Projeto 1 - Protocolo de Ligação de Dados

Licenciatura em Engenharia Informática e Computação
Redes de Computadores

Turma 9, Grupo 8:

Maria Leonor Monteiro Beirão | 201806798

José Pedro Nogueira Rodrigues | 201708806

22 de Dezembro de 2021

Sumário

O objetivo deste projeto é desenvolver um protocolo de ligação de dados que permita transferir um ficheiro de um computador para outro através da porta de série.

O trabalho foi concluído satisfatoriamente, sendo possível na maioria dos casos transmitir o ficheiro com sucesso. No entanto, alguns problemas podem ser encontrados quando a ligação é interrompida inesperadamente.

1. Introdução

O objetivo deste trabalho é desenvolver um protocolo de transferência de dados entre dois computadores recorrendo a uma porta de série. Sendo a porta de série um dos mecanismos mais básicos de ligação, este projeto permitirá captar o essencial que está por detrás de uma transferência de dados. Para atingir o objetivo proposto, foi necessário recorrer a protocolos de ligação e regras de encapsulamento de dados tal como algoritmos de captação e correção de erros para permitir uma transferência final com sucesso independentemente da ocorrência de falhas na conexão. Este relatório segue portanto a seguinte estrutura:

- **Introdução:** Indicação dos objetivos do trabalho e do relatório; Descrição do tipo de informação a ser encontrado em cada uma das secções seguintes.
- **Arquitetura:** Explicação do funcionamento do programa e interfaces do utilizador.
- **Estrutura do código:** Principais estruturas de dados, principais funções e a sua relação com a arquitetura.
- **Casos de uso principais:** Identificação dos casos de uso principais; Identificação das sequências de chamada de funções.
- **Protocolo de ligação lógica:** Identificação dos principais aspetos funcionais da ligação lógica e da sua estratégia de implementação.
- **Protocolo de aplicação:** Identificação dos principais aspetos funcionais da aplicação e da sua estratégia de implementação.
- **Validação:** Descrição dos testes efetuados e dos resultados obtidos.
- **Eficiência do protocolo de ligação de dados:** Caracterização estatística da eficiência do protocolo.
- **Conclusões:** Considerações finais sobre o projeto desenvolvido.
- **Anexos:** Porções do código para análise.

2. Arquitetura

Este projeto baseia-se em três fases principais, uma primeira de estabelecimento de ligação, uma segunda e mais importante em que se começa por transmitir a informação sobre o ficheiro que vai ser transmitido, sendo de seguida transmitido o próprio ficheiro, e uma terceira final de fecho de ligação e terminação de programa.

Está também dividido em duas camadas, uma primeira de alto nível com uma máquina de controlo de estados que vai seguindo um processo lógico de início de ligação, execução da transferência do ficheiro e fecho da ligação. A segunda camada de baixo nível onde se constrói e desconstrói as tramas de dados ou de informação a serem transmitidas/recebidas, utilizando um protocolo de ligação de dados, tal como mecanismos de controlo de erros como o *byte stuffing*.

A nível de interface, o utilizador apenas tem de chamar o programa desejado, fornecendo o port onde executar a ligação e, no caso do emissor, o nome do ficheiro a ser transferido. Em caso de interrupção da ligação, mensagens de aviso ou, eventualmente, de erro são imprimidas na consola.

3. Estrutura do código

As principais funções relacionadas com a camada de alto nível são as seguintes:

```
int llopen(int fd, int flag);

int llread(int fd, unsigned char *buf);

int llwrite(int fd, unsigned char *buf, int length);

int llclose(int fd);
```

A função *llopen* trata de estabelecer a ligação entre os dois computadores, preparando as configurações corretas do canal de ligação e certificando-se de que a comunicação está a funcionar e que ambos os lados estão prontos para transmitir e receber dados.

As funções *llread* e *llwrite* são as funções que se seguem, executadas pelo **recetor** e pelo **emissor**, respetivamente, e tratam do objetivo principal deste programa, a transferência do ficheiro de um computador para o outro. O **emissor** divide o ficheiro escolhido em tramas para envio, e vai enviando-as uma a uma ao **recetor**, que à medida que as recebe, vai confirmando o sucesso ou insucesso da transferência. No caso de sucesso, faz o *destuffying* dos bytes e coloca-os no ficheiro que está a ser gerado do seu lado. No caso de insucesso, informa o **emissor** dessa situação, sendo a trama em questão reenviada.

Por fim, a função *llclose* trata do fecho da ligação, confirmando que ambos os lados estão de acordo com o fim da transferência, e retornando o canal de ligação às suas configurações originais.

