U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Relatório

2º Trabalho Laboratorial

Redes de Computadores

T06_G08

André Pereira – up201905650

Matilde Oliveira – up201906954

11/12/2021

Contents

Sumário 3

Introdução 3

Parte 1 - Aplicação *download* 3

Arquitetura 4

Principais Estruturas 4

Principais Funções e Percurso do Programa 4

Resultados 5

Parte 2 - Configuração e análise da Rede 5

Experiência 1 – Configuração de uma rede IP 5

Arquitetura da rede 5

Objetivos da experiência 5

Principais comandos de configuração 5

Análise 6

Experiência 2 – Configuração de duas virtual LANs num *switch* 8

Arquitetura da rede 8

Objetivos da experiência 8

Principais comandos de configuração 8

Análise 8

Experiência 3 – Configuração de um router 9

Arquitetura da rede 9

Objetivos da experiência 9

Principais comandos de configuração 9

Análise 9

Experiência 4 – Configuração de um router Linux e de um router CISCO 12

Configuração router Linux 12

Arquitetura da rede 12

Objetivos da experiência 12

Principais comandos de configuração 12

Análise 12

Configuração router Cisco 15

Arquitetura da rede 15

Objetivos da experiência 15

Principais comandos de configuração 15

Análise 16

Conclusão 16

Sumário

O relatório descreve o trabalho desenvolvido para o segundo trabalho laboratorial da cadeira de Redes de Computadores do terceiro ano do curso L.EIC. Tendo sido desenvolvido em conjunto pelos alunos André Pereira e Matilde Oliveira, tendo ambos trabalhado de igual forma para o sucesso do trabalho.

Introdução

O presente trabalho foi proposto no contexto da unidade curricular de Redes de Computadores e o seu desenvolvimento foi dividido em essencialmente duas partes explicadas nas próximas secções.

A primeira parte incidiu sobre o desenvolvimento de uma aplicação de download de ficheiros por FTP, enquanto a segunda foi constituída por uma séria de experiências que concluíram várias etapas de uma configuração de uma rede.

A primeira parte foi desenvolvida em contexto fora de sala de aula e implementada em linguagem C, contemplando o percurso normal do uso de um protocolo FTP.

Quanto à segunda parta, as experiências foram desta vez realizadas com recurso aos materiais dos laboratórios, tendo sido todas realizadas com sucesso e os seus objetivos sido alcançados como é mostrado na parte 2 deste relatório, onde são explicadas principais questões relativas às experiências.

Parte 1 - Aplicação *download*

A primeira parte deste segundo trabalho laboratorial, consistiu no desenvolvimento de uma **aplicação de transferência de ficheiros** de acordo com o protocolo **FTP** (*File Transfer Protocol*) descrito no RFC959. Este foi desenvolvido através de ligações *Transmission Control Protocol* (**TCP**) a partir de sockets, tornando a aplicação capaz de transferir quais quer tipos de ficheiros de um indicado servidor FTP.

A aplicação pode ser executada após compilada da seguinte forma:

./download **ftp**://[<**user**>:<**password**>@]<**host**>/<**path**>

O segundo argumento para execução segue uma configuração descrita para um *url* no RFC1728.

Para teste da aplicação o servidor FTP utilizado foi ftp.fe.up.pt, especificando por exemplo o user e password ambos como "rcom" seguido do host, netlab1.fe.up.pt, e o caminho para os ficheiros a testar.

Para um melhor entendimento do trabalho a desenvolver, foi importante começar por conhecer melhor o protocolo de transferência de ficheiros, descrito no RFC959, bem como o documento RFC1738 ("Uniform Resource Locators URL") para compreender como deveria ser tratada a informação proveniente dos URL's.

Arquitetura

Principais Estruturas

A principal e única estrutura de dados, é a *struct Args*, definida para guardar informação relativa aos valores resultantes do parsing do *url,* estando definida no ficheiro *utils.h*.

```
typedef struct {
    char user[DATA_SIZE];
    char password[DATA_SIZE];
    char host[DATA_SIZE];
    char path[DATA_SIZE];
    char ip[DATA_SIZE];
    char filename[DATA_SIZE];
    int port;
} Args;
```

Principais Funções e Percurso do Programa

Em primeiro lugar é necessário preencher a *struct Args* explicitada anteriormente com os dados provenientes do *url*, definido no segundo argumento da chamada do programa (*user*, *password*, *host*, *path*, *filename*). Este *parsing* é completado na função ***parseArguments***. Note-se que no caso dos argumentos user e password não serem facultados o programa assume os respetivamente, "anonymous" e "123", podendo prosseguir com a execução do programa.

De seguida, a função ***getIp***, segue os moldes da função equivalente fornecida sendo que recebendo o nome do *host* retorna o *ip*, preenchendo esse campo da *struct Args*, bem como a porta utilizada com 21.

A criação e conexão do socket é feita na função ***connectSocket***, de igual modo ao código fornecido.

Após estas configurações é possível começar com o envio de comandos e a receção de respostas, sendo que para ambos são sempre utilizadas as funções respetivas, ***writeSocket*** e ***readSocket***. Especificamente, quanto à receção de resposta, esta é auxiliada pela função ***readText*** que lê do *socket* e verifica se o código de três dígitos da resposta é o esperado mediante o comando executado, na função ***checkCode***.

Depois da conexão do socket é agora necessário fazer o login onde serão construídos e enviados os comandos "***user <user>***" e "***pass <password>***", sendo os códigos válidos de receção respetivamente 331 e 230.

Em seguida, é feita a entrada em modo passivo, com o envio do comando "***pasv***" que retorna informação relativa à porta para abrir o *socket* que vai servir para receber o ficheiro, esta porta é calculada na função ***parsePassivePort*** após a receção da mensagem do *socket* relativa ao comando *pasv* com o código 227.

A informação sobre a porta do socket que irá receber o ficheiro permite conectar novo socket, novamente com recurso a ***connectSocket***.

Finalmente, para o envio do ficheiro envia-se o comando "***retr <path>***" do primeiro socket aberto e no socket onde será realizado o download (o último a ser conectado) inicia-se o download do ficheiro, função ***downloadFile***, recorrendo às mesmas funções de ler do socket.

O programa, no primeiro socket, acaba por indicar a finalização da transferência com o código 226 e envia o comando de "***quit***" terminando a conexão na função ***closeSocket***.

Resultados

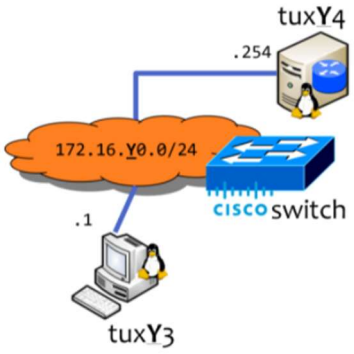
A aplicação teve sempre resultados dentro do esperado e foi sempre bem-sucedida, independentemente do tamanho e tipo do ficheiro escolhido, bem como da introdução ou não de *user* e *password*. Em baixo deixamos uma simulação da execução:

```
adbp@DESKTOP-N0C4DSC:/mnt/c/Users/adbp/Documents/FEUP/3ano/1semestre/RCOM/feup-rc-2021/proj2/src$ make
gcc -c utils.c -o utils.o
gcc download.c utils.o -o download
adbp@DESKTOP-N0C4DSC:/mnt/c/Users/adbp/Documents/FEUP/3ano/1semestre/RCOM/feup-rc-2021/proj2/src$ ./download ftp://ftp.up.pt/pub/kodi/timestamp.txt
No credentials given
Host: ftp.up.pt
Path: pub/kodi/timestamp.txt
FileName: timestamp.txt
user: anonymous
password: 123
ip: 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 123
230 Login successful.
pasv
227 Entering Passive Mode (193,137,29,15,197,106).
retr pub/kodi/timestamp.txt
150 Opening BINARY mode data connection for pub/kodi/timestamp.txt (11 bytes).
226 Transfer complete.
quit
221 Goodbye.
```

Parte 2 - Configuração e análise da Rede

Experiência 1 – Configuração de uma rede IP

Arquitetura da rede



Objetivos da experiência

Nesta primeira experiência o objetivo é configurar e estabelecer uma ligação entre os *tux23* e *tux24*, para tal será necessário ligar ambos os computadores a um *switch* e configurar os seus endereços *ip*.

