Faculdade de Engenharia da Universidade do Porto

1º Trabalho Laboratorial

Ligação de Dados

Redes de Computadores 2020/2021

3MIEIC01 - 1ºsemestre

24 Novembro 2020

Grupo 7

Ana Teresa Dias Silva (up201606703) - up201606703@fe.up.pt

Rodrigo Campos Reis (up201806534) - up201806534@fe.up.pt

Índice

1. Sumário

O presente relatório, elaborado como complemento do primeiro trabalho laboratorial da Unidade Curricular de Redes de Computadores, expõe mais detalhadamente pormenores da sua implementação, funcionamento e eficiência.

O trabalho, que consiste numa aplicação de transferência de ficheiros entre dois computadores através de uma porta de série, foi concluído com sucesso, visto que essa transferência é feita na totalidade sem qualquer perda de informação, mesmo quando sujeita a perturbações ou interrupções.

2. Introdução

O trabalho desenvolvido teve como base de objetivos: **implementar um protocolo de ligação de dados, testar esse protocolo com uma aplicação de transferência de ficheiros e medir a eficiência do mesmo.**

Este relatório é uma análise e reflexão do trabalho desenvolvido, procurando abordar os seguintes tópicos:

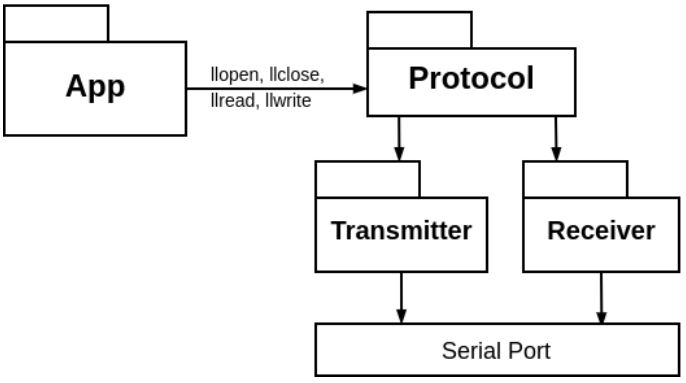
- **Arquitetura** - organização geral do software.
- **Estrutura do código -** estruturação do código e descrição do conteúdo dos ficheiros.
- **Casos de uso -** como usar o software.
- **Protocolo de ligação lógica -** descrição do funcionamento do protocolo através das funções principais.
- **Protocolo da aplicação -** descrição do funcionamento do protocolo da aplicação.
- **Validação -** descrição dos testes efetuados.
- **Eficiência -** avaliação da eficiência do protocolo avaliando diferentes tamanhos de ficheiros.

3. Arquitetura

O software está dividido nos seguintes blocos: **emissor**, **recetor**, **protocolo** e **aplicação**.

O **emissor** e **recetor** são responsáveis por escrever e ler na porta de série, respetivamente, seguindo o protocolo de ligação de dados. Para além disso, o **emissor** é responsáveis pela leitura do ficheiro e processamento das tramas para envio. Já o **recetor** verifica as tramas recebidas e cria um ficheiro com a informação previamente recebida e processada.

O **protocolo** é a camada que estabelece a ligação entre a **aplicação** e o **emissor** e **recetor**, permitindo executar as principais funções do protocolo de ligação de dados,



nomeadamente a conexão, o envio e a receção de dados e o fecho da ligação.

A **aplicação** é camada que permite interagir com o utilizador.

Imagem 1 - Diagrama de arquitetura lógica

4. Estrutura do código

4.1. Ficheiros principais

O código está dividido em quatro ficheiros principais e dois complementares.

Os ficheiros ***transmitter***, ***receiver***, ***protocol*** e ***app*** são responsáveis pela execução principal do programa: o ***transmitter*** e o ***receiver*** contém as funções relacionadas com

todas as ações efetuadas pelo computador que envia e pelo computador que recebe o ficheiro, respetivamente.

O ficheiro ***protocolo*** representa a camada do protocolo de ligação de dados - tudo o que está inerente à conexão entre as máquinas e ao envio da informação. Aqui incluem-se as funções *llopen*, *llwrite*, *llread* e *llclose*.

No ficheiro ***app*** é feita a interação entre o utilizador e o protocolo da ligação, nomeadamente o processamento dos argumentos e do nome do ficheiro.

4.2. Ficheiros auxiliares

Existe ainda um ficheiro ***utils*** com funções que são utilizadas por uma ou várias funções de outros ficheiros e que auxiliam no funcionamento das mesmas. No header desse mesmo ficheiro são definidas todas as macros e estruturas utilizadas de forma partilhada pelos ficheiros, nomeadamente as estruturas *FileInfo* e *State_Machine*.

O ficheiro ***alarm*** contém o handler do alarme utilizado na aplicação.

5. Casos de uso principais

O principal caso de uso da aplicação é a interface com o utilizador. No caso de ser emissor, o utilizador indica o nome do ficheiro que pretende enviar, juntamente com a porta de série através da qual será feita a ligação, no seguinte formato:

./app <porta de série> send <ficheiro> (ex. ./app /dev/ttyS0 send pinguim.gif)

Já como recetor, apenas é necessário indicar a porta de série através da qual o ficheiro será recebido:

./app <porta de série> receive (ex. ./app /dev/ttyS0 receive)

Após o processamento dos argumentos, a transferência do ficheiro é feita por esta ordem de acontecimentos:

- Configuração e estabelecimento da ligação entre os dois computadores através

da porta de série

- Emissor lê e processa o ficheiro que será enviado
- Emissor envia os dados divididos em tramas de informação
- Recetor recebe as tramas de informação
- Recetor guarda os dados recebidos num ficheiro criado por si cujo nome segue o formato **received_<nome original>** (ex. received_pinguim.gif)
- Término da ligação entre os computadores

6. Protocolo de ligação lógica

A função do protocolo de ligação lógica é estabelecer uma estratégia de comunicação entre duas máquinas através da porta de série.

