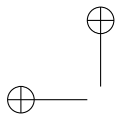
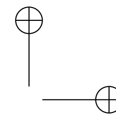



Capítulo 9

Encaminamiento dinámico

Al terminar este capítulo, entenderás:

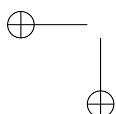
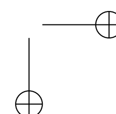
- La necesidad y el funcionamiento del encaminamiento dinámico.
- Qué son los sistemas autónomos y su importancia en el funcionamiento de Internet.
- Cómo funcionan los algoritmos de encaminamiento vector-distancia, estado de enlace y vector ruta.
- Las nociones básicas de los protocolos RIP y OSPF.

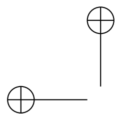
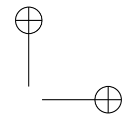
Cuando la interred es muy pequeña, las tablas de encaminamiento las crea y mantiene manualmente el administrador. Tanto si las calcula a mano como si utiliza algún programa de vez en cuando, llamamos a eso «encaminamiento estático». Cuando se produce algún cambio, el administrador es el responsable de detectarlo y adaptar las tablas.

Estos cambios pueden deberse a múltiples causas: puede ser un nuevo router (o uno que desaparece), un enlace que falla, una ruta mejor para llegar a un destino ya conocido, un cambio en la forma en la que se calcula la bondad de una ruta o una decisión puramente administrativa.

En cuanto la interred crece un poco, y esos cambios empiecen a producirse de forma habitual, mantener las tablas manualmente se convierte en una tarea compleja y propensa a errores. Se necesita una forma de detectar cambios en la topología y en las condiciones de la red, y algoritmos que puedan adaptar las tablas de encaminamiento inmediatamente y de acuerdo a esos cambios. En eso consiste el encaminamiento dinámico y su principal característica es que es **adaptativo**, es decir, detecta los cambios en la red y modifica las tablas de encaminamiento para adecuarse a la nueva situación.

Por contra, con encaminamiento estático, las tablas no cambian automáticamente, no hay nada que detecte los cambios, calcule nuevas rutas ni modifique las tablas.

La tabla de encaminamiento no es diferente en un caso u otro. De hecho es posible y habitual que una misma tabla contenga filas especificadas de forma estática o manual con otras obtenidas por un protocolo de encaminamiento dinámico, o incluso varios simultáneamente.

9.1. Algoritmos y protocolos de encaminamiento

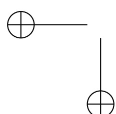
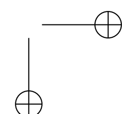
Un algoritmo de encaminamiento es un algoritmo distribuido utilizado por un grupo de routers que colaboran para deducir la topología de la subred que los conecta. Con la información que es capaz de recabar por sí mismo y la que los otros le pueden proporcionar, cada router calcula la mejor ruta para llegar a cada uno de los destinos y, a partir de eso, actualiza su tabla de encaminamiento.

Los routers colaboran compartiendo información sobre ellos mismos y también sobre lo que han aprendido. Obviamente esta información viaja dentro de mensajes, por lo que, asociados a los algoritmos, existen también protocolos de encaminamiento, con sus correspondientes formatos de mensaje y patrones de intercambio.

El conjunto de routers que ejecutan el algoritmo de encaminamiento funciona a todos los efectos como un sistema distribuido, formado por un conjunto de procesos que se comunican mediante paso de mensajes. En concreto se trata de un algoritmo de *consenso* distribuido, es decir, un algoritmo que permite a un grupo de procesos llegar a un acuerdo sobre el valor de una variable: la topología de la red, y eso a pesar de que algunos de ellos puedan fallar o no responder. Tampoco hay ningún elemento central, árbitro o coordinador que tenga un papel relevante. Aunque por razones de eficiencia o escalabilidad algunos protocolos eligen un router para algún rol especial, este mecanismo también es distribuido y permite elegir un sustituto si deja de responder.

En general, un protocolo de encaminamiento es un conjunto de reglas que define cómo los routers intercambian información sobre la topología de la red y cómo utilizan esa información para calcular las rutas óptimas. Pero del uso del protocolo deriva una limitación importante: cierto grado de obsolescencia de la información.

En los sistemas distribuidos, se dice que «no existe un reloj global», es decir, que es físicamente imposible sincronizar las acciones de un sistema formado por nodos que se comunican mediante mensajes. Este es exactamente el caso de los algoritmos de encaminamiento dinámico. No hay forma de que los routers envíen (y mucho menos reciban) la información topológica exactamente en el mismo instante. Esto implica que los routers trabajan siempre

con datos eventualmente obsoletos. A determinado nivel de precisión (quizá centésimas o milésimas de segundo) tampoco es posible saber cómo de actuales son los datos recibidos, ni las diferencias entre mensajes recibidos de orígenes diferentes. Esto debe considerarse en el diseño de los algoritmos y protocolos para que resulten robustos a pesar de esta limitación.

9.2. Sistemas autónomos

Antes de continuar debemos abordar el concepto de *sistema autónomo* (AS). Un AS es un conjunto de routers y enlaces que dependen o pertenecen a una misma entidad administrativa.

Es habitual que grandes empresas, gobiernos, universidades, proveedores de acceso a Internet (ISP) tengan su propio AS. Algunas de estas empresas pueden tener varios, como es el caso de Google o Amazon.

Cada AS se identifica con un número único de 16 bits (ASN) que, como es habitual, es asignado por la IANA y gestionado por los RIR. Con el crecimiento de Internet en 2 007 se crearon los ASN de 32 bits. Actualmente ambos tipos de identificadores conviven. Se han asignado más de 100 000 ASN públicos entre ambos tipos. Además existe un rango de ASN privados que no se anuncian a Internet, y que se utilizan por las organizaciones para sus redes internas.

Los AS se interconectan entre sí por medio de un tipo especial de router que se denomina *pasarela de borde* (*border gateway*) y que es capaz de encaminar paquetes a través de varios AS. Internet en una colección de AS interconectados.

La existencia de los AS da lugar a dos tipos diferentes de encaminamiento dinámico:

- **Encaminamiento interno** designa los algoritmos y protocolos de pasarela interior o IGP (Interior Gateway Protocol). Se refiere al encaminamiento de paquetes dentro de un mismo AS. Como cada AS es una unidad independiente, cada administrador puede decidir el algoritmo y protocolo de encaminamiento IGP que más le convenga.

- **Encaminamiento externo** designa los algoritmos y protocolos de pasarela exterior o EGP (Exterior Gateway Protocol). Se refiere al encaminamiento de paquetes entre AS independientes. A diferencia del interno, en este caso, todos los sistemas autónomos deben utilizar el mismo protocolo de encaminamiento.

A su vez, existen dos tipos principales de algoritmos de encaminamiento interno: *vector distancia* y *estado de enlace*. En el caso del encaminamiento exterior se utiliza esencialmente el algoritmo *vector ruta*. Dedicaremos las siguientes secciones a estudiarlos en detalle.

9.3. Topología de referencia: triple delta

Para explicar los algoritmos de encaminamiento interno, utilizaremos una topología de red concreta a modo de ejemplo que llamaremos «triple delta», porque visualmente forma tres triángulos. Esto ayudará a entender sus diferencias y comparar ventajas y desventajas. La topología en cuestión la puedes ver en la Figura 9.1.

