

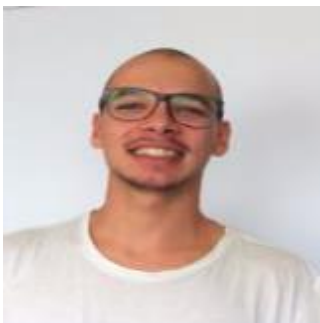


Universidade do Minho
Escola de Engenharia

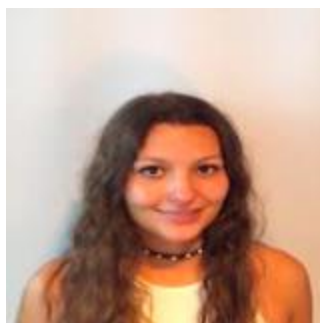
Engenharia de Telecomunicações e Informática

REDES DE COMPUTADORES II

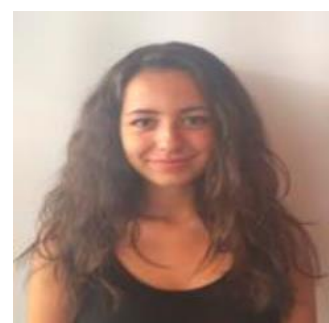
Grupo de trabalho:



Hugo Daniel da Costa Cunha
Machado
A80362



Joana Ramalho Querido
A81459



Rita Dias Rosas Lopes
A81111

Índice

Índice de Figuras	3
Índice de Tabelas	3
1. Introdução	4
2. Descrição do <i>Software</i>	5
2.1 Pesquisa e Compreensão de funções	5
2.2 Interpretação de endereços IP	8
3. Construção da topologia.....	9
4. Construção do <i>Software</i>	11
5. Ligação à <i>Internet</i>	12
6. Descrição dos testes realizados.....	13
Testes sobre a camada de transporte	13
7. Conclusão.....	16
8. Referências	17

Índice de Figuras

Figura 1-Topologia da rede.....	10
Figura 2-Configurações da firewall do router NAT.....	12
Figura 3-Comando <i>ping</i> analisado WireShark	13
Figura 4-Análise da ligação TCP	14
Figura 5-Análise da ligação UDP	14
Figura 6-Análise dos pacotes ao sair e entrar no nosso router.....	15
Figura 7-Análise dos pacotes ao sair e entrar no router NAT	15

Índice de Tabelas

Tabela 1-Tabela de Endereçamento.....	9
Tabela 2-Tabela de Encaminhamento.....	9

1. Introdução

No âmbito da unidade curricular Redes de Computadores II, foi proposto a elaboração deste projeto com objetivo de adquirir competências relacionadas com o processo de encaminhamento de pacotes em redes IP.

O projeto consiste no desenvolvimento de um router que seja capaz de receber pacotes IP (IPv4) e de os encaminhar, apenas através do processo de *forwarding*, com uma configuração manual da tabela de encaminhamento.

Iremos desenvolver um *software* que deverá estar à escuta em todas as interfaces de rede do *host*, e sempre que receber uma trama *Ethernet* verifica se esta transporta um pacote IP que precise de ser encaminhado. Caso se verifique este transporte, o *software* deve proceder ao respetivo *forwarding*.

Para o código fonte, optamos por usar a linguagem de programação C pois estamos mais familiarizados com esta e é uma linguagem mais popular no âmbito das redes e computadores.

2. Descrição do *Software*

2.1 Pesquisa e Compreensão de funções

Na pesquisa inicial, deparamo-nos com vários códigos tipo para o nosso objetivo, que nos ajudaram a perceber o principal problema (opensourceforu).

A diferença entre *rawsockets* e outros *sockets* é que os *rawsockets* dão-nos acesso aos níveis mais baixos da trama, tais como camada de *Ethernet*, camada de IP e camada de transporte, enquanto que os outros tipos de *sockets* apenas nos dão acesso à camada de transporte, na maior parte dos casos.

Primeira função com que nos deparamos foi a *socket AF_PACKET, SOCK_RAW, htons(ETH_P_ALL)*). Testamos a função e reparamos que esta recebe tudo, desde o cabeçalho *ethernet* à camada de aplicação. Para percebermos ao certo o funcionamento desta função, pesquisamos pelas várias *flags* inseridas.

socket(int domain, int type, int protocol)

- *AF_PACKET*: usado para utilizar o pacote ao nível do protocolo.
- *SOCK_RAW*: usado para receber o pacote completo (com todas as camadas).
- *htons(ETH_P_ALL)*: usado para restringir os pacotes recebidos, onde esta em específico recebe tudo.

