U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Configuração e Estudo de uma rede

Redes de Computadores

Mestrado Integrado de Engenharia Informática e Computação

23 de dezembro de 2020

Daniel Garcia Silva, up201806524@fe.up.pt

Mariana Truta, up201806543@fe.up.pt

3ºano, Turma 3, Grupo 5

Índice

Sumário

No âmbito da unidade curricular de *Redes de Computadores*, foi elaborado um projeto que consistia no desenvolvimento de uma aplicação de download, usando o protocolo *FTP*, bem como na configuração e estudo de uma rede.

Ao longo deste relatório, será explicado como foi possível desenvolver uma aplicação funcional e sem perdas de dados e como foi possível utilizá-la através de uma rede configurada por nós, nas aulas práticas.

Introdução

O projeto consiste em duas partes: na **primeira parte**, foi desenvolvida uma aplicação de *download*, na linguagem de programação *C*, de acordo com o protocolo *FTP*, onde se utilizou *sockets* para estabelecer ligações *TCP*; na **segunda parte**, pretendia-se que fosse configurada e analisada uma rede de computadores, através de duas *LANs* virtuais dentro de um *switch*, onde fosse possível executar a aplicação da primeira parte.

O objetivo deste relatório é expor, de forma organizada e coerente, a componente teórica que nos permitiu compreender todos os procedimentos necessários para a realização do projeto com sucesso. Divide-se, assim, da seguinte forma:

- **Parte 1 – Aplicação de *Download*:** descrição da arquitetura da aplicação de *download* e respetivos resultados;
- **Parte 2 - Análise e Configuração de Rede:** análise pormenorizada de cada experiência realizada bem como o estudo dos respetivos *logs*;
- **Conclusão:** síntese da informação apresentada nas secções anteriores e reflexão sobre os objetivos de aprendizagem alcançados.

Parte 1 – Aplicação de Download

Na primeira parte deste trabalho, foi desenvolvida uma aplicação de download, cujo objetivo é descarregar qualquer tipo de ficheiro de um servidor *FTP* para a máquina do cliente. Esta aplicação recebe apenas um argumento com uma sintaxe normalizada "*ftp://[<user>:<password>@]<host>/<url-path>"*, onde são indicados o protocolo de aplicação a utilizar (neste caso, será sempre *ftp*), opcionalmente o *username* e a respetiva *password*, o nome do computador (*host*) na *ethernet* que irá corresponder a um endereço *ip* e o endereço (*path*) do ficheiro dentro desse mesmo computador.

Arquitetura da aplicação de Download

Inicialmente, na função *parse_file_url()*, após a leitura do argumento, toda a sua informação é guardada numa *struct, url_struct,* que irá armazenar todos os dados

essenciais para a realização do *download* do ficheiro pedido. É necessário, de seguida, obter o endereço *ip* do servidor através do nome do computador, na função *getIpAddress(), fornecida pelos docentes*.

Antes de realmente se explicar os passos do *download* de um ficheiro, é necessário mencionar três funções que permitem a troca de mensagens entre servidor e cliente: *read_response()* que permite ler uma mensagem enviada pelo servidor, retornando, em caso de sucesso, o código da resposta e guardando num *char** toda a mensagem recebida que começa com esse código; *send_command()* que envia para o servidor um comando passado como argumento; *send_command_receive_response()* que, utilizando as duas funções referidas anteriormente, envia um comando para o servidor, recebe uma resposta e confirma se o código da resposta corresponde ao esperado.

Assim, com o auxílio da função *ftp_connect()*, começa-se por estabelecer uma conexão *TCP* onde se abre um *socket TCP* que irá permitir a comunicação entre o servidor e o cliente, usando a porta 21 que é a porta *default* deste protocolo. Para notificar o cliente de que está preparado para receber novos comandos, o servidor envia uma mensagem com o código 220. Posteriormente, na função *ftp_login()*, é necessário fazer *login* do utilizador com o *username* e *password* indicados no *url*, se estiverem presentes, caso contrário, envia-se o *username* "*anonymous*" e uma *password* qualquer.

Após o servidor enviar uma mensagem a indicar que o login foi feito com sucesso, usando a função *ftp_passive_mode()*, envia-se o comando "*pasv*" que possibilita a entrada em modo passivo. A resposta a este pedido permite que seja aberta uma nova porta para ser usada unicamente para a transferência de dados com um novo *socket*.

Assim, reúnem-se as condições necessárias para o *download* do ficheiro. Em *ftp_request_file()*, pede-se ao servidor para enviar o ficheiro pretendido com o comando "*retr path*". De seguida, com a ajuda da função *ftp_download_file()*, abre-se um novo ficheiro na máquina do cliente e, à medida que o servidor vai enviando os dados do ficheiro, estes serão escritos para o ficheiro criado, estando assim a ser realizado o download.

