U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

MIEIC 3º Ano

RCOM

Dezembro 2020

Configuração e Estudo de uma Rede IP

Relatório

(2º Trabalho Laboratorial)

Turma 1 - Grupo 6

Rui Filipe Mendes Pinto - up201806441

Tiago Gonçalves Gomes – up201806658

Sumário

Este relatório foi elaborado no sentido de descrever os passos relativos às experiências realizadas no segundo trabalho laboratorial, cujo objetivo principal é a configuração e estudo de uma rede de computadores. No final, é executada uma aplicação programada em *C*, que realiza o *download* de um ficheiro de um servidor *FTP* (*File Transfer Protocol*), verificando assim o bom funcionamento das ligações estabelecidas na rede, sendo depois necessário analisar vários aspetos relativos ao mesmo.

As experiências referidas previamente, baseiam-se na configuração de IPs para diversos membros da rede, dentro dos quais um *Router* e um *Switch*, configuração do *DNS* (*Domain Name System*), implementação de duas *VLANs* (*Virtual Local Area Network*) no *switch* e do *NAT* (*Network Address Translation*).

Todos os objetivos propostos para este trabalho foram concluídos com sucesso.

Introdução

O trabalho, tal como referido, tem dois objetivos principais: a configuração de uma rede e o desenvolvimento de uma aplicação de *download*.

Relativamente à configuração de uma rede, o objetivo principal focou-se na comunicação entre duas *VLANs* ligadas a um *Switch* que por sua vez estavam ligadas a um *Router* que estabelecia a ligação à internet utilizando *NAT*. Seguidamente, foi desenvolvida uma aplicação de *download* relativa ao protocolo *FTP*, através de ligações *TCP* criadas pelas *Berkeley Sockets*.

No que diz respeito ao relatório, o seu objetivo é expor os aspetos relacionados com:

- Parte 1: Aplicação de *Download*
  - Arquitetura e resultados obtidos.
- Parte 2: Configuração e Estudo de uma Rede
  - Aspetos relacionados com cada experiência.
- Conclusões
  - Análise final à informação apresentada.

É de notar que a realização da primeira experiência ocorreu na bancada 1, enquanto que as restantes ocorreram na bancada 4.

Parte 1: Aplicação de *Download*

A primeira parte deste projeto foi desenvolver uma aplicação de *download* de ficheiros usando o protocolo FTP, que adota a sintaxe URL **ftp://[<user>:<password>@]<host>/<url-path>**. Para a realização da mesma foram consultados o RFC959 (*Request for Comments)* que retrata o protocolo *FTP* e o RFC1738 relativo à formatação dos *URLs*, neste caso o do protocolo *FTP*.

Arquitetura

Primeiramente, é feito o *parsing* dos argumentos do programa na função ***parseArguments***, nomeadamente do URL. Guardamos as variáveis *user*, *password*, *host*, *url-path, filename* numa *struct FTPclientArgs*, sendo que o nome do ficheiro é obtido a partir da variável *path*. Adicionalmente, usamos a função ***getIP*** para preenchermos uma *struct hostent*, a partir do *hostname*, que a traduz num endereço IP (permitindo a sua localização num *host* com domínio determinado), através da função ***gethostbyname*** que irá estabelecer uma comunicação com um servidor *DNS*, de modo a poder fazer essa tradução. A porta usada é a 21, uma vez que está associada ao protocolo *FTP*.

As funções principais são a ***readResponse*** que se baseia numa máquina de estados e analisa a resposta dada pelo servidor a cada comando, retornando a parte principal da mensagem e o *status code* da mesma que depois servirá para verificar se existiu algum erro ou não. O primeiro dígito do *status code* é utilizado em várias partes do programa e está associado a vários significados distintos. *Status* codes começados em: **1** indicam que o servidor irá enviar mais informação e o ***readResponse*** irá ser chamado novamente; **2** sucesso; **3** o servidor necessita de mais informação; **4** o comando não foi aceite mas podemos enviar de novo; **5** ocorreu um erro levando ao término do programa.

O *flow* do programa começa com a abertura de um *socket* no qual será estabelecida uma conexão entre o cliente e o servidor. De seguida são enviados os comandos necessários para a realização do login: **USER user** com o *user* definido em cima (*anonymous* caso contrário) e a **PASS password** também definida em cima (123 caso contrário).

Seguidamente, é enviado o comando **PASV** que indica a entrada em modo passivo, sendo que o servidor irá informar o cliente do endereço IP e porta a que o mesmo se deve conectar para iniciar a troca de dados através de outro *socket*.

Em seguida é enviado o comando **RETR url-path** para pedir o ficheiro e após isso é feito o *download* do mesmo utilizando a função ***readServerData*** que constrói ficheiro de destino *byte* a *byte* e apresenta regularmente a percentagem de *download* do ficheiro. No final são libertadas ambas as conexões: transferência de comandos e de dados.

Resultados

A nossa aplicação de *download* foi testada em diversas condições: modo anónimo e não anónimo, ficheiros de vários tipos e tamanhos (até 100MB). Além disso, lidamos também com casos de erros, inexistência do ficheiro, mal formatação da URL passada inicialmente ao programa, etc. Um exemplo da aplicação em funcionamento pode ser consultado na figura 1 do anexo I.

Parte 2: Configuração de Rede e Análise

Nesta parte do projeto, tivemos de fazer um conjunto de experiências no laboratório, com o objetivo final de configurar uma rede onde seria usada a aplicação de *download* desenvolvida na primeira parte.

Experiência 1 – Configurar um IP de rede

O objetivo desta experiência foi a compreensão da configuração de IPs em máquinas diferentes, de modo a que estas consigam comunicar entre si. Para tal, após a configuração dos IPs da carta *eth0* de dois computadores (*tux3* e *tux4*), foi enviado o sinal *ping* de um computador para o outro, para verificar que, de facto, as máquinas tinham uma ligação entre si.

Os principais comandos utilizados nesta experiência foram:

- *ifconfig < cartaDeRede > < endereçoIP > / < Nr. de bits da máscara >* – atribuição de um endereço IP a uma máquina. (ex.: ifconfig eth0 172.16.40.1/24).
- *ping < endereçoDestino >* – fazer ping de uma máquina para outra, com o endereço IP *endereçoDestino*.

O ARP (*Address Resolution Protocol*) é um protocolo de comunicação que serve para descobrir o endereço da camada de ligação associado ao endereço IPv4, ou seja, serve para mapear o endereço de rede a um endereço físico, como o endereço MAC (*Media Access Control*) que é um endereço único de *hardware* e identifica um nó numa rede . Para controlar esse mapeamento, é mantida uma tabela ARP em cada *tux*, sendo que sempre que um novo pacote com endereços MAC ou IP chega e ainda não estão na tabela ARP, o protocolo modifica-a com os novos dados, adicionando uma nova entrada com os dados relativos ao IP e endereço MAC *source* do pacote. Os pacotes ARP são utilizados quando um *host* deseja obter um endereço físico (endereço MAC) de outro *host*, tendo apenas o seu endereço IP.
Assim, quando um *tux* necessita de enviar um pacote para um determinado IP, apenas precisa de consultar a sua tabela ARP e enviar para o endereço MAC correspondente. Caso não exista uma entrada relativa ao endereço IP referido, é enviado um pedido ARP em *broadcast* para todos os computadores na mesma rede que basicamente funciona como uma pergunta: “Alguém sabe onde está o IP (X.X.X.X) ?” , sendo a resposta o endereço MAC para onde o computador deverá enviar o pacote, caso haja algum computador da rede que saiba como chegar àquele nó da rede. No contexto da experiência 1, é de notar que a pergunta referida vem na forma de um pacote ARP com o endereço IP e endereço MAC do *tux* que faz a pergunta (*tux3*) 172.16.10.1 e 00:21:5a:61:2d:ef respetivamente e com o endereço IP do *tux* alvo (*tux4)* 172.16.10.254.