Principais comandos de configuração

Tux23:

ifconfig eth0 down
ifconfig eth0 up 172.16.20.1

Tux24:

ifconfig eth0 down
ifconfig eth0 up 172.16.20.254

Análise

Tux23: **ip** = 172.16.20.1 **MAC** = 00:21:5A:5A:78:C7
Tux24: **ip** = 172.16.20.254 **MAC** = 00:22:64:A7:26:A2

ARP significa "*Address Resolution Protocol*", pelo que o objetivo deste protocolo é mapear um endereço IP ao respetivo endereço *MAC*. Na imagem seguinte é possível visualizar a ocorrência de um pedido *ARP*.

```
24 21.033424336  HewlettP_5a:78:c7   HewlettP_a7:26:a2   ARP      42 Who has 172.16.20.254? Tell 172.16.20.1
25 21.033545722  HewlettP_a7:26:a2   HewlettP_5a:78:c7   ARP      60 172.16.20.254 is at 00:22:64:a7:26:a2
```

Este protocolo é usado quando um computador quer enviar um pacote a uma outra máquina na mesma rede local e não tem esse *IP* na sua tabela *ARP*. De forma a descobrir o destinatário do pacote de mensagem o computador irá primeiramente enviar um pacote *ARP* em *broadcast*, para todas as outras máquinas na mesma rede, a perguntar qual é o endereço *MAC* associado ao IP a que se pretende contactar. O destinatário ao ver o pedido vai enviar um pacote *ARP* como resposta com o seu endereço *MAC*. Assim as duas máquinas podem começar a efetuar a troca de pacotes.

Na imagem seguinte é possível ver com mais detalhe o conteúdo de um pacote ARP.

```
24 21.033424336  HewlettP_5a:78:c7   HewlettP_a7:26:a2   ARP      42 Who has 172.16.20.254? Tell 172.16.20.1
25 21.033545722  HewlettP_a7:26:a2   HewlettP_5a:78:c7   ARP      60 172.16.20.254 is at 00:22:64:a7:26:a2
26 21.065457671  172.16.20.1         172.16.20.254       ICMP     98 Echo (ping) request  id=0x1167, seq=6/1536, ttl=64 (reply in 27)
27 21.065583108  172.16.20.254       172.16.20.1         ICMP     98 Echo (ping) reply    id=0x1167, seq=6/1536, ttl=64 (request in 26)
28 21.116326499  HewlettP_a7:26:a2   HewlettP_5a:78:c7   ARP      60 Who has 172.16.20.1? Tell 172.16.20.254
29 21.116333413  HewlettP_5a:78:c7   HewlettP_a7:26:a2   ARP      42 172.16.20.1 is at 00:21:5a:5a:78:c7
30 22.053750323  Cisco_7c:9c:86      Spanning-tree-(for-… STP     60 Conf. Root = 32768/1/00:1e:14:7c:9c:80  Cost = 0  Port = 0x8006
```

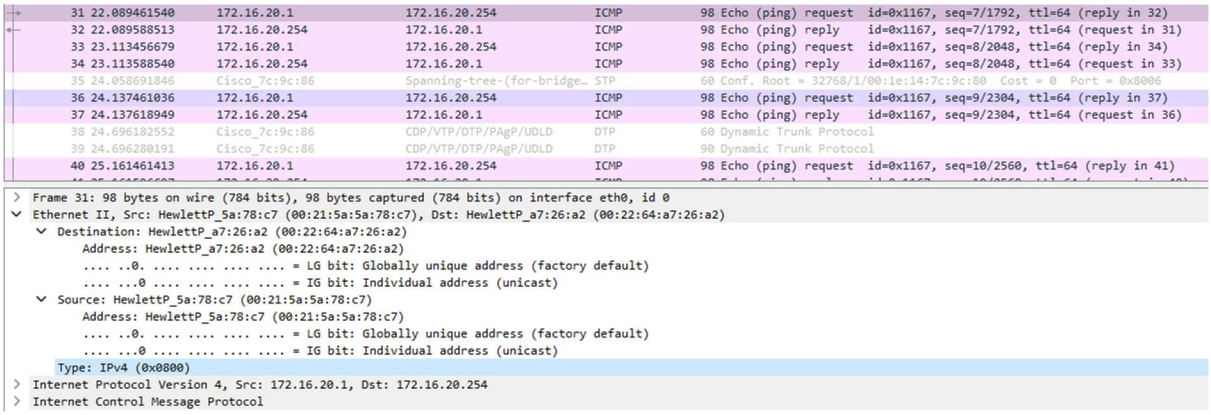
```
   .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
   .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
Type: ARP (0x0806)
dress Resolution Protocol (request)
Hardware type: Ethernet (1)
Protocol type: IPv4 (0x0800)
Hardware size: 6
Protocol size: 4
Opcode: request (1)
Sender MAC address: HewlettP_5a:78:c7 (00:21:5a:5a:78:c7)
Sender IP address: 172.16.20.1
Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
Target IP address: 172.16.20.254
```

No **primeiro pacote *ARP***, o endereço *IP* do **emissor** é **172.16.20.1** e o seu endereço *MAC* é 00:21:5A:5A:78:C7, ou seja o *tux23* quer descobrir qual é a máquina que tem um dado endereço *IP*. O endereço *IP* do **destinatário** é **172.16.20.254**, o endereço da máquina que o tux23 quer descobrir. O endereço *MAC* do destinatário é 00:00:00:00:00:00, o valor de um **pedido *broadcast***, pois o *tux23* não conhece o destinatário.

No segundo pacote *ARP*, **pacote de resposta**, o endereço *IP* do **emissor** é **172.16.20.254** e o endereço *MAC* é 00:22:64:A7:26:A2, ou seja o *tux24* pretende dizer qual é o seu endereço *MAC* a quem lhe perguntou. O *IP* do **destinatário** é **172.16.20.1** e o endereço *MAC* é 00:21:5A:5A:78:C7, como no pacote *ARP* de pergunta vinha indicado o endereço *MAC* do emissor o *tux24* irá enviar uma mensagem direta para o *tux23*, reduzindo assim a sobrecarga da rede local.

