flagUA = 0b01111110; //todas as flags teem este valor, slide 10

char addressUA = 0b00000001; //header do emissor, slide 10

char controlUA = 0b00000111; //UA ,slide 7

char bccUA = (addressUA ^ controlUA); //XOR dos bytes anteriores ao mesmo

buf[4] = flagUA;

buf[3] = bccUA;

buf[2] = controlUA;

buf[1] = addressUA;

buf[0] = flagUA;

/\* COMMENT

for (size\_t i = 0; i < 5; i++)

{

printf(" " BYTE\_TO\_BINARY\_PATTERN, BYTE\_TO\_BINARY(buf[i]));

} \*/

res = write(fd, buf, 5);

printf("%s\n", "\nUA sent!");

return 0;

}

int readSet(int fd)

{

char buf[1];

printf("\n%s\n", "Waiting for SET...");

int stop = 0;

int flag = 0;

int res = 0;

while (stop == 0)

{ //state machine

if(flag == 0)

res = read(fd, buf, 1);

if (buf[0] == 0b01111110)

{ //flag

int res = read(fd, buf, 1);

if (buf[0] == 0b00000011)

{ //address

int res = read(fd, buf, 1);

if (buf[0] == 0b00000011)

{ //control

int res = read(fd, buf, 1);

if (buf[0] == (0b00000011 ^ 0b00000011))

{ //bcc

int res = read(fd, buf, 1);

if (buf[0] == 0b01111110)

{ //final flag

stop = 1;

}

else

printf("Not the correct final flag\n");

}

else

printf("Not the correct bcc\n");

}

else

printf("Not the correct control\n");

}

else

printf("Not the correct address\n");

}

flag = 0;

if(buf[0] == 0b01111110) //if its a flag

flag = 1;

}

printf("SET received sucessfuly!\n");

return 0;

}

#pragma endregion

#pragma region ///////DATA

int dataProtocol(int fd){

//Read Data and then answer

STOP = FALSE;

while (STOP == FALSE)

{ /\* loop for input \*/

int res = readData(fd);

if (res == -1) // has received Disc

STOP = TRUE;

else if (res == 1) // bcc2 detected errors

sendREJ(fd);

else{

sendRR(fd);

if(isStart == 1)

STOP = TRUE;

if(duplicate == 0 && isStart == 0) //discard data if duplicate

saveFileData(fd);

}

}

return 0;

}

int readData(int fd){

memset(data, 0, maxFrameSize\*2);

char tmpData[maxFrameSize\*2];

char buf[1];

printf("\n%s\n", "Waiting for Data...");

int flag = 0;

int stop = 0;

int res = 0;

int control;

char bcc2;

duplicate = 0;

while (stop == 0)

{ //state machine

if(flag == 0)

res = read(fd, buf, 1);

if (buf[0] == 0b01111110)

{ //flag

int res = read(fd, buf, 1);

if (buf[0] == 0b00000011)

{ //address

int res = read(fd, buf, 1);

if(dataFrameNum == 0){ //S e N(s), slide 7

control = 0b00000000;

if(buf[0] == 0b01000000)

duplicate = 1;

}

else{

control = 0b01000000;

if(buf[0] == 0b00000000)

duplicate = 1;

}

if (buf[0] == control)

{ //control

int res = read(fd, buf, 1);

if (buf[0] == (0b00000011 ^ control))

{ //bcc1

//DATA

int receiving = 0;

int i = 0;

int res = read(fd, buf, 1);

bcc2 = buf[0];

while(receiving == 0){

int res = read(fd, buf, 1);

if (buf[0] == 0b01111110){ //final flag

receiving = 1;

stop = 1;

} else {

tmpData[i] = bcc2;

bcc2 = buf[0];

/\* COMMENT

printf(" " BYTE\_TO\_BINARY\_PATTERN, BYTE\_TO\_BINARY(tmpData[i])); \*/

}

i++;

}

char stuffflag = STUFF ^ FLAG;

char stuffesc = STUFF ^ ESC;

if (tmpData[i-2] == ESC)

{

i--;

if(bcc2 == stuffflag)

{

bcc2 = FLAG;

}

else if (bcc2 == stuffesc)

{

bcc2 = ESC;

}

}

int xor = 0, n = 0, j;

for(j = 0; j < (i-1); j++)

{

if (tmpData[j] == ESC)

{

j++;

if(tmpData[j] == stuffflag)

{

data[n] = FLAG;