As principais funções relacionadas com a camada de baixo nível são as seguintes:

```
unsigned char *create_trama_S(unsigned char SENDER, unsigned char C, unsigned
char SEND, int trama_num);

int send_trama_S(int fd, unsigned char SENDER, unsigned char C, unsigned char
SEND, int trama_num);

unsigned char *create_trama_I(unsigned char type, unsigned char SEND);

int send_trama_I(int fd, unsigned char *buf, int length);

unsigned char *createControlPacket(unsigned char type);

unsigned char *createDataPacket();
```

As funções *create_trama_S* e *send_trama_S* são responsáveis pela criação e envio de tramas de supervisão, respetivamente.

As funções *create_trama_I* e *send_trama_I* são responsáveis pela criação e envio, respetivamente, das tramas de informação, tanto de controlo *Start* (que marca o início da transferência de dados) e *End* (que marca o fim da transferência de dados), como as de pacotes de dados.

A função *createControlPacket* gera um pacote de controlo *Start* ou *End*, enquanto que a função *createDataPacket* gera o próximo pacote de dados a ser enviado.

Em termos de estruturas de dados, não foi utilizada uma estrutura para o registo das informações do ficheiro, sendo essas informações apenas guardadas em variáveis globais:

```
FILE *file;

unsigned char *fileName;

off_t fileSize, lastTramaSize;
```

4. Casos de uso principais

O caso de uso principal deste programa é o da transferência, de um computador para outro, de um ficheiro à escolha do utilizador. Para tal acontecer, os seguintes passos devem ocorrer:

- Escolha do ficheiro a enviar;
- Configuração da ligação;
- Verificação da existência do ficheiro e guardar em memória a sua informação;
- Estabelecimento da ligação com o outro computador;
- Transferência de dados pelo **emissor** e receção e escrita desses dados para um ficheiro de output pelo **recetor**, simultaneamente;
- Fecho do ficheiro aberto;
- Terminação da ligação e restauro das suas configurações originais.

5. Protocolo de ligação lógica

O objetivo do protocolo de ligação lógica é fornecer um serviço de comunicação de dados fiável entre dois sistemas ligados por um meio (canal) de transmissão, que neste caso é um cabo série.

5.1. Principais aspetos funcionais

A camada de ligação de dados é responsável por:

- Alterar as configurações da ligação pela porta série para a transferência ocorrer com sucesso, ao mesmo tempo que guarda as definições prévias para mais tarde repor;
- Estabelecer e mais tarde terminar a ligação que unirá ambos os computadores pela porta série;
- Sincronismo (delimitação) dos dados organizados em tramas, delimitadas por flags que indicam o início e fim de cada trama, e controlo de transparência dos dados através de *byte stuffing* e *destuffing*;

- Controlo de erros, utilizando temporizadores e efetuando retransmissões onde necessário, decidido através de mensagens RR ou REJ.

5.2. Implementação destes aspetos

A função *llopen()* é a responsável por configurar a porta série de acordo com as definições desejadas, guardando também as definições prévias numa estrutura *termios*, para mais tarde restaurar. A função vai de seguida inicializar a conexão enviando um comando **SET** e aguardar por uma resposta **UA**, para garantir que a conexão foi estabelecida com sucesso antes de avançar. Se tal não for possível. No caso de obter uma resposta inválida, a função tenta de novo, e no caso de não obter resposta passado 3 segundos (tempo de timeout customizável pelo utilizador), o programa lança um alarme, repetindo o comando, até um total máximo de 3 vezes (valor também customizável). Se o número máximo de repetições for atingido, o programa lança um erro e termina.

Referência ao código #1

A função *llwrite()*, que recebe como parâmetro quer um pacote de dados, já pré construídos pela função *create_trama_I*, é responsável por o enviar para o **recetor**, utilizando a função *send_trama_I* para o efeito, ficando depois o programa do **emissor** a aguardar uma resposta. Caso a resposta seja positiva, ou seja, **RR**, o programa avança para a próxima trama, incrementado o número da trama para o próximo envio. Caso a resposta seja negativa, ou seja, **REJ**, o programa repete o envio da trama atual, de modo a tentar que seja captada com sucesso desta vez. Por último, caso uma situação de timeout seja atingida, a trama é reenviada até um total de 3 vezes, caso final em que o programa lança um erro e termina.

Referência ao código #2

A função *llread()* é a responsável para processar a trama de informação enviada pelo **transmissor**, utilizando uma máquina de estados para o efeito. Começa primeiro por analisar se a trama é uma trama de controlo ou uma trama de dados, registando-o na variável *packetIsData*. Caso seja uma trama de controlo de início, o nome e o tamanho do ficheiro são processados e o ficheiro é criado, e o número da última trama necessária para completar a transferência é calculado, para mais tarde ser utilizado para verificar o fim de transmissão com sucesso. Caso seja uma trama de controlo de fim, é verificado se o nome do ficheiro e o seu tamanho estão corretos. No caso de ser uma trama de dados, a sua leitura será feita, e no caso de algum erro uma resposta **REJ** é enviada, caso contrário, envia um **RR** e termina com sucesso.