6.1. Funções do protocolo

`int llopen(int fd, int status)` **- Estabelece a ligação entre o emissor e o recetor através da porta de série.**

O valor de *VTIME* é definido como 3 e *VMIN* como 0.

Mediante o status recebido por argumento (*TRANSMITTER* ou *RECEIVER*) é executada a função ***setReceiver***, no caso de se tratar do recetor ou ***setTransmitter*** , no caso de se tratar do emissor**.**

`int llwrite(int fd, FILE* file, char* filename)` - **Função do emissor responsável pelo processamento do ficheiro, divisão dos dados em tramas de informação, processo de stuffing e envio das tramas.**

Inicialmente o ficheiro é lido e a informação a processar é armazenada num buffer.

Antes de enviar as tramas de informação, é enviada uma trama de controlo com recurso à função ***sendControl***.

Posteriormente, na função ***sendData*** as tramas de informação são criadas de acordo com o formato pretendido, usando o buffer com a informação do ficheiro.
Os pacotes de informação são criados na função ***generateDataPackage***, o cálculo do BCC2 para cada trama é feito na função ***calculateBCC2*** e o processo de stuffing dos dados na função ***stuffingData***. Por fim, a trama de informação é criada com os dados processados e enviados para o recetor.

Se a informação for enviada corretamente, é enviado no fim uma trama de controlo recorrendo novamente à função ***sendControl***.

`int llread(int fd)` - **Função do recetor responsável pela leitura das tramas enviadas.**

Primeiramente é chamada a função ***receiveControlPackage*** que se encarrega de receber a informação do pacote de controlo enviado e armazená-la numa estrutura fileInfo.

Seguidamente, é calculado o número de tramas que irá ser recebido com base no tamanho do ficheiro. Através do número de tramas é executado um ciclo que chama a função ***stateMachine***, recebe os dados enviados, verifica o seu tamanho, se é um duplicado ou se tem algum erro (descarta neste caso). Se os dados estiverem bem guarda a informação num buffer.

Quando termina o ciclo volta a ser chamada a função ***receiveControlPackage*** que está agora à espera de receber a trama de controlo a indicar o fim do envio de dados.

Neste momento, é chamada a função ***createFile*** que cria o ficheiro com a informação obtida.

`int llclose(int fd, int status)` - **Termina a ligação entre o emissor e o recetor através da porta de série.**

O emissor começa por enviar uma trama de supervisão DISC na função ***closeConnection***. O recetor, por sua vez, quando recebe essa trama e a processa na máquina de estados, envia uma trama de supervisão semelhante para o emissor na

função ***handleDisconnection***. Para terminar, o emissor envia uma trama do tipo UA ao recetor e a ligação é terminada.

7. Protocolo de aplicação

O protocolo de aplicação garante o envio e processamento dos pacotes de controlo, a divisão do ficheiro em tramas pelo emissor após a leitura do ficheiro, a escrita das tramas, após devidamente processadas pelo recetor, para um novo ficheiro criado e ainda o processamento das diferentes tramas enviadas.

`unsigned char *generateControlPackage(int fileSize, unsigned char *fileName, int *packageSize, int controlfield)` - **Cria e retorna um pacote de controlo** com as informações relativas ao nome e tamanho do ficheiro.

`void sendControlPackage(int fd, unsigned char *controlPackage, int *size, unsigned char bcc2, int s)` - **Escreve num buffer** as informações recebidas no argumento ***controlPackage*** e ***bcc2*** e **envia-o ao recetor.**

`int checkControlPackage(unsigned char *controlPackage, int *size, fileInfo *fileinfo)` - **Verifica o pacote de controlo** recebido com as informações sobre o nome e o tamanho do ficheiro e **armazena esses dados** na estrutura ***fileInfo***.

`void createFile(fileInfo info, unsigned char *fileData)` - **Cria o ficheiro com o nome e o tamanho passado na estrutura *fileInfo*** e escreve a informação armazenada no buffer ***fileData***

`unsigned char * stateMachine(int fd, unsigned char header, char controlField, int type, int size` - Máquina de estados onde é feito o **processamento das tramas** S e das tramas I.

8. Validação

Para testarmos a aplicação de forma exaustiva e abrangente efetuamos os seguintes testes:

- Envio de ficheiros de diferentes tipos (png, gif, pdf)
- Utilização de ficheiros de dimensões variadas e com fundo transparente e não transparente
- Interrupção momentânea da ligação durante o envio dos ficheiros
- Introdução de ruído na porta de série durante a transferência dos ficheiros
- Junção dos dois tópicos anteriores no envio do ficheiro

9. Eficiência do protocolo de ligação de dados

Para os cálculos da eficiência do protocolo foi utilizada uma porta de série virtual. Utilizámos três ficheiros com extensão .gif de dimensões diferentes e medimos o tempo de execução do programa para o emissor e para o recetor.