A primeira mudança efetuada foi no parâmetro *protocol* da *socket*, onde restringimos a receção apenas a pacotes IP, *htons(ETH_P_IP)*. A partir deste ponto já só recebemos pacotes IP. A *socket* usada apenas recebe pacotes, não funciona para o envio.

Com a primeira parte da receção do pacote feita, falta-nos a receção em si. Para tal, usamos a função *recvfrom*(int sockfd, void *restrict buffer, size_t length, int flags, struct sockaddr *restrict address, socklen_t *restrict address_len).

- *sockfd*: descritor da socket;
- *buffer*: a variável que vai armazenar o pacote (char *buffer);
- *length*: o tamanho do buffer;
- *flags*: 0 (nenhuma flag adicional);
- *address*: estrutura que recebe o endereço de origem do pacote recebido;
- *address_len*: tamanho da estrutura que armazena o endereço de origem.

A partir deste ponto, só precisamos de descartar o cabeçalho *ethernet* visto que não será necessário no processo de encaminhamento. Para tal, como declaramos a variável

onde está armazenado o pacote como apontador, apenas precisamos de apontar para o início do cabeçalho IP.

```
buffer = (unsigned char *)(buffer + sizeof(eth)).
```

Após esta etapa concluída, prosseguimos para a fase mais importante: o encaminhamento.

Para esta fase, precisamos de abrir um *socket* novo para o envio e especificar a interface de saída conforme o IP destino.

Foi nesta etapa também que nos deparamos com a falta de uma função adicional, cujo objetivo é incluir o cabeçalho IP no envio do pacote pela socket.

Esta nova socket tem uns parâmetros diferentes:

```
socket(INET,RAW,IP)
```

- *INET*: AF_INET especifica o IPv4 como protocolo;
- *INET*: SOCK_RAW define o tipo de socket como raw;
- *IP*: htons(ETH_P_IP) restringe o protocolo IP.

Após a abertura da socket verificamos com um algoritmo nosso qual a interface de saída, cujo iremos explicar posteriormente.

De seguida definimos as opções do socket.

```
setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen)
```

Na primeira definição de opções fizemos a ligação da socket à interface, e para tal usamos:

- *int sockfd*: socket de envio;
- *int level*: SOL_socket usado para manipular sockets ao nível da aplicação;
- *int optname*: SO_BINDTODEVICE usado para fazer a ligação do socket à interface;
- *const void *optval*: “nome da interface” (ex: eth0);
- *socklen_t optlen*: tamanho do nome da interface (ex: 4bytes).

Na segunda especificamos que o cabeçalho IP seria enviado por nós, ao invés de ser feito automaticamente.

- *int sockfd*: socket de envio;
- *int level*: 0 (não especificado);
- *int optname*: IP_HDRINCL opção para incluir cabeçalho IP;
- *const void *optval*: um inteiro com valor de 1;
- *socklen_t optlen*: tamanho do inteiro.

Para finalizar o envio é preciso enviar o pacote, e visto que com *raw sockets* temos de usar a função `sendto()`, é necessário preencher uma estrutura com o endereço de destino. Nesta estrutura, apenas preenchemos os campos:

- *sin_family*: AF_INET endereço IPv4;
- *sin_addr.s_addr*: IP destino.

`sendto(int sockfd ,void *restrict buffer, size_t length, int flags, struct sockaddr *restrict address, socklen_t *restrict address_len)`

- *sockfd*: socket de envio;
- *buffer*: variável que contem o pacote IP;
- *length*: tamanho do pacote;
- *flags*: 0 (nenhuma flag adicional);
- *address*: estrutura de endereço de destino;
- *address_len*: tamanho da estrutura de destino.

2.2 Interpretação de endereços IP

Como requisito deste projeto, é necessário que o nosso programa leia de um ficheiro de configuração a tabela de encaminhamento da topologia.

Nesta etapa, deparamo-nos com um impasse em relação aos endereços IP e as suas máscaras. Tivemos de arranjar uma maneira prática de mexer com endereços IP e as suas máscaras de forma a descobrir se o endereço destino do pacote está contido ou não nessa gama de endereços. Para tal, com a recomendação de um colega, resolvemos converter o endereço IP em inteiro.

