



UNIVERSIDAD DE BUENOS AIRES  
FACULTAD DE INGENIERÍA  
Año 2024 - 1<sup>er</sup> cuatrimestre

## REDES DE COMUNICACIONES (TB067)

### NIVELES DE TRANSPORTE Y RED

Fecha de entrega: 29/04/2024

ESTUDIANTE:

Rodríguez, Maximiliano Sebastián  
`masrodriguez@fi.uba.ar`

107604

## Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Ejercicio 1</b>	<b>2</b>
2.1. Análisis del tráfico en transferencia TCP masiva de host a servidor . . . . .	2
2.2. Datos enviados por unidad de tiempo . . . . .	6
<b>3. Ejercicio 2</b>	<b>7</b>
3.1. Conectividad. Uso de ping . . . . .	8
3.2. Relevamiento de rutas. Uso de traceroute . . . . .	11

## 1. Introducción

Para este trabajo práctico se tuvo como objetivo entender el funcionamiento de los Niveles de Transporte y Red en base a dos experimentos. El primero consistía en descargar una copia ASCII de Alicia en el país de las maravillas, enviarlo al servidor `gaia.cs.umass.edu` y capturar los paquetes mediante Wireshark. El segundo consistía en crear una red simulada mediante Imunes e investigar el comportamiento de esa red.

## 2. Ejercicio 1

### 2.1. Análisis del tráfico en transferencia TCP masiva de host a servidor

1. Se buscó el mensaje HTTP POST que cargó el archivo de texto en el servidor, este nos dice en qué segmento TCP comienza y en cual termina. Este nos indica que el *First boundary* y el *Last boundary* están marcados por la línea:

—WebKitFormBoundarypTNcrkMdaS6Yp1Kf.

271	2024-04-25 18:48:29,303279	192.168.0.40	128.119.245.12	TCP	1514	54299 → 80	[ACK] Seq=148091 Ack=1 Win=262656 Len=1460	[TCP segment of a reassembled PDU]
272	2024-04-25 18:48:29,303279	192.168.0.40	128.119.245.12	TCP	1514	54299 → 80	[ACK] Seq=149551 Ack=1 Win=262656 Len=1460	[TCP segment of a reassembled PDU]
273	2024-04-25 18:48:29,303279	192.168.0.40	128.119.245.12	TCP	1514	54299 → 80	[ACK] Seq=151011 Ack=1 Win=262656 Len=1460	[TCP segment of a reassembled PDU]
274	2024-04-25 18:48:29,303279	192.168.0.40	128.119.245.12	HTTP	539	POST /wireshark-labs/lab3-1-reply.htm HTTP/1.1	(text/plain)	
275	2024-04-25 18:48:29,458879	128.119.245.12	192.168.0.40	TCP	60	80 → 54299	[ACK] Seq=1 Ack=100399 Min=183296 Len=0	
Frame 274: 539 bytes on wire (4312 bits), 539 bytes captured (4312 bits) on interface \Device\NPF{FCB3AC3E-41ED-456B-AE9... Ethernet II, Src: GigaByteTech_d4:3a:c8 (b4:2e:99:d4:3a:c8), Dst: SagemcomBroa_7b:80:31 (b0:98:2b:7b:80:31) Internet Protocol Version 4, Src: 192.168.0.40, Dst: 128.119.245.12 Transmission Control Protocol, Src Port: 54299, Dst Port: 80, Seq: 152471, Ack: 1, Len: 485 Source Port: 54299 Destination Port: 80 [Stream index: 8] [Conversation completeness: Incomplete, DATA (15)] [TCP Segment Len: 485] Sequence Number: 152471 (relative sequence number) Sequence Number (raw): 3544030476 [Next Sequence Number: 152956 (relative sequence number)] Acknowledgment Number: 1 (relative ack number) Acknowledgment number (raw): 1286932597 0101 .... = Header Length: 20 bytes (5) Flags: 0x018 (PSH, ACK) Window: 1026 [Calculated window size: 262656] [Window size scaling factor: 256] Checksum: 0x3854 [unverified] [Checksum Status: Unverified] Urgent Pointer: 0 [Timestamps] [SEQ/ACK analysis] TCP payload (485 bytes) TCP segment data (485 bytes) [108 Reassembled TCP Segments (152955 bytes): #85(1460), #86(1460), #87(1460), #88(1460), #89(1460), #90(1460), #91(1460)] Hypertext Transfer Protocol POST /wireshark-labs/lab3-1-reply.htm HTTP/1.1\r\n Host: gaia.cs.umass.edu\r\n Connection: keep-alive\r\n Content-Length: 152321\r\n Cache-Control: max-age=0\r\n Upgrade-Insecure-Requests: 1\r\n User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/124.0.0.0 Safari/53... Origin: null\r\n Content-Type: multipart/form-data; boundary=---WebKitFormBoundarypTNcrkMdaS6Yp1Kf\r\n Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/si... Accept-Encoding: gzip, deflate\r\n Accept-Language: es-US,es-419;q=0.9,es;q=0.8,en;q=0.7\r\n \r\n [Full request URI: http://gaia.cs.umass.edu/wireshark-labs/lab3-1-reply.htm] [HTTP request 1/1] [Response in frame: 291] File Data: 152321 bytes MIME Multipart Media Encapsulation, Type: multipart/form-data, Boundary: "---WebKitFormBoundarypTNcrkMdaS6Yp1Kf" [Type: multipart/form-data] First boundary: ---WebKitFormBoundarypTNcrkMdaS6Yp1Kf\r\n Encapsulated multipart part: (text/plain) Last boundary: \r\n---WebKitFormBoundarypTNcrkMdaS6Yp1Kf--\r\n								