Como não se sabe o MAC do *tux* alvo, este está registado como 00:00:00:00:00:00 (consultar figura 1 do anexo II). No pacote da resposta (consultar figura 2 do anexo II), o endereço IP e MAC de origem são do *tux4* (172.16.10.254 e 00:22:64:a6:a4:f8 respetivamente) e o endereço IP e MAC de destino são do *tux3* (172.16.10.1 e 00:21:5a:61:2d:ef).

O comando *ping* gera primeiro pacotes ARP para obter os endereços MAC, caso o endereço físico não esteja na tabela ARP, e de seguida gera pacotes ICMP (*Internet Control Message Protocol*). O protocolo ICMP é usado por dispositivos de rede para gerar mensagens de erro quando existem problemas de rede que não permitem a transferência de pacotes IP.

Os pacotes ICMP contêm o endereço MAC e IP do transmissor e do recetor, e são usados para efeitos de encaminhamento. Quando fazemos um *ping* ao *tux*4, a partir do *tux3* os endereços IP e MAC dos pacotes (origem e destino), vão ser os destes *tux's*.

*Request Packet (*figura 3 do anexo II):

- Endereço MAC de origem: 00:21:5a:61:2d:ef
- Endereço MAC de destino: 00:22:64:a6:a4:f8
- Endereço IP de origem: 172.16.10.1
- Endereço IP de destino: 172.16.10.254

*Reply Packet (*figura 4 do anexo II):

- Endereço MAC de origem: 00:22:64:a6:a4:f8
- Endereço MAC de destino: 00:21:5a:61:2d:ef
- Endereço IP de origem: 172.16.10.254
- Endereço IP de destino: 172.16.10.1

Analisando o *Ethernet header* de um pacote, conseguimos determinar o tipo de trama. Caso o *EtherType* (composto por 2 octetos) tenha o valor 0x0800, esta trama é do tipo IP. Sabendo que é do tipo IP, analisando o *header* conseguimos concluir que caso tenha o seu valor a 1, o tipo de protocolo é ICMP. Por outro lado, se o valor do *EtherType* for 0x0806, a trama é do tipo ARP (consultar figuras 5 e 6 do anexo II).

Para determinar o tamanho de uma trama recebida, temos de inspecionar o pacote utilizando o wireshark, como demonstrado na figura 7 do anexo II.

A interface *loopback* é uma interface virtual da rede que permite ao computador receber respostas de si mesmo e é usada para testar se a carta de rede está configurada corretamente (consultar figura 8 do anexo II).

Experiência 2 – Implementação de duas VLANs num switch

Nesta experiência, foram criadas duas VLANs no *switch*, sendo que duas máquinas (*tux3* e *tux4*) foram ligadas a uma VLAN (*vlan40*), e uma outra máquina (*tux2*) à outra VLAN (*vlan41*).

O objetivo foi compreender como é efetuada a configuração destas VLANs e como é que estas permitem e influenciam a troca de informação entre as máquinas.

Configuração da *vlan40 e vlan 41*:

Na régua 1 da bancada, a porta T4 tem de estar ligada à porta *switch console* da régua 2. A porta T3 da régua 1, estará ligada à porta S0 do *tux* que se deseja estar ligado à consola do *switch*. Deste modo é possível executar os comandos no *GTKTerm* com o objetivo de configurar o *switch*. Os mesmos foram:

- Criar as VLANs:
  - *configure terminal*
  - *vlan 40*
  - *end*
  - *configure terminal*
  - *vlan 41*

- *end*

- Adicionar as portas às VLANs:
  - *configure terminal*
  - *interface fastethernet 0/X* (sendo X o número da porta, no *switch*, à qual o computador em questão está ligado)
  - *switchport mode access*
  - *switchport access vlan 4Y* (sendo Y o número da VLAN à qual se deve ligar o computador em questão)
  - *end*

Neste procedimento, existem 2 domínios de transmissão em *broadcast*, uma vez que o *tux3* recebe resposta do *tux4* quando faz *ping* em *broadcast*, mas não do *tux2* (consultar figura 9). O *tux2* não recebe nenhuma resposta quando faz *ping* em *broadcast*, tal como é possível observar na figura 10 do anexo II. Portanto, os 2 domínios são o que contém os *tux3* e *tux4* e o que contém o *tux2*.

Experiência 3 – Configuração de um router em Linux

Nesta experiência, o *tux4* foi configurado de modo a funcionar como um *router*, que está ligado a ambas as VLANs configuradas na experiência anterior. Para que isso acontecesse foi necessário configurar a carta *eth1* do mesmo com o endereço 172.16.y1.253. Isto permite a conexão e transmissão de informação entre máquinas em VLANs diferentes (*tux3* e *tux2*).

Os principais comandos utilizados nesta experiência foram:

- *ifconfig*, *route* (comandos para a configuração do router).
- *echo 1 > /proc/sys/net/ipv4/ip_forward* – ativar o IP *forwarding* para permitir que o *tux* funcione como um *router* e dê continuidade aos pacotes.
- *echo 0 > /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts* – desativar o *ignore broadcasts.*

As rotas existentes nos tuxes são as seguintes (resultado do *route -n*):

- Rotas *tux2*
  - vlan 40 (172.16.40.0) pela *gateway* 172.16.41.253 (pacotes para a rede 172.16.y0.0 deverão ir pelo *tux4 - eth1*).
  - vlan 41 (172.16.41.0) pela *gateway* 0.0.0.0 (pacotes para a rede 172.16.y1.0 não têm de atravessar uma *gateway*).
- Rotas *tux3*
  - vlan 41 (172.16.41.0) pela *gateway* 172.16.40.254 (pacotes para a rede 172.16.y1.0 deverão ir pelo *tux4 - eth0*).
  - vlan 40 (172.16.40.0) pela *gateway* 0.0.0.0 (pacotes para a rede 172.16.y0.0 não têm de atravessar uma *gateway*).
- Rotas *tux4*
  - vlan 40 (172.16.40.0) pela *gateway* 0.0.0.0
  - vlan 41 (172.16.41.0) pela *gateway* 0.0.0.0

Acerca das rotas do *tux4*, pacotes para a rede 172.16.y0.0 e 172.16.y1.0 não têm de atravessar qualquer *gateway*, uma vez que o *tux4* pertence a ambas e funciona como um *router* entre as mesmas.

As rotas obtidas anteriormente têm o seguinte formato (tabela de *forwarding)*:

- **Destination:** o destino da rota.
- **Gateway:** o IP do próximo ponto por onde a rota passará.

- **Netmask:** usado para determinar o ID da rede a partir do endereço IP do destino.
- **Flags:** dá-nos informação sobre a rota.
- **Metric:** o custo de cada rota.
- **Ref:** número de referências para esta rota (não usado no *kernel* do Linux).
- **Use:** contador de pesquisa da rota, dependendo do uso de -F ou -C, isto vai ser o número de falhas da cache (-F) ou o número de sucessos (-C).
- **Interface:** qual a placa de rede responsável pela *gateway* (eth0 ou eth1).

Quando um *tux* efetua um *ping* a outro e não sabe para onde enviar o pacote, começa por enviar um *ARP Request* "perguntando" para onde deve enviar o pacote. No caso da experiência, é efetuado um *ping* do *tux3* para o *tux2*. Como as tabelas *ARP* haviam sido previamente apagadas, o *tux3* não sabe para onde enviar o pacote. Assim, como referido, envia um *ARP Request* em *broadcast* "perguntando" quem é que consegue chegar ao *tux2* . O *tux4* responde com o endereço MAC da carta *eth0* dizendo que sabe como chegar lá (consultar figura 11 do anexo II). Posteriormente também o *tux4* não sabe como chegar ao *tux2* então também ele efetua um *ARP Request* em *broadcast* (consultar figura 12 do anexo II). Por fim, o *tux2* responderá com o seu MAC address, pois ele próprio é o *tux* de destino e a mensagem será entregue com sucesso. No ato da resposta, como as tabelas *ARP* dos 3 *tuxs* já foram previamente preenchidas, o processo será bastante mais veloz, uma vez que a rota já está completamente definida nessa altura.

Sendo assim, os pacotes ICMP observados serão os de *request* e *reply*, uma vez que após adicionadas todas as rotas, todos os *tux*'s conseguem ver onde cada um está.

