Faculdade de Engenharia da Universidade do Porto

2º Trabalho Laboratorial – Desenvolvimento de uma aplicação de Download e Configuração e Estudo de uma Rede de Computadores

Relatório

Redes de computadores (CD)

Margarida Mesquita Leal – up201906884
Miguel Augusto Marombal Araújo – up201905077

Índice

1. Introdução 5

2. Parte I – Download Application 5

2.1. Arquitetura da Aplicação de Download 5

2.1.1. Processamento da string de input 5

2.1.2. Estabelecimento da ligação com o servidor FTP e transferência de dados 6

2.1.3. Visão geral (main.c) 7

2.2. Resultados 7

3. Parte II – Network configuration and analysis 8

3.1. Experiência 1 8

3.1.1. What are the ARP packet and what are they used for? 8

3.1.2. What are the MAC and IP addresses of ARP packets and why? 9

3.1.3. What packets does the ping command generate? 9

3.1.4. What are the MAC and IP addresses of the ping packets? 10

3.1.5. How to determine if a receiving Ethernet frame is ARP, IP, ICMP? 10

3.1.6. How to determine the length of a receiving frame? 10

3.1.7. What is the loopback and why is it important? 11

3.2. Experiência 2 12

3.2.1. How to configure vlan60? 12

3.2.2. How many broadcast domains are there? How can you conclude it from the logs? 12

3.3. Experiência 3 12

3.3.1. What routes are there in the tuxes? What are their meaning? 13

3.3.2. What information does an entry of the forwarding table contain? 13

3.3.3. What ARP messages, and associated MAC addresses, are observed and why? 13

3.3.4. What ICMP packets are observed and why? 14

3.3.5. What are the IP and MAC addresses associated to ICMP packets and why? 14

3.4. Experiência 4 14

3.4.1. How to configure a static route in a commercial router? 14

3.4.2. What are the paths followed by the packets in the experiments carried out and why? 15

3.4.3. How to configure NAT in a commercial router? 15

3.4.4. What does NAT do? 15

3.5. Experiência 5 15

3.5.1. How to configure the DNS service at an host? 15

3.5.2. What packets are exchanged by DNS and what information is transported? 16

3.6. Experiência 6 16

3.6.1. How many TCP connections are opened by your ftp application? 16

3.6.2. In what connection is transported the FTP control information? 16

3.6.3. What are the phases of a TCP connection? 16

3.6.4. How does the ARQ TCP mechanism work? What are the relevant TCP fields? What relevant information can be observed in the logs? 17

3.6.5. How does the TCP congestion control mechanism work? What are the relevant fields. How did the throughput of the data connection evolve along the time? Is it according the TCP congestion control mechanism? 17

3.6.6. Is the throughput of a TCP data connections disturbed by the appearance of a second TCP connection? How? 17

4. Conclusões 17

5. Anexos 18

5.1. Anexo 1 – Código Fonte 18
5.1.1. Main.c 18
5.1.2. Args.h 21
5.1.3. Args.c 23
5.1.4. Connection.c 27
5.1.5. Connection.h 30

Sumário

O trabalho sobre o qual incide este relatório foi realizado no âmbito da unidade curricular Redes de Computadores. O mesmo previa dois objetivos: o desenvolvimento de uma aplicação de download de um cliente FTP e a configuração de uma rede de computadores no laboratório.

No final das aulas reservadas para o trabalho os requisitos foram cumpridos pelo que o mesmo foi concluído com sucesso sendo capaz de transferir um ficheiro sem erros numa rede de computadores configurada pelo grupo como pretendido, no processo foram desenvolvidas capacidades quer de interpretação quer de implementação no domínio de Redes de Computadores.

1. Introdução

O trabalho desenvolvido foi dividido em duas partes. Na primeira, foi desenvolvida uma **aplicação de *download*** capaz de transferir um ficheiro alojada num servidor, garantindo todas as comunicações necessárias com este. Já na segunda parte do trabalho, foi **configurada progressivamente uma rede de computadores**, analisando passo-a-passo o comportamento da rede em cada uma das experiências.

Com este relatório pretende-se analisar e interpretar o desenvolvimento do trabalho em cada uma das partes (implementação da aplicação de download e configuração da rede de computadores). Para além disso, o presente relatório visa clarificar as estratégias de implementação e desenvolvimento do trabalho fornecendo uma justificação clara das decisões tomada para a realização do mesmo.

Para garantir uma estrutura organizada o relatório segue divido de forma coerente com as duas partes do trabalho:

- **Parte 1** – Download Application:

  - Explicação da arquitetura utilizada

  - Exposição dos resultados obtidos com a aplicação

- **Parte 2** – Configuração e Analise de uma rede de computadores, seguindo a ordem cronológica das experiências realizadas no laboratório (experiência 1 a 4), dando resposta às questões fornecidas pelo guião em cada uma das experiências.

2. Parte I – Download Application

2.1. Arquitetura da Aplicação de Download

A aplicação desenvolvida foi arquitetada em duas partes distintas. Na primeira é feito o processamento dos argumentos extraindo dos mesmos toda a informação necessária para o estabelecimento da comunicação com o servidor. Na segunda parte, é feito o estabelecimento da comunicação com o servidor *FTP* e transferência do ficheiro pretendido.

A aplicação permite a transferência de dados em 2 modos, o modo anonimo e o modo utilizador. Seguindo para cada um o seguinte formato no *input*:

Modo anonimo: ftp://<hostname>/<pathname>
Modo utilizador: ftp://<user>:<password>@<hostname>/<pathname>

2.1.1. Processamento da string de input

O processamento da *string* de input foi feito com recurso à função ***parseArgs,*** cuja responsabilidade é preencher a estrutura de dados ***connectionArgs*** de modo a organizar toda a informação necessária para o posterior estabelecimento da comunicação. No caso do *input* não respeitar as regras estabelecidas ou caso seja impossível de obter o IP necessário, a função retorna ERRO (-1), de modo a assinalar a falha no processamento do *input* e, além disso, imprime no terminal uma mensagem de erro que identifica em que ponto da função o mesmo

ocorreu permitindo distinguir, por exemplo, se a falha da função ocorreu na formatação do input ou na obtenção do IP. Para obter o endereço IP necessário e o nome do ficheiro a transferir são utilizadas duas funções auxiliares, ***getIp*** e ***getFileName***, respetivamente. Estas 3 funções estão programadas no ficheiro **args.c**, enunciadas juntamente com a estrutura mencionada no ficheiro **args.h**. O ficheiro **args.c** conta também com a função ***printArgs*** que permite imprimir os parâmetros da estrutura.

