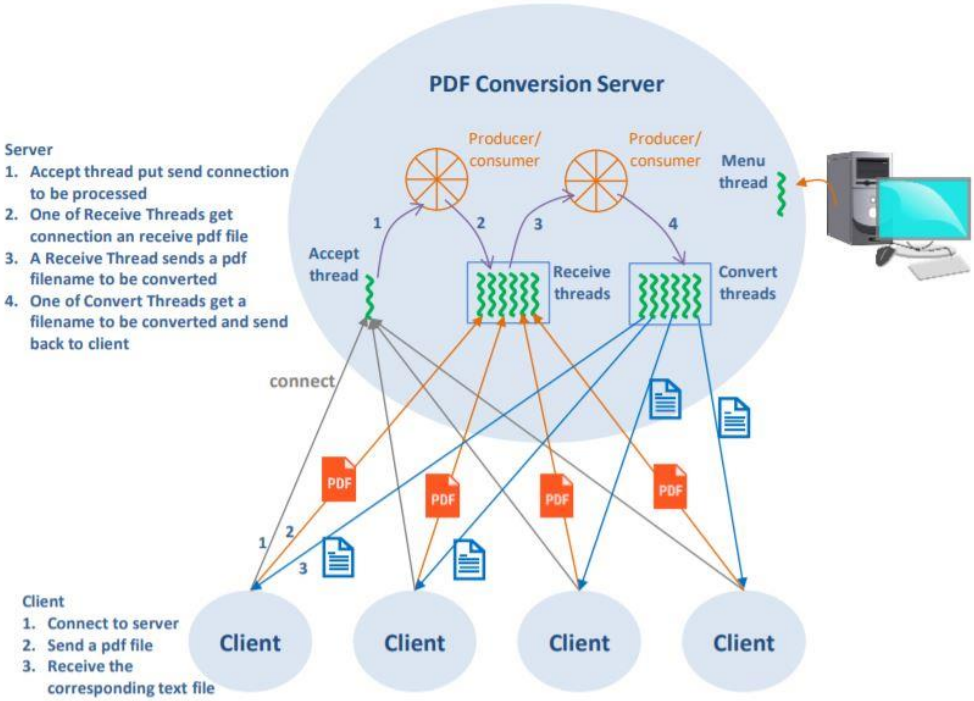
Sistemas Operativos

Licenciatura em Engenharia Informática, Redes e Telecomunicações

# 3º Trabalho de Sistemas Operativos





**Grupo 7**

43858 Mafalda Gonçalves

43874 João Florentino

44124 Francisco Soares


2º Semestre letivo 2018/2019

29 de maio de 2018

# Resumo

Este trabalho incide sobre a conceção de programas baseados no paradigma cliente/servidor utilizando *sockets* como mecanismo de comunicação entre processos e programas concorrentes com base em múltiplas tarefas.

O trabalho tem como foco principal a familiarização com o ambiente UNIX/LINUX; o sincronismo entre múltiplas tarefas em POSIX; a utilização de sinais UNIX; a construção de bibliotecas dinâmicas para o sistema UNIX e a consolidação da programação ao nível de sistema.

# Índice

# Lista de Figuras

# 1. Introdução

Este trabalho tem por base o trabalho anterior (2º trabalho) e pretende-se a evolução e consolidação do seu desenvolvimento segundo as propostas apresentadas no enunciado. A arquitetura desta nova versão do processo servidor é apresentada, esquematicamente, na Figura 1.