Caso contrário, se após o processo do protocolo ARP um dos *tux's* fosse incapaz de chegar a outro, a mensagem *ICMP* de *Host Unreachable* seria enviada.

Por fim, os endereços IP e MAC associados a um pacote ICMP são os endereços de IP e MAC de origem e destino. Por exemplo, quando se faz um *ping* do *tux3* para o *tux2*, os endereços de origem serão 172.16.40.1 (IP *tux3*) e 00:21:5a:61:2f:d4 (MAC *tux3*) e de destino 172.16.41.1 (IP *tux2*) e 00:21:5a:5a:7b:ea (MAC *tux4* - *eth0*, pois é a *gateway* entre a rota estabelecida).

Experiência 4 – Configurar um router comercial e implementar NAT

Esta experiência teve como objetivo a configuração de um router comercial, efetuando a ligação deste à rede do laboratório (172.16.1.0/24) e também à *vlan41* (bancada 4). Foi também efetuada a configuração do router de modo a este utilizar a técnica NAT para garantir a conexão entre as máquinas, rede IP e a internet. Desta forma, além de termos que compreender a maneira em que o NAT influencia a conexão entre as máquinas e a Internet, tivemos também que analisar e perceber o mecanismo de redirecionamento de pacotes ICMP.

Os principais comandos usados nesta experiência foram os comandos de configuração do router, tal como nas experiências anteriores, e da NAT (slides 45 e 46 do guião), bem como os comandos *route* e *traceroute*.

De modo a configurar uma rota estática num router comercial, é necessário iniciar sessão no router através do *GTKTerm*. Para tal, partindo da configuração da experiência 2, é apenas necessário trocar a ligação da porta T4 da régua 1 do *switch console* para o *router console*. Para configurar as rotas temos que executar o comando *ip route* dentro do *GTKTerm*. Este comando segue a seguinte estrutura: *ip route prefix mask { ip-address | interface-type interface-number [ip-address]}*.

Exemplo de adicionar uma rota ao router:

- *conf t*
- *ip route [destino] [máscara] [gateway]*
- *exit*

Relativamente a esta experiência, após a adição das rotas *default* para cada um dos *tux's* executaram-se três sub-experiências principais.

Inicialmente fez-se *disable* dos *redirects* no *tux2* e removeu-se a rota que o mesmo tinha para a rede 172.16.y0.0/24 via *tux4*. Em seguida fez-se um *ping* para o *tux3* e, como é possível ver na figura 13 do anexo II, a rota agora utilizada passa pelo *router* que por sua vez direciona para o *tux4* e do *tux4* para o *tux3*. Este processo decorre em todos os pacotes ICMP enviados, onde o *router* informa o *tux2* de que existe um caminho melhor para enviar o pacote sem ser pelo *router* (e sim diretamente para o *tux4*), mas o *tux2* acaba sempre por enviar o pacote para o *router*. Neste processo são trocadas mensagens do tipo ICMP *redirect*, mensagens essas que são utilizadas pelos *routers* para notificar os *hosts* de que uma rota melhor está disponível para um destino particular. Como se pode ver, através do *log*, o *tux2* nunca "memoriza" a opção de enviar o pacote diretamente para o *tux4* sendo constantemente trocadas mensagens do tipo *redirect* para este efeito. Isto por causa da configuração referida inicialmente do facto de ter sido dado *disable* aos *redirects*.

Seguidamente, após nova adição da rota no *tux2* para a rede 172.16.y0.0/24 via *tux4* verifica-se que já não há passagem dos pacotes pelo *router* pois a rota mais curta é através do *tux4* (consultar figura 14 do anexo II).

Por fim, ativam-se os *redirects*, mas volta-se a retirar a rota no *tux2* para a rede 172.16.y0.0/24 via *tux4*. E o comportamento dos pacotes é o esperado, ou seja, inicialmente acontece um *redirect*, pois o *router* informa o *tux2* de que há uma melhor rota disponível para enviar os pacotes para o *tux3* do que pelo *router* e nos próximos envios de pacotes ICMP, o *tux2* não irá receber novamente essa informação pois já a "memorizou" e sabe que os próximos pacotes podem ser enviados diretamente para o *tux4* (consultar figura 15 do anexo II).

Na configuração atual, não existe acesso à internet pois qualquer *ping*, de um PC que não o *router*, para o *netlab* ou outra máquina qualquer na *internet* não funcionará, uma vez que a máquina de destino não saberá como enviar uma resposta. No entanto, um ping através do *router* funcionaria, porque a máquina de destino sabe como chegar ao *router*. O problema aqui é que os *tux*'s ao enviarem um *ping*, por exemplo, farão com que o endereço de destino da *reply* a esse *ping* seja o próprio endereço do *tux*. Como não há NAT implementado não há tradução de endereços e a comunicação falhará. Com o NAT haverá a associação e transformação de um endereço IP noutro endereço IP, de forma a "mascarar" o remetente/destinatário dos pacotes enviados, podendo ter vários fins como assegurar a privacidade e segurança de máquinas numa subrede privada local que estão comunicar com máquinas "externas" (é implementado com frequência em ambientes de acesso remoto), permite assim que redes IP privadas usem endereços não registados, que se conectem e comuniquem com a Internet ou redes públicas. As máquinas dessa rede, para as máquinas exteriores, são reconhecidas através de um IP único que representa todos os dispositivos da mesma. O NAT opera num *router* e foi configurado da seguinte forma:

Configuração do endereço do *router* para a rede local (*nat inside*) mesmo após o *router* ser desligado:

- *conf t*
- *interface gigabitethernet 0/0*
- *ip address 172.16.41.254 255.255.255.0*
- *no shutdown*
- *ip nat inside*
- *exit*

Configuração do endereço do *router* para a fora da rede local (*nat outside*) mesmo após o *router* ser desligado:

- *interface gigabitethernet 0/1*
- *ip address 172.16.1.49 255.255.255.0*
- *no shutdown*
- *ip nat outside*
- *exit*

Garantir a gama de endereços:

- *ip nat pool ovrld 172.16.1.49 172.16.1.49 prefix 24*
- *ip nat inside source list 1 pool ovrld overload*

Listas representativas dos conjunto de endereços IP das redes 172.16.40.0 e 172.16.41.0 com "acesso" à tradução de endereço IP por parte do *router*:

- *access-list 1 permit 172.16.40.0 0.0.0.7*
- *access-list 1 permit 172.16.41.0 0.0.0.7*

Configuração das rotas (*default* direcionada para o *netlab* (172.16.1.254) e para a rede 172.16.40.0 para o *tux4 - eth1*):

- *ip route 0.0.0.0 0.0.0.0 172.16.1.254*
- *ip route 172.16.40.0 255.255.255.0 172.16.41.253*
- *end*

Após esta configuração estar concluída, quando, por hipótese, o *tux3* quiser enviar um pacote para uma rede pública (172.16.1.254, tal como referido na experiência), o pacote é enviado para o *router* que por sua vez irá modificar o *source* do pacote para o seu endereço de *nat outside* (172.16.1.49), "mascarando" assim o verdadeiro remetente do pacote (*tux3*) e assegurando a sua privacidade e segurança. O pacote é enviado para 172.16.1.254 que responde com um pacote que terá como destino o *router* (172.16.1.49). O *router*, como guarda um estado de ligação com cada *tux* internamente, irá enviar esse pacote para o *tux3*, mudando o destinatário do pacote para o seu endereço, possibilitando então a comunicação entre a rede privada local e a rede pública. Como já foi referido acima, caso o *router* não estivesse configurado com o protocolo NAT, a máquina 172.16.1.254 ao receber um pacote que iria ter como *source* o *tux3* (verdadeiro remetente) não saberia como responder a esse endereço IP pois não haveria qualquer rota previamente estabelecida entre ambos.

Experiência 5 – DNS

Nesta experiência foi necessário configurar o DNS nos *tux's* 2, 3 e 4. Um servidor DNS, no nosso caso, *services.netlab.fe.up.pt*, contém uma base de dados dos endereços IP públicos e dos seus respetivos *host-names*. É usado para traduzir os *hostnames* para os seus respetivos endereços de IP.

