

Computación Centrada en Redes

IP, forwarding y NAT

Redes locales e IP

Gerson Garrido

Semana 4

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

Sesión A: IP y routers

1. Forwarding y routing
2. El datagrama IPv4
3. Direcciones y CIDR
4. Longest prefix match
5. Dentro de un router

Sesión B: lo que hace usable a IP

6. ICMP y `traceroute`
7. DHCP
8. NAT
9. IPv6
10. Recorrido completo de un paquete

> **Lectura**
>
> Kurose y Ross, secs. 4.1, 4.2, 4.3.1 y CIDR de 4.3.2 (sesión A); 4.3.2 (DHCP), 4.3.3, 4.3.4 y 5.6 (sesión B). CS168: *IP Header*, *Addressing*, *Router Hardware*, *DHCP* y *NAT*. Complementarias: RFC 791, 4632, 792, 2131, 1918 y 8200.

SECCIÓN 01

Forwarding y routing

01



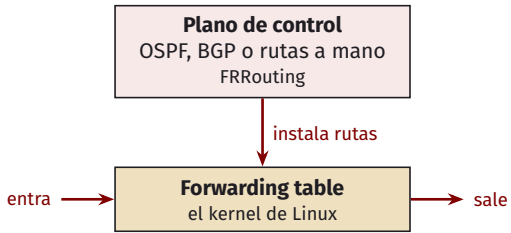
Dos trabajos, dos planos

	Forwarding	**Routing**
Pregunta	¿Por cuál interfaz sale este paquete?	¿Qué caminos hay hacia cada destino?
Alcance	Un router	Toda la red
Frecuencia	Cada paquete	Cuando algo cambia
Tiempo	Nanosegundos	Milisegundos a minutos
Se hace en	Hardware o kernel	Software
Plano	Plano de datos	Plano de control

Como un viaje por carretera

Forwarding es doblar en cada cruce mirando el letrero. Routing es planear la ruta con el mapa antes de salir.

La forwarding table los une



Ojo

Un router sigue haciendo forwarding con el protocolo de routing caído. Y con la tabla mal, el protocolo puede estar perfecto y los paquetes se pierden igual.

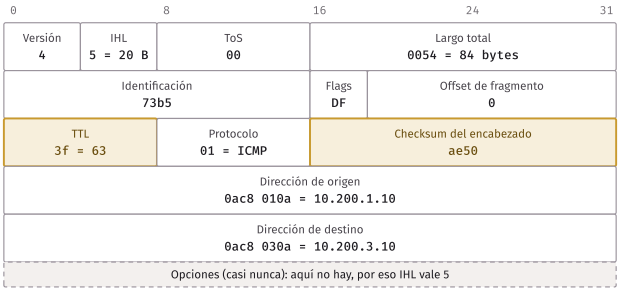
Esta semana el plano de control somos nosotros: rutas a mano. Las semanas 5 y 6 lo reemplazan por protocolos.

SECCIÓN 02

El datagrama IPv4

02

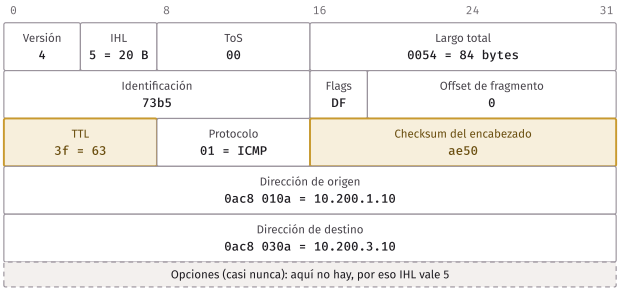
El encabezado: 20 bytes sin opciones



Al cruzar r1: `TTL 40 → 3f` `checksum ad50 → ae50` y nada más

- **TTL:** cada router le resta 1. En 0, se descarta y se avisa con ICMP. Evita las vueltas infinitas.
- **Protocolo:** a quién se le entregan los datos. 1 ICMP, 6 TCP, 17 UDP.
- **Checksum:** solo del encabezado.

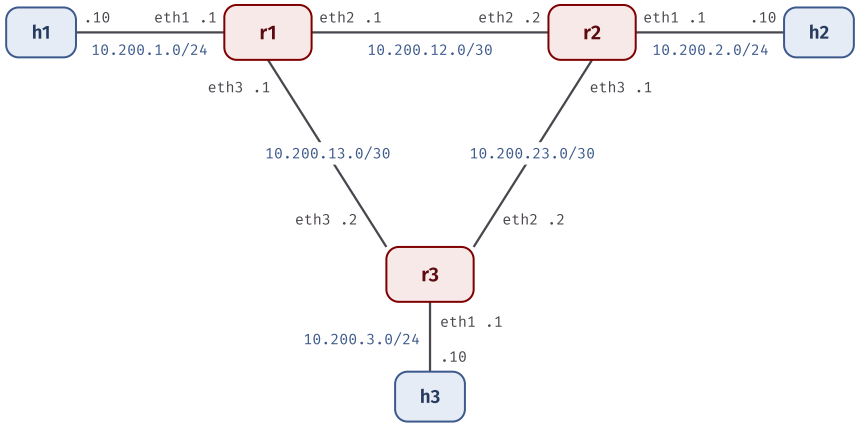
El encabezado: 20 bytes sin opciones



Al cruzar r1: `TTL 40 → 3f` `checksum ad50 → ae50` y nada más

- **TTL:** cada router le resta 1. En 0, se descarta y se avisa con ICMP. Evita las vueltas infinitas.
- **Protocolo:** a quién se le entregan los datos. 1 ICMP, 6 TCP, 17 UDP.
- **Checksum:** solo del encabezado.
- Lo único que cambia en un router normal: TTL y checksum.

La topología de la Sesión A



El mismo datagrama, a cada lado de `r1`

```
tcpdump -n -x, en r1

[r1.eth1]
06:45:44.672717 IP 10.200.1.10 > 10.200.3.10: ICMP echo request, id 153, seq 0, length 64
        0x0000:  4500 0054 73b5 4000 4001 ad50 0ac8 010a
        0x0010:  0ac8 030a 0800 5973 0099 0000 e30f bae3
[r1.eth3]
06:45:44.672729 IP 10.200.1.10 > 10.200.3.10: ICMP echo request, id 153, seq 0, length 64
        0x0000:  4500 0054 73b5 4000 3f01 ae50 0ac8 010a
        0x0010:  0ac8 030a 0800 5973 0099 0000 e30f bae3
```

- TTL: `40` (64) pasa a `3f` (63).
- Checksum: `ad50` pasa a `ae50`.
- `0ac8 010a` son 10, 200, 1 y 10.

Adivinen

¿Por qué el checksum subió justo `0100`?

El checksum, a mano

suma de 16 bits

```
  4500
+ 0054 + 73b5
+ 4000 + 4001
+ 0000   (checksum en 0)
+ 0ac8 + 010a
+ 0ac8 + 030a
-------------
 152ae
  52ae + 1 = 52af
~52af = ad50
```

- Suma de complemento a uno, y complemento al final.

El checksum, a mano

suma de 16 bits

```
  4500
+ 0054 + 73b5
+ 4000 + 4001
+ 0000   (checksum en 0)
+ 0ac8 + 010a
+ 0ac8 + 030a
-------------
 152ae
  52ae + 1 = 52af
~52af = ad50
```

- Suma de complemento a uno, y complemento al final.
- El que recibe suma todo, con el checksum, y debe dar `ffff`.

El checksum, a mano

suma de 16 bits

```
  4500
+ 0054 + 73b5
+ 4000 + 4001
+ 0000   (checksum en 0)
+ 0ac8 + 010a
+ 0ac8 + 030a
-------------
 152ae
  52ae + 1 = 52af
~52af = ad50
```

- Suma de complemento a uno, y complemento al final.
- El que recibe suma todo, con el checksum, y debe dar `ffff`.
- El TTL y el protocolo forman la palabra `4001`. Al pasar a `3f01`, la suma baja `0100`, y su complemento sube `0100`.

El checksum, a mano

suma de 16 bits

```
  4500
+ 0054 + 73b5
+ 4000 + 4001
+ 0000   (checksum en 0)
+ 0ac8 + 010a
+ 0ac8 + 030a
-------------
 152ae
  52ae + 1 = 52af
~52af = ad50
```

- Suma de complemento a uno, y complemento al final.
- El que recibe suma todo, con el checksum, y debe dar `ffff`.
- El TTL y el protocolo forman la palabra `4001`. Al pasar a `3f01`, la suma baja `0100`, y su complemento sube `0100`.
- Un router ajusta, no vuelve a sumar.