No comando *ping*, são primeiramente gerados pacotes *ARP* caso o destinatário não seja conhecido pelo computador. De seguida são gerados 2 pacotes *ICMP "Internet Control*

Message Protocol ", um de pedido e um de resposta, que transportam informação relativa ao comando *ping*. Como podemos verificar na seguinte imagem:



Como foi referido em cima são gerados 2 pacotes *ICMP* um de pedido e um de resposta, que no caso de um *ping* do tux23 para o tux24, os endereços serão deste modo:

Pacote de pedido:

IP do destino: 172.10.20.254 (tux24) *MAC* do destino: 00:22:64:A7:26:A2 (tux24)
IP de origem: 172.10.20.1 (tux23) *MAC* de origem: 00:21:5A:5A:78:C7 (tux23)

Pacote de resposta:

IP do destino: 172.10.20.1 (tux23) *MAC* do destino: 00:21:5A:5A:78:C7 (tux23)
IP de origem: 172.10.20.254 (tux24) *MAC* de origem: 00:22:64:A7:26:A2 (tux24)

É de notar que nenhum dos pacotes é enviado em *broadcast*, pois ambas as máquinas sabem os endereços *MAC* uma da outra.

De maneira a descobrirmos qual é o tipo do pacote temos de olhar para o cabeçalho da trama *Ethernet* recebida.

- Para ser uma trama do tipo ***ARP*** o 13 e 14 *bytes* da trama *Ethernet* têm de ser 0x08 e 0x06, respetivamente.
- Para ser uma trama do tipo ***IP*** o 13 e 14 *bytes* da trama *Ethernet* têm de ser 0x08 e 0x00, respetivamente.
- Para ser uma trama ***ICMP*** a trama terá de ser primeiramente uma trama do tipo *IP* de seguida devemos ir analisar o cabeçalho *IP* e caso o byte associado ao tipo de protocolo seja 0x01 podemos concluir que se trata de uma trama *ICMP*.

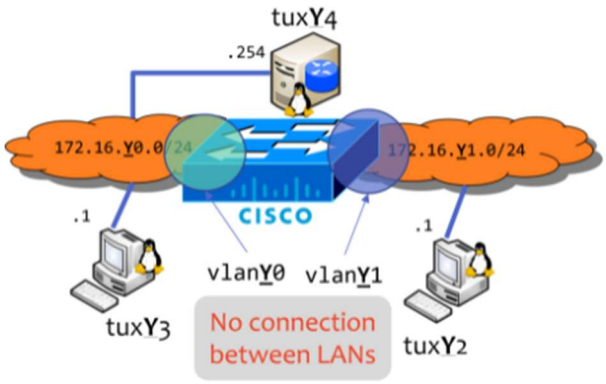
A trama *Ethernet* não apresenta diretamente no seu cabeçalho o tamanho de todo o pacote.

- Para se verificar o tamanho de um pacote, caso este seja do tipo *IP* é necessário ler o valor do 3º e 4º *bytes* do cabeçalho *IP* do pacote, pois este indica o tamanho do cabeçalho *IP* + o tamanho do cabeçalho ICMP + tamanho da *payload*. Após a verificação desse valor, é possível chegar ao tamanho total da trama com a adição de 14 bytes, tamanho constante do cabeçalho da trama *Ethernet*.
- No caso de um pacote *Ethernet* do tipo *ARP* este apresenta um tamanho constante de 42 bytes.

A interface *loopback* tem associada a gama de endereços *IP* 127.0.0.0/8 e até mesmo o nome *localhost*. Esta interface permite ao computador receber respostas de si próprio, o que permite realizar testes de software e conectividade do próprio computador certificando-nos assim da correta configuração da placa de rede.

Experiência 2 – Configuração de duas virtual LANs num *switch*

Arquitetura da rede



Objetivos da experiência

Nesta experiência o objetivo é criar duas *VLANs* no switch, *VLAN* 20 para os tux23 e tux24 e *VLAN* 21 para o tux22. Com esta experiência iremos entender como podemos configurar *VLANs*, bem como estas podem influenciar a nossa rede local.

Principais comandos de configuração

Tux22:

ifconfig eth0 down
ifconfig eth0 up 172.16.21.1

Switch (GTKTerm):

Criar *VLANs*: *configure terminal*
vlan 20
vlan 21
end

Adicionar porta X do switch a *VLAN* 20: *configure terminal*
interface fastethernet 0/X
switchport mode access
switchport access vlan 20
end

Mostrar *VLANs*: *show vlan*

Análise

De forma a configurar uma *vlan* no *switch* será primeiramente necessário aceder a ele, no caso do nosso laboratório é necessário conectar a porta de série de um dos *tuxs* a interface

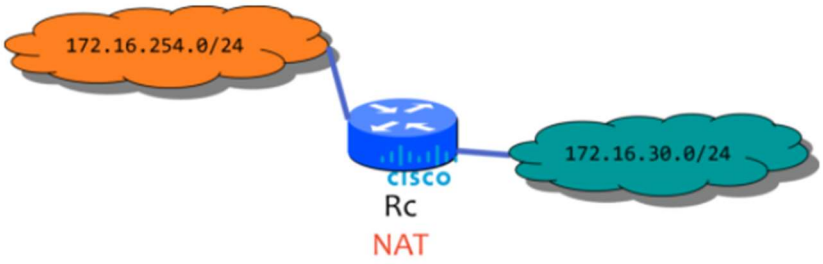
do *switch*. Após realizar essa conexão basta abrir o *GTKTerm* do *tux* que foi ligado e iniciar os comandos de configuração de *vlan*, listados acima.

Na nossa rede local existem pelo menos 2 domínios pois foram criadas duas *VLANs* novas. Esta resposta é verificável nos *logs* do *wireshark*, pois ao fazer um *ping* em *broadcast* no *tux23* podemos ver que apenas os *tuxs* da *vlan* 20 (*tux23* e *tux24*) recebem este pedido. Adicionalmente ao realizar um ping em *broadcast* no *tux22* apenas o próprio, que está na *vlan* 21, dá sinal da sua receção.

Concluímos que apesar dos computadores estarem todos a usar o mesmo switch para mapear os pedidos, alguns não conseguem estabelecer uma ligação entre si, pois não existe nenhum computador que faça a ligação entre a vlan20 e vlan21.

Experiência 3 – Configuração de um router

Arquitetura da rede



Objetivos da experiência

A experiência 3, tem como objetivo a configuração de um router, numa máquina local, visto que esta foi realizada de modo remote. Passará por analisar um ficheiro de configuração de um Cisco Router, bem como realizar testes de entradas de DNS.

Principais comandos de configuração

Configuração do *Cisco Router*

```
interface FastEthernet0/0
    ip address 172.16.254.0 255.255.255.0
    ip nat inside
```

```
interface FastEthernet0/1
    ip address 172.16.30.0 255.255.255.0
    ip nat outside
```

```
ip route 0.0.0.0 0.0.0.0 172.16.30.0
ip route <ip interno> 255.255.255.0 172.16.254.0
```

```
ip nat pool ovrld 172.16.30.0 172.16.30.0 prefix-length 24
ip nat inside source list 1 pool ovrld overload
access-list 1 permit <ip interno> 0.0.0.7
```

Análise

Configuração do Cisco Router

A configuração de rotas num router comercial passa por, inicialmente, perceber quais as portas *Ethernet* disponíveis e de que tipo são (exemplo: *fast-ethernet, gigabit*, etc) para o *router* especificado no *hostname* do ficheiro de configuração.
No caso da configuração utilizada as portas são do tipo *FastEthernet* havendo, como visível na imagem da arquitetura, uma para uma rede local (neste caso 172.16.254.0/24) e outra para a rede exterior (172.16.30.0/24). No exemplo dado nos principais comandos de configuração, assume-se a porta 0 como estando ligada à rede local, e a porta 1 à rede exterior, daí essa configuração. No que toca às rotas definidas assume-se a *default* como a rede exterior, sendo que para aceder a um *ip* interno deve-se seguir um caminho pela rede local.

