Enrutamiento avanzado y control de tráfico en Linux

Bert Hubert

Netherlabs BV

<bert.hubert@netherlabs.nl>

Gregory Maxwell

<greg@linuxpower.cx>

Remco van Mook

<remco@virtu.nl>

Martijn van Oosterhout

<kleptog@cupid.suninternet.com>

Paul B Schroeder

<paulsch@us.ibm.com>

Jasper Spaans

<jasper@spaans.ds9a.nl>

Ricardo J. Cárdenes Medina

Una introducción bastante práctica a iproute2, el control de tráfico y un poco de netfilter.

Tabla de contenidos

1. Dedicatoria
2. Introducción
    2.1. Descargo de responsabilidad y licencia
    2.2. Conocimientos previos
    2.3. Qué puede hacer Linux por usted
    2.4. Notas de mantenimiento
    2.5. Acceso, CVS y envío de actualizaciones
    2.6. Lista de correo
    2.7. Disposición de este documento
3. Introducción a iproute2
    3.1. ¿Por qué iproute2?
    3.2. revisión de iproute2
    3.3. Prerequisitos
    3.4. Explorar la configuración actual
    3.4.1. **ip** nos muestra nuestros enlaces
    3.4.2. **ip** nos muestra nuestras direcciones IP
    3.4.3. **ip** nos muestra nuestras rutas
    3.5. ARP
4. Reglas (base de datos de normas de rutado)
    4.1. Normas de encaminamiento por origen sencillas
    4.2. Encaminamiento con varios enlaces de salida/proveedores
    4.2.1. Acceso dividido
    4.2.2. Equilibrio de carga
5. GRE y otros túneles
    5.1. Breve inciso sobre los túneles:
    5.2. Túneles IP sobre IP
    5.3. Túneles GRE
    5.3.1. Túneles IPv4
    5.3.2. Túneles IPv6
    5.4. Túneles en espacio de usuario
6. Túneles IPv6 con Cisco o la 6bone
    6.1. Túneles IPv6
7. IPsec: IP segura sobre Internet
8. Enrutado multicast
9. Disciplinas de colas (qdiscs) para gestión del ancho de banda
    9.1. Las colas y disciplinas de cola explicadas
    9.2. Disciplinas de cola simples, sin clases
    9.2.1. pfifo_fast
    9.2.2. Token Bucket Filter
    9.2.3. Stochastic Fairness Queueing
    9.3. Consejos sobre en qué momento usar qué cola
    9.4. Terminología
    9.5. Disciplinas de cola con clases
    9.5.1. El flujo dentro de las qdisc con clases y sus clases
    9.5.2. La familia qdisc: raíces, controladores, hermanos y padres
    9.5.3. La qdisc PRIO
    9.5.4. La famosa qdisc CBQ

9.5.5. Hierarchical Token Bucket
9.6. Clasificar paquetes con filtros
9.6.1. Algunos ejemplos sencillos de filtrado
9.6.2. Todas las órdenes de filtrado que necesitará normalmente
9.7. El dispositivo intermedio de encolado (IMQ)
9.7.1. Configuración de ejemplo
10. Compartir la carga sobre varias interfaces
10.1. Problemas
10.2. Otras posibilidades
11. Netfilter e iproute (marcado de paquetes)
12. Filtros avanzados para (re)clasificar paquetes
12.1. El clasificador `u32`
12.1.1. Selector U32
12.1.2. Selectores generales
12.1.3. Selectores específicos
12.2. El clasificador `route`
12.3. Filtros de control (Policing filters)
12.3.1. Formas de control
12.3.2. Acciones de sobrelímite
12.3.3. Ejemplos
12.4. Filtros de hash para filtrado masivo muy rápido
13. Parámetros de red del núcleo
13.1. Reverse Path Filtering
13.2. Configuraciones oscuras
13.2.1. ipv4 genérica
13.2.2. Configuración por dispositivo
13.2.3. Normas de vecinos (Neighbor policy)
13.2.4. Configuración de encaminamiento
14. Disciplinas de cola avanzadas y poco conocidas
14.1. `bfifo`/`pfifo`
14.1.1. Parámetros y uso
14.2. Algoritmo Clark-Shenker-Zhang (CSZ)
14.3. DSMARK
14.3.1. Introducción
14.3.2. ¿Con qué se relaciona Dsmark?
14.3.3. Principios de los Servicios Diferenciados
14.3.4. Trabajar con Dsmark
14.3.5. Cómo trabaja SCH_DSMARK.
14.3.6. Filtro TC_INDEX
14.4. Qdisc de entrada (Ingress)
14.4.1. Parámetros y uso
14.5. Random Early Detection (RED)
14.6. Generic Random Early Detection
14.7. Emulación VC/ATM
14.8. Weighted Round Robin (WRR)
15. Recetario
15.1. Llevar varios sitios con diferentes SLA
15.2. Proteger la máquina frente a inundaciones SYN
15.3. Limitar la tasa de ICMP para prevenir dDoS
15.4. Priorizado de tráfico interactivo

- 15.5. Caché transparente de web usando netfilter, iproute2, ipchains y squid
- 15.5.1. Diagrama de flujo del tráfico tras la implementación
- 15.6. Sortear los problemas de Path MTU Discovery con configuraciones de MTU por ruta
- 15.6.1. Solución
- 15.7. Sortear los problemas de Path MTU Discovery con MSS Clamping (para usuarios de ADSL, cable, PPPoE y PPtP)
- 15.8. El acondicionador de tráfico definitivo: baja latencia, envíos y descargas rápidos
- 15.8.1. Por qué no funciona bien por defecto
- 15.8.2. El script (CBQ)
- 15.8.3. El script (HTB)
- 15.9. Limitar la tasa a una única máquina o máscara de red

16. Hacer bridges y pseudo-bridges con Proxy ARP
    - 16.1. Estado del bridging e iptables
    - 16.2. Bridging y ajustes (shaping)
    - 16.3. Pseudo-bridges con Proxy-ARP
    - 16.3.1. ARP y Proxy-ARP
    - 16.3.2. Implementándolo
17. Encaminamiento dinámico - OSPF y BGP
18. Otras posibilidades
19. Further reading
20. Acknowledgements

Capítulo 1. Dedicatoria

Este documento está dedicado a un montón de personas, y es mi intención devolverles algo. Una lista de unos pocos:

- Rusty Russell
- Alexey N. Kuznetsov
- Los buenos chicos de Google
- El personal de Casema Internet

Capítulo 2. Introducción

Bienvenido, gentil lector.

Con este documento espero iluminarte en cómo hacer más con el enrutamiento de Linux 2.2/2.4. Aunque la mayoría de los usuarios lo desconozca, están usando herramientas que permiten hacer cosas espectaculares. Ordenes tales como **route** e **ifconfig** en realidad son envolturas realmente delgadas alrededor de la poderosa infraestructura de iproute2.

Espero que este Cómo sea tan legible como los de Rusty Russell, famoso (entre otras cosas) por netfilter.

Siempre puede localizarnos escribiendo al equipo del HOWTO (hágalo en inglés). Sin embargo, le pedimos que considere enviar el mensaje a la lista de correo (vea la sección correspondiente) si tiene dudas que no estén relacionadas directamente con este documento. No somos un servicio de atención gratuita, aunque a menudo respondamos a las preguntas que nos hacen en la lista.

Antes de que se pierda en este Cómo, si lo único que desea es hacer un control de tráfico sencillo, sáltese todo lo demás y vaya directo al capítulo *Other possibilities*, y lea sobre CBQ.init.

2.1. Descargo de responsabilidad y licencia

Este documento se distribuye con la esperanza de que sea útil, pero SIN NINGUN TIPO DE GARANTIA; incluso sin las garantías implícitas MERCANTILES o DE ADECUACION PARA UN PROPOSITO PARTICULAR.

En breve, si su backbone STM-64 se desconfigura y empieza a repartir pornografía a sus clientes más estimados, no es culpa nuestra. Lo sentimos.

Copyright (c) 2002 by bert hubert, Gregory Maxwell, Martijn van Oosterhout, Remco van Mook, Paul B. Schroeder and others. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v1.0 or later (the latest version is presently available at http://www.opencontent.org/openpub/).

Copie y distribuya (vendiéndolo o regalándolo) este documento en cualquier formato. Le pedimos que envíe correcciones y comentarios al mantenedor del documento.

También le pedimos que si publica este Cómo en papel, envíe algunas muetras a los autores para "propósitos de revisión" :-)

2.2. Conocimientos previos

Tal como implica el título, este es un Cómo "Avanzado". Aunque no es ciencia espacial de ninguna manera, se asumen ciertos conocimientos previos.

Aquí hay algunas referencias que pueden ayudarle a aprender algo más:

El Cómo de conceptos de redes de Rusty Russell. También dispone de una versión en castellano.

> Muy buena introducción, que explica lo que es una red, y cómo se conecta con otras.

Redes en Linux Cómo (Previamente Net-3 Como)

> Gran material, aunque un poco exhaustivo. Le enseña un montón de cosas que ya deben estar configuradas si es capaz de conectar a Internet. Debería estar en `/usr/doc/HOWTO/NET3-4-HOWTO.txt` pero también lo puede encontrar en línea (y en castellano aquí).

2.3. Qué puede hacer Linux por usted

Una pequeña lista de cosas posibles:

- Mejorar el ancho de banda de ciertos computadores
- Mejorar el ancho de banda HACIA ciertos computadores
- Ayudarle a compartir su ancho de banda de forma justa
- Proteger su red ante ataques DoS
- Proteger a Internet de sus clientes
- Multiplexar varios servidores como uno solo, para equilibrio de carga o disponibilidad mejorada
- Restringir el acceso a sus computadores
- Limitar el acceso de sus usuarios a otras máquinas
- Hacer enrutamiento basado en el id de los usarios (¡sí!), dirección MAC, dirección IP de origen, puerto, tipo de servicio, hora del día o contenido

Actualmente, no mucha gente usa estas capacidades avanzadas. Esto sucede por varias razones. Mientras que la documentación existente es exahustiva, no es muy práctica. El control de tráfico está casi sin documentar.

2.4. Notas de mantenimiento

Hay varias cosas que habría que resaltar sobre este documento. Aunque que lo he escrito en su mayoría, realmente no quería que fuese así. Creo mucho en el Open Source, de manera que le animo a que me envíe sus impresiones, actualizaciones, parches, etc. No dude en informarme de fallos de ortografía o errores por antigüedad. (Nota: lo siguiente sólo tiene sentido en el documento original) Si mi inglés parece algo acartonado, por favor, tenga en cuenta que no soy un hablante nativo. Siéntase libre de enviarme sugerencias.

Si cree que está mejor cualificado para mantener una sección, o piensa que puede crear y mantener nuevas secciones, sea bienvenido. El SGML de este Cómo está disponible vía CVS, y me gustaría que trabajase más gente en él.

Como ayuda para esto, podrá encontrar varias notas FIXME. ¡Los parches siempre vienen bien! Cuando se encuentre con un FIXME, debería saber que está entrando en territorio desconocido. Esto no quiere decir que no haya errores en otras partes, pero tenga especial cuidado. Si ha comprobado que algo es válido, por favor, háganoslo saber para eliminar la nota de FIXME.

Sobre este Cómo, me voy a tomar una serie de libertades. Por ejemplo, postulo una conexión a Internet de 10Mbit, aunque sé muy bien que esto no es nada común.

2.5. Acceso, CVS y envío de actualizaciones

El lugar canónico de este Cómo es éste.

Ahora disponemos de acceso anónimo a CVS para todo el mundo. Esto es bueno por varias razones. Puede actualizar de forma sencilla a nuevas versiones de este Cómo y enviar parches es fácil.

Más aún, permite a los autores trabajar en el fuente de forma independiente, lo cual es bueno.

```
$ export CVSROOT=:pserver:anon@outpost.ds9a.nl:/var/cvsroot
$ cvs login
CVS password: [introduzca "cvs" (sin comillas)]
$ cvs co 2.4routing
cvs server: Updating 2.4routing
U 2.4routing/lartc.db
```

Si ha hecho cambios y quiere contribuir con ellos, ejecute **`cvs -z3 diff -uBb`**, y envíe el resultado a <howto@ds9a.nl>, que nosotros podemos integrarlo de forma sencilla. ¡Gracias! Por favor, asegúrese de que edita el fichero .db. Por cierto, los otros ficheros se generan partiendo de éste.

Se proporciona un Makefile que le ayudará a crear postscript, dvi, pdf, html y texto plano. Puede que necesite instalar docbook, docbook-utils, ghostscript y tetex para obtener todos los formatos.

¡Procure no editar 2.4routing.sgml! Contiene una versión antigua de este HOWTO. El fichero correcto es lartc.db.

2.6. Lista de correo

Los autores reciben un creciente número de mensajes sobre este Cómo. Debido al claro interés de la comunidad, se ha decidido crear una lista de correo donde la gente pueda hablar entre sí sobre Advanced Routing and Traffic Control. Puede suscribirse a la lista (se habla en inglés) aquí.

Debería precisar que los autores no suelen responder preguntas que no se hagan a la lista. Nos gustaría que el archivo de la lista se convirtiera en algo así como una base de conocimiento. Si tiene una pregunta, sírvase buscar antes en el archivo, y sólo después envíela a la lista.

2.7. Disposición de este documento

Vamos a empezar a hacer cosas interesantes casi inmediatamente, lo que también significa que inicialmente habrán partes no del todo explicadas, o que no estén perfectas. Por favor, lea superficialmente esas partes y asuma que todo se aclarará más adelante.

Enrutar y filtrar son dos cosas distintas. El filtrado está bastante bien documentado en los Cómo de Rusty, disponibles aquí:

- Rusty's Remarkably Unreliable Guides

Vamos a centrarnos principalmente en las posibilidades de combinar netfilter e iproute2.

Capítulo 3. Introducción a iproute2

3.1. ¿Por qué iproute2?

La mayoría de las distribuciones de Linux, y la mayoría de los UNIX, usan actualmente las venerables órdenes **arp**, **ifconfig** y **route** Aunque funcionan, muestran cierto comportamiento inesperado apartir de Linux 2.2. Por ejemplo, los túneles GRE son parte integral del enrutado hoy día, pero precisan herramientas completamente diferentes.

Con iproute2, los túneles son una parte integral del juego de herramientas.

Los núcleos Linux 2.2 y superiores incluyen un subsistema de red completamente rediseñado. Este nuevo código de red proporciona a Linux un rendimiento y características con poca competencia en el panorama general de los SO. En realiadd, el nuevo código de enrutado, filtrado y clasificación tiene más posibilidades que el que porporcionan muchos enrutadores y cortafuegos dedicados y productos de control de tráfico.

Según se inventan nuevos conceptos de red, la gente encuentra maneras de emplastarlos encima de la infraestructura existente en los SO. Este continuo apilamiento de porquería ha llevado a código de red lleno de comportamientos extraños, muy parecido a lo que sucede con los idiomas humanos. En el pasado, Linux emuló la forma de SunOS de gestionar muchas de estas cosas, pero no era ideal.

Esta nueva infraestructura hace posible expresar claramente características que antes estaban más allá del alcance de Linux.

3.2. revisión de iproute2

Linux tiene un sistema sofisticado para proporcionar ancho de banda llamado Traffic Control. Este sistema soporta varios métodos de clasificación, priorizado, compartición y limitación tanto de tráfico entrante como saliente.

Empezaremos con un pequeño «tour» por las posibilidades de iproute2.

3.3. Prerequisitos

Debería asegurarse de que tiene instaladas las herramientas de espacio de usuario. Este paquete se llama «iproute» tanto en RedHat como en Debian, y en cualquier caso, puede encontrarlo en `ftp://ftp.inr.ac.ru/ip-routing/iproute2-2.2.4-now-ss??????.tar.gz"`.

También puede buscar aquí la última versión.

Algunas partes de iproute precisan que active ciertas opciones del núcleo. También debería saber que todas las versiones de RedHat hasta la 6.2, incluida, vienen con la mayoría de capacidades de control de tráfico en el núcleo de serie.

RedHat 7.2 lo tiene todo de serie.

También debe asegurarse de que tiene soporte de netlink, en caso de que escoja configurar su propio núcleo. Iproute2 lo necesita.

3.4. Explorar la configuración actual

Puede que le sorprenda, ¡pero iproute2 ya está configurado! Las órdenes **ifconfig** y **route** actuales ya usan las llamadas a sistema avanzadas, pero en su mayoría con configuraciones por defecto (es decir, aburridas).

La herramienta **ip** es central, y le pediremos que nos muestre nuestras interfaces.

3.4.1. ip nos muestra nuestros enlaces

```
[ahu@home ahu]$ ip link list
1: lo: <LOOPBACK,UP> mtu 3924 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: dummy: <BROADCAST,NOARP> mtu 1500 qdisc noop
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
3: eth0: <BROADCAST,MULTICAST,PROMISC,UP> mtu 1400 qdisc pfifo_fast
qlen 100
    link/ether 48:54:e8:2a:47:16 brd ff:ff:ff:ff:ff:ff
4: eth1: <BROADCAST,MULTICAST,PROMISC,UP> mtu 1500 qdisc pfifo_fast
qlen 100
    link/ether 00:e0:4c:39:24:78 brd ff:ff:ff:ff:ff:ff
3764: ppp0: <POINTOPOINT,MULTICAST,NOARP,UP> mtu 1492 qdisc pfifo_fast
qlen 10
    link/ppp
```

Puede que para usted varíe, pero esto es lo que muestra mi enrutador NAT en casa. Sólo voy a explicar parte de la salida ya que no todo es directamente relevante.

Primero vemos la interfaz loopback. Aunque su computador puede funcionar sin una, le advierto que no lo haga. El tamaño MTU (Maximum Transfer Unit) es de 3924 octetos, y no se supone que deba encolar. Lo cual tiene sentido porque la interfaz loopback es una fantasía en la imaginación del núcleo.

Dejaré de lado por ahora la interfaz dummy, que puede no estar presente en su computador. Después están mis dos interfaces de red físicas, una está del lado de mi cable módem, y la otra sirve a mi segmento ethernet casero. Más aún, vemos una interfaz ppp0.

Observe la ausencia de direcciones IP. iproute desconecta los conceptos de «enlace» y «dirección IP». De todas maneras, con el alias de IP, el concepto de «la» dirección IP se ha vuelto bastante irrelevante.

Sin embargo, nos muestra las direcciones MAC, el identificador en hardware de nuestras interfaces ethernet.

3.4.2. ip nos muestra nuestras direcciones IP

```
[ahu@home ahu]$ ip address show
1: lo: <LOOPBACK,UP> mtu 3924 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 brd 127.255.255.255 scope host lo
2: dummy: <BROADCAST,NOARP> mtu 1500 qdisc noop
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
3: eth0: <BROADCAST,MULTICAST,PROMISC,UP> mtu 1400 qdisc pfifo_fast
qlen 100
    link/ether 48:54:e8:2a:47:16 brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.1/8 brd 10.255.255.255 scope global eth0
4: eth1: <BROADCAST,MULTICAST,PROMISC,UP> mtu 1500 qdisc pfifo_fast
qlen 100
    link/ether 00:e0:4c:39:24:78 brd ff:ff:ff:ff:ff:ff
3764: ppp0: <POINTOPOINT,MULTICAST,NOARP,UP> mtu 1492 qdisc pfifo_fast
qlen 10
    link/ppp
    inet 212.64.94.251 peer 212.64.94.1/32 scope global ppp0
```

Esto contiene más información. Nos muestra todas nuestras direcciones, y a qué tarjetas pertenecen. «inet» significa Internet (IPv4). Hay muchas otras familias de direcciones, pero no nos importan por el momento.

Examinemos eth0 más de cerca. Dice que está relacionada con la dirección inet «10.0.0.1/8». ¿Qué significa esto? El /8 indica el número de bits que están en la Dirección de Red. Hay 32 bit, de manera que quedan 24 bits que son parte de nuestra red. Los primeros 8 bits de 10.0.0.1 corresponden a 10.0.0.0, nuestra Dirección de Red, y nuestra máscara de red (netmask) es 255.0.0.0.

Las otras máquinas están conectadas a esta interfaz, de manera que 10.250.3.13 es accesible de forma directa desde eth0, al igual que 10.0.0.1, por ejemplo.

Con ppp0, vemos el mismo concepto, aunque los números son diferentes. Su dirección es 212.64.94.251, sin máscara de subred. Esto significa que tenemos una conexión punto a punto y que cada dirección, con la excepción de 212.64.94.251, es remota. Hay más información, sin embargo. Nos dice que en la otra punta del enlace hay, de nuevo, una única dirección, 212.64.94.1. El /32 nos dice que no hay «bits de red».

Es absolutamente vital que comprenda bien estos conceptos. Remítase a la documentación mencionada al principio de este Cómo si tiene problemas.

También observará «qdisc», que significa Disciplina de Cola (Queueing Discipline). Más adelante veremos que es vital.

3.4.3. ip nos muestra nuestras rutas

Bien, ya sabemos cómo encontrar direcciones 10.x.y.z, y somos capaces de alcanzar 212.64.94.1. Sin embargo, esto no es suficiente, de manera que necesitamos instrucciones sobre cómo alcanzar al resto del mundo. La Internet está disponible

mediante nuestra conexión ppp, y parece que 212.64.94.1 está deseando esparcir nuestros paquetes por el mundo, y entregarnos resultados de vuelta.

```
[ahu@home ahu]$ ip route show
212.64.94.1 dev ppp0  proto kernel  scope link  src 212.64.94.251
10.0.0.0/8 dev eth0  proto kernel  scope link  src 10.0.0.1
127.0.0.0/8 dev lo  scope link
default via 212.64.94.1 dev ppp0
```

Bastante explícito. Las primeras 4 líneas indican explícitamente lo que quedó implícito con **ip address show**, y la última línea nos dice que el resto del mundo lo podemos encontrar mediante 212.64.94.1, nuestra pasarela por defecto. Podemos saber que es una pasarela por la palabra «via», que nos dice que necesitamos enviar paquetes a 212.64.94.1, que ya se encargará del resto.

Como referencia, esto es lo que la vieja utilidad **route** nos muestra:

```
[ahu@home ahu]$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref
Use
Iface
212.64.94.1     0.0.0.0         255.255.255.255 UH    0      0
0 ppp0
10.0.0.0        0.0.0.0         255.0.0.0       U     0      0
0 eth0
127.0.0.0       0.0.0.0         255.0.0.0       U     0      0
0 lo
0.0.0.0         212.64.94.1     0.0.0.0         UG    0      0
0 ppp0
```

3.5. ARP

ARP es el Address Resolution Protocol que se describe en el RFC 826. ARP lo usa una máquina en red para averiguar la localización/dirección hardware de otra máquina en la misma red local. Las máquinas en Internet se conocen generalmente por sus nombres que se corresponden a direcciones IP. Así es como una máquina en la red foo.com es capaz de comunicarse con otras máquinas que están en la red bar.net. Una dirección IP, sin embargo, no puede decirte la localización física de una máquina. Aquí es donde entra ARP.

Tomemos un ejemplo muy sencillo. Supongamos que tengo una red compuesta de varias máquinas. Dos de ellas que están en mi red son foo con dirección IP 10.0.0.1 y bar con dirección IP 10.0.0.2. Ahora foo quiere hacer ping hacia bar para ver si está viva, pero, ¡vaya!, foo no tiene idea de dónde está bar. De manera que cuando foo decide hacer ping hacia bar necesita realizar una consulta ARP. Esta consulta ARP es algo así como si foo gritase en la red «¡Bar (10.0.0.2)! ¿Dónde estás?» Como resultado de esto, cada máquina de la red escuchará el grito de foo, pero sólo bar (10.0.0.2) responderá. Bar enviará entonces una respuesta ARP directamente a foo, que viene a ser como si bar dijese, «Foo (10.0.0.1), estoy aquí en 00:60:94:E9:08:12». Después de esta sencilla transacción que sirve para localizar a su amigo en la red, foo es capaz de

comunicarse con bar hasta que olvide (su caché arp) dónde está bar (normalmente tras 15 minutos, en Unix).

Ahora, veamos cómo funciona. Puede ver la caché/tabla arp/neighbor actual de su máquina así:

```
[root@espa041 /home/src/iputils]# ip neigh show
9.3.76.42 dev eth0 lladdr 00:60:08:3f:e9:f9 nud reachable
9.3.76.1 dev eth0 lladdr 00:06:29:21:73:c8 nud reachable
```

Como puede ver, mi máquina espa041 (9.3.76.41) sabe dónde encontrar a espa042 (9.3.76.42) y espagate (9.3.76.1). Ahora añadamos otra máquina a la caché arp.

```
[root@espa041 /home/paulsch/.gnome-desktop]# ping -c 1 espa043
PING espa043.austin.ibm.com (9.3.76.43) from 9.3.76.41 : 56(84) bytes
of data.
64 bytes from 9.3.76.43: icmp_seq=0 ttl=255 time=0.9 ms

--- espa043.austin.ibm.com ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.9/0.9/0.9 ms

[root@espa041 /home/src/iputils]# ip neigh show
9.3.76.43 dev eth0 lladdr 00:06:29:21:80:20 nud reachable
9.3.76.42 dev eth0 lladdr 00:60:08:3f:e9:f9 nud reachable
9.3.76.1 dev eth0 lladdr 00:06:29:21:73:c8 nud reachable
```

Como resultado de que espa041 intente contactar con espa043, se ha añadido la dirección/localización hardware de espa043 a la caché arp/neighbor. De manera que mientras no caduque la entrada de espa043 (como resultado de la ausencia de comunicación entre ambas), espa041 sabe dónde encontrar a espa043 y no necesita enviar una consulta ARP.