Os resultados obtidos são apresentados no seguinte gráfico:

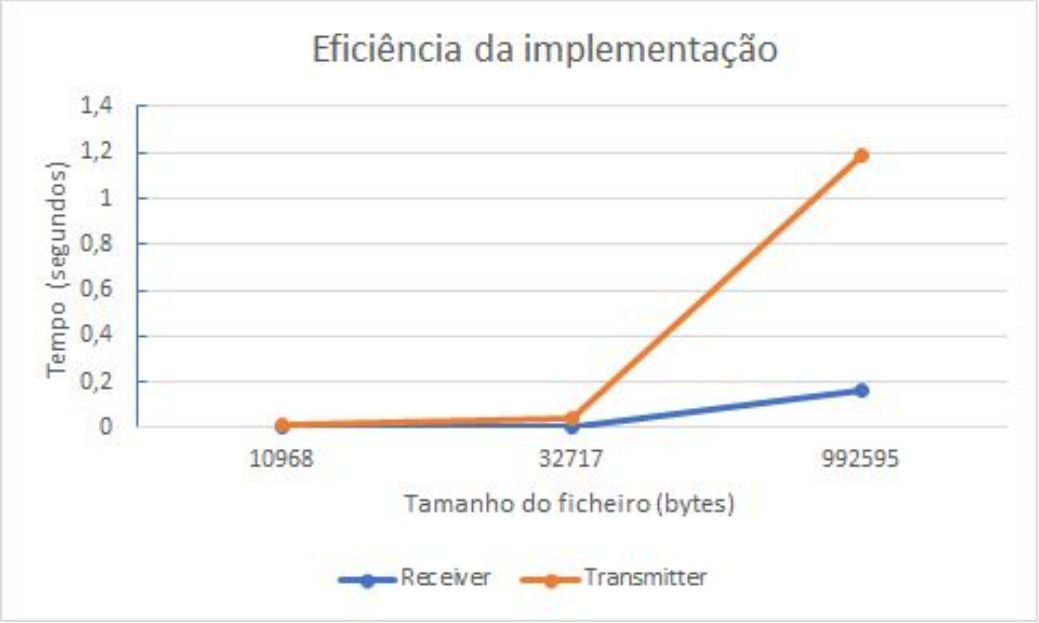


Imagem 2 - Gráfico de eficiência da implementação

Pela análise do gráfico podemos concluir, na generalidade, que quanto maior o ficheiro, maior o tempo de execução.

Em relação ao emissor e ao recetor, percebemos a diferença notória do seu tempo de execução, o que é justificado pelas verificações efectuadas nas tramas de informação, criação do ficheiro e escrita no mesmo, do lado do recetor.

Relativamente ao protocolo *Stop and Wait*, com o auxílio do *alarm,* o emissor envia continuamente a mesma trama até receber uma resposta **ACK** do recetor. Se o recetor receber um duplicado, simplesmente descarta-o.

10. Conclusões

Tendo em conta os requisitos de independência entre camadas, tivemos em conta essa mesma independência no sentido em que os detalhes da camada do protocolo da ligação de dados são totalmente desconhecidos para a camada da aplicação.

Durante o desenvolvimento deste projeto, os nossos conhecimentos sobre os diversos temas da Unidade Curricular foram aprofundados e, simultaneamente, permitiu-nos compreender melhor as aplicações práticas dos aspectos teóricos apresentados nas aulas.

Consideramos que o trabalho foi concluído com sucesso visto ser totalmente funcional e obedecer a todas as especificações iniciais.

11. Anexo I - código fonte

alarm.h

```
#pragma once
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

#define TRUE 1
#define FALSE 0

extern int numRetry;
extern int alarmFlag;

/**
 * @brief Runs when alarm is triggered
 *
 */
void alarmHandler();
```

alarm.c

```
#include "alarm.h"

void alarmHandler()
{
    printf("\nAlarm: %d\n", numRetry + 1);
    alarmFlag = TRUE;
    numRetry++;
}
```

app.c

```
#include "protocol.h"

int main(int argc, char **argv)
{
    int status;
    if (argc == 3)
    {
        status = RECEIVER;
    }
```

```
else if (argc == 4)
{
    status = TRANSMITTER;
}
else
{
    printf("Usage:\n");
    printf("\tTRANSMITTER: ./app /dev/sttyS0 send <filename>\n");
    printf("\tRECEIVER: ./app /dev/sstyS1 receive\n");
    return -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.
*/
int fd = open(argv[1], O_RDWR | O_NOCTTY);
if (fd < 0)
{
    perror(argv[1]);
    exit(-1);
}

FILE *file;
if (status == TRANSMITTER){
    if (!(file = fopen(argv[3], "rb"))){
        printf("Error opening the file or the file does not exist!\n");
        return (-1);
    }
    else{
        printf("Reading file...\n");
    }
}

if (llopen(fd, status) == -1)
    return -1;

switch (status)
{
case TRANSMITTER:
    llwrite(fd, file, argv[3]);
    break;
case RECEIVER:
    llread(fd);
    break;
```

```
    default:
        return -1;
        break;
    }

    llclose(fd, status);

    return 0;
}
```

protocol.h

```
#pragma once
#include "utils.h"
#include "receiver.h"
#include "transmitter.h"

#define TRANSMITTER 0
#define RECEIVER 1

/**
 * @brief
 *
 * @param fd serial port descriptor
 * @param status TRANSMITTER or RECEIVER
 * @return execution status
 */
int llopen(int fd, int status);

/**
 * @brief Write to Serial Port
 *
 * @param fd serial port descriptor
 * @param file file pointer
 * @param filename filename
 * @return execution status
 */
int llwrite(int fd, FILE *file, char *filename);

/**
 * @brief Read from Serial Port
 *
 * @param fd serial port descriptor
 * @return execution status
 */
int llread(int fd);
```

```
/**
 * @brief Close connection
 *
 * @param fd serial port descriptor
 * @param status TRANSMITTER or RECEIVER
 * @return execution status
 */
int llclose(int fd, int status);
```