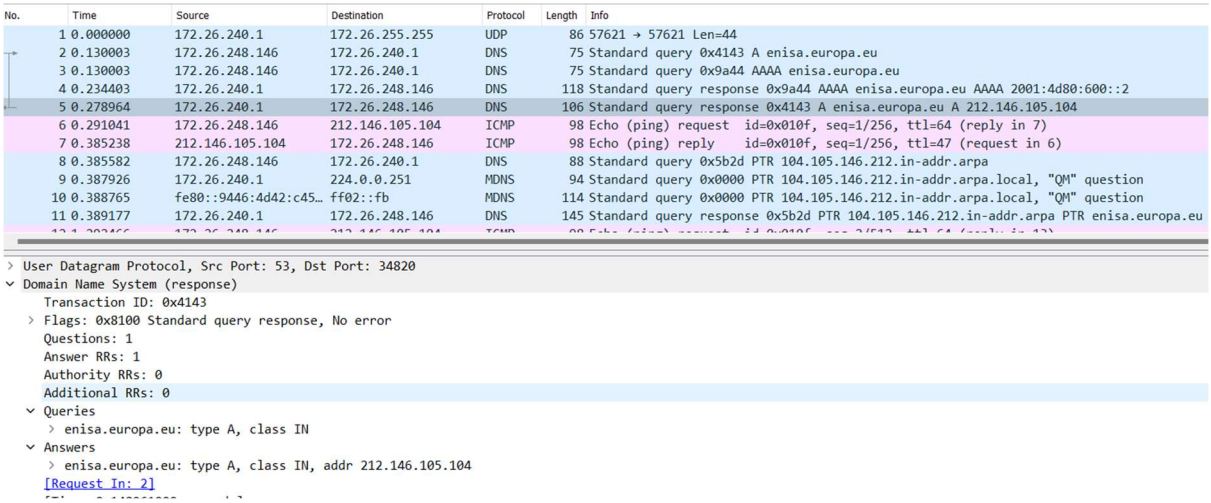
No que consta à configuração da NAT no router, entende-se que a interface conectada à *internet* será a especificada no comando "ip nat pool" (172.16.30.0 – rede exterior), havendo, o número de ips listados na access-list disponíveis para *NATing,* neste caso 1 abstrato, no entanto visto que o router está a fazer overloading permite que endereços de redes locais sejam identificados por um endereço global que pode ser aí indicado.

O objetivo principal da Network Adress Translation (NAT) é fazer a tradução/associação de endereços locais para o exterior e vice-versa, por exemplo, de forma a mascarar o remetente/destinatário para que certos endereços se consigam conectar com redes exteriores, usando, por exemplo, um *ip* único que representa todos os dispositivos da mesma rede local.

Configuração do DNS

Este passo de configuração é relevante para a conexão de máquinas da rede *IP* a um servidor de *DNS* que efetua a tradução de *hostnames* para endereços *IP* para analisar como isso muda a forma como as máquinas se conectam com a *Internet*.

O serviço de DNS foi configurado no ficheiro *resolv.conf* das nossas máquinas onde foi inserida uma nova entrada, com o nome e respetivo endereço *ip*: "142.250.200.142 youtubas". Esta inserção, no entanto, no momento de pedir pacotes (ping) para o endereço introduzido, não permite a visualização de pacotes DNS, sendo a justificação a rota para este *ip* já estar definida em */etc/hosts*. Já que, se forem pedidos pacotes a outro endereço como o explicitado "enisa.europa.eu" é realizado um pedido DNS para conseguir descobrir o seu *ip*.



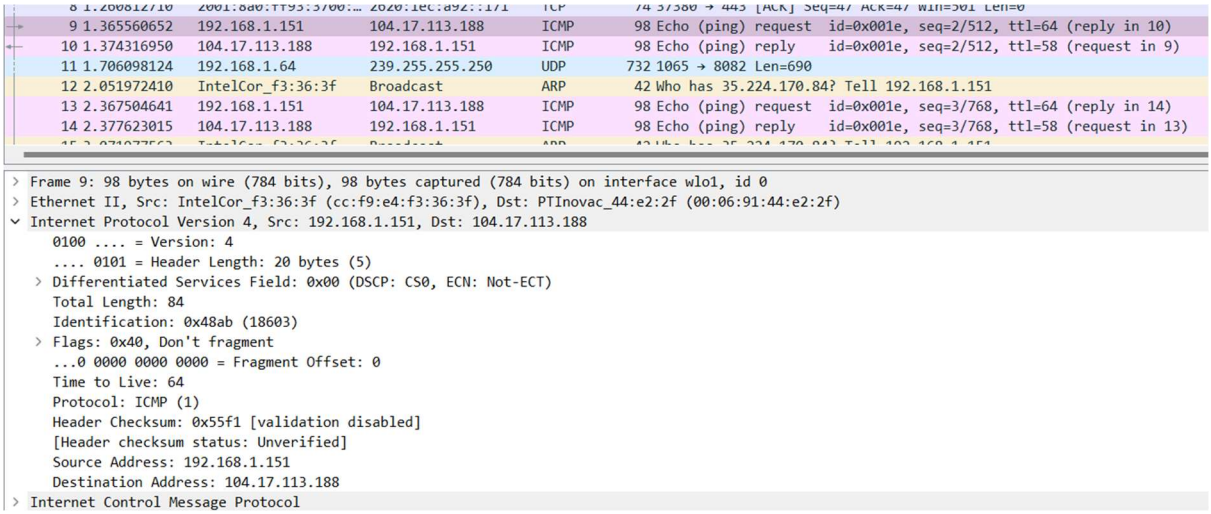
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.26.240.1	172.26.255.255	UDP	86	57621 → 57621 Len=44
2	0.130003	172.26.248.146	172.26.240.1	DNS	75	Standard query 0x4143 A enisa.europa.eu
3	0.130003	172.26.248.146	172.26.240.1	DNS	75	Standard query 0x9a44 AAAA enisa.europa.eu
4	0.234403	172.26.240.1	172.26.248.146	DNS	118	Standard query response 0x9a44 AAAA enisa.europa.eu AAAA 2001:4d80:600::2
5	0.278964	172.26.240.1	172.26.248.146	DNS	106	Standard query response 0x4143 A enisa.europa.eu A 212.146.105.104
6	0.291041	172.26.248.146	212.146.105.104	ICMP	98	Echo (ping) request id=0x010f, seq=1/256, ttl=64 (reply in 7)
7	0.385238	212.146.105.104	172.26.248.146	ICMP	98	Echo (ping) reply id=0x010f, seq=1/256, ttl=47 (request in 6)
8	0.385582	172.26.248.146	172.26.240.1	DNS	88	Standard query 0x5b2d PTR 104.105.146.212.in-addr.arpa
9	0.387926	172.26.240.1	224.0.0.251	MDNS	94	Standard query 0x0000 PTR 104.105.146.212.in-addr.arpa.local, "QM" question
10	0.388765	fe80::9446:4d42:c45…	ff02::fb	MDNS	114	Standard query 0x0000 PTR 104.105.146.212.in-addr.arpa.local, "QM" question
11	0.389177	172.26.240.1	172.26.248.146	DNS	145	Standard query response 0x5b2d PTR 104.105.146.212.in-addr.arpa PTR enisa.europa.eu

```
> User Datagram Protocol, Src Port: 53, Dst Port: 34820
v Domain Name System (response)
    Transaction ID: 0x4143
  > Flags: 0x8100 Standard query response, No error
    Questions: 1
    Answer RRs: 1
    Authority RRs: 0
    Additional RRs: 0
  v Queries
    > enisa.europa.eu: type A, class IN
  v Answers
    > enisa.europa.eu: type A, class IN, addr 212.146.105.104
    [Request In: 2]
```

Assim, como se mostra na figura acima, os pacotes trocados por DNS transportam, do *host* para o *server* um pacote com o *hostname*, sendo retornado o seu endereço *ip*.