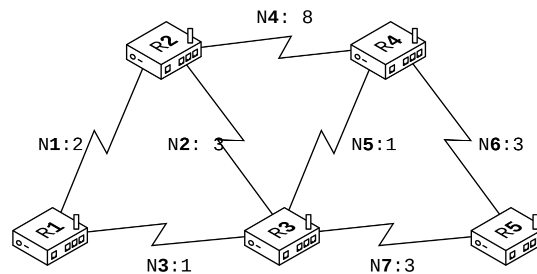


FIGURA 9.1: Topología «triple delta»

Aunque esta topología incluye detalles como la tecnología y tipo de enlaces (serie, en este caso), eso no es importante para el algoritmo de encaminamiento. Lo relevante es el grafo equivalente. En este, los nodos representan a los routers y las aristas a los enlaces. Esto se puede aplicar a cualquier red. Las redes a las que los routers dan acceso tampoco son relevantes para el cálculo de rutas dado que lo importante es poder llegar hasta el router que proporciona acceso a cada red. Por la misma razón, para el algoritmo tampoco son importantes las direcciones de los dispositivos. Verás que cuando se estudian los algoritmos desde un enfoque teórico la columna «destino» de las tablas de encaminamiento no contiene direcciones de red como habitualmente, sino simplemente los nombres de los routers.

Con todo eso, el grafo equivalente a la topología de la Figura 9.1 sería el que se muestra en la Figura 9.2. Como en el original, los enlaces del grafo indican su coste. En este ejemplo, se utiliza una métrica de latencia, pero podría ser cualquier otra y a efectos del algoritmo no es relevante, siempre que el objetivo sea minimizar el coste total de la ruta.

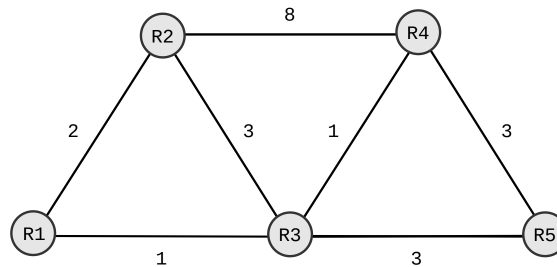


Figura 9.2: Grafo equivalente de la topología triple delta

9.4. Redundancia

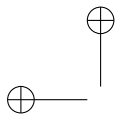
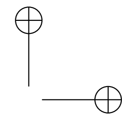
Esta topología de referencia es intencionadamente **redundante**, es decir, se proporcionan varias caminos para llegar a cualquier destino desde cualquier origen. Las redes e interredes suelen diseñarse de este modo porque aporta una gran ventaja que aprovecha automáticamente la conmutación de paquetes, que es disponer de caminos alternativos. La redundancia es la base para proporcionar tolerancia a fallos y balanceo de carga.

La tolerancia a fallos es la capacidad de la red[1] para seguir funcionando —llevando los paquetes a sus destinos— aún en presencia de fallos. En muchas situaciones, las prestaciones se verán afectas, porque quizá el camino óptimo no esté disponible a causa del fallo, pero esencialmente la red seguirá haciendo su trabajo. Por el modo en que funciona la conmutación de paquetes es muy probable que el servicio no llegue a interrumpirse en ningún momento y los usuarios no noten nada en absoluto. Del mismo modo, si el problema es temporal, la red volverá a funcionar con normalidad cuando se resuelva. En estas situaciones se dice que el servicio proporciona «transparencia de fallos».

El balanceo de carga cosiste en aprovechar la redundancia para distribuir el tráfico entre los caminos disponibles. Se trata de utilizar todos los recursos disponibles de forma equitativa evitando la saturación de unos recursos mientras otros quedan infrautilizados. El balanceo de carga puede ser complejo y puede requerir algoritmos capaces de manejar múltiples variables, criterios y estimaciones.

Los algoritmos de encaminamiento dinámico pueden sacar partido de la redundancia y proporcionar tolerancia a fallos y, los más avanzados, también balanceo de carga.

[1] Se puede aplicar a cualquier sistema

9.5. La ruta más corta

Todos los algoritmos de encaminamiento tratan de encontrar la mejor ruta (la óptima) desde un nodo origen a un nodo destino. Pero la «mejor ruta» o incluso aunque hablemos de la «ruta más corta» no es necesariamente la ruta que recorre la menor distancia. La «mejor ruta» es la que tiene un menor coste. El coste de la ruta normalmente se calcula como suma del coste de todos los enlaces que la forman. El coste del enlace puede ser cualquier propiedad medible que el administrador considere relevante. Esta propiedad determina la 'métrica' del algoritmo. La métrica más sencilla es el número de saltos, pero hay muchas otras: retardo, ancho de banda, fiabilidad, coste económico, etc., o combinación de ellas.

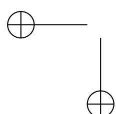
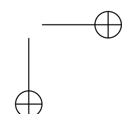
Algunas de estas métricas no cambian con el tiempo. Una ruta de 5 saltos seguirá teniendo 5 saltos independientemente de que la red esté libre o saturada, es decir, no depende de la carga. Otras, como el retardo o el ancho de banda sí dependen; incluso el propio algoritmo de encaminamiento podría influir en esas propiedades. Por ejemplo, si existe un enlace con bajo retardo y el algoritmo la prioriza, es probable que la carga adicional eleve el retardo haciendo que el propio algoritmo la considere menos adecuada en el futuro inmediato. Este es un efecto muy perjudicial porque afecta a la convergencia del algoritmo. Lo deseable es que el algoritmo no modifique las rutas a menos que ocurran cambios en la topología.

El cálculo de la ruta óptima se base en una idea sencilla: el «principio de optimalidad». Este principio dice que si existe una ruta óptima entre el nodo `A` y el `B`, que pasa por `C`, entonces la ruta entre `C` y `B` también es necesariamente óptima. En esta idea se fundamentan los algoritmos de Dijkstra, Bellman-Ford y otros, que son los más utilizados en encaminamiento dinámico.

El algoritmo Dijkstra [18] calcula el camino más corto (de menor coste) entre dos nodos de un grafo no dirigido y ponderado, es decir, un grafo cuyas aristas son transitables en ambas direcciones y tienen costes asociados a una métrica arbitraria. La aplicación general del algoritmo al grafo completo calcula la ruta más corta a todos los nodos. Como esas rutas tienen partes comunes, el resultado es el árbol de rutas más cortas (SPT) para el origen solicitado. Por ejemplo, para la topología de referencia anterior, el árbol de rutas más cortas desde `R1` sería el de Figura 9.3.

De manera resumida, el algoritmo de Dijkstra hace lo siguiente:

1. Inicializa el coste a todos los nodos a infinito, excepto el nodo origen, que se fija en 0.
2. Marca todos los nodos como no visitados.

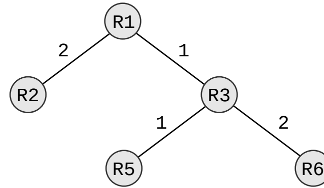


FIGURA 9.3: Árbol de rutas más cortas (SPT) para `R1`

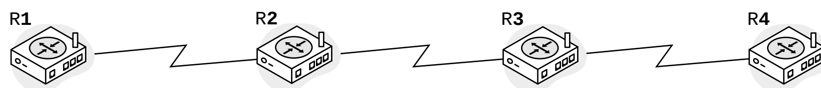
3. Selecciona el nodo no visitado con el menor coste y lo marca como visitado.
4. Actualiza los costes de los nodos vecinos del nodo visitado. Si el nuevo coste es menor que el coste actual, lo actualiza.
5. Repite los pasos 3 y 4 hasta que todos los nodos hayan sido visitados o no queden nodos accesibles.

En el repositorio de ejemplos puedes encontrar una implementación del algoritmo Dijkstra (/`dijkstra/dijkstra.py`) y un notebook que lo aplica a un ejemplo (/`dijkstra/dijkstra.ipynb`).