Finalmente, encerram-se as conexões criadas.

Resultados de um download de sucesso

Para testar o comportamento da aplicação desenvolvida, foram testadas diferentes condições, por exemplo, o *url* no formato incorreto, *login* inválido, ficheiro inexistente. Todos os resultados foram os esperados, sendo feito o download do ficheiro caso não haja nenhum erro ou, caso contrário, sendo enviada uma mensagem de erro para o cliente ter conhecimento do que causou o término do programa.

Parte 2 - Análise e Configuração de Rede

Experiência 1 – Configuração de um IP de rede

→ **O que são os pacotes *ARP* e para que são utilizados?**

O protocolo *ARP* (*Address Resolution Protocol*) serve para a conversão/resolução de endereços da camada de rede em endereços da camada de enlace. No caso concreto desta implementação, convertem-se endereços *IP* em endereços *MAC*. Isto é conseguido quando o computador que desconhece o endereço MAC correspondente a um endereço IP envia um pacote ARP de pedido em *broadcast*, sendo o endereço *MAC* de destino, no cabeçalho do pacote *Ethernet II*, preenchido com *Fs*. Se algum computador que tenha conhecimento do endereço *MAC* pedido for alcançado, envia um pacote *ARP* de resposta para o emissor.

→ **Quais são os endereços *MAC* e *IP* dos pacotes *ARP* e porquê?**

Numa trama *ARP*, existe, no total, quatro campos de endereços: endereço *MAC* do emissor, endereço *IP* do emissor, endereço *IP* do recetor e endereço *MAC* do recetor, sendo este último preenchido com zeros quando se trata de uma trama de pedido.

→ **Quais os pacotes gerados pelo comando *ping*?**

Se não tiver conhecimento do endereço *MAC*, o comando *ping* começa por gerar pacotes *ARP*. De seguida, gera pacotes do tipo ICMP (*Internet Control Message Protocol*) com o tipo 8 (*Echo request*), recebendo como resposta pacotes do mesmo género, mas com o tipo 0 (*Echo reply*).

→ **O que são os endereços *MAC* e *IP* dos pacotes de *ping*?**

O cabeçalho da trama *Ethernet II* contem o endereço *MAC* de destino e o endereço *MAC* de origem. No cabeçalho da trama *IPv4*, os *bytes* 12 a 15 contêm o endereço *IP* de origem e os *bytes* 16 a 19 contêm o endereço *IP* de destino.

→ **Como determinar se uma trama recebida de *Ethernet* é *ARP*, *IP* ou *ICMP*?**

É possível distinguir as tramas *Ethernet* através do campo do cabeçalho *EtherType*, que define o tipo de trama. Se este tiver o valor *0x0806*, corresponde a uma trama do tipo *ARP*. Se tiver o valor *0x0800*, corresponde a uma trama *IPv4*, sendo esta uma trama *ICMP* no caso do campo referente ao protocolo do cabeçalho da trama *IPv4* for *0x01*.

→ **Como determinar o tamanho de uma trama recebida?**

O cabeçalho de uma trama *Ethernet II* tem 14 *bytes* de comprimento.

Numa trama *ARP*:

- Os primeiros 5 campos têm um comprimento fixo, 8 bytes;

- O 3ª campo (*HLEN*) especifica o comprimento dos 6º e 8º campos;
- O 4º campo (*PLEN*) especifica o comprimento dos 7º e 9º campos;
- Comprimento = 14 + 8 + 2***HLEN* + 2***PLEN*.

Numa trama IPv4:

- O cabeçalho contêm o comprimento da trama *IPv4* no 5º campo (*TLEN*);
- Comprimento = 14 + *TLEN*.

→ **O que é a *interface loopback* e qual a sua importância?**

A *interface loopback* serve para testar a configuração, sendo esta uma *interface* virtual de rede que possibilita o computador de receber respostas de si mesmo. Ao enviar um pacote para si mesma (através do endereço de *loopback*, 172.0.0.0/8 em *IPv4* e ::1 em *IPv6*), uma máquina verifica se as suas ligações estão corretamente estruturadas.

Experiência 2 – Implementação de duas VLANs num switch

→ **Como configurar a *vlan10*?**

No *gkterm*, corre-se os seguintes comandos, sendo que *tux13* e *tux14* estão ligados às portas 3 e 4 do *switch*, respetivamente:

```
> configure terminal
> vlan 10                    //criar a vlan 10
> interface fastethernet 0/3     //adicionar o tux13 à vlan 10
> switchport mode access
> switchport access vlan 10
> exit
> interface fastethernet 0/4     //adicionar o tux14 à vlan 10
> switchport mode access
> switchport access vlan 10
> end
```

→ **Quantos domínios de *broadcast* existem? Como pode conclui-lo a partir dos *logs*?**

Existe dois domínios de *broadcast*, um para cada *vlan* que configuramos (172.16.10.255 para a *vlan10* e 172.16.11.255 para a *vlan11*). Os *logs* mostram que o *ping request* chega ao *tux14* quando é enviado a partir do *tux13*, mas não quando é enviado a partir do *tux2*.