Nas seguintes figuras podemos ver a enunciação da função ***parseArgs*** e da estrutura de

Figura 2 Estrutura connectionArgs

```
typedef struct connectionArgs{
    char user[SIZE];
    char password[SIZE];
    char host[SIZE];
    char url_path[SIZE];
    char file_name[SIZE];
    char host_name[SIZE];
    char ip[SIZE];
} connectionArgs;
```

```
/**
 * @brief This function processes the URL, separating it into different elements
 *        and storing them in the respective places in the structure (connectionArgs *args)
 *
 * @param url pointer to url string
 * @param args pointer to the struct where the information will be stored
 * @return ERROR (-1) on error; OK (1) on sucess
 */
int parseArgs(char *url, connectionArgs *args);
```

Figura 1 Função parseArgs

2.1.2. Estabelecimento da ligação com o servidor FTP e transferência de dados

Para o estabelecimento da ligação FTP e respetiva transferência do ficheiro pretendido foi usado o ficheiro **connection.c**, com a enunciação das funções no ficheiro **connection.h.** Sendo aqui que foram programadas as seguintes funções:

- **client_init()** – Responsável pela abertura de um *socket* para a comunicação entre o cliente e o servidor.
- **clientCommand()** – Responsável por enviar comandos para o servidor
- **pasvMode()** – Responsável pela leitura da resposta ao comando "*pasv*" que indica que iremos iniciar uma comunicação no modo passivo. Esta função calcula, portanto, a porta que será utilizada para tal (baseada na resposta do servidor, segundo o protocolo)
- **readResponse()** – Responsável por ler as respostas do Servidor
- **writeFile()** – Responsável por criar e escrever no ficheiro o que está a ser lido pelo servidor

2.1.3. Visão geral (main.c)

A função ***int main()*** (presente no ficheiro **main.c**) é a responsável por chamar todas as restantes funções de forma sequencial e organizada de modo a executar o programa como pretendido. Para tal segue a seguinte ordem:

Inicialmente, são declaradas todas as variáveis necessárias e é chamada a função ***parseArgs*** de modo a preencher a estrutura ***connectionArgs*** com todas as informações necessárias para a comunicação (***user, password, host, url_path, fila_name, host_name*** e ***IP***). É no final deste processo imprimido no terminal todas as informações guardadas na estrutura com recurso à função ***printArgs***.

Após o processamento dos argumentos é inicializado o primeiro ***Socket*** de comunicação com o servidor com recurso à função ***client_init()*** na porta **21** (número que identifica a porta TCP), este *sockat* será responsável pela troca de comandos entre cliente e servidor (veremos mais à frente a abertura de um segundo *sockat* responsável pela troca de dados), é lida a resposta com a função ***readResponse()***, sendo de seguida enviados o *user* e *password* (com respetivas leituras das respostas por parte do servidor). De seguida é enviado o comando '*pasv'* para o servidor indicado o modo passivo que será usado na comunicação. E através da função ***pasvMode()*** é lida a resposta do servidor pela qual se calcula a porta que será utilizada para a transferência do ficheiro. É então aberta, mais uma vez com recurso à função ***client_init()*** um novo *sockat* desta vez na porta calculada pela função ***pasvMode()***, este será o *sockat* utilizado para a transferência do ficheiro. É, neste momento e através do primeiro *sockat* mencionado e reservado para a troca de comandos enviado o comando '***retr'*** seguido do *url_path* para o ficheiro pretendido, e neste momento é lido no primeiro *sockat* a resposta do servidor ao pedido e no segundo a transferência do ficheiro (caso haja sucesso no pedido). Através da função ***writeFile()*** é aberto um ficheiro no PC para guardar a informação transferida do servidor.

Por fim, é enviado o comando '***quit'*** para indiciar o fim da ligação e ambos os *Sockats* são fechados.

2.2. Resultados

A aplicação desenvolvida foi testada quer em modo anonimo quer em modo de utilizador e cumpriu o proposto quando compilada no nosso computador pessoal. Contudo, quando compilada nos tuxs do laboratório a aplicação não corria devidamente, pelo que a sua demonstração na aula de avaliação foi feita através do computador pessoal.

A seguinte figura mostra um exemplo de uma transferência bem-sucedida de uma imagem com a nossa aplicação.

```
(base) MBP-de-Miguel:new2 miguelmarombal$ ./a.out ftp://anonymous:rcom@ftp.up.pt/pub/kodi/screenshots/kodi-addons.jpg
Information

[Username: anonymous]
[Password: rcom]
[Host: ftp.up.pt]
[HostName: mirrors.up.pt]
[FileName: kodi-addons.jpg]
[UrlPath: pub/kodi/screenshots/kodi-addons.jpg]
[IpAdress: 193.137.29.15]


220-Welcome to the University of Porto's mirror archive (mirrors.up.pt)
220----------------------------------------------------------------------
220-
220-All connections and transfers are logged. The max number of connections is 200.
220-
220-For more information please visit our website: http://mirrors.up.pt/
220-Questions and comments can be sent to mirrors@uporto.pt
220-
220-
220
< user anonymous

331 Please specify the password.
< pass rcom

230 Login successful.
< pasv

227 Entering Passive Mode (193,137,29,15,209,156).
IP: 193.137.29.15
Port Number: 53660
< retr pub/kodi/screenshots/kodi-addons.jpg

150 Opening BINARY mode data connection for pub/kodi/screenshots/kodi-addons.jpg (116223 bytes).
< quit

226 Transfer complete.


############################################################
######################---> Done <---######################
############################################################


(base) MBP-de-Miguel:new2 miguelmarombal$
```

Figura 3 Exemplo de uma transferência executada com a aplicação

3. Parte II – Network configuration and analysis

Na segunda parte do trabalho propunha-se a configuração e posterior análise de uma rede de computador num processo progressivo e dividido em quatro experiências. Neste capítulo cada uma das quatro experiências será analisada individualmente. Respondendo, em cada caso, ao leque de questões disponibilizadas no guião cuja resposta serve de base para a compreensão dos tópicos mais pertinentes e fundamentais para perceber a experiência em questão.

NOTA: Apesar de todo o trabalho ter sido desenvolvido na bancada 6, as capturas tiveram de ser repetidas fora das aulas pelo que só foi possível fazê-las na bancada 5. Deste modo, decidimos que para o relatório manter as respostas às perguntas tendo em conta a bancada em que as aulas decorreram, isto é bancada 6, contudo as capturas terão os endereços da bancada 5.

Devido à inexistência do tux1 nos laboratórios, o tux1 presente no guião corresponderá ao tux3 do laboratório. Os restantes mantém a coerência, isto é, tanto o tux4 como o tux2 são os mesmos quer no laboratório quer nos esquemas do guião.

3.1. Experiência 1

Na experiência 1 propunha-se a criação de uma rede onde os tux3 e tux4 se ligavam. Para tal, ligou-se o tux3 (eth0) e o tux4 (eth0) ao *switch* de modo que ambos pudessem comunicar entre si.

3.1.1. What are the ARP packet and what are they used for?

Os pacotes ARP (*Address Resolution Protocol*) são pacotes utilizados para fazer mapeamento de um endereço de rede a um endereço físico. Este protocolo surge da

necessidade de identificar o endereço físico de um dispositivo cuja única informação que temos de momento é o seu endereço IP.

Para descobrir o endereço MAC correspondente ao endereço IP do dispositivo com que se pretende comunicar é enviado em Broadcast um pacote ARP que contém o endereço IP do dispositivo para o qual se pretende comunicar e esperasse uma resposta com o endereço MAC correspondente.