9.6. Vector-distancia

Con el algoritmo vector-distancia inicialmente cada router solo tiene información sobre sus vecinos (routers directamente conectados). Esa información, que consiste esencialmente en la dirección/identificador y la distancia/coste a cada vecino, se coloca en una tabla llamada *vector-distancia*. La obtención de esta información es lo que llamamos *inicialización*. Después cada router envía su vector solo a los vecinos, que la utilizan para actualizar sus propios vectores-distancia y sus tablas de encaminamiento. A esa transferencia de vectores es a lo que llamamos una *iteración* del algoritmo. La inicialización no es una iteración puesto que no hay propagación de vectores-distancia.

Veamos un ejemplo trivial con 4 routers empleando una métrica de saltos:



La tabla 9.1 muestra cómo se propagan los vectores-distancia. La primera fila es la mencionada *inicialización*. Los elementos de estos vectores tienen el formato `destino:coste:vecino`, que corresponden respectivamente al router destino, el coste hasta él y el vecino siguiente salto —siendo - la forma de

indicar entrega directa. Por ejemplo, para `R1` aparece `R1:0:-` que significa que el coste para llegar a sí mismo es 0 (lógicamente) y que puede acceder con entrega directa. También aparece `R2:1:-`, que indica que puede llegar a `R2` con un coste de 1 y entrega directa. Fíjate que los vectores incluyen a los propios routers (los que tienen coste 0) dado que esta información debe enviarse también.

	R1	**R2**	**R3**	**R4**
inicialización				
	R1:0:-	R1:1:-	R2:1:-	R3:1:-
	R2:1:-	R2:0:-	R3:0:-	R4:0:-
		R3:1:-	R4:1:-	
primera iteración				
	R3:2:R2	R4:2:R3	R1:2:R2	R2:2:R3
segunda iteración				
	R4:3:R2	-	-	R1:3:R3

Cuadro 9.1: Propagación de vectores-distancia

La segunda y tercera filas muestran la nueva información que incorpora cada router a su vector-distancia en cada iteración del algoritmo. Por ejemplo, en la primera iteración `R1` recibe información de `R3` a través de `R2` y actualiza su vector-distancia añadiendo `R3:2:R2`, es decir, puede llegar a `R3` con un coste de 2 a través de `R2`. En la tercera `R1`, después de recibir información procedente de `R4` añade `R4:3:R2`, es decir, puede llegar a `R4` con un coste de 3 a través de `R2`. Por tanto, en esta segunda iteración, los routers de los extremos ya tienen la información necesaria para llegar a todos los demás routers con al menor coste posible. Cuando ocurre esto se dice que el algoritmo ha *convergido*. En este caso, la convergencia se produce en 2 iteraciones.

Ahora veamos un ejemplo más complejo sobre la topología de referencia de la Figura 9.1 y empleando una métrica basada en los costes de los enlaces. La tabla 9.2 muestra el vector-distancia inicial de cada router (un router por fila).

R1	R1:0:-	R2:2:-	R3:1:-			
R2	R1:2:-	R2:0:-	R3:3:-		R5:8:-	
R3	R1:1:-	R2:3:-	R3:0:-	R4:2:-		
R4			R3:2:-	R4:0:-	R5:1:-	R6:3:-
R5		R2:8:-		R4:1:-	R5:0:-	R6:3:-
R6				R4:3:-	R5:3:-	R6:0:-

Cuadro 9.2: Vector-distancia inicial de cada router

Lo mostramos aquí como una tabla para facilitar su lectura, pero en la práctica son vectores individuales. Es decir, lo que enviaría `R1` en la primera iteración sería algo como `(R1:0:-), (R2:2:-), (R3:1:-)`.

Dado que en cada iteración la información solo avanza un salto, si emplea una métrica de saltos, el algoritmo converge en un número de iteraciones igual al diámetro del grafo menos uno. Pero si la métrica es otra, podría requerir más iteraciones, porque rutas con más saltos podrían tener costes menores. El diámetro del grafo indica la cantidad de aristas (saltos) de la más costosa de las rutas menos costosas. Se define formalmente como:

$$D(G) = \max_{u,v \in V} d(u, v) \tag{9.1}$$

donde V es el conjunto de todos los vértices del grafo G y $d(u, v)$ representa el coste del camino óptimo entre los nodos u y v.

La tabla 9.3 muestra los vectores de todos los routers después de la primera iteración. Fíjate que `R1` utiliza `R2` para llegar a `R5` con un coste de 10. Claramente no es la mejor ruta. La ruta óptima es `R1-R3-R4-R5`, pero en la primera iteración `R1` aún no ha recibido el vector de `R4`. Aún no sabe que existe una ruta mejor para llegar a `R5`. Puedes comprobar que esta misma situación ocurre con otras rutas.

R1	R1:0:-	R2:2:-	R3:1:-	R4:3:R3	R5:10:R2	
R2	R1:2:-	R2:0:-	R3:3:-	R4:5:R3	R5:8:-	R6:11:R5
R3	R1:1:-	R2:3:-	R3:0:-	R4:2:-	R5:3:R4	R6:5:R4
R4	R1:3:R3	R2:5:R3	R3:2:-	R4:0:-	R5:1:-	R6:3:-
R5	R1:10:R2	R2:8:-	R3:3:R4	R4:1:-	R5:0:-	R6:3:-
R6		R2:11:R5	R3:5:R4	R4:3:-	R5:3:-	R6:0:-

CUADRO 9.3: Vector-distancia de cada router después de la primera iteración

La tabla 9.4 muestra los vectores de cada router después de la segunda iteración. Las celdas sombreadas corresponden a vectores que han mejorado su coste respecto a la primera iteración. En esta situación el algoritmo ha convergido y los vectores no cambiarán a menos que se produzca algún cambio en la topología.

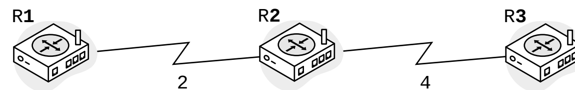
Para esta topología, también han sido necesarias 2 iteraciones (3 saltos entre los routers más distantes). Sin embargo, el algoritmo no acaba al alcanzar la convergencia. Sigue trabajando (ejecutando nuevas iteraciones) indefinidamente porque la topología puede cambiar en cualquier momento. El algoritmo debe adaptar las tablas de encaminamiento de los routers a las condiciones dinámicas de la red.

R1	R1:0:-	R2:2:-	R3:1:-	R4:3:R3	R5:4:R3	R6:6:R3
R2	R1:2:-	R2:0:-	R3:3:-	R4:5:R3	R5:6:R3	R6:8:R3
R3	R1:1:-	R2:3:-	R3:0:-	R4:2:-	R5:3:R4	R6:5:R4
R4	R1:3:R3	R2:5:R3	R3:2:-	R4:0:-	R5:1:-	R6:3:-
R5	R1:4:R4	R2:6:R4	R3:3:R4	R4:1:-	R5:0:-	R6:3:-
R6	R1:6:R4	R2:8:R4	R3:5:R4	R4:3:-	R5:3:-	R6:0:-

Cuadro 9.4: Vector-distancia de cada router tras alcanzar la convergencia

9.6.1. Problemas y limitaciones de vector-distancia

El principal problema del algoritmo de vector-distancia ya lo hemos visto: es lento. Tarda mucho en converger aunque todo vaya bien. Pero cuando las cosas van mal, es mucho más lento. Partamos de una topología muy simple, formada solo por tres routers R1, R2 y R3 y sus vectores-distancia una vez que algoritmo ha convergido.



R1	R1:0:-	R2:2:-	R3:6:R2
R2	R1:2:-	R2:0:-	R3:4:-
R3	R1:6:R2	R2:4:-	R3:0:-

Supongamos ahora que el R3 desaparece o el enlace que lo conecta con R2 se rompe. A partir de ese momento, R1 y R2 intentan utilizar la información disponible para determinar la nueva ruta y coste para llegar a R3. Eso provoca las siguientes acciones:

- R2 ya no recibe información de R3, solo de R1, que le indica que puede llegar a R3 con un coste de 6 (R3:6:R2). R2 actualiza su vector con R3:8:R1, es decir, puede llegar a R3 con un coste de 8 (6+2) a través de R1. Obviamente esto no es cierto, pero R2 no tiene forma de saberlo.
- R1 recibe el nuevo vector de R3, de modo que actualiza su vector a R3:10:R2.
- R2 actualiza su vector a R3:12:R1.
- ...