Referência ao código #3

A função *llclose()* é responsável por terminar a ligação com sucesso e repô-la às definições originais. Começa pelo **emissor** enviar um **DISC**, esperando pela resposta de um **DISC** do **recetor**, terminando com o envio de um **UA**, situação final em que as definições antigas da ligação são respostas, e o programa termina com sucesso. Esta função, tal como a *llopen()*, também se encontra protegida pelo mesmo mecanismo de reenvio e timeouts.

Referência ao código #4

---

6. Protocolo de aplicação

6.1. Principais aspetos funcionais

Na função *main* do **Emissor** é feito o parse dos argumentos da linha de comandos e, em caso de erro, o programa faz print de uma mensagem com o exemplo da utilização correta do programa.

O programa executa a função *llopen()* para estabelecer a ligação entre emissor e recetor. É chamada a função *registerFileData()* que abre o ficheiro com o nome fornecido, abre uma ligação a ele e guarda a numa variável global e com o tamanho do ficheiro determina o valor de *lastTrama*, uma variável global que indica qual é o número associado à última trama a ser enviada.

Referência ao código #5

De seguida, é iniciado um ciclo *while*, com a condição de terminação !allFinished, sendo *allFinished* um booleano global inicialmente definido como *false*, que gere a ordem de execução de todas as funções. Através de um comando *switch*, em que a variável ***state*** determina que o *case* é uma das 3 constantes: *CONTROLSTART*, *CONTROLDATA* e *CONTROLEND* definidas em **utils**, sendo 0x02, 0x01 e 0x03 respetivamente. Inicialmente, ***state*** = *CONTROLSTART* e nesse ***case*** é chamado o llwrite para enviar uma trama de controlo de início, e de seguida é chamado *llread()* (com um ciclo while e outro switch case) para receber a resposta do **Recetor**. Aqui é usada a variável global *trama_num*, que conta o número de tramas já enviado, para determinar o Ns e Nr.

Se tudo funcionar sem erro, *state* = *CONTROLDATA* e passamos para o case *CONTROLDATA*, onde é chamado o *llwrite()* para enviar o comando *CONTROLDATA*. De seguida, tal como no *case* anterior, é chamado o *llread()* para receber a resposta do **Recetor**. *trama_num* é mais uma vez utilizada e, em cada iteração, é verificada se esta é igual a *lastTrama*. Quando essa condição se verificar, *state* = *CONTROLEND*.

Para o *case CONTROLEND*, é chamado o *llwrite()* para enviar a trama de fecho. De seguida é chamado o *llread()*, tal como nos *cases* anteriores, para receber a resposta do **Recetor** e utilizando *trama_num* para calcular Ns e Nr. No final, *allFinished = true* para sair do ciclo *while*.

Referência ao código #6

Se tudo correr sem erro, o ciclo while acaba e é chamado o *llclose()*.

O **Recetor** tem uma estrutura parecida, tem o mesmo sistema no main de dividir as tramas de controlo de inicio, de dados, e de controlo do final num *switch case*, chamando apenas o *llread* e não o *llwrite* em cada um dos *cases*.

Em relação à formação dos pacotes de controlo e dos pacotes de dados, ambas são criadas pelas respetivas funções *createControlPacket()* e *createDataPacket()*.

No caso da *createControlPacket()*, esta função recebe como argumento o tipo de pacote de controlo que é, se o de início ou o de fim de transmissão, sendo que depois vai byte a byte

construir o pacote, utilizando os valores estabelecidos, e incluíndo o tamanho e o nome do ficheiro a ser transferido.

Referência ao código #7

No caso da *createDataPacket()*, esta função não necessita de argumentos, analisando automaticamente se está na última trama (que poderá ter um tamanho menor que o máximo) ou numa trama intermédia, indo depois ao ficheiro ler o número de bytes que necessita, executando o *byte stuffing* e colocando na trama de dados, construíndo o pacote de dados pronto para envio.

Referência ao código #8

---

7. Validação

Com fim a testar a nossa aplicação e garantir as suas funcionalidades tal como foram planeadas, experimentou-se a transferência de 3 ficheiros, de tamanhos diferentes, (26 bytes, 10968 bytes e 2619480 bytes), com e sem interrupções de ligação (simulando um erro no envio duma das tramas forçando ao envio de um REJ, que foi seguido do reenvio da trama falhada) e com e sem ruído na ligação, tendo as suas transferências ocorrido sempre com sucesso.