3.1.2. What are the MAC and IP addresses of ARP packets and why?

Os pacotes ARP enviados, como o da figura seguinte ilustra, contêm o endereço MAC e IP de quem os enviou bem como o endereço IP do destinatário. Visto que o propósito dos Pacotes ARP (enviados) é descobrir o endereço MAC de um dispositivo do qual apenas conhecemos o endereço IP (como já explicado na questão 3.1.1.) este ainda não é conhecido pelo que se encontra a 00:00:00_00:00:00

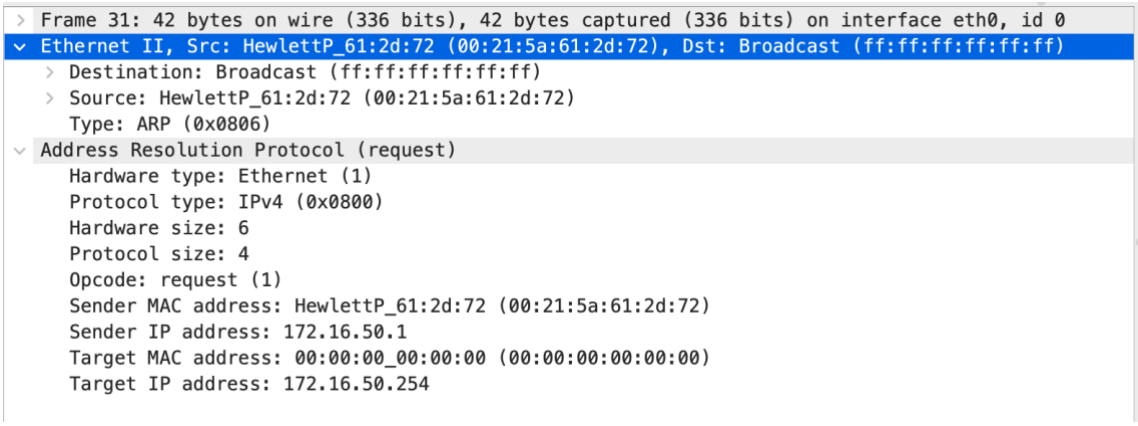

Figura 4 Pacote ARP enviado (request)

No pacote ARP de resposta já todos os endereços são conhecidos pelo que é possível ver quer o endereço IP e MAC de quem envia como o endereço IP e MAC de destino. Como podemos verificar na seguinte figura:

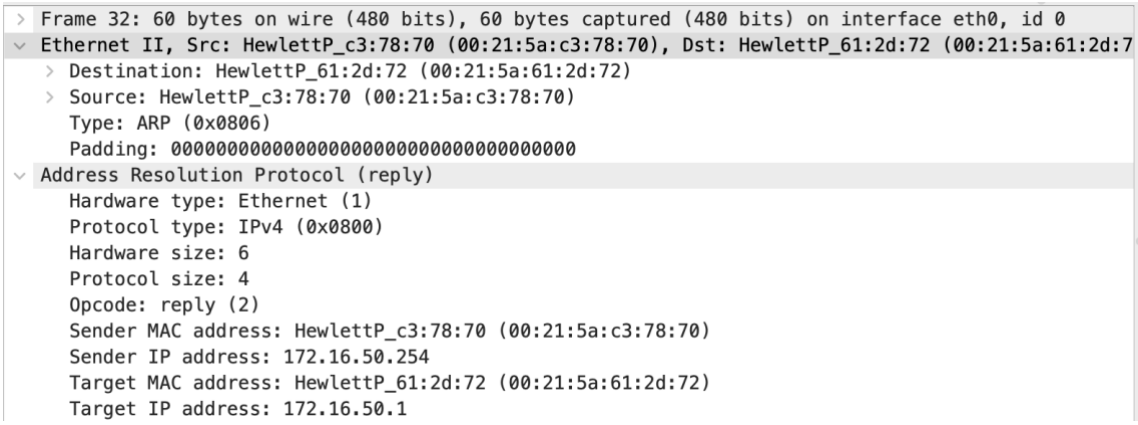

Figura 5 Pacote ARP recebido (reply)

3.1.3. What packets does the ping command generate?

O comando ping gera pacotes ICMP (*Internet Control Message Protocol*), sendo os mesmos usados para transferir mensagens de controlo entre endereços IP.

3.1.4. What are the MAC and IP addresses of the ping packets?

Os endereços MAC e IP presentes nos pacotes ICMP são relativos quer ao dispositivo que está a realizar o ping quer ao destinatário do mesmo.
Nesta experiência ao realizar um ping do tux4 (com o endereço IP 172.16.60.254) a partir do tux3 (identificado pelo endereço IP 172.16.60.1) os endereços de origem e destino IP e MAC dos pacotes ICMP gerados vão ser os dos tuxs em questão, tux3 e tux4 respetivamente.

3.1.5. How to determine if a receiving Ethernet frame is ARP, IP, ICMP?

O tipo da trama Ethernet recebida é determinado analisando o cabeçalho presente na mesma, campo "EtherType". Um valor de 0x0806 neste campo representa uma trama do tipo ARP e um valor de 0x0800 representa uma trama do tipo IP. Pode-se ainda verificar se estamos perante uma trama do tipo ICMP, caso se trate de uma trama IP e o seu cabeçalho contenha o valor 0x01 no campo respetivo ao protocolo.

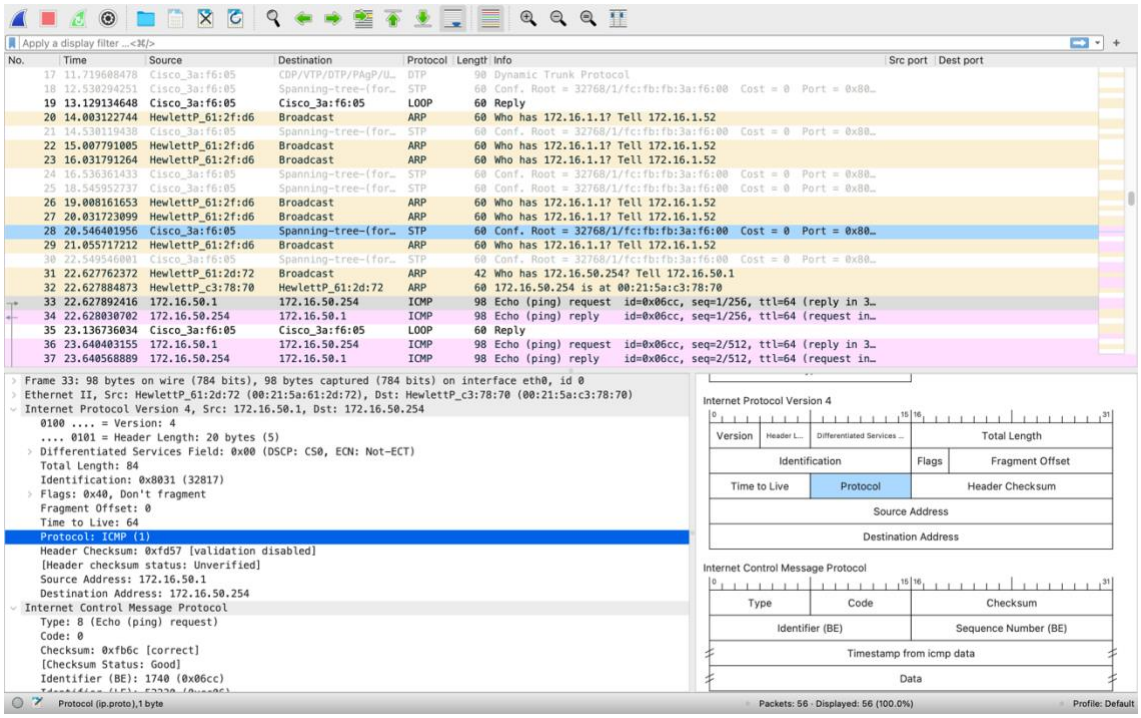

Figura 6 Exemplo de um pacote ICMP capturado

3.1.6. How to determine the length of a receiving frame?

O comprimento de uma trama está presente na mesma e, nesta experiência, foi obtido usado o ***wireshark***. Na figura podemos ver um exemplo retirado desta experiência de uma frame com 98 *bytes*.