Esta conversão partiu do pressuposto que tanto um endereço IP e um inteiro têm 32bits de tamanho, portanto apenas temos de pegar no endereço IP na sua estrutura real e convertê-lo para binário. Desta forma, ficamos com o endereço IP num inteiro, o que torna o endereçamento mais prático.

Em relação à máscara (no ficheiro apresenta esta forma “/25”), subtraímos a 32 para obter o numero de bits. Assim, caso façamos 2 elevado ao número de bits, obtemos o número de endereços dessa máscara e, se somarmos ao IP de rede, obtemos a gama de endereços, o inicio e o final em inteiro. Por fim, para encontrar a interface de saída, apenas convertemos o endereço de destino em inteiro e verificamos se está contido na gama de endereços existente por interface.

3. Construção da topologia

A nossa topologia consiste em 5 redes locais, denominadas por PC1, PC2, PC3, PC4 e a última é uma rede de comunicação do nosso *router* com o *router* que faz ligação à *Internet*.

A nossa gama de endereços foi escolhida a partir de um exercício de endereçamento das aulas práticas, com o objetivo de utilizar máscaras diferentes no nosso programa.

O exercício consistia em alocar uma rede de 1000 PC's, outra com 500 PC's e duas com 80 PC's, a partir do endereço 140.0.128.0/21.

Visto que restaram endereços livres resolvemos usar a mesma rede nesses endereços livres para a comunicação entre os dois *routers*.

Tabela 1-Tabela de Endereçamento

Nome	IP de Rede	Máscara	Gama	IP de Difusão	IP de Router
PC1	140.0.128.0	/22	.128.1-.131.254	.131.255	.128.1
PC2	140.0.132.0	/23	.132.1-.133.254	.133.255	.132.1
PC3	140.0.134.0	/25	.134.1-.134.126	.134.127	.134.1
PC4	140.0.134.128	/25	.134.129-.134.254	.134.255	.134.129
Ligação à Internet	140.0.135.0	/30	.135.1-.135.2	.135.3	.135.1

Tabela 2-Tabela de Encaminhamento

Router	Rede Destino	Máscara	Interface de Saída
PC1	140.0.128.0	/22	eth0
PC2	140.0.132.0	/23	eth1
PC3	140.0.134.0	/25	eth2
PC4	140.0.134.128	/25	eth3
Internet	0.0.0.0	/0	eth4

Na Figura 1, podemos ver a nossa topologia final que segue o modelo em Estrela com as 5 redes locais. Podemos encontrar uma sexta rede com o endereço 10.0.2.10/24 que pertence à rede interna da máquina virtual.