protocol.c

```
#include "protocol.h"

int llopen(int fd, int status)
{
    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] = 3; /* inter-unsigned character timer unused */
    newtio.c_cc[VMIN] = 0;  /* blocking read until 5 unsigned chars received */

    /*
  VTIME e VMIN devem ser alterados de forma a proteger com um temporizador a
  leitura do(s) proximo(s) caracter(es)
  */

    tcflush(fd, TCIOFLUSH);

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

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

    printf("\nStarting connection...\n\n");

    if (status == TRANSMITTER)
    {

        if (!setTransmitter(fd))
        {
            printf("\nCommunication protocol failed after %d tries!\n\n",
MAX_RETRY);
            return -1;
        }
    }
    else if (status == RECEIVER)
    {
        setReceiver(fd);
    }
    else
    {
        printf("Invalid status!\n");
        return -1;
    }
    return 0;
}

int llwrite(int fd, FILE *file, char *filename)
{
    int fileSize = getFileSize(file);

    unsigned char *buffer = malloc(sizeof(unsigned char) * fileSize);
    /*Lê ficheiro*/
    fread(buffer, sizeof(unsigned char), fileSize, file);

    int seqN = sendControl(fd, fileSize, filename, 0x02);

    sendData(fd, buffer, fileSize, seqN);

    sendControl(fd, fileSize, filename, 0x03);

    return 0;
}
```

```
int llread(int fd)
{
    fileInfo dataInfo = receiveControlPackage(fd);

    int nTramas = dataInfo.size / 256;

    int l1 = dataInfo.size / 256;
    int l2 = dataInfo.size % 256;

    if (l2 != 0)
    {
        nTramas++;
    }

    printf("NTRMAS: %d", nTramas);
    int *size = malloc(sizeof(int));

    unsigned char *fileData = malloc(sizeof(unsigned char) * dataInfo.size);

    int counter = 0;
    int n = -1;
    int currentN = 0;
    int fail = FALSE;
    for (int i = 0; i < nTramas; i++)
    {
        if (fail == TRUE){
            printf("NTramas: %d\n falhou aqui - %d\n", nTramas, i);
        }

        unsigned char *data = stateMachine(fd, A_TRM, 0x00, I, size);
        currentN = data[1];

        if (currentN == n + 1)
        {
            if (i < nTramas - 1 && (*size >= 255 && *size <= 261))
            {
                int cnt = 0;
                for (int d = 4; d < (*size) - 1; d++)
                {
                    cnt++;
                    fileData[counter++] = data[d];
                }
                printf("TRAMA - %d | SIZE: %d\n", data[1], cnt);
                fail = FALSE;
                n = currentN;
            }
```

```
        else if (i == nTramas - 1 && (*size >= l2 && *size <= l2 + 5))
        {
            int cnt = 0;
            for (int d = 4; d < (*size) - 1; d++)
            {
                cnt++;
                fileData[counter++] = data[d];
            }
            printf("TRAMA ULTIMA - %d | SIZE: %d\n", data[1], cnt);
            n = currentN;
        }
        else if (*size > 261)
        {
            int cnt = 0;
            for (int d = (*size) - 257; d < (*size) - 1; d++)
            {
                cnt++;
                fileData[counter++] = data[d];
            }
            printf("TRAMA - %d | SIZE: %d\n", data[1], cnt);
            fail = FALSE;
            n = currentN;
        }
        else
        {
            printf("%d\n", *size);
            printf("trama incorreta com i = %d\n", i);
            //  sendControlMsg(fd, A_TRM, 0x01);
            i--;
            printf("Diminui i: %d\n", i);
        }
    }
    else if (currentN == n)
    { //duplicado
        printf("RECEBI DUPLICADO com i = %d\n", i);
        i--;
        printf("Diminui i: %d\n", i);
    }
    else
    {
        printf("FALHOU CURRENTN - %d - I:  %d\n", currentN, i);
        printf("%s\n", data);
        i--;
    }
```

```
        if (currentN == 255)
        {
            n = -1;
        }
    }

    printf("SIZE: %d", counter);

    fileInfo dataInfoFinal = receiveControlPackage(fd);

    createFile(dataInfo, fileData);

    return 0;
}

int llclose(int fd, int status)
{
    if (status == TRANSMITTER)
    {
        closeConnection(fd);
    }
    if (status == RECEIVER)
    {
        handleDisconnection(fd);
    }
    close(fd);

    return 0;
}
```

receiver.h

```
#pragma once
#include "utils.h"


/**
 * @brief Set receiver connection
 *
 * @param fd serial port descriptor
 */
void setReceiver(int fd);
```

```
/**
 * @brief
 *
 * @param fd serial port descriptor
 * @return file information
 */
fileInfo receiveControlPackage(int fd);

/**
 * @brief Verify control package
 *
 * @param controlPackage control package
 * @param size size of control package
 * @param fileinfo file information
 * @return control bit (2 or 3)
 */
int checkControlPackage(unsigned char *controlPackage, int *size, fileInfo
*fileinfo);

/**
 * @brief Create a file and write
 *
 * @param info file info
 * @param fileData buffer with file data
 */
void createFile(fileInfo info, unsigned char *fileData);

/**
 * @brief Disconnect receiver
 *
 * @param fd serial port descriptor
 */
void handleDisconnection(int fd);
```