Linux Routing

Nas nossas máquinas pessoais, tentamos adicionar rotas ao ambiente Linux, começando por tomar notas relativas aos *ip adresses default*: 192.168.1.254. Posteriormente, apagamos a nossa entrada por omissão (*sudo ip route del 0/0*) o que levou a uma perda de conexão, já que o DNS também não estava disponível, pois nenhuma rota existia no momento. Desse modo não conseguíamos obter qualquer tipo de resposta no que toca a pacotes de informação. Apenas depois de se adicionar uma rota específica para a inicialmente rota por *default*, é que foi possível observar pedidos e pacotes a serem transmitidos, como é o caso dos pacotes em baixo.



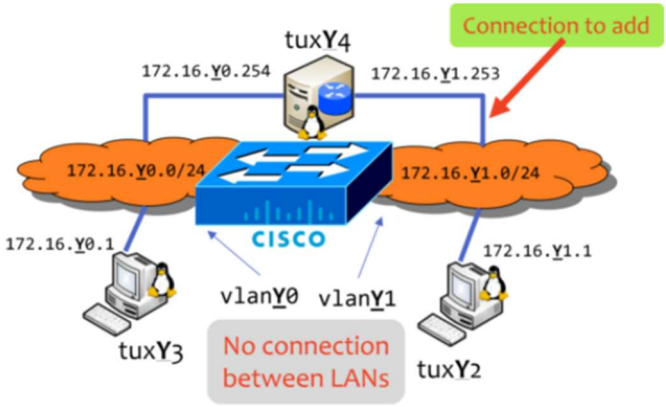
Como é visível na imagem superior os pacotes do tipo ICMP observados são do tipo pedido e resposta. Isto acontece, já que os *ip* reconhecem as rotas que os leva a permitir haver transferência de informação. Neste caso os *ip adresses* associados são os dois mencionados em cima (192.168.1.151 [MAC: cc:f9:e4:f3:36:3f] - emissor no pedido e destinatário na resposta e 104.17.113.188 [MAC: 00:06:91:44:e2:2f] – destinatário no pedido e emissor na resposta).

Assim após a experiência para além da rota inicial explicitado como default, ficamos com mais uma que passa por redirecionar do default para 104.17.113.188.

Experiência 4 – Configuração de um router Linux e de um router CISCO

Configuração router Linux

Arquitetura da rede



Objetivos da experiência

O objetivo desta experiência é configurar um router num computador Linux que nos permitirá fazer pedidos entre duas *VLANs* diferentes.

Principais comandos de configuração

Tux24:

ifconfig eth1 down
ifconfig eth1 up 172.16.21.253

Tux22: *ip route add 172.16.20.0/24 via 172.16.21.253*

Tux23: *ip route add 172.16.21.0/24 via 172.16.20.254*

Listar rotas: *route -n*

Análise

Todos os *tuxs* apresentam uma rota que lhes permite comunicar nas próprias *VLANs*. O *tux23* apresenta a rota para a gama de endereços 172.16.20.0/24 pela *gateway* 0.0.0.0 (*broadcast*), que lhe permite entrar em contacto com qualquer computador que esteja na *vlan20*. Semelhante o *tux*22 tem uma rota que lhe permite mandar informação na *vlan*21 e o *tux*23 apresenta uma rota para a *vlan*20 e outra para a *vlan*21, já que este computador tem uma porta *FastEthernet* dedicada para uma das *VLANs*.

Adicionalmente foi adicionado no *tux*23 uma rota com a gama de endereços 172.16.21.0/24 pela *gateway* 172.16.20.254 que faz o redireccionamento dos pedidos do *tux*23 para a *vlan*21. Esta rota permite ao *tux*23, por exemplo, mandar mensagens ao *tux22* que está numa *vlan* diferente da dele.

No *tux22* foi adicionado uma rota semelhante para a gama de endereços 172.16.20.0/24 pela *gateway* 172.16.21.253. Esta diz ao *tux*22 que quando quiser entrar em contato com algum computador da *vlan*20 terá de enviar o seu pedido para o router 172.16.21.253.

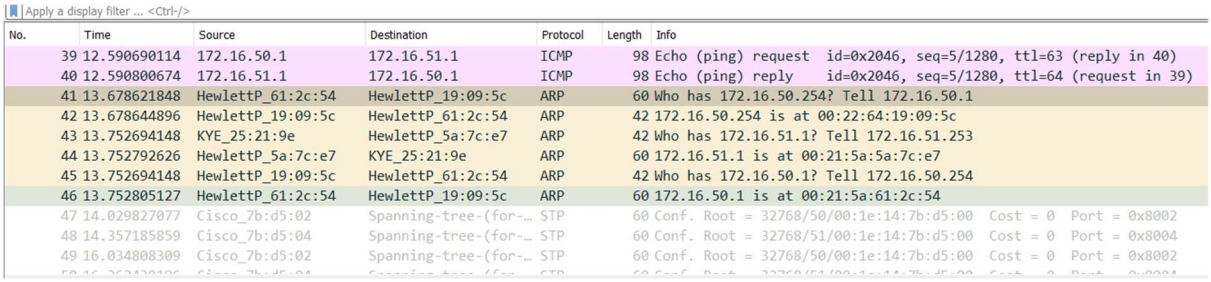
É de notar que tanto o *ip* 172.16.20.254 e o *ip* 172.16.21.253 se referem ao *tux*24, mas em placas de rede diferentes. Devido á tabela de encaminhamento definida neste *tux* ele vai conseguir encaminhar os pedidos que recebe das diferentes *VLANs*.

As informações mais relevantes que uma entrada da tabela de encaminhamento contem são: **Destino**/**Gateway**/**Interface**.

- Destino – gama de *IPs* ou IP do destino.
- Gateway – IP do computador para onde deve ser enviada a mensagem de forma a chegar ao destino.
- Interface – placa de rede usada para enviar a mensagem *eht0*/*eth1*/*eth2*.

Graças a esta tabela de encaminhamento os computadores sabem para onde devem enviar um dado pacote.