Experiência 3 – Configuração de um Router em Linux

De forma a que fosse possível a comunicação entre *tux13* e *tux12*, *tux14* foi configurado como um *router* nesta experiência.

→ **Que rotas existem nos *tuxes*? Qual o seu significado?**

As rotas existentes nos diferentes *tuxes* são:

- ***tux13***: tem uma rota para a *vlan10* (172.16.10.0) pela *gateway* 172.16.60.1 e uma rota para a *vlan11* (172.16.11.0) pela *gateway* 172.16.10.254;
- ***tux14***: tem uma rota para a *vlan10* (172.16.10.0) pela *gateway* 172.16.10.254 e uma rota para a *vlan11* (172.16.11.0) pela *gateway* 172.16.11.253;
- ***tux12***: tem uma rota para a *vlan10* (172.16.10.0) pela *gateway* 172.16.11.1 e uma rota para a *vlan11* (172.16.11.0) pela *gateway* 172.16.11.1.

A utilidade das rotas é indicar ao computador qual é o destino dos pacotes a enviar.

→ **Que informações contem uma entrada da tabela de encaminhamento?**

Uma entrada da tabela de encaminhamento contém:

- ***Destination*** - gama de endereços *ip* de destino da rota (0.0.0.0 para *default*);
- ***Gateway*** - endereço *ip* de passagem (0.0.0.0 para a rota *self*);
- ***Genmask*** - máscara de endereço (0.0.0.0 para a rota default);
- ***Flags*** - informação sobre a rota: U (*up*, ativa) e G (tem de passar pelo *gateway*);
- ***Metric***: número de *hoops* até ao destino, isto é, o custo de cada rota;
- ***Ref***: número de referências para esta rota;
- ***Use***: contagem de pesquisas pela rota; se se usar *-C*, corresponde ao número de sucessos, se se usar *-F*, corresponde ao número de falhas da cache;
- ***Iface***: indica qual a placa de responsável pela *gateway* (*eth0*, *eth1*).

→ **Que mensagens *ARP* e endereços *MAC* associados são observados e porquê?**

Quando um *tux* envia um *ping* a outro *tux* cujo endereço *MAC* ainda não conhece, pacotes *ARP* de *request* e *reply* são trocados para que este endereço seja conhecido.

→ **Que pacotes *ICMP* são observados e porque?**

São observados pacotes *ICMP* de pedido e resposta (*ping request* e *ping reply*), visto que, após a configuração das rotas, todos os *tuxes* são visíveis entre si.

→ **Quais são os endereços *IP* e *MAC* associados a um pacote *ICMP* e porquê?**

Os endereços *IP* e *MAC* associados aos pacotes *ICMP* são os endereços dos *tuxes* de origem e destino.

Experiencia 4 – Configuração de um Router Comercial e Implementação do NAT

→ **Como se configura um router estático num router comercial?**

Com o comando *ip route*, cria-se uma VLAN que torna isto possível. Por exemplo:

ip route *172.16.10.0* *255.255.255.0* *172.16.11.253*

A **B** **C**

A corresponde ao endereço IP de destino, **B** corresponde à sua máscara e **C** ao endereço *IP* a usar como *gateway*.

→ **Quais são as rotas seguidas pelos pacotes durante a experiência e porquê?**

No caso da rota exigir, segue-se esta rota segundo a tabela de encaminhamento. Se não existir, o pacote é enviado para a rota *default*, e assim sucessivamente até chegar ao destino, ou não conseguir lá chegar.

→ **Como se configura o NAT num router comercial?**

O NAT foi configurado de acordo com a informação contida no guião. Os comandos executados estão contidos no anexo 1.

→ **O que faz o NAT?**

Como o *ipv4* tem endereços de 32 *bits*, o espaço de endereçamento esgotou-se e houve necessidade de arranjar técnicas para resolver o problema, como por exemplo o *NAT* (*Network Address Translation*).

O NAT consiste numa técnica que permite que a todos os pacotes enviados de uma rede privada com múltiplos computadores seja associado um único endereço público e que, dentro do router, seja gerida a multiplicidade de computadores com o mesmo endereço de rede, com o auxílio de uma tabela, para que o pacote recebido tenha como destino o computador pretendido.