receiver.c

```
#include "receiver.h"

void setReceiver(int fd)
{
    int *size = malloc(sizeof(int));
    stateMachine(fd, A_TRM, C_SET, S, size);
    printf("\nTrama SET recebida\n");
    sendControlMsg(fd, A_TRM, C_UA);
    printf("\nTrama UA enviada\n");
}
```

```
fileInfo receiveControlPackage(int fd)
{

    int *sizeControlPackage = malloc(sizeof(int));
    unsigned char *controlPackage = stateMachine(fd, A_TRM, 0x00, I,
sizeControlPackage);

    fileInfo fileinfo;
    int controlPackageStatus = checkControlPackage(controlPackage,
sizeControlPackage, &fileinfo);

    printf("\nTrama I de controlo recebida - STATUS: %x\n", controlPackageStatus);

    return fileinfo;
};

int checkControlPackage(unsigned char *controlPackage, int *size, fileInfo
*fileinfo)
{
    //Filesize field
    if (controlPackage[1] == 0)
    {
        int fileSize = 0;
        int shift = 24;
        for (int i = 3; i < controlPackage[2] + 3; i++){
            fileSize |= (int)controlPackage[i] << shift;
            shift -= 8;
        }
        fileinfo->size = fileSize;
    }

    //Filename field
    if (controlPackage[7] == 1) {
        unsigned char *fileName = malloc(sizeof(unsigned char) *
controlPackage[8]);
        int counter = 0;

        for (int i = 9; i < controlPackage[8] + 9; i++){
            int nameSize = 0;
            fileName[counter++] = controlPackage[i];
        }
        fileinfo->filename = fileName;
    }

    return controlPackage[0];
```

```
}

void createFile(fileInfo info, unsigned char *fileData)
{
    char path[9 + strlen(info.filename)];
    strcpy(path, "received_");
    strcat(path, info.filename);
    printf("PATH: %s\n", path);
    FILE *fp = fopen(path, "wb+");

    fwrite((void *)fileData, 1, info.size, fp);
    fclose(fp);
}

void handleDisconnection(int fd)
{
    int *size = malloc(sizeof(int));

    stateMachine(fd, A_TRM, C_DISC, S, size);
    printf("Trama DISC recebida!\n");
    sendControlMsg(fd, A_REC, C_DISC);
    printf("Trama DISC enviada!\n");

    stateMachine(fd, A_REC, C_UA, S, size);
    printf("Trama UA recebida!\n");

    printf("Connection closed!\n");
}
```

transmitter.h

```
#pragma once

#include <fcntl.h>
#include "utils.h"
#include "alarm.h"

#define TIMEOUT 5
#define C_START 0x02
#define C_DATA 0x01
#define T1 0x00
#define T2 0x01
#define L1 0x04

extern int numRetry;
```

```c
extern int alarmFlag;

/**
 * @brief Set transmitter connection
 *
 * @param fd serial port descriptor
 * @return execution status
 */
int setTransmitter(int fd);

/**
 * @brief Send a control package
 *
 * @param fd serial port descriptor
 * @param controlPackage control package buffer
 * @param size size of control package
 * @param bcc2 bcc2 value
 * @param s message type
 */
void sendControlPackage(int fd, unsigned char *controlPackage, int *size, unsigned
char bcc2, int s);

/**
 * @brief Generate a control package
 *
 * @param fileSize file size
 * @param fileName file name
 * @param packageSize package size
 * @param controlfield control field (0x02 or 0x03)
 * @return pointer to control package
 */
unsigned char *generateControlPackage(int fileSize, unsigned char *fileName, int
*packageSize, int controlfield);

/**
 * @brief Send all data
 *
 * @param fd serial port descriptor
 * @param buffer data buffer
 * @param size size of data
 * @param seqN sequence number
 */
void sendData(int fd, unsigned char *buffer, int size, int seqN);
```

```
/**
 * @brief Generate a data package
 *
 * @param buffer file buffer
 * @param size file size
 * @param n data sequence number
 * @param l1 number of complete sequences
 * @param l2 number of bytes in last sequence
 * @return pointer to data package
 */
unsigned char *generateDataPackage(unsigned char *buffer, int *size, int n, int l1,
int l2);

/**
 * @brief Send Control Message
 *
 * @param fd serial port descriptor
 * @param fileSize file size
 * @param fileName file name
 * @param controlField control field
 * @return sequence number
 */
int sendControl(int fd, int fileSize, unsigned char *fileName, int controlField);

/**
 * @brief Close transmitter connection
 *
 * @param fd serial port descriptor
 */
void closeConnection(int fd);
```

