Capítulo 8

Interconexión

Al terminar este capítulo, entenderás:

- Cómo se mueven los paquetes IP a través de una interred.
- Qué son y qué hacen los routers.
- Qué es una tabla de encaminamiento, qué información contiene y cómo la utiliza.
- Cómo los routers colaboran con otros para llevar los paquetes hasta su destino.
- Cómo pueden los paquetes IP atravesar redes con tecnologías de enlace diferentes y qué es la fragmentación de paquetes.

Sin duda alguna, la posibilidad de llevar información desde prácticamente cualquier punto del planeta a cualquier otro es lo que ha hecho de Internet una innovación que ha cambiado para siempre la mayoría de las industrias, el comercio, la economía, el entretenimiento y entre otras muchas cosas, lo más importante, la forma en la que los humanos nos comunicamos.

Una de las tecnologías clave que ha hecho posible esta revolución es la interconexión de redes, particularmente, cuando se trata de tecnologías heterogéneas. Aunque es cierto que existieron y existen otras tecnologías que ofrecen una funcionalidad similar, ha sido TCP/IP la que ha logrado convertirse con diferencia en la más popular.

Por supuesto, llevar paquetes a través de múltiples redes es uno de los objetivos prioritarios de TCP/IP y en particular del protocolo IP. Citando la RFC 791 [9]:

> "The purpose of internet protocol is to move datagrams through an interconnected set of networks".

En español:

> "El propósito del protocolo de interred es mover datagramas a través de un conjunto interconectado de redes".

De nuevo insistimos aquí, por su importancia, que «Protocolo de Internet» debe entenderse como «protocolo de interconexión de redes», porque es IP el que hace posible las interredes.

Para interconectar redes es necesario resolver varios problemas. El principal es el encaminamiento, el mecanismo que determina la ruta que siguen los paquetes hasta su destino. Los routers juegan un papel clave. Son de hecho los dispositivos que crean las interredes. Disponen de varias interfaces, conectadas a redes distintas, y por eso pueden mover paquetes entre ellas. Pero también es necesario resolver los problemas que aparecen cuando se utilizan tecnologías de enlace diferentes al interconectar redes heterogéneas. Y además debemos entender el modo en que los routers informan de errores o situaciones especiales a otros routers y a los nodos finales.

8.1. Almacenamiento y reenvío

Definamos primero lo que es un router: Es un dispositivo conectado a más de una red, que además, tiene la capacidad de aceptar paquetes *dirigidos a terceros* y reenviarlos (del inglés *forwarding*) para colaborar en la tarea de llevarlos hasta su destino.

La diferencia clave entre un PC que actúa como un simple «nodo final» y otro que actúa como router, es que el segundo reenvía los paquetes que no son para él, es decir, aquellos cuya dirección destino no coincide con la dirección asignada a la interfaz por la que llegan. El nodo final en cambio, los descarta.

En un sistema GNU/Linux, ese *reenvío* se activa con el parámetro del núcleo `net.ipv4.ip_forward`:

```
# echo 1 > /proc/sys/net/ipv4/ip_forward
```

Recuerda que para hacer este cambio persistente, debes editar el archivo `/etc/sysctl.conf` (ver § 2.10).

Veamos una topología de ejemplo (Figura 8.1) que vamos a utilizar para ilustrar varios conceptos y mecanismos a lo largo del capítulo. Esta interred está formada por 5 nodos (`PC1` a `PC5`) conectados a 3 redes Ethernet (`N1` a `N3`). Los routers `R1` y `R2` tiene 2 interfaces Ethernet y una interfaz serie con encapsulación PPP. Para todas las interfaces se indica la dirección IP y el último octeto de la dirección MAC[1].

[1] Es solo una simplificación para poder usar aquí datos más sencillos

Aquí podemos ver cómo los routers tienen interfaces en varias redes que además pueden ser de distinta tecnología y protocolo de enlace (Ethernet y PPP en este caso).

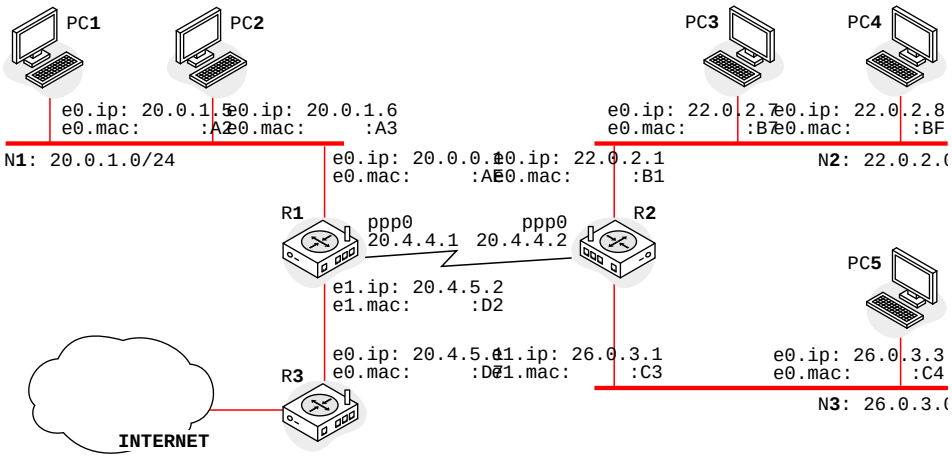

FIGURA 8.1: Topología de ejemplo

IP es una tecnología de conmutación de paquetes, que también se suele llamar «red de datagramas». Un datagrama es un bloque de datos independiente, que lleva incorporada —en su cabecera— toda la información necesaria para llegar a su destino. El router desconoce —o simplemente pasa por alto— si una secuencia de paquetes que está recibiendo pertenecen a la misma conexión o flujo, proceden del mismo origen o van al mismo destino. Aunque tuvieran relación, son tratados como si no la tuvieran. Por eso, paquetes que sí están relacionados podrían seguir rutas diferentes, y eso tiene consecuencias importantes.

Un router IP recibe y procesa los paquetes que van llegando a cualquiera de sus interfaces, sin importar la tecnología de enlace que emplean. Como la lógica del router puede estar ocupada cuando llega un nuevo paquete, es necesario guardar el paquete temporalmente en una cola. De forma similar, cuando el router quiere reenviar un paquete, la linea de salida puede estar ocupada, por lo que también hace falta una cola para almacenar los paquetes pendientes de enviar. Resumiendo, cada interfaz tiene una cola de entrada y otra de salida.

Por esta forma de trabajar se dice que el router es un dispositivo de *almacenamiento y reenvío* (*store and forward*). Veamos con más detalle el proceso que realiza el router cada vez que llega un paquete:

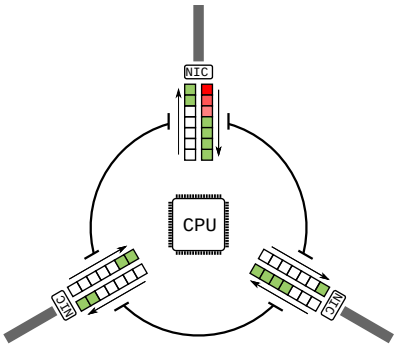

FIGURA 8.2: Esquema de las colas de entrada y salida de un router

1. Una vez recibido por la interfaz, el paquete se extrae de su envoltorio de enlace. La cabecera y cola del protocolo de la trama se descarta. Es una tarea que realiza la propia NIC como en cualquier otro dispositivo.
2. El paquete se almacena en memoria, en concreto, en la cola de entrada de esa interfaz; y espera allí hasta que el router disponga de tiempo/recursos para procesarlo.
3. Cuando por fin el router procesa el paquete, calcula su *checksum* y comprueba que coincida con el que aparece en la cabecera. Si esta comprobación falla, el paquete es descartado (y el origen es informado).
4. Comprueba el campo TTL de la cabecera. Si es 0, se descarta el paquete e informa al origen mediante un mensaje ICMP *time exceeded* (§ 8.6.2 para más detalles).
   En otro caso (si es mayor que 0), lo decrementa en 1, calcula el nuevo *checksum* y actualiza ambos campos en la cabecera. El campo TTL evita que un paquete pueda quedar viajando por la red indefinidamente en caso de producirse un bucle de encaminamiento.

El checksum al que nos referimos aquí es una función de suma de comprobación que permite detectar cualquier modificación que hayan sufrido los datos durante la transmisión, incluso aunque afecte a un único bit. Se conoce como *Internet checksum* y lo utilizan varios protocolos de la familia TCP/IP. Aparte de la cabecera IPv4 que estamos tratando aquí, lo usan TCP, UDP, ICMP, e IGMP, entre otros. En el repositorio de código del libro puedes encontrar una implementación en Python de este algoritmo (`/inet-checksum/checksum.py`) y un notebook con una explicación detallada de su funcionamiento (`/inet-checksum/checksum.ipynb`).

5. Determina la interfaz de salida en función exclusivamente de la dirección destino del paquete y del contenido de la tabla de encaminamiento. Si la tabla no contiene información sobre cómo proceder, el paquete se descarta y, de nuevo, informa del problema al origen con un mensaje ICMP *destination unreachable* (destino inalcanzable).
6. El paquete se coloca en la cola de la interfaz de salida elegida.
7. Cuando el medio físico asociado a la interfaz queda libre, la NIC encapsula el paquete IP con el formato de trama según la tecnología de ese enlace que lo transforma en una señal capaz de alcanzar el siguiente dispositivo.
   Cuando esta interfaz de salida está conectada a un enlace de difusión (*p. ej.* una LAN Ethernet), el router debe determinar primero la dirección física destino de la trama.

Todo este proceso es lo que llamamos «un salto», una expresión que enfatiza la idea de que el paquete ha pasado (saltado) de una red a otra. Los sucesivos routers a lo largo de la ruta efectúan cada uno de estos saltos hasta el destino. Es la base del funcionamiento cualquier interred IP.