Assim, é possível escalar de forma substancial o número de computadores com acesso à internet e reutilizar endereços em múltiplos locais para além de oferecer segurança.

Experiência 5 – DNS

→ **Como se configura o serviço *DNS* num *host*?**

Configura-se o serviço *DNS*, adicionando as linhas "*search netlab.fe.up.pt*" e "*nameserver 172.16.1.1*" ao ficheiro */etc/resolv.conf*: "*netlab.fe.up.pt"* corresponde ao nome do servidor DNS e "*172.16.1.1"* ao seu endereço de IP.

→ **Que pacotes são trocados pelo *DNS* e que informações são transportadas?**

São enviados pacotes para o servidor com o *hostname* desejado, que são respondidos com o respetivo endereço *IP* do *hostname*.

Experiência 6 – Conexões TCP

→ **Quantas conexões *TCP* foram abertas pela aplicação *FTP*?**

Na aplicação de *download*, são estabelecidas duas conexões *FTP*, sendo uma utilizada para a troca de mensagens entre servidor e cliente e outra para a transferência de dados enviados pelo servidor *FTP*.

→ **Em que conexão é transportado o controlo de informação?**

O controlo de informação é transportado pela conexão responsável pela troca de comandos entre servidor e cliente, isto é, a primeira referida na resposta anterior.

→ **Quais são as fases da conexão *TCP*?**

A conexão *TCP* divide-se em três fases: inicialmente estabelece-se a conexão, de seguida acontece a troca de dados e por fim, encerra-se a conexão.

→ **Como funciona o mecanismo ARQ TCP? Quais são os campos TCP relevantes? Que informação pode ser observada nos logs?**

O mecanismo *ARQ* utilizado pelo *TCP* é o da janela deslizante. Este usa *acknowlegment numbers*, *window size* e *sequence numbers* para determinar quais os pacotes que foram recebidos corretamente e quais podem ser enviados, bem como quais devem ser reenviados.

→ **Como funciona o mecanismo de controlo de congestionamento *TCP*? Quais são os campos relevantes? Como evoluiu o rendimento da conexão de dados ao longo do tempo? Está isto de acordo com o mecanismo de congestionamento *TCP*?**

O mecanismo de controlo de congestão *TCP* mantém uma estimativa dos *bytes* que a rede é capaz de encaminhar, e nunca são enviados mais *bytes* do que o mínimo entre a janela de congestão e a janela definida pelo recetor.

→ **O rendimento de uma conexão de dados é perturbado pela aparência de uma segunda conexão TCP e de que forma?**

Sim, é perturbado, como se pode ver pelo gráfico (anexo 6), o rendimento tem um decréscimo aquando do início da segunda conexão.

Conclusão

Com este projeto, foi possível compreender e consolidar todos os conceitos envolvidos na configuração de uma rede e no desenvolvimento de uma aplicação de *download*, com auxilio do protocolo *FTP*.

Neste novo método de ensino, foram muito mais complicadas a adaptação e a realização do projeto visto que o número de horas no laboratório não nos pareceu suficiente e o facto de só poder estar uma pessoa por semana presente na aula prática fez com que esta adaptação fosse muito mais lenta.

No entanto, pode-se concluir que foram cumpridos todos os objetivos com sucesso.

Anexos

Anexo 1: Configuração do *Router*

```
> configure terminal

> interface gigabitethernet 0/0
> ip address 172.16.11.254 255.255.255.0
> no shutdown
> ip nat inside
> exit

> interface gigabitethernet 0/1
> ip address 172.16.1.19 255.255.255.0
> no shutdown
> ip nat outside
> exit

> ip nat pool ovrld 172.16.1.19 172.16.1.19 prefix 24
> ip nat inside source list 1 pool ovrld overload

> access-list 1 permit 172.16.10.0 0.0.0.7
> access-list 1 permit 172.16.11.0 0.0.0.7

> ip route 0.0.0.0 0.0.0.0 172.16.1.254
> ip route 172.16.10.0 255.255.255.0 172.16.11.253

> end
```

Anexo 2: Configuração do *switch*

```
> vlan 10
> vlan 11
> interface fastethernet 0/2    //adicionar o tux2 à vlan 11
> switchport mode access
> switchport access vlan 11
> exit
> interface fastethernet 0/3    //adicionar o tux3 à vlan 10
> switchport mode access
> switchport access vlan 10
> exit
> interface fastethernet 0/4    //adicionar o tux4 à vlan 10
> switchport mode access
> switchport access vlan 10
> exit
> interface fastethernet 0/5    //adicionar o tux4 à vlan 10
> switchport mode access
> switchport access vlan 11
> exit
> interface fastethernet 0/6    //adicionar o router à vlan 11
> switchport mode access
> switchport access vlan 11
> exit
> end
```