transmitter.c

```
#include "transmitter.h"

int numRetry = 0;
int alarmFlag = FALSE;

int setTransmitter(int fd)
{
    (void)signal(SIGALRM, alarmHandler);

    int state = START;
    unsigned char c;
```

```
    do
    {
        sendControlMsg(fd, A_TRM, C_SET);
        printf("\nTrama SET enviada\n");

        alarmFlag = FALSE;
        alarm(TIMEOUT);

        int *size = malloc(sizeof(int));
        stateMachine(fd, A_TRM, C_UA, S, size);

    } while (alarmFlag && numRetry < MAX_RETRY);

    //Comunicação falha após atingir o número máximo de tentativas
    if (alarmFlag && numRetry == MAX_RETRY)
        return FALSE;

    else
    {
        printf("\nTrama UA recebida\n");
        alarmFlag = FALSE;
        numRetry = 0;
        printf("\nProtocol connection established!\n");
        return TRUE;
    }
}

void sendControlPackage(int fd, unsigned char *controlPackage, int *size, unsigned
char bcc2, int s){
/*
 *  Trama I : FLAG | A | C | BCC1 | Dados (pacote controlo) | BCC2 | FLAG
 */
    int bufferSize = *size + 6;
    unsigned char buffer[bufferSize];

    int counter = 0;
    buffer[counter++] = FLAG;
    buffer[counter++] = A_TRM;
    if (s == 0)
    {
        buffer[counter++] = 0x00;
    }
    else
    {
        buffer[counter++] = 0x40;
```

```
    }
    buffer[counter++] = buffer[1] ^ buffer[2]; //bcc

    //SEND CONTROL PACKAGE HERE
    for (int i = 0; i < (*size); i++)
    {
        buffer[counter++] = controlPackage[i];
    }

    buffer[counter++] = FLAG;

    write(fd, &buffer, bufferSize);
}

unsigned char *generateControlPackage(int fileSize, unsigned char *fileName, int
*packageSize, int controlfield)
{
    int sizeFileName = strlen(fileName);
    int packSize = 9 * sizeof(unsigned char) + sizeFileName; //C,T1,L1,T2,L2(5) +
sizeof(fileName) + tamanho campo filesize(4)
    unsigned char *controlPackage = malloc(sizeof(unsigned char) * (packSize + 1));

    /* controlPackage = [C,T1,L1,V1,T2,L2,V2]
    * C = 2 (start) || C=3 (end)
    * T = 0 (tamanho ficheiro) || T = 1(nome ficheiro)
    * L = tamanho campo V
    * V = valor
    */
    controlPackage[0] = controlfield;
    controlPackage[1] = T1; //file size
    controlPackage[2] = L1;
    controlPackage[3] = (fileSize >> 24) & 0xFF;
    controlPackage[4] = (fileSize >> 16) & 0xFF;
    controlPackage[5] = (fileSize >> 8) & 0xFF;
    controlPackage[6] = (fileSize & 0xFF);
    controlPackage[7] = T2; //filename
    controlPackage[8] = sizeFileName;
    for (int i = 0; i < sizeFileName; i++)
    {
        controlPackage[9 + i] = fileName[i];
    }

    *packageSize = packSize + 1;

    return controlPackage;
}
```

```
void sendData(int fd, unsigned char *buffer, int size, int seqN)
{
    //Cálculo nr tramas
    int nTramas;

    int l1 = size / 256;
    int l2 = size % 256;

    nTramas = l1;
    if (l2 != 0)
    {
        nTramas++;
    }

    unsigned char info[MAX_SIZE];

    info[0] = FLAG;
    info[1] = A_TRM;

    for (int i = 0; i < nTramas; i++)
    {
        (void)signal(SIGALRM, alarmHandler);
        numRetry = 0;

        do
        {
            int counter = 2;

            //send
            if (seqN == 0)
            {
                info[counter++] = 0x00;
            }
            else
            {
                info[counter++] = 0x40;
            }

            //BCC
            info[counter++] = info[1] ^ info[2];

            int *dataPackageSize = malloc(sizeof(int));
            *dataPackageSize = size;

            unsigned char *dataPackage = generateDataPackage(buffer,
```

```
dataPackageSize, i, l1, l2);

            //BCC2
            unsigned char bcc2 = calculateBCC2(dataPackage, *dataPackageSize - 1);

            dataPackage[*dataPackageSize - 1] = bcc2;

            //stuffing
            unsigned char *stuffedData = stuffingData(dataPackage,
dataPackageSize);

            //data
            for (int j = 0; j < (*dataPackageSize); j++)
            {
                info[counter++] = stuffedData[j];
            }

            info[counter++] = FLAG;

            alarmFlag = FALSE;
            alarm(TIMEOUT);

            write(fd, &info, counter);

            int *size = malloc(sizeof(int));

            int c_state;

            if (seqN == 0)
            {
                c_state = 0x05; //Expects positive ACK -> controlField val = 0x05
(R = 0)
            }
            else
            {
                c_state = 0x85; //Expects positive ACK -> controlField val = 0x85
(R = 1)
            }

            unsigned char *status = stateMachine(fd, A_TRM, c_state, S, size);
            printf("Status: %x\n", status[0]);
            if (status[0] == 0x0)
            {
                printf("Trama RR recebida!\n");
                break;
            }
```

```
            else if (status[0] == 0x1)
            {
                printf("Trama RJ recebida - send Data!\n");
                printf("ALARM FLAG %d numretry %d", alarmFlag, numRetry);
            }
            else
            {
                printf("Waiting.... \n");
            }

            (seqN == 0) ? seqN++ : seqN--;

        } while (alarmFlag && numRetry < MAX_RETRY);
        if (alarmFlag && numRetry == MAX_RETRY)
            break;

        (seqN == 0) ? seqN++ : seqN--;
    }

    if (alarmFlag && numRetry == MAX_RETRY)
        return;

    else
    {
        alarmFlag = FALSE;
        numRetry = 0;
        return;
    }
}

int sendControl(int fd, int fileSize, unsigned char *fileName, int controlField)
{
    //Envia trama de controlo
    int *size = malloc(sizeof(int));
    unsigned char *controlPackage = generateControlPackage(fileSize, fileName,
size, controlField);

    //Calculo do BCC com informacao
    unsigned char bcc2 = calculateBCC2(controlPackage, *size - 1);

    controlPackage[*size - 1] = bcc2;

    unsigned char *stuffedControlPackage = stuffingData(controlPackage, size);
```

```
    //Espera pelo Aknowledge - máquina de estados
    int seqN = 0;
    do
    {
        sendControlPackage(fd, stuffedControlPackage, size, bcc2, seqN);

        alarmFlag = FALSE;
        alarm(TIMEOUT);

        int *size = malloc(sizeof(int));
        int c_state;

        if (seqN == 0){
            c_state = 0x05; //Expects positive ACK -> controlField val = 0x05 (R =
0)
        }
        else{
            c_state = 0x85; //Expects positive ACK -> controlField val = 0x85 (R =
1)
        }

        unsigned char *status = stateMachine(fd, A_TRM, c_state, S, size);
        if (status[0] == 0x0)
        {
            printf("Trama RR recebida!\n");
            break;
        }
        else if (status[0] == 0x1)
        {
            printf("Trama RJ recebida - send Control!\n");
        }

        (seqN == 0) ? seqN++ : seqN--;

    } while (alarmFlag && numRetry < MAX_RETRY);

    if (alarmFlag && numRetry == MAX_RETRY)
        return -1;

    else
    {
        printf("\nTrama I de controlo enviada!\n");
        alarmFlag = FALSE;
        numRetry = 0;
        return seqN;
    }
```

```
}

unsigned char *generateDataPackage(unsigned char *buffer, int *size, int n, int l1,
int l2)
{
    unsigned char dataPackage[*size];

    dataPackage[0] = C_DATA;
    dataPackage[1] = n;
    dataPackage[2] = l1;
    dataPackage[3] = l2;

    int counter = 4;
    int dataSize = 4;
    //ultima trama

    if (n == l1)
    {
        for (int i = n * 256; i < n * 256 + l2; i++)
        {
            dataPackage[counter++] = buffer[i];
        }
        dataSize += l2;
    }
    else
    {
        for (int i = n * 256; i < n * 256 + 256; i++)
        {
            dataPackage[counter++] = buffer[i];
        }
        dataSize += 256;
    }

    int cnt = 0;
    unsigned char *dp = malloc((4 + dataSize) * sizeof(unsigned char));

    for (int i = 0; i < 4 + dataSize; i++)
    {
        dp[cnt++] = dataPackage[i];
    }

    *size = dataSize + 1;

    return dp;
}
```

```
void closeConnection(int fd)
{
    do
    {
        sendControlMsg(fd, A_TRM, C_SET);
        printf("\nTrama DISC enviada\n");

        alarmFlag = FALSE;
        alarm(TIMEOUT);

        int *size = malloc(sizeof(int));
        stateMachine(fd, A_REC, C_DISC, S, size);
        printf("Trama DISC recebida!\n");

        sendControlMsg(fd, A_REC, C_UA);
        printf("Trama UA enviada!\n");

    } while (alarmFlag && numRetry < MAX_RETRY);

    if (alarmFlag && numRetry == MAX_RETRY)
        return;

    else
    {
        printf("Connection closed!\n");
        alarmFlag = FALSE;
        numRetry = 0;
        return;
    }
}
```