Este proceso continua indefinidamente, aumentando el coste en cada iteración, y por eso se conoce como *cuenta a infinito* (*count to infinity*). Es un problema grave porque R1 y R2 se estarán reenviado entre sí los paquetes dirigidos a R3 hasta que el valor de TTL llegue a 0 y acaben por descartarse.

Esto se conoce como un *bucle de encaminamiento*, aunque también puede ser provocado por otras causas.

Existen algunas soluciones para el problema de la cuenta a infinito:

- **«Definir» o acotar el infinito**. Consiste en fijar un valor máximo de coste. Un valor mayor implica considerar inaccesible el router afectado. El inconveniente es que limita también el tamaño de la red. Ninguna ruta real podrá tener un coste mayor al valor fijado.
- **Horizonte dividido (*split horizon*)**. Consiste en no enviar a un vecino la información obtenida de ese vecino. Para el ejemplo implicaría que `R1` no envíe a `R2` el vector `R3:6:R2`, ya que `R1` ha aprendido sobre `R3` por medio de `R2`.
- **Ruta inversa envenenada *(poison reverse)***. Se utiliza en combinación con horizonte dividido. Consiste en informar al router del que ha aprendido un destino, que ese destino es inalcanzable para él. En el ejemplo, `R1` enviaría a `R2` el vector `R3:-1:`, en el entendido de que -1 indica inalcanzable.

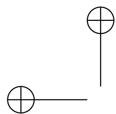
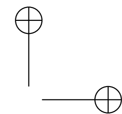
Aunque *horizonte dividido* y *ruta inversa envenenada* mejoran el desempeño del algoritmo, no resuelven completamente el el problema de la cuenta a infinito. Esto es porque estas técnicas solo afectan a vecinos, pero no evitan que la información errónea llegue a través de otros routers cuando la topología no es tan simple como la del ejemplo.

Además de la cuenta a infinito, el algoritmo de vector-distancia tiene otros problemas:

- La ya citada convergencia lenta. La información de cada router se propaga solo de uno en uno.
- Los routers envían información de encaminamiento periódicamente aunque no haya cambios en la topología, lo que consume ancho de banda y recursos.
- Puede ser inestable en redes grandes o topologías complejas, provocando cambios continuos en las tablas de encaminamiento.

9.6.2. RIP

Routing Information Protocol (RIP) es un protocolo de encaminamiento que emplea el algoritmo vector-distancia. Fue el protocolo de encaminamiento más utilizado en Internet hasta mediados de los 90. A partir de entonces empezó a ser reemplazo por OSPF, BGP e IS-IS. La primera versión de RIP [19] empleaba el algoritmo de Bellman-Ford para el cálculo de la ruta más corta.

Utiliza una métrica de saltos y para mitigar el problema de la cuenta a infinito define un valor máximo de 15 saltos y aplica horizonte dividido. Es un protocolo *classfull*, es decir, no envía información sobre la máscara de subred y por tanto solo puede trabajar con las redes de clase A, B y C originales.

Envía actualizaciones de encaminamiento (vectores-distancia) cada 30 segundos a la dirección broadcast `255.255.255.255`.

El mensaje RIP indica el comando (1 para petición, 2 para respuesta) y la versión del protocolo. Después incluye un máximo de 25 vectores-distancia. Cada uno de ellos consta de:

- Familia de direcciones.
- Dirección IP de la red destino.
- Número de saltos hasta la red destino.

En 1993 se publicó RIP versión 2 [20] que utiliza el algoritmo de Ford-Fulkerson para el cálculo de la ruta óptima. Incluye varias mejoras que solventan los problemas del RIP original. Incluye soporte para VLSM y CIDR, utiliza multicast para enviar las actualizaciones (grupo 224.0.0.9) e incluye un mecanismo de autenticación.

Aunque a día de hoy muchos equipos aún ofrecen soporte, su uso es residual y se limita a redes pequeñas y simples, o entornos de laboratorio.

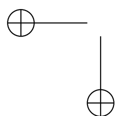
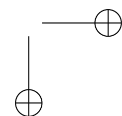
9.7. Estado de enlace

El «otro» algoritmo importante para encaminamiento interno es estado de enlace (*link state*). La diferencia principal con vector-distancia es que aquí los routers tienen información topológica de toda la subred, y no solo una medida de coste a cada router.

Por su cuenta cada router:

- Descubre sus vecinos y sus direcciones.
- Mide el coste a cada vecino.
- Construye un mensaje con la información obtenida.
- Envía ese mensaje a todos los routers de la subred (o AS).
- Con la información propia y la recibida crea el grafo de la subred.
- Calcula la ruta más corta a todos los routers.

Cuando un enlace o router cambia de estado, es decir, se activa, desactiva o cambia su coste, los routers que lo detecten envían inmediatamente una actualización a todos los demás.

Este algoritmo requiere más cantidad de memoria al tener que almacenar la topología, requiere más computación al tener que calcular la ruta a cada destino y más ancho de banda ya que genera más tráfico. A cambio, converge más rápido, es más estable y se puede utilizar en subredes más grandes y complejas que vector-distancia.

9.7.1. OSPF

Open Short Path First (OSPF) es el protocolo de encaminamiento de estado de enlace más utilizado en Internet en la actualidad. La primera versión funcional (la 2) fue publicada en 1991 [21] y se diseñó para sustituir a RIP, resolviendo la mayoría de sus problemas, aunque a costa de mayor complejidad y consumo de recursos. Es un protocolo *classless* desde su diseño y por tanto soporta CIDR y VLSM. Utiliza Dijkstra para el cálculo de la ruta más corta y ancho de banda como métrica.

Segmenta el SA en áreas para mejorar la escalabilidad y reducir el tráfico del propio protocolo. La distribución de áreas es jerárquica y hay dos niveles: áreas de tránsito y áreas de borde. El área *backbone* (la 0) es la principal área de tránsito y debe estar en cualquier despliegue de OSPF. Las áreas de tránsito conectan con las de borde routers ABR (Area Border Router), es decir, el tráfico de las otras áreas debe pasar por el área *backbone*. Las áreas de borde solo manejan tráfico propio. Utilizan los ABR para comunicarse con el resto del AS y los ASBR (Autonomous System Border Router) para comunicarse con otros AS.

Cada router mantiene una LSDB (Link State Database) con la información de la topología. Se actualiza periódicamente o cuando se producen cambios a partir del intercambio de mensajes LSA. Hay 7 tipos distintos de LSA en función del tipo de router que lo envía, la información que contiene y su alcance. Los mensajes LSA se envían al grupo multicast `224.0.0.5`.

9.8. Vector-ruta y BGP

A diferencia de vector-distancia y estado de enlace, vector-ruta es un algoritmo de encaminamiento externo, es decir, se utiliza para encaminamiento entre AS. Los EGP son los encargados de este tipo de encaminamiento. BGP (Border Gateway Protocol) es el EGP más utilizado. Obviamente hay muchas diferencias entre los EGP, que utilizan vector-ruta, respecto a los IGP, pero hay dos que resultan muy evidentes:

- Las rutas se determinan entre AS, no entre routers.
- Las tablas indican la ruta completa hasta el AS destino, no solo el siguiente salto. Estas sí son literalmente «tablas de rutas».