Solo el encabezado

Los datos los protegen el CRC del enlace y el checksum del transporte.

MTU 600 en `r1–r3`: si no cabe, con DF

h1, con tcpdump -n -v

```
IP (tos 0x0, ttl 64, id 51977, offset 0, flags [DF], proto ICMP (1), length 1428)
    10.200.1.10 > 10.200.3.10: ICMP echo request, id 216, seq 0, length 1408
IP (tos 0xc0, ttl 64, id 48930, offset 0, flags [none], proto ICMP (1), length 576)
    10.200.1.1 > 10.200.1.10: ICMP 10.200.3.10 unreachable - need to frag (mtu 600), length 556
```

MTU 600 en r1–r3: si no cabe, con DF

h1, con tcpdump -n -v

```
IP (tos 0x0, ttl 64, id 51977, offset 0, flags [DF], proto ICMP (1), length 1428)
    10.200.1.10 > 10.200.3.10: ICMP echo request, id 216, seq 0, length 1408
IP (tos 0xc0, ttl 64, id 48930, offset 0, flags [none], proto ICMP (1), length 576)
    10.200.1.1 > 10.200.1.10: ICMP 10.200.3.10 unreachable - need to frag (mtu 600), length 556
```

h1

```
$ ip route get 10.200.3.10
10.200.3.10 via 10.200.1.1 dev eth1 src 10.200.1.10 uid 0
    cache expires 592sec mtu 600
```

MTU 600 en `r1–r3`: si no cabe, con DF

h1, con tcpdump -n -v

```
IP (tos 0x0, ttl 64, id 51977, offset 0, flags [DF], proto ICMP (1), length 1428)
    10.200.1.10 > 10.200.3.10: ICMP echo request, id 216, seq 0, length 1408
IP (tos 0xc0, ttl 64, id 48930, offset 0, flags [none], proto ICMP (1), length 576)
    10.200.1.1 > 10.200.1.10: ICMP 10.200.3.10 unreachable - need to frag (mtu 600), length 556
```

h1

```
$ ip route get 10.200.3.10
10.200.3.10 via 10.200.1.1 dev eth1 src 10.200.1.10 uid 0
    cache expires 592sec mtu 600
```

Path MTU discovery

Linux marca DF por defecto. El origen aprende el MTU con ICMP. Al repetir el `ping`, `h1` fragmenta antes de enviar.

Sin DF: ahora sí fragmenta el router

r1 eth3: entró 1 datagrama de 1428 bytes y salen 3

```
IP (tos 0x0, ttl 63, id 19220, offset 0, flags [+], proto UDP (17), length 596)
    10.200.1.10.54364 > 10.200.3.10.9999: UDP, length 1400
IP (tos 0x0, ttl 63, id 19220, offset 576, flags [+], proto UDP (17), length 596)
    10.200.1.10 > 10.200.3.10: ip-proto-17
IP (tos 0x0, ttl 63, id 19220, offset 1152, flags [none], proto UDP (17), length 276)
    10.200.1.10 > 10.200.3.10: ip-proto-17
```

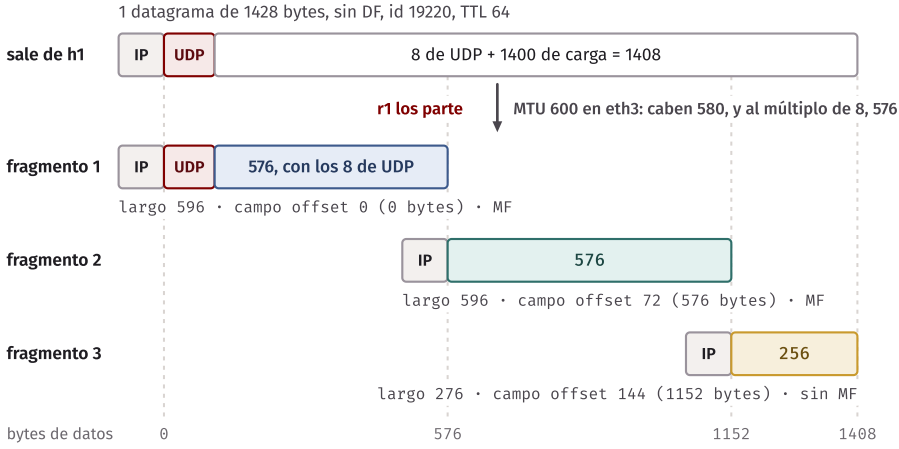
Datos	Largo total	Campo offset	MF
576	596	0	sí
576	596	72	sí
256	276	144	no

600 − 20 = 580; al múltiplo de 8: **576**. El offset cuenta unidades de 8 bytes.

Solo el destino vuelve a armarlo

Si se pierde un fragmento, se pierde el datagrama entero. Sin puertos en los últimos fragmentos, sufren firewalls y NAT.

Los tres fragmentos, a escala



SECCIÓN 03

Direcciones y CIDR

03

Prefijo y máscara

- Una dirección identifica una **interfaz**, no una máquina.
- Una **subred** *(subnet)*: interfaces que se alcanzan sin pasar por un router. Comparten el prefijo.

```
dirección AND máscara = red

10.200.1.10     00001010 11001000 00000001 00001010
255.255.255.0   11111111 11111111 11111111 00000000
10.200.1.0      00001010 11001000 00000001 00000000
```

- Con h bits de host: 2^h direcciones, $2^h - 2$ para hosts. Excepciones: `/31` (dos, RFC 3021) y `/32` (una).

Prefijo y máscara

- Una dirección identifica una **interfaz**, no una máquina.
- Una **subred** *(subnet)*: interfaces que se alcanzan sin pasar por un router. Comparten el prefijo.

```
dirección AND máscara = red

10.200.1.10     00001010 11001000 00000001 00001010
255.255.255.0   11111111 11111111 11111111 00000000
10.200.1.0      00001010 11001000 00000001 00000000
```

- Con h bits de host: 2^h direcciones, $2^h - 2$ para hosts. Excepciones: `/31` (dos, RFC 3021) y `/32` (una).
- Si el destino está en mi red, ARP por él. Si no, ARP por el router.

Prefijo y máscara

- Una dirección identifica una **interfaz**, no una máquina.
- Una **subred** *(subnet)*: interfaces que se alcanzan sin pasar por un router. Comparten el prefijo.

```
dirección AND máscara = red

10.200.1.10     00001010 11001000 00000001 00001010
255.255.255.0   11111111 11111111 11111111 00000000
10.200.1.0      00001010 11001000 00000001 00000000
```

- Con h bits de host: 2^h direcciones, $2^h - 2$ para hosts. Excepciones: `/31` (dos, RFC 3021) y `/32` (una).
- Si el destino está en mi red, ARP por él. Si no, ARP por el router.
- **CIDR**: un prefijo de cualquier largo. Acabó con las clases A, B y C.

Ejemplo: `10.20.0.0/22` en cuatro subredes

4 subredes $= 2^2$: tomamos **2 bits** de host.

`/22` pasa a `/24`.

Tercer byte:

```
000000|00 = 0
000000|01 = 1
000000|10 = 2
000000|11 = 3
```

La máscara: 252 es `11111100`. Son 16 + 6 = 22 unos.

cidr.c

```
$ ./cidr 10.20.0.0/22
prefijo    10.20.0.0/22
máscara    255.255.252.0
total      1024 direcciones
broadcast  10.20.3.255
hosts      10.20.0.1 - 10.20.3.254 (1022)
$ ./cidr 10.20.0.0/22 4
10.20.0.0/22 en 4 subredes de /24 (256 direcciones cada una):
  10.20.0.0/24       hosts 10.20.0.1 - 10.20.0.254 (254)
  10.20.1.0/24       hosts 10.20.1.1 - 10.20.1.254 (254)
  10.20.2.0/24       hosts 10.20.2.1 - 10.20.2.254 (254)
  10.20.3.0/24       hosts 10.20.3.1 - 10.20.3.254 (254)
```

Ejemplo: subredes de distinto tamaño, en `10.200.70.0/24`

Ordenar de mayor a menor, redondear a una potencia de 2 y asignar en orden.