8. Eficiência do protocolo de ligação de dados

Infelizmente não tivemos oportunidade de experimentar na sala o programa final de modo a poder calcular valores de Tf e Tprop experimentais que nos fossem depois permitir calcular a eficiência do nosso programa, que caso tivéssemos obtido esses valores, seria calculada da seguinte forma:

$$Ef = \frac{1-pe}{1+2a} \text{ , em que } a = \frac{Tprop}{Tf}$$

pe = Probabilidade de ocorrer um erro na transmissão de uma trama

$Tprop$ = Tempo que o envio da trama demora até chegar ao seu destino

Tf = Tempo que demora uma trama a ser preparada e o seu envio começar

---

Em baixo encontra-se também uma tabela que correlaciona o tempo gasto na transferência de cada ficheiro que foi utilizado na validação deste programa:

Nome do ficheiro	Tamanho do ficheiro	Tempo gasto na sua transferência
texto.txt	26 bytes	0.004869 segundos
pinguim.gif	10968 bytes	0.014400 segundos
big.jpeg	2619480 bytes	2.773605 segundos

Sendo que este programa opera com um sistema de *Stop-and-Wait*, o tempo gasto para transferência de um ficheiro aumenta, mas por outro lado garante-se um protocolo muito mais

eficaz e capaz de lidar com interrupções ou ruído na ligação, permitindo uma transferência de ficheiros mais fiável em troca de algum tempo de processamento.

9. Conclusões

De um modo geral estamos satisfeitos com a conclusão com sucesso dos objetivos propostos, tendo a transferência dos ficheiros sido conseguida, mesmo quando eram introduzidos no sistema interrupções de ligação ou ruído nas transferências.

O programa é capaz de detetar erros, utilizando o método de *stuffing* e *destuffing* dos dados e BCC (Block Check Character), tal como os corrigir, enviado REJ nos casos com insucesso resultando num reenvio da trama falhada pelo **emissor** até ela ser recebida com sucesso.

Consideramos também que as duas camadas do programa foram bem divididas, tal como sugerido, na camada da ligação lógica e na camada da aplicação.

Desta forma concluímos que o objetivo foi alcançado com sucesso, pois conseguimos estabelecer uma comunicação de dados fiável entre dois computadores ligados por um cabo.

10. Anexos

10.1. Referência ao código #1

```
int llopen(int fd, int flag) {
 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]    = 30;   /* inter-character timer unused */
 newtio.c_cc[VMIN]     = 0;   /* 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("New termios structure set\n");

 while(timeout <= RETRY_ATTEMPTS) {
   if(send_trama_S(fd, TRANSMITTER, SET, CMD_SEND, trama_num))
           printf("SET command sent\n");
       else
           errorMsg("Failed to send SET command!");

   alarm(TIMEOUT_TIME);

   unsigned char new_buf[255];
   int counter = 0;

   while (STOP==FALSE && ERROR==FALSE) {
     read(fd, new_buf + counter, 1);
     switch (counter) {
       case 0:
         if(new_buf[0] != FLAG) ERROR = TRUE;
         break;
       case 1:
         if(new_buf[1] != RES_REC) ERROR = TRUE;
         break;
       case 2:
         if(new_buf[2] != UA) ERROR = TRUE;
         break;
       case 3:
         if(new_buf[3] != (RES_REC^UA)) ERROR = TRUE;
         break;
       case 4:
         if(new_buf[4] != FLAG) ERROR = TRUE;
         break;
     }
     if(counter == 4) STOP = TRUE;
     else counter++;
   }

   if(STOP == TRUE) break;
   else {
     ERROR = FALSE;
     counter = 0;
   }
 }

 if(timeout > RETRY_ATTEMPTS) {
       errorMsg("Failed to receive UA response!");
       return -1;
   } else {
       printf("UA received successfully\n");
       timeout = 1;
       return fd;
   }
}
```

10.2. Referência ao código #2

```
int llwrite(int fd, unsigned char *buf, int length) {
   return send_trama_I(fd, buf, length);
}

int send_trama_I(int fd, unsigned char *buf, int length) {
 if(write(fd, buf, length) >= 0) return length;
 else return FALSE;
}

unsigned char * create_trama_I(unsigned char type, unsigned char SEND) {
   unsigned char BCC2 = 0x00;
   unsigned char *packet, *buf;
   int packetSize;
   if(type == CONTROLDATA) {
       packet = createDataPacket();
       packetSize = lastDataPacketSize;
       buf = (unsigned char *) malloc(6 + packetSize);
   } else {
       packet = createControlPacket(type);
       packetSize = 5 + sizeof(fileSize) + strlen(fileName);
       buf = (unsigned char *) malloc(6 + packetSize);
   }
 buf[0] = FLAG;
 buf[1] = SEND;
 buf[2] = get_Ns(trama_num) << 6;
 buf[3] = SEND ^ (get_Ns(trama_num) << 6);
   memmove(buf + 4, packet, packetSize);
   if(type == CONTROLDATA) {
       buf[4 + packetSize] = lastDataPacketBCC2;
   } else {
       for(int i = 0; i < packetSize; i++)
           BCC2 ^= packet[i];
       buf[4 + packetSize] = BCC2;
   }
   buf[4 + packetSize + 1] = FLAG;

 return buf;
}
```