utils.h

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

```c
#define I 1
#define S 0

#define FLAG 0x7E
#define A_TRM 0x03
#define A_REC 0x01
#define C_SET 0x03
#define C_UA 0x07
#define C_DISC 0x0B

//Stuffing macros
#define ESCAPEMENT 0x7D
#define REPLACE_FLAG 0x5E
#define REPLACE_ESCAPEMENT 0x5D

#define MAX_RETRY 5
#define MAX_SIZE 1024

#define BAUDRATE B38400
#define _POSIX_SOURCE 1 /* POSIX compliant source */

#define NULL ((void *)0)

struct termios oldtio, newtio;

typedef enum
{
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    BCC_OK,
    STOP
} State_Machine;

typedef struct
{
    unsigned char *filename;
    int size;
} fileInfo;

/**
 * @brief Get file size
 *
 * @param file file descriptor
 * @return file size
```

```
 */
int getFileSize(FILE *file);

/**
 * @brief Send Control Message
 *
 * @param fd serial port descriptor
 * @param header header (A)
 * @param controlField control field (C)
 */
void sendControlMsg(int fd, unsigned char header, unsigned char controlField);

/**
 * @brief Message State Machine
 *
 * @param fd serial port descriptor
 * @param header header (A)
 * @param controlField control field (C)
 * @param type I or S
 * @param size size of message
 * @return pointer to data from message
 */
unsigned char *stateMachine(int fd, unsigned char header, char controlField, int
type, int *size);

/**
 * @brief Stuff Data
 *
 * @param buffer buffer to be stuffed
 * @param size size of buffer
 * @return stuffed data
 */
unsigned char *stuffingData(unsigned char *buffer, int *size);

/**
 * @brief Destuff Data
 *
 * @param buffer buffer to be destuffed
 * @param size size of buffer
 * @return destuffed data
 */
unsigned char *destuffingData(unsigned char *buffer, int *size);
```

```
/**
 * @brief Calculate BCC2
 *
 * @param buffer buffer to calculate bcc
 * @param size size of buffer
 * @return bcc2 value
 */
unsigned char calculateBCC2(const unsigned char *buffer, unsigned int size);
```