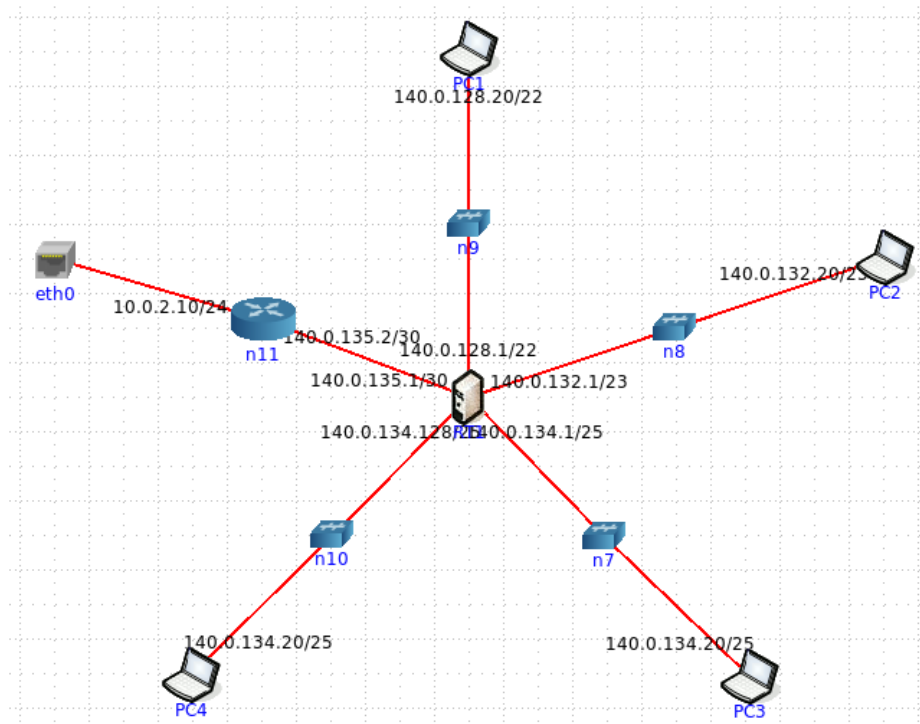


Figura 1-Topologia da rede

4. Construção do *Software*

Na *main()* inicializamos o nosso *software* com a função *readFile()* para fazer a leitura do ficheiro “*routingTable.txt*” que contém a tabela de encaminhamento.

Esta função lê linha a linha do ficheiro e guarda o IP de rede, máscara e *Interface* de saída na tabela de encaminhamento do programa através da função *routingTable()* que inicialmente verifica se a máscara é válida. Depois, pega na interface que leu e guarda na tabela de encaminhamento. Seguidamente, guarda o IP de rede, converte-o em inteiro (para uma utilização mais dinâmica do IP), guarda a máscara e para finalizar converte a máscara em número de bits e soma ao endereço IP em inteiro, para formar uma gama de endereços. Caso haja algum erro nos parâmetros inseridos retorna o valor -1 e sai do programa.

De seguida, através da função *rawSocket()* criamos uma *socket* para ouvir todo o tráfego pertencente ao protocolo IPv4.

A função *InterfaceIP()* guarda os endereços IP das interfaces do router. De seguida encontramos a função *fork()* que cria um filho que executa a função *ListenTraffic()*. Como parâmetro de segurança, para um programa mais robusto se por alguma razão o filho que está a ouvir o tráfego morrer, o pai vai criar outro filho para tomar o lugar dele.

Na função *ListenTraffic()* o pacote e o endereço de origem são recebidos através da função *recvfrom()*. Após a receção do pacote descartamos o cabeçalho *Ethernet*. De seguida o filho cria o neto para tratar do encaminhamento do pacote recebido já sem o cabeçalho *Ethernet*, a partir da função *sendMsg()*. O pai do processo neto, volta para a função *recvfrom()* para continuar à escuta do tráfego.

Na função *sendMsg()* é aberta uma *socket* para envio. Posteriormente com a função *findInt()* pegamos no cabeçalho IP do pacote e verificamos se o IP destino corresponde com um dos IP's das interfaces do *router*, se sim o filho suicida-se; caso contrário encontramos a interface de saída. Com a função *setsockopt()* estabelecemos a ligação entre a *socket* e a interface de saída.

Por fim preenchemos a estrutura com o endereço de destino, guardamos o tamanho total do pacote e enviamos o pacote para o respetivo destino. O filho que efetuou o envio suicida-se.

5. Ligação à *Internet*

Para conectar a nossa topologia à Internet tivemos que inserir uma porta (elemento do Core) RJ45 para nos ligarmos ao IP da máquina virtual.

Inicialmente ligamos a porta RJ45 diretamente ao nosso router, definimos a opção *DefaultRoute* do nosso *router* com:

```
"ip route add default via 10.0.2.2"
```

Para encaminhar o tráfego o IP de *gateway* com o WireShark verificamos que o pacote era enviado para a *Internet* mas não obtíamos resposta; em vez da resposta verificamos que recebíamos um ARP a perguntar quem continha o endereço de origem do pacote.

A partir desta troca de mensagem percebemos que o remetente para receber a resposta necessitaria de estar diretamente ligado à porta RJ45. Para dar a volta à situação resolvemos usar os nossos conhecimentos do projeto de Redes I e implementar um router NAT para mascarar o IP de origem com o IP de saída do router para quando receber o ARP, o ARP pedir o endereço de saída do *router* NAT. Desta forma, o *router* recebe o pacote, devolve o IP de origem e reencaminha para o nosso router chegando ao remetente.

```
iptables -t nat -A POSTROUTING -o eth1 -j MASQUERADE  
iptables -A FORWARD -i eth0 -j ACCEPT  
iptables -A FORWARD -o eth0 -j ACCEPT
```