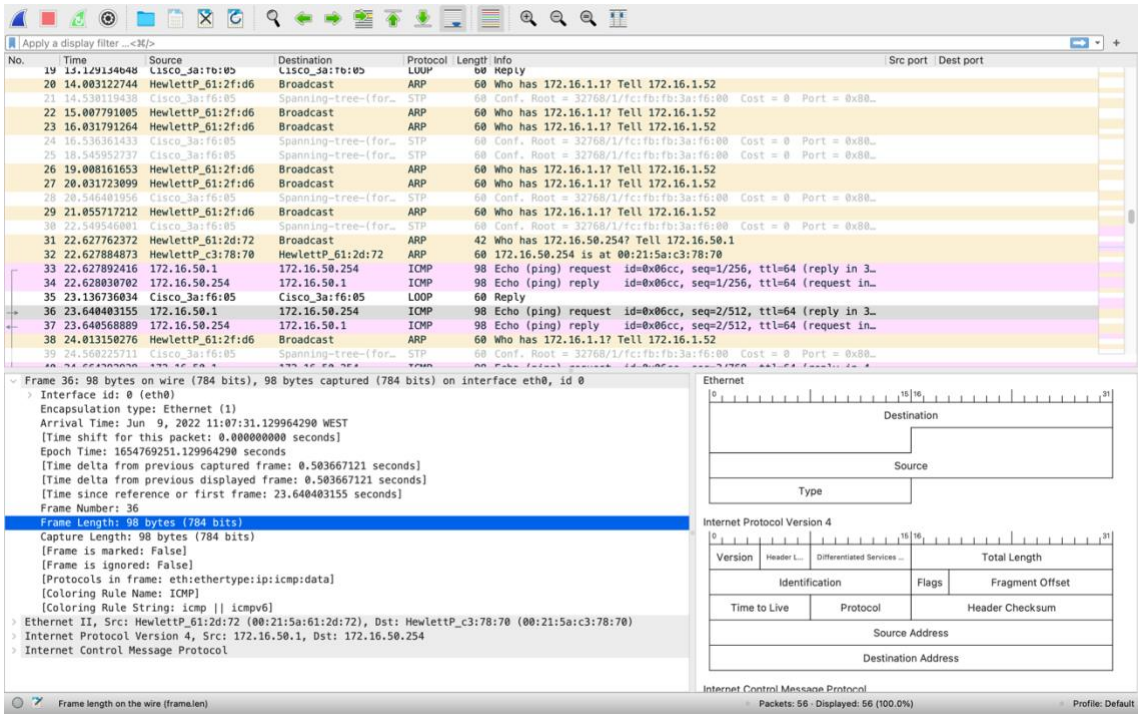

Figura 7 Tamanho de um pacote visto no wireshark

3.1.7. What is the loopback and why is it important?

Loopback é uma interface de rede virtual utilizada para realizar testes de diagnóstico (espaçados de 10 segundos). Esta interface permite ter um endereço de IP no router que está sempre ativo, não se sujeitando a uma interface física. Na seguinte figura pudemos ver nas linhas 5 e 19 dois pacotes deste tipo e confirmar, através da coluna *TIME*, a separação de 10 segundos entre os mesmos. Na seguinte figura podemos analisar um pacote deste tipo com recurso ao ***wireshark***.

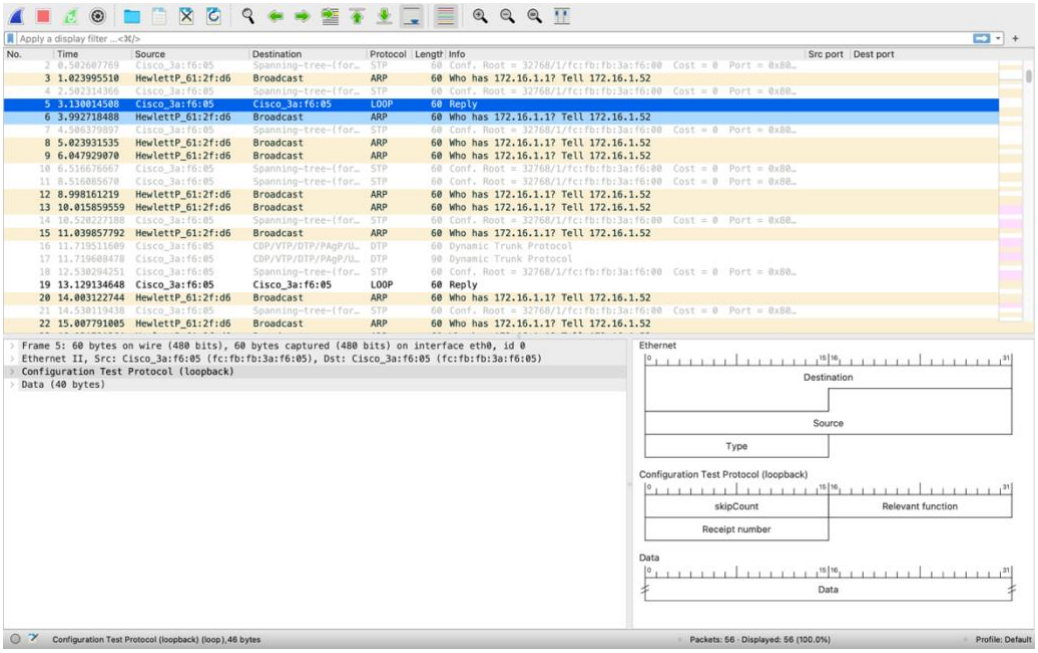

Figura 8 Exemplo de um pacote Loopback capturado com o wireshark

3.2. Experiência 2

Na experiência 2 propunha-se a configuração no *switch* de duas VLAN's (LAN's virtuais), **Vlan60** e **Vlan61**. Para o efeito, foi ligado o tux3 (eth0) e o tux4 (eth0) à VLAN60 e o tux2 (eth0) e tux4 (eth1) à Vlan61.

3.2.1. How to configure vlan60?

Configurar a vlan 60 exige ligar fisicamente a carta de rede eth0 do tux3 e a carta de rede eth1 do tux4 ao *switch.* Para esta experiência ligamos a carta de rede eth0 do tux3 à porta 3 do *switch* e a carta de rede eth1 do tux4 à porta 4.

Depois, ligou-se através da porta série o *switch* ao tux4 para que o mesmo possa ser configurado através deste tux e executou-se os seguintes códigos no terminal da posta série.

> configure terminal
> vlan 60
> end
> configure terminal
> interface fastethernet 0/(2/3) ('2' para a porta 2 do *switch* '3' para a porta 3)
> switchport mode access
> switchport access vlan 60
> end

3.2.2. How many broadcast domains are there? How can you conclude it from the logs?

Na rede configurada durante esta experiência existiam duas sub-redes distintas, ou seja, dois domínios de Broadcast também eles distintos. Estas duas sub-redes são consequência da criação das duas vlan's (vlan60 e vlan61).