8.2. Entrega directa vs. indirecta

El trabajo del router es entregar el paquete «al siguiente» en la ruta hacia su destino. En este camino, hay dos tipos de entrega:

- La entrega directa (o local) la realiza cuando el destino del paquete es un vecino del origen. Por ejemplo, si ambos (origen y destino) están conectados a una red Ethernet, el origen encapsula el paquete en una trama que lleva en su cabecera la dirección física del destino. Tanto la dirección MAC destino (en la cabecera de la trama) como la IP destino (en la cabecera del paquete) identifican al nodo destino.
- La entrega indirecta ocurre, como puedes suponer, cuando el destino no es vecino y, de hecho, probablemente el origen no tiene ninguna información en absoluto de dónde se encuentra ese destino. En ese caso, el origen debe entregar el paquete a un router asumiendo que él sabe cómo llevarlo al destino. Aquí también se encapsula el paquete en una trama, pero la dirección física destino es la del siguiente router, no la del destino, es decir, la trama va dirigida al router, pero el paquete encapsulado va dirigido al destino.

Fíjate que aunque en esta explicación sirve tanto para routers como para nodos finales. Ambos, cuando necesitan enviar un paquete a un vecino, realizan una entrega directa, y cuando el destino está fuera de la red, realizan una entrega indirecta.

En la sección 8.5 veremos un ejemplo práctico con datos y mensajes concretos que ilustra ambos tipos de entrega en una interred sencilla, pero no trivial.

8.3. Tabla de encaminamiento

La tabla de encaminamiento contiene la información que necesita el router para hacer su trabajo principal: decidir dónde enviar cada paquete que llega. Cada fila de esta tabla especifica un destino (normalmente una red) y qué hacer para llegar a él. Sin embargo, la tabla no tiene instrucciones precisas de cómo llevar el paquete hasta allí, solo habla del siguiente salto. Veamos un ejemplo en la Tabla 8.1.

#	dst	mask	next hop	iface
1	30.0.0.0	255.255.255.0	0.0.0.0	e0
2	40.0.0.0	255.255.255.0	0.0.0.0	e1
3	130.10.20.0	255.255.255.0	30.0.0.2	e0
4	210.20.30.0	255.255.64.0	40.0.0.3	e1
5	0.0.0.0	0.0.0.0	50.1.1.2	s0

CUADRO 8.1: Ejemplo de tabla de encaminamiento

Vale la pena hacer aquí una aclaración para evitar malentendidos. La tabla de encaminamiento (*routing table*) se denomina muy a menudo en español «tabla de rutas». Pero si entendemos que una «ruta» es un «itinerario o camino para un viaje»[2], la traducción «tabla de rutas» es incorrecta, o al menos, confusa. Como acabamos de explicar, este tipo de tabla por norma general no contiene rutas. Por eso optaremos aquí por el término «tabla de encaminamiento». A pesar de esto, en muchas ocasiones (incluso en inglés) para hacer referencia a las filas de la tabla, se habla de «rutas».

Toda tabla de encaminamiento contiene al menos la siguiente información:

Destino *(dst)*
: Una dirección IP que identifica una red. Aunque menos frecuente, también puede ser una dirección de nodo. Un caso especial es la dirección IP nula (`0.0.0.0`) que indica que esa fila es un *router por defecto*.

Máscara *(mask)*
: La máscara asociada al destino (la primera columna) y determina qué parte de aquella corresponde al prefijo de red (ver sección 7.2). En ocasiones la columna «destino» se combina con la máscara en formato CIDR (*p. ej.* `120.10.12.0/24`).

[2] Así lo define la RAE.

Siguiente salto *(next-hop)*
: La dirección IP de un router vecino, o bien la dirección IP nula (`0.0.0.0`) que aquí significa entrega directa.

Interfaz *(iface)*
: El nombre de la interfaz por la que debe salir el paquete.

Para cada nuevo paquete que llega, al router busca la correspondencia entre la dirección destino del paquete y la combinación destino-máscara de de cada fila. Como varias filas podrían cumplir esta correspondencia, se aplica el siguiente criterio de prioridad:

1. Entrega directa.
2. Entrega indirecta.
3. Router por defecto.

Dentro de cada categoría se aplica LPM (Longest Prefix Match), es decir, la máscara más larga (más específica) tiene más prioridad. Si aún así hay empate, se aplica la primera fila que aparezca en la tabla.

Si no es posible encontrar ninguna coincidencia, el paquete es descartado (y el origen informado con un mensaje ICMP). Sin embargo, la tabla puede incluir *routers por defecto* (*default routers*). Normalmente solo hay uno y suele aparecer en la última fila (como en la Tabla 8.1), pero eso es solo una convención y en realidad la posición dónde aparezca es irrelevante.

Pero exactamente, ¿cuál es esa «correspondencia» que se evalúa para cada fila? Consiste simplemente en una operación AND a nivel de bits entre la dirección destino del paquete y la máscara de esa fila. La fila es adecuada si el resultado de esa operación coincide con el valor de la columna «destino».

Cuando el router determina qué fila utilizar, envía el paquete a la dirección que aparece en la columna «siguiente salto» a través de la interfaz indicada en la columna «interfaz». Fíjate que la columna «interfaz» siempre identifica interfaces del propio router.

Siguiendo el ejemplo, supongamos que al router de la Tabla 8.1 llega un paquete con destino `40.0.0.9`. La única fila coincidente sería la , lo cual se determina al aplicar la operación indicada:

$$(40.0.0.9 \text{ AND } 255.255.255.0) == 40.0.0.0$$

Por tanto, el paquete se reenviará al destino, que es un vecino (es una entrega directa) a través de la interfaz `e1`. El procesamiento del paquete termina y el router procede con el siguiente paquete.

Veamos un fragmento de pseudocódigo que ilustra el modo en que el router procesa la tabla de encaminamiento por cada paquete que llega a través de cualquiera de sus interfaces:

```
destino = datagrama.ip_destino
for fila in tabla_de_encaminamiento:
    if (destino AND fila.máscara) == fila.destino:
        reenviar(datagrama, fila.siguiente_salto)
        return

if router_por_defecto:
    reenviar(datagrama, router_por_defecto)
else:
    enviar_ICMP(destino_inalcanzable, datagrama.origen)
```

8.3.1. Analizando la tabla de ejemplo

Estudiemos en detalle la Tabla 8.1 que corresponde con un hipotético router `R0` que podría ser perfectamente realista. Veamos la información que ofrece. La primera columna se ha añadido únicamente para numerar las filas y referirnos a ellas.

- La tabla corresponde a un router que dispone de 3 interfaces: `e0`, `e1` y `s0`. Aunque cada fabricante y SO sigue su propia nomenclatura para nombrar las interfaces, las Ethernet tienen nombres como `e0` o `eth1` y las interfaces serie `s0` o bien con el protocolo de enlace que utilizan, como `ppp1`.
- Las filas 1 y 2 corresponden a entrega directa. Se reconocen porque el valor para el campo *next-hop* es la dirección IP nula: `0.0.0.0`. Esto significa que si el destino está en la red `30.0.0.0/24` (accesible por medio de su interfaz `e0`) o en la red `40.0.0.0/24` (a través de su interfaz `e1`), el router y el destino son vecinos y el propio router se encarga de entregar el paquete.
- Las filas 3 y 4 especifican entregas indirectas. La 3 indica que todo paquete dirigido a la red `130.10.20.0/24` debe reenviarse al router `30.0.0.2`. La fila 4 que todo paquete dirigido a la red `120.20.30/17` debe reenviarse al encaminador `40.0.0.3`.
- Por último, la fila 5 establece un router por defecto. Cualquier paquete que no haya satisfecho ninguna de las otras filas, será enviado al router `50.1.1.2`.

Es importante señalar que la tabla de encaminamiento no dice nada sobre dónde están las redes `130.10.20.0/24` y `210.20.30.0/17` ni a cuántos saltos se encuentran. La tabla solo dice que para llegar a las redes *distantes*, se debe delegar la entrega al router vecino indicado. Pero hay parte de la

topología que podemos deducir (ver Figura 8.3), la que involucra a las redes *locales*, es decir, de las que `R0` es vecino.

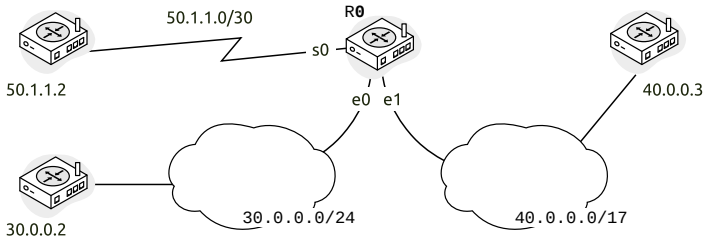

Figura 8.3: Topología (parcial) que se deduce de la Tabla 8.1

8.3.2. Tabla de nodo final

Cualquier nodo o dispositivo conectado a Internet (incluso un *smart phone* o una consola de videojuegos) tiene una tabla de encaminamiento, aunque eso no lo convierte en router. Como vimos, para comportarse como router debe ser capaces de hacer reenvío (ver § 8.1).

La tabla de encaminamiento indica al nodo final cómo enviar tráfico a sus vecinos y cómo enviar tráfico fuera del enlace (una LAN normalmente). Una tabla de encaminamiento típica de un nodo en una red doméstica con enlace WiFi[3] será muy parecida a la siguiente:

dst	mask	next hop	iface
192.168.0.0	255.255.255.0	0.0.0.0	wlan0
0.0.0.0	0.0.0.0	192.168.0.1	wlan0

¿Qué significa esta tabla?

- La primera fila dice dos cosas: quiénes son los vecinos (los conectados a la red `192.168.0.0/24`), y que para llegar a ellos el nodo debe hacer una entrega directa (*next-hop* = `0.0.0.0`).
- La segunda fila indica el router por defecto, que identifica un router de la red domestica (`192.168.0.1`); y a través de éste, el sistema enviará tráfico hacia Internet. Los sistemas operativos de escritorio suelen denominar a este router «pasarela de enlace».

En un sistema GNU/Linux, puedes ver la tabla de encaminamiento con el comando `ip route`. Como probablemente tu computador es un PC conectado a una red doméstica, verás algo muy similar a lo siguiente:

[3]Suele estar detrás de un router NAT. Más adelante veremos qué implica eso.

```
$ ip route
default via 192.168.8.1 dev wlp1s0 proto dhcp metric 600
192.168.0.0/24 dev wlp1s0 proto kernel scope link metric 600
```

Aunque el formato es bastante espartano, obviando los datos adicionales, podemos extraer la siguiente información:

dest/mask	next hop	iface
default	192.168.0.1	wlp1s0
192.168.0.0/24	scope link	wlp1s0

Que se corresponde directamente a esa tabla básica de la que hablábamos justo antes. Las diferencias más significativas son:

- Las columnas «destino» y «máscara» aparecen como un solo dato en notación CIDR: *destino/máscara*.
- Como destino *nulo*, en lugar de `0.0.0.0`, muestra *default* (por *default router*).
- Para el siguiente salto, en lugar de `0.0.0.0` muestra *scope link*, que significa que esos dispositivos (`192.168.0.0/24`) son accesibles en el «ámbito del enlace», es decir, son vecinos.