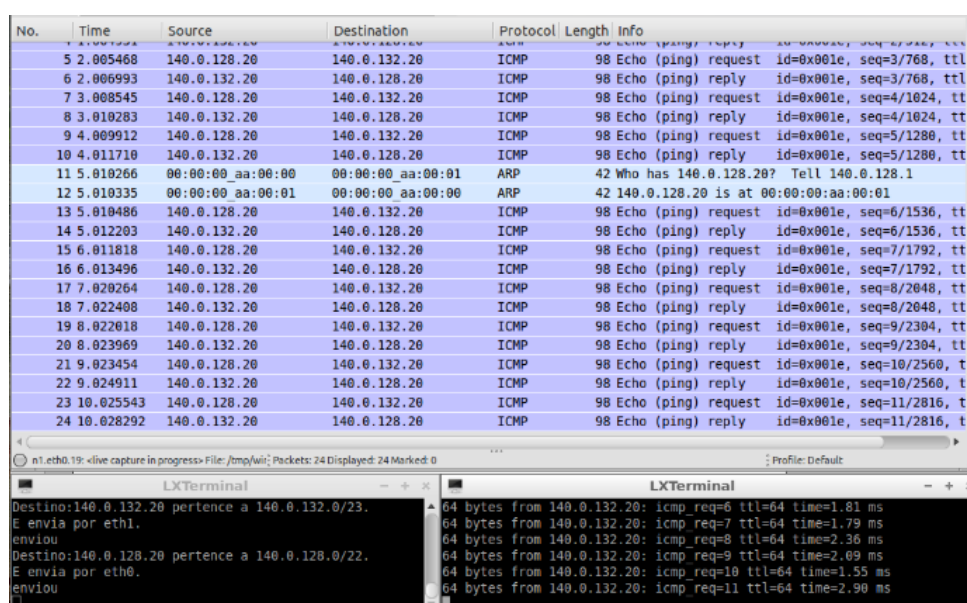
Figura 2-Configurações da firewall do router NAT

6. Descrição dos testes realizados

Para verificarmos se o nosso projeto estava a funcionar corretamente foram realizados vários testes no ambiente *Core* usando o WireShark para visualizarmos os resultados.

O primeiro teste, Figura 3, foi um simples *ping* entre dois computadores em redes diferentes ligadas ao nosso router para verificar a existência de comunicação entre as duas.

[Origem: 140.0.128.20; Destino: 140.0.132.20]



No.	Time	Source	Destination	Protocol	Length	Info
5	2.005468	140.0.128.20	140.0.132.20	ICMP	98	Echo (ping) request id=0x001e, seq=3/768, ttl=64
6	2.006993	140.0.132.20	140.0.128.20	ICMP	98	Echo (ping) reply id=0x001e, seq=3/768, ttl=64
7	3.008545	140.0.128.20	140.0.132.20	ICMP	98	Echo (ping) request id=0x001e, seq=4/1024, ttl=64
8	3.010283	140.0.132.20	140.0.128.20	ICMP	98	Echo (ping) reply id=0x001e, seq=4/1024, ttl=64
9	4.009912	140.0.128.20	140.0.132.20	ICMP	98	Echo (ping) request id=0x001e, seq=5/1280, ttl=64
10	4.011710	140.0.132.20	140.0.128.20	ICMP	98	Echo (ping) reply id=0x001e, seq=5/1280, ttl=64
11	5.010266	00:00:00 aa:00:00	00:00:00 aa:00:01	ARP	42	Who has 140.0.128.20? Tell 140.0.128.1
12	5.010335	00:00:00 aa:00:01	00:00:00 aa:00:00	ARP	42	140.0.128.20 is at 00:00:00 aa:00:01
13	5.010486	140.0.128.20	140.0.132.20	ICMP	98	Echo (ping) request id=0x001e, seq=6/1536, ttl=64
14	5.012203	140.0.132.20	140.0.128.20	ICMP	98	Echo (ping) reply id=0x001e, seq=6/1536, ttl=64
15	6.011818	140.0.128.20	140.0.132.20	ICMP	98	Echo (ping) request id=0x001e, seq=7/1792, ttl=64
16	6.013496	140.0.132.20	140.0.128.20	ICMP	98	Echo (ping) reply id=0x001e, seq=7/1792, ttl=64
17	7.020264	140.0.128.20	140.0.132.20	ICMP	98	Echo (ping) request id=0x001e, seq=8/2048, ttl=64
18	7.022408	140.0.132.20	140.0.128.20	ICMP	98	Echo (ping) reply id=0x001e, seq=8/2048, ttl=64
19	8.022018	140.0.128.20	140.0.132.20	ICMP	98	Echo (ping) request id=0x001e, seq=9/2304, ttl=64
20	8.023969	140.0.132.20	140.0.128.20	ICMP	98	Echo (ping) reply id=0x001e, seq=9/2304, ttl=64
21	9.023454	140.0.128.20	140.0.132.20	ICMP	98	Echo (ping) request id=0x001e, seq=10/2560, ttl=64
22	9.024911	140.0.132.20	140.0.128.20	ICMP	98	Echo (ping) reply id=0x001e, seq=10/2560, ttl=64
23	10.025543	140.0.128.20	140.0.132.20	ICMP	98	Echo (ping) request id=0x001e, seq=11/2816, ttl=64
24	10.028292	140.0.132.20	140.0.128.20	ICMP	98	Echo (ping) reply id=0x001e, seq=11/2816, ttl=64