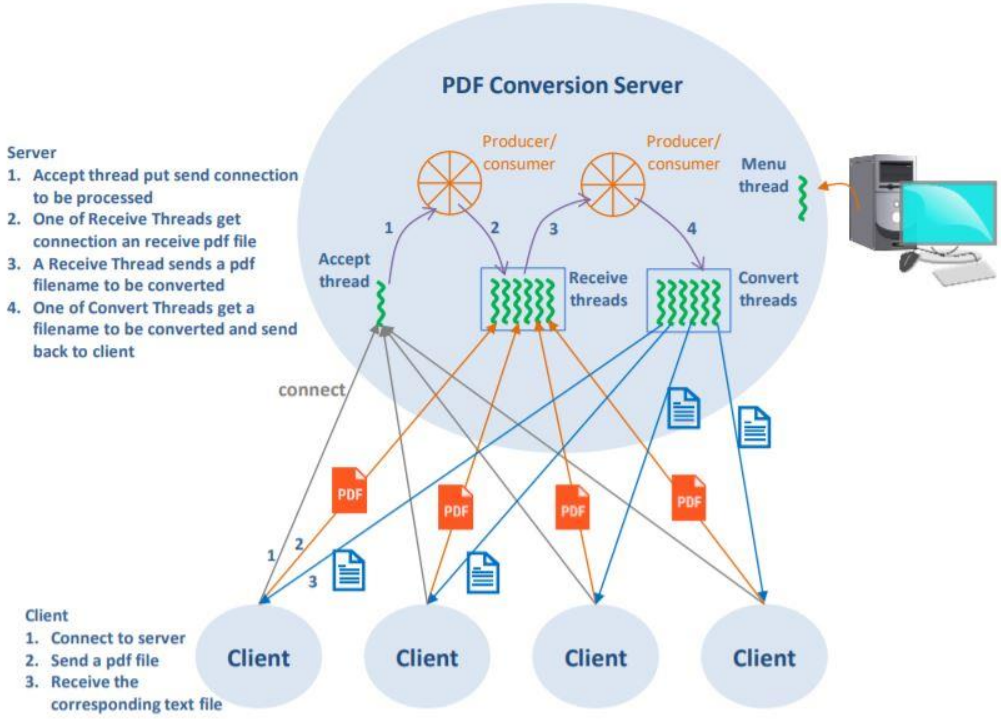



**Figura 1 - Arquitetura do Processo Servidor**

## 1.1 Ferramentas utilizadas para o desenvolvimento do trabalho

O trabalho desenvolvido e aqui descrito foi elaborado em ambiente UNIX/LINUX, com a ferramenta de desenvolvimento Eclipse, na máquina virtual disponibilizada na plataforma Moodle pelo professor docente da disciplina.

Recorremos à utilização da ferramenta Microsoft Office Word para a realização deste relatório.

# 2. Formulação dos problemas do trabalho

## 2.1 Entidades

As entidades presentes na figura acima apresentada são: os clientes responsáveis pela transmissão de ficheiros do tipo .pdf destinados a conversão; o servidor que irá realizar uma série de ações para ter a certeza de que os ficheiros são convertidos com sucesso e enviá-los em formato .txt de volta para os clientes.

## 2.2 Estruturas de dados e Interação com entidades

A entidade "Servidor" engloba outras entidades, nomeadamente a *threadAccept*, a *threadMenu*, a *receiveFile* e a *convertFile*. Esta entidade é responsável pela manutenção de toda a informação relacionada com o serviço, permitindo acessos simultâneos por parte dos clientes e sincronizando adequadamente as ações, por forma a evitar bloqueios desnecessários e garantindo que o sistema esteja isento de anomalias relacionadas com a concorrência.

Do ponto de vista do Servidor, começa-se por criar um *socket* que apoia IPv4 (AF_INET) e comunicação fiável *full-duplex* (SOCK_STREAM) através da função *socket()*. De seguida, atribui-se a este *socket* um endereço local e o DEFAULT_PORT, neste código especificado como 8000, através da função *bind()*. Depois ativa-se o *socket* com uma fila de espera de tamanho 5 através da função *listen()*.

O *threadAccept* (***Accept Threads***) é a tarefa responsável pela receção das ligações dos processos clientes. Esta tarefa passa a depositar cada nova ligação numa fila de ligações (Produtor/Consumidor). Estas ligações serão processadas por um conjunto de tarefas (*receiveFile*), previamente criadas e que ficam responsáveis por receberem os ficheiros a converter.