Necesidad	Hacen falta	Bloque	Prefijo	Rango
LAN de 60	62	64	`10.200.70.0/26`	`.0` a `.63`
LAN de 28	30	32	`10.200.70.64/27`	`.64` a `.95`
LAN de 12	14	16	`10.200.70.96/28`	`.96` a `.111`
Enlace	2	4	`10.200.70.112/30`	`.112` a `.115`

Error común

`10.200.70.70/27` no es una subred: un `/27` empieza en un múltiplo de 32. Al asignar de mayor a menor, esto sale solo.

SECCIÓN 04

Longest prefix match

04



Agregar, y la excepción

Cuatro redes por la misma interfaz, una sola entrada:

```
10.200.0.0/24   000000|00
10.200.1.0/24   000000|01
10.200.2.0/24   000000|10
10.200.3.0/24   000000|11
```

`resumen: 10.200.0.0/22`

Internet tiene del orden de un millón de prefijos, no miles de millones de entradas.

Agregar, y la excepción

Cuatro redes por la misma interfaz, una sola entrada:

```
10.200.0.0/24   000000|00
10.200.1.0/24   000000|01
10.200.2.0/24   000000|10
10.200.3.0/24   000000|11
```

resumen: 10.200.0.0/22

Internet tiene del orden de un millón de prefijos, no miles de millones de entradas.

¿Y si uno de esos `/24` se muda a otro proveedor? Ahora dos rutas coinciden.

Agregar, y la excepción

Cuatro redes por la misma interfaz, una sola entrada:

```
10.200.0.0/24   000000|00
10.200.1.0/24   000000|01
10.200.2.0/24   000000|10
10.200.3.0/24   000000|11
```

`resumen: 10.200.0.0/22`

Internet tiene del orden de un millón de prefijos, no miles de millones de entradas.

¿Y si uno de esos `/24` se muda a otro proveedor? Ahora dos rutas coinciden.

Longest prefix match

De todas las entradas que contienen al destino, se elige la de **prefijo más largo**.

`0.0.0.0/0` coincide con todo, y gana solo si no hay nada mejor.

Ejemplo: `10.200.5.9` y `10.200.6.1`

Prefijo	Salida
`10.200.0.0/16`	A
`10.200.4.0/22`	B
`10.200.5.0/24`	C
`0.0.0.0/0`	D

- 5 es `00000101`, y 4 es `00000100`: los primeros 6 bits coinciden.

Error común

Elegir la ruta más general, o la primera de la tabla. Siempre gana el prefijo más largo, esté donde esté. Las métricas solo eligen entre rutas al *mismo* prefijo.

Ejemplo: `10.200.5.9` y `10.200.6.1`

Prefijo	Salida
`10.200.0.0/16`	A
`10.200.4.0/22`	B
`10.200.5.0/24`	C
`0.0.0.0/0`	D

- 5 es `00000101`, y 4 es `00000100`: los primeros 6 bits coinciden.
- `10.200.5.9` coincide con las cuatro. Gana el `/24`: C.

Error común

Elegir la ruta más general, o la primera de la tabla. Siempre gana el prefijo más largo, esté donde esté. Las métricas solo eligen entre rutas al *mismo* prefijo.

Ejemplo: `10.200.5.9` y `10.200.6.1`

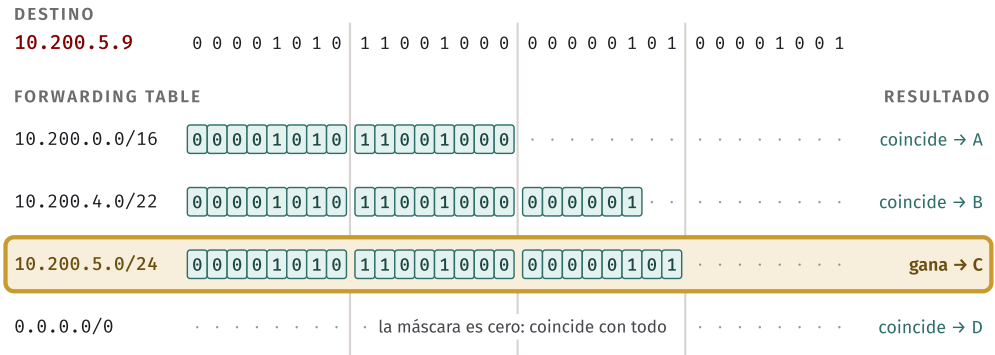
Prefijo	Salida
`10.200.0.0/16`	A
`10.200.4.0/22`	B
`10.200.5.0/24`	C
`0.0.0.0/0`	D

- 5 es `00000101`, y 4 es `00000100`: los primeros 6 bits coinciden.
- `10.200.5.9` coincide con las cuatro. Gana el `/24`: C.
- 6 es `00000110`: entra en `/22`, no en `/24`. Gana B.

Error común

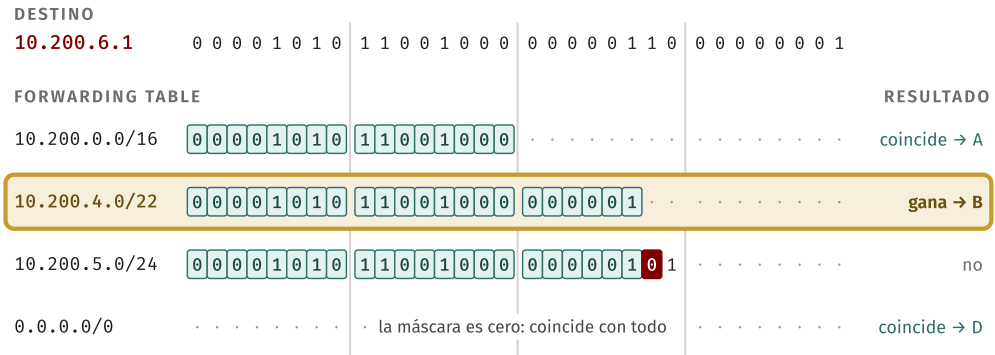
Elegir la ruta más general, o la primera de la tabla. Siempre gana el prefijo más largo, esté donde esté. Las métricas solo eligen entre rutas al *mismo* prefijo.

Longest prefix match, bit a bit



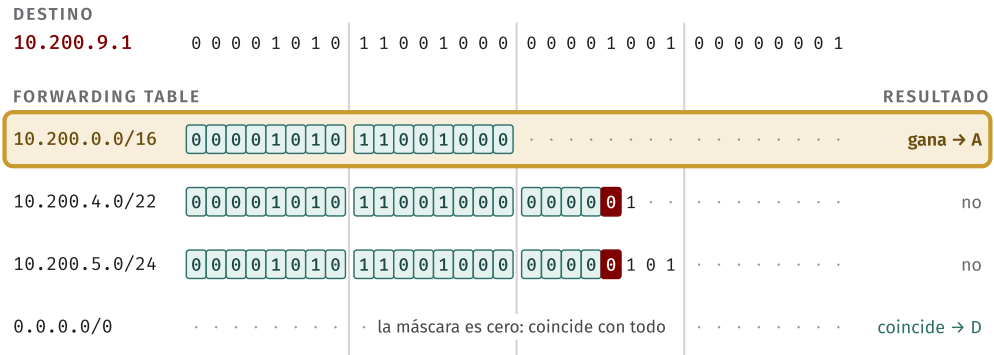
Coincide con las cuatro entradas. Gana la más larga, el /24, y el paquete sale por C.

Longest prefix match, bit a bit



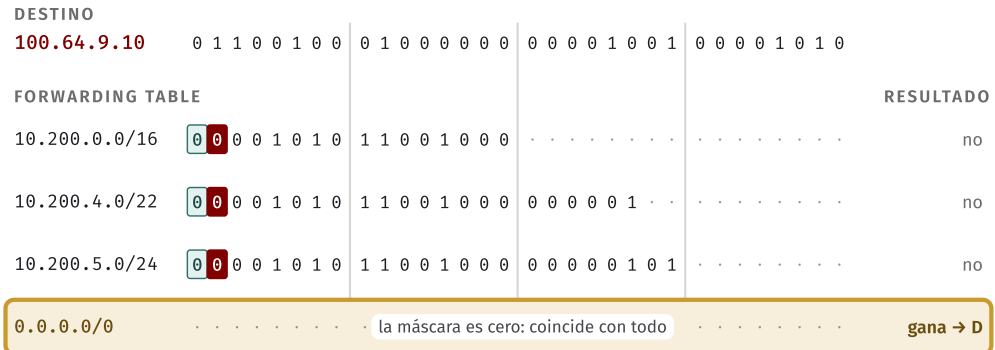
Sus primeros 22 bits entran en el /22, pero el tercer byte ya no es 5. Gana el /22: sale por B.