Ahora, eliminemos a espa043 de nuestra caché arp:

```
[root@espa041 /home/src/iputils]# ip neigh delete 9.3.76.43 dev eth0
[root@espa041 /home/src/iputils]# ip neigh show
9.3.76.43 dev eth0  nud failed
9.3.76.42 dev eth0 lladdr 00:60:08:3f:e9:f9 nud reachable
9.3.76.1 dev eth0 lladdr 00:06:29:21:73:c8 nud stale
```

Ahora espa041 ha vuelto a olvidar dónde encontrar a espa043 y necesitará enviar otra consulta ARP la siguiente vez que necesite comunicarse con espa043. También puede ver en el listado anterior que espagate (9.3.76.1) ha cambiado al estado «stale». Esto significa que la localización mostrada todavía es válida, pero tendrá que ser confirmada en la primera transacción que se haga con esa máquina.

Capítulo 4. Reglas (base de datos de normas de rutado)

Si tiene una red grande, probablemente tenga que encargarse de las necesidades de diferentes personas, que deberían ser servidas de forma diferente. La base de datos de normas de rutado (routing policy database) le permite hacerlo teniendo varios conjuntos de tablas de rutado.

Si quiere usar esta característica, asegúrese de que compila su núcleo con las opciones «IP: advanced router» e «IP: policy routing».

Cuando el núcleo necesita tomar una decisión de encaminamiento, busca la tabla que necesita consultar. Por defecto, hay tres tablas. La antigua herramienta "route" modifica las tablas principal y local, al igual que la herramienta ip (por defecto).

Las reglas por defecto:

```
[ahu@home ahu]$ ip rule list
0:      from all lookup local
32766:  from all lookup main
32767:  from all lookup default
```

Aquí se lista la priorida de todas las reglas. Vemos que todas son aplicables a todos los paquetes («from all»). Hemos visto anteriormente la tabla «main», mostrada por **`ip route ls`**, pero las tablas «local» y «default» son nuevas.

Si queremos hacer cosas interesantes, generaremos reglas que se refieran a diferentes tablas que nos permitirán saltarnos las reglas generales de rutado del sistema.

Si desea ver la semántica exacta de lo que hace el núcleo cuando hay más de una regla válida, vea la documentación ip-cref de Alexey.

4.1. Normas de encaminamiento por origen sencillas

Tomando de nuevo un ejemplo real, tengo 2 cable módems (en realidad 3, pero acabé devolviendo uno), conectados a un router Linux NAT («masquerading»). La gente que vive aquí me paga por acceder a Internet. Suponga que uno de mis compañeros de casa sólo visita hotmail y desea pagar menos. Esto me parece bien, pero acabará usando el cable módem de menos prestaciones.

El cable módem «rápido» se conoce como 212.64.94.251 y es un enlace PPP a 212.64.94.1. El «lento» es conocido por varias IP, por ejemplo 212.64.78.148, y es un enlace a 195.96.98.253.

La tabla local:

```
[ahu@home ahu]$ ip route list table local
broadcast 127.255.255.255 dev lo  proto kernel  scope link  src
127.0.0.1
local 10.0.0.1 dev eth0  proto kernel  scope host  src 10.0.0.1
broadcast 10.0.0.0 dev eth0  proto kernel  scope link  src 10.0.0.1
local 212.64.94.251 dev ppp0  proto kernel  scope host  src
212.64.94.251
broadcast 10.255.255.255 dev eth0  proto kernel  scope link  src
10.0.0.1
broadcast 127.0.0.0 dev lo  proto kernel  scope link  src 127.0.0.1
local 212.64.78.148 dev ppp2  proto kernel  scope host  src
212.64.78.148
local 127.0.0.1 dev lo  proto kernel  scope host  src 127.0.0.1
local 127.0.0.0/8 dev lo  proto kernel  scope host  src 127.0.0.1
```

Montón de cosas obvias, pero que hace falta especificar en algún sitio. Bien, aquí están. La tabla por defecto está vacía.

Veamos la tabla «main»:

```
[ahu@home ahu]$ ip route list table main
195.96.98.253 dev ppp2  proto kernel  scope link  src 212.64.78.148
212.64.94.1 dev ppp0  proto kernel  scope link  src 212.64.94.251
10.0.0.0/8 dev eth0  proto kernel  scope link  src 10.0.0.1
127.0.0.0/8 dev lo  scope link
default via 212.64.94.1 dev ppp0
```

Ahora generaremos una nueva regla que llamaremos «John», por nuestro hipotético compañero. Aunque podemos trabajar con números, es mucho más sencillo añadir nuestras tablas a /etc/iproute2/rt_tables.

```
# echo 200 John >> /etc/iproute2/rt_tables
# ip rule add from 10.0.0.10 table John
# ip rule ls
0:      from all lookup local
32765:  from 10.0.0.10 lookup John
32766:  from all lookup main
32767:  from all lookup default
```

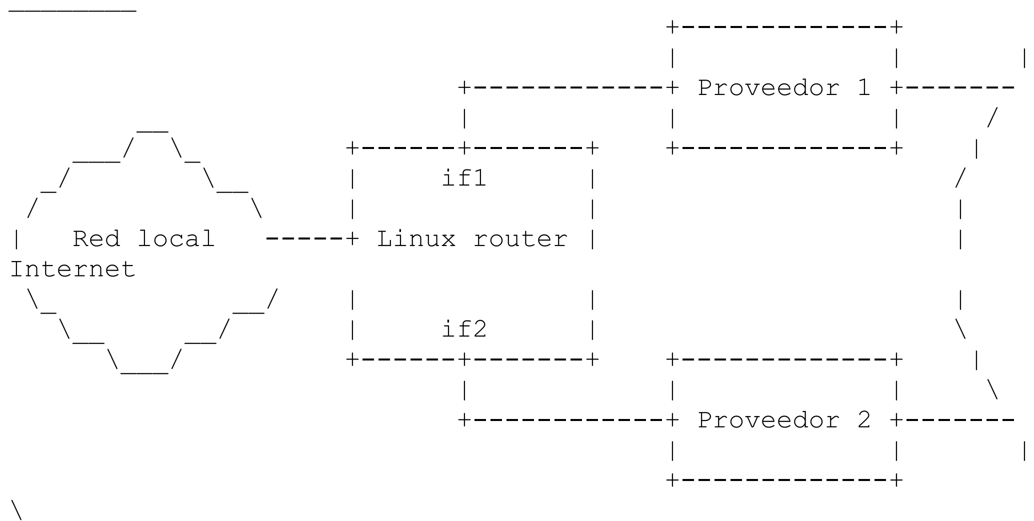
Ahora todo lo que queda es generar la tabla John, y refrescar la caché de rutas:

```
# ip route add default via 195.96.98.253 dev ppp2 table John
# ip route flush cache
```

Ya hemos terminado. Dejamos como ejercicio al lector implementar esto en ip-up.

4.2. Encaminamiento con varios enlaces de salida/proveedores

La siguiente es una configuración común, en la que hay dos proveedores que conectan una red local (o incluso una única máquina) a la gran Internet.



Normalmente surgen dos preguntas dada esta configuración.

4.2.1. Acceso dividido

La primera es cómo enrutar respuestas a paquetes que vienen de un proveedor particular, por ejemplo Proveedor 1, de vuelta por el mismo camino.

Primero establezcamos algunos nombres simbólicos. Digamos que **$IF1** es el nombre de la primera interfaz (if1 en la figura) y **$IF2** el nombre de la segunda. Sean entonces **$IP1** la dirección IP asociada con **$IF1** y **$IP2** la IP asociada con **$IF2**. Luego, digamos que **$P1** es la dirección IP de la pasarela a Proveedor 1, y **$P2** la IP de la pasarela a Proveedor 2. Por último, **$P1_NET** será la red IP donde está **$P1**, y **$P2_NET** la red IP donde está **$P2**.

Creamos dos tablas de encaminamiento adicionales, llamémoslas **T1** y **T2**. Las añadimos a /etc/iproute2/rt_tables. Entonces las configuramos de la siguiente manera:

```
ip route add $P1_NET dev $IF1 src $IP1 table T1
ip route add default via $P1 table T1
ip route add $P2_NET dev $IF2 src $IP2 table T2
ip route add default via $P2 table T2
```

Nada espectacular; simplemente hemos montado una ruta hacia una pasarela, y una ruta por defecto mediante ella, tal como sería el caso con un único proveedor, pero ponemos las rutas en tablas separadas, una por proveedor. Observe que basta la ruta hacia la red, ya que le indica cómo encontrar cualquier máquina dentro de esa red, lo que incluye la pasarela, como se especificó anteriormente.

Después configuramos la tabla de rutas principal (main). Es una buena idea encaminar las cosas a vecinos directos mediante la interfaz conectada a ese vecino. Observe las opciones «src», que se aseguran que se escoge la dirección IP correcta.

```
ip route add $P1_NET dev $IF1 src $IP1
ip route add $P2_NET dev $IF2 src $IP2
```

Luego, la ruta por defecto preferente:

```
ip route add default via $P1
```

A continuación, configuramos las reglas de encaminamiento. Estas son las que escogen qué tabla de rutas se usa. Querrá asegurarse de que encamina por una interfaz dada si ya tenemos una dirección de origen correspondiente:

```
ip rule add from $IP1 table T1
ip rule add from $IP2 table T2
```

Estas órdenes se aseguran de que todas las respuestas al tráfico proveniente de una interfaz en particular será contestado por esta interfaz.

Esta es la configuración más básica. Funcionará para todos los procesos que estén funcionando en el propio router, y para la red local, si está enmascarada, y si no, entonces puede que tenga un espacio IP de ambos proveedores, o que vaya a enmascarar la salida por uno de los proveedores. En ambos casos, querrá añadir reglas escogiendo por cual proveedor encaminar basándose en las direcciones IP de las máquinas en la red local.

4.2.2. Equilibrio de carga

La segunda pregunta es cómo equilibar el tráfico que va por los dos proveedores. En realidad no es difícil si ya ha configurado un acceso dividido como se indicó previamente.

En lugar de escoger uno de los proveedores como la salida por defecto, configuraremos la ruta por defecto para que sea multicamino (multipath route). Por defecto, el núcleo equilibrará las rutas sobre los dos proveedores. Esto se hace como sigue (una vez más, nos apoyamos en el ejemplo de la sección sobre acceso dividido):

```
            ip route add default scope global nexthop via $P1 dev $IF1
weight 1 \
            nexthop via $P2 dev $IF2 weight 1
```

Esto equilibrará las rutas sobre ambos proveedores. Los parámetros **weight** se pueden modificar para favorecer a un proveedor sobre el otro.

Tenga en cuenta que el equilibrio no será perfecto, ya que se basa en rutas, y las rutas están en caché. Esto significa que las rutas usadas más a menudo siempre irán sobre el mismo proveedor.

Más aún, si realmente quiere hacer esto, probablemente también quiera los parches de Julian Anastasov que hay en http://www.linuxvirtualserver.org/~julian/#routes, la página del parche de rutas de Julian. Hará más sencillo el trabajo.

Capítulo 5. GRE y otros túneles

Hay tres tipos de túneles en Linux. Están los túneles IP sobre IP, los túneles GRE y túneles que se realizan fuera del núcleo (como por ejemplo, PPTP).

5.1. Breve inciso sobre los túneles:

Los túneles se pueden usar para hacer varias cosas poco usuales y bastante interesantes. También pueden hacer que las cosas vayan horriblemente mal si no los configura bien. No ponga su ruta por defecto sobre un dispositivo de túnel a menos que sepa *EXACTAMENTE* lo que está haciendo :-). Más aún, los túneles incrementan la carga, porque necesitan un juego extra de cabeceras IP. Normalmente, esto significa 20 bytes por paquete, de manera que si el tamaño normal de un paquete (MTU) en una red es de 1500 bytes, un paquete enviado por un túnel sólo puede ser de 1480 bytes como mucho. Esto no es necesariamente un problema, pero asegúrese de leer algo sobre fragmentación/reensamblaje de paquetes IP si planea conectar redes grandes con túneles. Oh, y por supuesto, la manera más rápida de cavar un túnel es cavar desde los dos extremos.

5.2. Túneles IP sobre IP

Este tipo de túneles lleva disponible en Linux mucho tiempo. Precisa dos módulos del núcleo, ipip.o y new_tunnel.o

Digamos que tenemos tres redes: las redes internas A y B, y una red intermedia C (o Internet, por ejemplo). De manera que tenemos la red A:

```
network 10.0.1.0
netmask 255.255.255.0
router  10.0.1.1
```

El router tiene la dirección 172.16.17.18 en la red C.

y la red B:

```
network 10.0.2.0
netmask 255.255.255.0
router  10.0.2.1
```

Cuyo router tiene la dirección 172.19.20.21 en la red C.

Hasta donde le concierne a C, asumimos que pasará cualquier paquete que vaya de A a B y viceversa. Incluso puede usar Internet para esto.

Esto es lo que haremos:

Primero, asegurarnos de que los módulos están instalados:

```
insmod ipip.o
insmod new_tunnel.o
```

Luego, en el router de la red A, hacemos lo siguiente:

```
ifconfig tunl0 10.0.1.1 pointopoint 172.19.20.21
route add -net 10.0.2.0 netmask 255.255.255.0 dev tunl0
```

Y en el de la red B:

```
ifconfig tunl0 10.0.2.1 pointopoint 172.16.17.18
route add -net 10.0.1.0 netmask 255.255.255.0 dev tunl0
```

Y si ha dejado de usar el túnel:

```
ifconfig tunl0 down
```

Listo, ya lo tiene. Sin embargo, no puede enviar tráfico de difusión (broadcast) o IPv6 mediante un túnel IP-sobre-IP. Simplemente puede conectar dos redes IPv4 que normalmente no podrían comunicarse entre ellas; eso es todo. En lo que respecta a la compatibilidad, este código lleva ahí mucho tiempo, de manera que es compatible hasta con los núcleos 1.3. Los túneles IP-sobre-IP de Linux no funcionan con otros sistemas operativos o routers, hasta donde yo sé. Es sencillo, y funciona. Uselo si lo necesita, en cualquier otro caso, use GRE.

5.3. Túneles GRE

GRE es un protocolo de tunelizado que desarrolló Cisco originalmente, y que puede hacer unas cuántas cosas más que los túneles IP-sobre-IP. Por ejemplo, puede transportar tráfico multicast e IPv6 sobre un túnel GRE.

En Linux, necesitará el módulo ip_gre.o.

5.3.1. Túneles IPv4

Primero hagamos un túnel IPv4:

Digamos que tenemos 3 redes: las redes internas A y B, y una red intermedia C (por ejemplo, Internet).

De manera que tenemos la red A:

```
network 10.0.1.0
netmask 255.255.255.0
router  10.0.1.1
```

El router tiene la dirección 172.16.17.18 en la red C. Llamaremos a esta red neta (ok, no es muy original)

Y la red B:

```
network 10.0.2.0
netmask 255.255.255.0
router  10.0.2.1
```

El router tiene la dirección 172.19.20.21 en la red C. Llamemos a esta red netb (seguimos con nuestra originalidad)

Hasta donde concierne a la red C, asumiremos que dejará pasar los paquetes enviados de A a B y vicebersa. Cómo y por qué, no nos interesa.

En el router de la red A, haremos lo siguiente:

```
ip tunnel add netb mode gre remote 172.19.20.21 local 172.16.17.18 ttl
255
ip link set netb up
ip addr add 10.0.1.1 dev netb
ip route add 10.0.2.0/24 dev netb
```

Miremos esto con más atención. En la línea 1, hemos añadido un dispositivo de túnel, y le hemos llamado netb (bastante obvio porque es a donde queremos llegar). Más aún, le hemos dicho que use el protocolo GRE (mode gre), que la dirección remota es 172.19.20.21 (el router en el otro extremo), que nuestros paquetes de túnel deberían ser originados por 172.16.17.18 (lo que permite a nustro router tener varias direcciones IP en la red C y decidir cual usar para el tunelizado) y que el campo TTL del paquete debería establecerse en 255 (ttl 255).

La segunda línea habilita el dispositivo.

En la tercera línea le hemos dado a la recién nacida interfaz netb la dirección 10.0.1.1. Esto está bien para redes pequeñas, pero cuando empiece una expedición de zapadores (MUCHOS túneles), quizá debiera considerar usar otro rango de IP para las interfaces de túneles (en este ejemplo, podría usar 10.0.3.0).

En la cuarta línea hemos establecido la ruta hacia la red B. Fíjese la notación diferente para la máscara de red. Si no está familiarizado con esta notación, así es como funciona: escriba la máscara de red en forma binaria, y cuente todos los unos. Si no sabe cómo hacerlo, limítese a recordar que 255.0.0.0 es /8, 255.255.0.0 es /16 y 255.255.255.0 es /24. Ah, y 255.255.254.0 es /23, en caso de que tuviera curiosidad.

Pero ya hemos tenido bastante de esto; veamos el router de la red B.

```
ip tunnel add neta mode gre remote 172.16.17.18 local 172.19.20.21 ttl
255
ip link set neta up
ip addr add 10.0.2.1 dev neta
ip route add 10.0.1.0/24 dev neta
```

Y cuando vaya a eliminar el túnel del router A:

```
ip link set netb down
ip tunnel del netb
```

Por supuesto, puede cambiar netb por neta para el router B.

5.3.2. Túneles IPv6

Vea la sección 6 si quiere una pequeña introducción a las direcciones IPv6.

Vamos con los túneles.

Asumamos que tiene la siguiente red IPv6, y que quiere conectarse a la 6bone, o con un amigo.

```
Red 3ffe:406:5:1:5:a:2:1/96
```

Su dirección IPv4 es 172.16.17.18, y el router 6bone tiene la dirección IPv4 172.22.23.24.

```
ip tunnel add sixbone mode sit remote 172.22.23.24 local 172.16.17.18
ttl 255
ip link set sixbone up
ip addr add 3ffe:406:5:1:5:a:2:1/96 dev sixbone
ip route add 3ffe::/15 dev sixbone
```

Vamos a comentar esto. En la primera línea, hemos creado un dispositivo de túnel llamado sixbone. Le hemos dado modo sit (que es un túnel IPv6 sobre IPv4) y le dijimos dónde debe ir (remote) y de dónde viene (local). TTL se pone al máximo, 255. Después activamos el dispositivo (up). Tras esto, añadimos nuestra propia dirección de red, y establecemos una ruta para 3ffe::/15 (que actualmente es la totalidad de la 6bone) a través del túnel.

Los túneles GRE son actualmente el tipo preferido de túneles. Es un estándar que está ampliamente adoptado fuera de la comunidad de Linux, y por tanto una Cosa Buena.

5.4. Túneles en espacio de usuario

Literalmente, hay docenas de implementaciones de túneles fuera del núcleo. Los más conocidos por supuesto son PPP y PPTP, pero hay mucho más (algunos propietarios, algunos seguros, otros que ni siquiera usan IP) y que realmente están más allá del ámbito de este Cómo.

Capítulo 6. Túneles IPv6 con Cisco o la 6bone

Por Marco Davids <marco@sara.nl>

NOTA al mantenedor:

Hasta donde sé, estos túneles IPv6-IPv4 no son por definición túneles GRE. Podría tunelizar IPv6 sobre IPv4 mediante dispositivos GRE (GRE tuneliza CUALQUIER cosa sobre IPv4), pero el dispositivo que se usa quí («sit») sólo tuneliza IPv6 sobre IPv4 y por lo tanto es algo diferente.

6.1. Túneles IPv6

Esta es otra aplicación de las capacidades para tunelizado de Linux. Es popular entre la gente que ha adoptado IPv6 tempranamente, o pioneros, si lo prefiere. El ejemplo práctico descrito más adelante no es, ciertamente, la única manera de hacer túneles IPv6. Sin embargo, es el método que se usa a menudo para hacer túneles entre Linux y un router Cisco con posibilidades de IPv6 y la experiencia nos dice que tras lo que mucha gente anda. Apuesto 10 a 1 a que esto se aplica a usted también ;-)

Notas breves sobre las direcciones IPv6:

Las direcciones IPv6 son, comparadas con las IPv4, realmente grandes: 128 bits frente a 32. Y nos proporciona justo lo que necesitamos: muchas, muchas direcciones IP: 340.282.266.920.938.463.463.374.607.431.768.211.465, para ser precisos. Aparte de esto, se supone que IPv6 (o IPng, de IP Next Generation) traerá una reducción el tamaño de las tablas de rutas de los router principales de Internet, una configuración más sencilla para el equipamiento, mejor seguridad en el nivel de IP y mejor soporte para QoS (calidad de servicio).

Un ejemplo: 2002:836b:9820:0000:0000:0000:836b:9886

Escribir direcciones IPv6 puede ser una gran molestia. Por tanto, nos facilitaremos la vida siguiendo algunas reglas:

- No use ceros sin significado (a la izquierda). Igual que hace en IPv4.
- Use dos puntos (:) para separara cada grupo de 16 bits o dos bytes.
- Cuando tenga muchos ceros consecutivos, puede abreviarlos con ::. Sólo puede hacer esto una vez por cada dirección, y sólo en grupos de 16 bits.

La dirección 2002:836b:9820:0000:0000:0000:836b:9886 puede escribirse 2002:836b:9820::836b:9886, que es algo más sencilla.

En otro ejemplo, la dirección 3ffe:0000:0000:0000:0000:0020:34A1:F32C puede simplificarse como 3ffe::20:34A1:F32C, que es mucho más corta.

Se pretende que IPv6 sea el sucesor de la actual IPv4. Debido a que es una tecnología relativamente nueva, no hay ninguna red mundial IPv6 nativa. Se está introduciendo la 6bone para posibilitar la aceleración del cambio.

Las redes IPv6 nativas se conectan unas a otras encapsulando el protocolo IPv6 en paquetes IPv4 y enviándolos mediante la infraestructura IPv4 ya existente desde un sitio IPv6 a otro.

Aquí es justo donde entran los túneles.

Para poder usar IPv6, deberíamos tener un núcleo que lo admita. Hay muchos documentos buenos sobre cómo conseguirlo. Pero todo se reduce a unos pocos pasos:

- Obtenga una distribución Linux reciente, con una glibc adecuada.

- Luego, consiga una fuente actualizada del núcleo.

Si lo tiene todo, puede seguir adelante y compilar un núcleo con capacidades de IPv6:

- Vaya a /usr/src/linux y escriba:
- make menuconfig
- Escoja «Networking Options»
- Marque «The IPv6 protocol», «IPv6: enable EUI-64 token format», «IPv6: disable provider based addresses»

CONSEJO: No use la opción de módulo. A menudo no funciona bien.

En otras palabras, compile IPv6 dentro del núcleo. Una vez hecho esto, guarde su configuración y compile el núcleo.

CONSEJO: Antes de hacerlo, considere la posibilidad de editar el Makefile: EXTRAVERSION = -x ; --> ; EXTRAVERSION = -x-IPv6

Hay un montón de documentación sobre cómo compilar e instalar un núcleo, sin embargo este documento se refiere a otras cosas. Si tiene problemas en esta etapa, vaya a buscar algo de documentación sobre la compilación del núcleo de acuerdo con sus propias especificaciones.

El fichero /usr/src/linux/README podría ser un buen sitio donde empezar. Tras haber hecho todo esto, y reiniciado con su nuevo núcleo, puede que quiera ejecutar "/sbin/ifconfig -a" y ver el nuevo "sit0-device". SIT significa Simple Internet Transition. Puede felicitarse a sí mismo; ahora está un paso más cerca de IP, la Próxima Generación ;-)

Ahora vamos con el siguiente paso. Quiere conectar su máquina, o incluso su LAN entera a otras redes que puedan trabajar con IPv6. Puede ser la «6bone», que está configurada especialmente para este propósito particular.

Asumamos que tiene la siguiente red IPv6: 3ffe:604:6:8::/64 y que quiere conectar con la 6bone, o con un amigo. Fíjese que la notación /64 de subred funciona igual que con cualquier dirección IP normal.

Su dirección IPv4 es 145.100.24.181 y el router de 6bone tiene la dirección IPv4 145.100.1.5

```
# ip tunnel add sixbone mode sit remote 145.100.1.5 [local
145.100.24.181 ttl 255]
# ip link set sixbone up
# ip addr add 3FFE:604:6:7::2/126 dev sixbone
# ip route add 3ffe::0/16 dev sixbone
```

Comentemos esto. En la primera línea, hemos creado un dispositivo de túnel llamado sixbone. Le hemos dado el modo sit (que es tunelizado IPv6 sobre IPv4) y le hemos dicho dónde debe ir (remote) y de dónde viene (local). El TTL se establece al máximo, 255.