Window	Content
LXTerminal	Destino:140.0.132.20 pertence a 140.0.132.0/23. E envia por eth1. enviou Destino:140.0.128.20 pertence a 140.0.128.0/23. E envia por eth0. enviou
LXTerminal	64 bytes from 140.0.132.20: icmp_req=6 ttl=64 time=1.81 ms 64 bytes from 140.0.132.20: icmp_req=7 ttl=64 time=1.79 ms 64 bytes from 140.0.132.20: icmp_req=8 ttl=64 time=2.36 ms 64 bytes from 140.0.132.20: icmp_req=9 ttl=64 time=2.09 ms 64 bytes from 140.0.132.20: icmp_req=10 ttl=64 time=1.55 ms 64 bytes from 140.0.132.20: icmp_req=11 ttl=64 time=2.90 ms

Figura 3-Comando *ping* analisado WireShark

Testes sobre a camada de transporte

O segundo teste, Figura 4, consistiu em testar o protocolo TCP entre dois computadores novamente em redes diferentes. Para testar o protocolo usamos o comando *-nc(netcat)* que consiste na criação de um “*chat*” entre dois terminais. No primeiro computador executamos o comando *-nc 9999 -l* que representa o servidor da ligação TCP, e no cliente executamos o comando *-nc 140.0.132.20 9999*.

Na Figura 4 podemos observar também o estabelecimento da ligação TCP (SYN/SYN+ACK/ACK) e também a finalização da mesma (FIN/FIN+ACK/ACK).

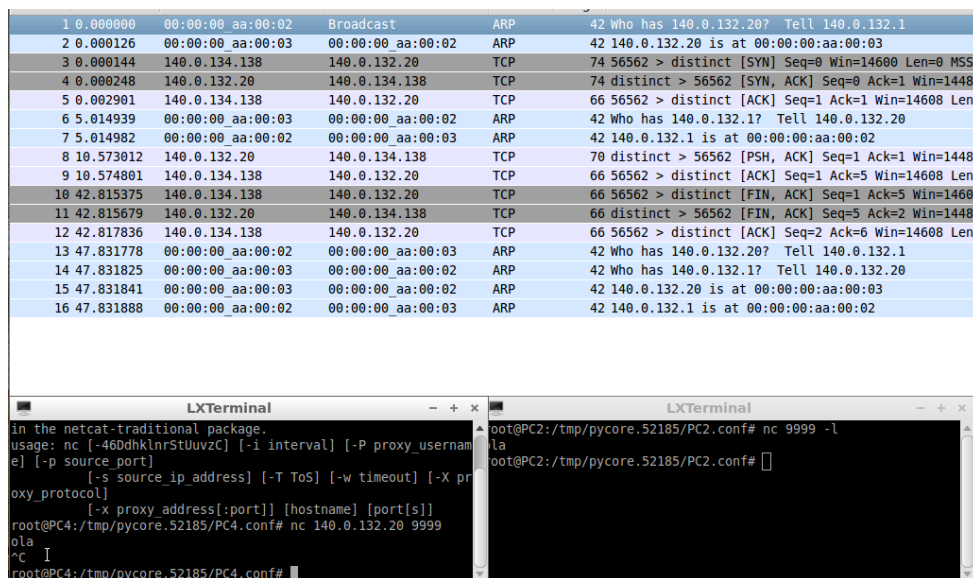


Figura 4-Análise da ligação TCP

No próximo teste, Figura 5, testamos o protocolo UDP, novamente através do comando `-nc` mas agora com o argumento `-u` que faz com que a ligação use o protocolo UDP.