Como vimos en la §9.2, cada AS se identifica con un ASN. Una ruta a través de distintos AS se especifica con una secuencia de ASN (se llama ASPATH). Los routers de frontera (*border router*), como su nombre indica, son los que conectan un AS a otros. Son los encargados de anunciar las ASPATH a los demás AS. Cuando un router BGP recibe una ruta de este tipo puede comprobar fácilmente si esta pasa por su AS. Si ese es el caso, la descarta, y de ese modo evita bucles de encaminamiento. En caso contrario, añade su ASN al final y anuncia esa nueva ruta a los demás AS. La métrica más sencilla para medir la bondad de una ruta es su longitud. Una ruta será mejor si implica menos AS. Las preferencias del administrador son determinantes en la elección, mucho más que en los IGP, incluso por encima de las métricas puramente técnicas.

Fíjate que los routers de frontera tienen que ejecutar tanto un IGP como un EGP. Un IGP (por ejemplo OSPF para encaminar los paquetes dentro del AS) y un EGP (como BGP) para encaminarlos entre AS. Además BGP está compuesto por dos protocolos: eBGP es el que comunica rutas BGP entre distintos AS, mientras que iBGP lo hace entre los routers de un mismo AS.

En la Figura 9.4 aparecen tres AS interconectados. Los marcados en amarillo son routers frontera que tiene que ejecutar tanto un IGP como un EGP.

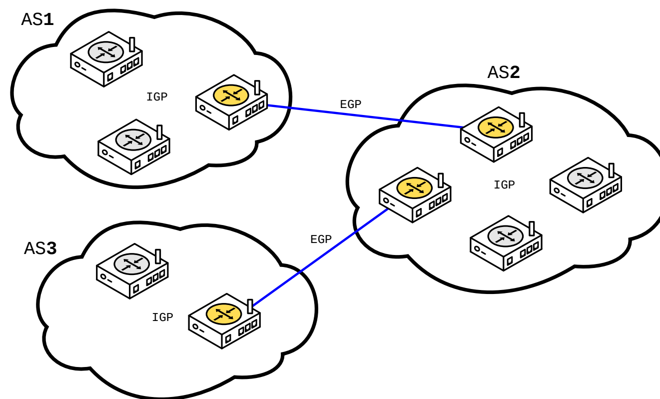


FIGURA 9.4: Sistemas Autónomos interconectados

9.9. Laboratorio de encaminamiento Delta

Para ver los protocolos de encaminamiento desde un punto de vista mucho más práctico, vamos a montar un pequeño laboratorio emulando la topología de la Figura 9.5, que llamaremos «Delta».

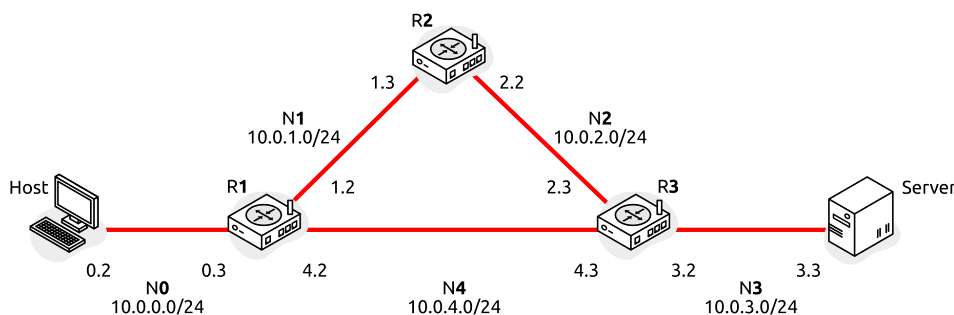


Figura 9.5: Topología Delta

Aunque es una topología ciertamente muy simple, tiene una característica importante: es redundante. Algo que, como ya hemos visto, es siempre buena idea. Por eso los protocolos de encaminamiento deben encontrar la mejor de las rutas disponibles. Sin embargo, los protocolos actuales no descartan las rutas no elegidas. Es mejor idea mantenerlas como rutas alternativas para poder utilizarlas en caso de fallo o para balancear el tráfico.

El nodo `Host` de la figura es tu PC real. Los routers y el nodo `Server` son contenedores docker. Eso significa que te puedes comunicar con ellos ejecutando comandos reales en tu terminal.

Todos los archivos necesarios para ejecutar este laboratorio (llamado `lab-routing-delta`) están disponibles en el repositorio de ejemplos del libro (página XXIX).

La topología completa está especificada en el archivo `compose.yml` (Listado 9.1).

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

services:
  r1:
    image: lab-delta-router
    container_name: r1
    hostname: r1
```

```
    <<: *common-settings
    volumes:
      - ./daemons:/etc/frr/daemons
      - ./frr-r1.conf:/etc/frr/frr.conf
    networks:
      N0: { ipv4_address: 10.0.0.3 }
      N1: { ipv4_address: 10.0.1.2 }
      N4: { ipv4_address: 10.0.4.2 }

  r2:
    image: lab-delta-router
    container_name: r2
    hostname: r2
    <<: *common-settings
    volumes:
      - ./daemons:/etc/frr/daemons
      - ./frr-r2.conf:/etc/frr/frr.conf
    networks:
      N1: { ipv4_address: 10.0.1.3 }
      N2: { ipv4_address: 10.0.2.2 }

  r3:
    image: lab-delta-router
    container_name: r3
    hostname: r3
    <<: *common-settings
    volumes:
      - ./daemons:/etc/frr/daemons
      - ./frr-r3.conf:/etc/frr/frr.conf
    networks:
      N2: { ipv4_address: 10.0.2.3 }
      N3: { ipv4_address: 10.0.3.2 }
      N4: { ipv4_address: 10.0.4.3 }

  server:
    image: lab-delta-server
    container_name: server
    privileged: true
    networks:
      N3: { ipv4_address: 10.0.3.3 }

networks:
  N0:
    driver: bridge
    driver_opts: { com.docker.network.bridge.name: N0 }
    ipam: { config: [{subnet: 10.0.0.0/24}] }
  N1:
    driver: bridge
    driver_opts: { com.docker.network.bridge.name: N1 }
    ipam: { config: [{subnet: 10.0.1.0/24}] }
  N2:
    driver: bridge
    driver_opts: { com.docker.network.bridge.name: N2 }
    ipam: { config: [{subnet: 10.0.2.0/24}] }
  N3:
    driver: bridge
    driver_opts: { com.docker.network.bridge.name: N3 }
    ipam: { config: [{subnet: 10.0.3.0/24}] }
  N4:
```

```
    driver: bridge
    driver_opts: { com.docker.network.bridge.name: N4 }
    ipam: { config: [{subnet: 10.0.4.0/24}] }
```

Listado 9.1: Descripción del laboratorio de encaminamiento Delta
/lab-routing-delta/compose.yml

Este archivo se procesa con el comando `docker-compose`[2]. Además se necesitan otros archivos de configuración y scripts.

Se proporciona un archivo `Makefile` en el mismo directorio que ejecuta algunas tareas básicas necesarias —que se explican a continuación. Es importante utilizarlo en lugar de lanzar directamente los comandos de `docker`. Necesitas instalar el paquete Debian `make` si no lo tienes. Para arrancar la **configuración básica** del laboratorio ejecuta:

```
$ make up
[...]
[+] Running 4/4
   Container r1      Running
   Container r2      Running
   Container r3      Running
   Container server  Running
```

Una vez todo en marcha, puedes ver los contenedores con:

```
$ docker compose ps
NAME     IMAGE    COMMAND                  SERVICE   CREATED          STATUS
r1       router   "sh -c 'ip route del…"  r1        26 seconds ago   Up
r2       router   "sh -c 'ip route del…"  r2        26 seconds ago   Up
r3       router   "sh -c 'ip route del…"  r3        26 seconds ago   Up
server   server   "sh -c 'ip route del…"  server    26 seconds ago   Up
```

Y las redes con:

```
$ docker network ls
NETWORK ID     NAME                   DRIVER    SCOPE
266d2cb29af4   bridge                 bridge    local
059b37dd8ae6   host                   host      local
030f578f0ff0   none                   null      local
cbcf75bf8048   lab-routing-delta_N0   bridge    local
ab4139a63510   lab-routing-delta_N1   bridge    local
2520be21d05b   lab-routing-delta_N2   bridge    local
6fe6bdc69c3f   lab-routing-delta_N3   bridge    local
270e2a372326   lab-routing-delta_N4   bridge    local
```

Estas redes se implementan como *bridges*. Un bridge es una especie de conmutador (*switch*) virtual que proporciona Linux y que funciona como una red Ethernet. Docker crea un *bridge* por cada red (de N0 a N4). También crea interfaces Ethernet virtuales para conectar los contenedores a las redes. Todo esto lo puedes ver si ejecutas el comando `ip a` en tu PC. Aquí se

[2]En versiones más recientes puede estar disponible como `docker compose` (sin guion).

muestra un fragmento de lo que verás. Los `br-<número>` son los *bridge* y los `veth<número>` son las interfaces virtuales.