Después activamos el dispositivo (up). Tras esto, añadimos nuestra propia dirección de red, y establecemos una ruta para 3ffe::/15 (que actualmente es la totalidad de la 6bone) a través del túnel. Si la máquina en la que trabaja es su pasarela IPv6, quizá le interesa añadir estas líneas:

```
# echo 1 >/proc/sys/net/ipv6/conf/all/forwarding
# /usr/local/sbin/radvd
```

En la última arrancamos radvd, que es (como zebra) un demonio anunciador de rutas, para dar soporte a las capacidades de autoconfiguración de IPv6. Búsquelo en su motor de búsqueda favorito si lo desea. Puede comprobar las cosas así:

```
# /sbin/ip -f inet6 addr
```

Si tiene radvd ejecutándose en la pasarela IPv6 y arranca una máquina Linux con capacidades IPv6 en su LAN, podrá disfrutar de los beneficios de la autoconfiguración de IPv6:

```
# /sbin/ip -f inet6 addr
1: lo: <LOOPBACK,UP> mtu 3924 qdisc noqueue inet6 ::1/128 scope host

3: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 100
inet6 3ffe:604:6:8:5054:4cff:fe01:e3d6/64 scope global dynamic
valid_lft forever preferred_lft 604646sec inet6
fe80::5054:4cff:fe01:e3d6/10
scope link
```

Podría ir más allá y configurar el programa bind para que trabaje con direcciones IPv6. El tipo A tiene un equivalente para IPv6: AAAA. El equivalente de in-addr.arpa es: ip6.int. Hay mucha información disponible sobre este tema.

Hay disponible un número creciente de aplicaciones que pueden trabajar con IPv6, incluyendo secure shell, telnet, inetd, el navegador Mozilla, el servidor web Apache y muchos otros. Pero se sale del ámbito de este documento sobre Rutado ;-)

Por el lado del Cisco la configuración debería parecerse a esto:

```
!
interface Tunnel1
description IPv6 tunnel
no ip address
no ip directed-broadcast
ipv6 enable
ipv6 address 3FFE:604:6:7::1/126
tunnel source Serial0
tunnel destination 145.100.24.181
tunnel mode ipv6ip
!
ipv6 route 3FFE:604:6:8::/64 Tunnel1
```

Pero si no tiene un Cisco a su disposición, pruebe uno de los muchos proveedores de túneles IPv6 disponibles en Internet. Están deseando configurar su Cisco con un túnel extra para usted. La mayoría lo hacen mediante una interfaz web amigable. Busque «ipv6 tunnel broker» en su buscador favorito.

Capítulo 7. IPsec: IP segura sobre Internet

FIXME: el puesto de editor esta vacante. Mientras tanto, vea: El proyecto FreeS/WAN. Otra implementación de IPSec para Linux es Cerberus, de NIST. Sin embargo, sus páginas web parecen no haberse actualizado en al menos dos años, y su versión se ha quedado atrás con respecto al núcleo de Linux. USAGI, que es una implementación alternativa de IPv6 para Linux, también incluye una implementación de IPSec, pero sólo sería para IPv6.

Capítulo 8. Enrutado multicast

FIXME: ¡Sin editor!

El Multicast-HOWTO es antiguo (relativamente hablando) y por esa razón podría ser poco preciso o incluso una mala guía en algunos puntos.

Antes de que pueda encaminar tráfico multicast, necesita configurar el núcleo Linux para que admita el tipo de enrutado multicast que desea. Para esto, a su vez, se necesita que decida qué tipo de encaminamiento multicast espera usar. Esencialmente hay cuatro tipos "comunes": DVMRP (la versión Multicast del protocolo RIP unicas), MOSPF (lo mismo pero para OSPF), PIM-SM ("Protocol Independent Multicasting - Sparse Mode", que asume que los usuarios de cualquier grupo multicast están esparcidos, en lugar de concentrados), y PIM-DM (lo mismo, pero en "Modo Denso", que asume que hay concentraciones significativas de usuarios en el mismo grupo multicast).

En el núcleo Linux, verá que no aparecen estas opciones. Esto es debido a que el protocolo en sí lo controla una aplicación de enrutado, como Zebra, mrouted o pimd. Sin embargo, aún así hace falta que tenga una buena idea de lo que va a usar, para escoger las opciones adecuadas en el núcleo.

Para todo el encaminamiento multicasat, definitivamente necesitará activar «multicasting» y «multicast routing». Esto es suficiente para DVMRP y MOSPF. Si va a usar PIM, también debe activar PIMv1 o PIMv2, dependiendo de si el tipo de red al que conecta usa la versión 1 o la 2 del protocolo PIM.

Una vez tenga preparado todo lo que necesita, y tenga el núcleo de Linux compilado, podrá ver que los protocolos IP que se listan durante el arranque ahora incluyen IGMP. Este protocolo sirve para gestionar grupos multicast. En el momento de escribir esto, Linux admite sólo las versiones 1 y 2 de IGMP, aunque existe la versión 3 y ha sido documentada. Esto en realidad no nos afecta mucho, ya que IGMPv3 todavía es demasiado nueva como para que sus capacidades extra vayan a ser muy útiles. Como IGMP trata con grpos, sólo vamos a usar las características presentes en la versión más sencilla de IGMP sobre grupos enteros. En la mayoría de los casos encontrará IGMPv2, aunque todavía se usa mucho IGMPv1.

Seguimos. Hemos activado multicasting. Ahora, tenemos que decirle al núcleo de Linux que haga algo con él, para poder empezar a encaminar. Esto significa añadir la red virtual Multicast a la tabla del router:

ip route add 224.0.0.0/4 dev eth0

(¡Asumiendo, por supuesto, que está haciendo multicast sobre eth0! Sustituya el dispositivo por el que haya escogido.)

Ahora, dígale a Linux que reenvíe paquetes...

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

En este momento, puede que se esté preguntando si con esto llegará a hacer algo en algún momento. Por tanto, para probar nuestra conexión, haremos ping al grupo por defecto, 224.0.0.1, para ver si alguien está vivo. Todas las máquinas de su LAN con el multicast activo *deberían* responder, pero nadie más. Comprobará que ninguna de las máquinas que responden tiene la dirección IP 224.0.0.1. ¡Qué sorpresa! Esta es una dirección de grupo («difusión» o «broadcast» para suscriptores), y todos los miembros del grupo responderán con su propia dirección, no con la dirección de grupo.

ping -c 2 224.0.0.1

En este momento, estamos preparados para hacer el verdadero encaminamiento de multicast. Bueno, eso asumiendo que tenga dos redes entre las que crear la ruta.

(¡Continuará!)

Capítulo 9. Disciplinas de colas (qdiscs) para gestión del ancho de banda

Cuando descubrí esto, *realmente* me impactó. Linux 2.2/2.4 viene con todo lo necesario para gestionar el ancho de banda en formas comparables a los sistemas dedicados de alto nivel para gestión de ancho de banda.

Linux va incluso más allá de lo que proporcionan Frame y ATM.

Para prevenir confusiones, sepa que **tc** usa las siguientes reglas para la especificación de ancho de banda:

```
mbps = 1024 kbps = 1024 * 1024 bps => byte/s
mbit = 1024 kbit => kilobit/s.
mb = 1024 kb = 1024 * 1024 b => byte
mbit = 1024 kbit => kilobit.
```

Internamente, los números se almacenan en bps y b.

Pero cuando **tc** imprime las tasas, usa lo siguiente:

```
1Mbit = 1024 Kbit = 1024 * 1024 bps => byte/s
```

9.1. Las colas y disciplinas de cola explicadas

Con el encolamiento determinamos la manera en que se *ENVIAN* los datos. Es importante darse cuenta de que sólo podemos dar forma a lo que transmitimos.

De la manera en que trabaja Internet, no tenemos control directo sobre lo que la gente nos envía. Es un poco como el buzón (¡físico!) de su casa. No hay manera en que pueda influenciar al mundo para modificar la cantidad de correo que le envían, ya que no puede ponerse en contacto con todo el mundo.

Sin embargo, la Internet se basa en su mayoría en TCP/IP, que tiene algunas características que nos ayudarán. TCP/IP no tiene manera de saber la capacidad de la red entre dos sistemas, de manera que simplemente empieza a enviar datos más y más rápido («comienzo lento») y cuando se empiezan a perder paquetes, porque no hay espacio para enviarlos, reduce la marcha. En realidad, es un poco más inteligente que todo esto, pero hablaremos de ello más adelante.

Esto es equivalente a no leer la mitad del correo, y esperar que la gente deje de enviárselo. Con la diferencia de que para Internet, funciona :-)

Si tiene un router y desea evitar que ciertas máquinas dentro de su red descarguen demasiado rápido, necesita dar forma (shape) a la interfaz *interna* del router, la que envía los datos a sus computadores.

También tiene que asegurarse de que controla el cuello de botella del enlace. Si tiene una NIC de 100Mbit y un router con un enlace de 256kbit, tiene que asegurarse de que no envía más datos de los que el router puede manejar. Por otro lado, será el router el que controle el enlace y ajuste el ancho de banda disponible. Necesitamos «poseer la cola» por decirlo así, y ser el enlace más lento de la cadena. Por suerte, esto es muy posible.

9.2. Disciplinas de cola simples, sin clases

Como dije, con las disciplinas de cola, cambiamos el modo en que se envían los datos. Las disciplinas de cola sin clases son aquellas que, mayormente, aceptan datos y se limitan a reordenarlos, retrasarlos, o descartarlos.

Esto se puede usar para ajustar el tráfico de una interfaz entera, sin subdivisiones. ¡Es vital que comprenda esta parte del encolado antes de que pasemos a los qdisc-contenedores-de-qdiscs con clases!

La disciplina más usada, con mucho, es la qdisc pfifo_fast (se usa por defecto). Esto también explica por qué estas características avanzadas son tan robustas. No son más que «simplemente otra cola».

Cada una de estas colas tiene puntos fuertes y debilidades específicos. Puede que no todas estén bien probadas.

9.2.1. pfifo_fast

Esta cola es, como su nombre indica, First In, First Out (el primero que entra es el primero que sale), lo que significa que ningún paquete recibe un tratamiento especial. Al menos, no mucho. Esta cola tiene 3 de lo que llamamos «bandas». Dentro de cada banda, se aplican las reglas FIFO. Sin embargo, no se procesará la banda 1 mientras haya paquetes esperando en la banda 0. Lo mismo se aplica para las bandas 1 y 2.

El núcleo obedece la marca llamada Type of Service que hay en los paquetes, y tiene cuidado de insertar los paquetes de «mínimo retraso» en la banda 0.

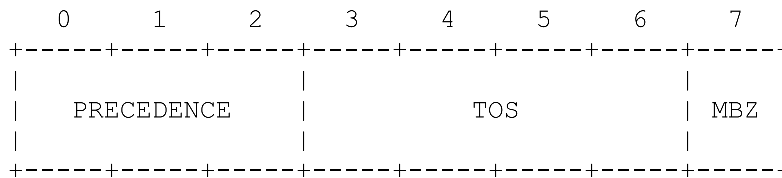
¡No confunda esta qdisc sencilla y sin clases con la PRIO con clases! Aunque se comportan de manera similar, la pfifo_fast no tiene clases y no puede añadir otras qdisc a ella con la orden tc.

9.2.1.1. Parámetros y forma de uso

No puede configurar la qdisc pfifo_fast ya que es el la cola por defecto fija. Así es como va configurada de serie:

priomap

> Determina cómo se corresponden las prioridades de los paquetes, tal como las asigna el núcleo, a las bandas. La correspondencia se basa en el octeto TOS del paquete, que es así:



```
   0     1     2     3     4     5     6     7
+-----+-----+-----+-----+-----+-----+-----+-----+
|                 |                       |     |
|   PRECEDENCE    |          TOS          | MBZ |
|                 |                       |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

Los cuatro bits TOS (el «campo TOS») se define como:

Binary	Decimcal	Meaning
1000	8	Minimizar retraso (md)
0100	4	Maximizar transferencia (mt)
0010	2	Maximizar fiabilidad (mr)
0001	1	Minimizar el coste monetario (mmc)
0000	0	Servicio normal

Como hay 1 bit a la derecha de estos cuatro, el valor real del campo TOS es el doble del valor de sus bits. Tcpdump -v -v muestra el valor del campo TOS completo, no sólo sus cuatro bits. Este es el valor que ve en la primera columna de esta tabla:

TOS	Bits	Significa	Prioridad Linux	Banda
0x0	0	Servicio normal	0 Mejor esfuerzo	1
0x2	1	Minimizar coste monet.	1 Relleno	2
0x4	2	Maximizar fiabilidad	0 Mejor esfuerzo	1
0x6	3	mmc+mr	0 Mejor esfuerzo	1
0x8	4	Mazimizar transferencia	2 En masa	2
0xa	5	mmc+mt	2 En masa	2
0xc	6	mr+mt	2 En masa	2
0xe	7	mmc+mr+mt	2 En masa	2
0x10	8	Minimizar retrasos	6 Interactivo	0
0x12	9	mmc+md	6 Interactivo	0
0x14	10	mr+md	6 Interactivo	0
0x16	11	mmc+mr+md	6 Interactivo	0
0x18	12	mt+md	4 Int. en masa	1
0x1a	13	mmc+mt+md	4 Int. en masa	1
0x1c	14	mr+mt+md	4 Int. en masa	1
0x1e	15	mmc+mr+mt+md	4 Int. en masa	1

Muchos números. La segunda columna contiene el valor de los cuatro bits TOS relevantes, seguidos por su significado traducido. Por ejemplo, el 15 significa que un paquete espera un Mínimo coste monetario, la Máxima fiabilidad, la Máxima transferencia Y un Retraso mínimo. Yo llamaría a esto un «paquete holandés» (N. del T: el autor es holandés, y supongo que se reirá mucho con este chiste).

La cuarta columna indica la manera en que el núcleo Linux interpreta los bits del TOS, mostrando qué prioridad les asigna.

La última columna indica el resultado del priomap por defecto. En la línea de órdenes el priomap por defecto se parece a:

```
1, 2, 2, 2, 1, 2, 0, 0 , 1, 1, 1, 1, 1, 1, 1, 1
```

Esto significa que a la prioridad 4, por ejemplo, se asigna la banda número 1. El priomap también le permite listar prioridades mayores (> 7) que no se corresponden a asignaciones del TOS, sino que se configuran por otros medios.

Esta tabla del RFC 1349 (léalo para obtener más detalles) le dice cómo deberían activar las aplicaciones los bits del TOS:

```
TELNET                    1000          (minimizar retraso)
FTP
  Control          1000           (minimizar retraso)
        Datos             0100          (maximizar
transferencia)

TFTP                      1000          (minimizar retraso)

SMTP
  Fase de órdenes  1000           (minimizar retraso)
        Fase de datos     0100          (maximizar
transferencia)

Domain Name Service
  Consulta UDP     1000           (minimizar retraso)
  Consulta TCP     0000
  Transf. de zona  0100           (maximizar transferencia)

NNTP                      0001          (minimizar coste
monetario)

ICMP
  Errores          0000
  Peticiones       0000 (la mayoría)
  Respuestas       <igual que las peticiones> (la mayoría)
```

txqueuelen

La longitud de esta cola se obtiene de la configuración de la interfaz, que puede ver y modificar con ifconfig o ip. Para establecer la longitud de la cola a 10, ejecute: ifconfig eth0 txqueuelen 10

¡No puede establecer este parámetro con tc!

9.2.2. Token Bucket Filter

El Token Bucket Filter (TBF) es un qdisc sencillo que se limita a dejar pasar paquetes que lleguen a una tasa que no exceda una impuesta administrativamente, pero con la posibilidad de permitir ráfagas cortas que excedan esta tasa.

TBF es muy preciso, amigable para la red y el procesador. ¡Debería ser su primera elección si sólo quiere ralentizar una interfaz!

La implementación de TBF consiste en un búfer (el bucket o balde), que se llena constatemente con piezas virtuales de información denominadas tokens, a una tasa específica (token rate). El parámetro más importante del bucket es su tamaño, que es el número de tokens que puede almacenar.

Cada token que llega toma un paquete de datos entrante de la cola de datos y se elimina del bucket. Asociar este algoritmo con los dos flujos (tokens y datos), nos da tres posibles situaciones:

- Los datos llegan a TBF a una tasa que es *igual* a la de tokens entrantes. En este caso, cada paquete entrante tiene su token correspondiente y pasa a la cola sin retrasos.
- Los datos llegan al TBF a una tasa *menor* a la de los token. Sólo una parte de los tokens se borran con la salida da cada paquete que se envía fuera de la cola, de manera que se acumulan los tokens, hasta llenar el bucket. Los tokens sin usar se pueden utilizar para enviar datos a velocidades mayores de la tasa de tokens, en cuyo caso se produce una corta ráfaga de datos.
- Los datos llegan al TBF a una tasa *mayor* a la de los token. Esto signigica que el bucket se quedará pronto sin tokens, lo que causará que TBF se acelere a sí mismo por un rato. Esto se llama una «situación sobrelímite». Si siguen llegando paquetes, empezarán a ser descartados.

Esta última situación es muy importante, porque permite ajustar administrativamente al ancho de banda disponible a los datos que están pasando por el filtro.

La acumulación de tokens permite ráfagas cortas de datos extralimitados para que pasen sin pérdidas, pero cualquier sobrecarga restante causará que los paquetes se vayan retrasando constantemente, y al final sean descartados.

Tenga en cuenta que en la implementación actual, los tokens se corresponden a bytes, no a paquetes.

9.2.2.1. Parámetros y uso

Incluso aunque probablemente no necesite hacerle cambios, tbf tiene algunos controles ajustables. Primero, los parámetros que estarán disponibles siempre:

limit o latency

> Limit es el número de bytes que pueden ser encolados a la espera de que haya tokens disponibles. También puede especificar esto estableciendo el parámetro latency, que indica el periodo máximo de tiempo que puede pasar un paquete en el TBF. Este último cálculo tiene en cuenta el tamaño del bucket, la tasa y posiblemente el peakrate (la tasa de picos, si es que la ha configurado).

burst/buffer/maxburst

> Tamaño del bucket, en bytes. Esta es la máxima cantidad de bytes para los que pueden haber tokens disponibles instantáneamente. En general, grandes tasas precisan grandes búferes. Para 10mbit/s sobre Intel, ¡necesitará al menos un búfer de 10kbyte si desea alcanzar la tasa que ha configurado!
>
> Si el búfer es demasiado pequeño, se descartarán paquetes debido a que llegan más tokens por tick del temporizador de los que caben en el bucket.

mpu

> Un paquete de tamaño cero no usa un ancho de banda cero. En ethernet, ningún paquete usa menos de 64 bytes. La Minimum Packet Unit determina el uso mínimo de tokens por paquete.

rate

> El ajuste de la velocidad. ¡Vea las indicaciones hechas anteriormente sobre la velocidad!

Si el paquete contiene tokens y se le permite estar vacío, por defecto tendrá velocidad infinita. Si esto no es aceptable, use los siguientes parámetros:

peakrate

> Si hay tokens disponibles, y llegan paquetes, por defecto se envían inmediatamente, a «la velocidad de la luz» por decirlo de alguna manera. Puede que esto no sea lo que usted quiere, especialmente si tiene un bucket grande.
>
> La tasa de picos se puede usar para especificar cuán rápido se le permite al bucket vaciarse. Si está haciendo todo según el libro, esto se consigue enviando un paquete, y esperando después lo suficiente antes de enviar el siguiente. Hemos calculado nuestras esperas de manera que se envíen justo a la tasa de picos (peakrate).
>
> Sin embargo, debido a la resolución por defecto de 10ms del temporizador de Unix, con paquetes de 10.000 bits de media, ¡estaremos limitados a una tasa de picos de 1mbit/s!

mtu/minburst

> La peakrate de 1mbit/s no es muy útil si la tasa normal es mayor que ésa. Es posible tener una tasa de picos mayor enviando más paquetes por fracción del temporizador, ¡lo que significa de forma efectiva que hemos creado un segundo bucket!
>
> Este segundo bucket contiene por defecto un único paquete, por lo que no es un bucket realmente.
>
> Para calcular la peakrate máxima posible, multiplique la mtu configurada por 100 (o más correctamente, HZ, que es 100 en Intel, y 1024 en Alpha).

9.2.2.2. Configuración de ejemplo

Esta es una configuración sencilla pero *muy* útil:

```
# tc qdisc add dev ppp0 root tbf rate 220kbit latency 50ms burst 1540
```

Ok, ¿por qué es esto útil? Si tiene un dispositivo de red con una cola grande, como un módem para DSL o un cable módem, y quiere comunicarse con él mediante un dispositivo rápido, como una interfaz ethernet, se encontrará conque enviar cualquier cosa destruye completamente la interactividad.

Esto se debe a que enviar datos llena la cola del módem, que probablamente es *enorme* porque realmente ayuda a conseguir una buena transferencia de datos al enviar. Pero esto no es lo que usted quiere; lo que quiere es tener una cola no tan grande de manera que la interactividad se mantenga de manera que aún pueda hacer otras cosas mientras envía los datos.

La línea anterior reduce los envíos a una tasa que no lleve formar colas en el módem (la cola estará en Linux, donde podemos ajustarla a un tamaño limitado).

Cambie 220kbit por su velocidad *real* de envío, menos un pequeño porcentaje. Si tiene un módem realmente rápido, suba «burst» un poco.

9.2.3. Stochastic Fairness Queueing

Stochastic Fairness Queueing (SFQ) es una implementación sencilla de la familia de algoritmos de colas justas (fair queueing). Es menos preciso que los otros, pero también necesita menos cálculos mientras que resulta ser casi perfectamente justo.

La palabra clave en SFQ es conversación (o flujo), que se corresponde en su mayoría a una sesión TCP o a un flujo UDP. El tráfico se divide en un número bastante grande de colas FIFO, una por cada conversación. Entonces se envía el tráfico de una manera parecida a round robin, dando a cada sesión por turnos la oportunidad de enviar datos.

Esto lleva a un comportamiento bastante equitativo y evita que una única conversación ahogue a las demás. SFQ se llama «estocástica» porque realmente no crea una cola para cada sesión, sino que tiene un algoritmo que divide el tráfico en un número limitado de colas usando un algoritmo de hash (troceo).

Debido al hash, varias sesiones pueden acabar en el mismo bucket, lo que dividirá por dos las posibilidades de cada sesión de enviar un paquete, reduciendo a la mitad de esta forma la velocidad efectiva disponible. Para evitar que esta situación acabe siendo detectable, SFQ cambia a menudo su algoritmo hash de manera que dos sesiones sólo colisionen durante unos pocos segundos.

¡Es importante tener en cuenta que SFQ sólo es útil en caso de que la interfaz real de salida esté realmente llena! Si no lo está, entonces la máquina Linux no encolará paquetes y no se producirá efecto alguno. Más tarde describiremos cómo combinar SFQ con otros qdisc para obtener lo mejor de ambos mundos.

Específicamente, configurar SFQ sobre una interfaz ethernet que esté apuntando al cable módem o a un router DSL, ¡no tiene sentido si no se hace algún ajuste más!

9.2.3.1. Parámetros y uso

SFQ es mayormente autoajustable:

perturb

Reconfigurar el hash una vez cada estos segundos. Si no se indica, el hash no se reconfigurará nunca. No es recomendable. 10 segundos es probablemente un buen valor.

quantum

Cantidad de bytes de un flujo que se permiten sacar de la cola antes de que le toque el turno a la siguiente cola. Por defecto es 1 paquete de tamaño máximo (tamaño MTU). ¡No lo ponga por debajo del MTU!

9.2.3.2. Configuración de ejemplo

Si tiene un dispositivo que tenga velocidad de enlace y tasa actual disponible idénticas, como una línea telefónica, esta configuración ayudará a mejorar la equitatividad:

```
# tc qdisc add dev ppp0 root sfq perturb 10
# tc -s -d qdisc ls
qdisc sfq 800c: dev ppp0 quantum 1514b limit 128p flows 128/1024
perturb 10sec
 Sent 4812 bytes 62 pkts (dropped 0, overlimits 0)
```

El número 800c: es el número de manejador asignado de forma automática, limit indica que pueden esperar 128 paquetes en esta cola. Hay 1024 hashbuckets disponibles para la contabilidad, de los cuales puede haber 128 activos al mismo tiempo (¡no caben más paquetes en la cola!) Los hash se reconfiguran una vez cada 10 segundos.

9.3. Consejos sobre en qué momento usar qué cola

Resumiendo, estas son las colas sencillas que realmente gestionan el tráfico reordenando, ralentizando o eliminando paquetes.

Los siguientes consejos pueden ayudarle a escoger qué cola usar. Menciona algunas qdisc descritas en el capítulo *Capítulo 14*.

- Para simplemente ralentizar el tráfico de salida, use el Token Bucket Filter. Funciona para grandes anchos de banda, si escala el bucket.
- Si su enlace está realmente lleno y quiere asegurarse de que ninguna sesión domina el ancho de banda de salida, use Stochastical Fairness Queueing.
- Si tiene un backbone grande y sabe lo que está haciendo, considere usar Random Early Drop (vea el capítulo Avanzado).
- Para «dar forma» al tráfico de entrada que no está reenviando, use el Ingress Policer. El ajuste del tráfico de entrada se denomina «policing», por cierto, no «shaping».
- Si *lo está* reenviando, use un TBF en la interfaz hacia la que está redirigiendo los datos. A menos que quiera ajustar el tráfico que vaya a salir por varias

interfaces, en cuyo caso el único factor común es la interfaz de entrada. En tal caso, use el Ingress Policer.

- Si no quiere ajustar, sino ver si su interfaz está tan cargada que tiene una cola, use la cola pfifo (no pfifo_fast). Carece de bandas internas pero lleva la cuenta del tamaño de su búfer.
- Por último, siempre puede usar el "ajuste social". No siempre es posible usar la tecnología para conseguir lo que se quiere. Los usuarios se toman las limitaciones técnicas como hostilidad. ¡Una palabra amable también puede ayudar a dividir correctamente su ancho de banda!

9.4. Terminología

Para entender correctamente configuraciones más complicadas se necesita explicar primero algunos conceptos. Debido a la complejidad y la relativa novedad de este tema, la gente usa un montón de términos diferentes cuando en realidad quieren decir lo mismo.

Lo que sigue se basa remotamente en `draft-ietf-diffserv-model-06.txt`, *Un modelo informal de gestión para routers Diffserv*. Actualmente lo puede encontrar en http://www.ietf.org/internet-drafts/draft-ietf-diffserv-model-06.txt.

Léalo para ver las definiciones estrictas de los términos usados.

Disciplina de colas

> Un algoritmo que controla la cola de un dispositivo, sea de entrada (ingress) o de salida (egress).

Qdisc sin clases

> Una qdisc sin subdivisiones internas configurables.

Qdisc con clases

> Una qdisc con clases contiene múltiples clases. Cada una de ellas contiene otra qdisc, que a su vez puede ser con clases, pero no tiene por qué. De acuerdo con la definición estricta, pfifo_fast *es* con clases, porque contiene tres bandas que son, en realidad, clases. Sin embargo, desde la perspectiva de configuración del usuario, no tiene clases ya que las clases no se pueden tocar con la herramienta tc.

Clases

> Una qdisc con clases puede tener muchas clases, cada una de las cuales es interna a ella, pudiendo a su vez contener cada clase qdisc reales.

Clasificador

Cada qdisc con clases necesita determinar a qué clase necesita enviar un paquete. Esto se hace usando el clasificador.

Filtro

La clasificación se puede realizar usando filtros. Un filtro contiene varias condiciones que pueden ser cumplidas.

Scheduling (ordenamiento)

Una qdisc puede, con la ayuda de un clasificador, decidir que algunos paquetes necesitan salir antes que otros. Este proceso se denomina Scheduling, y lo realiza por ejemplo la qdisc qfifo_fast anteriormente mencionada. El Scheduling también se denomina «reordenamiento», pero esto es confuso.

Shaping (ajuste)

El proceso de retrasar paquetes antes de que salgan para hacer que el tráfico sea conforme a una tasa máxima configurada. El Shapping se realiza durante la salida («egress»). Coloquialmente, al descarte de paquetes para ralentizar el tráfico también se le suele denominar Shapping.

Policing

Retrasar o descartar paquetes para que el tráfico se mantenga por debajo de un ancho de banda configurado. En Linux, el «policing» sólo puede descartar paquetes, no retrasarlo (no hay una «cola de ingreso» - ingress queue).

Conservativa de trabajo

Una qdisc conservativa de trabajo (work-conserving) siempre distribuye paquetes si los hay disponibles. En otras palabras, nunca retrasa un paquete si el adaptador de red está preparado para enviarlo (en el caso de una qdisc de salida - egress).

No conservativa de trabajo

Algunas colas, como la Token Bucket Filter, por ejemplo, pueden necesitar retrasar un paquete durante un cierto tiempo para limitar el ancho de banda. Esto significa que algunas veces rechazarán enviar un paquete, incluso aunque los haya disponibles.

Ahora que conocemos la terminología, veamos dónde están todas estas cosas.