Prossegue-se para a leitura do conteúdo do ficheiro do *socket* e escrita no novo ficheiro. A escrita termina assim que o *size* do ficheiro é atingido, ou seja, quando se chega ao final do conteúdo do ficheiro. Faz-se também o incremento do número total de ficheiros convertidos assim que se termina a escrita no novo ficheiro.

O *receiveFile* (***Receive Threads***), após receber um ficheiro, coloca a indicação de existência de um novo ficheiro numa outra fila de ficheiros a imprimir (Produtor/Consumidor) e aguarda a existência de nova ligação para receberem novo ficheiro.

Começa-se por realizar um *Get* para receber o *socket* respetivo de um cliente e realiza um Put para passar o *socket* da conexão anterior ao *threadConvert*.

O *convertFile (**Convert Threads**)* é responsável pela obtenção de um ficheiro da respetiva fila (Produtor/Consumidor) e por proceder à sua conversão para o formato de texto. Após finalizada a conversão envia o ficheiro ao cliente e espera por novo ficheiro na fila de ficheiros a converter.

Para tal, começa-se por realizar um Get para obter do *socket* o ficheiro em fila de espera e é iniciado o processo de conversão com a função "process_init_comand" realizada nos trabalhos anteriores, redireciona-se o processo para o *socket* do cliente e inicia-se a sua execução. Incrementa-se ainda o número de conversões e a soma total do tamanho de ficheiros convertidos após as chamadas das funções anteriormente referidas.

Depois, recorre-se à função *strtok* para retirar a extensão ".pdf" e envia-se o ficheiro .txt de volta para o cliente.

A outra tarefa é a tarefa *threadMenu (**Menu thread**)* que é responsável pelo processamento da interface (menu de texto em consola) com o utilizador. Esta interface permite ao utilizador monitorizar o estado do servidor através da apresentação de informações sobre o número de ficheiros processados, dimensão média dos ficheiros convertidos, etc. Para a sua implementação, são mostrados na consola os valores que vão sendo atualizados na tarefa *threadAccept*, fazendo um *while* infinito que imprime na consola o número total de ficheiros convertidos, o tamanho total dos ficheiros convertidos e o tamanho médio dos ficheiros convertidos.

Na perspetiva dos processos clientes, começa-se por ler o número de clientes que pretende converter ficheiros PDF em ficheiros de texto da consola e somar ao número total de ficheiros o valor equivalente ao número de clientes que estabelecem ligação com o servidor.

Os clientes começam por estabelecer uma ligação TCP com o servidor e, para tal, começa-se por verificar se os argumentos passados são válidos e têm o formato: "%s <host> <port_number>\n".

Seguidamente, determina-se o endereço IP do servidor, abre-se um *socket* TCP, preenche-se a estrutura do endereço do servidor criada inicialmente com o endereço do servidor que se pretende contactar e estabelece-se a ligação com esse servidor.

De seguida, aloca-se espaço na memória para guardar o nome do ficheiro .pdf que se pretende converter, sendo que o nome do ficheiro é passado pela linha de comandos. Abre-se o ficheiro e preenche-se a estrutura que contém as informações relativas ao ficheiro especificado no *file descriptor*.

Após este passo, prossegue-se para a leitura do ficheiro, caractere a caractere, e escrita no *socket* para que o seu conteúdo seja enviado para o servidor de conversão. Depois, recorre-se à função *strtok* para retirar a extensão ".pdf" e cria-se um ficheiro .txt na diretoria "./txt_files/%s.txt": caso já exista um ficheiro com o mesmo nome na mesma diretoria, este será reescrito pelo novo ficheiro, caso contrário, é criado um novo.

Nesta fase é feita a leitura do *socket*, caractere a caractere, e escrita do seu conteúdo no novo ficheiro.

Por fim, é terminada a sua execução através dos *closes*.