Longest prefix match, bit a bit



Con el tercer byte 00001001 ya no entra en el /22. Solo quedan el /16 y la ruta por defecto: sale por A.

Longest prefix match, bit a bit



Falla en los primeros bits de todos los prefijos de 10.200. La ruta /0 coincide con todo, y gana porque no hay nada mejor: sale por D.

En C, y en la práctica

lpm.c

```c
for (int i = 0; i < count; i++) {
    const struct route *route = &table[i];
    if ((addr & route->mask) == route->network
        && (best == NULL
            || route->length > best->length))
     {
        best = route;
    }
}
```

./lpm table.txt

```
10.200.6.1
  coincide  10.200.0.0/16 -> A
  coincide  10.200.4.0/22 -> B
  coincide  0.0.0.0/0 -> D
  gana      10.200.4.0/22 -> B
```

- La máscara de `/0` es 0: todas las direcciones coinciden.
- Esto revisa toda la tabla por cada paquete. El hardware usa **TCAM**: compara con todas a la vez. El software usa **tries**: el costo depende de los 32 bits, no del número de rutas.

Lo mismo en un router: una ruta /32 en r1

r1

```
$ ip route show 10.200.3.0/24
10.200.3.0/24 nhid 16 via 10.200.13.2 dev eth3 proto 196 metric 20
$ ip route show 10.200.3.10
10.200.3.10 nhid 25 via 10.200.12.2 dev eth2 proto 196 metric 20
$ ip route get 10.200.3.10
10.200.3.10 via 10.200.12.2 dev eth2 src 10.200.12.1 uid 0
    cache
$ ip route get 10.200.3.11
10.200.3.11 via 10.200.13.2 dev eth3 src 10.200.13.1 uid 0
    cache
```

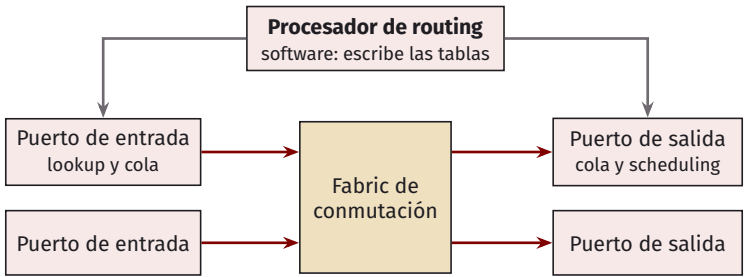
- Se configuró en `vtysh`: `ip route 10.200.3.10/32 10.200.12.2`. Eso es el plano de control.
- `ip route` lee lo instalado en el kernel: el plano de datos.
- Los nodos no tienen ruta por defecto a Docker: la borramos para que lo que no tiene ruta falle.

SECCIÓN 05

Dentro de un router

05

Las cuatro piezas



Fabric	Cómo funciona	Límite
Memoria	Una CPU copia a memoria y luego a la salida	Ancho de banda de la memoria
Bus	Un bus compartido	Pasa un datagrama a la vez
Interconexión	Crossbar y otras: pares de puertos a la vez	Dos entradas a la misma salida

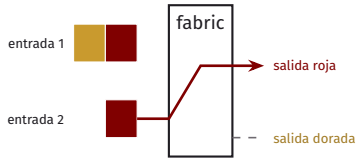
Dónde se forman las colas

A la salida: llega para un enlace más de lo que este transmite. Es la cola de la semana 1.

Dónde se forman las colas

A la salida: llega para un enlace más de lo que este transmite. Es la cola de la semana 1.

A la entrada: si el fabric es lento.



El rojo del frente espera porque la entrada 2 ocupa su salida. El dorado espera detrás, con su salida libre.

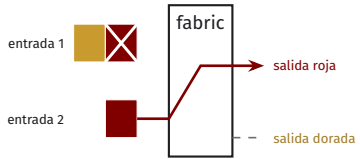
Dónde se forman las colas

A la salida: llega para un enlace más de lo que este transmite. Es la cola de la semana 1.

A la entrada: si el fabric es lento.

Head of the line blocking

El primero de la cola espera por su salida ocupada, y los de atrás esperan con él, aunque sus salidas estén libres. Solución común: una cola por salida.



El rojo del frente espera porque la entrada 2 ocupa su salida. El dorado espera detrás, con su salida libre.

Una cola, medida

Limitamos a 10 Mbit/s el enlace `r1–r3`, y lo llenamos con `iperf3`. Depende de la máquina.

r1

```
$ tc -s qdisc show dev eth3
qdisc tbf 1: root refcnt 15 rate 10Mbit burst 16Kb lat 197ms
 Sent 6296487 bytes 4186 pkt (dropped 9, overlimits 11001 requeues 0)
 backlog 233156b 154p requeues 0
```

Rep.	Sin carga	Con carga	Cola
1	0,145 ms	177,939 ms	154 p
2	0,054 ms	178,672 ms	143 p
3	0,049 ms	178,785 ms	144 p

$233\,156 \times 8/10^7 \approx$ **187 ms**.
Muy cerca del RTT: todo ese retardo es tiempo en la cola.

Una cola, medida

Limitamos a 10 Mbit/s el enlace `r1–r3`, y lo llenamos con `iperf3`. Depende de la máquina.

r1

```
$ tc -s qdisc show dev eth3
qdisc tbf 1: root refcnt 15 rate 10Mbit burst 16Kb lat 197ms
 Sent 6296487 bytes 4186 pkt (dropped 9, overlimits 11001 requeues 0)
 backlog 233156b 154p requeues 0
```

Rep.	Sin carga	Con carga	Cola
1	0,145 ms	177,939 ms	154 p
2	0,054 ms	178,672 ms	143 p
3	0,049 ms	178,785 ms	144 p

$233\,156 \times 8/10^7 \approx$ **187 ms**.
Muy cerca del RTT: todo ese retardo es tiempo en la cola.

Los `dropped` no cupieron: es la señal que TCP usa para bajar la velocidad. El bufferbloat se ve en la semana 11.

Scheduling: ¿a quién le toca esperar?

- **FIFO:** orden de llegada.
- **Prioridad:** sale primero la clase más alta. Puede dejar sin servicio a las demás.
- **Round robin y WFQ:** turnos, con una fracción para cada clase.

prio, con ICMP en la banda alta y el mismo iperf3

```
8 packets transmitted, 8 packets received, 0% packet loss
round-trip min/avg/max = 0.037/0.117/0.217 ms
class prio 10:1 parent 10:
 Sent 784 bytes 8 pkt (dropped 0, overlimits 0 requeues 0)
 backlog 0b 0p requeues 0
class prio 10:2 parent 10:
 Sent 6090187 bytes 4032 pkt (dropped 0, overlimits 0 requeues 0)
 backlog 364874b 241p requeues 0
```

De 178 ms a 0,1 ms

TCP sigue con 241 paquetes en cola, y cada `ping` se adelanta a todos. Una red con voz hace esto, clasificando por DSCP.

Salto por salto: h1 a r1 a r3 a h3



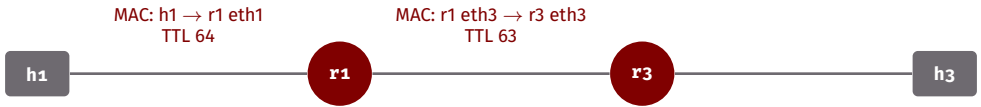
IP en los tres tramos: 10.200.1.10 > 10.200.3.10, id 27682

Enlace	MAC (origen → destino, en esa ejecución)	TTL
h1–r1	aa:c1:ab:ab:19:72 → aa:c1:ab:93:e2:1d	64
r1–r3	aa:c1:ab:28:28:61 → aa:c1:ab:ed:eb:db	63
r3–h3	aa:c1:ab:6c:6f:c8 → aa:c1:ab:d2:bb:08	62

Error común

La MAC de destino es la del *próximo paso*, no la del host final.