}

else if (tmpData[j] == stuffesc)

{

data[n] = ESC;

}

}

else

data[n] = tmpData[j];

xor ^= data[n];

n++;

}

/\* COMMENT

printf("\nData after Destuffing:\n");

for(int w = 0; w < n; w++)

printf(" " BYTE\_TO\_BINARY\_PATTERN, BYTE\_TO\_BINARY(data[w])); \*/

if(bcc2 != xor){

printf("BCC2 shows errors in data fields!\n");

return 1;

}

}

else

printf("Not the correct bcc1\n");

}

else if(duplicate == 1){

dataFrameNum = 1-dataFrameNum; //para pedir de novo a trama seguinte visto que esta era duplicate

printf("Data was a Duplicate\n");

return 0;

}

else

printf("Not the correct control\n");

}

else

printf("Not the correct address\n");

}

flag = 0;

if(buf[0] == 0b01111110) //if its a flag

flag = 1;

}

printf("\n\*\*Data received sucessfuly!\*\*\n");

return 0;

}

int sendRR(int fd){

//Send RR back

char buf[255];

int res;

printf("\n%s\n", "Sending RR back...");

char flagRR = 0b01111110; //todas as flags teem este valor, slide 10

char addressRR = 0b00000001; //header do emissor, slide 10

char controlRR;

dataFrameNum = 1-dataFrameNum;

if(dataFrameNum == 0) //RR e R = dataFrameNum ,slide 7

controlRR = 0b00000101;

else

controlRR = 0b10000101;

char bccRR = (addressRR ^ controlRR); //XOR dos bytes anteriores ao mesmo

buf[4] = flagRR;

buf[3] = bccRR;

buf[2] = controlRR;

buf[1] = addressRR;

buf[0] = flagRR;

/\* COMMENT

for (size\_t i = 0; i < 5; i++)

{

printf(" " BYTE\_TO\_BINARY\_PATTERN, BYTE\_TO\_BINARY(buf[i]));

} \*/

res = write(fd, buf, 5);

printf("\nRR with R = %d sent!\n", dataFrameNum);

return 0;

}

int sendREJ(int fd){

//Send REJ back

char buf[255];

int res;

printf("\n%s\n", "Sending REJ back...");

char flagREJ = 0b01111110; //todas as flags teem este valor, slide 10

char addressREJ = 0b00000001; //header do emissor, slide 10

char controlREJ;

if(dataFrameNum == 0) //REJ e R = dataFrameNum ,slide 7

controlREJ = 0b00000001;

else

controlREJ = 0b10000001;

char bccREJ = (addressREJ ^ controlREJ); //XOR dos bytes anteriores ao mesmo

buf[4] = flagREJ;

buf[3] = bccREJ;

buf[2] = controlREJ;

buf[1] = addressREJ;

buf[0] = flagREJ;

/\* COMMENT

for (size\_t i = 0; i < 5; i++)

{

printf(" " BYTE\_TO\_BINARY\_PATTERN, BYTE\_TO\_BINARY(buf[i]));

} \*/

res = write(fd, buf, 5);

printf("\nREJ with R = %d sent!\n", dataFrameNum);

return 0;

}

#pragma endregion

#pragma region ///////DISC

int disconnectProtocol(int fd)

{

//Read Disc

STOP = FALSE;

while (STOP == FALSE)

{ /\* loop for input \*/

if (readDisc(fd) == 0)

STOP = TRUE;

}

//Send Disc

nAlarm = 0;

STOP = FALSE;

printf("\nDisconecting...\n");

(void)signal(SIGALRM, sendDiscWithAlarm);

sendDiscWithAlarm(fd);

//Read UA

while (STOP == FALSE && nAlarm < 3)

{ /\* loop for input \*/

if (readUA(fd) == 0)

STOP = TRUE;

}

printf("\nDisconnected Sucessfully!\n");

return 0;

}

int readDisc(int fd)

{

char buf[1];

printf("\n%s\n", "Waiting for DISC...");

int stop = 0;

int flag = 0;

int res = 0;

while (stop == 0)

{ //state machine

if(flag == 0)

res = read(fd, buf, 1);

if (buf[0] == 0b01111110)

{ //flag

int res = read(fd, buf, 1);

if (buf[0] == 0b00000011)

{ //address

int res = read(fd, buf, 1);

if (buf[0] == 0b00001011)

{ //control

int res = read(fd, buf, 1);

if (buf[0] == (0b00000011 ^ 0b00001011))