# 3. Conclusão

Neste último trabalho foram implementadas técnicas de sincronismo que permitiram que as *threads* utilizadas no desenvolvimento do trabalho pudessem comunicar entre si. Desta forma, as *threads* podem passar informação entre si a partir de *"Produtores\Consumidores"* e *"Leitores\Escritores"*. Foi interessante aplicar a matéria de *forks*, *pipes*, *threads*, *sockets* e sincronismo no âmbito de um projeto e programa únicos e visualizar como se afetavam uns aos outros ao longo dos testes que realizávamos.

# 4. Anexo – Código Completo

**Figura 2 - Código Desenvolvido (Servidor)**

```
#ifdef STDC_ALLOC_LIB
#define STDC_WANT_LIB_EXT2 1
#else
#define _POSIX_C_SOURCE 200809L
#endif

#include "myinet.h"
#include "errorUtils.h"
#include "include.h"
#include "process_t.h"
#include "sharedBuffer.h"

#define NTHREADS 5

typedef struct menu{
    int convertedFiles;
    int totalSizeConvFiles;
    int avgSizeConvFiles;
} Menu;

typedef struct sharedBuffers{
    SharedBuffer *sBufferReceive;
    SharedBuffer *sBufferConvert;
} SharedBuffers;

typedef struct threadArgs{
    int clientID;
    char *filename;
    int fileSize;
    int socket;
    Menu *menu;
    SharedBuffers *SharedBuffer;
} ArgsThread;

void sigintHandler(int sig_num)
{
    /* Reset handler to catch SIGINT next time.
       Refer http://en.cppreference.com/w/c/program/signal */
    signal(SIGINT, sigintHandler);
    printf("\n Cannot be terminated using Ctrl+C \n");
    fflush(stdout);
}
```

```
int readLine(int socket, char *buffer, int size){
    int i;
    for(i = 0; i < size; i++) {
        read(socket, &buffer[i], sizeof(char));
        if(buffer[i] == '\n') break;
    }
    buffer[i] = '\0';
    return i;
}

void *threadMenu(void *_args){
    ArgsThread *threadArgs = (ArgsThread *) _args;

    int nConnections = 1;

    threadArgs->menu->convertedFiles = 0;
    threadArgs->menu->totalSizeConvFiles = 0;

    while(1){
        if(nConnections == threadArgs->menu->convertedFiles){
            printf("\n_____________MENU_____________\n");
            printf("Total de ficheiros convertidos: %d \n", threadArgs-
>menu->convertedFiles);
            printf("Tamanho total dos ficheiros convertidos: %d \n",
threadArgs->menu->totalSizeConvFiles);
            threadArgs->menu->avgSizeConvFiles = threadArgs->menu-
>totalSizeConvFiles/threadArgs->menu->convertedFiles;
            printf("Tamanho médio dos ficheiros convertidos: %d \n",
threadArgs->menu->avgSizeConvFiles);
            printf("\n À espera de uma ligação... \n");
            nConnections++;
        }
    }
}

void threadAccept(int socket, char *file_name, Menu *menu){

    char *filename = malloc(100);   // nome do ficheiro
    char *size = malloc(100);       // tamanho do ficheiro
    char *aux = malloc(100) ;       // \n de terminação do cabeçalho

    readLine(socket, filename, 100);
    readLine(socket, size, 100);
    readLine(socket, aux, 100);

    int filesize = atoi(size);  // dimensão do ficheiro

    printf("Nome do ficheiro: %s\n", filename);
    printf("Tamanho do ficheiro: %d\n", filesize);