Salto por salto: h1 a r1 a r3 a h3



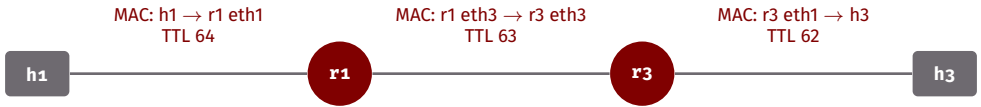
IP en los tres tramos: 10.200.1.10 > 10.200.3.10, id 27682

Enlace	MAC (origen → destino, en esa ejecución)	TTL
h1–r1	aa:c1:ab:ab:19:72 → aa:c1:ab:93:e2:1d	64
r1–r3	aa:c1:ab:28:28:61 → aa:c1:ab:ed:eb:db	63
r3–h3	aa:c1:ab:6c:6f:c8 → aa:c1:ab:d2:bb:08	62

Error común

La MAC de destino es la del *próximo paso*, no la del host final.

Salto por salto: h1 a r1 a r3 a h3



IP en los tres tramos: 10.200.1.10 > 10.200.3.10, id 27682

Enlace	MAC (origen → destino, en esa ejecución)	TTL
h1–r1	aa:c1:ab:ab:19:72 → aa:c1:ab:93:e2:1d	64
r1–r3	aa:c1:ab:28:28:61 → aa:c1:ab:ed:eb:db	63
r3–h3	aa:c1:ab:6c:6f:c8 → aa:c1:ab:d2:bb:08	62

Error común

La MAC de destino es la del *próximo paso*, no la del host final.

SECCIÓN 06

ICMP

06



ICMP: el canal de los errores

Tipo	Cód.	Nombre	Quién lo envía
8	0	Echo request	`ping`
0	0	Echo reply	El destino
3	0	Network unreachable	Router sin ruta
3	1	Host unreachable	Último router
3	3	Port unreachable	El destino
3	4	Fragmentation needed	Router, con DF
11	0	Time exceeded	Router, con TTL 0

- Va **dentro de IP**, protocolo 1.
- Un error trae el encabezado IP y al menos 8 bytes del datagrama que lo causó.
- Nunca un error sobre otro error.

Quién responde importa

El primer router, el último o el destino: esa es la información con la que se diagnostica.

Los errores, en h1

tcpdump -n, en h1 (sin la hora)

```
# Sin ruta en r1: responde r1
IP 10.200.1.10 > 10.200.2.10: ICMP echo request, id 92, seq 0, length 64
IP 10.200.1.1 > 10.200.1.10: ICMP net 10.200.2.10 unreachable, length 92

# Nadie responde a ARP: responde r3, 3 s después
IP 10.200.1.10 > 10.200.3.99: ICMP echo request, id 180, seq 0, length 64
IP 10.200.13.2 > 10.200.1.10: ICMP host 10.200.3.99 unreachable, length 92

# Nadie en ese puerto: responde el destino
IP 10.200.1.10.35713 > 10.200.3.10.9999: UDP, length 5
IP 10.200.3.10 > 10.200.1.10: ICMP 10.200.3.10 udp port 9999 unreachable, length 41

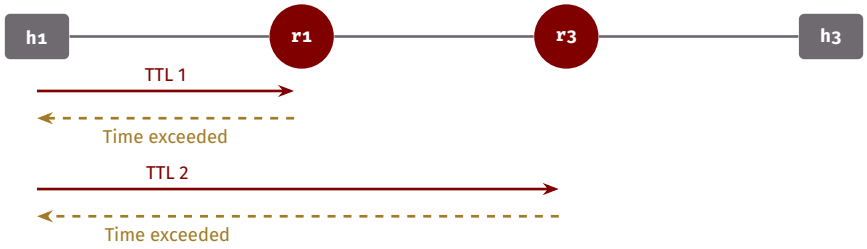
# TTL 1: responde el primer router
IP 10.200.1.10 > 10.200.3.10: ICMP echo request, id 166, seq 0, length 64
IP 10.200.1.1 > 10.200.1.10: ICMP time exceeded in-transit, length 92
```

`tcpdump -v` además decodifica la copia del datagrama original que viene adentro de cada error.

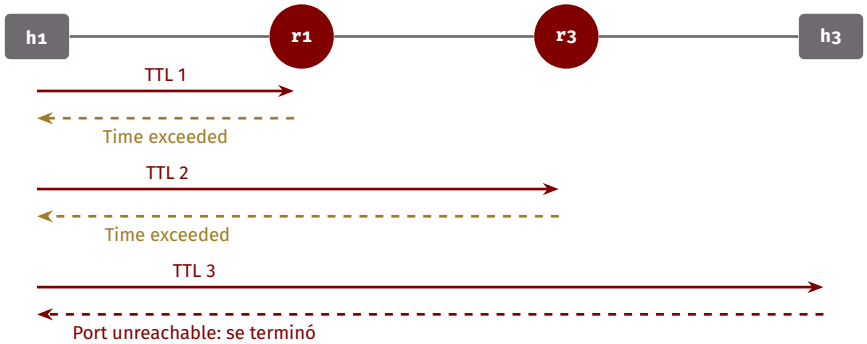
`traceroute`: un truco con dos errores



`traceroute`: un truco con dos errores



`traceroute`: un truco con dos errores



Por debajo de `traceroute`

h1

```
$ traceroute -n -q1 -w1 10.200.3.10
traceroute to 10.200.3.10 (10.200.3.10), 30 hops max, 46 byte packets
 1  10.200.1.1  0.003 ms
 2  10.200.13.2  0.002 ms
 3  10.200.3.10  0.000 ms
```

tcpdump -n, en r1 eth1

```
06:46:16.067892 IP 10.200.1.10.50136 > 10.200.3.10.33435: UDP, length 18
06:46:16.067920 IP 10.200.1.1 > 10.200.1.10: ICMP time exceeded in-transit, length 54
06:46:16.067945 IP 10.200.1.10.50136 > 10.200.3.10.33436: UDP, length 18
06:46:16.067969 IP 10.200.13.2 > 10.200.1.10: ICMP time exceeded in-transit, length 54
06:46:16.067979 IP 10.200.1.10.50136 > 10.200.3.10.33437: UDP, length 18
06:46:16.067992 IP 10.200.3.10 > 10.200.1.10: ICMP 10.200.3.10 udp port 33437 unreachable, length 54
```

Cada sonda usa otro puerto, y el error trae la copia del original: así se emparejan.

¿Y si un router no envía el error?

Con la `/32` por `r2` y `r3` descartando sus Time exceeded:

h1

```
$ traceroute -n -w1 10.200.3.10
traceroute to 10.200.3.10 (10.200.3.10), 30 hops max, 46 byte packets
 1  10.200.1.1  0.003 ms  0.005 ms  0.002 ms
 2  10.200.12.2  0.001 ms  0.002 ms  0.002 ms
 3  *  *  *
 4  10.200.3.10  0.003 ms  0.002 ms  0.002 ms
```

- El salto 4 responde: `r3` sigue haciendo forwarding.
- Los asteriscos significan “no me llegó respuesta”, no “el paquete se perdió”.
- Muchos routers de Internet están configurados así.

Lo que `traceroute` no dice

de h1 a h3

```
1  10.200.1.1  0.004 ms  0.003 ms  0.004 ms
2  10.200.12.2  0.001 ms  0.003 ms  0.002 ms
3  10.200.13.2  0.001 ms  0.004 ms  0.002 ms
4  10.200.3.10  0.000 ms  0.003 ms  0.001 ms
```

de h3 a h1

```
1  10.200.3.1  0.003 ms  0.003 ms  0.003 ms
2  10.200.12.1  0.001 ms  0.003 ms  0.002 ms
3  10.200.1.10  0.002 ms  0.004 ms  0.002 ms
```

- Salto 3: entró a `r3` por `eth2`, pero aparece `eth3`: el origen es la interfaz por la que **sale el mensaje**.

Lo que `traceroute` no dice

de h1 a h3

```
1  10.200.1.1   0.004 ms  0.003 ms  0.004 ms
2  10.200.12.2  0.001 ms  0.003 ms  0.002 ms
3  10.200.13.2  0.001 ms  0.004 ms  0.002 ms
4  10.200.3.10  0.000 ms  0.003 ms  0.001 ms
```

de h3 a h1

```
1  10.200.3.1   0.003 ms  0.003 ms  0.003 ms
2  10.200.12.1  0.001 ms  0.003 ms  0.002 ms
3  10.200.1.10  0.002 ms  0.004 ms  0.002 ms
```

- Salto 3: entró a `r3` por `eth2`, pero aparece `eth3`: el origen es la interfaz por la que **sale el mensaje**.
- Cuatro saltos de ida, tres de vuelta: la `/32` solo está en `r1`.