```
1 $ ip a
2 767: br-203457108c6e: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 state UP group default
3     link/ether d2:aa:22:09:37:28 brd ff:ff:ff:ff:ff:ff
4     inet 10.0.0.1/24 brd 10.0.0.255 scope global br-203457108c6e
5        valid_lft forever preferred_lft forever
6 768: veth28e8f66@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master br-203457108c6e
  ↪  state UP group default
7     link/ether fe:87:85:50:cb:f8 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

Vale la pena mencionar lo que hace `make up` del `Makefile` para evitar confusiones. Es esta:

```
up: build
    docker compose up --remove-orphans -d
    sudo ./rm-bridge-addrs.sh
    sudo ./setup-hosts.sh
```

Veamos su función:

- `docker compose up` arranca los contenedores y los mantiene en *background*. También crea las redes que acabamos de comentar.
- `rm-bridge-addrs.sh` elimina las direcciones IP de esos bridge, excepto el que conecta el host con la red `N0`. Esto es necesario porque de otro modo tu PC tendría acceso directo a todas la redes invalidando el ejercicio. Lo que nosotros pretendemos es que sean los routers los que lleven el tráfico a esas redes.
- `setup-hosts.sh` configura en tu PC la ruta estática que envía a `r1` el tráfico dirigido a `10.0.0.0/16` (todas las redes de la figura). También configura `r3` como router por defecto para `Server`.

Ahora puedes ejecutar comandos en cualquiera de los contenedores. Por ejemplo, puedes ver la tabla de encaminamiento de `r1` con el siguiente comando:

```
$ docker exec r1 ip route
10.0.0.0/24 dev eth2 proto kernel scope link src 10.0.0.3
10.0.1.0/24 dev eth0 proto kernel scope link src 10.0.1.2
10.0.4.0/24 dev eth1 proto kernel scope link src 10.0.4.2
```

Las filas de esta tabla las ha creado automáticamente el SO al activar las interfaces del router. Estamos seguros de eso porque la propia tabla lo indica con *proto kernel*. Ambas filas indican que para llegar a las redes `N0`, `N1` y `N4`se debe hacer entrega directa a través de las interfaces `eth2`, `eth0` y `eth1` respectivamente. Docker asigna los nombres a estas interfaces automáticamente, y podrían cambiar en cada ejecución, así que quizá no

coincidan con los que ves en tu PC. Puedes probar el mismo comando con r2 y r3 para ver sus tablas.

A diferencia de los routers, el nodo Server solo tiene una interfaz y está conectada a la red N3. Mira el resultado en este caso:

```
$ docker exec server ip route
10.0.3.0/24 dev eth0 proto kernel scope link src 10.0.3.3
default via 10.0.3.2 dev eth0
```

Esta tabla de encaminamiento solo tiene una fila para entrega directa a la red N3. La segunda fila la ha configurado el script setup-hosts.sh, del que hablamos antes, indicando que r3 (10.0.3.2) es su router por defecto. Solo de ese modo podrá enviar tráfico a las otras redes.

Con esta configuración básica, el nodo Server tiene conectividad con r3:

```
$ docker exec server ping -c1 10.0.3.2
PING 10.0.3.2 (10.0.3.2) 56(84) bytes of data.
64 bytes from 10.0.3.2: icmp_seq=1 ttl=64 time=0.188 ms
```

Pero no con las otras redes, porque los routers no las conocen:

```
$ docker exec server ping -c1 10.0.1.2
PING 10.0.1.2 (10.0.1.2) 56(84) bytes of data.
From 10.0.3.2 icmp_seq=1 Destination Net Unreachable
```

En las siguientes secciones vamos a configurar el encaminamiento de esta interred de tres formas: encaminamiento estático, RIP y OSPF.

9.9.1. Encaminamiento estático

El encaminamiento estático es muy sencillo en este caso. Simplemente hay que decirle a los routers cómo llegar a las redes distantes, es decir, a las que no están directamente conectados. Por ejemplo, a r1 debes indicarle cómo llegar a las redes N2 y N3. Los comandos necesarios se incluyen en el script setup-static.sh.

```
1 docker exec r1 ip route add default via 10.0.4.3
2 docker exec r2 ip route add 10.0.0.0/24 via 10.0.1.2
3 docker exec r2 ip route add default via 10.0.2.3
4 docker exec r3 ip route add default via 10.0.4.2
```

Fíjate que r3 se configura como router por defecto para r1, porque él sabrá cómo llegar a las otras 2 redes (**línea 1**). Hacemos algo equivalente también con r3 para r2 (**línea 3**) y con r1 para r3 (**línea 4**). Sin embargo, para que r2 pueda llegar a la red N0 debemos indicar explícitamente que lo envíe a través de r1 (**línea 2**), porque no puede tener dos routers por defecto.

Para reiniciar la topología, aplicar los scripts que hemos visto y configurar el encaminamiento estático, ejecuta:

```
$ make static
```

Ahora puedes comprobar que tienes conectividad con `Server`. Desde una consola en tu PC puedes probarle ping:

```
$ ping -c1 10.0.3.3
PING 10.0.3.3 (10.0.3.3) 56(84) bytes of data.
64 bytes from 10.0.3.3: icmp_seq=1 ttl=64 time=0.161 ms
```

O `traceroute`:

```
$ traceroute 10.0.3.3
traceroute to 10.0.3.3 (10.0.3.3), 30 hops max, 60 byte packets
 1  10.0.0.3 (10.0.0.3)  0.767 ms  0.666 ms  0.635 ms
 2  10.0.4.3 (10.0.4.3)  0.608 ms  0.535 ms  0.496 ms
 3  10.0.3.3 (10.0.3.3)  0.459 ms  0.377 ms  0.330 ms
```

En esta salida de `traceroute` se puede ver cómo contestan `r1`, `r3` y el nodo `Server` para cada uno de los 2 saltos necesarios.

También puedes arrancar un servidor en `Server` y acceder a él desde tu PC para comprobar así la conectividad a nivel de transporte. Con el siguiente comando puedes arrancar un servidor `ncat` que simplemente envía la hora al cliente que conecte:

```
$ docker exec server ncat -lp 2000 -c date
```

Y en otra consola, ejecuta el cliente (también con `ncat`):

```
$ ncat 10.0.3.3 2000
Mon Mar  2 20:08:19 CET 2026
```

Con estas tres pruebas no queda ninguna duda de que el encaminamiento está haciendo su trabajo.

9.9.2. Zebra

Actualmente[3] muchos de los SO basados en GNU/Linux utilizan un servicio llamado FRR (Free Range Routing) para implementar encaminamiento dinámico. FRR ofrece un servicio para cada uno de los protocolos de encaminamiento soportados. Todos ellos se comunican con otro serivicio llamado `Zebra`, qué es quién coordina y decide qué información se incluye finalmente en la tabla de encaminamiento, permitiendo incluso que varios protocolos trabajen a la vez.