```

```
    char path[DIM_BUFFER];
    sprintf(path, "./pdf_files/%s", filename);

    // abre o ficheiro PDF
    int fd = open(path, O_CREAT | O_WRONLY | O_TRUNC, 0644);
    if(fd  < 0) {
        perror("Erro a abrir o ficheiro ");
        exit(-1);
    }

    // Ler do socket para um ficheiro
    char buffer[DIM_BUFFER+1];  // reserva um byte extra para o char de
terminação
    int n_bytes;
    int fileContent = 0;
    while ( (n_bytes = read(socket, buffer, sizeof(buffer))) != 0 ) {
//leitura caractere a caractere
        if ( n_bytes < 0 )
            FatalErrorSystem("Erro na leitura do socket");
        if (write(fd, buffer, n_bytes) <  0)
            FatalErrorSystem("Erro na escrita do ficheiro");
        fileContent += n_bytes;
        if(fileContent == filesize) {
            break;
        }
    }
    close(fd);
    free(aux);
    free(size);
    free(filename);

    menu->totalSizeConvFiles += filesize; // soma do size do ficheiro ao
total
}

void *receiveFile(void *_sBuffer){
    SharedBuffers *sBuffer = (SharedBuffers *)_sBuffer;
    while(1){
        void *fileInfo = sharedBuffer_Get(sBuffer->sBufferReceive);
        ArgsThread *threadArgs = (ArgsThread *) fileInfo;
        threadAccept(threadArgs->socket, threadArgs->filename,
threadArgs->menu);
        sharedBuffer_Put(threadArgs->SharedBuffer->sBufferConvert,
threadArgs);
    }
    return NULL;
}
```

```c
void *convertFile(void *_sBuffer){
    SharedBuffers *sBuffer = (SharedBuffers *)_sBuffer;
    while(1){
        void *args = sharedBuffer_Get(sBuffer->sBufferConvert);
        ArgsThread *_threadArgs = (ArgsThread *) args;

        char path[64];
        sprintf(path,"./ficheiros/%s", _threadArgs->filename);

        char *cmd[] = {"pdftotext", "-layout", path, "-", NULL};
        process_t *process = process_init_command(cmd); //iniciar o
processo

        // redirecionar o processo para o socket do cliente
        process_set_stdout_to_file(process, _threadArgs->socket);

        // ficheiro para converter
        process_start_execution(process);

        _threadArgs->menu->convertedFiles++;   // incremento do nº de
conversões

        char *filenameTxt = strtok(_threadArgs->filename, ".");

        close(_threadArgs->socket);

        printf("O ficheiro %s.txt foi enviado para o cliente com
sucesso.\n", filenameTxt);
    }

    return NULL;
}


int main(int argc, char * argv[]) {
    int                 sockfd;
    int                 newsockfd;
    struct sockaddr_in serv_addr;
    struct sockaddr_in client_addr;
    socklen_t           clientSize;
    pthread_t threadReceive[25];
    pthread_t threadConvert[25];
    unsigned int        serverPort = DEFAULT_PORT;

    printf("Programa de teste dos SOCKETS (server TCP) ...\n\n");

    if (argc == 2) {
        serverPort = atoi(argv[1]);
    }
```

```
    else if (argc != 1) {
        printf("Argumentos inválidos.\nUse: %s <port_number>\n",
argv[0]);
        exit(EXIT_FAILURE);
    }

    if ( (serverPort < 1) || (serverPort > 65536) ) {