10.3. Referência ao código #3

```
int llread(int fd, unsigned char *buf) {
   STOP = FALSE, ERROR = FALSE;
   int counter = -1, parametro = 0, packetIsData = FALSE, packetIsRead =
FALSE;
   int fileNameSize = 0, fileSizeSize = 0, actualSize, l1, l2;
   unsigned char BCC2 = 0x00;
   unsigned char *data, *tmpName;

   while (STOP==FALSE && ERROR==FALSE) {
       counter++;
       read(fd, buf + counter, 1);
       switch (counter) {
       case 0:
           if(buf[0] != FLAG) ERROR = TRUE;
           break;
       case 1:
           if(buf[1] != CMD_REC) ERROR = TRUE;
           break;
       case 2:
           if(buf[2] != get_Ns(trama_num) << 6) ERROR = TRUE;
           break;
       case 3:
           if(buf[3] != (CMD_REC^get_Ns(trama_num) << 6)) ERROR = TRUE;
           break;
       case 4:
           if(buf[4] == CONTROLDATA) packetIsData = TRUE;
           else
               if(!((trama_num == 1 && buf[4] == CONTROLSTART) || (trama_num
>= lastTrama && buf[4] == CONTROLEND))) ERROR = TRUE;
           break;
       }
       if(counter >= 5) {
           if(packetIsRead) {
               if(packetIsData) {
                   if(trama_num < lastTrama)
                       for(int i = 0; i < MINK; i++) {
                           BCC2 ^= data[i];
                       }
```

```
                else
                    for(int i = 0; i < lastTramaSize; i++) {
                        BCC2 ^= data[i];
                    }
            } else
                for(int i = 4; i < counter; i++)
                    BCC2 ^= buf[i];
            if(buf[counter] != BCC2) {
                ERROR = TRUE;
            }
            else {
                counter++;
            read(fd, buf + counter, 1);
                if(buf[counter] != FLAG) ERROR = TRUE;
                else STOP = TRUE;
            }
        } else {
            if(packetIsData) {
                switch (parametro) {
                    case 0:
                        BCC2 ^= CONTROLDATA;
                        if(buf[counter] == trama_num % 255) parametro++;
                        else ERROR = TRUE;
                        break;
                    case 1: // L2

                        //if(test == 8) ERROR = TRUE;
                        //test++;

                        BCC2 ^= buf[counter - 1];
                        l2 = buf[counter];
                        parametro++;
                        break;
                    case 2: // L1 e cálculo do actualSize
                        BCC2 ^= buf[counter - 1];
                        l1 = buf[counter];
                        actualSize = 256 * l2 + l1;
                        parametro++;
                        break;
                    case 3:
                        BCC2 ^= buf[counter - 1];
```

```
                        if(trama_num < lastTrama) data = malloc(MINK);
                        else data = malloc(lastTramaSize);
                        int j = 0;
                        for(int i = 0; i < actualSize; i++) {
                            if(ERROR == TRUE) break;
                            if(buf[counter] == ESCAPE) {
                                i++;
                                counter++;
                                read(fd, buf + counter, 1);
                                if(buf[counter] == ESCAPED_FLAG)
                                    data[j] = FLAG;
                                else if(buf[counter] == ESCAPED_ESCAPE)
                                    data[j] = ESCAPE;
                            } else
                                data[j] = buf[counter];
                            counter++;
                            if(i + 1 < actualSize) read(fd, buf +
counter, 1);
                            j++;
                        }
                        parametro = 0;

                        if(trama_num < lastTrama) fwrite(data,
sizeof(unsigned char), MINK, file);
                        else fwrite(data, sizeof(unsigned char),
lastTramaSize, file);
                        packetIsRead = TRUE;
                        break;
                    }
                } else {
                    if(buf[4] == CONTROLSTART) {
                        switch (parametro) {
                            case 0:
                                if(buf[counter] == 0x00) parametro++;
                                else ERROR = TRUE;
                                break;
                            case 1:
                                fileSizeSize = buf[counter];
                                parametro++;
                                break;
                            case 2:
```

```
                            for(int i = 0; i < fileSizeSize; i++) {
                                fileSize = fileSize | (buf[counter] <<
(8*((fileSizeSize-i)-1)));
                                if(!(i == fileSizeSize-1)) {
                                    counter++;
                                    read(fd, buf + counter, 1);
                                }
                            }
                            lastTramaSize = fileSize % MINK;
                            lastTrama = fileSize / MINK;
                            if(lastTramaSize > 0) lastTrama++;
                            else lastTramaSize = MINK;
                            lastTrama += 1;
                            parametro++;
                            break;
                        case 3:
                            if(buf[counter] == 0x01) parametro++;
                            else ERROR = TRUE;
                            break;
                        case 4:
                            fileName = (unsigned char *)
malloc(buf[counter] + 1);
                            fileNameSize = buf[counter];
                            parametro++;
                            break;
                        case 5:
                            for(int i = 0; i < fileNameSize; i++) {
                                fileName[i] = buf[counter];
                                if(!(i == fileNameSize-1)) {
                                    counter++;
                                    read(fd, buf + counter, 1);
                                }
                            }
                            fileName[fileNameSize] = '\0';
                            createFile();
                            parametro = 0;
                            packetIsRead = TRUE;
                            break;
                    }
                } else if(buf[4] == CONTROLEND) {
                    switch (parametro) {
```

```
                            case 0:
                                if(buf[counter] == 0x00) parametro++;
                                else ERROR = TRUE;
                                break;
                            case 1:
                                fileSizeSize = buf[counter];
                                parametro++;
                                break;
                            case 2:
                                for(int i = 0; i < fileSizeSize; i++) {
                                    tmpSize = tmpSize | (buf[counter] <<
(8*((fileSizeSize-i)-1)));
                                    if(!(i == fileSizeSize-1)) {
                                        counter++;
                                        read(fd, buf + counter, 1);
                                    }
                                }
                                if(tmpSize == fileSize) parametro++;
                                else ERROR = TRUE;
                                break;
                            case 3:
                                if(buf[counter] == 0x01) parametro++;
                                else ERROR = TRUE;
                                break;
                            case 4:
                                tmpName = (unsigned char *)
malloc(buf[counter] + 1);
                                fileNameSize = buf[counter];
                                parametro++;
                                break;
                            case 5:
                                for(int i = 0; i < fileNameSize; i++) {
                                    tmpName[i] = buf[counter];
                                    if(!(i == fileNameSize-1)) {
                                        counter++;
                                        read(fd, buf + counter, 1);
                                    }
                                }
                                tmpName[fileNameSize] = '\0';
                                parametro = 0;
```

```
                                if(!strcmp(tmpName, fileName)) packetIsRead =
TRUE;
                                else ERROR = TRUE;
                                break;
                            }
                        }
                    }
                }
            }

    if(ERROR == TRUE) {
        char trash[MINK + 255];
        read(fd, trash, MINK + 255);
        if(send_trama_S(fd, RECEIVER, REJ, RES_SEND, trama_num))
            printf("REJ response sent\n");
        else
            errorMsg("Failed to send REJ response!");
        return -1;
    } else {
      printf("DATA received successfully\n");
      if(send_trama_S(fd, RECEIVER, RR, RES_SEND, trama_num)) {
      printf("RR response sent\n");
      } else
            errorMsg("Failed to send RR response!");
        return counter;
    }
}
```