Verificou-se também que fazendo ping Broadcast (ping -b 172.16.60.255) no tux3 este recebe uma resposta apenas do tux4 (172.16.60.254) e, fazendo ping Broadcast no tux2 (ping -b 172.16.61.255) este não recebe resposta de nenhum dos restantes tux's (está isolado na vlan na qual foi configurado)

Desta forma, os tux's 3 e 4 estão num domínio de Broadcast diferente do domínio onde se encontra o tux2, conforme era previsto.

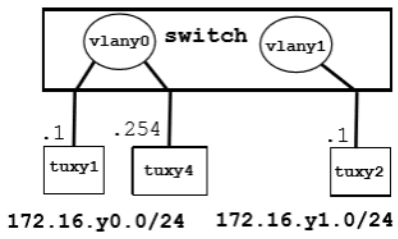

Figura 9 Esquema da rede configurada na experiência 2

3.3. Experiência 3

Na experiência 3 propunha-se a configuração e análise de um *router* em Linux. Este iria permitir a comunicação entre o tux2 e tux3 localizados em vlan's distintas.

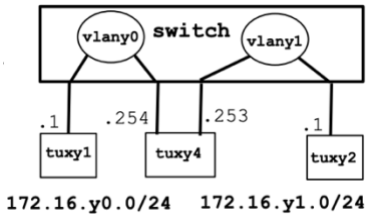

Figura 10 Esquema da rede configurada na experiência 3

3.3.1. What routes are there in the tuxes? What are their meaning?

No tux1:

- **172.16.61.0/24 via 172.16.60.254 dev eth0** – Esta rota tem como significado que qualquer comunicação com a vlan 61 será feita recorrendo ao tux4 como intermediário.
- **172.16.60.0/24 dev eth0** – significa que o tux1 está conectado à sub-rede 172.16.60.0/24 através da interface eth0.

No tux2:

- **172.16.60.0/24 via 172.16.61.253 dev eth0** - Esta rota tem como significado que qualquer comunicação com a vlan 60 será feita recorrendo ao tux4 como intermediário.
- **172.16.61.0/24 dev eth0** – significa que o tux2 está conectado à sub-rede 172.16.61.0/24 através da interface eth0

No tux4:

- **172.16.60/24 dev eth0** – significa que o tux4 está conectado à sub-rede 172.16.60.0/24 através da interface eth0
- **172.16.61.0/24 dev eth1** – significa que o tux4 está conectado à sub-rede 172.16.61.0/24 através da interface eth1

3.3.2. What information does an entry of the forwarding table contain?

Na tabela de *forwarding* está presente a seguinte informação:

- **Destination**: indica o endereço de IP de destino de um determinado pacote
- **Gateway**: indica o endereço de IP do próximo local por onde deve passar a rota
- **Netmask**: indica a máscara de rede correspondente à rede de destino, esta informação associada ao endereço de destino permite determinar o ID da rede de destino
- **Flags**: indica informações sobre a rota (como, por exemplo, a sua disponibilidade)
- **Metric**: indica o custo de cada rota
- **Ref**: indica o número de vezes que a rota foi referenciada para estabelecer uma ligação
- **Use**: indica o número de vezes que a rota foi vista
- **Interface**: indica o nome da interface localmente utilizada pela rota

3.3.3. What ARP messages, and associated MAC addresses, are observed and why?

Ao realizar um *ping* do tux1 para o tux2 este *ping* é redirecionado através do tux4. O tux 1 envia um *ARP packet* de modo a obter o endereço físico (MAC) do tux4 e, posteriormente, o tux4 envia também um pacote ARP para conhecer o endereço físico do tux2.

3.3.4. What ICMP packets are observed and why?

São observados tanto os pacotes de pedido (*request*) como as respetivas resposta (*reply*). Isto acontece porque já existe ligação entre as duas vlans. Na experiencia anterior (3.3.) tal ligação ainda não existia pelo que se tentássemos fazer um *ping* do tux3 para o tux2 (ou um ping broadcast para a sub-rede 172.16.61.0 onde o mesmo se encontrava) obteríamos a mensagem "*Network is unreachable*".

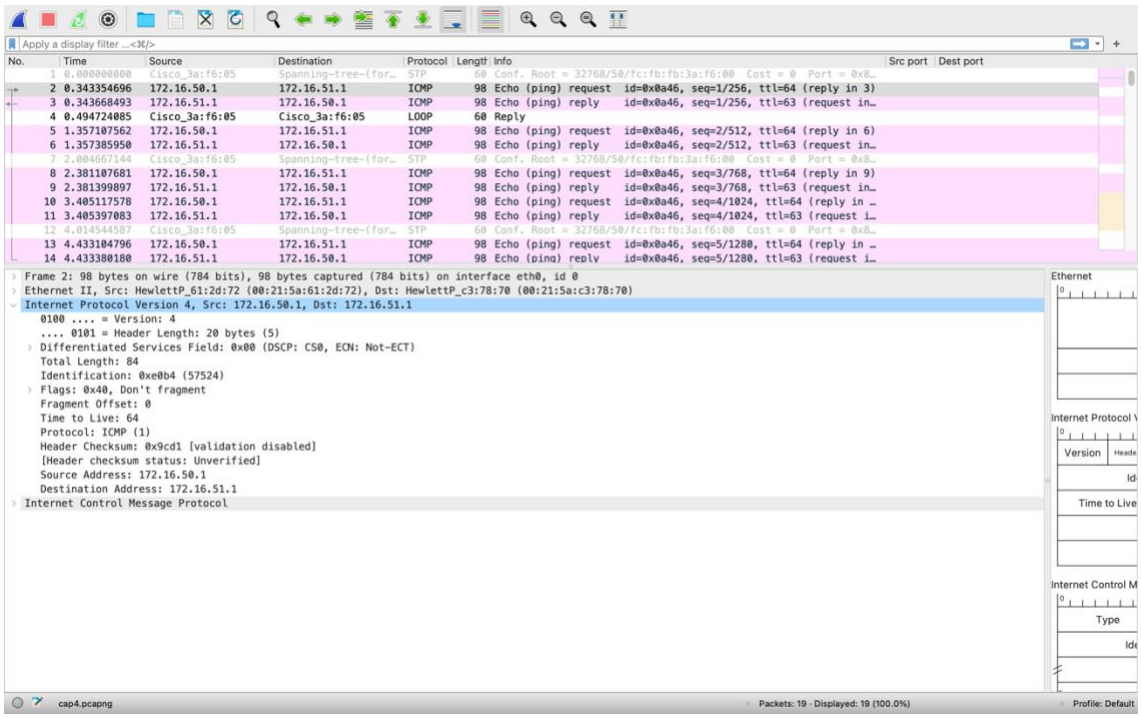

Figura 11 Pings entre o tux2 e o tux3

```
root@tux53:~# ping -b 172.16.51.0
connect: Network is unreachable
```

Figura 12 Exemplo do que seria visto no terminal tentando um ping entre sub-redes distintas na experiência anterior

3.3.5. What are the IP and MAC addresses associated to ICMP packets and why?

Os endereços IP e MAC associados com os pacotes ICMP são os endereços IP e MAC dos tux's de origem e destino (como é possível verificar na primeira figura apresentada na questão anterior (3.3.4.).