{ //bcc

int res = read(fd, buf, 1);

if (buf[0] == 0b01111110)

{ //final flag

stop = 1;

}

else

printf("Not the correct final flag\n");

}

else

printf("Not the correct bcc\n");

}

else

printf("Not the correct control\n");

}

else

printf("Not the correct address\n");

}

flag = 0;

if(buf[0] == 0b01111110) //if its a flag

flag = 1;

}

printf("DISC received sucessfuly!\n");

return 0;

}

int sendDiscWithAlarm(int fd)

{

char buf[255];

if (nAlarm < 3)

{

//Send Disc back

int res;

printf("\n%s\n", "Sending DISC back...");

char flag = 0b01111110; //todas as flags teem este valor, slide 10

char address = 0b00000001; //header do emissor, slide 10

char control = 0b00001011; //UA ,slide 7

char bcc = (address ^ control); //XOR dos bytes anteriores ao mesmo

buf[4] = flag;

buf[3] = bcc;

buf[2] = control;

buf[1] = address;

buf[0] = flag;

/\* COMMENT

for (size\_t i = 0; i < 5; i++)

{

printf(" " BYTE\_TO\_BINARY\_PATTERN, BYTE\_TO\_BINARY(buf[i]));

} \*/

res = write(fd, buf, 5);

printf("%s\n", "\nDISC sent!");

printf("\nalarme # %d\n", nAlarm + 1);

nAlarm++;

}

else

{

printf("\nCan't connect\n");

exit(1);

}

alarm(3);

}

int readUA(int fd)

{

char buf[1];

printf("\n%s\n", "Waiting for UA...");

int stop = 0;

int flag = 0;

int res = 0;

while (stop == 0)

{ //state machine

if(flag == 0)

res = read(fd, buf, 1);

if (buf[0] == 0b01111110)

{ //flag

int res = read(fd, buf, 1);

if (buf[0] == 0b00000011)

{ //address

int res = read(fd, buf, 1);

if (buf[0] == 0b00000111)

{ //control

int res = read(fd, buf, 1);

if (buf[0] == (0b00000011 ^ 0b00000111))

{ //bcc

int res = read(fd, buf, 1);

if (buf[0] == 0b01111110)

{ //final flag

stop = 1;

}

else

printf("Not the correct final flag\n");

}

else

printf("Not the correct bcc\n");

}

else

printf("Not the correct control\n");

}

else

printf("Not the correct address\n");

}

flag = 0;

if(buf[0] == 0b01111110) //if its a flag

flag = 1;

}

printf("UA received sucessfuly!\n");

return 0;

}

#pragma endregion

#pragma region //////APP

int llopen(int porta,int flag){

setProtocol(porta);

}

int llread(int fd, char\* buffer)

{

data = malloc(maxFrameSize\*2);

recieveStart(fd);

data = malloc(maxFrameSize\*2);

fileData = malloc(fileSize);

dataProtocol(fd);

writeToFile();

}

int saveFileData(int fd){ //TO DO receber tudo em pacotes de aplicacoa

printf("Trying to save\n");

int numOfFrames = fileSize / maxFrameSize;

int bytesLeft = fileSize - (numOfFrames\*maxFrameSize);

int bytes = 0;

if(cntFileBytesRead < fileSize){

if(cntFileBytesRead != (fileSize-bytesLeft)){

memcpy(fileData + cntFileBytesRead, data, maxFrameSize);

cntFileBytesRead += maxFrameSize;

bytes = maxFrameSize;

} else {

memcpy(fileData + cntFileBytesRead, data, bytesLeft);

cntFileBytesRead += bytesLeft;

bytes = bytesLeft;

}

numOfFrame++;

printf("SAVED DATA -> %d | Total Bytes read %d | Num of frame %d\n", bytes, cntFileBytesRead, numOfFrame);

} else{

printf("Data ignored %d\n", cntFileBytesRead);

if(recieveEnd(fd) == 0)

STOP = TRUE;

}

}

int writeToFile(){

printf("Writing file data to File:");

int sizeName = strlen(fileName);

char\* str = "./received";

char nameDest[sizeName+2];

strcpy(nameDest,str);

strcat(nameDest,fileName);

printf(" %s\n", nameDest);

FILE \*fp;

fp = fopen(nameDest, "w+");

fwrite(fileData,1,fileSize,fp);

fclose(fp);

}

int recieveStart(int fd)