La designación *proto dhcp* indica que esa entrada se ha creado a partir de información recibida por medio del servicio DHCP, mientras que *proto kernel* indica que ha sido creada por el SO, probablemente cuando se activó la interfaz.

Con *metric* se indica la prioridad. Si hay varias filas aplicables para un mismo paquete, se elige la que tenga menor valor de prioridad. Si la prioridad también coincide, en el caso de GNU/Linux, el SO aplica un algoritmo de balanceo de carga llamado ECMP que puede tener en cuenta cosas como la carga de la interfaz o la caché (prioriza la última usada). Un router dedicado, normalmente implementado como un ASIC o FPGA puede utilizar los criterios que considere el fabricante o su configuración específica.

8.4. Agregación

En ocasiones una tabla de encaminamiento puede contener filas que especifican redes que comparten un prefijo y son contiguas. En estos casos, es posible realizar *agregación* (*summarization*) de esos destinos. Por ejemplo, supongamos la topología de la Figura 8.4.

Siendo la tabla de encaminamiento de `R1`:

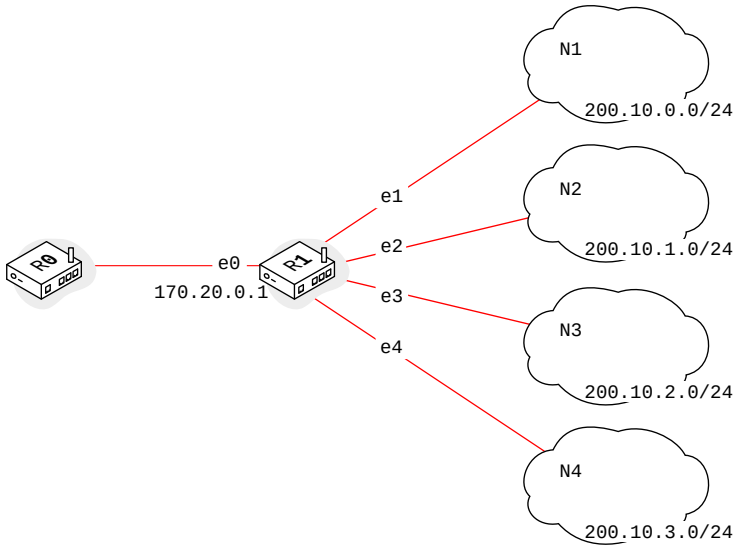

FIGURA 8.4: Topología con redes agregables

dst/mask	next hop	iface
200.10.0.0/24	0.0.0.0	e0
200.10.1.0/24	0.0.0.0	e1
200.10.2.0/24	0.0.0.0	e2
200.10.3.0/24	0.0.0.0	e3
170.20.0.0/24	0.0.0.0	e4

La tabla de `R0` en principio debería incluir también una fila para llegar a cada una de esas redes:

dst/mask	next hop	iface
200.10.0.0/24	170.20.0.1	e0
200.10.1.0/24	170.20.0.1	e0
200.10.2.0/24	170.20.0.1	e0
200.10.3.0/24	170.20.0.1	e0
170.20.0.0/24	0.0.0.0	e0

Pero una mejor opción sería «agregar» esas 4 filas en una sola, es decir, puede indicar que para llevar el paquete a una hipotética red `200.10.0.0/22` se debe enviar a `R1`. La tabla de `R0` quedaría así:

dst/mask	next hop	iface
200.10.0.0/22	170.20.0.1	e0
170.20.0.0/24	0.0.0.0	e0

Esto reduce el tamaño de la tabla de `R0` (y de todos a los que llegue este destino) sin perder información. Conlleva un aumento de rendimiento y una mejora de la escalabilidad de la red. Lo podemos ver como una especie de *supernetting* con la diferencia de que esta red agregada no existe en ningún sentido, salvo a afectos de encaminamiento.

8.5. Un par de ejemplos detallados

Sobre la topología de ejemplo (Figura 8.1) veamos cómo funciona el encaminamiento con un par de escenarios concretos. Analizamos con todo detalle el proceso que sigue un paquete IP desde su creación en el origen hasta alcanzar su destino, incluyendo los cambios en su encapsulación a lo largo de la ruta.

Obviamente hace falta conocer las tablas de encaminamiento de routers y nodos. Las de `R1` y `R2` aparecen en la Figura 8.2.

dst/mask	next hop	iface
20.0.1.0/24	0.0.0.0	e0
22.0.2.0/24	20.4.4.2	ppp0
26.0.3.0/24	20.4.4.2	ppp0
20.4.4.2/32	0.0.0.0	ppp0
20.4.5.0/30	0.0.0.0	e1
0.0.0.0	20.4.5.1	e1

dst/mask	next hop	iface
22.0.2.0/24	0.0.0.0	e0
26.0.3.0/24	0.0.0.0	e1
0.0.0.0/0	20.4.4.1	ppp0
20.4.4.1/32	0.0.0.0	ppp0

CUADRO 8.2: Tablas de encaminamiento de `R1` y `R2`

Las demás tablas de encaminamiento son tablas básicas típicas que suelen tener los nodos finales como la que hemos visto en § 8.3.2. Por ejemplo, la tabla de `PC1` sería algo como:

dst/mask	next hop	iface
20.0.1.0/24	0.0.0.0	e0
0.0.0.0	20.0.1.1	e0

8.5.1. Escenario 1: entrega local

Supongamos que `PC1` envía un paquete IP a `PC2`, por ejemplo, ejecutando `ping -c1 20.0.1.6`. El proceso implica los siguientes pasos:

1. `PC1`[4] construye el mensaje ICMP y lo encapsula en un paquete IP.

[4]Nombramos los nodos, pero lógicamente estas tareas las realizan sus SO.

IP	ver+IHL 0x45 | tos 0 | len 0x0054 id 0x1C46 | flags+offset 0x0000 ttl 64 | proto 1 (ICMP) | cksum src 20.0.1.5 | dst 20.0.1.6
ICMP	type 8 (Echo Request) | code 0 | cksum id 0x1234 | seq 1

2. PC1 consulta su tabla de encaminamiento. La primera fila es aplicable (20.0.1.6 AND 255.255.255.0 == 20.0.1.0), de modo que realiza entrega directa.
3. Como la interfaz de salida es Ethernet, PC1 debe encapsular el paquete en una trama Ethernet, pero desconoce la dirección MAC de PC2.
4. Para averiguarla, envía una petición ARP a la dirección broadcast, por lo que llegará a todos los nodos de la red. La petición ARP pregunta «¿Quién tiene la dirección IP 20.0.1.6?». La cabecera ARP incluye esencialmente lo siguiente:
   - IP origen: 20.0.1.5, IP destino: 20.0.1.6.
   - MAC origen: :A2, MAC destino: 00:00:00:00:00:00.

   El mensaje completo sería:

Ethernet	dst FF:FF:FF:FF:FF:FF | src :A2 | type 0x0806 (ARP)
ARP	hw 0x0002 (eth) | proto 0x0800 (IP) hw-len 6 | proto-len 4 | op 0x0001 sender :A2 | 20.0.1.5 target 00:00:00:00:00:00 | 20.0.1.6

5. PC2 recibe la petición ARP y responde con su dirección MAC (:A3) en una respuesta ARP.

Ethernet	dst :A2 | src :A3 | type 0x0806 (ARP)
ARP	hw 0x0002 | proto 0x0800 hw-len 6 | proto-len 4 | op 0x0002 sender :A3 | 20.0.1.6 target :A2 | 20.0.1.5

6. PC1 puede ahora construir y enviar el paquete IP encapsulado en otra trama Ethernet.

Ethernet	dst :A3 | src :A2 | type 0x0800 (IP)
IP	ver+IHL 0x45| tos 0 | len 0054 id 0x1C46 | flags+offset 0x0000 ttl 64 | proto 1 (ICMP) | cksum src 20.0.1.5 | dst 20.0.1.6
ICMP	type 8 (Echo Request) | code 0 | cksum id 0x1234 | seq 1

Como ya dice el título de la sección, se trata de una entrega **directa** dado que PC1 y PC2 son vecinos, pero en todo caso a la vista del mensaje es fácil de comprobar. En una entrega directa ambas direc-

ciones destino, tanto física (`:A3`) como lógica (`20.0.1.6`) corresponden al nodo destino (`PC2`).

7. La trama llega a `PC2`, que la desencapsula y procesa el paquete IP.
8. Para responder al mensaje `ping`, `PC2` crea una respuesta *Echo Reply* y la envía a `PC1` aplicando el mismo procedimiento que se acaba de describir, solo que a la inversa.

IP	ver+IHL `0x45`| tos 0 | len `0054` id `0x1C46` | flags+offset `0x0000` ttl 64 | proto 1 (ICMP) | cksum src `20.0.1.6` | dst `20.0.1.5`
ICMP	type 0 (Echo Reply) | code 0 | cksum id `0x1234` | seq 1

8.5.2. Escenario 2: dos saltos

Probemos algo un poco más interesante. Supongamos que `PC1` ejecuta ahora `ping -c1 26.0.3.3`. En este caso, el paquete debe atravesar 2 routers y 3 redes hasta `PC5`.

1. `PC1` construye el paquete IP con las direcciones correspondiente:

IP	ver+IHL `0x45`| tos 0 | len `0054` id `2D57` | flags+offset `0x0000` ttl 64 | proto 1 (ICMP) | cksum src `20.0.1.5` | dst `26.0.3.3`
ICMP	type 8 (Echo Request) | code 0 | cksum id `0x5678` | seq 1

2. `PC1` consulta su tabla de encaminamiento. Ninguna fila convencional es aplicable a la dirección destino, pero hay un router por defecto (`20.0.1.1`), así que realiza una entrega indirecta hacia él.
3. Como antes, necesita averiguar la dirección MAC de la interfaz *e0* del router, para lo cual envía una petición ARP con las siguientes direcciones:
   - IP origen: `20.0.1.5`, IP destino: `20.0.1.1` (`R1`).
   - MAC origen: `:A2`, MAC destino: `00:00:00:00:00:00`.
4. `R1` responde con la dirección MAC (`:AE`) en una respuesta ARP.
5. `PC1` puede ahora construir y enviar el paquete IP encapsulado en otra trama Ethernet.