Na experiência proposta o objetivo é do *tux*23 pingar o *tux*22 que estão em *VLANs* diferentes, mas que conseguem estabelecer ligação através do *tux*24. Na imagem a seguir é possível ver quais foram os pacotes ARP observados:

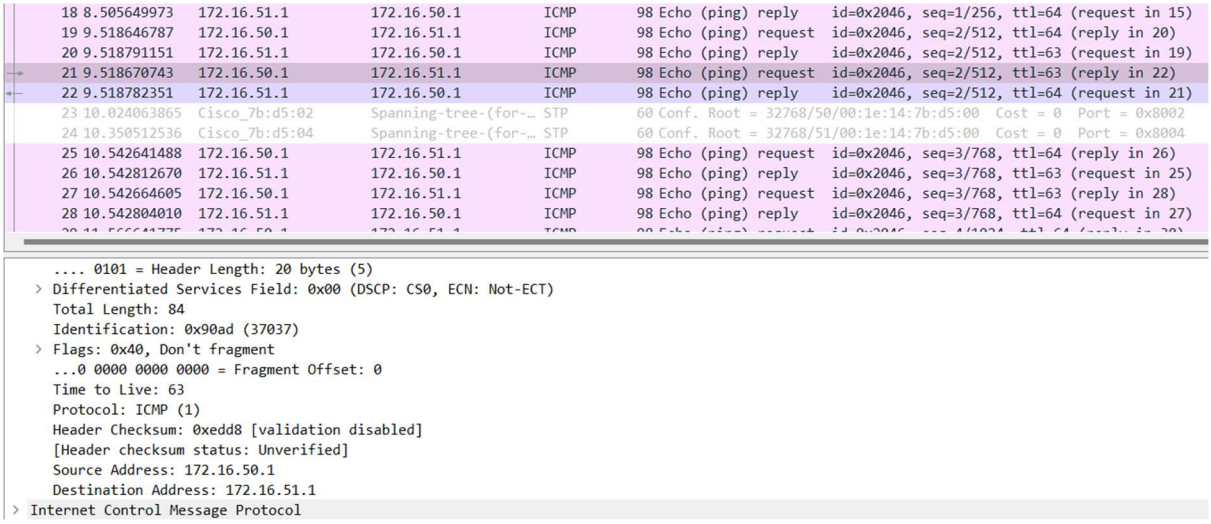

Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
39	12.590690114	172.16.50.1	172.16.51.1	ICMP	98	Echo (ping) request id=0x2046, seq=5/1280, ttl=63 (reply in 40)
40	12.590800674	172.16.51.1	172.16.50.1	ICMP	98	Echo (ping) reply id=0x2046, seq=5/1280, ttl=64 (request in 39)
41	13.678621848	HewlettP_61:2c:54	HewlettP_19:09:5c	ARP	60	Who has 172.16.50.254? Tell 172.16.50.1
42	13.678644896	HewlettP_19:09:5c	HewlettP_61:2c:54	ARP	42	172.16.50.254 is at 00:22:64:19:09:5c
43	13.752694148	KYE_25:21:9e	HewlettP_5a:7c:e7	ARP	42	Who has 172.16.51.1? Tell 172.16.51.253
44	13.752792626	HewlettP_5a:7c:e7	KYE_25:21:9e	ARP	60	172.16.51.1 is at 00:21:5a:5a:7c:e7
45	13.752694148	HewlettP_19:09:5c	HewlettP_61:2c:54	ARP	42	Who has 172.16.50.1? Tell 172.16.50.254
46	13.752805127	HewlettP_61:2c:54	HewlettP_19:09:5c	ARP	60	172.16.50.1 is at 00:21:5a:61:2c:54
47	14.029827077	Cisco_7b:d5:02	Spanning-tree-(for-…	STP	60	Conf. Root = 32768/50/00:1e:14:7b:d5:00 Cost = 0 Port = 0x8002
48	14.357185859	Cisco_7b:d5:04	Spanning-tree-(for-…	STP	60	Conf. Root = 32768/51/00:1e:14:7b:d5:00 Cost = 0 Port = 0x8004
49	16.034808309	Cisco_7b:d5:02	Spanning-tree-(for-…	STP	60	Conf. Root = 32768/50/00:1e:14:7b:d5:00 Cost = 0 Port = 0x8002

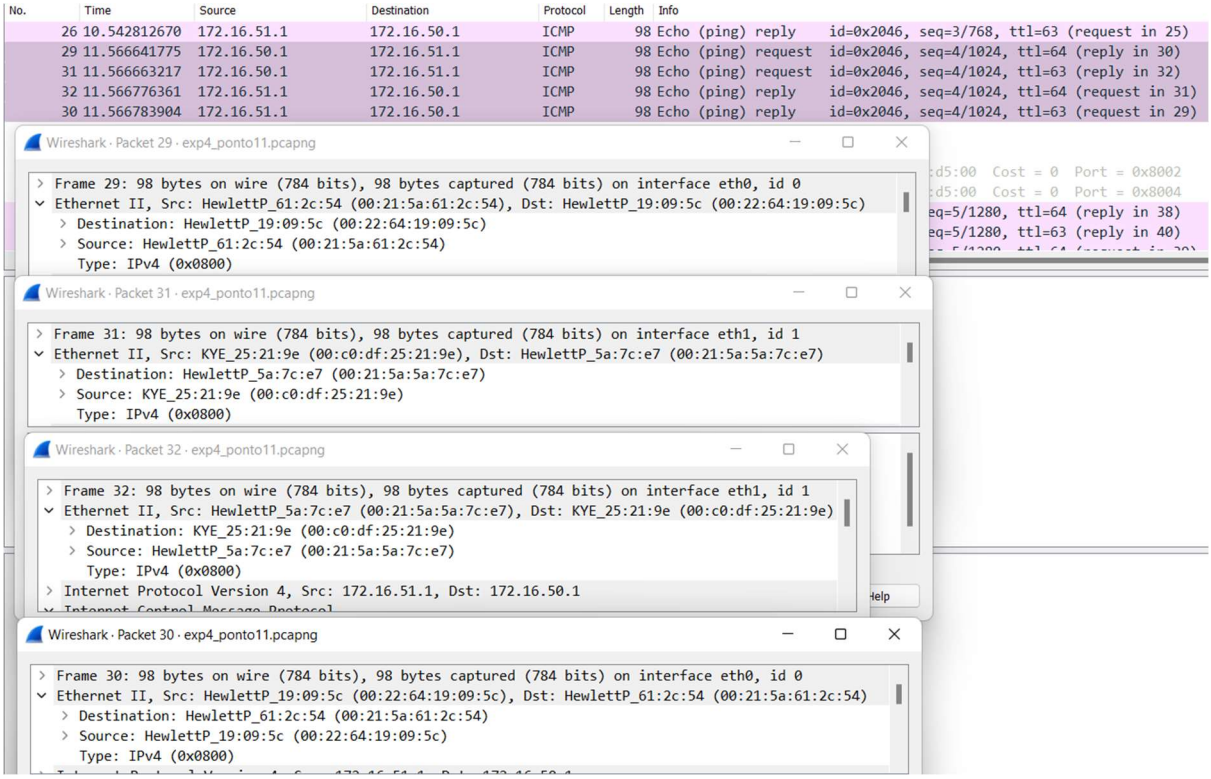
É de notar que neste log é referida a bancada 5, pois neste dia foi preciso trocar a bancada devido a um problema nas placas de rede da bancada 2.

Para se estabelecer a ligação pretendida o *tux23* terá de saber os endereços *MAC* de todas as máquinas que se encontram no caminho entre ele e o *tux22*, com *IP* 172.16.21.1. Devido a isso ele inicialmente irá perguntar em *broadcast*, endereço *MAC* = 00:00:00:00:00:00, na sua rede, quem tem o endereço 172.16.20.254, pois devido as rotas definidas na sua tabela de encaminhamento, esse é o *IP* para onde deve mandar o seu pedido. Após esse pedido o *tux*24 irá responder com o seu endereço *MAC*, ficando assim o *tux23* a saber parte do caminho a tomar. De seguida o próximo endereço desconhecido é do 172.16.51.1 por parte do *tux24,* devido a isso sairá um pedido da porta *eth1* do *tux*24 de *IP* 172.16.51.253, uma vez mais com o endereço *MAC* a 00:00:00:00:00:00, à procura do computador associado. Desta vez será o *tux22* a responder com o seu endereço *MAC*, ficando assim estabelecida mais uma entrada nas tabelas de ARP. Nesta fase o caminho *tux23* -> *tux22*, está estabelecido, falta agora estabelecer o caminho *tux22* -> *tux23*. Para tal como o *tux22* já conhece o caminho para o *tux24* vai ser apenas mandado um pacote *ARP* em *broadcast* à procura do endereço *MAC* do *tux23*, por parte do *tux*24. O *tux23* ao receber o pedido irá enviar o seu endereço *MAC* ficando assim todos os *tuxs* com conhecimento dos endereços MAC para onde têm de enviar o pacote de informação *ICMP*. A partir deste qualquer comunicação entre o *tux*23 e *tux*22 vai ser realizada sem qualquer envido de pacotes *ARP,* pois os endereços *MAC* dos intervenientes são todos conhecidos.