Exemplo:

[Origem: 140.0.132.20]: `nc 9999 -l -u`

[Origem: 140.0.134.138]: `nc -u 140.0.132.20 9999`

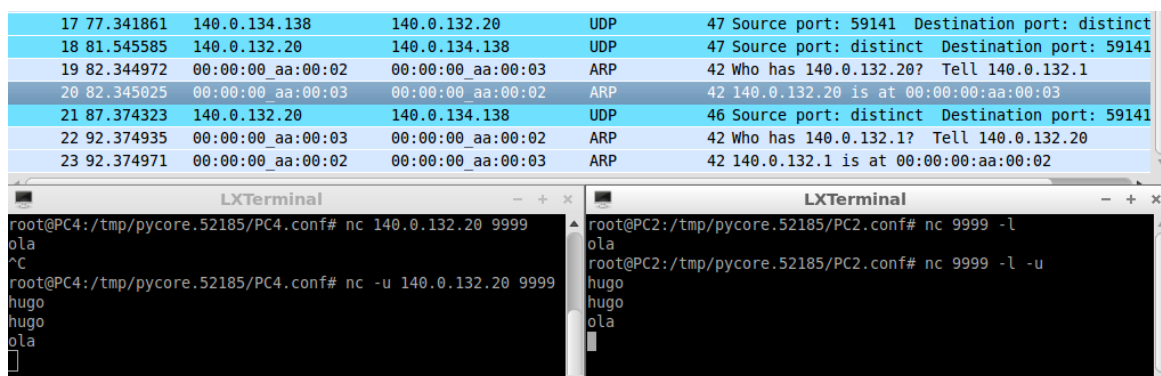


Figura 5-Análise da ligação UDP

Por fim, Figura 6 e Figura 7, testamos a ligação à *Internet* com um simples *ping* para o IP 8.8.8.8 e analisámos os pacotes antes e depois de passar no router NAT. De forma a verificar a alteração do IP de origem do pacote na saída do router NAT e da mesma forma na entrada.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	140.0.134.20	8.8.8.8	ICMP	98	Echo (ping) request id=0x001b, seq=1/256, ttl=64
2	0.122422	8.8.8.8	140.0.134.20	ICMP	98	Echo (ping) reply id=0x001b, seq=1/256, ttl=64
3	1.001609	140.0.134.20	8.8.8.8	ICMP	98	Echo (ping) request id=0x001b, seq=2/512, ttl=64
4	1.022614	8.8.8.8	140.0.134.20	ICMP	98	Echo (ping) reply id=0x001b, seq=2/512, ttl=64
5	2.002997	140.0.134.20	8.8.8.8	ICMP	98	Echo (ping) request id=0x001b, seq=3/768, ttl=64
6	2.021909	8.8.8.8	140.0.134.20	ICMP	98	Echo (ping) reply id=0x001b, seq=3/768, ttl=64
7	3.007406	140.0.134.20	8.8.8.8	ICMP	98	Echo (ping) request id=0x001b, seq=4/1024, ttl=64
8	3.025917	8.8.8.8	140.0.134.20	ICMP	98	Echo (ping) reply id=0x001b, seq=4/1024, ttl=64
9	4.008803	140.0.134.20	8.8.8.8	ICMP	98	Echo (ping) request id=0x001b, seq=5/1280, ttl=64
10	4.027645	8.8.8.8	140.0.134.20	ICMP	98	Echo (ping) reply id=0x001b, seq=5/1280, ttl=64
11	5.010218	140.0.134.20	8.8.8.8	ICMP	98	Echo (ping) request id=0x001b, seq=6/1536, ttl=64
12	5.016471	00:00:00_aa:00:08	00:00:00_aa:00:09	ARP	42	Who has 140.0.135.2? Tell 140.0.135.1
13	5.016664	00:00:00_aa:00:09	00:00:00_aa:00:08	ARP	42	140.0.135.2 is at 00:00:00_aa:00:09
14	5.031483	8.8.8.8	140.0.134.20	ICMP	98	Echo (ping) reply id=0x001b, seq=6/1536, ttl=64