O serviço DNS é configurado no ficheiro *resolv.conf*, localizado no diretório */etc/* do tux em questão. A configuração foi feita através de dois comandos, um que representa o nome do servidor DNS, e um outro com o respetivo endereço IP:

- *search netlab.fe.up.pt*
- *nameserver 172.16.1.1*

O *host* envia para o server um pacote com o *hostname*, e espera que seja retornado o seu endereço IP. O servidor responde com um pacote que contém o endereço IP do *hostname* em causa. Estes acontecimentos poderão ser observados na figura 16 do anexo II onde é feita a tradução de endereços do *google.com*.

Experiência 6 – Conexões TCP

O objetivo desta experiência foi a observação do funcionamento e comportamento do protocolo TCP (*Transmission Control Protocol*), sendo para tal usada a aplicação de *download* de ficheiros desenvolvida na primeira parte do projeto.

São abertas duas conexões TCP pela aplicação: uma quando se entra em contacto com o servidor e através da qual se enviam e recebem comandos para preparar a transferência do ficheiro, e outra para fazer a transferência do ficheiro em si. O controlo de informação é transportado na primeira conexão.

Existem três fases numa conexão TCP: primeiramente, estabelece-se a conexão; depois, ocorre a troca de dados; por último, a conexão é encerrada.

O TCP utiliza o mecanismo ARQ (*Automatic Repeat Request* ou *Automatic Repeat Query*) com o método da janela deslizante (*Selective Repeat*), que consiste no controlo de erros na transmissão de dados. Para este efeito, são utilizados *acknowledgement numbers*, que indicam a correto recebimento da trama, *window size*, que indica a gama de pacotes que o emissor pode enviar, e *sequence number*, que é o número do pacote a ser enviado.

O mecanismo de controlo de congestão é usado quando o TCP mantém uma janela de congestão, que consiste numa estimativa do número de octetos que a rede consegue encaminhar, não enviando mais octetos do que o número mínimo da janela definida pelo recetor e pela janela de congestão. No início do primeiro *download* do *tux3*, a taxa de transferência aumentou e atingiu o seu pico aos 14 segundos. Quando iniciamos um segundo *download* através do *tux2*, verificamos uma descida acentuada, resultante da ocupação do canal por parte dos dois *tux*'s. A dada altura o número de pacotes transferidos por segundo estabiliza num nível mais baixo do que quando apenas existia o *tux3* a efetuar o *download*. O gráfico representativo da situação referida demonstrativo deste mecanismo de controlo de congestão está descrito na figura 17 do anexo II.

Assim sendo, o *throughput* de uma conexão de dados TCP é perturbado pelo aparecimento de uma segunda conexão TCP. A taxa de transmissão de pacotes da conexão TCP que já estava iniciada diminui, uma vez que à outra conexão foi atribuída uma capacidade para a transmissão de pacotes na mesma, de modo a que a taxa de transferência seja distribuída de igual forma para cada ligação.

Conclusões

O objetivo do segundo projeto da unidade curricular de Redes e computadores teve como objetivo o desenvolvimento de uma aplicação que permitisse o *download* de um ficheiro, através dos protocolos FTP e TCP. Para além disso, também existiu um processo de configuração de uma rede IP que inclui vários passos e nos permitiu compreender o funcionamento de várias máquinas e dispositivos como o *switch* e o *router*, e também de modo a reconhecer diversas técnicas e protocolos utilizados na comunicação entre esses dispositivos.

Assim, após a conclusão deste projeto, o grupo conseguiu aprofundar os conhecimentos teóricos e práticos nesta área e, visto que todos os objetivos do projeto foram concluídos, consideramos que o projeto foi um sucesso.

Referências

Este projeto foi desenvolvido com recurso à consulta dos slides das aulas teóricas, bem como do guião fornecido.

Anexo I

```
ftp://ftp.up.pt/debian/extrafiles
--------------------------------------------
User: anonymous
Password: 123
Host: ftp.up.pt
Url Path: debian/extrafiles
Filename: extrafiles
Host name  : mirrors.up.pt
IP Address : 193.137.29.15
--------------------------------------------

[220] < Connection Established  [ftp.up.pt:21]
> USER anonymous
[331] < Please specify the password.
> PASS 123
[230] < Login successful.
[227] < Entering Passive Mode (193,137,29,15,203,240).
Host name  : 193.137.29.15
IP Address : 193.137.29.15

--------------------------------------------
< Connection Established  [193.137.29.15:52208]
--------------------------------------------

> RETR debian/extrafiles
[150] < Opening BINARY mode data connection for debian/extrafiles (226155 bytes).
Downloaded 5% of the file.
Downloaded 10% of the file.
Downloaded 15% of the file.
Downloaded 20% of the file.
Downloaded 25% of the file.
Downloaded 30% of the file.
Downloaded 35% of the file.
Downloaded 40% of the file.
Downloaded 45% of the file.
Downloaded 50% of the file.
Downloaded 55% of the file.
Downloaded 60% of the file.
Downloaded 65% of the file.
Downloaded 70% of the file.
Downloaded 75% of the file.
Downloaded 80% of the file.
Downloaded 85% of the file.
Downloaded 90% of the file.
Downloaded 95% of the file.
Downloaded 100% of the file.
[226] < Transfer complete.
> Finished file download!
```