{

printf("\nRecieving START...\n");

isStart = 1;

dataProtocol(fd);

char c = data[0];

if(c != 2){

printf("Not a start frame");

return 1;

}

//FILE SIZE

char \*v1;

int t1 = data[1]; //type

int l1 = data[2]; //length

startSize = 3;

v1 = malloc(l1);

for (int i = 0; i < l1; i++)

{

v1[i] = data[startSize]; //value

startSize++;

}

int aux0;

if(v1[0] < 0)

aux0 = (v1[0] & 0xFFFF) ^ 0xFF00;

else

aux0 = v1[0];

int aux1;

if(v1[1] < 0)

aux1 = (v1[1] & 0xFFFF) ^ 0xFF00;

else

aux1 = v1[1];

int aux2;

if(v1[2] < 0)

aux2 = (v1[2] & 0xFFFF) ^ 0xFF00;

else

aux2 = v1[2];

int aux3;

if(v1[3] < 0)

aux3 = (v1[3] & 0xFFFF) ^ 0xFF00;

else

aux3 = v1[3];

fileSize = (aux0 << 24) | (aux1 << 16) | (aux2 << 8) | (aux3);

printf("File size is %d\n",fileSize);

//FILE NAME

int t2 = data[startSize];

int l2 = data[++startSize];

startSize++;

fileName = malloc(l2);

for (int i = 0; i < l2; i++)

{

fileName[i] = data[startSize]; //value

startSize++;

}

printf("Name is %s\n", fileName);

//MaxFrameSize

char \*v3;

int t3 = data[startSize]; //type

int l3 = data[++startSize]; //length

startSize++;

v3 = malloc(l3);

for (int i = 0; i < l3; i++)

{

v3[i] = data[startSize]; //value

startSize++;

}

if(v3[0] < 0)

aux0 = (v3[0] & 0xFFFF) ^ 0xFF00;

else

aux0 = v3[0];

if(v3[1] < 0)

aux1 = (v3[1] & 0xFFFF) ^ 0xFF00;

else

aux1 = v3[1];

if(v3[2] < 0)

aux2 = (v3[2] & 0xFFFF) ^ 0xFF00;

else

aux2 = v3[2];

if(v3[3] < 0)

aux3 = (v3[3] & 0xFFFF) ^ 0xFF00;

else

aux3 = v3[3];

maxFrameSize = (aux0 << 24) | (aux1 << 16) | (aux2 << 8) | (aux3);

printf("Max frame size is %d\n",maxFrameSize);

printf("\nSTART Recieved!\n");

start = malloc(startSize);

strcpy(start, data);

isStart = 0;

return 0;

}

int recieveEnd(int fd)

{

printf("\nRecieving End...\n");

char c = data[0];

if(c != 3){

printf("Not a end frame");

return 1;

}

if (strncmp(start+1, data+1, startSize-1) != 0)

{

printf("\nEND different from START\n");

return 1;

}

printf("\nEND Recieved and value is correct!\n");

return 0;

}

int llclose(int porta){

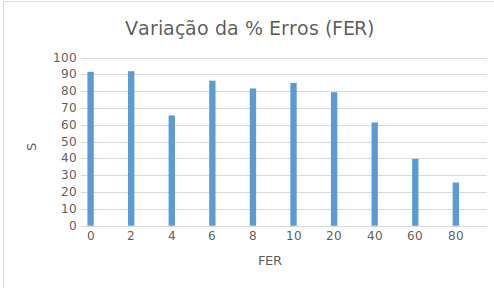
disconnectProtocol(porta);

}

#pragma endregion

# Anexo II - Gráficos

## Variação de Frame Error Ratio

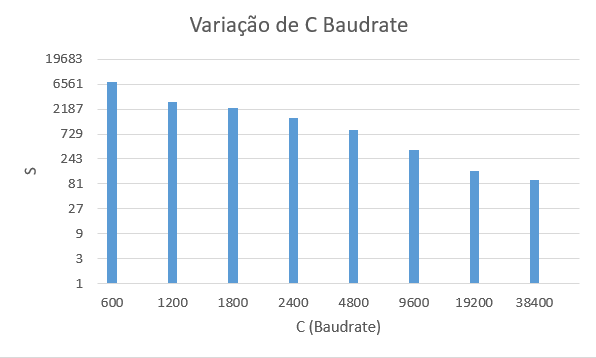


## 

## Variação do Tamanho das Tramas

## 

## Variação da Capacidade de Ligação (Baudrate)



## Variação do Tempo de Propagação

## 