Lo que `traceroute` no dice

de h1 a h3

```
1  10.200.1.1   0.004 ms  0.003 ms  0.004 ms
2  10.200.12.2  0.001 ms  0.003 ms  0.002 ms
3  10.200.13.2  0.001 ms  0.004 ms  0.002 ms
4  10.200.3.10  0.000 ms  0.003 ms  0.001 ms
```

de h3 a h1

```
1  10.200.3.1   0.003 ms  0.003 ms  0.003 ms
2  10.200.12.1  0.001 ms  0.003 ms  0.002 ms
3  10.200.1.10  0.002 ms  0.004 ms  0.002 ms
```

- Salto 3: entró a `r3` por `eth2`, pero aparece `eth3`: el origen es la interfaz por la que **sale el mensaje**.
- Cuatro saltos de ida, tres de vuelta: la `/32` solo está en `r1`.

Routing asimétrico: lo normal en Internet. Un `traceroute` muestra la ida; para ver la vuelta, se mide desde el otro lado.

SECCIÓN 07

DHCP

07



DHCP: pedir una dirección sin tener una



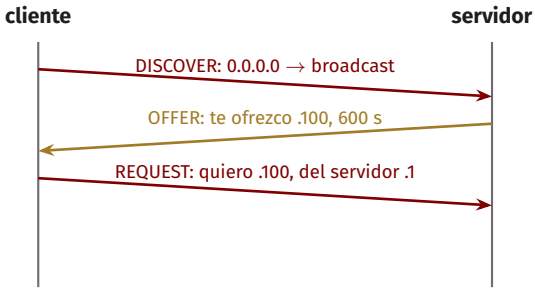
- UDP, puertos 67 y 68.
- Entrega dirección, máscara, router por defecto y servidor de nombres.

DHCP: pedir una dirección sin tener una



- UDP, puertos 67 y 68.
- Entrega dirección, máscara, router por defecto y servidor de nombres.

DHCP: pedir una dirección sin tener una



- UDP, puertos 67 y 68.
- Entrega dirección, máscara, router por defecto y servidor de nombres.
- El `REQUEST` va por broadcast para que los otros servidores suelten su oferta.

DHCP: pedir una dirección sin tener una



- UDP, puertos 67 y 68.
- Entrega dirección, máscara, router por defecto y servidor de nombres.
- El `REQUEST` va por broadcast para que los otros servidores suelten su oferta.
- Es una **concesión** *(lease)*, con plazo.
- Un broadcast no cruza routers: para eso existe el **DHCP relay**.

En vivo: a3 pide, gw responde

tcpdump -n -e -v: primer mensaje del cliente y primera respuesta, recortados

```
aa:c1:ab:37:a1:5c > ff:ff:ff:ff:ff:ff, ethertype IPv4 (0x0800), length 342: ...
    0.0.0.0.68 > 255.255.255.255.67: BOOTP/DHCP, Request from aa:c1:ab:37:a1:5c, length 300, xid 0xc944a441, Flags [none]
          DHCP-Message (53), length 1: Discover
aa:c1:ab:db:70:c5 > ff:ff:ff:ff:ff:ff, ethertype IPv4 (0x0800), length 310: ...
    10.200.50.1.67 > 255.255.255.255.68: BOOTP/DHCP, Reply, length 268, xid 0xc944a441, Flags [none]
         Your-IP 10.200.50.100
          DHCP-Message (53), length 1: Offer
          Lease-Time (51), length 4: 600
          Subnet-Mask (1), length 4: 255.255.255.0
          Default-Gateway (3), length 4: 10.200.50.1
```

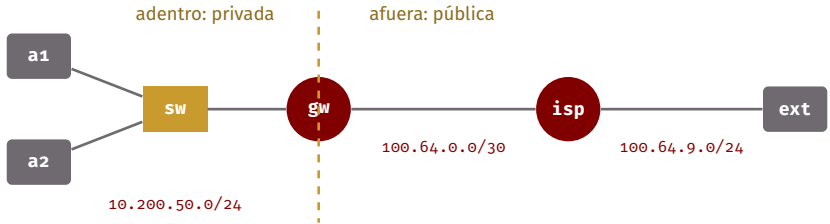
`Request` y `Reply` son códigos de BOOTP; el tipo real es la **opción 53**. Nuestro servidor siempre responde por broadcast porque es simple: según el RFC 2131, con `Flags [none]` se responde por unicast.

SECCIÓN 08

NAT

08

La idea



- Direcciones privadas (RFC 1918): `10.0.0.0/8`, `172.16.0.0/12` y `192.168.0.0/16`. Se pueden reutilizar, y no tienen ruta en Internet.
- El router de borde **reescribe** dirección y puerto de origen al salir, y lo deshace al volver.
- Sin NAT, `a1` llega a `ext`, pero la respuesta no vuelve: `isp` no conoce `10.200.50.0/24`.

Una regla, y el mismo paquete a cada lado

nat/masquerade.nft

```
table ip nat {
    chain postrouting {
        type nat hook postrouting priority srcnat; policy accept;
        ip saddr 10.200.50.0/24 oifname "eth2" masquerade
    }
}
```

tcpdump -n, un ping de a1 a ext: ext nunca se entera de que existe a1

```
[gw.eth1]
06:48:32.642207 IP 10.200.50.10 > 100.64.9.10: ICMP echo request, id 80, seq 0, length 64
06:48:32.642259 IP 100.64.9.10 > 10.200.50.10: ICMP echo reply, id 80, seq 0, length 64
[gw.eth2]
06:48:32.642220 IP 100.64.0.2 > 100.64.9.10: ICMP echo request, id 80, seq 0, length 64
06:48:32.642255 IP 100.64.9.10 > 100.64.0.2: ICMP echo reply, id 80, seq 0, length 64
```

Dos hosts, un solo puerto

conntrack -L -p tcp, en gw: a1 y a2 se conectan a ext, ambos desde el puerto 40000

```
tcp      6 431997 ESTABLISHED src=10.200.50.11 dst=100.64.9.10 sport=40000 dport=9000
                              src=100.64.9.10 dst=100.64.0.2 sport=9000 dport=48838 [ASSURED] mark=0 use=1
tcp      6 431997 ESTABLISHED src=10.200.50.10 dst=100.64.9.10 sport=40000 dport=9000
                              src=100.64.9.10 dst=100.64.0.2 sport=9000 dport=40000 [ASSURED] mark=0 use=1
```

Adentro	Afuera	Destino
10.200.50.10:40000	100.64.0.2:40000	100.64.9.10:9000
10.200.50.11:40000	100.64.0.2:48838	100.64.9.10:9000

Un NAT tiene estado

Una respuesta a `100.64.0.2:48838` coincide con una sola fila. Si el NAT se reinicia sin esa tabla, todas las conexiones se cortan.

Dos hosts, un solo puerto: paso a paso

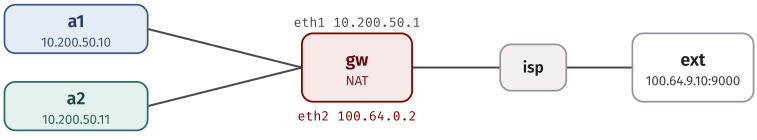


TABLA DE TRADUCCIÓN DE GW

adentro (origen)	afuera (origen)	destino
10.200.50.10:40000	100.64.0.2:40000	100.64.9.10:9000

a1 abre una conexión desde el puerto 40000. gw cambia el origen por su dirección pública y anota la fila. El puerto 40000 estaba libre, así que lo conserva.