Anexo 3: Configuração do *tux2*

```
> ifconfig eth0 up
> ifconfig eth0 172.16.11.1/24
> route add -net 172.16.10.0/24 gw 172.16.11.253
> route add default gw 172.16.11.254
> echo $'search netlab.fe.up.pt\nnameserver 172.16.1.1' >
/etc/resolv.conf
```

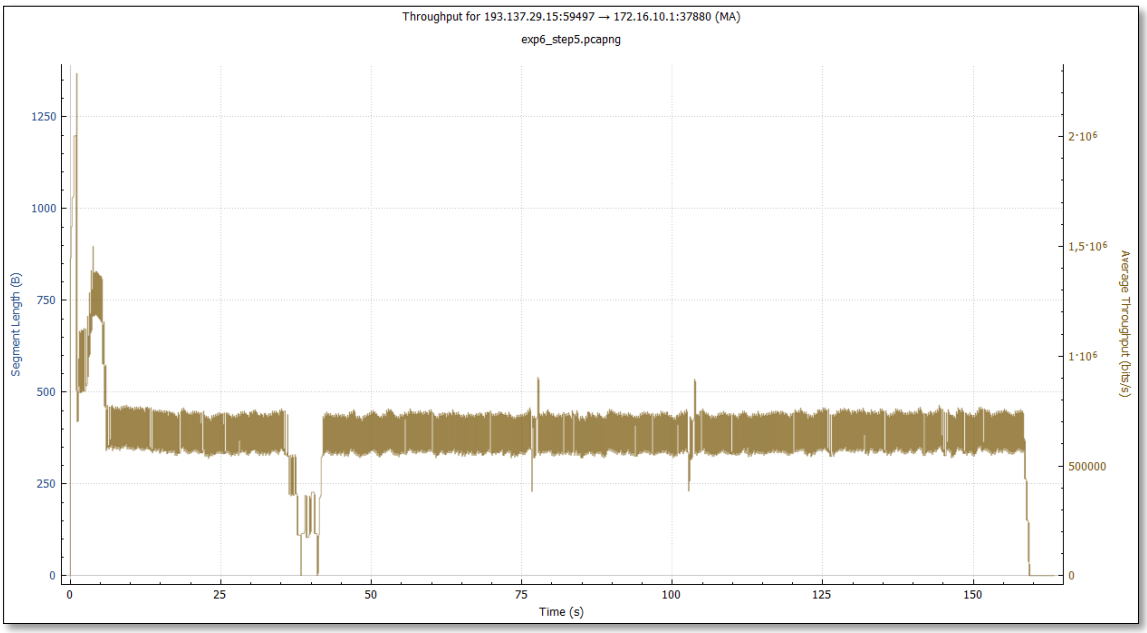
Anexo 4: Configuração do *tux3*

```
> ifconfig eth0 up
> ifconfig eth0 172.16.10.1/24
> route add -net 172.16.11.1/24 gw 172.16.10.254
> route default gw 172.16.10.254
> echo $'search netlab.fe.up.pt\nnameserver 172.16.1.1' >
/etc/resolv.conf
```

Anexo 5: Configuração do *tux4*

```
> ifconfig eth0 up
> ifconfig eth0 172.16.10.254/24
> ifconfig eth1 up
> ifconfig eth1 172.16.11.253/24
> route add default gw 172.16.11.254
> echo 1 > /proc/sys/net/ipv4/ip_forward
> echo 0 > /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts
```

Anexo 6: Throughput



Anexo 7: Aplicação de Download

main.c

```
#include <stdio.h>
#include "url.h"
#include "ftp_connection.h"
void printURL(url_struct *url ){
    printf("\nPRINTING URL INFO: \n");
    printf("user = %s\n", url->user);
    printf("password = %s\n", url->password);
    printf("host = %s\n", url->host);
    printf("url_path = %s\n", url->url_path);
    printf("filename = %s\n", url->filename);
    printf("ip address = %s\n\n", url->ip_address);
}
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: ./download
ftp://[<user>:<password>@]<host>/<url-path>\n");
        exit(1);
    }
    printf("Parsing URL...\n");
    url_struct *url = createUrlStruct();
    if(parse_file_url(argv[1], url) < 0){
        fprintf(stderr, "Url parser failed!\n");
        exit(1);
    }
    printf("URL is parsed!\n\n");
    printf("Getting IP address... \n");
    if(getIpAddress(url) < 0) exit(1)
    printf("Ip address is ok!\n\n");
    printURL(url);
    printf("Establishing a TCP connection... \n");
    int socket_fd = ftp_connect(url->port, url->ip_address);
    if(socket_fd < 0) exit(1);
    printf("Established TCP connection!\n\n");
    printf("Logging in...\n");
    if(ftp_login(socket_fd, url->user, url->password) < 0)
```

```
exit(1);
    printf("Logged in!\n\n");
    printf("Entering Passive Mode... \n");
    int data_fd = ftp_passive_mode(socket_fd);
    if(data_fd < 0) exit(1);
    printf("Entered Passive Mode\n\n"
    printf("Downloading file...\n");
    if(ftp_request_file(socket_fd, url->url_path) < 0) exit(1);
    if(ftp_download_file(data_fd, url->url_path, url->filename)
< 0) exit(1);
    printf("Downloaded file\n\n");
    printf("Disconecting...\n");
    if(close(socket_fd) < 0) {
        perror("Error closing socket file descriptor!\n");
        exit(1);
    }
    if(close(data_fd) < 0) {
        perror("Error closing data socket file descriptor!\n");
        exit(1);
    }
    printf("Disconected with success!\n");
    free(url);
    return 0;
}
```