Ethernet	dst :AE | src :A2 | type `0x0800` (IP)
IP	ver+IHL `0x45` | tos 0 | len `0054` id `2D57` | flags+offset `0x0000` ttl 64 | proto 1 (ICMP) | cksum src `20.0.1.5` | dst `26.0.3.3`
ICMP	type 8 (Echo Request) | code 0 | cksum id `0x5678` | seq 1

En este caso se puede comprobar que la dirección física destino (`:AE`) corresponde a `R1` mientras que la lógica (`26.0.3.3`) corresponde a `PC5`, es decir, se trata de una entrega **indirecta**.

6. La trama llega a `R1`, que la desencapsula y procesa el paquete IP.
7. Como el paquete no es para él (`26.0.3.3` $\neq$ `20.0.1.1`) procede a reenviarlo. Consulta su tabla de encaminamiento. La única fila aplicable es la tercera (`26.0.3.3` AND `255.255.255.0` == `26.0.3.0`) de modo que realiza una nueva entrega indirecta a `R2`.
8. El enlace `R1`–`R2` es serie y utiliza una encapsulación PPP. En este caso no se utiliza ARP porque no hay otro destino posible que el otro extremo del enlace. Tampoco hay direcciones MAC en este tipo de interfaz. La trama PPP será algo como:

PPP	addr 0xFF | control 0x03 | proto 0x0021 (IP)
IP	ver+IHL `0x45`| tos:0 | len `0054` id `2D57` | flags+offset `0x0000` ttl 64 | proto 1 (ICMP) | cksum src `20.0.1.5` | dst `26.0.3.3`
ICMP	type 8 | code 0 | cksum id `0x5678` | seq 1

9. `R2` recibe la trama PPP, la desencapsula y procesa el paquete IP. Tampoco es para él (`26.0.3.3` $\neq$ `20.4.4.2`), así que lo reenvía. La segunda fila de su tabla de encaminamiento es aplicable, e indica una entrega directa por la interfaz `e1`.
10. `R2` tiene que averiguar la dirección MAC del destino. Envía una petición ARP preguntando por `26.0.3.3` y `PC5` responde con su dirección: `:C4`.
11. `R2` puede ahora construir y enviar el paquete IP en una trama Ethernet con las siguientes direcciones:
    - MAC origen: `:C3`, MAC destino: `:C4`.
12. `PC5` recibe la trama, desencapsula el paquete IP y a su vez el mensaje ICMP. Como es un *Echo request*, procede a construir una respuesta *Echo reply* dirigida a `PC1`, que recorrerá el camino en sentido contrario.

8.6. Mensajes de control de interred (ICMP)

Ya hemos hablado de ICMP en § 5.7, y lo hemos estado utilizando a menudo con el comando `ping`, que utiliza uno de los tipos de mensajes ICMP, y en este mismo capítulo hemos visto otros usos. En esta sección veremos con más detalle los distintos mensajes y su finalidad [11]. Primero recordemos el formato general del mensaje:

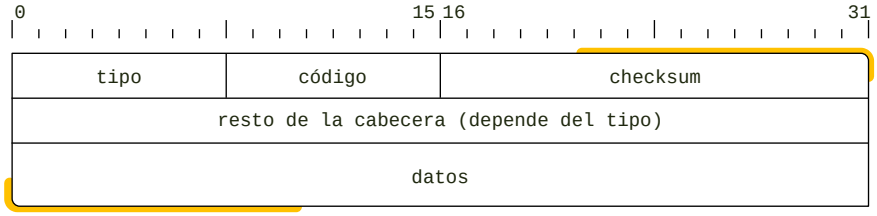

FIGURA 8.5: Formato del mensaje ICMP

Hay dos grupos de mensajes ICMP: los de notificación de errores y los de consulta/diagnóstico. El campo `tipo` indica el tipo de mensaje mientras que el campo `código` indica un subtipo. Veremos a continuación los más comunes.

Los mensajes de notificación de errores son enviados cuando un router detecta un problema en el encaminamiento o entrega de un paquete[5]. Estos mensajes son enviados siempre al nodo origen del paquete que ha causado el problema. Es importante entender que el objetivo de estos mensajes es meramente informativo. IP es un protocolo esencialmente no confiable e ICMP no pretende cambiar eso. Del mismo modo que no hay garantías de que un paquete IP pueda ser entregado, tampoco las hay para un mensaje ICMP, de hecho los mensajes ICMP se encapsulan sobre paquetes IP. Algunos de los mensajes de notificación son los siguientes (se muestra entre corchetes el valor del campo `tipo`):

- Destino inalcanzable [3]
- Tiempo excedido [11]
- Problema en parámetro [12]
- Supresión al origen [4]
- Redirección [5]

Los mensajes ICMP de error incluyen como carga útil al menos la cabecera IP del paquete problemático y los primeros 8 bytes de la carga útil de aquel, lo que normalmente corresponderá a los puertos origen y destino de un mensaje TCP o UDP que este estuviera transportando en el momento de producirse el error. Esta información es muy útil para que el SO del nodo origen determine qué proceso es el responsable y puede informar al programador mediante un error o excepción, para que él pueda identificar y tratar el problema.

Nunca se generan mensajes ICMP cuando el paquete problemático porta un mensaje ICMP de notificación de error. Esto es porque informar de un

[5]Ya hemos visto algunos casos en este mismo capítulo.

problema de entrega podría generar a su vez otro problema de entrega, provocando un bucle. Como ICMP Echo-request no es un mensaje de error, por regla general sí se envían mensajes ICMP de error si hay problemas al entregarlo.

Los mensajes de consulta/diagnóstico, como su nombre indica, tratan de averiguar cierta información o comprobar el estado. Por eso estos mensajes vienen por pares petición-respuesta. Son los siguientes:

- Ping (petición [8]/respuesta [0])
- Marca de tiempo (petición [13]/respuesta [14])
- Información (petición [15]/respuesta [16])
- Máscara de subred (petición [17]/respuesta [18])
- Router (solicitud [10]/anunciamiento [9])

Estudiémoslos uno a uno.

8.6.1. Destino inalcanzable *(Destination unreachable)*

Es una notificación muy frecuente. Un router envía un mensaje *Destino inalcanzable* (`tipo`=3) cuando no puede entregar un paquete a su destino. Hay varios motivos por lo que eso puede ocurrir, y corresponden con los distintos valores para el campo `código`:

0. Red inalcanzable (*network unreachable*). Lo utiliza el router si no encuentra una fila en su tabla de encaminamiento que pueda utilizar para reenviar el paquete.
1. Nodo inalcanzable (*host unreachable*). El router tiene acceso a la red destino, pero no puede efectuar la entrega directa.
2. Protocolo inalcanzable (*protocol unreachable*). El destino no soporta el protocolo encapsulado en el paquete IP.
3. Puerto inalcanzable (*port unreachable*). El puerto destino en el nodo destino está cerrado.
4. Fragmentación necesaria (*fragmentation needed and DF set*). El paquete necesita ser fragmentado, pero el origen ha solicitado que no se fragmente (ver § 8.7).
5. Fallo en la ruta en origen (*source route failed*). No se ha podido aplicar la opción *source route* de IP[6].

Es sencillo reproducir algunos de estos errores porque tu propio sistema los genera en caso de problemas cuando intenta enviar tráfico. Puedes provocar un mensaje *nodo inalcanzable* enviando `ping` a una dirección IP que sabes que no está asignada a ningún nodo de la red. En el siguiente ejemplo

[6] *Source route* es una función de diagnostico de IP con un uso muy marginal.

asumimos que la IP de tu nodo es `192.168.0.37/24` y la del nodo que no existe es `192.168.0.66/24`. Cambia estas direcciones por las adecuadas en tu caso y pruébalo por ti mismo.

```
$ ping -c 1 192.168.0.66
PING 192.168.0.66 (192.168.0.66) 56(84) bytes of data.
From 192.168.0.37 icmp_seq=1 Destination Host Unreachable
```

Si capturas el tráfico verás el mensaje ICMP en detalle. Como siempre eliminamos los campos irrelevantes para el propósito que nos ocupa:

```
1  $ sudo tshark -i any -f icmp -V
2  Internet Protocol Version 4, Src: 192.168.0.37, Dst: 192.168.0.37
3      Total Length: 112
4      Protocol: ICMP (1)
5      Source Address: 192.168.0.37
6      Destination Address: 192.168.0.37
7  Internet Control Message Protocol
8      Type: 3 (Destination unreachable)
9      Code: 1 (Host unreachable)
10     Internet Protocol Version 4, Src: 192.168.0.37, Dst: 192.168.0.66
11         Total Length: 84
12         Protocol: ICMP (1)
13         Source Address: 192.168.0.37
14         Destination Address: 192.168.0.66
15     Internet Control Message Protocol
16         Type: 8 (Echo (ping) request)
17         Code: 0
```

Fíjate que el mensaje ICMP se lo envía el sistema a sí mismo (`192.168.0.37`) porque ha sido el propio sistema el que ha detectado el problema. También puedes ver el mensaje ICMP Echo-request (con su cabecera IP) encapsulado en el mensaje de error (**líneas 10-17**).

El programador puede acceder a esta información desde el código. Así pues, un intento de conexión a un nodo inexistente eleva una excepción `socket.error` como consecuencia directa de que el SO reciba y procese el mensaje ICMP. Puedes verlo en el Listado 8.1. Al ejecutar este programa verás un mensaje similar a «[Errno 113] No route to host».

```
import socket

try:
    sock = socket.socket()
    sock.connect(('192.168.0.66', 1234))
except socket.error as e:
    print(e)
```

LISTADO 8.1: Captura del error «nodo inalcanzable» con Python

8.6.2. Tiempo excedido *(Time exceeded)*

Recuerda que los routers decrementan el campo TTL cuando procesan un paquete IP. Pues bien, si el valor de TTL llega a cero, el router descarta el paquete y envía un mensaje ICMP de este tipo con `código`=0, que significa formalmente *tiempo de vida excedido en tránsito* (*time to live exceeded in transit*).