Figura 1

Anexo II

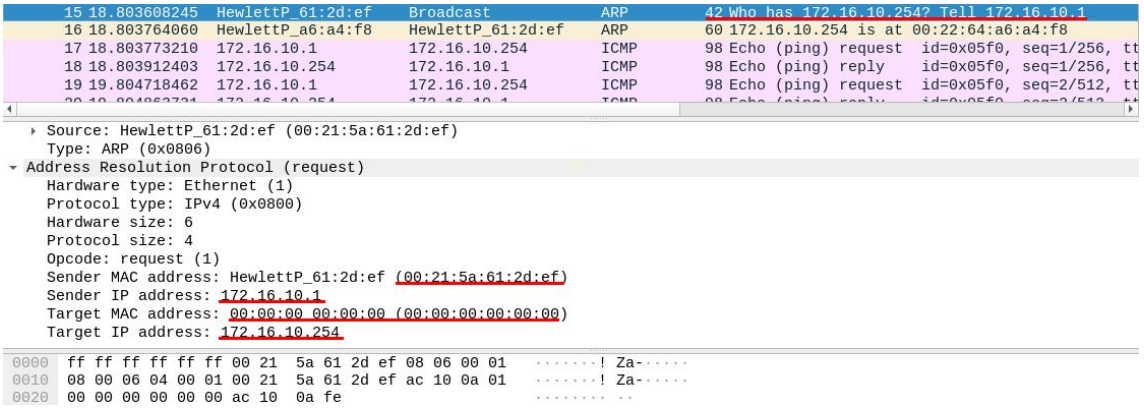

Figura 1

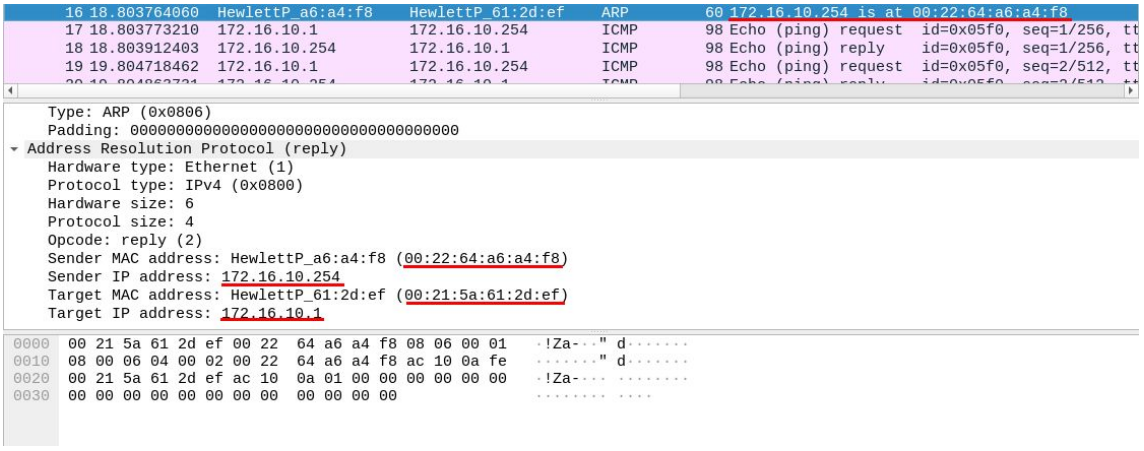

Figura 2

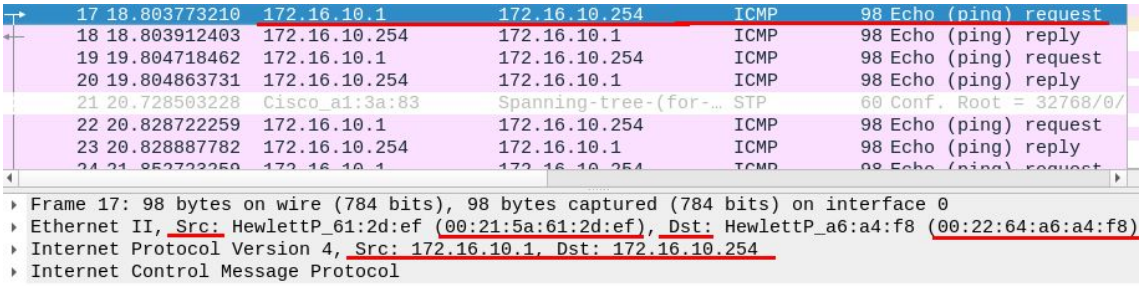

Figura 3

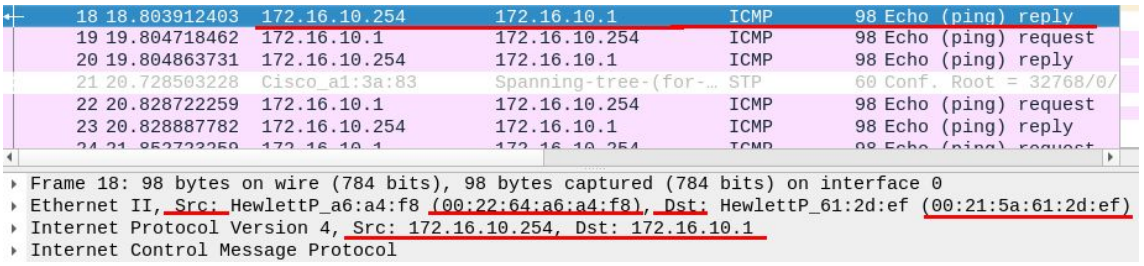

Figura 4

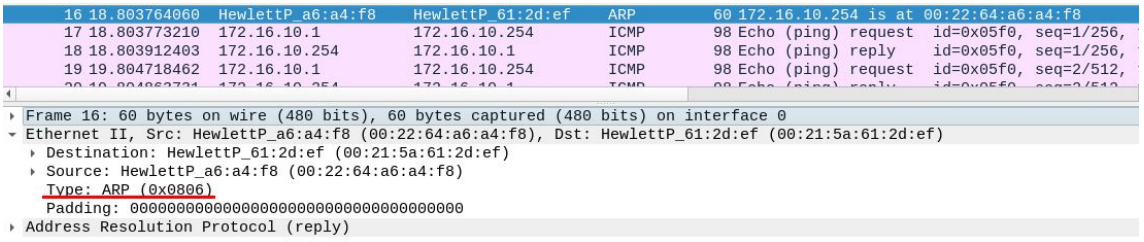

Figura 5

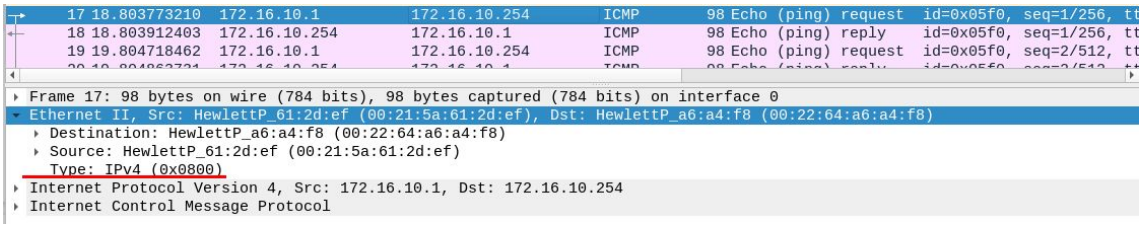

Figura 6

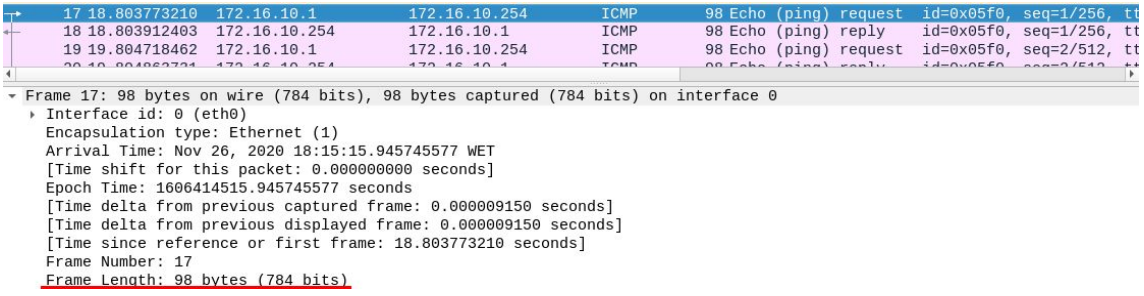

Figura 7

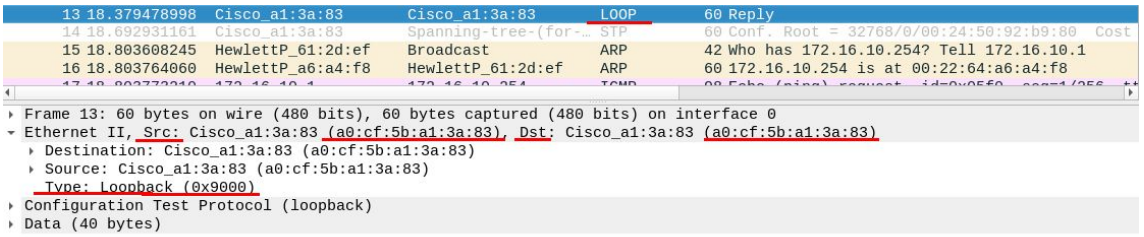

Figura 8

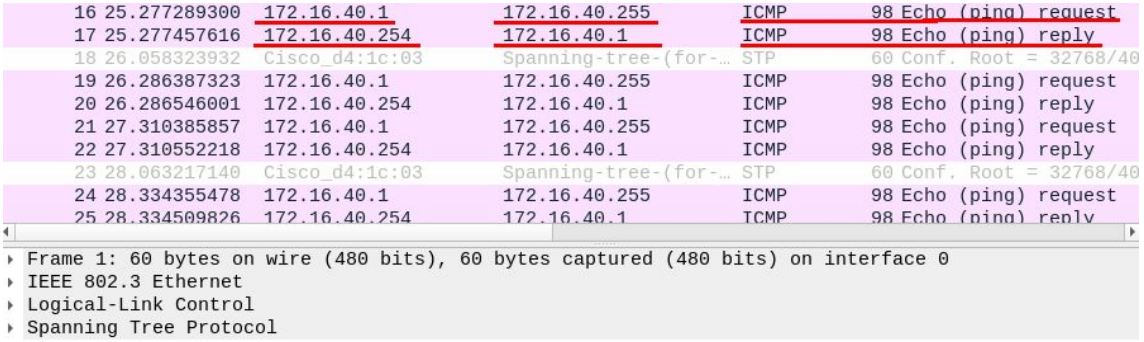

Figura 9

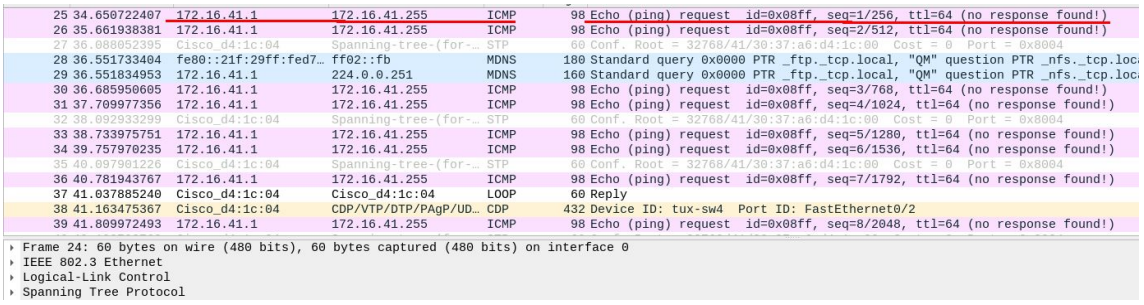

Figura 10

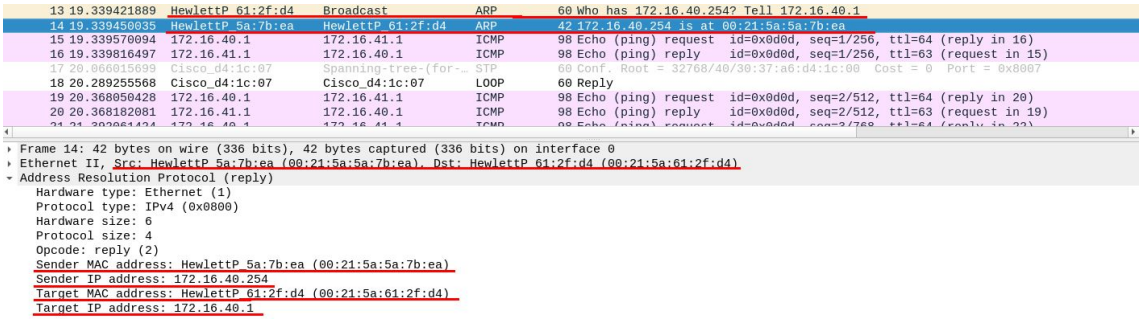

Figura 11

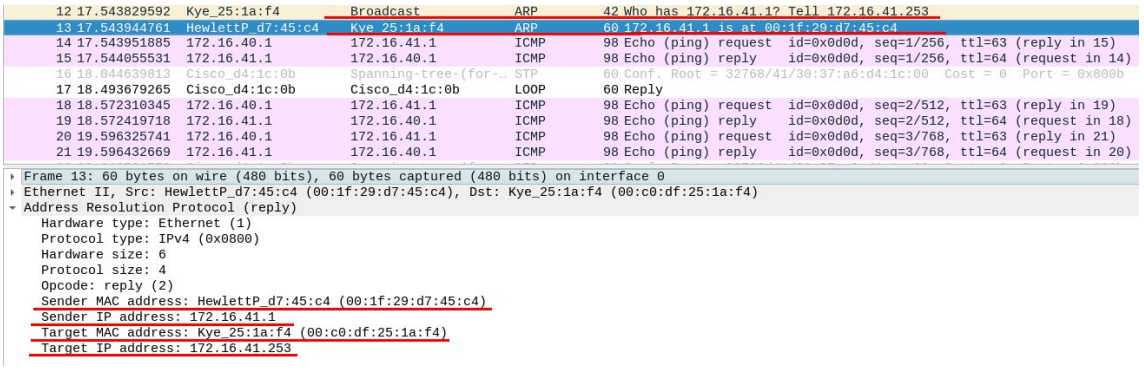

Figura 12

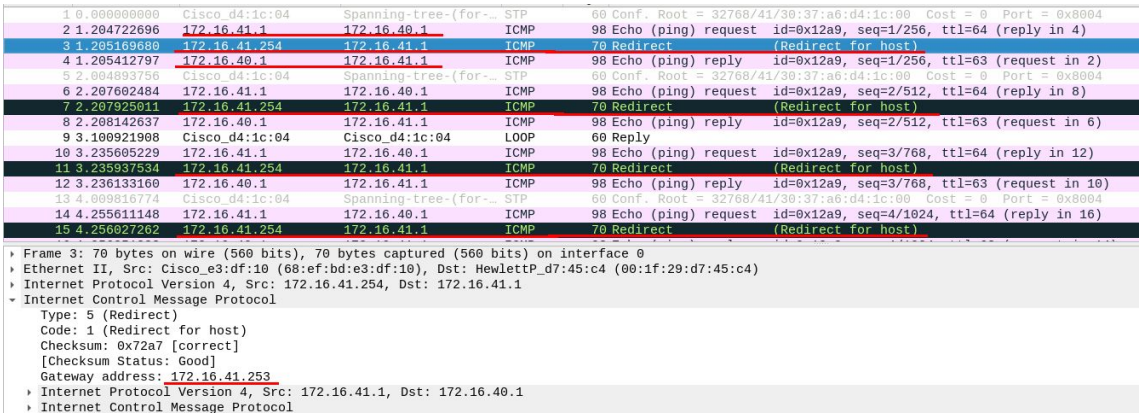

Figura 13

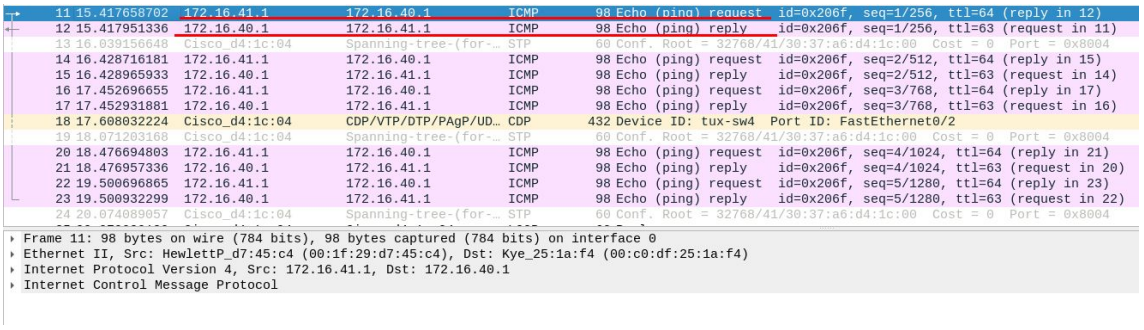

Figura 14

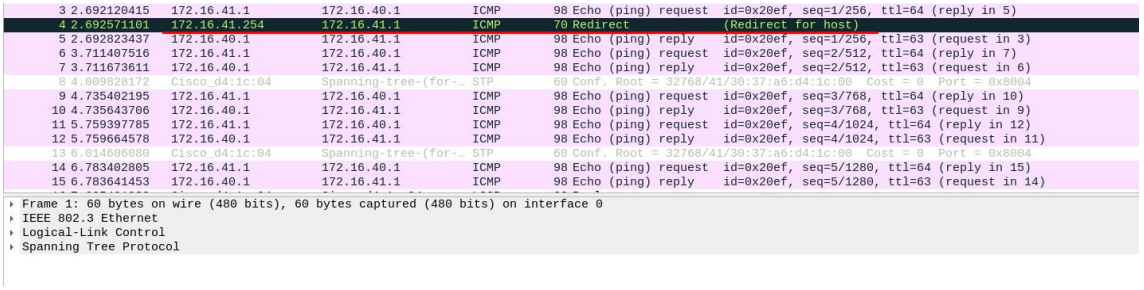

Figura 15

Figura 16

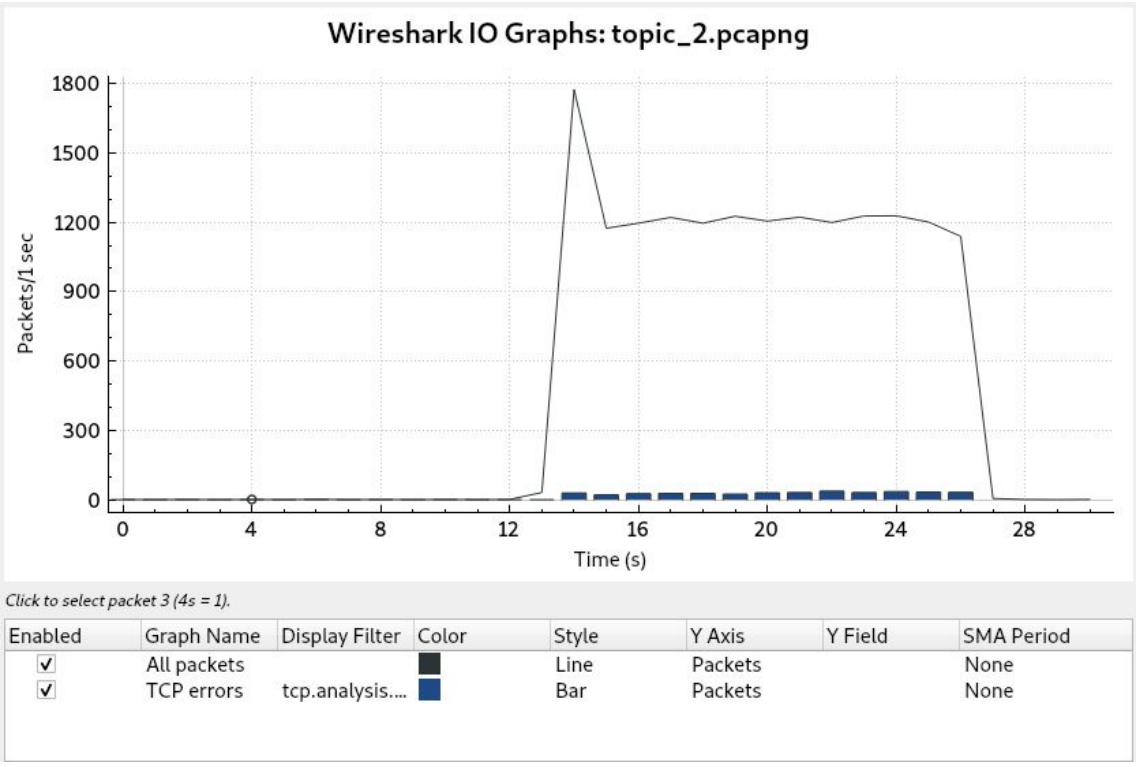

Figura 17

Anexo III

clientTCP.C

```
#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 "macros.h"
#include <string.h>
#include <fcntl.h>
#include <stdbool.h>
#include <ctype.h>

// Error messages handler
void errorMessage(char* error, int statusCode) {
	printf("%s\n", error);
	exit(statusCode);
}

// Parse URL arguments
struct FTPclientArgs parseArguments(char* argv) {
	int atIndex;

	struct FTPclientArgs clientArgs;
```

```
	char* colon = strchr(&argv[4], ':');
	char* at = strchr(argv, '@');
	char* hostSlash = strchr(&argv[URL_HEADER_LEN], '/');
	char* urlPathSlash = strchr(&argv[URL_HEADER_LEN], '/') + 1;

	if(colon != NULL && at != NULL) {
		if(colon >= at)
			errorMessage("Usage: ./client
ftp://[<user>:<password>@]<host>/<url-path>\n", 1);

		// User
		int colonIndex = colon - argv;
		clientArgs.user = (char *) malloc((colonIndex - URL_HEADER_LEN) + 1);
		memcpy(clientArgs.user, &argv[URL_HEADER_LEN], colonIndex -
URL_HEADER_LEN);

		// Password
		atIndex = at - argv;
		clientArgs.password = (char *) malloc((atIndex - (colonIndex + 1)) + 1);
		memcpy(clientArgs.password, &argv[colonIndex + 1], atIndex - (colonIndex +
1));
	}
	else {
		clientArgs.user = (char *) malloc(11);
		strcpy(clientArgs.user, DEFAULT_USER);
		clientArgs.password = (char *) malloc(4);
		strcpy(clientArgs.password, DEFAULT_PASSWORD);
		atIndex = 5;
	}

	char header[7];
	strncpy(header, argv, URL_HEADER_LEN);
	if(hostSlash == NULL || urlPathSlash == NULL || strcmp(header, "ftp://") != 0)
		errorMessage("Usage: ./client
ftp://[<user>:<password>@]<host>/<url-path>\n", 1);

	// Host
	int slashIndex = hostSlash - argv;
	clientArgs.host = (char *) malloc((slashIndex - (atIndex + 1)) + 1);
	memcpy(clientArgs.host, &argv[atIndex + 1], slashIndex - (atIndex + 1));

	// url-path
	int pathIndex = urlPathSlash - argv;
	int urlPathLen = strlen(argv) - pathIndex ;
	clientArgs.urlPath = (char *) malloc(urlPathLen + 1);
	memcpy(clientArgs.urlPath, &argv[slashIndex + 1], urlPathLen);

	// filename
	char* filenameSlash = strrchr(argv, '/');
	int filenameSlashIndex = (filenameSlash - argv);
	clientArgs.filename = (char *) malloc(strlen(argv) - filenameSlashIndex);
	memcpy(clientArgs.filename, &argv[filenameSlashIndex + 1], strlen(argv) -
filenameSlashIndex - 1);

	if(strcmp(clientArgs.user, "") == 0 || strcmp(clientArgs.password, "") == 0 ||
		strcmp(clientArgs.host, "") == 0 || strcmp(clientArgs.urlPath, "") == 0 ||
```

```
                strcmp(clientArgs.filename, "") == 0)
                errorMessage("Usage: ./client