10.4. Referência ao código #4

```
int llclose(int fd) {
 while(timeout <= RETRY_ATTEMPTS) {
   if(send_trama_S(fd, TRANSMITTER, DISC, CMD_SEND, trama_num))
            printf("DISC command sent\n");
        else
            errorMsg("Failed to send DISC command!");

    alarm(TIMEOUT_TIME);

    unsigned char new_buf[5];
```

```
  int counter = 0;

  while (STOP==FALSE && ERROR==FALSE) {
    read(fd, new_buf + counter, 1);
    switch (counter) {
      case 0:
        if(new_buf[0] != FLAG) ERROR = TRUE;
        break;
      case 1:
        if(new_buf[1] != RES_REC) ERROR = TRUE;
        break;
      case 2:
        if(new_buf[2] != DISC) ERROR = TRUE;
        break;
      case 3:
        if(new_buf[3] != (RES_REC^DISC)) ERROR = TRUE;
        break;
      case 4:
        if(new_buf[4] != FLAG) ERROR = TRUE;
        break;
    }
    if(counter == 4) STOP = TRUE;
    else counter++;
  }

  if(STOP == TRUE) break;
  else {
    ERROR = FALSE;
    counter = 0;
  }
}

if(timeout > RETRY_ATTEMPTS) {
      errorMsg("Failed to receive DISC response!");
      return -1;
  } else {
      printf("DISC received successfully\n");
      timeout = 1;
      if(send_trama_S(fd, TRANSMITTER, UA, CMD_SEND, trama_num))
          printf("UA command sent\n");
      else
```

```
            errorMsg("Failed to send UA command!");

        clock_gettime(CLOCK_REALTIME, &finish);
        double time_taken = (finish.tv_sec - start.tv_sec) + (finish.tv_nsec
- start.tv_nsec) / 1E9;

        printf("Tempo utilizado pelo programa: %f segundos\n", time_taken);

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

        return fd;
    }
}
```

10.5. Referência ao código #5

```
void registerFileData(unsigned char *fname) {
   fileName = fname;
 struct stat st;

 if((file = fopen(fileName, "r")) == NULL) {
   errorMsg("Failed to open file!");
   exit(-1);
 }
 rewind(file);
 if(stat(fileName, &st) == -1) {
   errorMsg("Failed to get file's metadata!");
   exit(-1);
 }
 fileSize = st.st_size;
   lastTramaSize = fileSize % MINK;
   lastTrama = fileSize / MINK;
   if(lastTramaSize > 0) lastTrama++;
   else lastTramaSize = MINK;
   lastTrama += 1;
}
```