```
     Programas en espacio de usuario
                   ^
                   |
   +---------------+-----------------------------------------+
   |               Y                                         |
   |    -------> Pila IP                                     |
   |   |              |                                      |
   |   |              Y                                      |
   |   |              Y                                      |
   |   ^              |                                      |
   |   |  / ----------> Reenvío ->                           |
   |   ^ /                     |                             |
   |   |/                      Y                             |
   |   |                       |                             |
   |   ^                       Y              /-qdisc1-\   |
   |   |                 Clasificador de   /--qdisc2--\  |
--->-> Qdisc de              Salida (Egress)   ---qdisc3---- | ->
   |   Entrada (Ingress)                       \__qdisc4__/  |
   |                                            \-qdiscN_/   |
   |                                                         |
   +----------------------------------------------------------+
```

Gracias a Jamal Hadi Salim por esta representación ASCII.

El gran bloque representa al núcleo. La flecha de la izquierda es el tráfico entrando en la máquina desde la red. Entonces se le pasa a la Qdisc de Entrada que puede aplicar filtros a los paquetes, y decidir descartarlos. A esto se le llama «Policing».

Esto ocurre en una etapa muy temprana, antes de que se haya visto mucho del núcleo. Por tanto, es un buen lugar para descartar tráfico sin consumir mucho tiempo de CPU.

Si se le permite continuar al paquete, puede estar destinado a una aplicación local, en cuyo caso entra en la pila IP para ser procesado, y se envía a un programa en espacio de usuario. El paquete también puede ser reenviado sin pasar por una aplicación, en cuyo caso se destina a la salida (egress). Los programas de espacio de usuario también pueden distribuir datos, que serán examinados y reenviados por el Clasificador de Salida.

Ahí es observado y encolado por cualquiera de varias qdisc. En el caso por defecto, sin configurar, sólo hay instalada una qdisc de salida, pfifo_fast, que siempre recibe el paquete. A esto se le llama «encolado» (enqueueing).

Ahora el paquete está en la qdisc, esperando a que el núcleo pida que sea retransmitido por la interfaz de salida. A esto se le llama «desencolado».

Esta figura también funciona en el caso de que haya un único adaptador de red (las flechas de entrada y salida al núcleo no se deben tomar demasiado literalmente). Cada adaptador de red tiene ranuras tanto para entrada como para salida.

9.5. Disciplinas de cola con clases

Las qdisc con clases son muy útiles si tiene diferentes tipos de tráfico a los que quiere dar un tratamiento separado. Una de las qdisc con clases se denomina «CBQ», «Class Based Queueing», y se la menciona tan frecuentemente que la gente suele identificar el encolado con clases sólo con CBQ, pero éste no es el caso.

CBQ es meramente la que lleva más tiempo (y también la más compleja). No siempre es la que necesita. Esto puede ser un trauma para los partidarios del «efecto sendmail», que nos enseña que cualquier tecnología compleja que no viene documentada debe ser lo mejor que hay disponible.

En breve diremos más sobre CBQ y sus alternativas.

9.5.1. El flujo dentro de las qdisc con clases y sus clases

Cuando entra tráfico dentro de una qdisc con clases, hay que enviarlo a alguna de las clases que contiene (se necesita «clasificarlo»). Para determinar qué hay que hacer con un paquete, se consulta a los «filtros». Es importante saber que los filtros se llaman desde dentro de una qdisc, ¡y no al revés!

Los filtros asociados a esa qdisc devuelven entonces una decisión, y la qdisc la usa para encolar el paquete en una de las clases. Cada subclase puede probar otros filtros para ver si se imparten más instrucciones. En caso contrario, la clase encola el paquete en la qdisc que contiene.

Aparte de contener otras qdisc, la mayoría de las qdisc con clases también realizan «shaping». Esto es útil tanto para reordenar paquetes (con SFQ, por ejemplo) como para controlar tasas. Necesitará esto en caso de tener una interfaz de gran velocidad (por ejemplo, ethernet) enviando a un dispositivo más lento (un cable módem).

Si sólo fuera a usar SFQ, no debería pasar nada, ya que los paquetes entrarían y saldrían de su router sin retrasos: la interfaz de salida es mucho más rápida que la velocidad del enlace en sí. No habrá cola que reordenar.

9.5.2. La familia qdisc: raíces, controladores, hermanos y padres

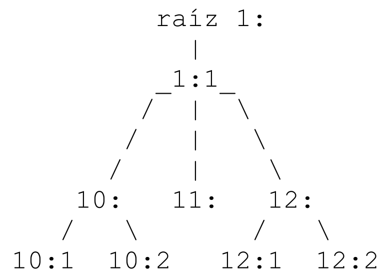
Cada interfaz tiene una «qdisc raíz» de salida, que por defecto es la disciplina de colas pfifo_fast sin clases que mencionamos anteriormente. A cada qdisc se le puede asignar un controlador (handle), que puede usar en posteriores sentencias de configuración para referirse a la qdisc. Aparte de la qdisc de salida, la interfaz también puede tener una de entrada, que dicta las normas sobre el tráfico que entra.

Los controladores de estas qdisc consisten en dos partes, un número mayor y un número menor. Es habitual darle a la qdisc de raíz el nombre «1:», que es lo mismo que «1:0». El número menor de una qdisc siempre es 0.

Las clases deben tener el mismo número mayor que sus padres.

9.5.2.1. Cómo se usan los filtros para clasificar el tráfico

Recapitulando, una jerarquía típica puede ser como ésta:



¡Pero no deje que este árbol le engañe! *No* debe imaginarse que el núcleo está en la cima del árbol y la red abajo, ya que no es el caso. Los paquetes se encolan y desencolan en el qdisc raíz, que es la única cosa con la que habla el núcleo.

Un paquete se clasifica en una cadena como ésta:

1: -> 1:1 -> 12: -> 12:2

Ahora el paquete reside en una cola de una qdisc asociada a la clase 12:2. En este ejemplo, se asocia un filtro a cada «nodo» del árbol, y cada cual escoge qué rama se toma en su paso. Esto puede tener sentido, Sin embargo, también es posible:

1: -> 12:2

En este caso, un filtro asociado a la raíz decidió enviar el paquete directamente a 12:2.

9.5.2.2. Cómo se desencolan los paquetes para enviarlos al hardware

Cuando el núcleo decide que necesita extraer paquetes para enviarlos a la interfaz, la qdisc 1: raíz recibe una petición de desencolar, que se pasa a 1:1, que a su vez la pasa a 10:, 11:, y 12:, cada una de las cuales consulta a sus descendientes, e intenta hacer dequeue() sobre ellos. En este caso, el núcleo necesita recorrer todo el árbol, porque sólo 12:2 contiene un paquete.

En breve, las clases anidadas SOLO hablan a sus qdisc paternas, y nunca a una interfaz. ¡Sólo la qdisc raíz recibe peticiones de desencolado del núcleo!

La consecuencia de esto es que las clases nunca desencolan más rápido de lo que sus padres permiten. Y esto es exactamente lo que queremos: de esta manera, podemos tener SFQ como clase interna, que no hace ajustes, sólo reordena, y tenemos una qdisc externa, que es la que hace los ajustes.

9.5.3. La qdisc PRIO

La qdisc PRIO en realidad no hace ajustes, sino que sólo subdivide el tráfico basándose en cómo haya configurado los filtros. Puede considerar la qdisc PRIO como una pfifo_fast con esteroides, en la que cada banda es una clase separada, en lugar de una simple FIFO.

Cuando se encola un paquete a la qdisc PRIO, se escoge una clase basándose en las órdenes de filtrado que haya dado. Por defecto, se crean tres clases. Estas clases contienen qdisc que son puras FIFO sin estructura interna, pero puede sustituirlas por cualquier qdisc que haya disponible.

Siempre que se necesite desencolar un paquete, se intenta primero con la clase :1. Las clases más altas sólo se usan si no se ha conseguido el paquete en las clases más bajas.

Esta qdisc es muy útil en caso de que quiera dar prioridad a cierto tráfico sin usar sólo las marcas TOS sino usando el potencial de los filtros de tc. También puede contener cualquier qdisc, mientras que pfifo_fast está limitada a qdisc de fifo sencillas.

Como en realidad no hace ajustes, se le aplica el mismo aviso que a SFQ: úsela solamente si el enlace físico está realmente lleno o métala dentro de una qdisc con clases que haga ajustes. Esto último se aplica a la mayoría de dispositivos DSL y cable módems.

Hablando formalmente, la qdisc PRIO es un reorganizador conservativo.

9.5.3.1. Parámetros y uso de PRIO

tc reconoce los siguientes parámetros:

bands

> Número de bandas a crear. Cada banda es una clase. Si cambia este número, también deberá cambiar:

priomap

> Si no proporciona filtros de tc para clasificar el tráfico, la qdisc PRIO examina la prioridad TC_PRIO para decidir cómo encolar el tráfico.
>
> Esto funciona igual que con la qdisc pfifo_fast mencionada previamente, refiérase a ella si desea más detalles.

Las bandas son clases, y todas se llaman de mayor:1 a mayor:3 por defecto, de manera que si nuestra qdisc PRIO se llama 12:, tc filtrará el tráfico a 12:1 para garantizar la mayor prioridad.

Repetimos: ¡la banda 0 va al número menor 1! La banda 1 al número menor 2, etc.

9.5.3.2. Configuración de ejemplo

Crearemos este árbol:

```
     raíz 1: prio
       /   |   \
     1:1  1:2  1:3
      |    |    |
     10:  20:  30:
     sfq  tbf  sfq
banda 0    1    2
```

El tráfico masivo irá a 30:, el interactivo a 20: o 10:.

Líneas de órdenes:

```
# tc qdisc add dev eth0 root handle 1: prio
## Esto crea *instantáneamente las clases 1:1, 1:2, 1:3

# tc qdisc add dev eth0 parent 1:1 handle 10: sfq
# tc qdisc add dev eth0 parent 1:2 handle 20: tbf rate 20kbit buffer
1600 limit 3000
# tc qdisc add dev eth0 parent 1:3 handle 30: sfq
```

Ahora veamos qué hemos creado:

```
# tc -s qdisc ls dev eth0
qdisc sfq 30: quantum 1514b
 Sent 0 bytes 0 pkts (dropped 0, overlimits 0)

 qdisc tbf 20: rate 20Kbit burst 1599b lat 667.6ms
 Sent 0 bytes 0 pkts (dropped 0, overlimits 0)

 qdisc sfq 10: quantum 1514b
 Sent 132 bytes 2 pkts (dropped 0, overlimits 0)

 qdisc prio 1: bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
 Sent 174 bytes 3 pkts (dropped 0, overlimits 0)
```

Como puede ver, la banda 0 ya ha tenido algo de tráfico, ¡y se envió un paquete mientras ejecutábamos la orden!

Ahora vamos a hacer alguna transferencia masiva con una herramienta que ajuste pertinentemente las marcas TOS, y echemos otro vistazo:

```
# scp tc ahu@10.0.0.11:./
ahu@10.0.0.11's password:
tc                  100% |*****************************|   353 KB
00:00
# tc -s qdisc ls dev eth0
qdisc sfq 30: quantum 1514b
 Sent 384228 bytes 274 pkts (dropped 0, overlimits 0)

 qdisc tbf 20: rate 20Kbit burst 1599b lat 667.6ms
 Sent 2640 bytes 20 pkts (dropped 0, overlimits 0)

 qdisc sfq 10: quantum 1514b
 Sent 2230 bytes 31 pkts (dropped 0, overlimits 0)
```

```
 qdisc prio 1: bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
 Sent 389140 bytes 326 pkts (dropped 0, overlimits 0)
```

Como puede ver, todo el tráfico fue al controlador 30:, que es la banda de menor prioridad, tal como esperábamos. Ahora, para verificar que el tráfico interactivo va a bandas más altas, crearemos un poco:

```
# tc -s qdisc ls dev eth0
qdisc sfq 30: quantum 1514b
 Sent 384228 bytes 274 pkts (dropped 0, overlimits 0)

 qdisc tbf 20: rate 20Kbit burst 1599b lat 667.6ms
 Sent 2640 bytes 20 pkts (dropped 0, overlimits 0)

 qdisc sfq 10: quantum 1514b
 Sent 14926 bytes 193 pkts (dropped 0, overlimits 0)

 qdisc prio 1: bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
 Sent 401836 bytes 488 pkts (dropped 0, overlimits 0)
```

Funcionó (todo el tráfico adicional se ha ido a 10:, que es nuestra qdisc de más alta prioridad). No se ha enviado tráfico a la de más baja prioridad, que recibió anteriormente todo nuestro scp.

9.5.4. La famosa qdisc CBQ

Como se dijo antes, CBQ es la qdisc más compleja disponible, la más publicitada, la menos comprendida, y probablemente la más difícil de configurar correctamente. Esto no se debe a que los autores sean malvados o incompetentes, ni mucho menos, sino sólo que el algoritmo CBQ no es tan preciso y realmente no se ajusta del todo a la manera de trabajar de Linux.

Además de ser de clases, CBQ también es ajustadora (shaper) y es en este aspecto en el que no funciona del todo bien. Debería funcionar de esta manera. Si intenta ajustar a 1mbit/s una conexión de 10mbit/s, el enlace debería estar ocioso el 90% del tiempo. Si no lo está, necesitamos acelerar de manera que realmente ESTE ocioso el 90% del tiempo.

Esto es bastante difícil de medir, de manera que en su lugar CBQ deriva el tiempo ocioso del número de microsegundos que se tarda entre cada petición de más datos por parte de la capa de hardware. Combinados, se pueden usar para aproximar cómo de lleno o vacío está el enlace.

Esto es bastante circunspecto y no siempre lleva a resultados adecuados. Por ejemplo, ¿qué pasaría si la verdadera velocidad del enlace no es capaz de transmitir realmente todos los 100mbit/s de datos, quizá debido a un driver mal implementado? Una tarjeta de red PCMCIA tampoco alcanzará nunca los 100mbit/s debido a la manera en que está diseñado el bus (de nuevo, ¿cómo calculamos el tiempo?)

Se vuelve aún peor si consideramos dispositivos de red no del todo reales, como PPP sobre Ethernet o PPTP sobre TCP/IP. El ancho de banda efectivo en ese caso probablemente se determina por la eficiencia de los canales al espacio de usuario (que es enorme).

La gente que ha hecho mediciones ha descubierto que CBQ no es siempre muy preciso, y a veces se pierde del todo.

Sin embargo, en muchas circunstancias funciona bien. Con la documentación que proporcionamos aquí, debería ser capaz de configurarlo para que funcione bien en la mayoría de los casos.

9.5.4.1. El «shaping» de CBQ en detalle

Como dije antes, CBQ trabaja asegurándose de que el enlace está ocioso sólo lo necesario para reducir el ancho de banda real hasta la tasa configurada. Para hacerlo, calcula el tiempo que debería pasar entre los paquetes medios.

Mientras opera, se mide el tiempo ocioso efectivo usando una media de movimiento por exponencial proporcional (EWMA - exponential weighted moving average), que considera los paquetes recientes exponencialmente más importantes que los pasados. La media de carga de UNIX (loadaverage) se calcula de la misma manera.

El tiempo ocioso calculado se resta al medido mediante EWMA, y el número resultante se llama «avgidle». Un enlace cargado perfectamente tiene un avgidle de cero: los paquetes llegan exactamente una vez cada intervalo calculado.

Un enlace sobrecargado tiene un avgidle negativo y si se vuelve muy negativo, CBQ lo cierra durante un rato y entonces se produce un «sobrelímite».

Por el contrario, un enlace ocioso puede amasar un avgidle enorme, lo que permitiría anchos de banda infinitos tras unas horas de silencio. Para evitarlo, avgidle se trunca en maxidle.

Si hay un sobrelímite, en teoría, la CBQ debería acelerarse a sí misma durante exactamente el tiempo que se ha calculado que pasa entre paquetes, entonces pasa un paquete, y se acelera de nuevo. Pero lea más adelante sobre el parámetro «minburst».

Estos son parámetros que puede especificar para configurar el ajuste:

avpkt

> Tamaño medio de un paquete, medido en bytes. Se necesita para calcular maxidle, que se deriva de maxburst, que va especificado en paquetes.

bandwidth

> El ancho de banda físico de su dispositivo, necesario para cálculos de tiempo ocioso.

cell

> El tiempo que tarda un paquete en ser transmitido sobre un dispositivo está escalonado, y se basa en el tamaño del paquete. Un paquete de tamaño 800 y

uno de 806 pueden tardar lo mismo en ser enviados, por ejemplo (esto establece la granularidad). A menudo está a "8". Debe ser una potencia entera de dos.

maxburst

Este número de paquetes se usa para calcular maxidel de manera que cuando avgidle esté a maxidel, se puede enviar una ráfaga de esta cantidad de paquetes medios antes de que avgidle caiga a 0. Póngalo alto si quiere que sea tolerante a ráfagas. No puede establecer maxidel directamente, sino sólo mediante este parámetro.

minburst

Como se mencionó previamente, CBQ necesita acelerar en caso de sobrelímite. La solución ideal es hacerlo exactamente durante el tiempo ocioso calculado, y pasar un paquete. Sin embargo, los núcleos de Unix normalmente lo pasan mal organizando eventos menores a 10ms, de manera que es mejor acelerar durante un periodo algo mayor, y hacer pasar entonces "minburst" paquetes de una sóla tanda, para después dormir "minburst" veces más.

El tiempo de espera se denomina offtime. Los valores altos de minburst llevan a ajustes más precisos a largo plazo, pero a ráfagas más grandes a escala de milisegundos.

minidle

Si avgidle está por debajo de 0, estaremos en sobrelímite y necesitaremos esperar hasta que avgidle sea suficientemente grande como para mandar un paquete. Para prevenir una ráfaga súbita después de haber detenido el enlace durante un periodo prolongado, avgidle se reinicia a minidle si cae demasiado bajo.

Minidle se especifica en microsegundos negativos, de manera que 10 significa que avgidle se corta a -10us.

mpu

Tamaño mínimo del paquete (necesario porque incluso los paquetes de tamaño cero se rellenan con 64 bytes en ethernet, y por tanto lleva cierto tiempo transmitirlos). CBQ necesita saber esto para calcular de forma adecuada el tiempo ocioso.

rate

La tasa deseada de tráfico saliente de esta qdisc (¡éste es el control de velocidad!)

Internamente, CBQ tiene un montón de ajustes más precisos. Por ejemplo, a las clases que se sabe no contienen datos encolados no se les pregunta. Se penaliza a las clases sobrelimitadas reduciendo su prioridad efectiva. Todo muy inteligente y complicado.

9.5.4.2. Comportamiento de la CBQ con clases

Aparte del ajuste, usando las aproximaciones de tiempo ocioso ya mencionadas, CBQ también actúa igual que la cola PRIO en el sentido de que sus clases pueden tener diferentes prioridades y que los números pequeños de prioridad se examinan antes que los grandes.

Cada vez que la capa de hardware pide un paquete para enviarlo a la red, se inicia un proceso de round robin por pesos («WRR»), comenzando por las clases de menor prioridad.

Estas se agrupan y se les pregunta si tienen datos disponibles. En tal caso, se devuelven. Después de que se haya permitido desencolar una serie de bytes a una clase, se prueba con la siguiente clase de esa misma prioridad.

Los siguientes parámetros controlan el proceso WRR:

allot

> Cuando se le pide a la CBQ externa un paquete para enviar por la interfaz, buscará por turnos en todas sus qdisc internas (en las clases), en el orden del parámetro de «prioridad». Cada vez que le toca el turno a una clase, sólo puede enviar una cantidad limitada de datos. «Allot» es la unidad básica de esta cantidad. Vea el parámetro «weight» para más información.

prio

> La CBQ también puede actuar como un dispositivo PRIO. Primero se prueba con las clases internas de menor prioridad y mientras tengan tráfico, no se mira en las otras clases.

weight

> Weight ayuda en el proceso de Weighted Round Robin. Cada clase tiene una oportunidad por turnos para enviar. Si tiene una clase con un ancho de banda significativamente menor que las otras, tiene sentido permitirle enviar más datos en su ronda que a las otras.
>
> Una CBQ suma todos los pesos bajo una clase, y los normaliza, de manera que puede usar números arbitrarios: sólo son importantes las equivalencias. La gente viene usando «tasa/10» como regla general y parece que funciona bien. El peso renormalizado se multiplica por el parámetro «allot» para determinar cuántos datos se envían en una ronda.

¡Tenga en cuenta que todas las clases dentro de una jerarquía CBQ necesitan compartir el mismo número mayor!

9.5.4.3. Parámetros CBQ que determinan la compartición y préstamo de enlaces

Aparte de meramente limitar determinados tipos de tráfico, también es posible especificar qué clases pueden tomar prestada capacidad de otras clases o, al revés, prestar ancho de banda.

Isolated/sharing

> Una clase que está configurada como «isolated» (aislada) no prestará ancho de banda a sus hermanas. Uselo si tiene varios agentes competidores o mutuamente hostiles sobre el enlace que no quieren dejarse espacio entre sí.
>
> El programa de control tc también conoce el «sharing», que es lo contrario que «isolated».

bounded/borrow

> Una clase también puede estar «bounded» (limitada), lo que significa que no tratará de tomar ancho de banda prestado de sus clases hermanas. tc también conoce «borrow», que es lo contrario de «bounded».

En una situación típica podríamos encontrarnos a dos agentes sobre un mismo enlace que al mismo tiempo son «isolated» y «bounded», lo que significa que están realmente limitadas a sus tasas aisladas, y que no permitirán préstamos entre sí.

Junto a tales clases, puede haber otras que tengan permiso de ceder ancho de banda.

9.5.4.4. Configuración de ejemplo

Esta configuración limita el tráfico del servidor web a 5mbit y el SMTP a 3mbit. Juntos, no pueden alcanzar más de 6mbit. Tenemos una NIC de 100mbit y las clases pueden tomar ancho de banda prestado unas de otras.