Los nombres tanto del error como del campo de la cabecera IP *time to live* (tiempo de vida) al que está asociado, son una reminiscencia de la Internet primigenia cuando los routers tardaban aproximadamente un segundo en procesar cada paquete. El campo TTL expresaba (en segundos) la vida que aún le quedaba al paquete. Hoy en día hay routers que pueden procesar millones de paquetes por segundo y el significado práctico del campo TTL ahora es el número de saltos que el paquete tiene permitido dar aún. En esta línea y para ser coherente con el significado actual, en el diseño de IPv6 el campo equivalente pasó a llamarse *hop-count* (cuenta de saltos).

Si el campo `código`=1, el mensaje lo envía el nodo destino para indicar que ha agotado el tiempo límite en espera de recibir todos los fragmentos de un paquete. En este caso significa *tiempo excedido en el reensamblado* (*time to live exceeded in reassembly*). El valor recomendado para ese tiempo límite es de 60 segundos [9], aunque por ejemplo en GNU/Linux es configurable mediante el parámetro del núcleo `net/ipv4/ipfrag_time`.

traceroute

El programa `traceroute` aprovecha el mecanismo que acabamos de describir para descubrir la ruta que sigue un paquete IP hacia su destino. `traceroute` comienza enviando tres mensajes ICMP *Echo*, o un segmento UDP o TCP encapsulado en un paquete IP con TTL=1. Eso provoca que el router local descarte el paquete, informe del error y con ello revele su dirección IP (porque aparece como dirección origen del mensaje de error). A continuación, `traceroute` repite el envío con TTL=2, obteniendo la dirección del segundo router en la ruta, y así sucesivamente hasta alcanzar el nodo destino.

A continuación aparece el resultado de una ejecución de `traceroute` dirigida a `rediris.es` desde la ESI de Ciudad Real:

```
1 $ traceroute rediris.es
2 traceroute to rediris.es (130.206.13.20), 30 hops max, 60 byte packets
3  1  161.67.101.1 (161.67.101.1)  0.694 ms  1.015 ms  1.256 ms
4  2  172.16.160.5 (172.16.160.5)  1.153 ms  1.465 ms  1.710 ms
5  3  ro-vlan170.ctic.cr.red.uclm.es (172.16.160.22)  0.514 ms  0.517 ms  0.571 ms
6  4  GE1-2-0.EB-CiudadReal0.red.rediris.es (130.206.200.1)  0.803 ms  0.939 ms  1.073 ms
7  5  CLM.SO1-1-1.EB-IRIS4.red.rediris.es (130.206.250.133)  4.529 ms  4.568 ms  4.656 ms
```

```
8   6  XE3-0-0-264.EB-IRIS6.red.rediris.es (130.206.206.133)  4.806 ms  4.318 ms  4.332 ms
9   7  www.rediris.es (130.206.13.20)  4.456 ms  4.465 ms  4.518 ms
```

El primer resultado es para un TTL=1 (**línea 3**) y corresponde al router local de esa LAN (`161.67.101.1`). Al lado aparecen los tiempos RTT para cada uno de los tres paquetes enviados. Para cada router aparece primero su nombre (si tiene) y después su dirección IP. Como se puede apreciar la ruta completa requiere 6 saltos, es decir, el paquete atraviesa 6 routers. La última entrada es el destino solicitado.

8.6.3. Problema en parámetro *(Parameter problem)*

Este mensaje de error lo envía un router o un nodo destino cuando detecta un valor incorrecto en la cabecera IP que le impide procesarla. El mensaje incluye un campo `puntero` que indica el campo de la cabecera en el que ha encontrado el problema.

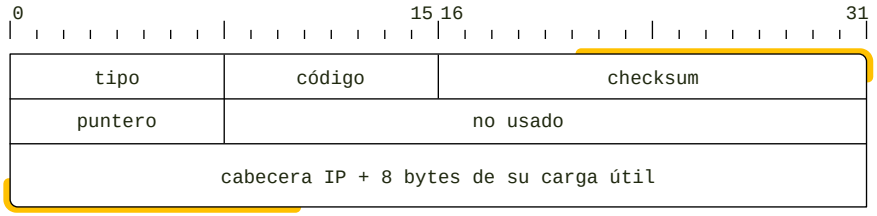

FIGURA 8.6: Mensaje ICMP de problema en parámetro

8.6.4. Supresión al origen *(Source quench)*

Este mensaje notifica un problema de exceso de tráfico, que puede estar relacionado con un problema de congestión si lo envía un router o como mecanismo de control de flujo si lo envía el destino. En todo caso, este mecanismo está obsoleto, no se implementa en sistemas modernos y no se recomienda su uso [15]. Trataremos este tema con más detalle en § 13.4.

8.6.5. Redirección *(Redirect)*

Este mensaje lo envía un router para indicar a un nodo origen que hay una ruta mejor para llegar a su destino. El mensaje incluye la dirección IP del router alternativo. Este mensaje no es técnicamente un error, es más bien informativo. En todo caso, el uso de este mensaje también está obsoleto porque podría ser explotado por un atacante malicioso en una técnica MITM.

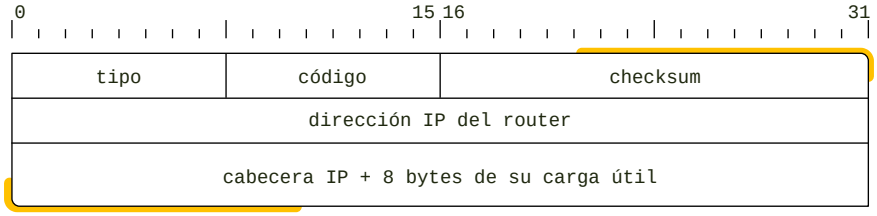

Figura 8.7: Mensaje ICMP de redirección

Existe un parámetro del núcleo `net/ipv4/conf/all/accept_redirects` que permite configurar el comportamiento de los mensajes de redirección (por defecto se ignoran).

8.6.6. Ping *(Echo)*

Ya hemos hablado bastante de `ping` y lo hemos utilizado en varias ocasiones, incluso hemos visto algunas capturas (ver § 5.7) de los mensajes ICMP que genera. El programa `ping` envía un mensaje ECHO request (`tipo`=8, `código`=0) a un nodo remoto. Éste al recibirlo envía de vuelta un mensaje ECHO reply (`tipo`=0, `código`=0).

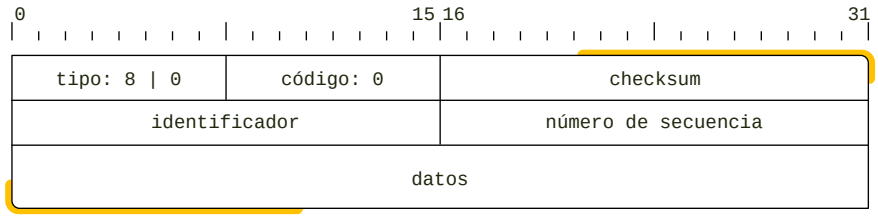

Figura 8.8: Mensaje ICMP Echo

El mensaje tiene un campo `identificador` y un campo `número de secuencia`. En una determinada ejecución del programa `ping` todos los mensajes de petición utilizan el mismo identificador y cada mensaje lleva un número de secuencia creciente empezando en 1. Aquí puedes ver una captura del comando `ping example.net`:

```
$ sudo tshark -f icmp
Capturing on 'eno1'
1 192.168.0.37 → 104.18.5.106 ICMP 98 Echo (ping) request  id=0x0002, seq=1/256,  ttl=64
2 104.18.5.106 → 192.168.0.37 ICMP 98 Echo (ping) reply    id=0x0002, seq=1/256,  ttl=57
3 192.168.0.37 → 104.18.5.106 ICMP 98 Echo (ping) request  id=0x0002, seq=2/512,  ttl=64
4 104.18.5.106 → 192.168.0.37 ICMP 98 Echo (ping) reply    id=0x0002, seq=2/512,  ttl=57
5 192.168.0.37 → 104.18.5.106 ICMP 98 Echo (ping) request  id=0x0002, seq=3/768,  ttl=64
6 104.18.5.106 → 192.168.0.37 ICMP 98 Echo (ping) reply    id=0x0002, seq=3/768,  ttl=57
```

Ese campo `id` hace posible dos o más ejecuciones de `ping` con el mismo origen y destino. Recuerda que ICMP se encapsula directamente sobre IP, aquí no hay puertos que permitan multiplexar[7]. El segundo número tras '`/`' en el campo `seq` es el mismo valor, pero expresado en *little endian*.

La carga útil del mensaje de petición tiene por defecto una longitud de 56 bytes. Los primeros 16 bytes suelen ser una marca de tiempo y después 40 bytes de datos aleatorios o consecutivos. Puedes comprobarlo con una captura:

```
$ sudo tshark -i lo -f icmp -V
Internet Control Message Protocol
  Type: 8 (Echo (ping) request)
  Code: 0
  Timestamp from icmp data: Apr 21, 2025 18:38:03.966021000 CEST
  Data (40 bytes)

0000  10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f   ................
0010  20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f    !"#$%&'()*+,-./
0020  30 31 32 33 34 35 36 37                          01234567

Data: 101112131415161718191a1b1c1d1e1f202122232425262728292a2b2c2d2e2f3031323334353637
```

Realmente no hay un campo *timestamp* en el mensaje ICMP, pero es tan habitual que la carga útil sea un *timestamp*, que `tshark` así lo interpreta. Puedes ver que los siguientes bytes empiezan con el valor `0x10` y van creciendo a partir de ahí hasta `0x37`.

El mensaje Echo reply que envía el receptor debe enviar de vuelta la misma carga útil que le envió el emisor —por eso se llama *eco*. El emisor puede entonces tomar la hora de llegada y restar la que él mismo envió en el mensaje de petición. Con ello obtiene el lapso de tiempo que ha transcurrido entre la petición y la respuesta. Ese es el tiempo de ida y vuelta o RTT (Round-Trip Time), que muestra la salida del programa con la etiqueta *time* el imprimir cada respuesta.

```
~$ ping -c2 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=117 time=4.25 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=117 time=4.27 ms

--- 8.8.8.8 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 4.481/4.526/4.571/0.045 ms
```

[7] distinguir entre procesos

8.6.7. Marca de tiempo *(Timestamp)*

El emisor envía su marca de tiempo en el mensaje de petición (`tipo`=13). El receptor envía un mensaje de respuesta (`tipo`=14) con tres marcas de tiempo:

- `origen`: La marca de tiempo que contenía el mensaje de petición.
- `recepción`: El instante en que llegó el mensaje de petición.
- `transmisión`: El instante en que envió el mensaje de respuesta.