        printf("O porto deve estar entre 1 e 65536\n");
        exit(EXIT_FAILURE);
    }

    printf("O servidor vai registar-se no porto: %d\n", serverPort);

    /* Criar socket TCP */
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
        FatalErrorSystem("Erro ao pedir o descritor");

    /* Registar o endereço local de modo a que os clientes possam
contactar com o servidor */
    memset( (char*)&serv_addr, 0, sizeof(serv_addr) );
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port        = htons(serverPort);

    /* Criar os SharedBuffers para receber e converter os ficheiros */
    SharedBuffer *sBufferRec = malloc(sizeof(*sBufferRec));
    SharedBuffer *sBufferConv = malloc(sizeof(*sBufferConv));

    sharedBuffer_init(sBufferRec,15);
    sharedBuffer_init(sBufferConv,15);

    if ( bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) <
0 )
        FatalErrorSystem("Erro ao efetuar o bind");

    /* Ativar socket com fila de espera de dimensão 5 */
    if (listen( sockfd, NTHREADS) < 0 ){
        FatalErrorSystem("Erro no listen");
    }

    ArgsThread *threadArgs = malloc(sizeof(ArgsThread));

    pthread_t menuThread;
    int m = pthread_create(&menuThread, NULL, threadMenu, threadArgs);
    if (m != 0)
        FatalErrorSystem("Erro na criação do menu");

    /* Criar a estrutura SharedBuffers */
    SharedBuffers *sBuffer = malloc(sizeof(SharedBuffers));
```

```
    sBuffer->sBufferConvert = malloc(sizeof(SharedBuffers));
    sBuffer->sBufferReceive = malloc(sizeof(SharedBuffers));
    sBuffer->sBufferConvert = sBufferConv;
    sBuffer->sBufferReceive = sBufferRec;

    /* Criar as threads para receber o ficheiro */
    for (int IDThread = 0; IDThread < NTHREADS; IDThread++) {
        printf("O ID da thread que recebe o ficheiro é: %d \n",
IDThread);
        pthread_create( &threadReceive[IDThread], NULL, receiveFile,
sBuffer);
    }

    //Criar aas threads para converter o ficheiro
    for (int IDThread = 0; IDThread < NTHREADS; IDThread++) {
        printf("O ID da thread que converte o ficheiro é: %d \n",
IDThread);
        pthread_create( &threadConvert[IDThread], NULL, convertFile,
sBuffer);
    }

    printf("À espera de uma ligação...\n");

    signal(SIGINT, sigintHandler);

    while(1) {
        clientSize = sizeof( client_addr );
        newsockfd = accept(sockfd, (struct sockaddr *)(&client_addr),
&clientSize);
        if ( newsockfd < 0 )
            FatalErrorSystem("Erro ao efetuar o accept");

        printf("Ligação estabelecida\n");

        threadArgs->SharedBuffer = malloc(sizeof(SharedBuffers));
        threadArgs->SharedBuffer = sBuffer;
        threadArgs->filename = malloc(64);
        threadArgs->socket = newsockfd;
        threadArgs->clientID++;

        pthread_t clientThread;
        pthread_create(&clientThread, NULL, convertFile, threadArgs);  //
inicializar a thread

        pthread_detach(clientThread);
    }

    pthread_detach(menuThread);
    sharedBuffer_destroy(sBufferConv);
```

```
    sharedBuffer_destroy(sBufferRec);
    close(sockfd);
    return 0;
}
```