Figura 6-Análise dos pacotes ao sair e entrar no nosso router

1	0.000000	00:00:00_aa:00:0a	Broadcast	ARP	42	Who has 10.0.2.2? Tell 10.0.2.10
2	0.000455	RealtekU_12:35:02	00:00:00_aa:00:0a	ARP	60	10.0.2.2 is at 52:54:00:12:35:02
3	0.000718	10.0.2.10	8.8.8.8	ICMP	98	Echo (ping) request id=0x001a, seq=1/256, ttl=64
4	0.019758	8.8.8.8	10.0.2.10	ICMP	98	Echo (ping) reply id=0x001a, seq=1/256, ttl=64
5	1.002217	10.0.2.10	8.8.8.8	ICMP	98	Echo (ping) request id=0x001a, seq=2/512, ttl=64
6	1.023600	8.8.8.8	10.0.2.10	ICMP	98	Echo (ping) reply id=0x001a, seq=2/512, ttl=64
7	2.004127	10.0.2.10	8.8.8.8	ICMP	98	Echo (ping) request id=0x001a, seq=3/768, ttl=64
8	2.024986	8.8.8.8	10.0.2.10	ICMP	98	Echo (ping) reply id=0x001a, seq=3/768, ttl=64
9	3.005621	10.0.2.10	8.8.8.8	ICMP	98	Echo (ping) request id=0x001a, seq=4/1024, ttl=64
10	3.025077	8.8.8.8	10.0.2.10	ICMP	98	Echo (ping) reply id=0x001a, seq=4/1024, ttl=64
11	4.007553	10.0.2.10	8.8.8.8	ICMP	98	Echo (ping) request id=0x001a, seq=5/1280, ttl=64
12	4.025888	8.8.8.8	10.0.2.10	ICMP	98	Echo (ping) reply id=0x001a, seq=5/1280, ttl=64
13	5.017229	10.0.2.10	8.8.8.8	ICMP	98	Echo (ping) request id=0x001a, seq=6/1536, ttl=64
14	5.037199	8.8.8.8	10.0.2.10	ICMP	98	Echo (ping) reply id=0x001a, seq=6/1536, ttl=64

Figura 7-Análise dos pacotes ao sair e entrar no router NAT

7. Conclusão

Com este projeto conseguimos por em prática os conceitos aprendidos em Redes I e II juntamente com os conhecimentos em linguagem de programação C noutras unidades curriculares.

Inicialmente deparamo-nos com algumas dificuldades na pesquisa das *raw sockets* devido ao pouco suporte *online* de informação relevante para os nossos objetivos.

Apesar destas dificuldades conseguimos arranjar soluções para a concretização integral do projeto, terminando com todas as etapas propostas concluídas.

8. Referências

1. https://www.binarytides.com/raw-sockets-c-code-linux/?fbclid=IwAR2edwjsMFggwdbJhrDula4xf0zrOzoqllzLKNi99tL_ycxoux-3WO-iEoA
2. https://www.quora.com/Whats-the-difference-between-the-AF_PACKET-and-AF_INET-in-python-socket?fbclid=IwAR0lUE9VWlmlI9zgUoEH9FLRWljxULNKs-E86hc1GDCtuFG-Mpnay2vC4GA
3. <http://plasmixs.github.io/raw-sockets-programming-in-c.html?fbclid=IwAR3pljrJw31MlsYd6ES2ZVUtmJoE-uLvbGSKnhw63ZCU-BSVrVFfk5HqcY8>
4. <https://linux.die.net/man/7/packet>
5. <https://linux.die.net/man/7/raw>
6. https://sock-raw.org/papers/sock_raw?fbclid=IwAR04lPaMx88iURmDak22JVja_vlleXthlXIN552MSlOSgPeJbhNbN6krudg
7. https://beej.us/guide/bgnet/html/multi/index.html?fbclid=IwAR12o_us6pXDn63GxdKdrwawmTLn0FKey9TVxfpIV2SkaJvOzuOwU1e8sp4
8. <https://opensourceforu.com/2015/03/a-guide-to-using-raw-sockets/?fbclid=IwAR3LXQX4G1M-S7CBcAf3nOehsTB43KYEe9LDUo6Ueti9RUztGkEXlt0fg0>