Dos hosts, un solo puerto: paso a paso

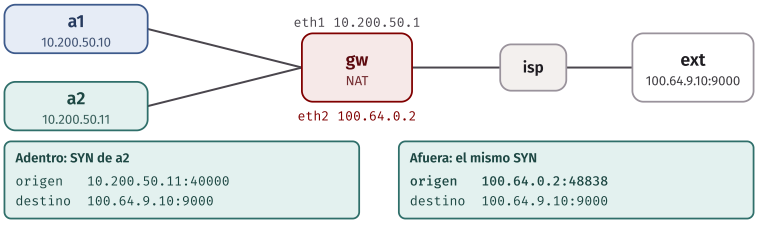


TABLA DE TRADUCCIÓN DE GW

adentro (origen)	afuera (origen)	destino
10.200.50.10:40000	100.64.0.2:40000	100.64.9.10:9000
10.200.50.11:40000	100.64.0.2:48838	100.64.9.10:9000

a2 usa el mismo puerto. Afuera tendrían la misma dirección y el mismo puerto, así que gw le da otro: el 48838. Ahora ext ve dos conexiones distintas.

Dos hosts, un solo puerto: paso a paso

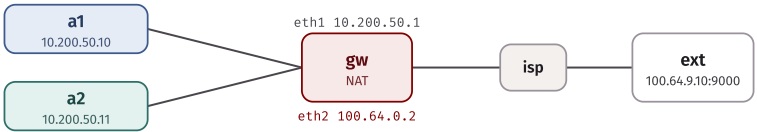


TABLA DE TRADUCCIÓN DE GW

adentro (origen)	afuera (origen)	destino
10.200.50.10:40000	100.64.0.2:40000	100.64.9.10:9000
10.200.50.11:40000	100.64.0.2:48838	100.64.9.10:9000

ext responde al puerto 48838. Solo una fila coincide: gw deshace la traducción y entrega el paquete a a2, en su dirección y puerto de adentro.

De afuera hacia adentro

tcpdump -n (sin la hora)

```
# La dirección privada, vista en ext: la rechaza el primer router
IP 100.64.9.10.41568 > 10.200.50.10.8080: Flags [S], seq 2854544516, win 64240, ...
IP 100.64.9.1 > 100.64.9.10: ICMP net 10.200.50.10 unreachable, length 68
# La dirección pública, vista en gw: sin fila en la tabla, gw responde con RST
IP 100.64.9.10.57644 > 100.64.0.2.8080: Flags [S], seq 1619685849, win 64240, ...
IP 100.64.0.2.8080 > 100.64.9.10.57644: Flags [R.], seq 0, ack 1619685850, win 0, length 0
```

De afuera hacia adentro

tcpdump -n (sin la hora)

```
# La dirección privada, vista en ext: la rechaza el primer router
IP 100.64.9.10.41568 > 10.200.50.10.8080: Flags [S], seq 2854544516, win 64240, ...
IP 100.64.9.1 > 100.64.9.10: ICMP net 10.200.50.10 unreachable, length 68
# La dirección pública, vista en gw: sin fila en la tabla, gw responde con RST
IP 100.64.9.10.57644 > 100.64.0.2.8080: Flags [S], seq 1619685849, win 64240, ...
IP 100.64.0.2.8080 > 100.64.9.10.57644: Flags [R.], seq 0, ack 1619685850, win 0, length 0
```

nat/dnat_demo.nft: dentro de table ip nat

```
1  chain prerouting {
2      type nat hook prerouting priority dstnat; policy accept
         ;
3      iifname "eth2" tcp dport 8080 dnat to 10.200.50.10:8080
4  }
```

ahora sí, en ext

```
$ echo x | nc -w2 100.64.0.2 8080
hola-desde-a1
```

Lo que NAT cuesta

- Los routers no deberían meterse en el transporte ni guardar estado por conexión.
- Cambia también el checksum de TCP, y rompe protocolos que llevan direcciones en sus datos.
- Lo peor: cualquier host ya no le puede hablar a cualquier otro.

Error común

Un NAT no es un firewall. Bloquea lo entrante como efecto lateral, porque no sabe a quién entregarlo. Una política de seguridad se escribe como tal: semana 15.

Lo que NAT cuesta

- Los routers no deberían meterse en el transporte ni guardar estado por conexión.
- Cambia también el checksum de TCP, y rompe protocolos que llevan direcciones en sus datos.
- Lo peor: cualquier host ya no le puede hablar a cualquier otro.
- Por eso existen STUN y hole punching, y por eso los proveedores usan carrier-grade NAT en `100.64.0.0/10`.

Error común

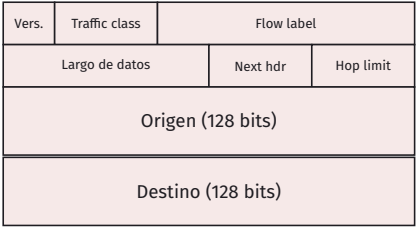
Un NAT no es un firewall. Bloquea lo entrante como efecto lateral, porque no sabe a quién entregarlo. Una política de seguridad se escribe como tal: semana 15.

SECCIÓN 09

IPv6

09

128 bits y un encabezado más simple



40 bytes fijos. Se pueden omitir ceros: `2001:db8:1::10`. IANA entregó sus últimos bloques IPv4 en febrero de 2011.

IPv4	En IPv6
TTL	Hop limit
Protocolo	Next header
IHL	Se fue: tamaño fijo
Id, flags, offset	Se fueron: los routers no fragmentan
Checksum	Se fue: no se rehace en cada salto
Opciones	Encabezados de extensión

`fe80::/10` link-local, `ff00::/8` multicast (no hay broadcast), `2001:db8::/32` documentación. Una LAN es un `/64`.

No hay ARP: neighbor discovery

tcpdump -n -v, en h1: ping de h1 a r3 (sin hora ni sumas)

```
IP6 (hlim 255, next-header ICMPv6 (58) payload length: 32) 2001:db8:1::10 > ff02::1:ff00:1:
    ICMP6, neighbor solicitation, length 32, who has 2001:db8:1::1
IP6 (hlim 255, next-header ICMPv6 (58) payload length: 32) 2001:db8:1::1 > 2001:db8:1::10:
    ICMP6, neighbor advertisement, length 32, tgt is 2001:db8:1::1, Flags [router, solicited, override]
IP6 (flowlabel 0x507a4, hlim 64, next-header ICMPv6 (58) payload length: 64) 2001:db8:1::10 > 2001:db8:13::2:
    ICMP6, echo request, id 376, seq 0
IP6 (flowlabel 0xda6f7, hlim 63, next-header ICMPv6 (58) payload length: 64) 2001:db8:13::2 > 2001:db8:1::10:
    ICMP6, echo reply, id 376, seq 0
```

- La solicitud va a un grupo multicast, no a todos: `ff02::1:ff` + los últimos 24 bits de la dirección buscada.
- **ICMPv6 no es opcional**: sin él no hay vecinos. Bloquearlo deja sin red.

No hay ARP: neighbor discovery

tcpdump -n -v, en h1: ping de h1 a r3 (sin hora ni sumas)

```
IP6 (hlim 255, next-header ICMPv6 (58) payload length: 32) 2001:db8:1::10 > ff02::1:ff00:1:
    ICMP6, neighbor solicitation, length 32, who has 2001:db8:1::1
IP6 (hlim 255, next-header ICMPv6 (58) payload length: 32) 2001:db8:1::1 > 2001:db8:1::10:
    ICMP6, neighbor advertisement, length 32, tgt is 2001:db8:1::1, Flags [router, solicited, override]
IP6 (flowlabel 0x507a4, hlim 64, next-header ICMPv6 (58) payload length: 64) 2001:db8:1::10 > 2001:db8:13::2:
    ICMP6, echo request, id 376, seq 0
IP6 (flowlabel 0xda6f7, hlim 63, next-header ICMPv6 (58) payload length: 64) 2001:db8:13::2 > 2001:db8:1::10:
    ICMP6, echo reply, id 376, seq 0
```

- La solicitud va a un grupo multicast, no a todos: `ff02::1:ff` + los últimos 24 bits de la dirección buscada.
- **ICMPv6 no es opcional**: sin él no hay vecinos. Bloquearlo deja sin red.
- A cada salto: las MAC cambian, las IP no, y el hop limit baja de 64 a 63. Ningún checksum de IP se recalcula.
- El frame del mismo `ping`: 118 bytes (14 + 40 + 64); en IPv4 eran 98.