[3]Escribo esto en 2026.

Los contenedores de los routers ejecutan `Zebra`, aunque no lo hemos utilizado en la sección anterior. Lo puedes ver en el archivo `router/Dockerfile`:

```
CMD ["sh", "-c", \
  "ip route del default && \
  sysctl -w net.ipv4.ip_forward=1 && \
  /usr/lib/frr/zebra"]
```

Cuando el servico arranca, elimina la ruta por defecto que configura docker para evitar que interfiera con nuestro esquema de encaminamiento, activa el reenvío y ejecuta `zebra`, que queda en ejecución mientras el contenedor esté activo. Es posible utilizar `vtysh`[4] para contactar con `Zebra` y obtener información o configurar parámetros de encaminamiento. Con el siguiente comando puedes ver la tabla de encaminamiento de `r1` tal como quedó configurada en la sección anterior:

```
$ docker exec -it r1 vtysh
r1# show ip route
Codes: K - kernel route, C - connected, S - static, R - RIP,
       O - OSPF, I - IS-IS, B - BGP, E - EIGRP, N - NHRP,
       T - Table, v - VNC, V - VNC-Direct, A - Babel, F - PBR,
       f - OpenFabric,
       > - selected route, * - FIB route, q - queued, r - rejected, b - backup
       t - trapped, o - offload failure

K>* 0.0.0.0/0 [0/0] via 10.0.4.3, eth2, 00:04:14
C>* 10.0.0.0/24 is directly connected, eth0, 00:04:18
C>* 10.0.1.0/24 is directly connected, eth1, 00:04:18
C>* 10.0.4.0/24 is directly connected, eth2, 00:04:18
```

Puedes ejecutar un comando de `vtysh` directamente con la opción `-c` como en `docker exec r1 vtysh -c "show ip route"`. En la salida verás las 3 redes directamente conectadas (C) y la ruta por defecto (K) que hemos configurado. Al final de cada fila aparece el tiempo transcurrido desde que se estableció.

9.9.3. Encaminamiento RIP

Es momento de configurar RIP en nuestra topología. Partiendo del despliegue original, es necesario proporcional un archivo de configuración para cada uno de los routers. Por ejemplo, para el router `r1` el archivo es:

```
router rip
    network 10.0.0.0/24
    network 10.0.1.0/24
    network 10.0.4.0/24
```

[4] `ctysh` está incluido en el paquete `frr`.

Listado 9.2: Configuración RIP en el laboratorio de encaminamiento Delta
/lab-routing-delta/rip/R1-ripd.conf

Esta configuración le indica al router las redes que debe anunciar. Los 3 routers anuncian las 3 redes a las que están conectados. Para utilizar el protocolo RIP, el router debe ejecutar el servicio `ripd`. El archivo de configuración lo montamos en el propio archivo `compose.yml`. Para ejecutar el servicio `ripd` usamos también `docker exec`. El script `setup-rip.sh` lo hace para los 3 routers.

```
$ docker exec $router /usr/lib/frr/ripd -f /etc/frr/ripd.conf -d
```

Para reiniciar la topología, aplicar los scripts que hemos visto y configurar el encaminamiento RIP simplemente ejecuta:

```
$ make rip
```

Puedes consultar la configuración de cualquiera de los routers con `vtysh`:

```
$ docker exec r3 vtysh -c "show running-config"
Building configuration...

Current configuration:
!
frr version 8.4.4
frr defaults traditional
hostname r3
no ipv6 forwarding
service integrated-vtysh-config
!
router rip
 network 10.0.2.0/24
 network 10.0.3.0/24
 network 10.0.4.0/24
exit
!
end
```

Ahora espera hasta que RIP converja. Para eso puedes simplemente hacer `ping` a `Server` desde tu PC y esperar. Puede tardar alrededor de un minuto. Hasta que las rutas se establezcan, los mensajes no podrán llegar hasta `Server` y verás mensajes de error que te lo indican. Cuando RIP haya establecido las rutas, `ping` empezará a funcionar.

```
$ ping 10.0.3.3
PING 10.0.3.3 (10.0.3.3) 56(84) bytes of data.
From 10.0.0.3 icmp_seq=1 Destination Net Unreachable
From 10.0.0.3 icmp_seq=2 Destination Net Unreachable
From 10.0.0.3 icmp_seq=27 Destination Net Unreachable
64 bytes from 10.0.3.3: icmp_seq=47 ttl=62 time=0.092 ms
64 bytes from 10.0.3.3: icmp_seq=48 ttl=62 time=0.072 ms
64 bytes from 10.0.3.3: icmp_seq=49 ttl=62 time=0.075 ms
```

En ese momento puedes comprobar la tabla de encaminamiento de cualquiera de los routers con `ip route`:

```
$ docker exec r3 ip route
10.0.0.0/24 nhid 9 via 10.0.4.2 dev eth2 proto rip metric 20
10.0.1.0/24 nhid 10 via 10.0.2.2 dev eth0 proto rip metric 20
10.0.2.0/24 dev eth0 proto kernel scope link src 10.0.2.3
10.0.3.0/24 dev eth1 proto kernel scope link src 10.0.3.2
10.0.4.0/24 dev eth2 proto kernel scope link src 10.0.4.3
```

Fíjate en el *proto rip* en las dos primeras filas. Indica que esas rutas las ha aprendido por medio de RIP. También puedes comprobar la tabla de encaminamiento de `r1` con `vtysh`:

```
$ docker exec r3 vtysh -c "show ip route"
Codes: K - kernel route, C - connected, S - static, R - RIP,
       O - OSPF, I - IS-IS, B - BGP, E - EIGRP, N - NHRP,
       T - Table, v - VNC, V - VNC-Direct, A - Babel, F - PBR,
       f - OpenFabric,
       > - selected route, * - FIB route, q - queued, r - rejected, b - backup
       t - trapped, o - offload failure

R>* 10.0.0.0/24 [120/2] via 10.0.4.2, eth2, weight 1, 00:00:53
R>* 10.0.1.0/24 [120/2] via 10.0.2.2, eth0, weight 1, 00:00:53
C>* 10.0.2.0/24 is directly connected, eth0, 00:00:54
C>* 10.0.3.0/24 is directly connected, eth1, 00:00:54
C>* 10.0.4.0/24 is directly connected, eth2, 00:00:54
```

También puedes comprobar el estado del protocolo RIP con `vtysh`:

```
$ docker exec r3 vtysh -c "show ip rip"
Codes: R - RIP, C - connected, S - Static, O - OSPF, B - BGP
Sub-codes:
      (n) - normal, (s) - static, (d) - default, (r) - redistribute,
      (i) - interface

     Network            Next Hop         Metric From            Tag Time
R(n) 10.0.0.0/24        10.0.4.2              2 10.0.4.2          0 02:45
R(n) 10.0.1.0/24        10.0.2.2              2 10.0.2.2          0 02:50
C(i) 10.0.2.0/24        0.0.0.0               1 self              0
C(i) 10.0.3.0/24        0.0.0.0               1 self              0
C(i) 10.0.4.0/24        0.0.0.0               1 self              0
```

Prueba también `vtysh -c "show ip rip status"` para obtener información adicional.

Fíjate que aunque hay dos formas de llegar desde `r3` a la red Ṇ o desde `r1` a la red `N2`, RIP solo anuncia una de ellas.

Igual que para encaminamiento estático, puedes comprobar la conectividad con `Server` (o cualquiera de las routers) desde tu PC mediante `ping`, `traceroute` o `ncat`.