Figura 1: Paquete 274

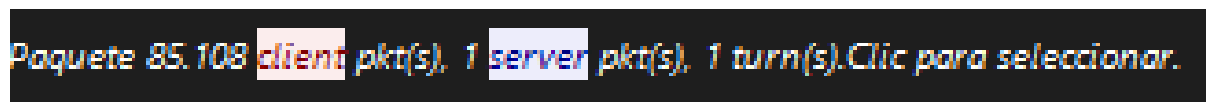


Figura 2: Búsqueda del primer segmento TCP

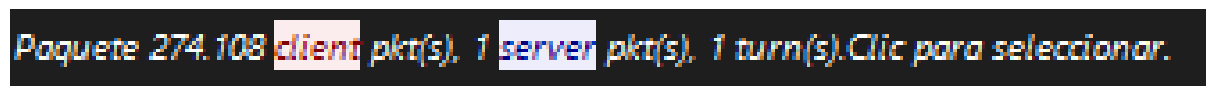


Figura 3: Búsqueda del último segmento TCP

Se buscó el paquete 85 y se pudo confirmar que efectivamente contenía el mensaje POST:

```
POST /wire  
shark-labs/lab3-  
1-reply.htm HTTP  
/1.1 Host: gaia  
.cs.umas.edu.C  
onnection: keep-  
alive Content-L  
ength: 152321 C  
ache-Control: ma  
x-age=0 Upgrade  
-Insecure-Request  
s: 1 User-Agen  
t: Mozilla/5.0 (Windows  
NT 10.0; Win64; x64) App  
leWebKit/537.36  
(KHTML, like Gec  
ko) Chrome/124.0  
.0.0 Safari/537.  
36 Origin: null  
Content-Type:  
multipart/form-d  
ata; boundary=--  
--WebKit FormBoun  
darypTNc rkMdaS6Y  
p1Kf Accept: te  
xt/html, applicat  
ion/xhtml+xml, ap
```

Figura 4: Búsqueda del mensaje POST

2) Como se verificó, no está todo en un segmento TCP sino que por lo visto en la Figura 5 está dividido en 108 segmentos TCP.

[108 Reassembled TCP Segments]

Figura 5: Segmentación de paquetes TCP

3) Por lo visto anteriormente en la Figura 4, el comienzo del mensaje POST se encuentra en el paquete 85.

4) El segundo segmento del paquete POST se encuentra en el 274 por lo visto en la Figura 3.

5) Sabiendo que el acuerdo de 3 fases se basa en un [SYN] enviado por el cliente, un [SYN-ACK] enviado por el servidor y un [ACK] como respuesta del cliente, se buscaron esos 3 segmentos.

```

66 54300 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
66 80 → 54299 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_P..
54 54299 → 80 [ACK] Seq=1 Ack=1 Win=262656 Len=0

```

Figura 6: Búsqueda del acuerdo de 3 fases

Estos son los paquetes 75, 77 y 78. Y su largo de *bytes* es

$$(66 + 66 + 54)[bytes] = 186[bytes] \quad (1)$$

6) Por lo visto en la Figura 6, el número de secuencia del primer segmento TCP de petición de conexión es 0.

7) En hexadecimal, los bits de flag que están seteados en el segmento (SYN,ACK) es el 0x012.

```

Flags: 0x012 (SYN, ACK)
Window: 29200

```

Figura 7: Flag (SYN,ACK)

8) Por lo visto en la Figura 7, en el segmento (SYN,ACK), el número de ACK es Ack=1. Esto significa que el ACK enviado por el servidor es 1. Esto indica que el servidor ha recibido el segmento SYN del cliente y está confirmando la recepción de ese segmento mediante el envío de un ACK con un número de secuencia incrementado en 1.

9) Se observó la información del paquete 85 y se obtuvo:

```

Window: 1026
[Calculated window size: 262656]
[window size scaling factor: 256]

```

Figura 8: Ventana de recepción

Como se verá en el punto siguiente, el tamaño del campo de ventana de recepción es de 16 bits, si se quiere representar un numero mas grande se utiliza el *window size scaling factor* que nos dice que tenemos que multiplicar *window* por 256 para tener un *calculated window size* de 262654.

a) El tamaño del campo de ventana de recepción en un segmento TCP es de 16 bits.

TCP Header				
Bits	0-15			16-31
0	Source port			Destination port
32	Sequence number			
64	Acknowledgment number			
96	Offset	Reserved	Flags	Window size
128	Checksum			Urgent pointer
160	Options			

Figura 9: Esquema del TCP Header

b) El campo de ventana de recepción en el encabezado TCP tiene una longitud de 16 bits, lo que significa que puede representar un máximo de

$$2^{16} - 1 = 65535 \text{ bytes} \quad (2)$$

Para calcular la máxima cantidad de KB que esto representa, se dividió por 1024 ya que 1 KB son 1024 bytes.

$$\frac{65535 \text{ bytes}}{1024} \approx 64 \text{ KB} \quad (3)$$

c) Siendo que el buffer de entrada son aproximadamente 256 KB, corresponden aproximadamente a 4 ventanas.

d) Corresponden aproximadamente a 256 KB.

10) Retomando lo que se vio en el punto 5, y sabiendo que el (SYN) lo manda el cliente y el (SYN,ACK) lo manda el servidor, tenemos que:

Source	Destination	Protocol	Length	Info
192.168.0.40	128.119.245.12	TCP	66	54299 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
128.119.245.12	192.168.0.40	TCP	66	80 → 54299 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM WS=128
192.168.0.40	128.119.245.12	TCP	54	54299 → 80 [ACK] Seq=1 Ack=1 Win=262656 Len=0

Figura 10: IP del cliente e IP del servidor

Y para determinar el puerto de origen se tuvo que:

```

Frame 85: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits) on interface \Device\NPF_{F
Ethernet II, Src: GigaByteTech_d4:3a:c8 (b4:2e:99:d4:3a:c8), Dst: SagemcomBroa_7b:80:31 (b0:98:2b:7b:80
Internet Protocol Version 4, Src: 192.168.0.40, Dst: 128.119.245.12
Transmission Control Protocol, Src Port: 54299, Dst Port: 80, Seq: 1, Ack: 1, Len: 1460
Source Port: 54299
Destination Port: 80

```

Figura 11: Puerto de origen

Por lo tanto, la IP del cliente es 192.168.0.40 y se usó el puerto 54299.