Como el mensaje Echo, incluye un campo *identificador* y *número de secuencia* para que el emisor pueda realizar la correspondencia entre petición y respuesta.

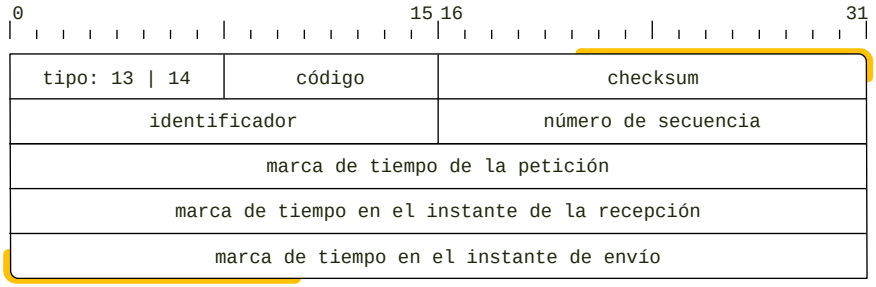

Figura 8.9: Mensaje ICMP Timestamp

Puedes enviar un mensaje de marca de tiempo con el programa `hping3`, incluido en el paquete del mismo nombre. A continuación puedes ver un ejemplo:

```
~$ sudo hping3 -c 4 --icmp --icmp-ts  example.net
HPING example.net (eno1 23.215.0.141): icmp mode set, 28 headers + 0 data bytes
len=46 ip=23.215.0.141 ttl=50 id=42928 icmp_seq=0 rtt=91.9 ms
ICMP timestamp: Originate=63921874 Receive=63921915 Transmit=63921915
ICMP timestamp RTT tsrtt=92

len=46 ip=23.215.0.141 ttl=50 id=43150 icmp_seq=1 rtt=91.8 ms
ICMP timestamp: Originate=63922874 Receive=63922915 Transmit=63922915
ICMP timestamp RTT tsrtt=92

len=46 ip=23.215.0.141 ttl=50 id=43869 icmp_seq=2 rtt=91.7 ms
ICMP timestamp: Originate=63923875 Receive=63923915 Transmit=63923915
ICMP timestamp RTT tsrtt=91

len=46 ip=23.215.0.141 ttl=50 id=44269 icmp_seq=3 rtt=91.6 ms
ICMP timestamp: Originate=63924875 Receive=63924915 Transmit=63924915
ICMP timestamp RTT tsrtt=91

--- example.net hping statistic ---
4 packets transmitted, 4 packets received, 0% packet loss
```

```
round-trip min/avg/max = 91.6/91.7/91.9 ms
```

Lo que vemos a la salida son los mensajes de respuesta, que incluyen las tres marcas: `Originate`, `Receive` y `Transmit`. Las marcas `Receive` y `Transmit` son idénticas porque el tiempo que transcurre entre la recepción de la petición y el envío de la respuesta es despreciable.

El mensaje ICMP Timestamp permite calcular un RTT preciso sólo si los relojes de ambos nodos están sincronizados o el desfase (sesgo) es conocido. Como esto es poco habitual, un uso que puede ser interesante es precisamente para realizar una sincronización o medir el sesgo. El emisor puede combinar el mensaje Echo para estimar y corregir el sesgo de su reloj, y de ese modo sincronizar su reloj con el del destino.

8.6.8. Información *(Information)* y máscara *(Address mask)*

Los mensajes de petición/respuesta de información (tipos 15 y 16), y petición/respuesta de máscara (tipos 17 y 18) constituían un mecanismo rudimentario para que un nodo pudiera averiguar la dirección IP o máscara asignada a ese nodo. Están obsoletos [16] y no se utilizan en la actualidad. El protocolo DHCP suple con creces esta necesidad.

8.6.9. Solicitud y anunciamiento de router

Estos mensajes forman parte del protocolo IRDP (ICMP Router Discovery Protocol) [17], y como sus nombres indican, sirven respectivamente para que un nodo pueda solicitar routers locales para poder salir de la LAN y para que un router pueda anunciare, ya sea como respuesta al mensaje de solicitud o de forma proactiva para informar de su presencia.

Como podrás suponer, es una funcionalidad en desuso ya que el protocolo DHCP cubre esta necesidad con creces. Puedes ver información detallada sobre configuración de nodos con DHCP en § 10.2.

8.7. Fragmentación

La función principal del router es interconectar redes, y en muchas situaciones esas redes pueden ser heterogéneas, es decir, estar basadas en tecnologías de enlace diferentes, como el caso de la Figura 8.1.

Uno de los problemas que surge al interconectar redes heterogéneas es la disparidad en los tamaños de trama. En concreto el problema aparece cuando se extrae la carga útil que llega en la trama entrante y se necesita encapsular en una trama de un tamaño menor para ser enviada por una interfaz diferente. Al tamaño máximo (en bytes) que puede transportar una trama

(la carga útil) de una determinada tecnología se le llama MTU (Maximum Transmission Unit).

La MTU es una característica básica inherente a cualquier interfaz de red y tecnología de enlace, es decir, todas la interfaces de una determinada tecnología tienen siempre la misma MTU en cualquier lugar o instalación; es una característica dada por diseño y fabricación. En GNU/Linux lo puedes consultar con el comando `ip link show`. En el siguiente ejemplo puedes ver que la interfaz `eno1` (una NIC Ethernet) tiene una MTU de 1 500 bytes.

```
$ ip link show eno1
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT
  group default qlen 1000
  link/ether f8:5f:2a:c0:ff:ee brd ff:ff:ff:ff:ff:ff
```

Aunque es muy poco habitual y a pesar de lo dicho, el administrador puede cambiar la MTU de una interfaz. Salvo que haya una buena razón no es nada recomendable y desde luego nunca para configurar valores mayores que la MTU real del enlace. Provocaría todo tipo de problemas y errores difíciles de detectar.

```
$ sudo ip link set dev eth0 mtu 1400
```

La discrepancia entre MTUs puede ocurrir, por ejemplo, cuando un paquete IP que viaja encapsulado en una trama Ethernet o WiFi (MTU=1 500 bytes) llega a un router que lo debe enviar por un enlace serie (MTU=256 bytes). La solución a este problema es que el router **fragmente** (trocee) la carga útil del paquete IP original y cree nuevos paquetes IP (fragmentos) que quepan en las tramas del enlace de salida. A este proceso es a lo que se llama «fragmentación».

El problema no termina ahí. En su viaje hacía el destino, los fragmentos podrían tener que atravesar otras redes con MTU aún menores, con lo que el router correspondiente debería dividirlos a su vez en fragmentos más pequeños. Aunque la probabilidad es baja, podría ocurrir además que los fragmentos, que son a todos los efectos paquetes IP individuales, sigan eventualmente rutas distintas. Eso significa que, para un mismo flujo, podrían llegar al destino paquetes completos y fragmentos de distintos tamaños dependiendo de la ruta que ha seguido cada uno.

Por estos dos motivos: fragmentaciones sucesivas y rutas distintas, el «reemsamblado», es decir, la unión de los fragmentos para obtener de nuevo el paquete original, solo lo puede realizar el destinatario. Es el único lugar dónde es seguro que van a pasar todos los fragmentos.

La fragmentación es una tarea costosa que consume tiempo y recursos tanto en los routers como en el destino. El destino debe almacenar en memoria los fragmentos que va recibiendo hasta que llegue el último de ellos, y esto puede estar ocurriendo para varios datagramas en varios flujos al mismo tiempo. En muchas ocasiones es preferible evitar la fragmentación. Eso es posible si el nodo origen crea paquetes lo suficientemente pequeños como para quepan en la menor MTU de la ruta probable, que llamamos PMTU (Path MTU).

La cabecera del paquete IP tiene 4 campos relacionados con la fragmentación. Todos ellos están en la segunda palabra de 32 bis, es decir, la segunda fila tal como se suele representar.

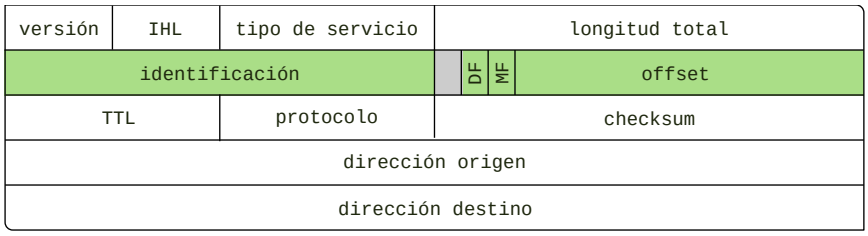

FIGURA 8.10: Campos relacionados con la fragmentación en la cabecera IP

Veamos su significado con más detalle:

- `identificación` *(identification)*: es un número único que identifica el paquete original. Todos los fragmentos creados a partir de un mismo paquete comparten el mismo número de identificación. De este modo, el destinatario puede determinar qué fragmentos corresponden con qué paquete original.
- Flag DF. Si está activo, indica que el paquete no debe ser fragmentado: DF (Don't Fragment).
- Flag MF. Si está activo significa que ese paquete no es el último fragmento, hay más: MF (More Fragments).
- `offset`: Es un número que indica la posición que ocupa la carga útil de ese fragmento respecto al paquete original. El primer fragmento siempre tiene un `offset`=0. Fíjate que la longitud de este campo es de solo 13 bits, mientras que el tamaño del paquete IP se expresa con un entero de 16 bits. Eso es porque este campo está expresado en múltiplos de 8 bytes, lo que lógicamente implica que los fragmentos no pueden tener cualquier tamaño, tiene que ser múltiplo de 8 bytes.

Con esto, se pueden distinguir cuatro situaciones al inspeccionar una cabecera IP:

1. Si `offset`=0 y el flag MF no está activo, se trata de un paquete completo no fragmentado.
2. Si `offset`=0 y el flag MF está activo, el paquete ha sido fragmentado y este es el primer fragmento.
3. Si `offset` es distinto de 0 y el flag MF está activo, el paquete ha sido fragmentado y este es un fragmento intermedio.
4. Si `offset` es distinto de 0 y el flag MF no está activo, el paquete ha sido fragmentado y este es el último fragmento.