ftp_connection.c

```
#include "ftp_connection.h"

int ftp_connect(int port, char *ip_address) { //after successful
connection, server sends a line of welcome text with 220 as code
to indicate the ready state

    int socket_fd = socket_establish_connection(port,
ip_address);
    if(socket_fd < 0) return -1;

    /* Receives the presentation message from the server */
```

```
    char response[MAX_LINE_SIZE];

    int status_code = read_response(socket_fd, response);

    if(status_code < 0) return -1;

    if(status_code != CMD_READY_STATE) {
        fprintf(stderr, "Server is not ready!\n");
        return -1;
    }

    memset(response, 0, MAX_LINE_SIZE);
    return socket_fd;
}

int ftp_login(int socket_fd, char *username, char *password) {

    /*
    > user anonymous
    < 331 Password required for euproprio.
    > pass qualquer-password
    < 230 User anonymouslogged in.
    */
    char user_command[MAX_LINE_SIZE];
    sprintf(user_command, "USER %s%s", username,
CMD_TERMINATOR);

    char user_response[MAX_LINE_SIZE];
    if (send_command_receive_response(socket_fd, user_command,
CMD_USERNAME_CORRECT, user_response) < 0) return -1;

    char pass_command[MAX_LINE_SIZE];
    sprintf(pass_command, "PASS %s%s", password,
CMD_TERMINATOR);

    char pass_response[MAX_LINE_SIZE];
    if (send_command_receive_response(socket_fd, pass_command,
```

```
CMD_LOGIN_CORRECT, pass_response) < 0) return -1;


    memset(user_command, 0, MAX_LINE_SIZE);
    memset(user_response, 0, MAX_LINE_SIZE);
    memset(pass_command, 0, MAX_LINE_SIZE);
    memset(pass_response, 0, MAX_LINE_SIZE);

    return 0;
}

int ftp_passive_mode(int socket_fd) {

    /*> pasv
    < 227 Entering Passive Mode (193,136,28,12,19,91)
    */

    char pasv_command[MAX_LINE_SIZE];
    sprintf(pasv_command, "PASV%s", CMD_TERMINATOR);

    char response[MAX_LINE_SIZE];
    if (send_command_receive_response(socket_fd, pasv_command,
CMD_PASSIVE_MODE, response) < 0) return -1;

    memset(pasv_command, 0, MAX_LINE_SIZE);

    char * response_values = strrchr(response, '(');

    int ip1, ip2, ip3, ip4, port1, port2;
    if (sscanf(response_values, "(%d,%d,%d,%d,%d,%d)", &ip1,
&ip2, &ip3, &ip4, &port1, &port2) != 6){
        fprintf(stderr, "Error parsing ip address and port
number\n");
        return -1;
    }

    //gets the port number
```

```
    int data_port = port1 * 256 + port2;

    //gets ip address
    char ip_address[MAX_STRING_SIZE];
    sprintf(ip_address, "%d.%d.%d.%d", ip1, ip2, ip3, ip4);

    //connects to the data port
    int data_fd = socket_establish_connection(data_port,
ip_address);

    if(data_fd < 0) return -1;

    memset(ip_address, 0, MAX_STRING_SIZE);
    memset(response, 0, MAX_LINE_SIZE);

    return data_fd;
}

int ftp_request_file(int socket_fd, const char*path) {

    /*
        > retr path
        < ... sending file
    */

    char retr_command[MAX_LINE_SIZE];
    sprintf(retr_command, "RETR %s%s", path, CMD_TERMINATOR);
    if(send_command(socket_fd, retr_command) < 0) return -1;

    memset(retr_command, 0, MAX_LINE_SIZE);
    return 0;
}

int ftp_download_file(int data_fd, const char *path, const char
*filename) {

    int fd = open(filename, O_WRONLY | O_CREAT, 0666);
```

```
    if(fd < 0) {
        fprintf(stderr, "Error opening file!\n");
        return -1;
    }

    char buf[MAX_LINE_SIZE];
    int bytes;

    while((bytes = read(data_fd, buf, MAX_LINE_SIZE)) > 0){
        if(write(fd, buf, bytes) < bytes){
//write the information that server is sending to the created
file
            fprintf(stderr, "Error writing to file!\n");
            close(fd);
            return -1;
        }
    }


    memset(buf, 0, MAX_LINE_SIZE);

    if(close(fd) < 0){
        fprintf(stderr, "Error closing file!\n");
        return -1;
    }

    return 0;
}
```