11) Se determinó que la IP del servidor es 128.119.245.12 por la Figura 10. Para confirmar esto se usó la CMD de *Windows* con el comando *nslookup*

```

PS C:\Users\Maxi> nslookup gaia.cs.umass.edu
Servidor: dns.google
Address: 2001:4860:4860::8888

Respuesta no autoritativa:
Nombre: gaia.cs.umass.edu
Address: 128.119.245.12

PS C:\Users\Maxi> |

```

Figura 12: Confirmación de IP

Y se verificó el puerto usado:

```

▶ Frame 85: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits) on interface \Device\NPF_{F
▶ Ethernet II, Src: GigaByteTech_d4:3a:c8 (b4:2e:99:d4:3a:c8), Dst: SagemcomBroa_7b:80:31 (b0:98:2b:7b:80
▶ Internet Protocol Version 4, Src: 192.168.0.40, Dst: 128.119.245.12
▶ Transmission Control Protocol, Src Port: 54299, Dst Port: 80, Seq: 1, Ack: 1, Len: 1460
  Source Port: 54299
  Destination Port: 80

```

Figura 13: Puerto de destino

12) Para conocer el número de secuencia del segmento TCP que contiene el encabezado del comando HTTP POST se buscó la información de este en el *Wireshark*.

```

[Completeness Flags: ..DASS]
[TCP Segment Len: 485]
Sequence Number: 152471    (relative sequence number)
Sequence Number (raw): 3544030476

```

Figura 14: Número de secuencia

Se determinó que el número era 152471 y que la cantidad de bytes es la diferencia del largo del segmento con el largo del *TCP Header*.

Por lo tanto, hay 465 bytes en el campo de carga útil.

13) El primer segmento (85) se envió a las 18:48:28,802310 y el primer ACK de este primer segmento (111) se recibió a las 18:48:28,966061

85	2024-04-25	18:48:28,802310	192.168.0.40	128.119.245.12	TCP	1514	54299 → 80	[ACK]	Seq=1	Ack=1	Win=26265
86	2024-04-25	18:48:28,802310	192.168.0.40	128.119.245.12	TCP	1514	54299 → 80	[ACK]	Seq=1461	Ack=1	Win=26
87	2024-04-25	18:48:28,802310	192.168.0.40	128.119.245.12	TCP	1514	54299 → 80	[ACK]	Seq=2921	Ack=1	Win=26
88	2024-04-25	18:48:28,802310	192.168.0.40	128.119.245.12	TCP	1514	54299 → 80	[ACK]	Seq=4381	Ack=1	Win=26
89	2024-04-25	18:48:28,802310	192.168.0.40	128.119.245.12	TCP	1514	54299 → 80	[ACK]	Seq=5841	Ack=1	Win=26
90	2024-04-25	18:48:28,802310	192.168.0.40	128.119.245.12	TCP	1514	54299 → 80	[ACK]	Seq=7301	Ack=1	Win=26
91	2024-04-25	18:48:28,802310	192.168.0.40	128.119.245.12	TCP	1514	54299 → 80	[ACK]	Seq=8761	Ack=1	Win=26
92	2024-04-25	18:48:28,802310	192.168.0.40	128.119.245.12	TCP	1514	54299 → 80	[ACK]	Seq=10221	Ack=1	Win=2
93	2024-04-25	18:48:28,802310	192.168.0.40	128.119.245.12	TCP	1514	54299 → 80	[ACK]	Seq=11681	Ack=1	Win=2
94	2024-04-25	18:48:28,802310	192.168.0.40	128.119.245.12	TCP	1514	54299 → 80	[ACK]	Seq=13141	Ack=1	Win=2
111	2024-04-25	18:48:28,966061	128.119.245.12	192.168.0.40	TCP	60	80 → 54299	[ACK]	Seq=1	Ack=1461	Win=32

Figura 15: Horario de envío

14) Viendo la Figura 15, los primeros 4 paquetes tienen un tamaño de 1514 bytes. Si desestimamos 14 bytes del *Header* de Ethernet, tenemos 1500. Sabemos que 1460 son carga útil de TCP, 20 son del *TCP Header* y otros 20 son del *Header* del protocolo IP. Por lo tanto, si consideramos solo el encabezado TCP, nos quedaría 1480 bytes de carga útil más el encabezado.

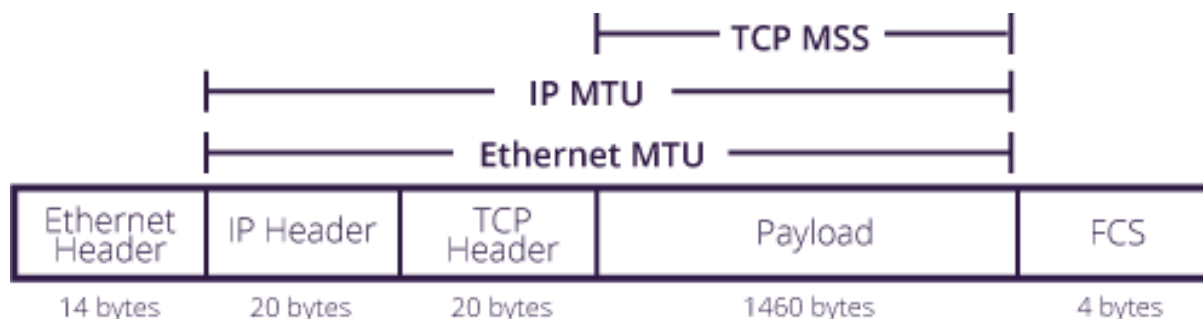


Figura 16: Horario de envío

## 2.2. Datos enviados por unidad de tiempo

Para esta parte del experimento, se seleccionó un segmento TCP enviado por el cliente correspondiente a la transferencia. Luego, en el apartado de Estadísticas, se realizó el siguiente gráfico de secuencias TCP.