ftp://[<user>:<password>@]<host>/<url-path>\n", 1);

        return clientArgs;
}

// Gets the IP of a specific hostname
struct hostent* getIP(char* hostName)
{
        struct hostent* h = (struct hostent *) malloc(sizeof(struct hostent));

        /*
        struct hostent {
                char    *h_name;Official name of the host.
                char    **h_aliases;        A NULL-terminated array of alternate names for the
host.
                int     h_addrtype;         The type of address being returned; usually AF_INET.
                int     h_length;   The length of the address in bytes.
                char    **h_addr_list;      A zero-terminated array of network addresses for the
host.
                                            Host addresses are in Network Byte Order.
        };
                #define h_addr h_addr_list[0]      The first address in h_addr_list.
        */
        if ((h = gethostbyname(hostName)) == NULL) {
                herror("gethostbyname");
                exit(1);
        }

        printf("Host name  : %s\n", h->h_name);
        printf("IP Address : %s\n",inet_ntoa(*((struct in_addr *)h->h_addr)));

        return h;
}

// State Machine handler that parses the server response
enum readState getState(char character, enum readState previousState) {
        if(isdigit(character) && (previousState == LineChange || previousState == StatusCode))
                return StatusCode;
        else if(character == '\n') {
                if(previousState == Space || previousState == MainMsgText)
                        return EndMessage;
                return LineChange;
        }
        else if(character == ' ') {
                if(previousState == StatusCode)
                        return Space;
                if(previousState == MainMsgText)
                        return MainMsgText;
                return DummyMsgText;
        }
        else if(character == '-')
                if(previousState == MainMsgText)
                        return previousState;
                else return Dash;
```

```
	else {
		if(previousState == Space || previousState == MainMsgText)
			return MainMsgText;
		return DummyMsgText;
	}
}

//https://stackoverflow.com/questions/29152359/how-to-read-multiline-response-from-ftp-server
// Reads a response and determines what to do according to the state of each char
char* readResponse(int sockfd, char* statusCode) {
	enum readState rStatus = LineChange;
	char c;
	int statusCodeIndex = 0, msgIndex = 0;
	char* message = (char *) malloc(0);

	while(rStatus != EndMessage) {
		read(sockfd, &c, 1);
		rStatus = getState(c, rStatus);

		switch (rStatus)
		{
			case StatusCode:
				statusCode[statusCodeIndex++] = c;
				break;
			case LineChange:
				statusCodeIndex = 0;
				break;
			case MainMsgText:
				message = (char* ) realloc(message, msgIndex + 1);
				message[msgIndex++] = c;
			default:
				break;
		}
	}

	message[msgIndex] = '\0';
	statusCode[3] = '\0';

	if(strcmp(message, "\r") != 0) {
		printf("[%s] < %s\n", statusCode, message);
		fflush(stdout);
	}

	return message;
}

// Send command
void sendCmd(int sockfd, char mainCMD[], char* contentCMD) {
	write(sockfd, mainCMD, strlen(mainCMD));
	write(sockfd, contentCMD, strlen(contentCMD));
	write(sockfd, "\n", 1);
	printf("> %s%s\n", mainCMD, contentCMD);
}

// Loads all the info transferred to a local file
```

```