3.4. Experiência 4

3.4.1. How to configure a static route in a commercial router?

Uma rota estática num router comercial é configurada acedendo á consola do mesmo e, no caso do *router* CISCO presente no laboratório onde decorreram as aulas, utilizando a seguinte lista de comandos:

> Configure terminal
> ip route [endereço ip da rota de destino] [máscara] [IP gateway]
> exit

3.4.2. What are the paths followed by the packets in the experiments carried out and why?

O tux4 representa o router padrão para o tux1, por sua vez o *router* comercial representa o *router* padrão para o tux2 e 4. Qualquer ligação entre a vlan60 e vlan61 foi, então, realizada tendo o tux4 como intermediário (como já anteriormente acontecera). Já tentativas de ligação à rede do laboratório (172.16.1.0/24) foram redirecionadas para o *router* comercial, tal como se esperava, visto ser este o *router* padrão entre qualquer um dos tux's da vlan 61.

3.4.3. How to configure NAT in a commercial router?

Também tendo como referência o *router* utilizado nas aulas laboratoriais, o NAT foi configurado utilizando os seguintes comandos (fornecidos no guião do trabalho em questão):

> Ip nat pool ovrld 172.16.1.69 172.16.1.69 prefix 24
> Ip nat inside source list 1 pool ovrld overload
> Access-list 1 permit 172.16.60.0 0.0.0.7
> Access-list 1 permit 172.16.61.0 0.0.0.7

3.4.4. What does NAT do?

O NAT (*Network Address Translation*) é responsável pela tradução de endereços privados (não globalmente únicos) de uma rede interna para endereços públicos.

Deste modo, o NAT é útil para a função de segurança, visto que, numa determinada rede privada, apenas é necessário um endereço de IP público para conectar os restantes. Escondendo, assim, todo o restante da rede privada.

3.5. Experiência 5

Na experiência 5 proponha-se a configuração e análise do DNS (*Domain Name System*) em todos os tux'x. Um servidor DNS tem como responsabilidade armazenar os endereços IP públicos associados aos respetivos *hostnames*. É, então, utilizado para traduzir os *hostnames* para os respetivos endereços IP. No caso particular do nosso trabalho foi utilizado o servidor DNS **netlab.fe.up.pt**

3.5.1. How to configure the DNS service at an host?

A configuração do servidor DNS implica alterar o ficheiro resolv.conf, presente em /etc. Este ficheiro deve passar a conter respetivamente:

- **Search netlab.fe.up.pt**
- **Nameserver 172.16.1.2**

Onde, netlab.fe.up.pt indica o nome do servidor DNS, enquanto que 172.16.1.2 indica o endereço de IP do servidor DNS em questão.
Após a realização desta mesma experiência passa a ser possível aceder à internet nos tux's.

3.5.2. What packets are exchanged by DNS and what information is transported?

Primeiramente é enviado um pacote do *host* para o server que contem o *hostname* pretendido e do qual pretende-se saber o respetivo endereço IP. O Servidor responde então com um pacote contendo o IP desejado. A figura a seguir exemplifica um pedido ao servidor DNS capturado com o *wireshark.*

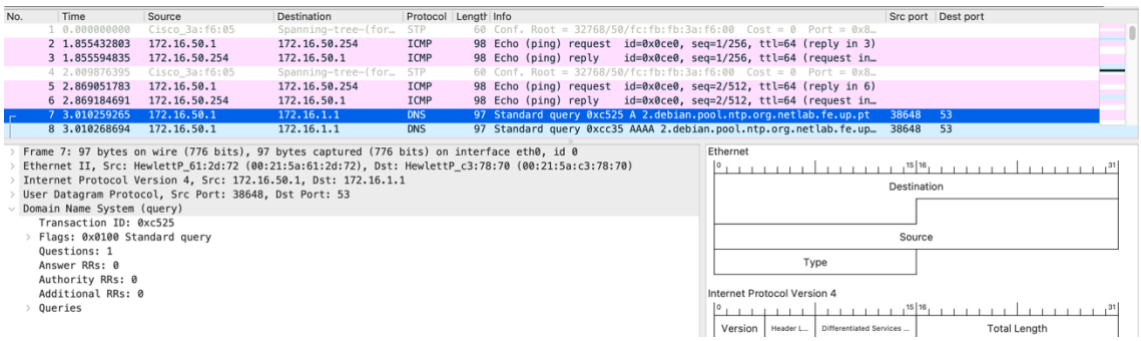

Figura 13 Exemplo de um pacote DNS capturado com o wireshark

3.6. Experiência 6

Na experiência 6 propunha-se observar e analisar o comportamento do protocolo TCP, recorrendo, para tal, à aplicação desenvolvida na parte 1 do trabalho (já previamente analisada e explicada detalhadamente na parte 1 deste guião).

3.6.1. How many TCP connections are opened by your ftp application?

A aplicação por nós desenvolvida abre duas ligações TCP, uma de controlo que permite enviar os comandos FTP ao servidor e receber as respetivas respostas, e outra de dados, para receber os dados do servidor.

3.6.2. In what connection is transported the FTP control information?

A informação de controlo é transportada através da conexão para esse efeito reservada identificada pela porta TCP **21.**

3.6.3. What are the phases of a TCP connection?

Uma ligação TCP pode ser divida em **3 fases.** Sendo as mesmas:

1. Estabelecimento da ligação;
2. Troca de dados;
3. Encerramento da ligação.

3.6.4. How does the ARQ TCP mechanism work? What are the relevant TCP fields? What relevant information can be observed in the logs?

O protocolo TCP utiliza o mecanismo ARQ do tipo **Go-Back-N**, utilizando mensagens de confirmação (ACKs) para reconhecer quais as tramas que foram enviadas corretamente e qual é o número de sequência esperado.

3.6.5. How does the TCP congestion control mechanism work? What are the relevant fields. How did the throughput of the data connection evolve along the time? Is it according the TCP congestion control mechanism?

O protocolo TCP utiliza uma janela de congestão de modo a limitar o número de bytes que podem ser enviados a cada instante. O tamanho desta janela deve ser determinado e para tal é utilizada uma estimativa resultante da análise da congestão da ligação. É para tal mecanismo implementado o algoritmo AIMD (*Additive Increase/Multiplicative Decrease*) de modo que o tamanho da janela aumente linearmente com a diminuição da congestão e que diminuía exponencialmente quando uma congestão é verificada.

3.6.6. Is the throughput of a TCP data connections disturbed by the appearance of a second TCP connection? How?

O surgimento de uma segunda ligação TCP provoca uma queda na taxa de transmissão uma vez que a taxa é distribuída de igual forma para cada ligação

4. Conclusões

A realização do trabalho proposto permitiu em ambas as partes consolidar os conhecimentos teóricos acerca do protocolo FTP (*File Transfer Protocol*) e das caraterísticas de uma rede de computadores e também ganhar destreza e conhecimento prático acerca do *hardware* utilizado para configurar a rede de computadores no laboratório nomeadamente o *Router* e o *Switch*.

No final do mesmo, consideramos benéfico para o desenvolvimento das nossas capacidades no âmbito da unidade curricular a realização deste trabalho.

Devido à incapacidade de utilizar a nossa aplicação quando compilada nos computadores do laboratório, a mesma foi demonstrada num computador pessoal onde funcionou com sucesso cumprindo os requisitos necessários. Por usa vez, a Rede de Computadores foi testada utilizando o protocolo *telnet* através de dois terminais do computador, esta solução permitiu-nos demonstrar no laboratório que todos os requisitos da rede estavam também cumpridos.