Los routers no fragmentan

MTU 1280 en `r1–r3`, el mínimo de IPv6, y un `ping` de 1400 bytes:

en h1

```
2001:db8:1::10 > 2001:db8:13::2: [icmp6 sum ok] ICMP6, echo request, id 414, seq 0
2001:db8:1::1 > 2001:db8:1::10: [icmp6 sum ok] ICMP6, packet too big, mtu 1280
```

Los routers no fragmentan

MTU 1280 en `r1–r3`, el mínimo de IPv6, y un `ping` de 1400 bytes:

en h1

```
2001:db8:1::10 > 2001:db8:13::2: [icmp6 sum ok] ICMP6, echo request, id 414, seq 0
2001:db8:1::1 > 2001:db8:1::10: [icmp6 sum ok] ICMP6, packet too big, mtu 1280
```

Al repetirlo, fragmenta el **origen**, con un encabezado de extensión:

en h1 (sin la hora)

```
IP6 (flowlabel 0x507a4, hlim 64, next-header Fragment (44) payload length: 1240) 2001:db8:1::10 > 2001:db8:13::2:
    frag (0x02aa4b39:0|1232) ICMP6, echo request, id 434, seq 0
IP6 (flowlabel 0x507a4, hlim 64, next-header Fragment (44) payload length: 184) 2001:db8:1::10 > 2001:db8:13::2:
    frag (0x02aa4b39:1232|176)
```

Los routers no fragmentan

MTU 1280 en `r1`–`r3`, el mínimo de IPv6, y un `ping` de 1400 bytes:

en h1

```
2001:db8:1::10 > 2001:db8:13::2: [icmp6 sum ok] ICMP6, echo request, id 414, seq 0
2001:db8:1::1 > 2001:db8:1::10: [icmp6 sum ok] ICMP6, packet too big, mtu 1280
```

Al repetirlo, fragmenta el **origen**, con un encabezado de extensión:

en h1 (sin la hora)

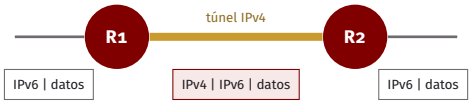
```
IP6 (flowlabel 0x507a4, hlim 64, next-header Fragment (44) payload length: 1240) 2001:db8:1::10 > 2001:db8:13::2:
    frag (0x02aa4b39:0|1232) ICMP6, echo request, id 434, seq 0
IP6 (flowlabel 0x507a4, hlim 64, next-header Fragment (44) payload length: 184) 2001:db8:1::10 > 2001:db8:13::2:
    frag (0x02aa4b39:1232|176)
```

40 de IPv6 + 8 del encabezado de fragmento + 1232 de datos = 1280. Los 1408 bytes de ICMPv6 se reparten en 1232 + 176.

Convivencia de IPv4 e IPv6

Doble pila *(dual stack)*: cada nodo tiene direcciones de los dos tipos. Lo más limpio, pero todo el camino debe hablar IPv6.

Túneles *(tunneling)*: el paquete IPv6 entero viaja como datos de un datagrama IPv4, con protocolo 41.



Otra vez las capas

Un túnel trata a una red completa como si fuera un enlace. Volvemos a esto con los overlays de la semana 14 y las VPN de la semana 15.

SECCIÓN 10

Recorrido completo

10

El primer SYN de a3, de punta a punta



CAMPO	a3 → gw
MAC origen	…37:a1:5c
MAC destino	…db:70:c5
IP origen	10.200.50.100
Puerto origen	45000
IP destino	100.64.9.10
TTL	64
Checksum TCP	0x92ea

a3 ya tiene su dirección por DHCP. El destino no está en su subred, así que busca con ARP la MAC del router y le envía el SYN.

El primer SYN de a3, de punta a punta



CAMPO	a3 → gw
MAC origen	…37:a1:5c
MAC destino	…db:70:c5
IP origen	10.200.50.100
Puerto origen	45000
IP destino	100.64.9.10
TTL	64
Checksum TCP	0x92ea

El switch reenvía el frame por el puerto donde aprendió la MAC de gw. No toca nada: un switch no aparece en un traceroute.

El primer SYN de a3, de punta a punta



CAMPO	a3 → gw	gw → isp
MAC origen	…37:a1:5c	…6e:fa:64
MAC destino	…db:70:c5	…f6:50:35
IP origen	10.200.50.100	100.64.0.2
Puerto origen	45000	45000
IP destino	100.64.9.10	100.64.9.10
TTL	64	63
Checksum TCP	0x92ea	0x6bd4

gw hace un frame nuevo, baja el TTL y aplica el NAT: cambia la IP de origen, y por eso cambia el checksum de TCP. El puerto 45000 estaba libre y se conserva.

El primer SYN de a3, de punta a punta



CAMPO	a3 → gw	gw → isp	isp → ext
MAC origen	…37:a1:5c	…6e:fa:64	…aa:25:b1
MAC destino	…db:70:c5	…f6:50:35	…a0:13:2a
IP origen	10.200.50.100	100.64.0.2	100.64.0.2
Puerto origen	45000	45000	45000
IP destino	100.64.9.10	100.64.9.10	100.64.9.10
TTL	64	63	62
Checksum TCP	0x92ea	0x6bd4	0x6bd4

isp también hace un frame nuevo y baja el TTL. El checksum de TCP no cambia, porque el TTL no está en el pseudo-encabezado. Ningún router tocó la IP de destino.

Lean la tabla por columnas

- **MAC:** cambian en cada enlace, siempre.

Lean la tabla por columnas

- **MAC:** cambian en cada enlace, siempre.
- **IP de origen:** cambia una sola vez, en el NAT.
- **Puerto de origen:** se mantuvo porque 45000 estaba libre.

Lean la tabla por columnas

- **MAC:** cambian en cada enlace, siempre.
- **IP de origen:** cambia una sola vez, en el NAT.
- **Puerto de origen:** se mantuvo porque 45000 estaba libre.
- **IP de destino:** no cambia nunca. Guía cada decisión de forwarding.
- **TTL:** baja en 1 en cada router. El switch no lo toca.

Lean la tabla por columnas

- **MAC:** cambian en cada enlace, siempre.
- **IP de origen:** cambia una sola vez, en el NAT.
- **Puerto de origen:** se mantuvo porque 45000 estaba libre.
- **IP de destino:** no cambia nunca. Guía cada decisión de forwarding.
- **TTL:** baja en 1 en cada router. El switch no lo toca.
- **Checksum de TCP:** cambia en el NAT, porque cubre las direcciones. El TTL no lo afecta.
- **`seq`, `win` y opciones:** no se tocan. Son asunto de los extremos.

Lean la tabla por columnas

- **MAC:** cambian en cada enlace, siempre.
- **IP de origen:** cambia una sola vez, en el NAT.
- **Puerto de origen:** se mantuvo porque 45000 estaba libre.
- **IP de destino:** no cambia nunca. Guía cada decisión de forwarding.
- **TTL:** baja en 1 en cada router. El switch no lo toca.
- **Checksum de TCP:** cambia en el NAT, porque cubre las direcciones. El TTL no lo afecta.
- **`seq`, `win` y opciones:** no se tocan. Son asunto de los extremos.

Si entienden esta tabla, entienden las semanas 3 y 4

El switch mira la MAC y no toca nada. El router mira la IP de destino y reescribe el enlace y el TTL. El NAT, además, reescribe la IP y el puerto de origen.

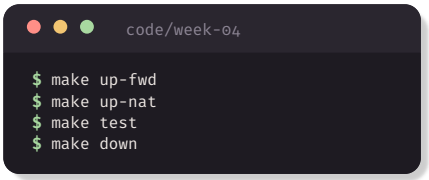
SECCIÓN 11

Laboratorio

11

Subredes y NAT con Containerlab

1. Completar las rutas de tres routers: `lab/routes.sh`.
2. Predecir la ruta `/32`, y comprobarla.
3. Armar la tabla de MAC, IP y TTL con `tcpdump`.
4. Cargar el NAT, capturar a los dos lados y leer `conntrack`.
5. Conexión entrante y redirección de puerto: `lab/portforward.nft`.
6. IPv6 hasta `h3`: `lab/ipv6_h3.sh`.
7. Avanzar con **P1**.



Las pruebas de `lab/` se saltan hasta que borren la línea `TODO`.

Para la próxima clase

- Terminar el laboratorio, y seguir con el plan de direcciones de P1.
- Leer Kurose y Ross, secs. 5.1 y 5.2, y CS168: *Model for Intra-Domain Routing*, *Distance Vector* y *Link State*.

Pregunta

Esta semana escribimos las rutas a mano. Si el enlace `r1`–`r3` se cae, ¿quién avisa a `r1`, y cómo se entera `r2`? ¿Qué tendría que hacer un protocolo para que nadie toque un router?