ftp_connection.h

```
#ifndef FTP_CONNECTION_H
#define FTP_CONNECTION_H

#pragma once

#include <stdio.h>
#include <stdlib.h>
```

```c
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "socket.h"
#include "macros.h"

/**
 * @brief Establishs a TCP connection and checks server welcome
message
 *
 * @param port server port
 * @param ip_address ip address of server
 *
 * @return Socket file descriptor on success; -1 on error
*/
int ftp_connect(int port, char *ip_address);

/**
 * @brief Log in user, sending user name and password
 *
 * @param socket_fd socket file descriptor
 * @param username  user name
 * @param password  user password
 *
 * @return 0 on success; -1 on error
*/
int ftp_login(int socket_fd, char *username, char *password);

/**
 * @brief Enters passive mode and establishs a tcp connection
with the new data port and ip address
 *
 * @param socket_fd socket file descriptor
 *
```

```
 * @return data socket descriptor on success; -1 on error
*/
int ftp_passive_mode(int socket_fd);

/**
 * @brief Resquests the download of the file
 *
 * @param socket_fd socket file descriptor
 * @param path      file path
 *
 * @return 0 on success; -1 on error
*/
int ftp_request_file(int socket_fd, const char* path);

/**
 * @brief Downloads file
 *
 * @param socket_fd socket file descriptor
 * @param path      file path
 * @param filename  file name
 *
 * @return 0 on success; -1 on error
*/
int ftp_download_file(int data_fd, const char *path, const char
*filename);

#endif /*FTP_CONNECTION_H*/
```

Macros.h

```
#define MAX_STRING_SIZE          256
#define MAX_LINE_SIZE            1024
#define CMD_TERMINATOR           "\r\n"
#define CMD_READY_STATE          220
#define CMD_USERNAME_CORRECT     331
#define CMD_PASSWORD_INCORRECT   530
#define CMD_LOGIN_CORRECT        230
#define CMD_PASSIVE_MODE         227
```

socket.c

```
#include "socket.h"

int socket_establish_connection(int port, char *ip_address) {
//clientTCP.c moodle

    int socketfd;
    struct sockaddr_in server_addr;

    /*server address handling*/
    bzero((char *)&server_addr, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr(ip_address);
    server_addr.sin_port = htons(port);

    /*open an TCP socket*/
    if ((socketfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Error on socket creation:");
        return -1;
    }

    /*connect to the server*/
    if (connect(socketfd, (struct sockaddr *)&server_addr,
sizeof(server_addr)) < 0) {
        perror("Can't connect to server");
        return -1;
    }

    return socketfd;
}

int isDigit(char c){
    return (c >= '0' && c <= '9');
}

int read_response(int socket_fd, char* response) {
    FILE *socket_file = fdopen(socket_fd, "r"); //Open socket
```

```
for reading

    if(socket_file == NULL) {
        perror("fdopen() failed:");
        return -1;
    }

    size_t bytes;
    char *buf = malloc(MAX_LINE_SIZE);

    while(getline(&buf, &bytes, socket_file) > 0){
        printf("%s", buf);
        if((buf[3] == ' ') && isDigit(buf[0]) && isDigit(buf[1])
&& isDigit(buf[2])) break;
    }

    strcpy(response, buf);

    int response_code = atoi(buf);
    printf("code = -%d-\n", response_code);

    free(buf);
    return response_code;
}

int send_command(int socket_fd, char * command){

    if(write(socket_fd, command, strlen(command)) <= 0){
        fprintf(stderr, "Error writitng to socket!\n");
        return -1;
    }

    return 0;
}

int send_command_receive_response(int socket_fd, char* command,
int response_code, char *response) {
```

```
    if(send_command(socket_fd, command) < 0) return -1;

    int code = read_response(socket_fd, response);

    if(code < 0) return -1;

    if(code != response_code) {
        if (response_code == CMD_USERNAME_CORRECT && code ==
CMD_LOGIN_CORRECT) return 0;
        fprintf(stderr, "Failed: Server sent a code that
indicates error!\n");
        return -1;
    }

    return 0;
}
```