```
# tc qdisc add dev eth0 root handle 1:0 cbq bandwidth 100Mbit
\
  avpkt 1000 cell 8
# tc class add dev eth0 parent 1:0 classid 1:1 cbq bandwidth 100Mbit
\
  rate 6Mbit weight 0.6Mbit prio 8 allot 1514 cell 8 maxburst 20
\
  avpkt 1000 bounded
```

Esta parte instala la raíz en la clase convenida 1:0. La clase 1:1 está limitada, de manera que su ancho de banda no puede pasar de 6mbit.

Como se dijo antes, CBQ necesita un *montón* de ajustes. Todos los parámetros quedaron explicados anteriormente, de todas maneras. La configuración HTB correspondiente es mucho más sencilla.

```
# tc class add dev eth0 parent 1:1 classid 1:3 cbq bandwidth 100Mbit
\
```

```
  rate 5Mbit weight 0.5Mbit prio 5 allot 1514 cell 8 maxburst 20
\
  avpkt 1000
# tc class add dev eth0 parent 1:1 classid 1:4 cbq bandwidth 100Mbit
\
  rate 3Mbit weight 0.3Mbit prio 5 allot 1514 cell 8 maxburst 20
\
  avpkt 1000
```

Estas son nuestras dos clases. Fíjese en cómo escalamos el peso con la tasa configurada. Ninguna clase está limitada, pero están conectadas a la clase 1:1, que sí lo está. De manera que la suma de anchos de banda de las dos clases pasará nunca de 6mbit. ¡Por cierto!, los identificadores de clase deben estar dentro del mismo número mayor que su CBQ madre.

```
# tc qdisc add dev eth0 parent 1:3 handle 30: sfq
# tc qdisc add dev eth0 parent 1:4 handle 40: sfq
```

Ambas clases llevan una qdisc FIFO por defecto. Pero las hemos reemplazado con una cola SFQ de manera que cada flujo de datos sea tratado de igual forma.

```
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip \
  sport 80 0xffff flowid 1:3
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip \
  sport 25 0xffff flowid 1:4
```

Estas órdenes, asociadas directamente a al raíz, envían el tráfico a las qdisc correctas.

Fíjese que usamos «tc class add» para CREAR clases dentro de una qdisc, pero que usamos «tc qdisc add» para añadir qdiscs a estas clases.

Puede preguntarse qué sucede al tráfico que no se clasifica en ninguna de esasa dos reglas. Parecería que en este caso, los datos serán procesados dentro de 1:0, y estarán sin límites.

Si el tráfico SMTP+web conjunto intenta exceder el límite impuesto de 6mbit/s, el ancho de banda se dividirá de acuerdo al parámetro de peso, dando 5/8 del tráfico al servidor web y 3/8 al servidor de correo.

Con esta configuración también podemos decir que el tráfico del servidor web tendrá como mínimo 5/8 * 6 mbit = 3.75 mbit.

9.5.4.5. Otros parámetros de CBQ: split y defmap

Como mencioné anteriormente, una qdisc con clases necesita llamar a los filtros para determinar en qué clase encolar un paquete.

Aparte de llamar al filtro, CBQ ofrece otras opciones: defmap y split. Es bastante complicado de entender, y no es vital. Pero como éste es el único sitio conocido donde se explican adecuadamente defmap y split, intentaré hacerlo lo mejor posible.

Como es probable que a menudo sólo quiera filtrar el Tipo de Servicio, se proporciona una sintaxis especial. Cuando CBQ necesita averiguar dónde encolar un paquete, comprueba si éste es un «split node». Si lo es, una de las sub-qdisc ha indicado que desea recibir todos los paquetes con una determinada prioridad, que se podría derivar del campo TOS, o por opciones de socket establecidas por las aplicaciones.

Se hace un OR con los bit de prioridad de los paquetes y el campo defmap para ver si coinciden. En otras palabras, es un atajo para crear filtros muy rápidos, que sólo se ajustan a ciertas prioridades. Un defmap de ff (hex) coincidirá con todo, un mapa de 0, con nada. Quizá una configuración de ejemplo ayude a aclarar las cosas:

```
# tc qdisc add dev eth1 root handle 1: cbq bandwidth 10Mbit allot 1514
\
  cell 8 avpkt 1000 mpu 64

# tc class add dev eth1 parent 1:0 classid 1:1 cbq bandwidth 10Mbit
\
  rate 10Mbit allot 1514 cell 8 weight 1Mbit prio 8 maxburst 20
\
  avpkt 1000
```

Preámbulo estándar de CBQ. ¡Nunca me acostumbraré a la inmensa cantidad de cosas que necesita!

Defmap se refiere a los bits TC_PRIO, que se definen así:

```
TC_PRIO..          Num  Corresponde al TOS
-------------------------------------------------
BESTEFFORT         0    Maximizar fiabilidad
FILLER             1    Minimizar coste
BULK               2    Maximizar transferencia (0x8)
INTERACTIVE_BULK   4
INTERACTIVE        6    Minimizar retrasos (0x10)
CONTROL            7
```

El número TC_PRIO.. se corresponde a bits, contando desde la derecha. Vea la sección pfifo_fast si desea más detalles sobre cómo se convierten los bits de TOS en prioridades.

Ahora, las clases interactiva y masiva:

```
# tc class add dev eth1 parent 1:1 classid 1:2 cbq bandwidth 10Mbit
\
  rate 1Mbit allot 1514 cell 8 weight 100Kbit prio 3 maxburst 20
\
  avpkt 1000 split 1:0 defmap c0

# tc class add dev eth1 parent 1:1 classid 1:3 cbq bandwidth 10Mbit
\
  rate 8Mbit allot 1514 cell 8 weight 800Kbit prio 7 maxburst 20
\
  avpkt 1000 split 1:0 defmap 3f
```

La «split qdisc» es la 1:0, que es donde se va a hacer la elección. C0 es el valor binario 11000000, 3F es 00111111, de manera que entre los dos coinciden con todo. La primera

clase coincide con los bits 7 y 6, y por tanto se corresponde con el tráfico «interactivo» y «de control». La segunda clase coincide con el resto.

El nodo 1:0 tiene ahora una tabla como ésta:

```
prioridad       enviar a
0               1:3
1               1:3
2               1:3
3               1:3
4               1:3
5               1:3
6               1:2
7               1:2
```

Si desea más diversión, también puede pasar una «change mask» (máscara de cambio), que indica qué prioridades desea cambiar. Sólo necesita usar esto si está ejecutando «tc class change». Por ejemplo, para añadir los mejores esfuerzos al tráfico de 1:2, podría ejecutar esto:

```
# tc class change dev eth1 classid 1:2 cbq defmap 01/01
```

El mapa de prioridades de 1:0 ahora es así:

```
prioridad       enviar a
0               1:2
1               1:3
2               1:3
3               1:3
4               1:3
5               1:3
6               1:2
7               1:2
```

FIMXE: no he probado «tc class change», sólo he mirado los fuentes.

9.5.5. Hierarchical Token Bucket

Martin Devera (<devik>) se dio cuenta de que CBQ es complejo y que no parece óptimo en varias situaciones típicas. Su enfoque Jerárquico se ajusta bien a configuraciones donde se tiene una cantidad fija de ancho de banda a dividir para diferentes propósitos, dándole a cada propósito un ancho de banda garantizado, con la posibilidad de especificar cuánto ancho se puede tomar prestado.

HTB funciona igual que CBQ, pero no recurre a cálculos de tiempo ocioso para los ajustes. En su lugar, es un Token Bucket Filter con clases (de ahí el nombre). Sólo tiene unos pocos parámetros que están bien documentados en su sitio.

Según se complique su configuración HTB, verá que escala bien. ¡Con CBQ ya es complejo incluso en casos simples! HTB3 (vea los detalles sobre las versiones de HTB en su página web) es ahora parte de las fuentes oficiales del núcleo (desde 2.4.20-pre1 y 2.5.31 en adelante). Sin embargo, quizá necesite una versión de «tc» parcheada para

HTB3: la versión de las partes del núcleo y espacio de usuario de HTB3 deben tener el mismo número mayor, o «tc» no trabajará con HTB.

Si ya tiene un núcleo moderno, o está en posición de parchear el núcleo, considere seriamente usar HTB.

9.5.5.1. Configuración de ejemplo

Funcionalmente es casi idéntica a la configuración de ejemplo anterior de CBQ:

```
# tc qdisc add dev eth0 root handle 1: htb default 30

# tc class add dev eth0 parent 1: classid 1:1 htb rate 6mbit burst 15k

# tc class add dev eth0 parent 1:1 classid 1:10 htb rate 5mbit burst
15k
# tc class add dev eth0 parent 1:1 classid 1:20 htb rate 3mbit ceil
6mbit burst 15k
# tc class add dev eth0 parent 1:1 classid 1:30 htb rate 1kbit ceil
6mbit burst 15k
```

El autor recomienda SFQ por debajo de estas clases:

```
# tc qdisc add dev eth0 parent 1:10 handle 10: sfq perturb 10
# tc qdisc add dev eth0 parent 1:20 handle 20: sfq perturb 10
# tc qdisc add dev eth0 parent 1:30 handle 30: sfq perturb 10
```

Añada los filtros que dirigirán el tráfico a las clases correctas:

```
# U32="tc filter add dev eth0 protocol ip parent 1:0 prio 1 u32"
# $U32 match ip dport 80 0xffff flowid 1:10
# $U32 match ip sport 25 0xffff flowid 1:20
```

Y esto es todo (nada de números desagradables sin explicación, ni parámetros sin documentar).

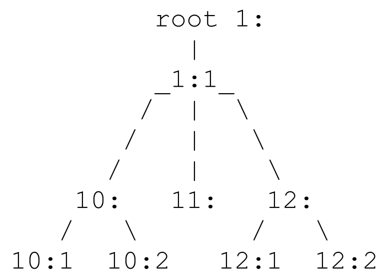
Realmente, HTB parece maravilloso (si 10: y 20: tienen ambos su ancho de banda garantizado, y se deja más a dividir, tomarán prestado en una relación de 5:3, tal como cabría esperar).

El tráfico sin clasificar va a parar a 30:, que tiene poco ancho de banda por sí mismo, pero que puede tomar prestado todo lo que queda. Como hemos escogido SFQ internamente, ¡de paso obtendremos equitatividad!

9.6. Clasificar paquetes con filtros

Para determinar qué clase deberá procesar un paquete, se llama a una «cadena clasificadora» cada vez que hay que tomar una decisión. Esta cadena consiste en todos los filtros asociados a la qdisc con clases que debe decidir.

Repitamos el árbol, que no es un árbol:



Cuando encolamos un paquete, se consulta la cadena de filtros para cada rama para que de las instrucciones relevantes. Una configuración típica podría tener un filtro en 1:1 que dirija un paquete a 12:, y uno en 12: que envíe el paquete a 12:2.

También podría asociar esta última regla a 1:1, pero puede ganar eficiencia haciendo pruebas más específicas más abajo en la cadena.

No se puede filtrar un paquete "hacia arriba", por cierto. Además, con HTB, ¡debería asociar todos los filtros a la raíz!

Y de nuevo: ¡los paquetes sólo se encolan hacia abajo! Cuando se desencolan, vuelven hacia arriba, donde está la interfaz. ¡No caen desde el final del árbol directamente al adaptador de red!

9.6.1. Algunos ejemplos sencillos de filtrado

Como se explicó en el capítulo del Clasificador, puede hacer coincidencias con literalmente cualquier cosa, usando una sintaxis bastante complicada. Para empezar, le mostraremos cómo hacer cosas obvias, que por suerte es muy sencillo.

Digamos que tenemos una qdisc PRIO llamada «10:» que contiene tres clases, y queremos asignar todo el tráfico desde y hacia el puerto 22 a la banda de prioridad más alta. Los filtros podrían ser:

```
# tc filter add dev eth0 protocol ip parent 10: prio 1 u32 match \
  ip dport 22 0xffff flowid 10:1
# tc filter add dev eth0 protocol ip parent 10: prio 1 u32 match \
  ip sport 80 0xffff flowid 10:1
# tc filter add dev eth0 protocol ip parent 10: prio 2 flowid 10:2
```

¿Qué dice esto? Dice: asociar a eth0, nodo 10: un filtro u32 de prioridad 1 que coincide *exactamente* con el puerto IP de destino 22 y envíalo ala banda 10:1. Y luego repetimos lo mismo para el puerto de origen 80. La última orden dice que cualquier cosa que no coincida debería ir a la banda 10:2, la de siguiente prioridad.

Necesita añadir «eth0», o como sea que se llame la interfaz, porque cada interfaz tiene un espacio de controladores propio.

Para escoger sobre una dirección IP, use esto:

```
# tc filter add dev eth0 parent 10:0 protocol ip prio 1 u32 \
  match ip dst 4.3.2.1/32 flowid 10:1
# tc filter add dev eth0 parent 10:0 protocol ip prio 1 u32 \
  match ip src 1.2.3.4/32 flowid 10:1
# tc filter add dev eth0 protocol ip parent 10: prio 2      \
  flowid 10:2
```

Esto asigna el tráfico hacia 4.3.2.1 y desde 1.2.3.4 a la cola de más alta prioridad, y el resto a la de siguiente prioridad.

Puede concatenar coincidencias, para capturar el tráfico desde 1.2.3.4, puerto 80. Se hace así:

```
# tc filter add dev eth0 parent 10:0 protocol ip prio 1 u32 match ip
src 4.3.2.1/32
  match ip sport 80 0xffff flowid 10:1
```

9.6.2. Todas las órdenes de filtrado que necesitará normalmente

La mayoría de las órdenes de ajuste que se presentan aquí empiezan con éste preámbulo:

```
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 ..
```

Estas son las coincidencias llamadas «u32», que pueden coincidir con CUALQUIER parte de un paquete.
Sobre la dirección de origen/destino

> Máscara de origen «match ip src 1.2.3.0/24», máscara de destino «match ip dst 4.3.2.0/24». Para coincidir con una única máquina, use /32 u omita la máscara.

Sobre puerto de origen/destino, todos los protocolos de IP

> Origen: «match ip sport 80 0xffff», destino: «match ip dport 0xffff»

Sobre protocolo IP (tcp, udp, icmp, gre, ipsec)

> Use los números de /etc/protocols. Por ejemplo, icmp es 1: «match ip protocol 1 0xff».

Sobre fwmark

> Puede marcar los paquetes con ipchains y hacer que la marca sobreviva al rutado a través de interfaces. Esto es realmente útil para por ejemplo ajustar el tráfico de eth1 sólo si viene de eth0. Sintaxis: # tc filter add dev eth1 protocol ip parent 1:0 prio 1 handle 6 fw flowid 1:1 ¡Fíjese que ésta no es una coincidencia u32!

Puede poner una marca así:

```
# iptables -A PREROUTING -t mangle -i eth0 -j MARK --set-mark 6
```

El número 6 es arbitrario.

Si no desea entender toda la sintaxis de tc, limítese a usar iptables, y aprenda sólo a seleccionar sobre fwmark.

Sobre el campo TOS

Para escoger el tráfico interactivo, de retraso mínimo:

```
# tc filter add dev ppp0 parent 1:0 protocol ip prio 10 u32 \
      match ip tos 0x10 0xff \
     flowid 1:4
```

Use 0x08 0xff para el tráfico masivo.

Si desea ver más órdenes de filtrado, vea el capítulo de Filtros avanzados.

9.7. El dispositivo intermedio de encolado (IMQ)

El dispositivo intermedio de encolado (Intermediate queueing device) no es una qdisc pero su uso está muy unido a las qdisc. Dentro de Linux, las qdisc se asocian a dispositivos de red y todo lo que se encola en el dispositivo se encola antes en la qdisc. Partiendo de este concepto, surgen dos limitaciones:

1. Sólo se pueden hacer ajustes de salida (existe una qdisc de entrada, pero sus posibilidades son muy limitadas comparadas a las qdisc con clases).

2. Una qdisc sólo puede ver el tráfico de una interfaz, de manera que no se pueden imponer limitaciones globales.

IMQ está aquí para resolver ambas limitaciones. En breve, se puede poner todo lo que se quiera en una qdisc. Los paquetes marcados de forma especial los interceptan las ranuras NF_IP_PRE_ROUTING y NF_IP_POST_ROUTING de netfilter y pasan por una qdisc asociada a un dispositivo imq. Se usa un objetivo de iptables para marcar los paquetes.

Esto permite que haga ajustes de entrada ya que puede marcar los paquetes que vienen de cualquier sitio o tratar las interfaces como clases para imponer límites globales. También puede hacer muchas otras cosas como poner el tráfico http en una qdisc, poner las peticiones de conexiones nuevas en una qdisc, ...

9.7.1. Configuración de ejemplo

La primera cosa que me viene a la mente es usar el ajuste de entrada para darnos a nosotros mismos un ancho de banda garantizado. ;) La configuración es igual que la de cualquier otra interfaz:

```
tc qdisc add dev imq0 root handle 1: htb default 20

tc class add dev imq0 parent 1: classid 1:1 htb rate 2mbit burst 15k

tc class add dev imq0 parent 1:1 classid 1:10 htb rate 1mbit
tc class add dev imq0 parent 1:1 classid 1:20 htb rate 1mbit

tc qdisc add dev imq0 parent 1:10 handle 10: pfifo
tc qdisc add dev imq0 parent 1:20 handle 20: sfq

tc filter add dev imq0 parent 10:0 protocol ip prio 1 u32 match \
               ip dst 10.0.0.230/32 flowid 1:10
```

En este ejemplo, se usa u32 para la clasificación. Otros clasificadores podrían funcionar como queremos. Lo siguiente es escoger y marcar el tráfico para encolarlo en imq0.

```
iptables -t mangle -A PREROUTING -i eth0 -j IMQ --todev 0

ip link set imq0 up
```

Los objetivos IMQ de iptables son válidos en las cadenas PREROUTING y POSTROUTING de la tabla mangle. Su sintaxis es

```
IMQ [ --todev n ]      n : número del dispositivo imq
```

También se proporciona un objetivo para ip6tables.

Tenga en cuenta que el tráfico no se encola cuando coincide con el objetivo, sino después. La localización exacta del punto donde el tráfico entra en el dispositivo imq depende de la dirección del tráfico (entrada/salida). Estas son las ranuras predefinidas de netfilter que usa iptables:

```
enum nf_ip_hook_priorities {
        NF_IP_PRI_FIRST = INT_MIN,
        NF_IP_PRI_CONNTRACK = -200,
        NF_IP_PRI_MANGLE = -150,
        NF_IP_PRI_NAT_DST = -100,
        NF_IP_PRI_FILTER = 0,
        NF_IP_PRI_NAT_SRC = 100,
        NF_IP_PRI_LAST = INT_MAX,
};
```

Para el tráfico de entrada, imp se registra con prioridad NF_IP_PRI_MANGLE + 1, lo que significa que los paquetes entran en el dispositivo imq justo después de que se haya pasado de la cadena PREROUTING de mangle.

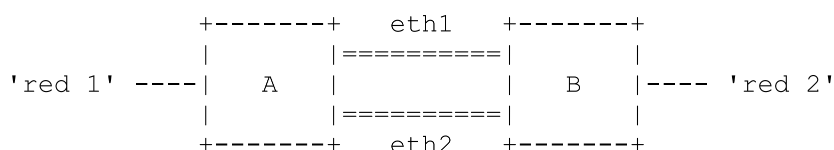
Para la salida imq usa NF_IP_PRI_LAST, que obedece al hecho de que los paquetes eliminados por la tabla filter no deben ocupar ancho de banda.

Encontrará parches e información en el sitio de imq.

Capítulo 10. Compartir la carga sobre varias interfaces

Hay varias maneras de hacer esto. Una de las más sencillas y directas es «TEQL» ("True" - o "trivial" - link equalizer). Como la mayoría de las cosas que tienen que ver con las colas, la división de cargas va en ambos sentidos. Puede que ambos extremos del enlace deban participar para conseguir un efecto completo.

Imagine esta situación:



A y B son routers, y por el momento asumiremos que ambos funcionan con Linux. Si el tráfico va de la red 1 a la red 2, el router A necesita distribuir los paquetes sobre ambos enlaces hacia B. Se necesita configurar el router B para que acepte esto. Lo mismo sucede en la otra dirección: cuando los paquetes van desde la red 2 a la red 1, el router B necesita enviar paquetes tanto sobre eth1 como eth2.

La parte de distribución la hace un dispositivo «TEQL», así (no podría ser más sencillo):

```
# tc qdisc add dev eth1 root teql0
# tc qdisc add dev eth2 root teql0
# ip link set dev teql0 up
```

¡No olvide la orden «ip link set up»!

Esto debe hacerse en ambas máquinas. El dispositivo teql0 básicamente es un distribuidor roundrobin sobre eth1 y eth2, para enviar paquetes. Nunca aparecerán datos viniendo por un dispositivo teql, sino que simplemente aparecen en las eth1 y eth2 "básicas".

Pero ahora sólo tenemos dispositivos, de manera que necesitamos hacer un rutado adecuado. Una manera de hacerlo es asignar una red /31 para ambos enlaces, y también al dispositivo teql0.

FIXME: ¿necesita esto algo como «nobroadcast»? Una /31 es demasiado pequeña para contener una dirección de red y una de difusión (si no funciona como debe, pruebe con /30 y ajuste las direcciones IP de acuerdo). ¡Incluso podría intentar que eth1 y eth2 vayan sin dirección IP!

En el router A:

```
# ip addr add dev eth1 10.0.0.0/31
# ip addr add dev eth2 10.0.0.2/31
# ip addr add dev teql0 10.0.0.4/31
```

En el router B:

```
# ip addr add dev eth1 10.0.0.1/31
# ip addr add dev eth2 10.0.0.3/31
# ip addr add dev teql0 10.0.0.5/31
```

El router A debería ser capaz ahora de hacer ping a 10.0.0.1, 10.0.0.3 y 10.0.0.5 sobre los dos enlaces reales y el ecualizado. El router B debería ser capaz de hacer ping a 10.0.0.0, 10.0.0.2 y 10.0.0.4 sobre los enlaces.

Si esto funciona, el router A debería hacer que 10.0.0.5 sea su ruta para alcanzar la red 2, y el router B debería hacer de 10.0.0.4 su ruta para alcanzar la red 1. Para el caso especial en que la red 1 sea su red casera y la red 2 Internet, el router A debería hacer de 10.0.0.5 su ruta por defecto.

10.1. Problemas

Nada es tan fácil como parece. Se necesita desactivar el filtrado de ruta de vuelta (return path filtering) en eth1 y eth2 de ambos router, porque en caso contrario descartarán los paquetes destinados direcciones IP que no sean las suyas:

```
# echo 0 > /proc/net/ipv4/conf/eth1/rp_filter
# echo 0 > /proc/net/ipv4/conf/eth2/rp_filter
```

Entonces nos encontramos con el desagradable problema de la reordenación de paquetes. Digamos que A envía 6 paquetes a B (eth1 podría llevar los 1, 3 y 5. eth2 llevaría entonces los 2, 4 y 6). En un mundo ideal, el router B los recibiría en este orden, 1, 2, 3, 4, 5, 6. Pero hay una posibilidad muy cierta de que el núcleo haga esto: 2, 1, 4, 3, 6, 5. El problema es que esto confunde a TCP/IP. Mientras que esto no es un problema para enlaces que llevan sesiones TCP/IP diferentes, no será capaz de juntar varios enlaces y conseguir enviar un único fichero por ftp mucho más rápido, a menos que el SO emisor o receptor sea Linux, que no se asusta fácilmente por un reordenado sencillo.

Sin embargo, para muchas aplicaciones, el equilibrio de carga sobre enlaces es una gran idea.

10.2. Otras posibilidades

William Stearns ha usado una configuración avanzada de túneles para conseguir un buen uso simultáneo de varias conexiones de internet sin relación entre ellas. Puede encontrarlo en su página sobre túneles.

Puede que en el futuro este Cómo diga más cosas a este respecto.

Capítulo 11. Netfilter e iproute (marcado de paquetes)

Hasta ahora hemos visto cómo funciona iproute, y hemos mencionado algunas veces netfilter. Este podría ser un buen momento para mirar las [Rusty's Remarkably Unreliable Guides](). Netfilter en sí lo puede encontrar [aquí]().

Netfilter nos permite filtrar paquetes, o cambiar sus cabeceras. Una capacidad especial es que podemos marcar un paquete con un número. Esto se hace con la estructura --set-mark.

Como ejemplo, esta orden marca todos los paquetes destinados al puerto 25, correo saliente:

```
# iptables -A PREROUTING -i eth0 -t mangle -p tcp --dport 25 \
 -j MARK --set-mark 1
```

Ahora digamos que tenemos varias conexiones, una que es rápida (y cara, por megabyte), y otra más lenta, pero de tarifa fija. Lo más seguro es que queramos hacer salir el correo mediante la ruta barata.

Ya hemos marcado los paquetes con un «1», y ahora le daremos instrucciones a la base de normas de rutado para que actúe sobre esto:

```
# echo 201 mail.out >> /etc/iproute2/rt_tables
# ip rule add fwmark 1 table correo.salida
# ip rule ls
0:      from all lookup local
32764:  from all fwmark        1 lookup correo.salida
32766:  from all lookup main
32767:  from all lookup default
```

Ahora generaremos la tabla correo.salida con una ruta al enlace lento pero barato:

```
# /sbin/ip route add default via 195.96.98.253 dev ppp0 table#
correo.salida
```

Y ya está. Si quisiéramos hacer algunas excepciones, hay muchas maneras de conseguirlo. Podemos modificar la orden de netfilter para excluir ciertas máquinas, o podemos insertar una regla con mayor prioridad que apunte a la tabla principal para dichas máquinas.

También podemos usar esta características para obedecer a los bits TOS marcando los paquetes con un tipo diferente de servicio con diferentes números, y creando reglas que actúen sobre esto. De esta manera podría dedicar, digamos, una línea RDSI a las sesiones interactivas.

No hace falta decir que esto funciona bien en una máquina que esté haciendo NAT («enmascaramiento»).

IMPORTANTE: Hemos recibido informes de que al menos MASQ y SNAT colisionan con el marcado de paquetes. Rusty Russell lo explica en este mensaje. Deshabilite el filtro de ruta inversa (reverse path filter) para que funcione correctamente.

Nota: para marcar paquetes, necesita activar algunas opciones en el núcleo:

```
IP: advanced router (CONFIG_IP_ADVANCED_ROUTER) [Y/n/?]
IP: policy routing (CONFIG_IP_MULTIPLE_TABLES) [Y/n/?]
IP: use netfilter MARK value as routing key (CONFIG_IP_ROUTE_FWMARK)
[Y/n/?]
```

Vea también el Sección 15.5 en el *Cookbook*.

Capítulo 12. Filtros avanzados para (re)clasificar paquetes

Como se explicó en la sección de disciplinas de cola con clases, se necesitan filtros para clasificar los paquetes en cualquiera de las subcolas. Estos filtros se llaman desde dentro de las qdisc con clases.

Aquí tiene una lista incompleta de los clasificadores disponibles:

fw

> Basa la decisión en la forma en que el cortafuegos ha marcado el paquete. Es una manera sencilla si no quiere aprender la sintaxis de filtros de tc. Vea el capítulo de Colas para más detalles.

u32

> Basa la decisión en campos del interior del paquete (dirección IP de origen, etc)

route

> Basa la decisión en la ruta a la que será enviado el paquete. Bases the decision on which route the packet will be routed by

rsvp, rsvp6

> Encamina los paquetes basándose en RSVP . Sólo es útil en las redes que usted controle (Internet no respeta RSVP).

tcindex

> Se usa con la qdisc DSMARK. Vea la sección relevante.

Fíjese que en general hay muchas maneras de clasificar los paquetes y que generalmente el sistema de que desea usar se reduce a sus preferencias.

En general los clasificadores aceptan unos pocos argumentos comunes. Aparecen aquí por conveniencia:

protocol

El protocolo que aceptará este clasificador. Generalmente sólo querrá aceptar tráfico IP. Se requiere.

parent

El controlador al que estará asociado este clasificador. Este controlador debe ser una clase ya existente. Requerido.

prio

La prioridad de este clasificador. Los números más bajos se comprueban antes.

handle

Este controlador significa cosas diferentes para filtros diferentes.

Todas las secciones que siguen asumen que está intentando dar forma al tráfico que va hacia `MáquinaA`. Asumen que se ha configurado la clase raíz en 1: y que la clase a la que desea enviar el tráfico elegido es 1:1.

12.1. El clasificador `u32`

El filtro U32 es el más avanzado disponible en la implementación actual. Se basa completamente en tablas hash, lo que lo hace robusto cuando hay muchas reglas de filtrado.

En su forma más simple el filtro U32 es una lista de registros, consistente cada una en dos campos: un selector y una acción. Los selectores, descritos más adelante, se comparan con el paquete IP que se está procesando hasta encontrar la primera coincidencia, y entonces se ejecuta la acción asociada. El tipo más sencillo de acción sería dirijir el paquete en una clase CBQ definida.

La línea de órdenes del programa `tc filter`, que se usa para configura el filtro, consiste en tres partes: especificación del filtro, selector y acción. La especificación del filtro puede definirse así:

```
tc filter add dev IF [ protocol PROTO ]
                     [ (preference|priority) PRIO ]
                     [ parent CBQ ]
```

El campo `protocol` describe el protocolo al que se aplicará el filtro. Sólo comentaremos el caso del protocolo `ip`. El campo `preference` (alternativamente se puede usar `priority`) establece la prioridad del filtro escogido. Esto es importante, porque puede tener varios filtros (listas de reglas) con diferentes prioridades. Se pasará

por cada lista en el orden en que se agreguen las reglas, y entonces se procesarán las listas de menor prioridad (numero "preference" más alto). El campo `parent` define la cima del árbol CBQ (por ejemplo, 1:0), a la que se asociará el filtro.

Las opciones descritas anterioremente se aplican a todos los filtros, no sólo a los U32.

12.1.1. Selector U32

El selector U32 contiene definiciones de los patrones, que serán comparados con el paquete procesado en cada momento. Para ser más precisos, define qué bits hay que comparar en la cabecera del paquete y nada más, pero este método sencillo es muy poderoso. Veamos los ejemplos siguientes, tomados directamente de un filtro muy complejo, del mundo real:

```
# tc filter add dev eth0 protocol ip parent 1:0 pref 10 u32 \
  match u32 00100000 00ff0000 at 0 flowid 1:10
```

Por ahora, olvidémonos de la primera línea (todos estos parámetros describen las tablas hash del filtro). Centrémonos en la línea del selector, que contiene la palabra clave `match`. Este selector coincidirá con cabeceras IP cuyo segundo byte sea 0x10 (0010). Como puede adivinar, el número 00ff es la máscara de comparación, que le dice al filtro qué bits exactamente tiene que comparar. Aquí es 0xff, de manera que el byte coincidirá si es exactamente 0x10. La clave `at` indica que la coincidencia debe comenzar en un desplazamiento específico (en bytes) (en este caso, al principio del paquete). Traduciendo todo esto a lenguaje humano, el paquete coincidirá si su campo Tipo de Servicio tiene los bits de «poco retraso» activos. Analicemos otra regla:

```
# tc filter add dev eth0 protocol ip parent 1:0 pref 10 u32 \
  match u32 00000016 0000ffff at nexthdr+0 flowid 1:10
```

La opción `nexthdr` indica la siguiente cabecera encapsulada en el paquete IP, esto es, la cabecera del protocolo de la capa superior. La coincidencia también debe empezar al principio de la siguiente cabecera. La coincidencia debería ocurrir en la segunda palabra de 32 bits te la cabecera. En los protocolos TCP y UDP este campo contiene el puerto de destino del paquete. El puerto se da en formato big-endian, es decir, los bits más significativos primero, de manera que simplemente leemos 0x0016 como 22 decimal, lo que indicaría el servicio SSH si fuera TCP. Como puede adivinar, esta coincidencia es ambigua sin un contexto, y de eso hablaremos más tarde.

Habiendo entendido todo lo anterior, encontraremos fácil de leer el siguiente selector: `match c0a80100 ffffff00 at 16`. Lo que tenemos aquí es una coincidencia de tres bytes partiendo del decimoséptimo byte, contando desde el principio de la cabecera IP. Coincidirá con los paquetes cuya dirección de destino sea cualquiera dentro de la red 192.168.1/24. Tras analizar los ejemplo, podemos resumir lo que hemos aprendido.

12.1.2. Selectores generales

Los selectores generales definen el patrón, máscara y desplazamiento del patrón que será comparado con el contenido del paquete. Usando los selectores generales se puede coincidir con virtualmente cualquier bit de la cabecera IP (o capas superiores). Sin

embargo, son más difíciles de leer y escribir que los selectores específicos descritos más adelante. La sintaxis general del selector es:

```
match [ u32 | u16 | u8 ] PATRON MASCARA [ at DESPL | nexthdr+DESPL]
```

Una de las palabras clave `u32`, `u16` o `u8` especifica la longitud en bits de los patrones. Deberían seguirle PATRON y MASCARA, de la longitud definida por la palabra clave anterior. El parámetro DESPL es el desplazamiento, en bytes, desde donde empezar a comparar. Si se da la palabra clave `nexthdr+`, el desplazamiento es relativo al inicio de una cabecera de capa superior.

Algunos ejemplos:

```
# tc filter add dev ppp14 parent 1:0 prio 10 u32 \
     match u8 64 0xff at 8 \
     flowid 1:4
```

Un paquete coincidirá con esta regla si su tiempo de vida (TTL) es 64. TTL es el campo que empieza justo después del octavo byte de la cabecera IP.

```
# tc filter add dev ppp14 parent 1:0 prio 10 u32 \
     match u8 0x10 0xff at nexthdr+13 \
     protocol tcp \
     flowid 1:3
```

FIXME: se nos ha indicado que esta sintaxis ya no funciona.

Use esto para capturar los ACK en paquetes menores de 64 bytes:

```
## captura de acks a la manera difícil:
## IP protocol 6,
## IP header length 0x5(32 bit words),
## IP Total length 0x34 (ACK + 12 bytes of TCP options)
## TCP ack set (bit 5, offset 33)
# tc filter add dev ppp14 parent 1:0 protocol ip prio 10 u32 \
            match ip protocol 6 0xff \
            match u8 0x05 0x0f at 0 \
            match u16 0x0000 0xffc0 at 2 \
            match u8 0x10 0xff at 33 \
            flowid 1:3
```

Esta regla sólo coincidirá con paquetes TCP cuando esté activo el bit ACK, y no haya contenido. Aquí podemos ver un ejemplo de uso de dos selectores, siendo el resultado final el AND lógico de sus resultados. Si echamos un vistazo al diagrama de la cabecera TCP, podemos ver que el bit ACK es el tercer bit de más peso (0x10) en el catorceavo byte de la cabecera TCP (`at nexthdr+13`). En lo que respecta al segundo selector, si quisiéramos hacernos la vida más complicada, podríamos escribir `match u8 0x06 0xff at 9` en lugar de usar el selector específico `protocol tcp`, porque 6 es el número del protocolo TCP, presente en el décimo byte de la cabecera IP. Por otro lado, en este ejemplo no podríamos usar selectores específicos para la primera coincidencia (simplemente porque no hay selectores específicos para el bit ACK de TCP).

12.1.3. Selectores específicos

La siguiente tabla contiene una lista de todos los selectores específicos que ha encontrado el autor de esta sección en el código fuente del programa `tc`. Simplemente te hacen la vida más sencilla e incrementan la legibilidad de la configuración de los filtros.

FIXME: aquí iría la tabla (la tabla está en un fichero aparte, "selector.html"

FIXME: y todavía está en polaco :-(

FIXME: debe ser sgmlizada

Algunos ejemplos:

```
# tc filter add dev ppp0 parent 1:0 prio 10 u32 \
     match ip tos 0x10 0xff \
     flowid 1:4
```

FIXME: el filtro tcp dst no funciona como se describe:

La regla anterior coincidirá con paquetes que tengan el campo TOS a 0x10. El campo TOS empieza en el segundo byte del paquete y es de un byte de largo, de manera que podríamos escribir un selector general equivalente: `match u8 0x10 0xff at 1`. Esto nos da una pista de las interioridades del filtro U32 (las reglas específicas siempre se traducen a generales, y es de esta manera que se almacenan en la memoria del núcleo). Lo que nos lleva a otra conclusión: los selectores `tcp` y `udp` son exactamente el mismo, y ésta es la razón de que no podamos usar un único selector `match tcp dst 53 0xffff` para capturar paquetes TCP enviados a dicho puerto (también coincidiría con los paquetes UDP). También debe recordar especificar el protocolo y terminará con la siguiente regla:

```
# tc filter add dev ppp0 parent 1:0 prio 10 u32 \
        match tcp dst 53 0xffff \
        match ip protocol 0x6 0xff \
        flowid 1:2
```

12.2. El clasificador route

Este clasificador filtra basándose en los resultados de las tablas de rutas. Cuando un paquete que está pasando por las clases llega a uno marcado con el filtro «route», queda clasificado basándose en la información de la tabla de rutas.

```
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 route
```

Aquí añadimos un clasificador de rutas al nodo padre 1:0 con prioridad 100. Cuando un paquete alcanza este nodo (que, como es el raíz, sucederá de inmediato) consultará la tabla de rutas y si una coincide, lo enviará a la clase dada y le dará prioridad 100. Entonces, para entrar en acción del todo, deberá añadir la ruta apropiada:

El truco aquí es definir «realm» (dominio) basándose en el destino o el origen. La forma de hacerlo es ésta:

```
# ip route add Máquina/Red via Pasarela dev Dispositivo realm
NúmeroDominio
```

Por ejemplo, podemos definir nuestra red de destino 192.168.10.0 con un número de dominio 10:

```
# ip route add 192.168.10.0/24 via 192.168.10.1 dev eth1 realm 10
```

Cuando añadimos filtros de rutas, podemos usar números de dominios para representar las redes o máquinas y especificar cómo se comparan estas redes con los filtros.

```
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 \
  route to 10 classid 1:10
```

La regla anterior dice que los paquetes que vayan a la red 192.168.10.0 coinciden con la clase de id 1:10.

El filtro de rutas también se puede usar para coincidir con rutas de origen. Por ejemplo, hay una subred asociada al router Linux en eth2.

```
# ip route add 192.168.2.0/24 dev eth2 realm 2
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 \
  route from 2 classid 1:2
```

Aquí el filtro especifica que el paquete de la subred 192.168.2.0 (dominio 2) coincidirá con la clase de id 1:2.

12.3. Filtros de control (Policing filters)

Para hacer las configuraciones incluso más complicadas, puede tener filtros que sólo funcionen hasta un determinado ancho de banda, o que sólo salten cuando el ancho de banda exceda una tasa determinada.

De manera que si decide controlar 4mbit/s pero hay 5mbit/s de tráfico, puede dejar de comprobar los 5mbit/s al completo, o sólo 1mbit/s y enviar 4mbit/s del total a la clase configurada.

Si el ancho de banda excede la tasa configurada, puede descartar un paquete, reclasificarlo, o ver si algún otro filtro lo captura.

12.3.1. Formas de control

Básicamente hay dos formas de control. Si compiló «Estimadores» en el núcleo, entonces se puede medir cuánto tráfico está pasando por cada filtro, más o menos. Estos estimadores consumen poca CPU, ya que sólo cuentan cuántos datos han estado pasando 25 veces por segundo, y con eso calculan la tasa de bits.

La otra forma es de nuevo mediante un Token Bucket Filter, que esta vez residirá dentro del filtro. El TBF sólo trabaja con tráfico HASTA el ancho de banda configurado, y si se ofrece más, sólo el exceso es sujeto a la acción de sobrelímite configurada.

12.3.1.1. Con el estimador del núcleo

Es muy simple y sólo tiene un parámetro: avrate. O el flujo se mantiene por debajo de avrate, y el filtro clasifica el tráfico en el id de clase indicado, o la tasa excede en cuyo caso se ejecuta la acción especificada, que es «reclassify» por defecto.

El núcleo usa una Media de Movimiento Exponencial Proporcional (Exponential Weighted Moving Average) que lo hace menos sensible a ráfagas cortas.

12.3.1.2. Con Token Bucket Filter

Usa los siguientes parámetros:

- buffer/maxburst
- mtu/minburst
- mpu
- rate

Que se comportan de forma casi idéntica que las descritas en la sección del Token Bucket Filter. Sin embargo, tenga en cuenta que si establece demasiado bajo el mtu de un control TBF, no pasará *ningún* paquete, mientras que la qdisc TBF de salida se limitará a pasarlos más lentamente.

Otra diferencia es que un filtro de control sólo puede dejar pasar un paquete, o descartarlo. No puede mantenerlo dentro de sí para retrasarlo.

12.3.2. Acciones de sobrelímite

Si nuestro filtro decide que está sobrelimitado, puede tomar «acciones». Actualmente, disponemos de tres:

continue

> Hace que el filtro no de coincidencia, aunque puede que lo hagan otros filtros.

drop

> Esta es una opción muy salvaje que se limita a descartar el tráfico que excede de una cierta tasa. Se usa a menudo en el control deentrada y tiene usos limitados. Por ejemplo, puede tener un servidor de nombres que se cae si se le pasa más de 5mbit/s de paquetes, en cuyo caso se debería usar un filtro de entrada para asegurarse de que no se le ofrecen más.

Pass/OK

Dejar pasar el tráfico. Se puede usar para desactivar un filtro complicado, sin eliminarlo de su sitio.

reclassify

Más a menudo se tiene a una reclasificación al Mejor Esfuerzo. Esta es la acción por defecto.

12.3.3. Ejemplos

El único ejemplo real conocido se menciona en la sección «Protección de la máquina frente a inundaciones por SYN».

FIXME: si ha usado esto, comparta su experiencia con nosotros

12.4. Filtros de hash para filtrado masivo muy rápido

Si alguna vez ha necesitado miles de reglas, por ejemplo si tiene muchos clientes o máquinas, todas con especificaciones QoS diferentes, habrá comprobado que el núcleo se pasa mucho tiempo comprobando todas esas reglas.

Por defecto, muchos filtros residen en una gran cadena que se comprueba en orden descendiente de prioridad. Si tiene 1000 reglas, puede hacer falta 1000 comprobaciones para determinar qué hacer con un paquete.

La comprobación podría ser mucho más rápida si tuviera 256 reglas cada una con cuatro reglas (si pudiera dividir los paquetes en esas 256 cadenas, de manera que ajustasen).

El hash hace esto posible. Digamosq ue tiene 1024 clientes de cable módem en su red, con direcciones IP desde 1.2.0.0 a 1.2.3.255, y que cada una va en otra categoría, por ejemplo «pequeña», «normal» y «premium». Entonces podría tener 1024 reglas como ésta:

```
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src
\
  1.2.0.0 classid 1:1
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src
\
  1.2.0.1 classid 1:1
...
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src
\
  1.2.3.254 classid 1:3
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src
\
  1.2.3.255 classid 1:2
```

Para acelerar esto, podemos usar la última parte de la dirección IP como una "clave hash". Entonces obtenemos 256 tablas, la primera de las cuales sería así:

```
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src
\
  1.2.0.0 classid 1:1
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src
\
  1.2.1.0 classid 1:1
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src
\
  1.2.2.0 classid 1:3
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src
\
  1.2.3.0 classid 1:2
```

La siguiente empezaría así:

```
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src
\
  1.2.0.1 classid 1:1
...
```

De esta manera, sólo se necesitan cuatro comprobaciones como mucho, y dos de media.

La configuración es muy complicada, pero valdrá la pena cuando llegue el momento en que tenga tantos clientes. Primero hacemos un filtro raíz, y luego creamos una tabla con 256 entradas:

```
# tc filter add dev eth1 parent 1:0 prio 5 protocol ip u32
# tc filter add dev eth1 parent 1:0 prio 5 handle 2: protocol ip u32
divisor 256
```

Ahora añadimos algunas reglas a las entradas que hemos creado en la tabla:

```
# tc filter add dev eth1 protocol ip parent 1:0 prio 5 u32 ht 2:7b: \
        match ip src 1.2.0.123 flowid 1:1
# tc filter add dev eth1 protocol ip parent 1:0 prio 5 u32 ht 2:7b: \
        match ip src 1.2.1.123 flowid 1:2
# tc filter add dev eth1 protocol ip parent 1:0 prio 5 u32 ht 2:7b: \
        match ip src 1.2.3.123 flowid 1:3
# tc filter add dev eth1 protocol ip parent 1:0 prio 5 u32 ht 2:7b: \
        match ip src 1.2.4.123 flowid 1:2
```

Esta es la entrada 123, que contiene coincidencias para 1.2.0.123, 1.2.1.123, 1.2.2.123, 1.2.3.123, y las envía a 1:1, 1:2, 1:3 y 1:2 respectivamente. Fíjese que necesitamos especificar nuestro hash en hexadecimal, 0x7b is 123.

Luego creamos un «filtro de hash» que dirige el tráfico a la entrada adecuada de la tabla de hash:

```
# tc filter add dev eth1 protocol ip parent 1:0 prio 5 u32 ht 800:: \
        match ip src 1.2.0.0/16 \
        hashkey mask 0x000000ff at 12 \
        link 2:
```

Bueno, hace falta explicar algunos números. La tabla de hash por defecto se llama 800:: y todo el filtrado empieza aquí. Entonces escogemos la dirección de origen, que reside en las posiciones 12, 13, 14 y 15 de la cabecera IP, e indicamos que sólo estamos interesados en la última parte. Entonces lo enviamos a la tabla de hash 2:, que creamos previamente.

Es bastante complicado, pero funciona en la práctica y el rendimiento será asombroso. Sepa que este ejemplo se puede mejorar hasta el caso ideal en que cada cadena ¡sólo contenga 1 filtro!

Capítulo 13. Parámetros de red del núcleo

El núcleo tiene muchos parámetros que se pueden ajustar para diferentes circunstancias. Aunque, como de costumbre, los parámetros por defecto sirven muy bien para el 99% de las instalaciones, ¡no llamamos a éste el Cómo Avanzado sólo por diversión!

Las cosas interesantes están en /proc/sys/net, así que eche un vistazo ahí. No todo lo documentaremos aquí al principio, pero estamos trabajando en ello.

Mientras tanto, puede que quiera echar un vistazo a las fuentes del núcleo de Linux; lea el fichero Documentation/filesystems/proc.txt. La mayoria de las características se explican allí.

(FIXME)

13.1. Reverse Path Filtering

Por defecto, los routers encaminan todo, incluyendo paquetes que «obviamente» no pertenecen a nuestra red. Un ejemplo común es un escape de un espacio IP privado a Internet. Si tiene una interfaz con una ruta de 195.96.96.0/24 hacia ella, no esperará que lleguen allí paquetes de 212.64.94.1.

Mucha gente querría desactivar esto, de manera que los hacker núcleo lo han hecho sencillo. Hay ficheros en /proc donde decir que quiere que el núcleo haga esto por usted. El método se llama «"Reverse Path Filtering». Básicamente, si la respuesta a un paquete no debería haber salido de la interfaz por la que vino, entonces este paquete es erróneo y debería ser ignorado.

El siguiente fragmento lo activará para todas las interfaces actuales y futuras.

```
# for i in /proc/sys/net/ipv4/conf/*/rp_filter ; do
>  echo 2 > $i
> done
```

Siguiendo el ejemplo anterior, si un paquete llega al router Linux por eth1 diciendo que viene de la red Oficina+ISP, debería ser eliminado. De forma similar, si un paquete viene de la subred Oficina, diciendo que viene de algún sitio de fuera del cortafuegos, también debería ser eliminado.

Lo anterior es un «reverse path filtering» completo. La acción por defecto es filtrar basándose sólo en direcciones IP que estén en las redes conectadas directamente. Esto se debe a que el filtrado completo a veces no funciona en el caso de encaminamiento

asimétrico, cuando los paquetes vienen por una vía y salen por otra, como tráfico de satélite, o si tiene rutas dinámicas (bgp, ospf, rip) en la red. Los datos vienen por la antena del satélite y las respuestas van mediante líneas terrestres normales.

Si esta excepción se le aplica (y probablemente lo sabrá si esto es así) sencillamente puede desactivar el rp_filter de la interfaz por donde vienen los datos del satélite. Si quiere ver si se están descartando paquetes, el fichero log_martians de ese mismo directorio le dirá al núcleo que los registre mediante syslog.