**Figura 3 - Código Desenvolvido (Cliente)**

```
#include "myinet.h"
#include "errorUtils.h"
#include "include.h"


int main(int argc, char * argv[])  {

    int clients;
    printf("Número de clientes: ");
    scanf("%d", &clients);
    int totalFiles = 0;
    for(int i = 1; i <= clients; i++) {
        totalFiles++;

        int                 sockfd;
        struct sockaddr_in  serv_addr;
        struct hostent     *phe;
        in_addr_t           serverAddress;
        char               *serverName = DEFAULT_HOST;
        unsigned int        serverPort = DEFAULT_PORT;

        printf("Programa de teste dos SOCKETS (cliente TCP) ...\n\n");

        if (argc == 3) {
            serverName = argv[1];
            serverPort = atoi(argv[2]);
        }
        else if (argc != 1) {
            printf("Argumentos inválidos.\nUse: %s <host>
<port_number>\n", argv[0]);
            exit(EXIT_FAILURE);
        }

        /* Determinar o endereço IP do servidor */
        if ((phe = gethostbyname(serverName)) != NULL)
            memcpy(&serverAddress, phe->h_addr_list[0], phe->h_length);
        else
            if ( (serverAddress = inet_addr(serverName)) == -1)
            FatalErrorSystem("Impossível determinar endereço IP para a
máquina \"%s\"",serverName);
```

```
        if ( (serverPort < 1) || (serverPort > 65536) ) {
            printf("O porto deve estar entre 1 e 65536\n");
            exit(EXIT_FAILURE);
        }


        /* Abrir um socket TCP (an Internet Stream socket) */
        if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
            FatalErrorSystem("Erro ao pedir o descritor");


        /* Preencher a estrutura serv_addr com o endereço do servidor que
pretendemos contactar */
        memset((char*)&serv_addr, 0, sizeof(serv_addr));
        serv_addr.sin_family      = AF_INET;
        serv_addr.sin_addr.s_addr = serverAddress;
        serv_addr.sin_port        = htons(serverPort);

        printf("O cliente vai ligar-se ao servidor na máquina %s:%d\n",
serverName, serverPort);
        printf("IP: %s\n", inet_ntoa(serv_addr.sin_addr));


        /* Ligar-se ao servidor */
        if ( connect(sockfd, (struct sockaddr *)&serv_addr,
sizeof(serv_addr)) < 0 )
            FatalErrorSystem("Falha na ligação");

        printf("Ligação estabelecida...\n");

        char *filename;
        filename = malloc(30);
        printf("Nome do ficheiro: ");
        scanf("%s", filename);

        int fd = open(filename, O_RDONLY);

        if(fd < 0)
            perror("Erro na abertura do ficheiro");

        struct stat st;
        fstat(fd, &st);   // preenche a estrutura com as informações
sobre o ficheiro especificado no file descriptor

        int size = st.st_size;  // tamanho do ficheiro
        char header[DIM_BUFFER];    // filename; size; /n

        sprintf(header, "%s\n%d\n\n", filename, size);
```

```
        write(sockfd, header, strlen(header));  // escrever o cabeçalho
no socket

        // Ler o ficheiro e escrevê-lo no socket e enviá-lo para o
servidor
        char buffer[DIM_BUFFER+1];    // reserva um byte extra para o
char de terminação
        int n_bytes;
        int fileContent = 0;
        while ( (n_bytes = read(fd, buffer, sizeof(buffer))) != 0 ) {
// leitura caractere a caractere
            if ( n_bytes < 0 )
                FatalErrorSystem("Erro na leitura do ficheiro");
            if (write(sockfd, buffer, n_bytes) <  0)
                FatalErrorSystem("Erro na escrita no socket");
            fileContent+= n_bytes;
            if(fileContent == size)
                break;
        }
        close(fd);
        printf("Envio do ficheiro para o servidor\n");

        // retirar o .pdf
        char path[DIM_BUFFER];
        char *filenametxt = strtok(filename, ".");  // elimina o "." no
filename

        sprintf(path, "./txt_files/%s.txt", filenametxt);

        // Criar um novo ficheiro na diretoria
        // Caso já exista um ficheiro com o mesmo nome, reescreve esse
ficheiro com o novo
        int fdtxt = open(path, O_CREAT | O_WRONLY | O_TRUNC, 0644);
        if(fdtxt  < 0) {
            perror("Erro a abrir o ficheiro ");
            exit(-1);
        }

        // Ler o socket e escrever no ficheiro o que está nele, mandar
para a pasta
        char buffertxt[DIM_BUFFER+1];   // reserva um byte extra para o
char de terminação
        int n_bytestxt;
        int fileContenttxt = 0;
        while ( (n_bytestxt = read(sockfd, buffertxt, sizeof(buffertxt)))
!= 0 ) { //leitura caractere a caractere
            if ( n_bytestxt < 0 )
                FatalErrorSystem("Erro na leitura do Socket");
```

```
            if (write(fdtxt, buffertxt, n_bytestxt) <  0)
                FatalErrorSystem("Erro no Write do Ficheiro");
            fileContenttxt += n_bytestxt;
            if(fileContenttxt == size){
                break;
            }
        }
        close(fdtxt);

        printf("O ficheiro .txt foi criado e recebido com sucesso\n");
        free(filename);

        close(sockfd);
        return EXIT_SUCCESS;
        printf("Número total de ficheiros : %d", totalFiles);
    }
}
```

# 5. Referências


- http://pages.cs.wisc.edu/~remzi/OSTEP/ - último acesso 04-06-2019