Fíjate que cuando un paquete está fragmentado no es posible saber cuántos fragmentos lo componen. El destino los irá recibiendo y almacenando temporalmente, pero no sabrá cuantos son hasta que rellene todos los huecos.

Veamos un ejemplo de fragmentación con la topología de la Figura 8.11.

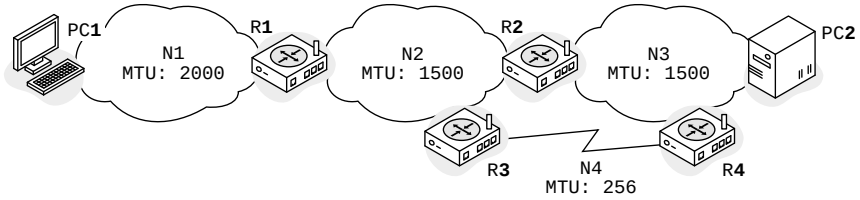

FIGURA 8.11: Topología para el ejemplo de fragmentación

Supongamos que `PC1` envía un paquete IP a `PC5` con una carga útil de 1 980 bytes, lo que implica un tamaño total de 2 000 bytes considerando que incluye una cabecera IP estándar sin opciones (20 bytes). Por regla general (como este caso) todo nodo evita crear paquetes mayores que la MTU de la red a la que está conectado, pues eso le obligaría a fragmentar ya desde el origen.

El valor para los campos relevantes sería el siguiente cuando el paquete sale a la red `N1`:

- Tamaño total: 2 000 bytes.
- `identificación`: `0x1234` (ejemplo).
- Flag DF: 0 (fragmentación permitida).
- Flag MF: 0 (es el último fragmento).
- `offset`: 0 (es el primer fragmento).
- Tamaño de la carga útil: 1 980 bytes.
- Rango de datos: 0-1 979.

Es el primer y último fragmento, es decir, no está fragmentado (el primer caso de la lista anterior). Al llegar a `R1`, éste debe fragmentarlo para poder

enviarlo a la red N2 que requiere una encapsulación con MTU=1 500 bytes, lo que implica que la carga útil máxima es de 1 480 bytes (1 500 - 20). Crea por tanto dos fragmentos:

- *Fragmento 0*
  - Tamaño total: 1 500 bytes.
  - identificación: 0x1234.
  - Flag MF: 1.
  - offset: 0.
  - Tamaño de la carga útil: 1 480 bytes.
  - Rango de datos: 0–1 479.
- *Fragmento 1*:
  - Tamaño total: 520 bytes.
  - identificación: 0x1234.
  - Flag MF: 0.
  - offset: 185 (1 480 / 8).
  - Tamaño de la carga útil: 500 bytes.
  - Rango de datos: 1 480–1 979.

Ahora supongamos que R1 encamina el fragmento 0 por R2, que está conectado a otra red con MTU=1500 (N3), es decir, no requiere más fragmentación y llega tal cual al destino. Sin embargo, R1 encamina el fragmento 1 por R3 y este a su vez hacía R4 hasta el destino. Eso obliga a R3 a fragmentar de nuevo pues el tamaño de este segundo fragmento (520 bytes) supera el MTU del enlace serie N4 (256 bytes). Restando la cabecera, la carga útil del paquete podría tener un máximo de 236 bytes (256 - 20). Pero 236 no es múltiplo de 8, así que el primer fragmento deberá tener 232 bytes de carga útil.

Los campos relevantes de estos nuevos fragmentos serían:

- *Fragmento 1.1*:
  - Tamaño total: 252 bytes.
  - identificación: 0x1234.
  - Flag MF: 1.
  - offset: 185 (1 480 / 8).
  - Tamaño de la carga útil: 232 bytes.
  - Rango de datos: 1 480–1 711.
- *Fragmento 1.2*:
  - Tamaño total: 252 bytes.
  - identificación: 0x1234.
  - Flag MF: 1.
  - offset: 232 (1 712 / 8).
  - Tamaño de la carga útil: 232 bytes.
  - Rango de datos: 1 712–1 943.
- *Fragmento 1.3*:
  - Tamaño total: 56

- `identificación`: `0x1234`.
- Flag MF: 0.
- `offset`: 243 (1 944 / 8).
- Tamaño de la carga útil: 36 bytes.
- Rango de datos: 1 943–1 979.

La Figura 8.12 representa las cabeceras de los fragmentos que se han creado. En la primera fila aparece el tamaño total del paquete (cabecera + carga útil), en la segunda fila los campos relacionados con la fragmentación: `identificador`, flag DF y `offset`, y en la parte inferior la carga útil, en la que se representa un rango que numera los bytes empezando en 0.

Como hemos visto, al entrar el paquete original en la red `N2` se crean los fragmentos 0 y 1, y posteriormente cuando el fragmento 1 entra en la red `N4` se crean a partir de él los fragmentos 1.1, 1.2 y 1.3. Por tanto al destino llegan 4 fragmentos: 0, 1.1, 1.2 y 1.3. Puedes comprobar que los rangos de datos representados en la carga útil son contiguos y coinciden con el paquete original.

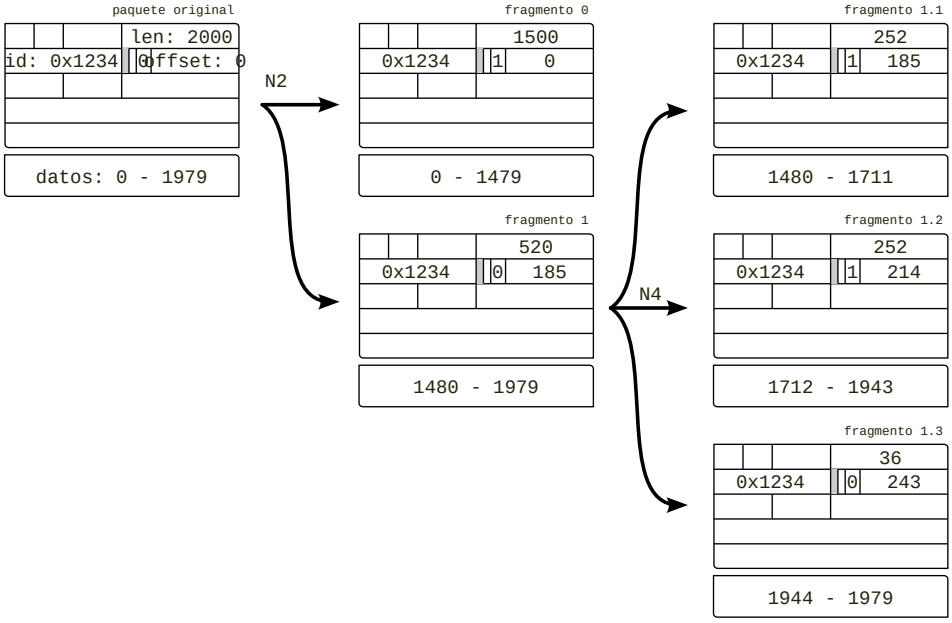

FIGURA 8.12: Ejemplo de fragmentación

La fragmentación también agrava la sobrecarga de cabeceras porque crea paquetes IP adicionales. Hemos pasado de un solo paquete IP a cuatro, es decir, 3 nuevas cabeceras IP, además de las cabeceras de las correspondien-

tes tramas adicionales. Si por ejemplo esas tramas fuesen Ethernet, esta sobrecarga se puede cuantificar en $3 \times (14 + 20) = 102$ bytes.

Llevemos esto a la práctica con un pequeño laboratorio llamado `lab-frag` basado en `docker`. Para ello, vamos a recrear la topología trivial de la Figura 8.13

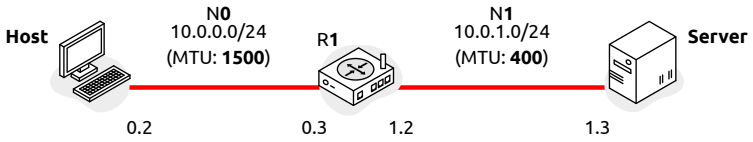

FIGURA 8.13: Topología del laboratorio de la fragmentación /lab-frag

Esta topología está especificada en el archivo `compose.yml` para `docker-compose`.

```yaml
x-common: &common-settings
  privileged: true
  restart: "no"
  ulimits:
    nofile:
      soft: 100000
      hard: 200000

services:
  router:
    build: ./router
    image: lab-frag-router
    container_name: router
    hostname: router
    <<: *common-settings
    networks:
      N0: { ipv4_address: 10.0.0.3 }
      N1: { ipv4_address: 10.0.1.2 }

  server:
    build: ./server
    image: lab-frag-server
    container_name: server
    <<: *common-settings
    networks:
      N1: { ipv4_address: 10.0.1.3 }

networks:
  N0:
    ipam: { config: [{subnet: 10.0.0.0/24}] }
  N1:
    driver_opts: { com.docker.network.driver.mtu: 400 }
    ipam: { config: [{subnet: 10.0.1.0/24}] }
```

LISTADO 8.2: Descripción del laboratorio de fragmentación /lab-frag/compose.yml

Con este tenemos un router y un nodo `Servidor` en la red `N1` que tiene una MTU de 400 bytes. El nodo `host` es tu computador real. Lo único que debes tener en cuenta es que previamente a la ejecución del ejemplo tu dirección de red no esté dentro de `10.0.0.0/8` ni tengas nada en tu tabla de rutas que lleve a esa red. De otro modo tendrías un conflicto y no funcionaría.

En el directorio `lab-frag` ejecuta `make` para arrancar el laboratorio. Una vez en marcha puedes ejecutar `ping` y `traceroute` para comprobar que puedes llegar al nodo `Server`.

```
$ ping -c1 10.0.1.3
PING 10.0.1.3 (10.0.1.3) 56(84) bytes of data.
64 bytes from 10.0.1.3: icmp_seq=1 ttl=63 time=0.049 ms

--- 10.0.1.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.049/0.049/0.049/0.000 ms
$ traceroute
traceroute to 10.0.1.3 (10.0.1.3), 30 hops max, 60 byte packets
 1  10.0.0.3 (10.0.0.3)  0.494 ms  0.409 ms  0.372 ms
 2  10.0.1.3 (10.0.1.3)  0.349 ms  0.316 ms  0.275 ms
```

Este primer *ping* es un mensaje muy pequeño (64 bytes) y no se ha visto afectado por la fragmentación. Ahora puedes enviar un mensaje mucho mayor (700 bytes de carga útil) con `ping -c1 -s700 10.0.1.3` para forzar a `R1` a fragmentar.