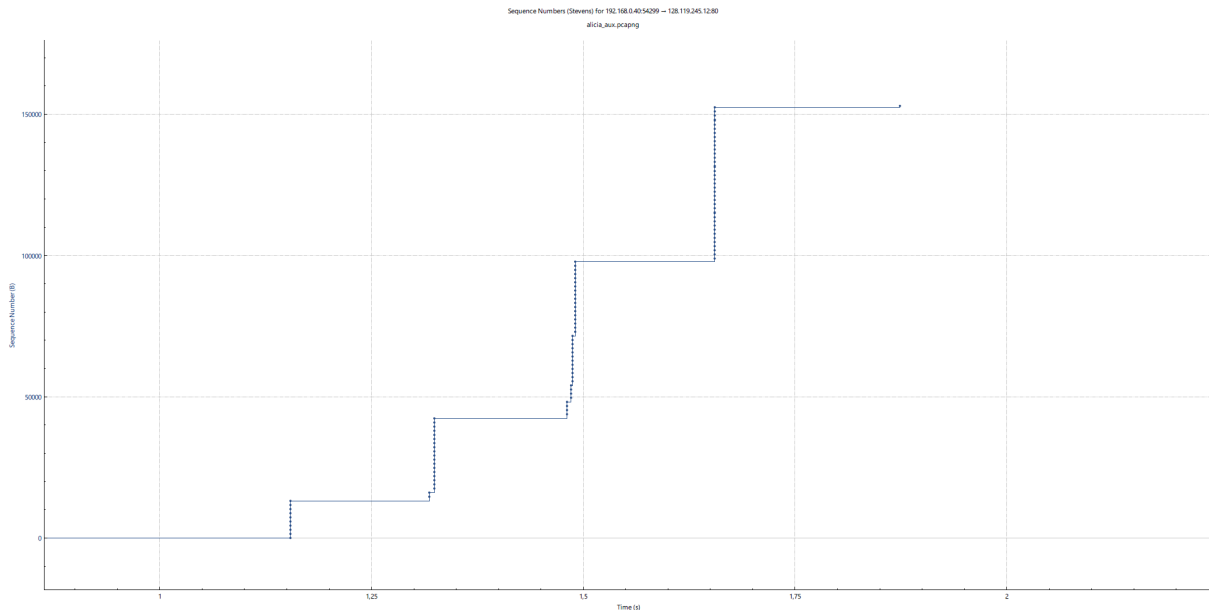


Figura 17: Secuencia de segmento vs Tiempo de envío

Vemos que el protocolo de congestión usado es *Slow Start*.

En *Slow Start* se envían una serie de paquetes, si el receptor manda la misma cantidad de ACK's, se mandan el doble de paquetes en la tanda siguiente.

Por cada ACK recibido, se incrementa cwnd (congestion window), siempre y cuando cwnd no sea mayor a slow start threshold (umbral de comienzo suave).

Vemos que este es nuestro caso, ya que envía el doble de paquetes cada periodo T que es lo que tarda en responder el receptor, excepto al final que entrega los paquetes restantes.

### 3. Ejercicio 2

Se realizó la siguiente simulación en *Imunes*.

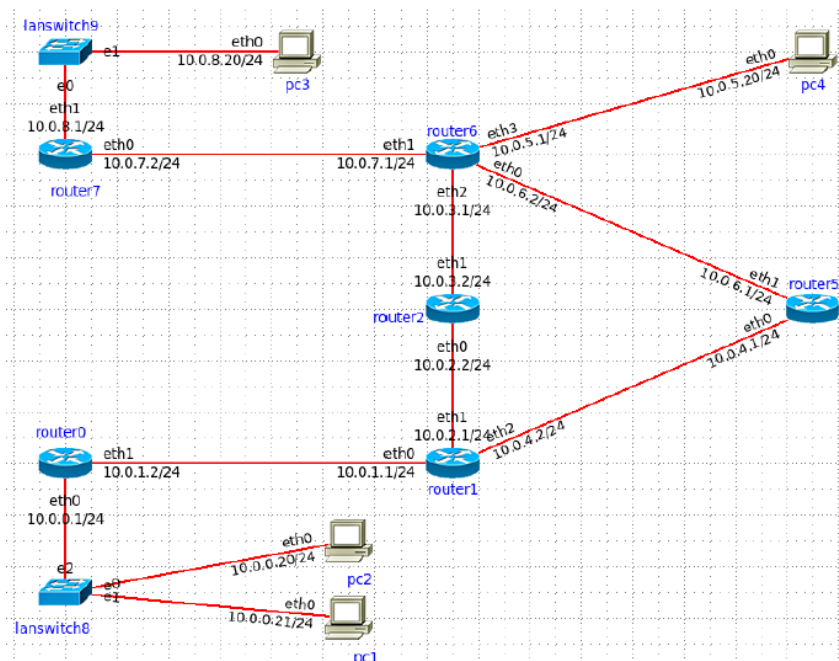


Figura 18: Topología



### 3.1. Conectividad. Uso de ping

1.a) Al ejecutar *ping* solamente indicando la IP de PC3, se obtuvo lo siguiente.

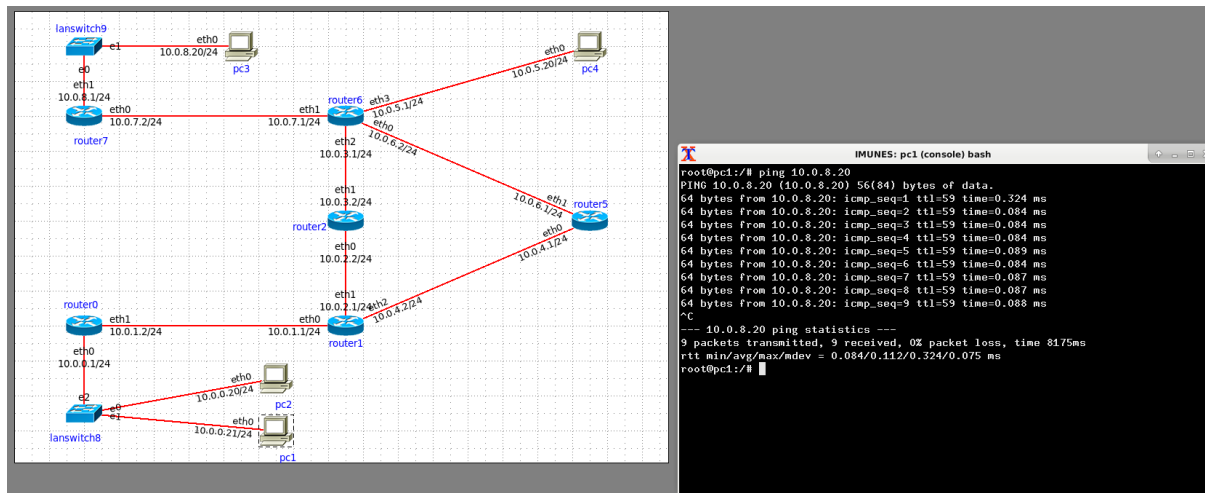


Figura 19: Ping de PC1 a PC3

Con lo cual verificamos la conexión entre ellos.

b) Al ejecutar el comando ping de manera de enviar un datagrama IP al destino con 1400 bytes de longitud total y sin permitir que se realice fragmentación, se obtuvo lo siguiente.