socket.h

```
#ifndef SOCKET_H
#define SOCKET_H

#pragma once

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <netdb.h>
#include <string.h>

#include "macros.h"
```

```
/**
 * @brief Establish a TCP connection
 *
 * @param port server port
 * @param ip_address ip address of server
 *
 * @return Socket file descriptor on success; -1 on error
*/
int socket_establish_connection(int port, char* ip_address);

/**
 * @brief Reads server response
 *
 * @param socket_fd socket file descriptor
 * @param response array where it will be stored the response
message
 *
 * @return response code on succes; -1 on error
*/
int read_response(int socket_fd, char* response);

/**
 * @brief Sends a message to server
 *
 * @param socket_fd socket file descriptor
 * @param command message to send
 *
 * @return 0 on succes; -1 on error
*/
int send_command(int socket_fd, char * command);

/**
 * @brief Sends a message to server and reads its response,
checking if it is valid
 *
 * @param socket_fd socket file descriptor
 * @param command message to send
```

```
 * @param response_code code that server will send on success
 * @param response array where it will be stored the response
message
 *
 * @return 0 on succes; -1 on error
*/
int send_command_receive_response(int socket_fd, char* command,
int response_code, char *response);

#endif /*SOCKET_H*/
```

url.c

```
#include "url.h"

url_struct * createUrlStruct() {

    url_struct *url = malloc (sizeof (url_struct));
    url->user = malloc(MAX_STRING_SIZE);
    url->password = malloc(MAX_STRING_SIZE);
    url->host = malloc(MAX_STRING_SIZE);
    url->url_path = malloc(MAX_STRING_SIZE);
    url->filename = malloc(MAX_STRING_SIZE);
    url->ip_address = malloc(MAX_STRING_SIZE);
    url->port = 21; //this protocol uses port 21 by default

    return url;
}

int parse_file_url(char * url, url_struct *urlInfo) { //format
ftp://[<user>:<password>@]<host>/<url-path>

    char *ftp = malloc(MAX_STRING_SIZE);
    memcpy(ftp, url, 6);
    ftp[6] = 0;

    if(strcmp(ftp, "ftp://") != 0) return -1;
```

```
    strtok(url, "/");
    char* rest_args = strtok(NULL, "") + 1;
//[<user>:<password>@]<host>/<url-path>

    char *user = malloc(MAX_STRING_SIZE);
    char *password = malloc(MAX_STRING_SIZE);
    char *host = malloc(MAX_STRING_SIZE);
    char *url_path = malloc(MAX_STRING_SIZE);

    if(strchr(rest_args, '@') == NULL){ //anonymous
        if(sscanf(rest_args, "%[^/]/%s", host, url_path) == 2){
            user = "anonymous";
            password = "password";
        }
        else return -1;
    }
    else if (sscanf(rest_args, "%[^:]:%[^@]@%[^/]/%s", user,
password, host, url_path) != 4) return -1;

    strcpy(urlInfo->user, user);
    strcpy(urlInfo->password, password);
    strcpy(urlInfo->host, host);
    strcpy(urlInfo->url_path, url_path);

    char *filename = strrchr(urlInfo->url_path, '/');
    if(filename == NULL) urlInfo->filename = urlInfo->url_path;
    else urlInfo->filename = ++filename;

    return 0;
}

int getIpAddress(url_struct *url){ //getip.c moodle

    struct hostent *h;

    if ((h = gethostbyname(url->host)) == NULL) {
```

```
        herror("gethostbyname");
        return -1;
    }

    url->ip_address = inet_ntoa(*((struct in_addr *)h->h_addr));

    return 0;
}
```

url.h

```
#ifndef URL_H
#define URL_H

#pragma once

#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "macros.h"


/**
 * @brief Struct where it is stored the url information
 */
typedef struct url_struct {
    char *user;
    char *password;
    char *host;
    char *url_path;
    char *filename;
    char *ip_address;
```

```
    int port;

} url_struct;

/**
 * @brief Allocates memory for the struct url_struct and its
members
 *
 * @return struct that was created
*/
url_struct * createUrlStruct();

/**
 * @brief Parses url and saves its data
 *
 * @param url url to be parsed
 * @param urlInfo struct where it will be saved the url's data
 *
 * @return 0 on success; -1 on error
*/
int parse_file_url(char * url, url_struct *urlInfo);

/**
 * @brief Get IP Address given a host name
 *
 * @param url struct to get host and to save the ip address
 *
 * @return 0 on success; -1 on error
*/
int getIpAddress(url_struct *url);

#endif /*URL_H*/
```