10.6. Referência ao código #6

```
while(!allFinished) {
       switch (state) {
           case CONTROLSTART: {
               if(llwrite(fd, create_trama_I(CONTROLSTART, CMD_SEND), 6 + 5
+ sizeof(fileSize) + strlen(fileName)) == -1) {
                   errorMsg("llwrite() falhou!");
                   return -1;
               } else printf("CONTROLSTART command sent\n");

               unsigned char *new_buf = malloc(5);
           int counter = 0;
               STOP = FALSE, ERROR = FALSE;

           alarm(TIMEOUT_TIME);

           while (STOP==FALSE && ERROR==FALSE) {
             read(fd, new_buf + counter, 1);
             switch (counter) {
               case 0:
                 if(new_buf[0] != FLAG) ERROR = TRUE;
                 break;
               case 1:
                 if(new_buf[1] != RES_REC) ERROR = TRUE;
                 break;
               case 2:
                 if(get_Nr(trama_num) == 0) {
                                 if(new_buf[2] != RR && new_buf[2] != REJ)
                                     ERROR = TRUE;
                             } else {
                                 if(new_buf[2] != (RR ^ 0x80) && new_buf[2] !=
(REJ ^ 0x80))
                                     ERROR = TRUE;
                             }
                 break;
               case 3:
                 if(get_Nr(trama_num) == 0) {
                                 if(new_buf[3] != (RES_REC ^ RR) && new_buf[3]
!= (RES_REC ^ REJ))
                                     ERROR = TRUE;
```

```
                        } else {
                            if(new_buf[3] != (RES_REC ^ (RR ^ 0x80)) &&
new_buf[3] != (RES_REC ^ (REJ ^ 0x80)))
                                ERROR = TRUE;
                        }
                        break;
                case 4:
                  if(new_buf[4] != FLAG) ERROR = TRUE;
                  break;
              }
              if(counter == 4) STOP = TRUE;
              else counter++;
            }

            if(STOP == TRUE) {
                    if(new_buf[2] == REJ || new_buf[2] == (REJ ^ 0x80)) {
                        printf("REJ received, repeating CONTROLSTART\n");
                    } else {
                        printf("RR received successfully\n");
                        state = CONTROLDATA;
                        trama_num++;
                    }
                    timeout = 1;
                } else {
                    if(timeout > RETRY_ATTEMPTS) {
                  errorMsg("Failed to receive valid response to
CONTROLSTART!");
                        return -1;
                    }
            }
                free(new_buf);
                break;
            }
            case CONTROLDATA: {
                unsigned char *tmp = malloc(lastDataPacketSize + 6);
                tmp = create_trama_I(CONTROLDATA, CMD_SEND);

                if(llwrite(fd, tmp, lastDataPacketSize + 6) == -1) {
                    errorMsg("llwrite() falhou!");
                    return -1;
                } else printf("CONTROLDATA command sent\n");
```

```
            unsigned char *new_buf = malloc(5);
          int counter = 0;
            STOP = FALSE, ERROR = FALSE;

          alarm(TIMEOUT_TIME);

          while (STOP==FALSE && ERROR==FALSE) {
            read(fd, new_buf + counter, 1);
            switch (counter) {
              case 0:
                if(new_buf[0] != FLAG) ERROR = TRUE;
                break;
              case 1:
                if(new_buf[1] != RES_REC) ERROR = TRUE;
                break;
              case 2:
                if(get_Nr(trama_num) == 0) {
                            if(new_buf[2] != RR && new_buf[2] != REJ)
                                ERROR = TRUE;
                        } else {
                            if(new_buf[2] != (RR ^ 0x80) && new_buf[2] !=
(REJ ^ 0x80))
                                ERROR = TRUE;
                        }
                        break;
              case 3:
                if(get_Nr(trama_num) == 0) {
                            if(new_buf[3] != (RES_REC ^ RR) && new_buf[3]
!= (RES_REC ^ REJ))
                                ERROR = TRUE;
                        } else {
                            if(new_buf[3] != (RES_REC ^ (RR ^ 0x80)) &&
new_buf[3] != (RES_REC ^ (REJ ^ 0x80)))
                                ERROR = TRUE;
                        }
                        break;
              case 4:
                if(new_buf[4] != FLAG) ERROR = TRUE;
                break;
            }
```

```
            if(counter == 4) STOP = TRUE;
            else counter++;
          }
              if(STOP == TRUE) {
                  if(new_buf[2] == REJ || new_buf[2] == (REJ ^ 0x80)) {
                      printf("REJ received, repeating CONTROLDATA\n");
                  } else {
                      printf("RR received successfully\n");
                      if(trama_num == lastTrama) {
                          state = CONTROLEND;
                          fclose(file);
                      }
                      trama_num++;
                  }
                  timeout = 1;
              } else {
                  if(timeout > RETRY_ATTEMPTS) {
                errorMsg("Failed to receive valid response to
CONTROLDATA!");
                      return -1;
                  }
          }
              break;
          }
          case CONTROLEND: {
              if(llwrite(fd, create_trama_I(CONTROLEND, CMD_SEND), 6 + 5 +
sizeof(fileSize) + strlen(fileName)) == -1) {
                  errorMsg("llwrite() falhou!");
                  return -1;
              } else printf("CONTROLEND command sent\n");

              unsigned char *new_buf = malloc(5);
          int counter = 0;
              STOP = FALSE, ERROR = FALSE;

          alarm(TIMEOUT_TIME);

          while (STOP==FALSE && ERROR==FALSE) {
            read(fd, new_buf + counter, 1);
            switch (counter) {
              case 0:
```

```
                if(new_buf[0] != FLAG) ERROR = TRUE;
                break;
              case 1:
                if(new_buf[1] != RES_REC) ERROR = TRUE;
                break;
              case 2:
                if(get_Nr(trama_num) == 0) {
                            if(new_buf[2] != RR && new_buf[2] != REJ)
                                ERROR = TRUE;
                        } else {
                            if(new_buf[2] != (RR ^ 0x80) && new_buf[2] !=
(REJ ^ 0x80))
                                ERROR = TRUE;
                        }
                break;
              case 3:
                if(get_Nr(trama_num) == 0) {
                            if(new_buf[3] != (RES_REC ^ RR) && new_buf[3]
!= (RES_REC ^ REJ))
                                ERROR = TRUE;
                        } else {
                            if(new_buf[3] != (RES_REC ^ (RR ^ 0x80)) &&
new_buf[3] != (RES_REC ^ (REJ ^ 0x80)))
                                ERROR = TRUE;
                        }
                        break;
              case 4:
                if(new_buf[4] != FLAG) ERROR = TRUE;
                break;
            }
            if(counter == 4) STOP = TRUE;
            else counter++;
          }

          if(STOP == TRUE) {
                  if(new_buf[2] == REJ || new_buf[2] == (REJ ^ 0x80)) {
                      printf("REJ received, repeating CONTROLEND\n");
                  } else {
                      printf("RR received successfully\n");
                      allFinished = TRUE;
                  }
```

```
                    timeout = 1;
                } else {
                    if(timeout > RETRY_ATTEMPTS) {
                  errorMsg("Failed to receive valid response to
CONTROLEND!");
                        return -1;
                    }
            }
                free(new_buf);
                break;
            }
        }
    }
```