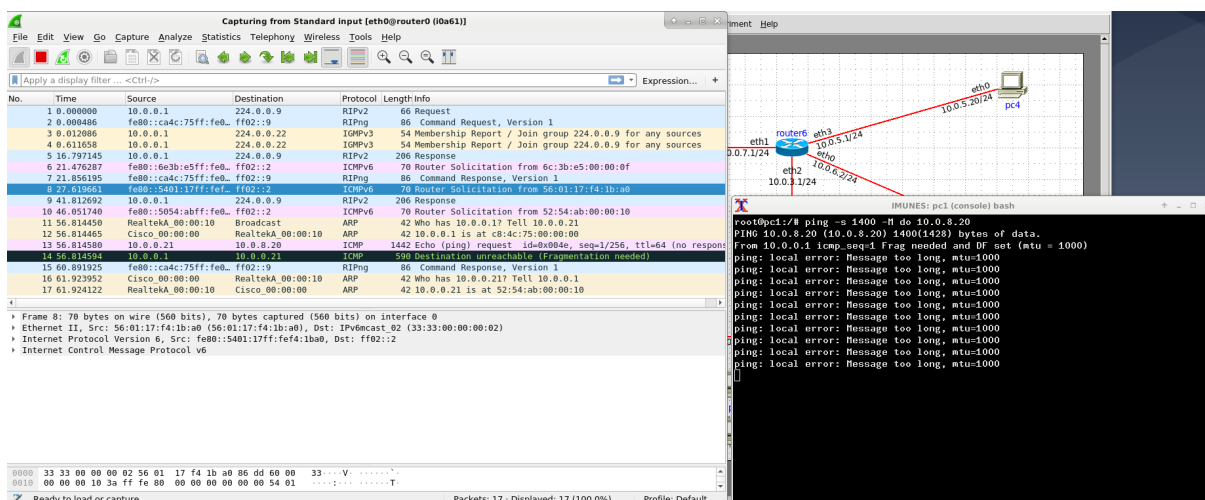
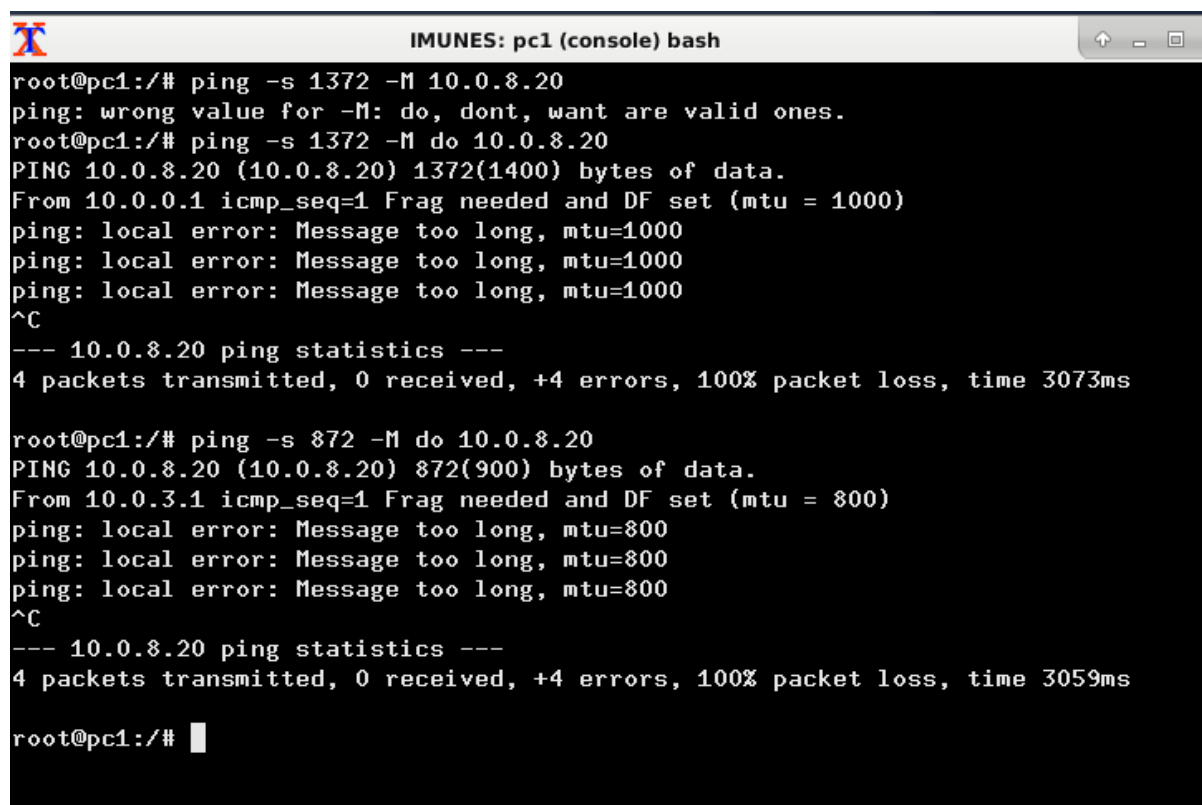


Figura 20: Estación de trabajo



```

IMUNES: pc1 (console) bash
root@pc1:/# ping -s 1372 -M 10.0.8.20
ping: wrong value for -M: do, dont, want are valid ones.
root@pc1:/# ping -s 1372 -M do 10.0.8.20
PING 10.0.8.20 (10.0.8.20) 1372(1400) bytes of data.
From 10.0.0.1 icmp_seq=1 Frag needed and DF set (mtu = 1000)
ping: local error: Message too long, mtu=1000
ping: local error: Message too long, mtu=1000
ping: local error: Message too long, mtu=1000
^C
--- 10.0.8.20 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 3073ms

root@pc1:/# ping -s 872 -M do 10.0.8.20
PING 10.0.8.20 (10.0.8.20) 872(900) bytes of data.
From 10.0.3.1 icmp_seq=1 Frag needed and DF set (mtu = 800)
ping: local error: Message too long, mtu=800
ping: local error: Message too long, mtu=800
ping: local error: Message too long, mtu=800
^C
--- 10.0.8.20 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 3059ms

root@pc1:/#