9.9.4. Encaminamiento OSPF

Por último, vamos a configurar OSPF en la topología Delta, partiendo del despliegue básico. Es necesario proporcionar un archivo de configuración para el servicio `ospfd`, aunque en este caso podemos usar el mismo archivo para los 3 routers. El archivo es `ospfd.conf` y se monta también en el archivo `compose.yml`. El archivo contiene simplemente:

```
router ospf
  redistribute connected
  network 0.0.0.0/0 area 0
\caption[Configuración de \ac{OSPF} en el laboratorio de encaminamiento Delta]
  {Configuración \ac{OSPF} en el laboratorio de encaminamiento Delta\\
  \codexample{lab-routing-delta/ospfd.conf}}
```

Puedes reiniciar la topología y configurar OSPF con:

```
$ make ospf
```

Como antes, puedes consultar la configuración de cualquiera de los routers con `vtysh`:

```
$ docker exec r3 vtysh -c "show running-config"
Building configuration...

Current configuration:
!
frr version 8.4.4
frr defaults traditional
hostname r3
no ipv6 forwarding
service integrated-vtysh-config
!
router ospf
 redistribute connected
 network 0.0.0.0/0 area 0
exit
!
end
```

Y también, cuando el protocolo haya convergido, puedes comprobar las tablas de encaminamiento con `ip route`. En este caso pueden pasar hasta 2 minutos hasta que las tablas se estabilicen.

```
$ docker exec r3 ip route
10.0.0.0/24 nhid 16 via 10.0.4.2 dev eth2 proto ospf metric 20
10.0.1.0/24 nhid 17 proto ospf metric 20
        nexthop via 10.0.4.2 dev eth2 weight 1
        nexthop via 10.0.2.2 dev eth0 weight 1
10.0.2.0/24 dev eth0 proto kernel scope link src 10.0.2.3
10.0.3.0/24 dev eth1 proto kernel scope link src 10.0.3.2
10.0.4.0/24 dev eth2 proto kernel scope link src 10.0.4.3
```

Aquí puedes ver que OSPF, a diferencia de RIP, proporciona dos formas de llegar a la red `N1`: a través de `r1` y de `r2`. También puedes verlo con `vtysh`:

```
$ docker exec r3 vtysh -c "show ip route"
Codes: K - kernel route, C - connected, S - static, R - RIP,
       O - OSPF, I - IS-IS, B - BGP, E - EIGRP, N - NHRP,
       T - Table, v - VNC, V - VNC-Direct, A - Babel, F - PBR,
       f - OpenFabric,
       > - selected route, * - FIB route, q - queued, r - rejected, b - backup
       t - trapped, o - offload failure

O>* 10.0.0.0/24 [110/20] via 10.0.4.2, eth2, weight 1, 00:01:35
O>* 10.0.1.0/24 [110/20] via 10.0.2.2, eth0, weight 1, 00:01:35
  *                      via 10.0.4.2, eth2, weight 1, 00:01:35
O   10.0.2.0/24 [110/10] is directly connected, eth0, weight 1, 00:02:20
C>* 10.0.2.0/24 is directly connected, eth0, 00:02:20
O   10.0.3.0/24 [110/10] is directly connected, eth1, weight 1, 00:02:20
C>* 10.0.3.0/24 is directly connected, eth1, 00:02:20
O   10.0.4.0/24 [110/10] is directly connected, eth2, weight 1, 00:02:20
C>* 10.0.4.0/24 is directly connected, eth2, 00:02:20
```

Fíjate que OSPF también anuncia las redes directamente conectadas, pero `Zebra` prioriza las que configuró el SO (las que llevan el prefijo `>`).

9.9.5. Reacción ante fallos

Vamos a ver ahora cómo reaccionan los protocolos de encaminamiento ante fallos. Está claro que la ruta óptima para llegar de `Host` a `Server` es `Host->r1->r2->Server`. Ya lo has comprobado con `traceroute`.

```
$ traceroute 10.0.3.3
traceroute to 10.0.3.3 (10.0.3.3), 30 hops max, 60 byte packets
 1  10.0.0.3 (10.0.0.3)  0.118 ms  0.038 ms  0.031 ms
 2  10.0.4.3 (10.0.4.3)  0.097 ms  0.083 ms  0.053 ms
 3  10.0.3.3 (10.0.3.3)  0.086 ms  0.068 ms  0.065 ms
```

Ahora vamos a desactivar la interfaz de `r1` en la red `N4` (la que tiene asignada la `10.0.4.2`) para ver cómo OSPF reacciona a este «fallo». Veamos primero qué interfaz es la que tiene esa dirección IP:

```
$ docker exec r1 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: eth0@if930: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 state UP group default
    link/ether 5e:a0:fb:3e:84:4b brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.0.1.2/24 brd 10.0.1.255 scope global eth0
       valid_lft forever preferred_lft forever
3: eth1@if933: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 state UP group default
    link/ether 2a:52:31:cc:59:20 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.0.4.2/24 brd 10.0.4.255 scope global eth1
```

```
       valid_lft forever preferred_lft forever
4: eth2@if935: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 state UP group default
    link/ether 76:54:4a:3d:54:dc brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.0.0.3/24 brd 10.0.0.255 scope global eth2
       valid_lft forever preferred_lft forever
```

La interfaz que buscamos es `eth1`. Para desactivarla, ejecuta:

```
$ docker exec r1 ip link set eth1 down
```

Comprueba que está desactivada (*DOWN*) con:

```
$ docker exec r1 ip link show eth1
3: eth1@if520: <BROADCAST,MULTICAST> mtu 1500 state DOWN mode DEFAULT group default
```

Después de converger, comprueba qué ruta siguen los paquetes hasta `Server`:

```
$ traceroute 10.0.3.3
traceroute to 10.0.3.3 (10.0.3.3), 30 hops max, 60 byte packets
 1  10.0.0.3 (10.0.0.3)  0.116 ms  0.040 ms  0.031 ms
 2  10.0.1.3 (10.0.1.3)  0.071 ms  0.050 ms  0.047 ms
 3  10.0.2.3 (10.0.2.3)  0.083 ms  0.065 ms  0.062 ms
 4  10.0.3.3 (10.0.3.3)  0.094 ms  0.079 ms  0.078 ms
```

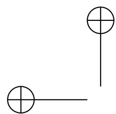
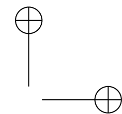
Ahora la ruta que sigue es `Host->r1->r2->r3->Server`. OSPF ha reaccionado al fallo en `N4` y ha elegido la ruta alternativa. Puedes hacer este mismo ejercicio con RIP y verás que también se adapta al fallo.

Y ¿qué más?

El encaminamiento dinámico permite a las interredes descubrir como es su topología, calcular las rutas óptimas a cada destino y adaptarse a los cambios que se produzcan, sean estos debidos a fallos puntuales o a la aparición de nuevos equipos, enlaces o rutas.

Hemos visto el funcionamiento de los algorimos más elementales: vector-distancia, estado de enlace y vector-ruta, y hemos estudiado sus problemas y limitaciones. Hemos visto en funcionamiento implementaciones reales de los protocolos RIP y OSPF, y del sistema FRR de Linux, gracias a despliegues docker sencillos, pero funcionales.

Sin encaminamiento dinámico sería imposible construir interredes de gran tamaño, incluso de mediano tamaño, y obviamente Internet no podría existir.

En todo caso, el encaminamiento dinámico genérico (como el que hemos visto aquí) tiene sus limitaciones: se centra en encontrar rutas óptimas y adaptarse a los cambios, pero no comtempla las necesidades particulares de usuarios o servicios. Existen muchas aplicaciones: videoconferencia, streaming de video, VoIP, tráfico en tiempo real; que son muy sensibles a la latencia, el jitter o cambios imprevistos en las condiciones del tráfico. Para paliar estas limitaciones existe toda una disciplina: la Calidad de Servicio. Hablaremos de ello en el capítulo 16.