Continuando a analise do *ping* do *tux*23 para o *tux*22, podemos ver na imagem a seguir mais informação sobre a trama *ICMP* enviada:



Nesta fase em que os endereços MAC dos intervenientes no transporte do pacote estão mapeados, todos os pacotes *ICMP* de *request* gerados pelo comando *ping*, vão conter como *IP* de destino 172.16.21.1 *IP* do *tux*22 e como *IP* de origem o *IP* 172.16.20.1 do *tux*23. Nos pacotes *ICMP reply* os *IP* de destino e origem vão se encontrar trocados. Mas porque é que podemos ver na imagem acima 4 pedidos *ICMP* com o mesmo número de sequência e *IPs*? Isto acontece, pois, apesar de os *IPs* de origem e destino se manterem constantes ao longo de um pedido *request*/*reply*, os endereços *MAC* vão variar! Na imagem a seguir podemos ver melhor os endereços *MAC* associados a todos os pedidos *ICMP* relacionados com o número de sequencia 1024:



Portanto na primeira trama *request ICMP* os endereços *MAC* são:

Origem *MAC (tux23 eth0)* -> Destino *MAC (tux24 eth0)*

Segunda trama request *ICMP*:

Origem *MAC* (*tux24 eth1*) -> Destino *MAC* (*tux22 eth0*)

O ping chegou ao destino pretendido agora vêm as tramas *reply* ICMP em que os endereços *MAC* se vão apresentar invertidos:

Origem *MAC (tux22 eth0)* -> Destino *MAC (tux24 eth1)*

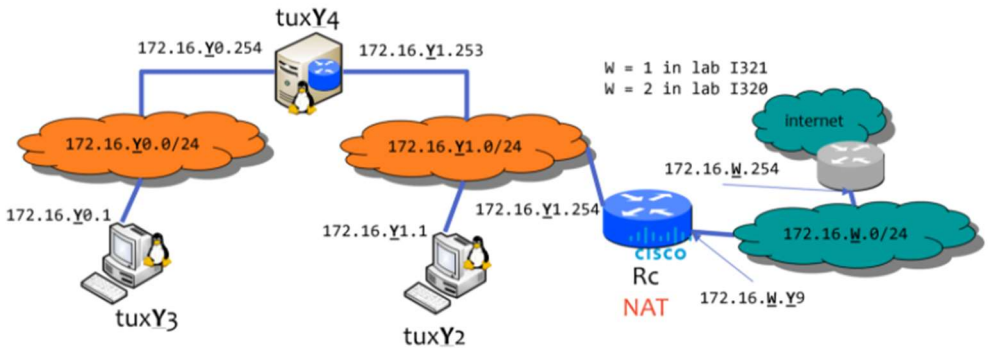
Segunda trama request *ICMP*:

Origem *MAC* (*tux24 eth0*) -> Destino *MAC* (*tux23 eth0*)

É deste modo que o ping que propaga ao longo da nossa rede.

Configuração router Cisco

Arquitetura da rede



Objetivos da experiência

O objetivo desta experiência é configurar um router Cisco que nos irá permitir contactar com uma rede externa. Esta rede externa fará a conexão com a internet, portanto no final desta experiencia devemos ser capazes de ter acesso á internet no nosso computador.

Principais comandos de configuração

Router CISCO (GTKTerm):

Modificar running-config do router CISCO:

enable
configure terminal
(comandos de configuração do cisco)
end

Persistir as modificações no router CISCO:

enable
configure terminal
copy running-config startup-config
reload

Análise

Desta vez o objetivo da nossa experiência é do *tux23* entrar em contacto, por exemplo, com os servidores da google (8.8.8.8). Para chegar a esse resultado foi necessário configurar o router CISCO com as definições que já foram explicadas na experiência 3. Foi também necessário adicionar no *tux24* e *tux22* uma rota *default* para o router CISCO de forma a espalhar os pedidos para servidores externos. No *tux23* adicionamos também uma rota *default* para o *tux24,* permitindo assim que este estabeleça contacto com a rede externa.

Tendo a nossa rede configurada é agora possível estabelecer contacto com a internet, a partir de qualquer um dos *tuxs* na nossa rede local. Se um *ping* para o endereço 8.8.8.8 for realizado a partir do *tux23* o caminho que os pacotes *ICMP* irão percorrer será:

- *tux23* - *tux24* - *router CISCO* (onde se irá realizar a operação de NAT) - *Router Internet*

Nesta fase o nosso pacote está a navegar pela rede pública, com o seu endereço de origem alterado para 172.16.2.254, endereço referente ao router da sala. Quando a resposta for recebida o seu endereço será alterado uma vez mais para o seu endereço original e enviado de volta para o *tux23*. Seguindo o seguinte caminho

- *Router Internet* - *router CISCO* (o NAT é desfeito) - *tux24* - *tux23*

Deste modo torna-se possível estabelecer uma comunicação entre a nossa rede privada local e a rede pública.

Conclusão

Este trabalho de RC teve como objetivo o desenvolvimento de uma aplicação que permitisse o download de um ficheiro através de conexões utilizando os protocolos *FTP* e *TCP*, e também a configuração em vários passos de uma rede *IP*, de modo a entender o funcionamento de várias máquinas, como o *switch*, o *router* e perceber diversas técnicas, dos quais são exemplo o *DNS* e o *NAT* e outros protocolos *ICMP* e *ARP*.

Do nosso ponto de vista, todos os objetivos foram cumpridos e sentimos que este foi bom para um melhor entendimento e consolidação dos temas abordados e estudados desta forma de modo mais explícito.

Nas páginas seguintes apresentamos, em anexo, o código correspondente à aplicação desenvolvida na primeira parte do projeto. Os principais comandos de configuração, bem como os logs capturados mais relevantes foram introduzidos