void readServerData(int dataSockfd, char* filename, int fileSize) {
	FILE* file = fopen(filename, "wb+");

	int numCharsRead = 0;
	char c;
	int percentageFileLoaded = 0;
	int nextPoint = 5;
	while(read(dataSockfd, &c, 1) > 0) {
		percentageFileLoaded = (float) ++numCharsRead / fileSize * 100;
		if(percentageFileLoaded == nextPoint) {
			nextPoint += 5;
			printf("Downloaded %d%% of the file.\n", percentageFileLoaded);
		}

		fwrite(&c, 1, 1, file);
	}

	fclose(file);
}

// Gets the size of the transferred file
int parseFileSize(char* response) {
	char* auxString = strrchr(response, '(');
	int fileSize;
	sscanf(auxString, "(%d bytes).", &fileSize);
	return fileSize;
}

// https://en.wikipedia.org/wiki/List_of_FTP_server_return_codes
// Sends a command and waits for a response
int sendCommandAndFetchResponse(int sockfd, char mainCMD[], char* contentCMD, char*
filename, int dataSockfd) {
	sendCmd(sockfd, mainCMD, contentCMD);

	while(true) {
		char statusCode[4];
		char* response = readResponse(sockfd, statusCode);
		switch (statusCode[0])
		{
			// Expect Another Reply
			case '1': {
				if(strcmp(mainCMD, "RETR ") == 0) {
					int fileSize = parseFileSize(response);
					readServerData(dataSockfd, filename, fileSize);
				}


				break;
			}
			// Request Completed.
			case '2':
				return 2;

			// Needs further information (Login Case)
			case '3':
				return 3;
```

```
                        // Command not accepted, but we can send it again.
                        case '4':
                                sendCmd(sockfd, mainCMD, contentCMD);
                                break;

                        // Command had errors!
                        case '5': {
                                int errorStatus;
                                sscanf(statusCode, "%d", &errorStatus);
                                exit(errorStatus);
                        }
                        default:
                                break;
                }
                free(response);
        }

}

// Parse Passive Mode String in order to fetch Port and IP Address.
char* parsePassiveModeArgs(char* response, int* port) {
        // (193,137,29,15,202,40) -> 193.137.29.15 | 202 * 256 + 40
        int nums[6];
        sscanf(response, "Entering Passive Mode
(%d,%d,%d,%d,%d,%d).",&nums[0],&nums[1],&nums[2],&nums[3],&nums[4],&nums[5]);

        char* ip_address = (char *) malloc(0);
        int ip_index = 0;
        char aux[5];
        int len;

        for (int i = 0; i < 4; i++) {
                sprintf(aux, "%d", nums[i]);
                len = strlen(aux);

                ip_address = (char* ) realloc(ip_address, ip_index + len + 1);

                memcpy(&ip_address[ip_index], aux, len);

                ip_address[ip_index + len] = '.';
                ip_index += len + 1;
        }

        *port = nums[4] * 256 + nums[5];
        ip_address[ip_index-1] = '\0';

        return ip_address;
}

// Opens a connection
int openSocketAndConnect(struct hostent* hostInfo, uint16_t serverPort) {
        int     sockfd;
        struct sockaddr_in server_addr;
        int     bytes;
```

```
	/*server address handling*/
	/*
		struct sockaddr_in {
			short int sin_family; // Address family, AF_INET
			unsigned short int sin_port; // Port number
			struct in_addr sin_addr; // Internet address
			unsigned char sin_zero[8]; // Same size as struct sockaddr
		};
		struct in_addr {