Contudo, numa eventual continuação do trabalho, ou caso o tempo para realizar o mesmo fosse maior, seria possível organizar o código da aplicação de download num conjunto maior de ficheiros de modo a aumentar a sua organização separando as funções em ficheiros segundo a sua funcionalidade sem que, deste modo, se aumentasse desnecessariamente a complexidade da arquitetura do mesmo (por exemplo, as funções relativas ao DNS estarem num ficheiro isolado e dedicado apenas a estas). Quanto à segunda parte do trabalho seria interessante realizar um conjunto de funções *bash* para automatizar todo o processo de configuração da rede.

5. Anexos

5.1. Anexo 1 – Código Fonte

5.1.1. Main.c

```c
#include "args.h"
#include "connection.h"

#define TCP_PORT 21

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

    /* Verifica se recebeu input */
    if(argc != 2){
        fprintf(stderr, "usage: download ftp://[<user>:<password>@]<host>/<url-path>\n");
        exit(1);  // exit(1) -> Exit Failure
    }


    connectionArgs args;    // struct to save arguments
    int socketfd_data;      // socket file descriptor to data
    int socketfd_control;   // socket file descriptor to control
    char urlcpy[2*SIZE];    // URL copy (2*SIZE = 256)
    char command[2*SIZE];   // Command :) (2*SIZE = 256)
    char rd[8*SIZE];        // (8*SIZE = 1024)

    strcpy(urlcpy, argv[1]);

    /* "Reset" the parameters */
    memset(args.file_name, 0, 128);
    memset(args.host, 0, 128);
    memset(args.host_name, 0, 128);
    memset(args.ip, 0, 128);
    memset(args.password, 0, 128);
    memset(args.url_path, 0, 128);
    memset(args.user, 0, 128);

    /* processing de arguments */
    if(parseArgs(argv[1], &args) == ERROR){
        printf("usage: %s ftp://<user>:<password>@<host>/<url-path>\n", argv[0]);
```

```
    exit(1);
}
/* Print the arguments */
printArgs(&args);

/* initialize the client */
if(client_init(args.ip, TCP_PORT, &socketfd_control) == ERROR){
    printf("Error: clint_init() control\n");
    exit(1);
}
if(readResponse(socketfd_control, rd, sizeof(rd)) == ERROR){
    printf("Error: readResponse() /1\n");
    exit(1);
}

sprintf(command, "user %s\r\n", args.user);

/* Send command */
if(clientCommand(socketfd_control, command) == ERROR){
    printf("Error: clientCommand() /1\n");
    exit(1);
}
if(readResponse(socketfd_control, rd, sizeof(rd)) == ERROR){
    printf("Error: readResponse() /1\n");
    exit(1);
}
//printf("\n\naqui3\n\n");
sprintf(command, "pass %s\r\n", args.password);

if(clientCommand(socketfd_control, command) == ERROR){
    printf("Error: clientCommand() /x\n");
    exit(1);
}
if(readResponse(socketfd_control, rd, sizeof(rd)) == ERROR){
    printf("Error: readResponse() /x\n");
    exit(1);
}

char ip[32];
int port;
```

```
sprintf(command, "pasv\r\n");

if(clientCommand(socketfd_control, command) == ERROR) {
    printf("Error: clientCommand() /2\n");
    exit(1);
}
if(pasvMode(socketfd_control, ip, &port) == ERROR) {
    printf("Error: pasvMode()\n");
    exit(1);
}

printf("IP: %s\nPort Number: %d\n", ip, port);

if((client_init(ip, port, &socketfd_data)) == ERROR){
    printf("Error: client_init() [data]\n");
    exit(1);
}
sprintf(command, "retr %s\r\n", args.url_path);

if(clientCommand(socketfd_control, command) == ERROR){
    printf("Error: clientCommand() /3\n");
    exit(1);
}
if(readResponse(socketfd_control, rd, sizeof(rd)) == ERROR) {
    printf("Error: readResponse() /2\n");
    exit(1);
}

if(writeFile(socketfd_data, args.file_name) == ERROR) {
    printf("Error: writeFile()\n");
    exit(1);
}

sprintf(command, "quit\r\n");
if(clientCommand(socketfd_control, command) == ERROR){
    printf("Error: clientCommand() /4\n");
    exit(1);
}
if(readResponse(socketfd_control, rd, sizeof(rd)) == ERROR) {
    printf("Error: readResponse() /3\n");
```

```
        exit(1);
    }

    close(socketfd_control);
    close(socketfd_data);

    printf("\n\n");
    printf("#############################################################\n");
    printf("######################---> Done <---######################\n");
    printf("#############################################################\n");
    printf("\n\n");

    exit(0);     // exit(0) -> Exit Success
}
```

5.1.2. Args.h

```
#ifndef ARGS_H
#define ARGS_H

#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <strings.h>
#include <arpa/inet.h>
#include <netdb.h>

#define SIZE 128
#define OK 1
#define ERROR -1

typedef struct connectionArgs{
    char user[SIZE];
```

```
    char password[SIZE];
    char host[SIZE];
    char url_path[SIZE];
    char file_name[SIZE];
    char host_name[SIZE];
    char ip[SIZE];
} connectionArgs;

typedef enum {
    USERNAME,
    PASSWORD,
    HOSTNAME,
    FILE_PATH
} States;

/**
 * @brief This function processes the URL, separating it into different elements
 *        and storing them in the respective places in the structure (connectionArgs *args)
 *
 * @param url pointer to url string
 * @param args pointer to the struct where the information will be stored
 * @return ERROR (-1) on error; OK (1) on sucess
 */
int parseArgs(char *url, connectionArgs *args);


/**
 * @brief Get the Ip object
 *
 * @param host pointer to the string with the host from where IP will be search
 * @param args pointer to the struct with the information
 * @return ERROR (-1) on error; 0 on sucess
 */
int getIp(char *host, connectionArgs *args);

/**
 * @brief Get the File Name object
 *
 * @param args pointer to the struct with the information
 * @return ERROR (-1) on error; 0 on sucess
```

```
 */
int getFileName(connectionArgs *args);


/**
 * @brief This function prints the arguments inside the struct (connectionArgs *args)
 *
 * @param args pointer to the struct to print
 */
void printArgs(connectionArgs *args);


#endif
```