```

Figura 21: Captura en Imunes

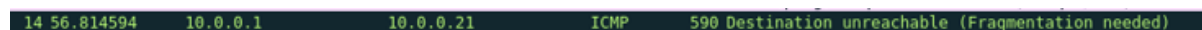
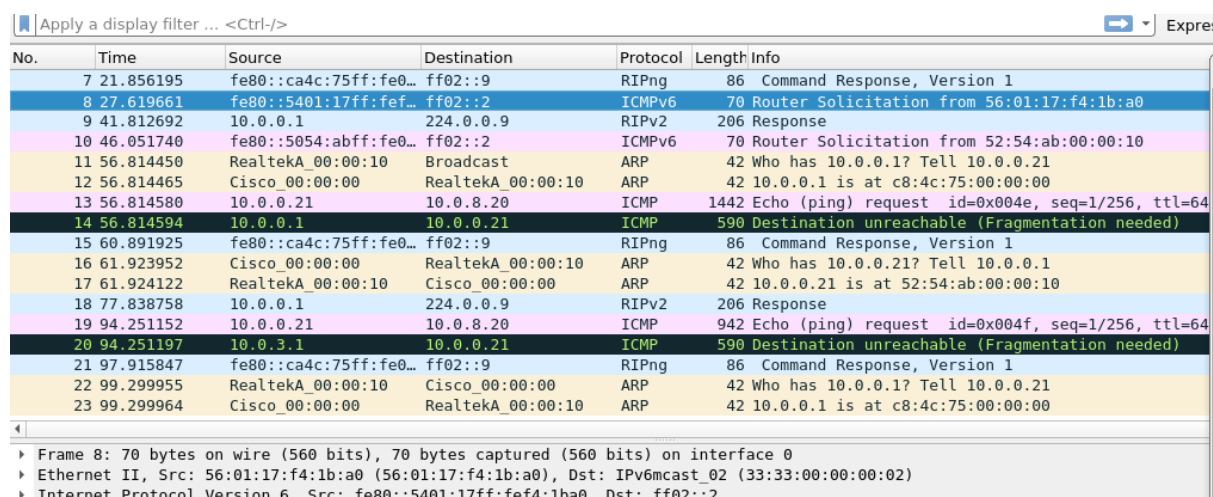


Figura 22: Captura en Wireshark

Se determinó que el error en la conexión se debía a que se necesita una fragmentación ya que la unidad máxima de transferencia estaba en 1000 en el router con la IP 10.0.0.1.

c) Lo mismo sucedió cuando se quiso enviar un datagrama IP al destino con 900 bytes pero en este caso con el router de IP 10.0.3.1 con un MTU de 800. Vemos que en la segunda parte de la Figura 21 está el comando utilizado, obteniendo el error de *Message too long, mtu = 800*.



No.	Time	Source	Destination	Protocol	Length	Info
7	21.856195	fe80::ca4c:75ff:fe0... ff02::9		RIPng	86	Command Response, Version 1
8	27.619661	fe80::5401:17ff:fef... ff02::2		ICMPv6	70	Router Solicitation from 56:01:17:f4:1b:a0
9	41.812692	10.0.0.1	224.0.0.9	RIPv2	206	Response
10	46.051740	fe80::5054:abff:fe0... ff02::2		ICMPv6	70	Router Solicitation from 52:54:ab:00:00:10
11	56.814450	RealtekA_00:00:10	Broadcast	ARP	42	Who has 10.0.0.1? Tell 10.0.0.21
12	56.814465	Cisco_00:00:00	RealtekA_00:00:10	ARP	42	10.0.0.1 is at c8:4c:75:00:00:00
13	56.814580	10.0.0.21	10.0.8.20	ICMP	1442	Echo (ping) request id=0x004e, seq=1/256, ttl=64
14	56.814594	10.0.0.1	10.0.0.21	ICMP	590	Destination unreachable (Fragmentation needed)
15	60.891925	fe80::ca4c:75ff:fe0... ff02::9		RIPng	86	Command Response, Version 1
16	61.923952	Cisco_00:00:00	RealtekA_00:00:10	ARP	42	Who has 10.0.0.21? Tell 10.0.0.1
17	61.924122	RealtekA_00:00:10	Cisco_00:00:00	ARP	42	10.0.0.21 is at 52:54:ab:00:00:10
18	77.838758	10.0.0.1	224.0.0.9	RIPv2	206	Response
19	94.251152	10.0.0.21	10.0.8.20	ICMP	942	Echo (ping) request id=0x004f, seq=1/256, ttl=64
20	94.251197	10.0.3.1	10.0.0.21	ICMP	590	Destination unreachable (Fragmentation needed)
21	97.915847	fe80::ca4c:75ff:fe0... ff02::9		RIPng	86	Command Response, Version 1
22	99.299955	RealtekA_00:00:10	Cisco_00:00:00	ARP	42	Who has 10.0.0.1? Tell 10.0.0.21
23	99.299964	Cisco_00:00:00	RealtekA_00:00:10	ARP	42	10.0.0.1 is at c8:4c:75:00:00:00

▶ Frame 8: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on interface 0  
 ▶ Ethernet II, Src: 56:01:17:f4:1b:a0 (56:01:17:f4:1b:a0), Dst: IPv6mcast\_02 (33:33:00:00:00:02)  
 ▶ Internet Protocol Version 6, Src: fe80::5401:17ff:fef4:1ba0, Dst: ff02::2

Figura 23: Captura en Wireshark

d) Para el último caso, vemos que si se envían los paquetes entre PC1 y PC3 al querer enviar datagramas IP al destino con 600 bytes de longitud sin fragmentación.

```

IMUNES: pc1 (console) bash
root@pc1:/# ping -s 572 -M do 10.0.8.20
PING 10.0.8.20 (10.0.8.20) 572(600) bytes of data.
580 bytes from 10.0.8.20: icmp_seq=1 ttl=59 time=0.393 ms
580 bytes from 10.0.8.20: icmp_seq=2 ttl=59 time=0.109 ms
580 bytes from 10.0.8.20: icmp_seq=3 ttl=59 time=0.090 ms
580 bytes from 10.0.8.20: icmp_seq=4 ttl=59 time=0.090 ms
580 bytes from 10.0.8.20: icmp_seq=5 ttl=59 time=0.087 ms
^C
--- 10.0.8.20 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4081ms
rtt min/avg/max/mdev = 0.087/0.153/0.393/0.120 ms
root@pc1:/#

```

Figura 24: Captura en Imunes

2) Para permitir que PC2 pueda enviar datagrams de hasta 1200 bytes de longitud al Router 5 sin necesidad de fragmentacion se modificó el *MTU* de los routers involucrados a 1200.