			uint32_t s_addr; // that's a 32-bit int (4 bytes)
		};
		struct sockaddr {
			u_short sa_family; // Address family - ex: AF_INET
			char sa_data[14]; // Protocol address - sa_data contains a destination
address and port number for the socket
		};
	*/

	bzero((char*)&server_addr,sizeof(server_addr));
	server_addr.sin_family = AF_INET;
	server_addr.sin_addr = *((struct in_addr *)hostInfo->h_addr);	/*32 bit Internet
address network byte ordered*/
	server_addr.sin_port = htons(serverPort);		/*server TCP port must be network
byte ordered */

	/*open an TCP socket*/
	if ((sockfd = socket(AF_INET,SOCK_STREAM,0)) < 0)
		errorMessage("socket() failed.", 2);

	/*connect to the server*/
   if(connect(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0)
		errorMessage("connect() failed.", 3);

	return sockfd;
}

int main(int argc, char** argv) {
	// ftp://[<user>:<password>@]<host>/<url-path>
	// ftp://ftp.up.pt/pub/README.html
	// ftp://up201806441:123@ftp.up.pt/pub/README.html

	struct FTPclientArgs clientArgs = parseArguments(argv[1]);

	printf("\n\n%s\n", argv[1]);
	printf("-----------------------------------------------\n");
	printf("User: %s\n", clientArgs.user);
	printf("Password: %s\n", clientArgs.password);
	printf("Host: %s\n", clientArgs.host);
	printf("Url Path: %s\n", clientArgs.urlPath);
	printf("Filename: %s\n", clientArgs.filename);

	struct hostent* hostInfo = getIP(clientArgs.host);
	printf("-----------------------------------------------\n\n");
	int sockfd = openSocketAndConnect(hostInfo, SERVER_PORT);
	int dataSockfd = -1;
```

```
	/* Read Server Welcoming message */
	char statusCode[4];
	char* response = readResponse(sockfd, statusCode);

	if(statusCode[0] != '2') {
		char errorMsg[50];
		sprintf(errorMsg, "Status [%s] : %s\n", statusCode, response);
		errorMessage(errorMsg, 4);
	} else
		printf("[%s] < Connection Established  [%s:%d]\n", statusCode, clientArgs.host,
SERVER_PORT);

	free(response);
	int ret = sendCommandAndFetchResponse(sockfd, "USER ", clientArgs.user,
clientArgs.filename, dataSockfd);
	if(ret == 3)
		sendCommandAndFetchResponse(sockfd, "PASS ", clientArgs.password,
clientArgs.filename, dataSockfd);

	write(sockfd, "PASV\n", 5);
	response = readResponse(sockfd, statusCode);

	int* port = malloc(sizeof(int));
	char* ipAdr = parsePassiveModeArgs(response, port);

	struct hostent* dataHostInfo = getIP(ipAdr);
	dataSockfd = openSocketAndConnect(dataHostInfo, *port);
	printf("\n-----------------------------------------------\n");
	printf("< Connection Established  [%s:%d]\n", ipAdr, *port);
	printf("-----------------------------------------------\n\n");

	ret = sendCommandAndFetchResponse(sockfd, "RETR ", clientArgs.urlPath,
clientArgs.filename, dataSockfd);
	if(ret == 2)
		printf("> Finished file download!\n");
	else
		printf("> Error downloading file!\n");

	free(ipAdr);
	free(port);
	free(clientArgs.user);
	free(clientArgs.password);
	free(clientArgs.host);
	free(clientArgs.urlPath);
	close(sockfd);
	close(dataSockfd);
	exit(0);
}
```

macros.h

```
#define SERVER_PORT 21
```

```
#define SERVER_ADDR "193.137.29.15"

#define DEFAULT_USER "anonymous"
#define DEFAULT_PASSWORD "123"
#define URL_HEADER_LEN 6

struct FTPclientArgs {
   char* user;
   char* password;
   char* host;
   char* urlPath;
   char* filename;
};

enum readState { StatusCode, Space, Dash, LineChange, DummyMsgText, MainMsgText,
EndMessage};
```