Anexo I - Código fonte

```c
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "utils.h"
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int dowloadFile(int downloadfd, char* file){
    //writing dowloaded data to file
    char buf[255];

    int fd = open(file, O_WRONLY | O_CREAT | O_TRUNC, 0666);

    int size;
    while ((size = readSocket(&downloadfd, buf)) > 0){
        write(fd,&buf,size);
    }
    close(fd);

    if (closeSocket(&downloadfd) != 0){
        return -1;
    };
    return 0;
}
```

```c
int main(int argc, char *argv[]){
    struct hostent *h;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s ftp://[<user>:<password>@]<host>/<url-path>\n", argv[0]);
        exit(-1);
    }

    Args args;
    if( parseArguments(argv[1], &args) != 0){
        return -1;
    }

    if( getIp(&args) != 0){
        return -1;
    }

    printf("Host: %s\n",args.host);
    printf("Path: %s\n",args.path);
    printf("FileName: %s\n",args.filename);
    printf("user: %s\n",args.user);
    printf("password: %s\n",args.password);
    printf("ip: %s\n",args.ip);

    int socketfd;
    char buf[255];
    char command[255] ="";
    int val = 0;
    int port;
    int downloadfd;


    //create and connect socket
    if (connectSocket(&args, &socketfd) != 0){
        fprintf(stderr, "Error connecting socket\n");
        return -1;
    }

    //read begin of connection
    if(readText(&socketfd, "220")!= 0){
        fprintf(stderr, "Error first message (220)\n");
        return -1;
    }
```

```c
strcpy(command, "user ");
strcat(command, args.user);
strcat(command, "\n");
writeSocket(&socketfd, command);
if(readText(&socketfd, "331")!= 0){
    fprintf(stderr, "Error in login user (331)\n");
    return -1;
}

//giving password for user credential
strcpy(command, "pass ");
strcat(command, args.password);
strcat(command, "\n");
writeSocket(&socketfd, command);
if(readText(&socketfd, "230")!= 0){
    fprintf(stderr, "Error in login password (230)\n");
    return -1;
}

//passive mode selection
writeSocket(&socketfd, "pasv\n");
do{
    if(readSocket(&socketfd, buf) < 0){
        return -1;
    }
    printf("%s",buf);

    val = checkCode(buf,"227");
    if(val == -1){
        fprintf(stderr, "Error entering passive mode (227)\n");
        return -1;
    }
}while(val == 0);

//calculating the port for the other tux
if((port = parsePassivePort(buf)) == -1){
    fprintf(stderr, "Error discovering port (227)\n");
    return -1;
}
args.port = port;

//creating new socket to receive file data
if (connectSocket(&args, &downloadfd) != 0){
    fprintf(stderr, "Error connecting socket\n");
    return -1;
}
```

```c
//passing retrive file command
strcpy(command, "retr ");
strcat(command, args.path);
strcat(command, "\n");
writeSocket(&socketfd, command);
if(readText(&socketfd, "150")!= 0){
    fprintf(stderr, "Error opening transfer (150)\n");
    return -1;
}

//reads and writes the data on the file
if( dowloadFile(downloadfd, args.filename) != 0){
    fprintf(stderr, "Error downloading file\n");
    return -1;
}

//waiting for transference complete
if(readText(&socketfd, "226")!= 0){
    fprintf(stderr, "Error transfer completed (226)\n");
    return -1;
}

//quiting socket
writeSocket(&socketfd, "quit\n");
if(readText(&socketfd, "221")!= 0){
    fprintf(stderr, "Error quiting (221)\n");
    return -1;
}

if (closeSocket(&socketfd) != 0){
    return -1;
};

if (closeSocket(&downloadfd) != 0){
    return -1;
};

return 0;
```

```c
int parseArguments(char *commandArgs, Args *args){
    char * data = strtok(commandArgs,"//");

    //checking protocol
    if(strcmp(data,"ftp:") != 0){
        fprintf(stderr, "Error parsing string\n");
        return -1;
    }

    //checking host
    data = strtok(NULL, "/");
    if(data == NULL){
        fprintf(stderr, "Error parsing string (host)\n");
        return -1;
    }
    char * host = data;
    memset(args->host, '\0', 255);
    strcpy(args->host, data);

    //checking path
    data = strtok(NULL, "\0");
    if(data == NULL){
        fprintf(stderr, "Error parsing string (path)\n");
        return -1;
    }
    memset(args->path, '\0', 255);
    strcpy(args->path, data);

    char path[255];
    strcpy(path, args->path);
    char* file = strtok(path, "/");
    while(file != NULL){
        memset(args->filename, '\0', 255);
        strcpy(args->filename, file);
        file = strtok(NULL, "/");
    }

    //Checking if credentials where given
    data = strtok(host, ":");
    if (strcmp(data,args->host) == 0){
        printf("No credentials given\n");
        memset(args->user, '\0', 255);
        strcpy(args->user, "anonymous");
        memset(args->password, '\0', 255);
        strcpy(args->password, "123");
        return 0;
    }

    //checking credentials - user
    if(data == NULL){
        fprintf(stderr, "Error parsing string (user)\n");
        return -1;
    }
    memset(args->user, '\0', 255);
    strcpy(args->user, data);

    //checking credentials - password
    data = strtok(NULL, "@");
    if(data == NULL){
        fprintf(stderr, "Error parsing string (password)\n");
        return -1;
    }
    memset(args->password, '\0', 255);
    strcpy(args->password, data);

    data = strtok(NULL, "\0");
    if(data == NULL){
        fprintf(stderr, "Error parsing string (host)\n");
        return -1;
    }
    memset(args->host, '\0', 255);
    strcpy(args->host, data);


    return 0;
}

int getIp(Args * args){
    struct hostent *h;

    if ((h = gethostbyname(args->host)) == NULL) {
        fprintf(stderr, "Error geting ip for %s", args->host);
        return -1;
    }

    memset(args->ip, '\0', 255);
    strcpy(args->ip, inet_ntoa(*((struct in_addr *) h->h_addr)));

    args->port = 21;

    return 0;
}
```

```c
int connectSocket(Args * args, 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(args->ip);    /*32 bit Internet address network byte ordered*/
    server_addr.sin_port = htons(args->port);        /*server TCP port must be network byte ordered */

    /*open a TCP socket*/
    if (((*socketfd) = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        fprintf(stderr, "Error creating socket\n");
        return -1;
    }

     /*connect to the server*/
    if (connect((*socketfd),(struct sockaddr *) &server_addr, sizeof(server_addr)) < 0) {
        fprintf(stderr, "Error connecting socket\n");
        return -1;
    }

    return 0;
}

int closeSocket(int *socketfd){
    if (close((*socketfd))<0) {
        fprintf(stderr, "Error closing socket\n");
        return -1;
    }
    return 0;
}


int writeSocket(int *socketfd, char* buf) {
    size_t bytes;

    /*send a string to the server*/
    bytes = write((*socketfd), buf, strlen(buf));
    if (bytes > 0){
        printf("%s", buf);
        //printf("Bytes escritos %ld\n", bytes);
        return 0;
    }
    else {
        fprintf(stderr, "Error writing to socket\n");
        return -1;
    }
}
```

```c
int readSocket(int *socketfd, char * buf) {
    size_t bytes;
    bzero(buf, 255);

    bytes = read((*socketfd),buf ,254);

    if (bytes < 0) {
        fprintf(stderr, "Error reading from socket\n");
        return -1;
    }
    return bytes;
}

int checkCode(char * buf, char * code){
    char *data = strtok(buf, "\n");

    while(data != NULL){
        for(int i=0; i<3; i++){
            if(code[i] != data[i]){
                return -1;
            }
        }
        if(data[3] != '-') return 1;
        data = strtok(NULL,"\n");
    }

    return 0;
}

int readText(int *socketfd, char * code){
    char buf[255];
    int val = 0;
    do{
        if(readSocket(socketfd, buf) < 0){
            return -1;
        }
        printf("%s",buf);

        val = checkCode(buf,code);
        if(val == -1) return -1;
    }while(val == 0);
    return 0;
}

int parsePassivePort(char * buf){
    int port = 0;
    strtok(buf, ",");
    strtok(NULL, ",");
    strtok(NULL, ",");
    strtok(NULL, ",");
    port = atoi(strtok(NULL, ","))*256;
    port += atoi(strtok(NULL, ")"));

    return port;
}
```