5.1.3. Args.c

```
#include "args.h"

int parseArgs(char *url, connectionArgs *args){
    if(!args){
        printf("Error parseArgs() invalid parameters\n");
        return ERROR;
    }


    char ftp_[] = "ftp://"; // "parte inicial da string"
    for(int i = 0; i < 6; i++){
        if(url[i] != ftp_[i]){
            printf("Usage: ftp://\n");
            return ERROR;
        }
    }

    int index = 6;
    int length = strlen(url);
    char curr_char;
    int arg_index = 0;
    States state = USERNAME;

    /* modo "anonimo" */
```

```
  if (strchr(url, '@') == NULL)  // strchr retorna apontador para o lugar onde encontrou o char ou null senao o encontrar
  {
    //printf("aqui fixe\n");
    strncpy(args->user, "anonymous", 10);
    strncpy(args->password, "1234", 5); //qualquer coisa random
    state = HOSTNAME;
  }

  while(index < length){
    curr_char = url[index];
    switch (state)
    {
    case USERNAME:
      if(curr_char == ':'){
        args->user[arg_index] = '\0';
        arg_index = 0;
        state = PASSWORD;
      }
      else{
        args->user[arg_index] = curr_char;
        arg_index++;
      }
      break;
    case PASSWORD:
      if(curr_char == '@'){
        args->password[arg_index] = '\0';
        arg_index = 0;
        state = HOSTNAME;
      }
      else{
        args->password[arg_index] = curr_char;
        arg_index++;
      }
      break;
    case HOSTNAME:
      if(curr_char == '/'){
        args->host[arg_index] = '\0';
        arg_index = 0;
        state = FILE_PATH;
```

```
            }
            else{
                args->host[arg_index] = curr_char;
                arg_index++;
            }
        break;
        case FILE_PATH:
            args->url_path[arg_index] = curr_char;
            arg_index++;
            break;
        }
        index++;
    }

    if(getIp(args->host, args) != 0){
        printf("Error getIp()\n");
        return ERROR;
    }

    if(getFileName(args) != 0){
        printf("Error getFileName()\n");
        return ERROR;
    }

    return OK;
}

int getIp(char *host, connectionArgs *args){
    if(!args){
        printf("Error getIp() invalid parameters\n");
        return ERROR;
    }
    struct hostent *h;

    if((h = gethostbyname(host)) == NULL){
        herror("Error getting host name\n");
        return ERROR;
    }

    strcpy(args->host_name, h->h_name);
```

```
    strcpy(args->ip, inet_ntoa( *( ( struct in_addr *)h->h_addr) )); //32 bits Internet addr with net ordenation for dotted decimal notation

    return 0;
}

int getFileName(connectionArgs *args){
    if(!args){
        printf("Error getFileName() invalid parameters\n");
        return ERROR;
    }
    char fullpath[256];

    for(int i = 0; i < strlen(args->url_path); i++) fullpath[i] = args->url_path[i];
    //strcpy(fullpath, args->url_path);



    char* aux = strtok(fullpath, "/");

    while(aux != NULL){
        for(int i = 0; i < (strlen(aux)-1); i++) args->file_name[i] = aux[i];
        //strcpy(args->file_name, aux);
        aux = strtok(NULL, "/");
    }

    return 0; //returns 0 if success
}

void printArgs(connectionArgs *args){
    if(!args){
        printf("Error printArgs() invalid parameters\n");
        return;
    }
    printf("Information\n\n");
    printf("[Username: %s]\n", args->user);
    printf("[Password: %s]\n", args->password);
    printf("[Host: %s]\n", args->host);
    printf("[HostName: %s]\n", args->host_name);
    printf("[FileName: %s]\n", args->file_name);
```

```
    printf("[UrlPath: %s]\n", args->url_path);
    printf("[IpAdress: %s]\n\n\n", args->ip);
    return;
}
```

5.1.4. Connection.c

```
#include "connection.h"

int client_init(char *ip, int port, int *socketfd){ //dado

    struct sockaddr_in server_addr;

    //server addr handling
    bzero((char*)&server_addr,sizeof(servaer_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr(ip);    // 32 bit Internet address network byte ordered
    server_addr.sin_port = htons(port);             // server TCP port must be network byte ordered

    //open TCP connection
    if((*socketfd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
        perror("socket()");
        return ERROR;
    }

    //connect to server
    if(connect(*socketfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0){
        perror("connect()");
        return ERROR;
    }

    return OK;
}

int clientCommand(int socketfd, char * command){    //client sends commands to socket to be received and
read in server
    int tam;
    if((tam = write(socketfd, command, strlen(command))) <= 0){
        printf("Error: command not sent\n");
        return ERROR;
```

```
    }
    printf("< %s\n", command);
    return OK;
}

int pasvMode(int socketfd, char *ip, int *port){    //pasvMode command reception (saves IP and Port number)
    char buf[1024];


    if(readResponse(socketfd, buf, sizeof(buf)) == ERROR){
        printf("Error: cannot enter passive mode\n");
        return ERROR;
    }

    //< 227 Entering Passive Mode (193,136,28,12,19,91), example
    //printf("\n\nbud antes: %s \n\n", buf);
    //printf("......");puts(buf);

    strtok(buf, "("); // pega no buf que tem< 227 Entering Passive Mode (193,136,28,12,19,91) e corta ate 193,136,28,12,19,91)

    //printf("\n\nbuf apos: %s \n\n", buf);
    char* ip1 = strtok(NULL, ",");
    //printf("......");puts(ip1);
    char* ip2 = strtok(NULL, ",");
    char* ip3 = strtok(NULL, ",");
    char* ip4 = strtok(NULL, ",");
    //guarda os numeros nos ips... ip 1 = 193, ip2 = 136 etc

    //printf(":::::");
    sprintf(ip, "%s.%s.%s.%s", ip1, ip2, ip3, ip4);

    char* aux1 = strtok(NULL, ",");
    char* aux2 = strtok(NULL, ")");

    *port = (256 * atoi(aux1)) + atoi(aux2);

    return OK;
}
```

```
int readResponse(int socketfd, char* rd, size_t size){  //reading the different commands responses from server to client
    char aux[3];
    long num;
    FILE* fd;

    if(!(fd = fdopen(socketfd, "r"))){
        printf("Error opening file to read\n");
        return ERROR;
    }

    do{
        memset(rd, 0, size);
        rd = fgets(rd, size, fd);
        printf("%s", rd);
    } while(rd[3] != ' ');

    strncpy(aux, rd, 3);
    num = atoi(aux);

    //printf("num: %ld", num); //debug

    if(num > 420 && num < 554){
        printf("Command error - «400» (command not accepted and requested action did not take place) or «500» (syntax error, command unrecognized and the requested action did not take place)\n");
        return ERROR;
    }

    return OK;
}


int writeFile(int socketfd, char* filename){    //writing on socketfile the communication between client and server
    int bytes;
    char buf[1024];

    FILE* fd;
```

```
    if(!(fd = fopen(filename, "w"))){
        printf("Error opening file to write\n");
        return ERROR;
    }

    while((bytes = read(socketfd, buf, sizeof(bytes))) > 0){
        if(bytes < 0){
            printf("Error: Nothing received from data socket\n");
            return ERROR;
        }
        if((fwrite(buf, bytes, 1, fd)) < 0){
            printf("Error writting file in socketfile\n");
            return ERROR;
        }
    }

    fclose(fd);

    return OK;}
```

5.1.5. Connection.h

```
#ifndef CONNEC_H
#define CONNEC_H

#include "args.h"

/**
 * @brief Initiate the connection between client and server (by creating a connection socket)
 *
 * @param ip pointer to the IP
 * @param port port number
 * @param socketfd pointer to socket file descriptor
 * @return ERROR (-1) on error; OK (1) on sucess
 */
int client_init(char *ip, int port, int *socketfd);

/**
 *
 *
```

```
 * @param socketfd pointer to socket file descriptor
 * @param command
 * @return ERROR (-1) on error; OK (1) on sucess
 */
int clientCommand(int socketfd, char * command);

/**
 *
 *
 * @param socketfd pointer to socket file descriptor
 * @param ip pointer to the IP
 * @param port port number
 * @return ERROR (-1) on error; OK (1) on sucess
 */
int pasvMode(int socketfd, char *ip, int *port);


/**
 *
 *
 * @param socketfd pointer to socket file descriptor
 * @param rd
 * @param size size of rd
 * @return ERROR (-1) on error; OK (1) on sucess
 */
int readResponse(int socketfd, char* rd, size_t size);

/**
 *
 *
 * @param socketfd pointer to socket file descriptor
 * @param filename
 * @return ERROR (-1) on error; OK (1) on sucess
 */
int writeFile(int socketfd, char* filename);

#endif
```