utils.c

```
#include "utils.h"

extern int alarmFlag;

int getFileSize(FILE *file)
{
    // saving current position
    long int currentPosition = ftell(file);

    // seeking end of file
    if (fseek(file, 0, SEEK_END) == -1){
        printf("ERROR: Could not get file size.\n");
        return -1;
    }

    // saving file size
    long int size = ftell(file);

    // seeking to the previously saved position
    fseek(file, 0, currentPosition);

    // returning size
    return size;
}

void sendControlMsg(int fd, unsigned char header, unsigned char controlField)
{
    unsigned char msg[5];
    msg[0] = FLAG;
    msg[1] = header;
    msg[2] = controlField;
    msg[3] = (A_TRM ^ controlField);
    msg[4] = FLAG;
```

```
    write(fd, msg, 5);
}

unsigned char *stateMachine(int fd, unsigned char header, char controlField, int
type, int *size)
{

    State_Machine state = START;
    unsigned char *message = malloc(sizeof(unsigned char) * MAX_SIZE);
    unsigned char c;
    int counter = 0;
    int seqN = 0;
    unsigned char *res = malloc(sizeof(unsigned char));
    res[0] = 0x3;

    while (state != STOP && !alarmFlag)
    {
        read(fd, &c, 1);

        switch (state)
        {
        case START:
            if (c == FLAG)
            {
                state = FLAG_RCV;
            }
            break;
        case FLAG_RCV:
            counter = 0;
            if (c == header)
            {
                state = A_RCV;
            }
            else{
                if (c == FLAG)
                    state = FLAG_RCV;
                else
                    state = STOP;
            }
            break;
        case A_RCV:
            if (type == S)
            {
                //ACK
                if (c == controlField || c == 0x85 || c == 0x05){
                    state = C_RCV;
```

```
                    res[0] = 0x0;
                }
                else
                {
                    if (c == FLAG)
                    {
                        state = FLAG_RCV;
                        res[0] = 0x1;
                        return res;
                    }
                    //REJ
                    // else if (c == 0x81 || c == 0x01)
                    // {
                    //     res[0] = 0x1;
                    //     return res;
                    // }
                    else{
                        res[0] = 0x0;
                        return res;
                    }
                }
            }
            else if (type == I){
                if (c == 0x00){
                    controlField = 0x00;
                    seqN = 0;
                    state = C_RCV;
                }
                else if (c == 0x40){
                    seqN = 1;
                    state = C_RCV;
                    controlField = 0x40;
                }
                else{
                    if (c == FLAG)
                        state = FLAG_RCV;
                    else
                        state = START;
                }
            }

            break;
        case C_RCV:
            if (c == (A_TRM ^ controlField) || c == (A_TRM ^ 0x05) || c == (A_TRM
^ 0x85)) //BCC = A_TRM ^ C
            {
```

```
            state = BCC_OK;
        }
        else
            state = START;
        break;
    case BCC_OK:
        if (c == FLAG)
        {
            if (type == I)
            {

                *size = counter;

                message = destuffingData(message, size);

                unsigned char bcc2 = message[*size - 1];

                int sizeBcc = *size - 1;
                unsigned char calcBcc2 = calculateBCC2(message, sizeBcc);

                unsigned char positiveACK; // R0000101 -> 0 ou 1
                unsigned char negativeACK; // R0000001 -> 0 ou 1
                if (bcc2 == calcBcc2)
                {

                    if (seqN == 0)
                    {
                        positiveACK = 0x05;
                        //     negativeACK = 0x01;
                    }
                    else
                    {
                        positiveACK = 0x85;
                        //     negativeACK = 0x81;
                    }
                    /*  if (counter <= 255)
                    {
                        sendControlMsg(fd, A_TRM, negativeACK);
                    }
                    else
                    {*/
                    sendControlMsg(fd, A_TRM, positiveACK);
                    //}
                }
                // else
                // {
```

```
                //      if (seqN == 0)
                //      {
                //          negativeACK = 0x01;
                //      }
                //      else
                //      {
                //          negativeACK = 0x81;
                //      }
                //      sendControlMsg(fd, A_TRM, negativeACK);
                //      printf("Enviei REJ\n");
                // }
            }
            state = STOP;
        }
        else{
            if (type == S){
                state = START;
            }
            else{
                message[counter++] = c;
                if (counter == MAX_SIZE)
                {
                    counter = 0;
                    state = START;
                    free(message);
                }
            }
        }
        break;
    default:
        break;
    }
}

if (type == I){
    unsigned char *data = malloc(sizeof(unsigned char) * (*size));
    for (int i = 0; i < (*size); i++){
        data[i] = message[i];
    }
    return data;
}
else
{
    return res;
}
```

```
}

unsigned char calculateBCC2(const unsigned char *buffer, unsigned int size)
{
    unsigned char bcc2 = 0;

    for (unsigned int i = 0; i < size; i++)
    {
        bcc2 ^= buffer[i];
    }
    return bcc2;
}

int calculateStuffedSize(unsigned char *buffer, int size)
{
    int counter = 0;

    for (int i = 0; i < size; i++)
    {
        if (buffer[i] == 0x7e)
        {
            counter++;
        }
        else if (buffer[i] == 0x7d)
        {
            counter++;
        }
        counter++;
    }
    return counter;
}

unsigned char *stuffingData(unsigned char *buffer, int *size)
{
    int startStuffedSize = *size;
    if (*size < MAX_SIZE)
        startStuffedSize = MAX_SIZE;

    int counter = 0;
    unsigned char stuffedBuffer[startStuffedSize];

    for (int i = 0; i < (*size); i++)
    {
        if (buffer[i] == FLAG)
        {
            stuffedBuffer[counter++] = ESCAPEMENT;
```

```
            stuffedBuffer[counter++] = REPLACE_FLAG;
        }
        else if (buffer[i] == ESCAPEMENT)
        {
            stuffedBuffer[counter++] = ESCAPEMENT;
            stuffedBuffer[counter++] = REPLACE_ESCAPEMENT;
        }
        else
        {
            stuffedBuffer[counter++] = buffer[i];
        }
    }

    *size = counter;

    unsigned char *sb = malloc(sizeof(unsigned char) * (counter));
    for (int i = 0; i < counter; i++)
    {
        sb[i] = stuffedBuffer[i];
    }

    return sb;
}

unsigned char *destuffingData(unsigned char *buffer, int *size)
{
    int counter = 0;
    unsigned char destuffedData[MAX_SIZE];

    for (int i = 0; i < (*size); i++)
    {
        if (buffer[i] == 0x7d)
        {
            if (buffer[i + 1] == 0x5e)
            {
                destuffedData[counter++] = 0x7e;
            }
            else if (buffer[i + 1] == 0x5d)
            {
                destuffedData[counter++] = 0x7d;
            }
            i++;
        }
        else
        {
            destuffedData[counter++] = buffer[i];
```

```
        }
    }

    unsigned char *db = malloc(sizeof(unsigned char) * counter);

    for (int j = 0; j < counter; j++)
    {
        db[j] = destuffedData[j];
    }

    *size = counter;

    return db;
}
```