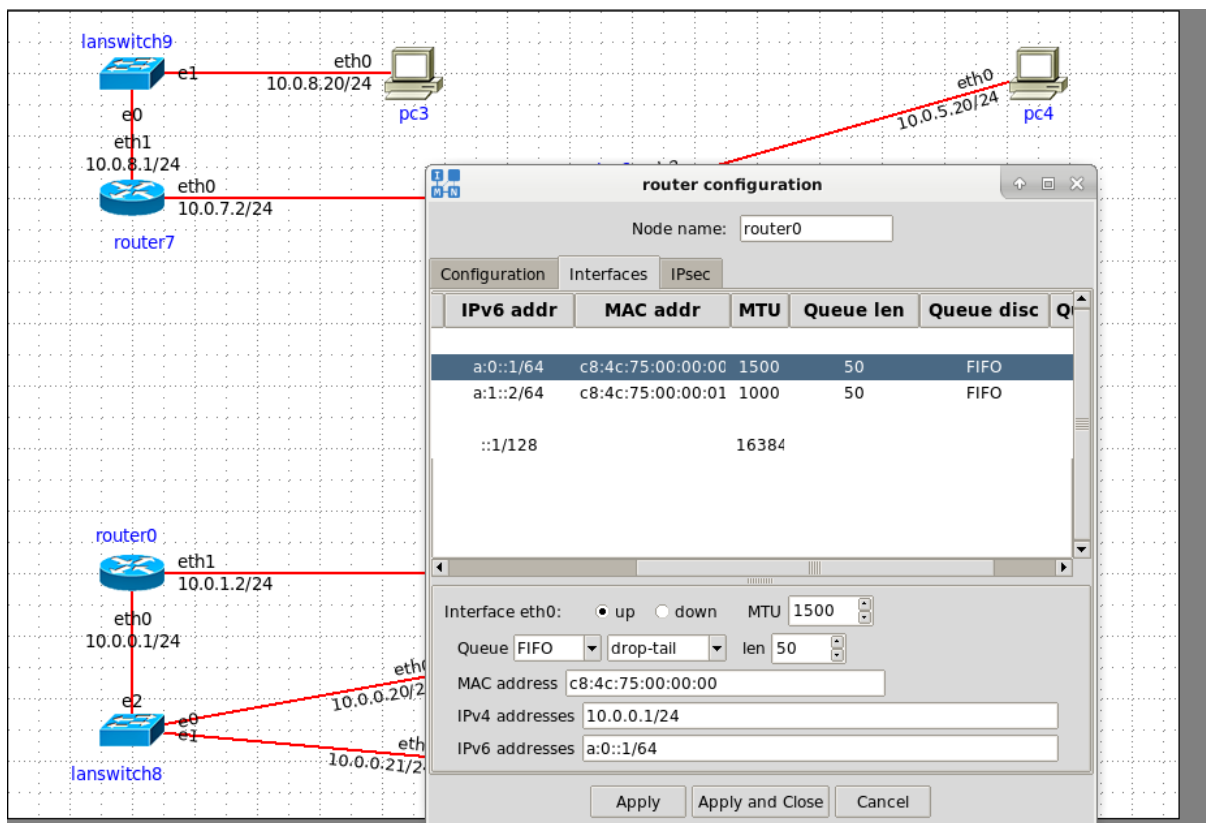


Figura 25: MTU afectado

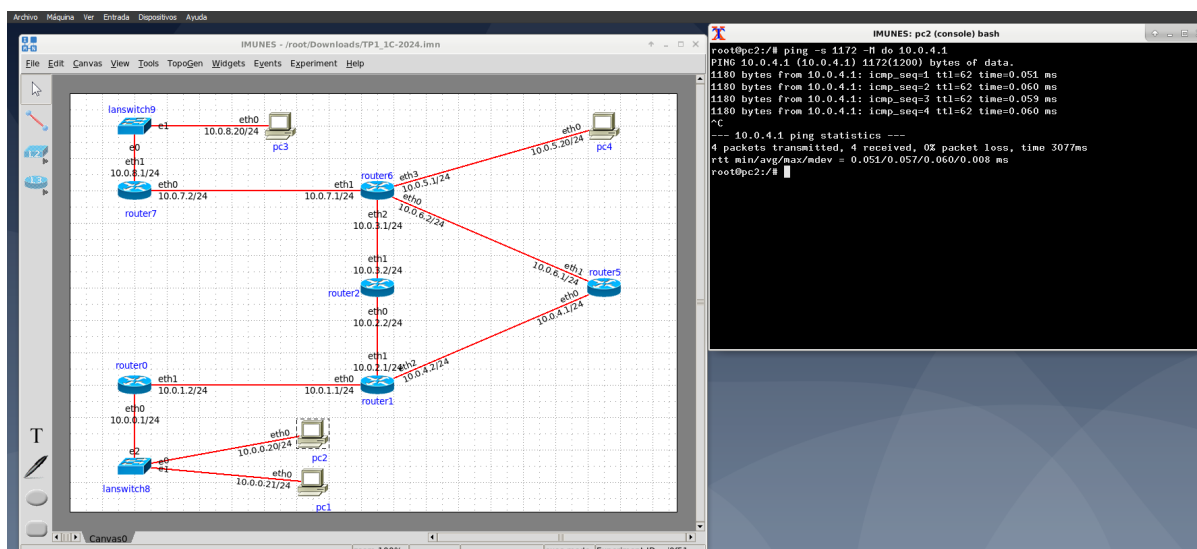


Figura 26: Verificación de conexión

### 3.2. Relevamiento de rutas. Uso de traceroute

- 1) Tomando de origen PC2 y  $Destino_1 = PC4$  y  $Destino_2 = PC3$ , se obtuvo la siguiente imagen.

```

IMUNES: pc2 (console) bash
root@pc2:/# traceroute 10.0.5.20
traceroute to 10.0.5.20 (10.0.5.20), 30 hops max, 60 byte packets
 1 10.0.0.1 (10.0.0.1)  1.897 ms  2.390 ms  2.391 ms
 2 10.0.1.1 (10.0.1.1)  2.393 ms  2.389 ms  2.387 ms
 3 10.0.2.2 (10.0.2.2)  2.333 ms  2.327 ms  2.324 ms
 4 10.0.3.1 (10.0.3.1)  2.324 ms  2.321 ms  2.316 ms
 5 10.0.5.20 (10.0.5.20)  2.321 ms  2.319 ms  2.316 ms
root@pc2:/# traceroute 10.0.8.20
traceroute to 10.0.8.20 (10.0.8.20), 30 hops max, 60 byte packets
 1 10.0.0.1 (10.0.0.1)  0.042 ms  0.009 ms  0.007 ms
 2 10.0.1.1 (10.0.1.1)  0.014 ms  0.009 ms  0.009 ms
 3 10.0.4.1 (10.0.4.1)  0.019 ms  0.012 ms  0.011 ms
 4 10.0.3.1 (10.0.3.1)  0.021 ms  0.014 ms  0.012 ms
 5 10.0.7.2 (10.0.7.2)  0.021 ms  0.015 ms  0.015 ms
 6 10.0.8.20 (10.0.8.20)  1.175 ms  0.813 ms  0.816 ms
root@pc2:/#