10.7. Referência ao código #7

```
unsigned char *createControlPacket(unsigned char type) {
 unsigned char *packet = (unsigned char *) malloc(5 + sizeof(fileSize) +
strlen(fileName));

 packet[0] = type;
 packet[1] = 0x00; //Represents filesize area
 packet[2] = sizeof(fileSize);
 for(int i = 0; i < sizeof(fileSize); i++)
   packet[i + 3] = (fileSize >> (8*(sizeof(fileSize)-1-i)));
 packet[3 + sizeof(fileSize)] = 0x01; //Represents filename area
 packet[3 + sizeof(fileSize) + 1] = strlen(fileName);
   for (int j = 0; j < strlen(fileName); j++)
       packet[5 + sizeof(fileSize) + j] = fileName[j];

       return packet;
}
```

10.8. Referência ao código #8

```
unsigned char *createDataPacket() {
   unsigned char *packet, *tmp;
   int toTransferSize, actualSize, j = 0;
   if(lastTrama == trama_num) {
       tmp = malloc(lastTramaSize);
       actualSize = lastTramaSize;
```

```
        toTransferSize = lastTramaSize;
    } else {
        tmp = malloc(MINK);
        actualSize = MINK;
        toTransferSize = MINK;
    }

    fread(tmp, sizeof(unsigned char), toTransferSize, file);

for(int i = 0; i < toTransferSize; i++)
        if(tmp[i] == FLAG || tmp[i] == ESCAPE) actualSize++;

    packet = malloc(4 + actualSize);

packet[0] = CONTROLDATA;
packet[1] = trama_num % 255;
    packet[2] = (int) actualSize / 256;
    packet[3] = (int) actualSize % 256;
    lastDataPacketBCC2 = 0x00;
    lastDataPacketBCC2 ^= packet[0];
    lastDataPacketBCC2 ^= packet[1];
    lastDataPacketBCC2 ^= packet[2];
    lastDataPacketBCC2 ^= packet[3];
for(int i = 0; i < toTransferSize; i++) {
        lastDataPacketBCC2 ^= tmp[i];
        if(tmp[i] == FLAG) {
            packet[4 + j] = ESCAPE;
            j++;
            packet[4 + j] = ESCAPED_FLAG;
        } else if(tmp[i] == ESCAPE) {
            packet[4 + j] = ESCAPE;
            j++;
            packet[4 + j] = ESCAPED_ESCAPE;
        } else {
            packet[4 + j] = tmp[i];
        }
        j++;
    }

    lastDataPacketSize = 4 + actualSize;
```

```
    return packet;
}
```