Si ejecutas `tshark` en `Server` y vuelves a ejecutar `ping` podrás ver los fragmentos.

```
$ docker exec server tshark -f icmp
Capturing on 'eth0'
1  10.0.0.1 ? 10.0.1.3  IPv4 410 Fragmented IP protocol (proto=ICMP 1, off=0, ID=073c)
2  10.0.0.1 ? 10.0.1.3  ICMP 366 Echo (ping) request  id=0x0008, seq=1/256, ttl=63
3  10.0.1.3 ? 10.0.0.1  IPv4 410 Fragmented IP protocol (proto=ICMP 1, off=0, ID=d41e)
4  10.0.1.3 ? 10.0.0.1  ICMP 366 Echo (ping) reply    id=0x0008, seq=1/256, ttl=64
```

Los mensajes 1 y 2 corresponden a la petición ICMP Echo. En el primer mensaje `tshark` ofrece información del fragmento (`offset` y `identificación`) porque en ese momento obviamente no tiene toda la información. Al llegar el sagundo fragmento ya puede reesamblar y determina que es un mensaje Echo. Los mensajes 3 y 4 son la respuesta que envía `Server`, que necesariamente tiene que ser del mismo tamaño, y por esa razón el propio nodo se ve obligado a fragmentar *en origen*.

Puedes hacer una captura más detallada para ver el valor de los campos de la cabecera IP. Mostramos aquí solo la información relevante de las cabeceras IP de los 2 primeros mensajes (el Echo request).

```
$ docker exec server tshark -f icmp -V
Frame 1:
Internet Protocol Version 4, Src: 10.0.0.1, Dst: 10.0.1.3
    Total Length: 396
    Identification: 0x85b8 (34232)
    001. .... = Flags: 0x1, More fragments
    ...0 0000 0000 0000 = Fragment Offset: 0
    Source Address: 10.0.0.1
    Destination Address: 10.0.1.3

Frame 2:
Internet Protocol Version 4, Src: 10.0.0.1, Dst: 10.0.1.3
    Total Length: 352
    Identification: 0x85b8 (34232)
    000. .... = Flags: 0x0
    ...0 0000 0010 1111 = Fragment Offset: 376
    Source Address: 10.0.0.1
    Destination Address: 10.0.1.3
    [2 IPv4 Fragments (708 bytes): 1(376), 2(332)]
        [Frame: 1, payload: 0-375 (376 bytes)]
        [Frame: 2, payload: 376-707 (332 bytes)]
        [Fragment count: 2]
        [Reassembled IPv4 length: 708]
        [Reassembled IPv4 data: [...]]
```

Es interesante señalar que el tamaño del primer fragmento es 376 y no 400, ya que 380 (400-20) no es múltiplo de 8. Fíjate en estos campos:

- Mensaje 1
  - `longitud total`: 396 bytes (376 + 20).
  - `identificación`: 0x85b8.
  - Flag MF: 1.
  - `offset`: 0.
- Mensaje 2
  - `longitud total`: 352 bytes (332 + 20).
  - `identificación`: 0x85b8.
  - Flag MF: 0.
  - `offset`: 101111=47 (376/8).

Además en el último fragmento `tshark` muestra información detallada del reensamblado. Indica el tamaño total del paquete original (700) y el de los fragmentos y el rango de los datos de ambos fragmentos.

```
[2 IPv4 Fragments (708 bytes): 1(376), 2(332)]
   [Frame: 1, payload: 0-375 (376 bytes)]
   [Frame: 2, payload: 376-707 (332 bytes)]
```

Puedes probar a enviar *ping* de otros tamaños o cambiar la MTU en el archivo `compose.yml`, y comprobar los resultados capturados en cada caso.

8.7.1. No fragmentar

Existen algunas situaciones en las que el origen necesita evitar la fragmentación, para lo cual marca el flag DF. Algunas de estas situaciones son:

- El destino es un dispositivo empotrado con capacidades muy limitadas y sin capacidad para reesamblar fragmentos.
- La ruta atraviesa una VPN o túnel. La fragmentación rompería la encapsulación.
- El flujo utiliza algún protocolo de tiempo real como RTP o VoIP en el que la fragmentación introduciría latencia y jitter muy perjudicial.

Cuando un router recibe un paquete marcado con el bit DF, descarta el paquete y envía un mensaje ICMP de tipo *Destination unreachable* y código *Fragmentation Needed* al nodo origen. El mensaje de error indica el tamaño máximo de la carga útil que puede aceptar la interfaz de salida de ese router.

Es sencillo provocar esta situación con el comando `ping` con los argumentos `-M do`, que fija el flag DF, y `-s`, que indica el tamaño de la carga útil. Para el caso de una interfaz Ethernet o WiFi fijamos una carga útil de 1 472 bytes, que sumando la cabecera ICMP (8 bytes) y la cabecera IP (20 bytes) da un total de 1 500 bytes, que es la MTU de esa tecnología. El comando por tanto sería:

```
$ ping -c 1 -M do -s 1472 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 1472(1500) bytes of data.
From 172.20.21.254 icmp_seq=1 Frag needed and DF set (mtu = 1370)
```

Si pones en marcha una captura antes de ejecutar el comando anterior recibirás algo similar a lo siguiente. Se omiten los campos poco relevantes:

```
$ sudo tshark -f "icmp" -V
Internet Protocol Version 4, Src: 172.20.21.254, Dst: 192.168.0.37
    Total Length: 576
    Time to Live: 62
    Protocol: ICMP (1)
    Source Address: 172.20.21.254
    Destination Address: 192.168.0.37
Internet Control Message Protocol
    Type: 3 (Destination unreachable)
    Code: 4 (Fragmentation needed)
    MTU of next hop: 1370
    Internet Protocol Version 4, Src: 192.168.0.37, Dst: 8.8.8.8
        Total Length: 1500
        Identification: 0x0000 (0)
        010. .... = Flags: 0x2, Don't fragment
        ...0 0000 0000 0000 = Fragment Offset: 0
        Time to Live: 62
        Protocol: ICMP (1)
```

```
        Source Address: 192.168.0.37
        Destination Address: 8.8.8.8
    Internet Control Message Protocol
        Type: 8 (Echo (ping) request)
        Code: 0
        [...]
```

Fíjate que el mensaje ICMP que informa del error incluye la cabecera IP y parte de la carga útil (el mensaje ping) que provocó el problema. También puedes comprobar que el mensaje de error lo está enviando `172.20.21.254`, que es el router que no ha podido reenviar el paquete. En concreto indica que el enlace saliente al que trataba de entregar el paquete tiene una MTU de 1 370 bytes.

8.7.2. Descubrimiento de MTU

El mensaje ICMP «*fragmentation needed*» (§ 8.6.1) se puede utilizar para descubrir el mayor valor de MTU que permite la ruta hasta un determinado destino. El origen puede empezar enviando un paquete *ping* con un tamaño igual a la MTU de su interfaz de salida y el flag DF activa. Si recibe un mensaje ICMP de este tipo, envía un nuevo mensaje del tamaño indicado en el error. Así sucesivamente hasta recibir la respuesta a *ping* por parte del destino. Este mecanismo se denomina PMTUD (Path MTU Discovery). Indirectamente, también determina el valor de MSS que se utilizará en las conexiones TCP.

Es posible consultar o modificar si el SO aplica PMTUD con el parámetro del núcleo `ip_no_pmtu_disc`. Esa abreviatura se puede leer como «no IP PMTU discovery», es decir, un valor de 0 significa que el descubrimiento está habilitado, que es la situación por defecto.

```
$ cat /proc/sys/net/ipv4/ip_no_pmtu_disc
0
```

Como algunos routers o incluso el destino podrían no procesar mensajes ICMP, existe una estrategia diferente basada en TCP llamada PLPMTUD (Packetization Layer Path MTU Discovery) o «*TCP MTU probing*». También es posible elegir qué estrategia PMTUD se ha de utilizar con la variable `tcp_mtu_probing`.

```
$ cat /proc/sys/net/ipv4/tcp_mtu_probing
0
```

Esta variable puede tomar 3 valores:

- 0: Utiliza siempre la estrategia basada en ICMP (PMTUD clásico).

- 1: Inicialmente utiliza la estrategia basada en ICMP, pero si no recibe respuestas, cambia a PLPMTUD.
- 2: Utiliza siempre PLPMTUD.

Por último, el socket también permite configurar la fragmentación y el comportamiento de PMTUD para un flujo específico. Se realiza por medio de la opción `IP_MTU_DISCOVER`. Puede tomar 6 valores con los siguientes significados:

- `IP_PMTUDISC_DONT (0)`: No realizar descubrimiento de PMTU y permitir la fragmentación.
- `IP_PMTUDISC_WANT (1)`: Intenta PMTUD clásico. Si no funciona, permite la fragmentación.
- `IP_PMTUDISC_PROBE (2)`: Aplicar PMTUD clásico, nunca fragmentar.
- `IP_PMTUDISC_DO (3)`: Aplicar PLPMTUD.
- `IP_PMTUDISC_INTERFACE (4)`: Utilizar la MTU de la interfaz.
- `IP_PMTUDISC_OMIT (5)`: Como la anterior, pero permite fragmentación si el tamaño excede la MTU de la interfaz.

Por ejemplo, puedes utilizar el siguiente código:

```
import socket
IP_MTU_DISCOVER = 10
IP_PMTUDISC_DO = 2

sock = socket.socket()
sock.setsockopt(socket.IPPROTO_IP, IP_MTU_DISCOVER, IP_PMTUDISC_DO)
```

Y ¿qué más?

En este capítulo hemos abordado una de las funciones clave de cualquier interred IP: llevar paquetes desde un origen a un destino remoto atravesando múltiples redes con tecnologías heterogéneas y probablemente incompatibles. Aquí se ve el papel *homogeneizador* del protocolo IP y la importancia de los routers como elementos de interconexión. Hemos visto también el protocolo ICMP y el problema que supone la disparidad de tamaños de trama propios de cada tecnología de enlace. En el capítulo 9 complementaremos toda esta funcionalidad estudiando como una interred puede descubrir automáticamente las rutas óptimas a cada posible destino gracias a los algoritmos y protocolos de encaminamiento dinámico.