```

Figura 27: Uso de *traceroute*

La topología de la figura anterior nos muestra que en la ruta que tiene como destino a PC4 no hay saltos, sin embargo, en la ruta que tiene como destino a PC3, hay un salto, ya que en el ítem 4 tendría que ir del 10.0.4.1 al 10.0.6.2, pero va al 10.0.3.1. Esto pasa ya que el *Router*<sub>6</sub> tiene como IP estática a 10.0.0.0/24 10.0.3.2, entonces solo recibe de la rama de "abajo". Esto se puede solucionar haciendo que el *Router*<sub>1</sub> apunte solo al *Router*<sub>2</sub>, inutilizando la rama del *Router*<sub>5</sub> como se ve en la siguiente figura.

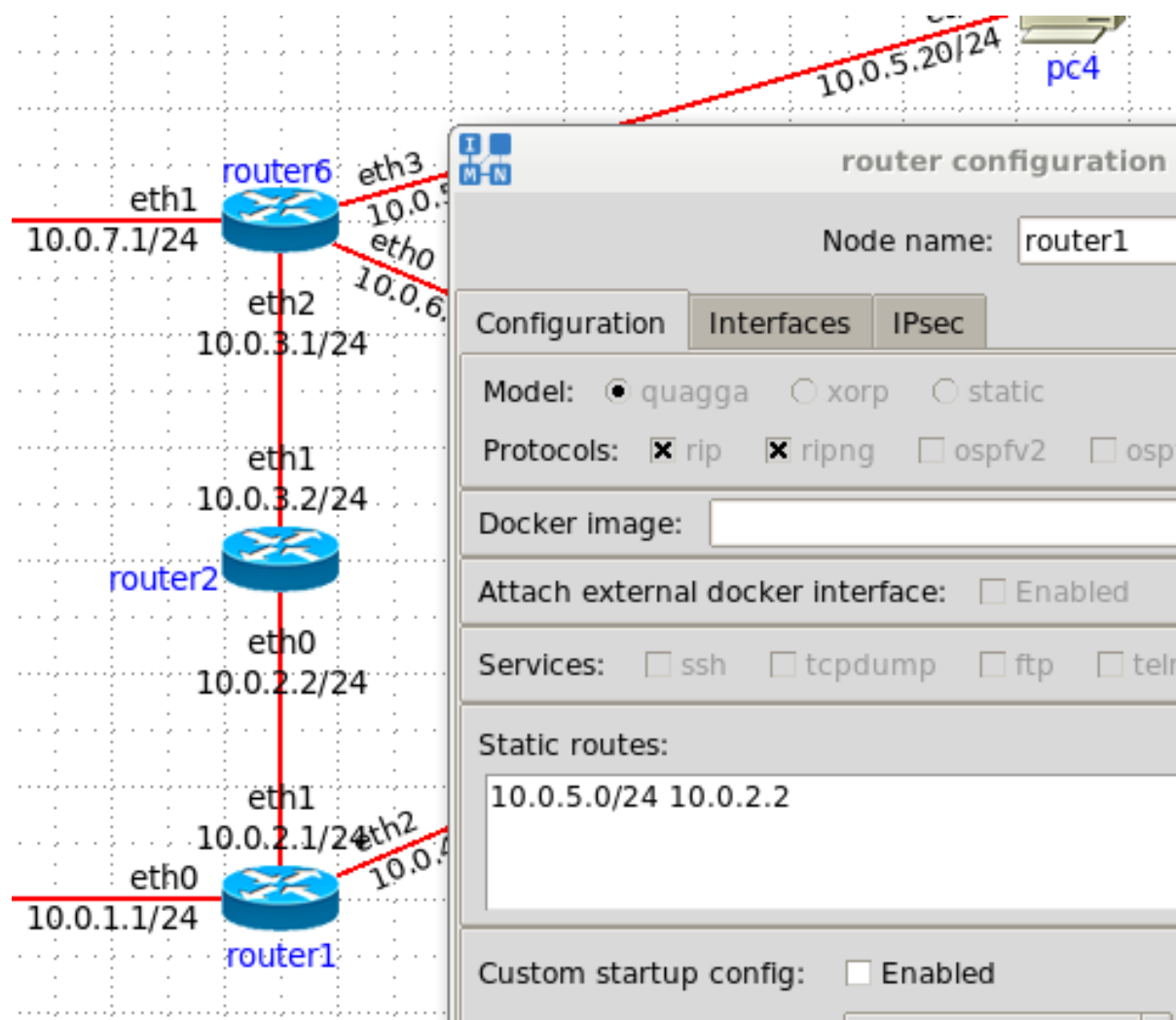


Figura 28: Cambio de IP estática

```

root@pc2:/# traceroute 10.0.8.20
traceroute to 10.0.8.20 (10.0.8.20), 30 hops max, 60 byte packets
 1  10.0.0.1 (10.0.0.1)  0.771 ms  1.048 ms  1.090 ms
 2  10.0.1.1 (10.0.1.1)  1.128 ms  1.202 ms  1.236 ms
 3  10.0.2.2 (10.0.2.2)  1.556 ms  1.644 ms  1.679 ms
 4  10.0.3.1 (10.0.3.1)  1.716 ms  1.797 ms  1.830 ms
 5  10.0.7.2 (10.0.7.2)  1.866 ms  1.934 ms  1.967 ms
 6  10.0.8.20 (10.0.8.20)  2.533 ms  2.196 ms  2.416 ms
root@pc2:/#

```

Figura 29: Traceroute de la ruta modificada

Por lo visto en la Figura 29, se confirmó que ya no hay saltos, ya que pasa del 1.0.2.2 al 1.0.3.1.