```
# echo 1 >/proc/sys/net/ipv4/conf/<nombredeinterfaz>/log_martians
```

FIXME: ¿basta con activar los ficheros conf/{default,all}/*? - martijn

13.2. Configuraciones oscuras

Bien, hay muchos parámetros que se pueden modificar. Intentaremos dar una lista de todos. También están documentados (parcialmente) en Documentation/ip-sysctl.txt.

Algunas de estas configuraciones tienen valores por defecto diferentes dependiendo de si ha respondido «Sí» a la pregunta «Configure as router and not host» al compilar el núcleo.

Oskar Andreasson también tiene una página con todas estas opciones y parece que es mejor que lo nuestro, de manera que mire también en http://ipsysctl-tutorial.frozentux.net/.

13.2.1. ipv4 genérica

Como nota general, la mayoría de las capacidades de limitación de velocidad no funcionan sobre el loopback, de manera que no las pruebe de forma local. Los límites se indican en «jiffies», y se fuerzan usando el token bucket filter que mencionamos anteriormente.

El núcleo tiene un reloj interno que funciona a «HZ» tics (o «jiffies») por segundo. En Intel, «HZ» suele ser 100. De manera que cambiar el fichero _rate a, digamos 50, permitirá 2 paquetes por segundo. El token bucket filter está configurado también para permitir ráfagas de como mucho 6 paquetes, si se han recibido ya suficientes paquetes.

Mucho del contenido de la siguiente lista se ha copiado de /usr/src/linux/Documentation/networking/ip-sysctl.txt, escrito por Alexey Kuznetsov <kuznet@ms2.inr.ac.ru> y Andi Kleen <ak@muc.de>

/proc/sys/net/ipv4/icmp_destunreach_rate

> Si el núcleo decide que no puede entregar un paquete, lo descartará, y enviará a su origen un aviso ICMP al efecto.

/proc/sys/net/ipv4/icmp_echo_ignore_all

> No actuar sobre los paquetes de eco. Por favor, no active esto por defecto, pero en caso de ser usado como reenvío para un ataque DoS, puede ser útil.

/proc/sys/net/ipv4/icmp_echo_ignore_broadcasts [Util]

> Si hace ping a la dirección de difusión de una red, se supone que responderán todas las máquinas. Esto se convierte en una herramienta de negación de servicio bastante buena. Póngalo a 1 para ignorar estos mensajes de difusión.

/proc/sys/net/ipv4/icmp_echoreply_rate

> La tasa a la que se envían mensajes de eco hacia cualquier destino.

/proc/sys/net/ipv4/icmp_ignore_bogus_error_responses

> Active esto para ignorar los errores ICMP causados por máquinas de la red que reaccionen mal a tramas enviadas a lo que perciben como la dirección de difusión.

/proc/sys/net/ipv4/icmp_paramprob_rate

> Un mensaje ICMP relativamente desconocido, que se envía en respuesta a paquetes incorrectos con cabeceras IP o TCP inválidas. Con este fichero puede controlar la tasa a la que son enviados.

/proc/sys/net/ipv4/icmp_timeexceed_rate

> Esta es la causa de la famosa «estrella del centro de Solaris» en las trazas de ruta. Limita el número de mensajes Time Exceeded ICMP que se envían.

/proc/sys/net/ipv4/igmp_max_memberships

> Número máximo de sockets igmp (multicast) a la escucha en la máquina. FIXME: ¿Es esto cierto?

/proc/sys/net/ipv4/inet_peer_gc_maxtime

> FIXME: ¿Añadir una breve explicación sobre el almacenamiento de inet peer? Intervalo mínimo entre las pasadas de recolección de basuras. Este intervalo toma efecto mientras hay baja (o no hay) presión de memoria en la reserva. Se mide en jiffies.

/proc/sys/net/ipv4/inet_peer_gc_mintime

> Intervalo mínimo entre pasadas de recolección de basura. Este intervalo tiene efecto durante presiones altas de memoria en la reserva. Se mide en jiffies.

/proc/sys/net/ipv4/inet_peer_maxttl

Tiempo máximo de vida de las entradas. Las que no se usan expirarán tras este periodo de tiempo si no hay presión de memoria en la reserva (osea, cuando el número de entradas en la reserva sea muy pequeño). Se mide en jiffies.

/proc/sys/net/ipv4/inet_peer_minttl

Tiempo mínimo de vida de las entradas. Debería ser suficiente para cubrir el tiempo de vida de los fragmentos en el lado del reensamblaje. Este tiempo de vida mínimo queda garantizado si el tamaño de la reserva es menor que inet_peer_threshold. Se mide en jiffies.

/proc/sys/net/ipv4/inet_peer_threshold

Tamaño aproximado del almacenamiento INET peer. Pasado este umbral, se eliminarán entradas de forma agresiva. Este umbral también determina el tiempo de vida de las entradas y los intervalos de tiempo entre pasadas de la recolección de basura. Más entradas, menos tiempo de vida, menos intervalos de recolección de basuras.

/proc/sys/net/ipv4/ip_autoconfig

Este fichero contiene el número 1 si la máquina recibió su configuración IP mediante RARP, BOOTP, DHCP o un mecanismo similar. En caso contrario, es cero.

/proc/sys/net/ipv4/ip_default_ttl

Tiempo de vida de los paquetes. Viene con un valor sensato de 64. Auméntelo si tiene una red enorme. No lo haga por diversión (los bucles de rutas causan mucho más daño de esta manera). Incluso podría considerar reducirlo bajo ciertas circunstancias.

/proc/sys/net/ipv4/ip_dynaddr

Necesita configurarlo si usa llamada bajo demanda (dial on demand) con una dirección de interfaz dinámica. Una vez que la demanda levanta la interfaz, se reenlazará con la dirección correcta cualquier socket TCP local que no haya visto respuestas. Esto resuelve el problema de que la conexión que levanta la interfaz en sí misma no funcione, sino al segundo intento.

/proc/sys/net/ipv4/ip_forward

Si el núcleo debería intentar reenviar paquetes. Inactivo por defecto.

/proc/sys/net/ipv4/ip_local_port_range

Rango de puertos locales para las conexiones de salida. En realidad es muy pequeño por defecto, 1024 a 4999.

/proc/sys/net/ipv4/ip_no_pmtu_disc

Active esto si desea desactivar Path MTU discovery (una técnica para determinar la Maximum Transfer Unit más grande posible en su ruta. Consulte también la sección sobre Path MTU discovery en el capítulo *Cookbook*.

/proc/sys/net/ipv4/ipfrag_high_thresh

Cantidad máxima de memoria a usar para reensamblar los fragmentos IP. Cuando se llegan a reservar ipfrag_high_thresh bytes de memoria para este propósito, el gestor de fragmentos descartará paquetes hasta que se alcance ipfrag_low_thresh.

/proc/sys/net/ipv4/ip_nonlocal_bind

Active esto si desea que sus aplicaciones sean capaces de asociarse a una dirección que no pertenezca a un dispositivo de su sistema. Esto puede ser útil cuando la máquina esté en un enlace no permanente (o incluso dinámico), de manera que los servicios puedan arrancar y asociarse a una dirección específica incluso cuando el enlace no está presente.

/proc/sys/net/ipv4/ipfrag_low_thresh

Memoria mínima usada para reensamblar fragmentos IP.

/proc/sys/net/ipv4/ipfrag_time

Tiempo en segundos que se mantendrá un fragmento IP en memoria.

/proc/sys/net/ipv4/tcp_abort_on_overflow

Marca booleana que controla el comportamiento bajo gran cantidad de peticiones de conexión. Cuando se activa, hace que el núcleo envíe paquetes RST de forma activa si un servicio está sobrecargado.

/proc/sys/net/ipv4/tcp_fin_timeout

Tiempo que se mantendrá un socket en estado FIN-WAIT-2, si ha sido cerrado en su lado. El otro extremo puede estar mal y no cerrar nunca por ese lado, o incluso terminar de forma inesperada. El valor por defecto es de 60 segundos. El valor habitual usado en 2.2 era 180 segundos, y puede restaurarlo, pero recuerde que incluso si su máquina es un servidor WEB de poca carga, se arriesga a inundar la memoria con kilotones de sockets muertos, ya que aunque los sockets FIN-WAIT-2 son menos peligrosos que los FIN-WAIT-1, porque consumen 1.5K de memoria, tienden a vivir más. Vea tcp_max_orphans.

/proc/sys/net/ipv4/tcp_keepalive_time

Frecuencia con que TCP envía mensajes keepalive cuando se activa keepalive. Por defecto: 2 horas.

/proc/sys/net/ipv4/tcp_keepalive_intvl

Con qué frecuencia se retransmiten sondas, cuando no son recibe respuesta a ellas. Por defecto: 75 segundos.

/proc/sys/net/ipv4/tcp_keepalive_probes

Cuantas sondas keepalive enviará TCP, hasta que decida que la conexión está mal. Valor por defecto: 9. Multiplicada por tcp_keepalive_intvl, nos da el tiempo que puede estar un enlace sin responder tras haber enviado un keepalive.

/proc/sys/net/ipv4/tcp_max_orphans

Número máximo de sockets TCP que no están asociados con ningún controlador de fichero de usuario, mantenido por el sistema. Si se excede de este número, se reinician inmediatamente las conexiones huérfanas y se imprime un mensaje de aviso. Este límite existe sólo para prevenir ataques DoS sencillos, y _no debe_ depender de él o reducirlo de forma artificial, sino incrementarlo, en todo caso (probablemente, tras un aumento de la memoria instalada), si las condiciones de red precisan más que el valor por defecto, y ajuste los servicios de red para que hagan persistir y destruyan tales estados de forma más enérgica. Permítame recordárselo de nuevo: cada huérfano consume hasta 64K de memoria física no intercambiable.

/proc/sys/net/ipv4/tcp_orphan_retries

Cuántas veces hay que reintentar antes de matar una conexión TCP, cerrada por su lado. El valor por defecto de 7 corresponde a 50seg-16min dependiendo en el RTO. Si la máquina es un servidor web muy cargado, debería pensar en reducir este valor, ya que tales sockets pueden consumir recursos significativos. Vea tcp_max_orphans.

/proc/sys/net/ipv4/tcp_max_syn_backlog

Número máximo de peticiones de conexión a recordar, que aún no hayan recibido confirmación del cliente que conecta. El valor por defecto es 1023 para sistemas con más de 128Mb de memoria, y 128 para máquinas con poca memoria. Si el servidor sufre sobrecargas, intente aumentar este número. ¡Cuidado! Si lo hace más grande que 1024, sería mejor que cambiase TCP_SYNQ_HSIZE en include/net/tcp.h para hacer que TCP_SYNQ_HSIZE*16<=tcp_max_syn_backlog y después recompile el núcleo.

/proc/sys/net/ipv4/tcp_max_tw_buckets

Número máximo de sockets en estado time-wait mantenidos de forma simultánea por el sistema. Si se excede este número se destruyen estos sockets inmediatamente y se imprime un aviso. Este límite existe sólo para prevenir ataques DoS sencillos, y _no debe_ bajar el límite de forma artificial, sino en

todo caso aumentarlo (probablemente, tras incrementar la memoria instalada), si las condiciones de la red precisan más que el valor por defecto.

/proc/sys/net/ipv4/tcp_retrans_collapse

Compatibilidad fallo-por-fallo con determinadas impresoras defectuosas. Durante la retransmisión intentará enviar paquetes más grandes para saltarse los fallos de ciertas pilas TCP.

/proc/sys/net/ipv4/tcp_retries1

Cuántas veces hay que reintentar antes de decidir que algo está mal y que se necesita informar de esta sospecha a la capa de red. El valor mínimo según el RFC es 3, y es el valor por defecto, que corresponde con 3seg-8min dependiendo del RTO.

/proc/sys/net/ipv4/tcp_retries2

Cuántas veces hay que reintentar antes de matar una conexión TCP viva. El RFC 1122 dice que este límite debería ser mayor de 100 segundos. Este número es demasiado pequeño. El valor por defecto de 15 corresponde a 13-30min dependiendo del RTO.

/proc/sys/net/ipv4/tcp_rfc1337

Este booleano activa una corrección para los «riesgos de asesinato de time-wait en tcp», descritos en el RFC 1337. Si se activa, hace que el núcleo descarte paquetes RST para los sockets que están en el estado time-wait. Por defecto: 0

/proc/sys/net/ipv4/tcp_sack

Usar ACK Selectivo que puede servir para comprobar si se están perdiendo paquetes específicos (por tanto, ayuda a una recuperación rápida).

/proc/sys/net/ipv4/tcp_stdurg

Usa la interpretación de requerimientos de la máquina del campo puntero urg de TCP. Muchas máquinas usan la interpretación BSD antigua, de manera que si pone esto en marcha en Linux podría no comunicarse correctamente con ellas. Por defecto: FALSO

/proc/sys/net/ipv4/tcp_syn_retries

Número de paquete SYN que enviará el núcleo antes de rendirse durante el inicio de una conexión.

/proc/sys/net/ipv4/tcp_synack_retries

Para abrir el otro lado de una conexión, en núcleo envía un SYN junto con un ACK, para confirmar el SYN que recibió anteriormente. Esta es la segunda parte

del saludo de tres vías (N. del T: threeway handshake, el inicio de una conexión TCP). Esta configuración determina el número de paquetes SYN+ACK a enviar antes de que el kernel decida rendirse.

/proc/sys/net/ipv4/tcp_timestamps

Los registros de tiempo se usan, entre otras cosas, para protegerse contra números de secuencia que se solapen. Es concebible que un enlace de 1 gigabit reencuentre un número de secuencia previo con un valor fuera de línea, porque fue de una generación previa. El registro de tiempo le ayudará a reconocer este "paquete antiguo".

/proc/sys/net/ipv4/tcp_tw_recycle

Activar un reciclado rápido de sockets TIME-WAIT. El valor por defecto es 1. No debería cambiarlo si no es por consejo/petición de expertos técnicos.

/proc/sys/net/ipv4/tcp_window_scaling

TCP/IP normalmente permite ventanas de hasta 65535 bytes. En redes realmente rápido, esto podría no ser suficiente. Las opciones de escalado permiten ventanas de hasta casi un gigabyte, lo cual es bueno para productos con gran ancho de banda y mucho retraso.

13.2.2. Configuración por dispositivo

DEV puede significar una interfaz real, o "todas" o "por defecto". Por defecto también cambia la configuración de las interfaces que aún están por crear.

/proc/sys/net/ipv4/conf/DEV/accept_redirects

Si un router decide que estamos usándolo para un propósigo equivocado (necesita reenviar los paquetes por la misma interfa que los recibión), nos enviará un ICMP Redirect. Sin embargo, esto implica un pequeño riesgo de seguridad, y puede que quiera desactivarlo, o usar secure redirects.

/proc/sys/net/ipv4/conf/DEV/accept_source_route

Ya no se usa mucho. Antes se podía dar a un paquete una lista de direcciones IP que debería visitar por el camino. Linux puede obedecer a esta opción de IP.

/proc/sys/net/ipv4/conf/DEV/bootp_relay

Aceptar paquetes con dirección de origen 0.b.c.d y destino diferente a esta máquina como si fueran locales. Se supone que un demonio de reenvío de BOOTP atrapará y reenviará estos paquetes.

Por defecto es 0, ya que esta característica no está implementada aún (hablamos de la versión del núcleo 2.2.12).

/proc/sys/net/ipv4/conf/DEV/forwarding

Activar o desactivar el reenvío de IP por esta interfaz.

/proc/sys/net/ipv4/conf/DEV/log_martians

Vea la sección sobre *Reverse Path Filtering*.

/proc/sys/net/ipv4/conf/DEV/mc_forwarding

Para hacer reenvío de multicast en esta interfaz.

/proc/sys/net/ipv4/conf/DEV/proxy_arp

Si lo pone a 1, esta interfaz responderá a consultas ARP para direcciones hacia las que el núcleo tiene rutas. Puede ser útil cuando se construyen "pseudo bridges ip". ¡Tenga cuidado de que las máscaras de red sean correctas antes de activar esto! ¡Tenga en cuenta también que rp_filter, mencionado anteriormente, también opera sobre las consultas ARP!

/proc/sys/net/ipv4/conf/DEV/rp_filter

Vea la sección sobre *Reverse Path Filtering*.

/proc/sys/net/ipv4/conf/DEV/secure_redirects

Aceptar mensajes ICMP redirect sólo en las pasarelas, que aparecen en la lista de pasarelas por defecto. Activo por defecto.

/proc/sys/net/ipv4/conf/DEV/send_redirects

Si vamos a enviar las redirecciones mencionadas previamente.

/proc/sys/net/ipv4/conf/DEV/shared_media

Si no está activo, el núcleo no asume que que las diferentes subredes a las que accede este dispositivo se puedan comunicar directamente. El valor por defecto es «sí».

/proc/sys/net/ipv4/conf/DEV/tag

FIXME: hay que rellenar esto

13.2.3. Normas de vecinos (Neighbor policy)

Dev puede significar una interfaz real, o "todas" o "por defecto". Por defecto también cambia la configuración de las interfaces que aún están por crear.

/proc/sys/net/ipv4/neigh/DEV/anycast_delay

Retraso aleatorio máximo de las respuestas a los mensajes Neigbor Solicitation en jiffies (1/100 seg). Aún no está implementado (Linux todavía no soporta anycast).

/proc/sys/net/ipv4/neigh/DEV/app_solicit

Determina el número de peticiones a enviar al demonio ARP de nivel de usuario. Use 0 para apagarlo.

/proc/sys/net/ipv4/neigh/DEV/base_reachable_time

Un valor de base usado para calcular al azar el tiempo de accesibilidad (Reachable Time) tal como se especifica en el RFC2461.

/proc/sys/net/ipv4/neigh/DEV/delay_first_probe_time

Retraso de la primera sonda de tiempo si se puede acceder al vecino (vea gc_stale_time)

/proc/sys/net/ipv4/neigh/DEV/gc_stale_time

Determina con qué frecuencia se comprueban las entradas congeladas en ARP. Tras congelarse una entrada ARP se intentará resolver de nuevo (lo cual es útil cuando una dirección IP pasa a otra máquina). Cuando ucast_solicit es mayor que 0, primero intenta enviar un paquete ARP directamente a la máquina conocida. Si esto falla y mcast_solicit es mayor que 0, se difunde una petición ARP.

/proc/sys/net/ipv4/neigh/DEV/locktime

Una entrada de ARP/neighbor sólo se reemplaza con una nueva si el valor antiguo es de hace al menos "locktime". Esto evita sacudidas en la caché ARP.

/proc/sys/net/ipv4/neigh/DEV/mcast_solicit

Número máximo de reintetnos para una solicitud multicast.

/proc/sys/net/ipv4/neigh/DEV/proxy_delay

Tiempo máximo (el tiempo real es aleatorio [0..proxytime]) antes de contestar a una consulta ARP para la cual tenemos una entrada proxy ARP. En algunos casos, se usa para evitar inundaciones de red.

/proc/sys/net/ipv4/neigh/DEV/proxy_qlen

Longitud máxima de la cola del temporizador proxy arp retrasado. (vea proxy_delay).

/proc/sys/net/ipv4/neigh/DEV/retrans_time

El tiempo, expresado en jiffies (1/100 sec), entre mensajes Neighbor Solicitation retransmitidos. Se usa para resolver direcciones y determinar si se puede alcanzar a un vecino.

/proc/sys/net/ipv4/neigh/DEV/ucast_solicit

Número máximo de reintentos para una solicitud unicast.

/proc/sys/net/ipv4/neigh/DEV/unres_qlen

Tamaño máximo de la cola para una consulta arp pendiente (el número de paquetes que se aceptan de otras capas mientras aún está por resolver una dirección ARP).

Internet QoS: Architectures and Mechanisms for Quality of Service, Zheng Wang, ISBN 1-55860-608-4

Libro en tapas duras que cubre temas relacionados con la Calidad de Servicio. Bueno para comprender los conceptos básicos.

13.2.4. Configuración de encaminamiento

/proc/sys/net/ipv4/route/error_burst

Estos parámetros se usan para limitar los mensajes de aviso escritos desde el código de encaminamiento al registro del núcleo. Cuanto más grande sea error_cost, menos mensajes se escriben. Error_burst controla cuando descartar mensajes. Los valores por defecto limitan los mensajes de aviso a uno cada cinco segundos.

/proc/sys/net/ipv4/route/error_cost

Estos parámetros se usan para limitar los mensajes de aviso escritos desde el código de encaminamiento al registro del núcleo. Cuanto más grande sea error_cost, menos mensajes se escriben. Error_burst controla cuando descartar mensajes. Los valores por defecto limitan los mensajes de aviso a uno cada cinco segundos.

/proc/sys/net/ipv4/route/flush

Escribir en este fichero hace que se vacíe la caché de rutas.

/proc/sys/net/ipv4/route/gc_elasticity

Valores para controlar la frecuencia y comportamiento del algoritmo recolector de basuras de la caché de rutas. Puede ser importante en alta disponibilidad cuando se hace el cambio de máquina. Pasarán al menos gc_timeout segundos antes de que Linux salte a otra ruta porque la anterior ha muerto. Por defecto se pone a 300, pero puede que quiera rebajarla para tener un cambio rápido.

Vea también este mensaje de Ard van Breemen.

/proc/sys/net/ipv4/route/gc_interval

Vea /proc/sys/net/ipv4/route/gc_elasticity.

/proc/sys/net/ipv4/route/gc_min_interval

Vea /proc/sys/net/ipv4/route/gc_elasticity.

/proc/sys/net/ipv4/route/gc_thresh

Vea /proc/sys/net/ipv4/route/gc_elasticity.

/proc/sys/net/ipv4/route/gc_timeout

Vea /proc/sys/net/ipv4/route/gc_elasticity.

/proc/sys/net/ipv4/route/max_delay

Retrasos para vaciar la caché de rutas.

/proc/sys/net/ipv4/route/max_size

Tamaño máximo de la caché de rutas. Las entradas viejas se purgan una vez que la caché alcance este tamaño.

/proc/sys/net/ipv4/route/min_adv_mss

FIXME: rellenar esto

/proc/sys/net/ipv4/route/min_delay

Retrasos al vaciar la caché de rutas.

/proc/sys/net/ipv4/route/min_pmtu

FIXME: rellenar esto

/proc/sys/net/ipv4/route/mtu_expires

FIXME: rellenar esto

/proc/sys/net/ipv4/route/redirect_load

Factores que determinan si se deberían enviar más ICMP Redirect a una máquina específica. No se enviarán más redirect una vez que se haya alcanzado el límite de carga o el número máximo de redirecciones.

/proc/sys/net/ipv4/route/redirect_number

Vea /proc/sys/net/ipv4/route/redirect_load.

/proc/sys/net/ipv4/route/redirect_silence

> Límite de tiempo para los redirect. Tras este periodo se enviará de nuevo un redirect, incluso si se había parado por alcanzar el límite de carga o de envíos.

Capítulo 14. Disciplinas de cola avanzadas y poco conocidas

Si encuentra que tiene necesidades que no cubren las colas mencionadas anteriormente, el núcleo contiene algunas otras colas más especializadas que se mencionan aquí.

14.1. *bfifo/pfifo*

Estas colas sin clases son incluso más sencillas que pfifo_fast ya que no tienen bandas internas (realmente todo el tráfico es igual). Sin embargo tienen un beneficio importante, dan estadísticas. De manera que si no necesita ajustar o priorizar tráfico, puede usar estas colas para determinar cuán llena está la interfaz.

pfifo mide en paquetes, y bfifo en bytes.

14.1.1. Parámetros y uso

limit

> Especifica la longitud de la cola. Se mide en bytes para bfifo, y en paquetes para pfifo. Por defecto es txqueuelen de la interfaz (vea el capítulo sobre pfifo_fast) en paquetes o txqueuelen*mtu bytes para bfifo.

14.2. Algoritmo Clark-Shenker-Zhang (CSZ)

Es tan teórico que ni siquiera Alexey (el autor principal de CBQ) dice entenderlo. De su código fuente:

David D. Clark, Scott Shenker y Lixia Zhang *Soporte para aplicaciones de tiempo real en una red de paquetes de servicios integrados: Arquitectura y mecanismos*.

Hasta donde lo entiendo, la idea principal es crear flujos WFQ para cada servicio garantizado y reservar el resto de ancho de banda para el flujo-0. El flujo-0 comprende los servicios predictivos y el tráfico de mejor esfuerzo; lo controla un organizador de prioridades con la banda de máxima prioridad reservada para los servicios restringidos, y el resto a los paquetes de mejor esfuerzo.

Tenga en cuenta que en CSZ los flujos NO están limitados a su ancho de banda. Se supone que el flujo pasa un control de admisión al borde de la red QoS y no necesita más ajustes. Cualquier intento de mejorar el flujo o de ajustarlo a un tocken bucket en saltos intermedios producirá retrasos indeseados y aumentará las variaciones (jitter).

Por el momento CSZ es el único organizador que proporciona servicio garantizado real. Otros esquemas (incluido CBQ) no proporcionan un retraso garantizado ni variaciones aleatorias."

No parece un buen candidato para ser usado, a menos que haya leído y entendido el artículo mencionado.

14.3. DSMARK

Esteve Camps

<marvin@grn.es>

Este texto es un extracto de mi tesis sobre *Soporte de QoS en Linux*, septiembre de 2000.

Documentos fuente:

- Draft-almesberger-wajhak-diffserv-linux-01.txt.
- Ejemplos en la distribución de iproute2.
- White Paper sobre protoclos y arquitecturas QoS y Preguntas frecuentes sobre QoS IP ambos del *Quality of Service Forum*.

Este capítulo ha sido escrito por Esteve Camps <esteve@hades.udg.es>.

14.3.1. Introducción

Antes de nada, sería una buena idea leer los RFC escritos a este respecto (RFC2474, RFC2475, RFC2597 y RFC2598) que encontrará en el sitio web del Grupo de trabajo IETF DiffServ y el sitio web de Werner Almesberger (ha escrito el código para dar soporte en Linux a los Servicios Diferenciados).

14.3.2. ¿Con qué se relaciona Dsmark?

Dsmark es una disciplina de colas que ofrece las capacidades necesarias para los Servicios Diferenciados (también conocidos como DiffServ o simplemente DS). DiffServ es una de las dos arquitecturas QoS actuales (la otra se llama Servicios Integrados) que se basa en un valor transportado por los paquetes en el campo DS de la cabecera IP.

Una de las primeras soluciones en IP diseñadas a ofrecer algún nivel de QoS fue el campo Type of Service (el byte TOS) de la cabecera IP. Cambiando este valor, podíamos escoger un nivel alto/bajo de transferencia, retraso o fiabilidad. Pero esto no proporcionaba suficiente flexibilidad para las necesidades de los nuevos servicios (como

las aplicaciones en tiempo real, las interactivas y otras). Tras esto, aparecieron nuevas arquitecturas. Una de ellas fue DiffServ, que mantuvo los bits TOS y los renombró como el campo DS.

14.3.3. Principios de los Servicios Diferenciados

Los Servicios Diferenciados están orientados al grupo. Quiero decir, no sabemos nada sobre flujos (éste es el propósito de los Servicios Integrados); sólo sabemos sobre agregaciones de flujos y aplicaremos diferentes comportamientosdependiendo a qué agregación pertenezca un paquete.

Cuando llega un paquete a un nodo exterior (edge node - nodo de entrada a un dominio DiffServ) entrando a un Dominio DiffServ, tendremos que controlar, ajustar o marcar estos paquetes (el marcado se refiere a asignar un valor al campo DS; exactamente igual que con las vacas :-)). Esta será la marca/valor que mirarán los nodos internos de nuestro Dominio DiffServ para determinar qué comportamiento o nivel de QoS aplicar.

Como puede deducir, los Servicios Diferenciados implican un dominio dentro del cual se deberán aplicar todas las reglas de DS. En realidad, puede pensar que voy a clasificar todos los paquetes que entran en mi dominio. Una vez que entran todos estarán sujetos a las normas que mi clasificación dicta y cada nodo que atraviesen aplicará ese nivel de QoS.

En realidad, se pueden aplicar normas propias en los dominios locales, pero se deben tener en cuenta algunos *Acuerdos de nivel de servicio* (Service Level Agreements) cuando se conecta a otros dominios DS.

En este punto, debe tener muchas preguntas. DiffServ es más de lo que he explicado. En realidad, entenderá que no puedo resumir más de 3 RFC en sólo 50 líneas :-).

14.3.4. Trabajar con Dsmark

Como especifica la bibliografía de DiffServ, diferenciamos entre nodos externos (o limítrofes) e internos. Son dos puntos importantes en el camino del tráfico. Ambos tipos realizan una clasificación cuando llega el paquete. Su resultado puede usarse en diferentes lugares a lo largo del proceso de DS antes de que se envié el paquete a la red. Es por esto que el código de diffserv proporciona una estructura llamada sk_buff, que incluye un campo nuevo llamado skb->tc_index donde almacenaremos el resultado de la clasificación inicial que puede usarse en varios momentos del tratamiento DS.

El valor de skb->tc_index lo establecerá inicialmente la qdisc DSMARK, sacándola del campo DS de la cabecera IP de cada paquete recibido. Aparte, el clasificador cls_tcindex leerá todo o parte del valor de skb->tcindex y lo usará para escoger las clases.

Pero antes que nada, echemos un vistazo a la orden de la qdisc DSMAR y sus parámetros:

```
... dsmark indices INDICES [ default_index INDICE_DEFECTO ] [
set_tc_index ]
```

¿Qué significan estos parámetros?

- *indices*: tamaño de la tabla de pares (máscara,valor). El tamaño máximo es 2ˆn, siendo n>=0.
- *default_index*: La entrada por defecto del índice de la tabla si el clasificador no encuentra coincidencia.
- *Set_tc_index*: instruye a la disciplina dsmark para que obtenga el campo DS y lo almacene en skb->tc_index.

Veamos el proceso de DSMARK.

14.3.5. Cómo trabaja SCH_DSMARK.

Esta qdisc seguirá los siguientes pasos:

- Si hemos declarado la opción set_tc_index en la orden de la qdisc, se obtiene el campo DS y se almacena en la variable skb->tc_index.
- Se invoca al clasificador. Será ejecutado y devolverá un ID de clase que será almacenado en la variable skb->tc_index. Si no se encuentra ningún filtro que concuerde, optaremos por la opción default_index como classId a almacenar. Si no se ha declarado ni set_tc_index ni default_index el resultado puede ser impredecible.
- Despues de ser enviado a las qdisc internas donde se puede reutilizar el resultado del filtro, se almacena el classid devuelto por la qdisc interna en skb->tc_index. Usaremos este valor en adelante para indizar una tabla máscara-valor. El resultado final a asignar al paquete será el que resulte de la siguiente operación:

  ```
  Nuevo_Campo_Ds = ( Viejo_Campo_Ds & máscara ) | valor
  ```

- Por tanto, el nuevo valor saldrá de aplicar AND a los valores ds_field y máscara, para después aplicar al resultado un OR con el parámetro valor. Vea el siguiente diagrama para entender todo el proceso:

```
                   skb->ihp->tos
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - >
     |                                                       |     ^
     | -- Si declara set_tc_index, almacenamos el            |     |
<-----Puede cambiar
     |    valor de DS en la variable skb->tc_index           |     |O
el campo DS
     |                                                      A|     |R
   +-|-+      +------+    +---+-+     Qdisc     +-+     +---N|-----|--
--+
   | | |      |filtro|--->|   | |-->  . . .  -->| |     |   D|     |
|
   | | |----->|índice|--->|   | |    Interna    | |---->|    v     |
|
   | | |      | tc   |--->| | | +---------------+ |   ----
>(másc,valor) |
-->| O |      +------+    +-|-+--------------^----+  /  |  (.  ,  .)
|
   | | |          ^         |                |       |  |  (.  ,  .)
|
   | | +----------|---------|----------------|-------|--+  (.  ,  .)
|
   | | sch_dsmark |         |                |       |
|
   +-|------------|---------|----------------|-------|----------------
--+
     |            |         | <- tc_index -> |       |
     |            |(leer)   |  puede cambiar |       |  <-------------
-Indice a la
     |            |         |                |       |
tabla de pares
     v            |         v                v       |
(másc,valor)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - ->
                         skb->tc_index
```

¿Cómo se marca? Cambie la máscara y el valor de la clase que quiere remarcar. Vea la siguiente línea de código:

```
tc class change dev eth0 classid 1:1 dsmark mask 0x3 value 0xb8
```

Esto cambia el par (másc,valor) de la tabla hash, para remarcar paquete que pertenecen a la clase 1:1. Tiene que "cambiar" estos valores por los valores por defecto que obtiene inicialmente (másc,valor). (Vea la tabla más adelante)

Ahora, explicaremos cómo funciona el filtro TC_INDEX y cómo cuadra en todo esto. Además, se puede usar el filtro TC_INDEX en otras configuraciones aparte de las que incluyen servicios DS.

14.3.6. Filtro TC_INDEX

Esta es la orden básica para declarar un filtro TC_INDEX:

```
... tcindex [ hash TAMAÑO ] [ mask MASCARA ] [ shift SHIFT ]
            [ pass_on | fall_through ]
            [ classid CLASSID ] [ police ESPEC_NORMA ]
```

Luego, mostramos unos ejemplos para explicar el modo de operación de TC_INDEx. Preste atención a las palabras resaltadas: tc qdisc add dev eth0 handle 1:0 root dsmark indices 64 *set_tc_index* tc filter add dev eth0 parent 1:0 protocol ip prio 1 tcindex *mask 0xfc shift 2* tc qdisc add dev eth0 parent 1:0 handle 2:0 cbq bandwidth 10Mbit cell 8 avpkt 1000 mpu 64 # class de tráfico EF tc class add dev eth0 parent 2:0 classid 2:1 cbq bandwidth 10Mbit rate 1500Kbit avpkt 1000 prio 1 bounded isolated allot 1514 weight 1 maxburst 10 # qdisc fifo de paquetes para tráfico EF tc qdisc add dev eth0 parent 2:1 pfifo limit 5 tc filter add dev eth0 parent 2:0 protocol ip prio 1 *handle 0x2e* tcindex *classid 2:1 pass_on* (Este código no está completo. Es sólo un extracto del ejemplo EFCBQ incluido en la distribución de iproute2).

Antes de nada, supongamos que recibimos un paquete marcado como EF. Si lee el RFC2598, verá que el valor DSCP recomendado para el tráfico EF es 101110. Esto significa que el campo DS será 101110 (recuerde que los bits menos significativos en el byte TOS no se usan en DS) o 0xb8 en código hexadecimal.

```
              FILTRO
              TC INDEX
   +----+     +-------+    +---+-+    +-------+                +-+
+-------+
   |    |     |       |    |   | |    |MANEJA|  +-+     +-+    | |     |
|
   |    |---->| MASC  | -> |   | | -> |FILTRO|->| |     | | -> | | -> |
|
   |    |  .  | =0xfc |    |   | |    |0x2E  |  | +----+ |    | |     |
|
   |    |  .  |       |    |   | |    +-------+  +--------+    | |     |
|
   |    |  .  |       |    |   | |                             | |     |
|
-->|    |  .  | SHIFT |    |   | |                             | |     |
|-->
   |    |  .  | =2    |    |   | +---------------------------+ |     |
|
   |    |     |       |    |   |       CBQ 2:0                 |     |
|
   |    |     +-------+    +---+-------------------------------+     |
|
   |    |                                                            |
|
   |    +------------------------------------------------------------+
|
   |                            DSMARK 1:0
|
   +-------------------------------------------------------------------
-------+
```

Entonces llega el paquete con el valor del campo DS puesto a 0xb8. Como explicamos anteriormente, la qdisc dsmark identificada por el id 1:0 en el ejemplo toma el campo DS y lo almacena en la variable skb->tc_index. El siguiente paso del ejemplo corresponde al filtro asociado con esta qdisc (segunda línea del ejemplo). Esto realizará las siguientes operaciones:

```
Valor1 = skb->tc_index & MASK
Clave = Valor1 >> SHIFT
```

En el ejemplo, MASC=0xFC i SHIFT=2.

```
Valor1 = 10111000 & 11111100 = 10111000
Clave = 10111000 >> 2 = 00101110 -> 0x2E en hexadecimal
```

El valor de recotno corresponderá a un controlador de filtro de una qdisc interna (en el ejemplo, el identificador 2:0). Si existe un filtro con este id, se verificarán las condiciones de control y medida (en caso de que el filtro las incluya) y se devolverá y almacenará en la variable skb->tc_index el classid (en nuestro ejemplo, classid 2:1).

Pero si se encuentra un filtro con ese identificador, el resultado dependerá de la declaración de fall_through. Si la hay, se devolverá como classid el valor clave. Si no, se devuelve un error y continúa el proceso con el resto de filtros. Sea cuidadoso si usa fall_through; se puede hacer si existe una relación simple entre los valores de la variable skb->tc_index y el id de la clase.

Los últimos parámetros que hay que comentar son hash y pass_on. El primero se refiere al tamaño de la tabla hash. Pass_on se usa para indicar que si no se encuentra un classid igual al resultado de este filtro, se debe intentar con el siguiente filtro. La acción por defecto es fall_through (vea la siguiente tabla)

Por último, veamos qué valores posibles se pueden dar a todos estos parámetros de TCINDEX:

Nombre TC	Valor	Por defecto
Hash	1...0x10000	Dependiente de la implementación
Mask	0...0xffff	0xffff
Shift	0...15	0
Fall through / Pass_on	Flag	Fall_through
Classid	Mayor:menor	Nada
Police	.....	Nada

Este tipo de filtro es muy poderoso. Es necesario explorar todas sus posibilidades. Más aún, este filtro no se usa sólo en las configuraciones con DiffServ. Lo puede usar como cualquier otro tipo de filtro.

Recomiendo que mire todos los ejemplos de DiffServ que se incluyen en la distribución de iproute2. Prometo que trataré de completar este texto lo más pronto que pueda. Además, todo lo que he explicado es el resultado de muchas pruebas. Le agradecería que me dijese si me he equivocado en algún momento.

14.4. Qdisc de entrada (Ingress)

Todas las qdisc comentadas hasta ahora eran de salida. Sin embargo, cada interfaz tiene también una qdisc de entrada que no se usa para enviar paquetes hacia el adaptador de red. En su lugar, permite aplicar filtros de tc a los paquetes que vienen desde la interfaz, independientemente de si su destino es local o han de ser reenviados.

Como los filtros de tc contienen una implementación completa del Token Bucket Filter, y también son capaces de trabajar con el estimador de flujo del núcleo, hay mucha funcionalidad disponible. Esto permite de forma efectiva controlar el tráfico entrante, antes incluso de que entre en la pila IP.

14.4.1. Parámetros y uso

La qdisc de entrada en sí misma no precisa parámetros. Difiere de las otas qdisc en que no ocupa la raíz de un dispositivo. Asóciela de esta manera:

```
# tc qdisc add dev eth0 ingress
```

Esto le permite tener otras qdisc de envío en el dispositivo aparte de la de entrada.

Encontrará un ejemplo contribuido sobre el posible uso de la qdisc de entrada en el Recetario.

14.5. Random Early Detection (RED)

Esta sección pretende ser una introducción al encaminamiento en «backbones» (N. del T.: espinas dorsales, las vías más anchas y rápidas de Internet), que a menudo implica anchos de banda de <100 megabits, lo que precisa de un enfoque diferente que para el módem ADSL de casa.

El comportamiento normal de las colas de router en la Internet se denomina "tail-drop". El tail-drop (descarte de últimos) funciona encolando hasta una cierta cantidad, y descartando después todo el tráfico que "se rebose". Esto es bastante poco justo, y lleva a sincronización de retransmisión. Cuando sucede esta sincronización, la súbita ráfaja de descartes del router que ha llegado a su tope causa una ráfaga retardada de retransmisiones, que vuelve a llenar en exceso el router congestionado.

Para copar con la congestión transitoria de los enlaces, los router de backbone a menudo implementan grandes colas. Desafortunadamente, mientras que estas colas están bien para la transferencia, pueden incrementar sustancialmente la latencia y causar que las conexiones TCP se comporten a ráfagas durante la congestión.

Los problemas de tail-drop están empezando a ser cada vez más acuciantes en Internet debido al uso creciente de aplicaciones poco amigables para las redes. El núcleo de Linux nos ofrece RED, abreviatura de Random Early Detect (detección aleatoria temprana), también denominada Random Early Drop (descarte aleatorio temprano), ya que es así como funciona.

RED no es un remedio milagroso para todo esto, y las aplicaciones que no implementen esperas tras colisión (backoff) exponenciales seguirán llenando una porción demasiado grande del ancho de banda, pero con RED no causan tanto daño a la transferencia y la latencia de otras conexiones.

RED descarta paquetes de los flujos estadísticamente antes de que lleguen a un límite absoluto (hard). Esto hace que un enlace congestionado en un backbone reduzca la marcha de una manera más elegante, y evita la sincronización de retransmisiones.

También ayuda a TCP a encontrar su velocidad "correcta" más rápido permitiendo que algunos paquetes caigan pronto manteniendo bajo el tamaño de las colas y la latencia bajo control. La probabilidad de que se descarte un paquete de una conexión particular es proporcional a su uso de ancho de banda en lugar de al número de paquetes que transmite.

RED es una buena cola para backbones, donde no te puedes permitir la complejidad de supervisar el estado por sesión que se necesita para un encolado correcto.

Para usar RED, debe decidir tres parámetros: Min, Max, y burst. Min establece el tamaño mínimo de la cola en bytes antes de que empiecen los descartes, Max es un máximo preventivo (soft) bajo el que intenta mantenerse el algoritmo, y burst especifica el número máximo de paquetes que pueden "pasar en ráfaga".

Debería establecer min calculando la latencia básica más alta aceptable que desea en las colas, multiplicándola por el ancho de banda. Por ejemplo, en mi enlace RDSI de 64kbit/s, querría una latencia base de cola de 200ms de manera que pongo un mínimo de 1600 bytes. Poner un min demasiado pequeño degradará la transferencia y demasiado grande degradará la latencia. Poner un min pequeño no es un reemplazo de la reducción de la MTU en un enlace lento para mejorar la respuesta interactiva.

Debería poner un max de al menos el doble que min para evitar la sincronización. En enlaces lentos con Min pequeño quizá fuera más sensato hacer max cuatro o cinco veces más grande que min.

Burst controla cómo responde el algoritmo RED a las ráfagas. Burst se debe hacer más grande que min/avpkt. De forma experimental he establecido que (min+min+max)/(3*avpkt) funciona bien.

Además, necesita indicar limit y avpkt. Limit es un valor de seguridad, y tras pasar la cola de limit bytes, RED "se pone en modo" tail-drop. Normalmente pongo el límite en ocho veces max. Avpkt debería ser el tamaño medio de los paquetes. 1000 funciona BIEN en enlaces de Internet de alta velocidad con una MTU de 1500.

Lea el documento sobre colas RED de Sally Floyd y Van Jacobson si de sea información técnica.

14.6. Generic Random Early Detection

No se sabe mucho sobre GRED. Se parece a RED con varias colas internas, siendo escogidas estas colas internas basándose en el campo tcindex de DiffServ. De acuerdo a una transparencia que se encuentra aquí, contiene las capacidades del «Distributed Weighted RED» de Cisco, así como del RIO de Dave Clark.

Cada paquete virtual puede tener especificados sus propios parámetros de descarte (Drop Parameters).

FIXME: hacer que Jamal o Werner nos cuenten más

14.7. Emulación VC/ATM

Este es un gran esfuerzo de Werner Almesberger para permitirle construir Circuitos Virtuales sobre sockets TCP/IP. Un Circuito Virtual es un concepto que viene de la teoría de redes ATM.

Si desea más información, consulte la página weeb de ATM sobre Linux.

14.8. Weighted Round Robin (WRR)

Esta qdisc no se incluye en los núcleos estándar pero la puede descargar de aquí. Actualmente la qdisc sólo está probada con núcleos Linux 2.2 pero probablemente funcionará también con núcleos 2.4/2.5.

La qdisc WRR distribuye ancho de banda entre sus clases usando el esquema de weighted round robin. Osea, que al igual que la qdisc CBQ contiene clases dentro de las cuales se pueden colocar qdisc arbitrarias. Todas las clases que tengan suficiente demanda obtendrán un ancho de banda proporcional a los pesos asociados con las clases. Se puede establecer los pesos de forma manual usando el programa `tc`. Pero también pueden decrementarse de forma automática para clases que transfieran demasiados datos.

La qdisc incluye un clasificador que asigna a diferentes clases los paquetes que vienen o van a diferentes máquinas. Se puede usar tanto la MAC como la IP y las direcciones de origen o destino. Sin embargo, la dirección MAC sólo se puede usar cuando la máquina Linux esté actuando como bridge ethernet. Las clases se van asignando automáticamente a máquinas basándose en los paquetes vistos.

Esta qdisc puede ser muy útil en sitios como una residencia estudiantil donde muchos individuos sin relación alguna comparten una conexión a Internet. La distribución de WRR contiene como uno de sus elementos centrales varios script que configuran un comportamiento relevante a tales sitios.

Capítulo 15. Recetario

Esta sección contiene "recetas" que pueden ayudarle a resolver problemas. Aún así, un recetario no reemplaza al conocimiento, de manera que intente comprender qué es lo que sucede.

15.1. Llevar varios sitios con diferentes SLA

N del T: SLA es Service Level Agreement. Se refiere a clientes que han firmado por tener diferentes servicios.

Esto se puede hacer de varias maneras. Apache tiene soporte para esto en un módulo, pero mostraremos cómo puede hacerse con Linux, y por tanto con otros servicios. Estas órdenes las robamos de una presentación de Jamal Hadi a la que se hace referencia más adelante.

Digamos que tenemos dos clientes, con http, ftp y streaming de sonido, y queremos venderle una cantidad limitada de ancho de banda. Lo haremos en el propio servidor.

El cliente A debería tener al menos 2 megabits, el cliente B ha pagado por 5 megabits. Separamos nuestros clientes creando IP virtuales en nuestro servidor.

```
# ip address add 188.177.166.1 dev eth0
# ip address add 188.177.166.2 dev eth0
```

Es su responsabilidad asociar los diferentes servidores a la dirección IP correcta. Todos los demonios populares soportan esto.

Primero asociamos una qdisc CBQ a eth0:

```
# tc qdisc add dev eth0 root handle 1: cbq bandwidth 10Mbit cell 8
avpkt 1000 \
  mpu 64
```

Entonces creamos clases para nuestros clientes:

```
# tc class add dev eth0 parent 1:0 classid 1:1 cbq bandwidth 10Mbit
rate \
  2MBit avpkt 1000 prio 5 bounded isolated allot 1514 weight 1
maxburst 21
# tc class add dev eth0 parent 1:0 classid 1:2 cbq bandwidth 10Mbit
rate \
  5Mbit avpkt 1000 prio 5 bounded isolated allot 1514 weight 1
maxburst 21
```

Luego añadimos filtros para nuestras dos clases:

```
##FIXME: ¿Por qué esta línea, qué hace?, ¿qué es un divisor?:
```

```
##FIXME: Un divisor tiene algo que ver con una tabla hash, y el número
de
#        buckets - ahu
# tc filter add dev eth0 parent 1:0 protocol ip prio 5 handle 1: u32
divisor 1
# tc filter add dev eth0 parent 1:0 prio 5 u32 match ip src
188.177.166.1
  flowid 1:1
# tc filter add dev eth0 parent 1:0 prio 5 u32 match ip src
188.177.166.2
  flowid 1:2
```

Y ya está.

FIXME: ¿por qué no usar el token bucket filter? ¿hay alguna pfifo_fast por defecto de respaldo en algún lado?

15.2. Proteger la máquina frente a inundaciones SYN

De la documentación de Alexey sobre iproute, adaptado a netfilter y con más vías plausibles. Si usa esto, asegúrese de ajustar los números a valores razonables para su sistema.

Si quiere proteger una red entera, sáltese este script, que se ajusta mejor a una única máquina.

Parece que se necesita la última versión de las herramientas iproute2 para que esto funcione con 2.4.0.

```
#! /bin/sh -x
#
# sample script on using the ingress capabilities
# this script shows how one can rate limit incoming SYNs
# Useful for TCP-SYN attack protection. You can use
# IPchains to have more powerful additions to the SYN (eg
# in addition the subnet)
#
#path to various utilities;
#change to reflect yours.
#
TC=/sbin/tc
IP=/sbin/ip
IPTABLES=/sbin/iptables
INDEV=eth2
#
# tag all incoming SYN packets through $INDEV as mark value 1
############################################################
$iptables -A PREROUTING -i $INDEV -t mangle -p tcp --syn \
  -j MARK --set-mark 1
############################################################
#
# install the ingress qdisc on the ingress interface
############################################################
$TC qdisc add dev $INDEV handle ffff: ingress
```

```
#
#
# SYN packets are 40 bytes (320 bits) so three SYNs equals
# 960 bits (approximately 1kbit); so we rate limit below
# the incoming SYNs to 3/sec (not very useful really; but
#serves to show the point - JHS
############################################################
$TC filter add dev $INDEV parent ffff: protocol ip prio 50 handle 1 fw
\
police rate 1kbit burst 40 mtu 9k drop flowid :1
############################################################


#
echo "---- qdisc parameters Ingress  ----------"
$TC qdisc ls dev $INDEV
echo "---- Class parameters Ingress  ----------"
$TC class ls dev $INDEV
echo "---- filter parameters Ingress ----------"
$TC filter ls dev $INDEV parent ffff:

#deleting the ingress qdisc
#$TC qdisc del $INDEV ingress
```

15.3. Limitar la tasa de ICMP para prevenir dDoS

Recientemente, los ataques distribuidos de negación de servicio se han convertido en una amenaza importante para la Internet. Filtrando y limitando tasas de forma adecuada en nuestra red, podemos evitar tanto ser una víctima como la causa de uno de estos ataques.

Debería filtrar sus redes de manera que no permita abandonarlas a ningún paquete con IP de origen no local. Esto hace que la gente no pueda enviar porquería a la Internet de forma anónima.

La limitación de tasas va como dijimos antes. Para refrescarle la memoria, aquí tiene de nuevo nuestro ASCIIgrama.

```
[La Internet] ---<E3, T3, da igual>--- [router Linux] ---
[Oficina+ISP]
                                      eth1          eth0
```

Primero configuramos los prerequisitos:

```
# tc qdisc add dev eth0 root handle 10: cbq bandwidth 10Mbit avpkt
1000
# tc class add dev eth0 parent 10:0 classid 10:1 cbq bandwidth 10Mbit
rate \
  10Mbit allot 1514 prio 5 maxburst 20 avpkt 1000
```

Si tenemos interfaces de 100Mbit o más, hay que ajustar estos números. Ahora tenemos que determinar cúanto tráfico ICMP deseamos permitir. Se pueden hacer mediciones con tcpdump, haciendo que escriba en un fichero durante un rato, y viendo cuánto ICMP pasa por la red. ¡No olvide aumentar la longitud de la captura!

Si hacer mediciones es poco práctico, quizá quiera escoger un 5% del ancho de banda disponible. Configuremos nuestra clase:

```
# tc class add dev eth0 parent 10:1 classid 10:100 cbq bandwidth
10Mbit rate \
  100Kbit allot 1514 weight 800Kbit prio 5 maxburst 20 avpkt 250 \
  bounded
```

Esto limita a 100Kbit. Ahora necesitamos un filtro para asignarle el tráfico ICMP a esta clase:

```
# tc filter add dev eth0 parent 10:0 protocol ip prio 100 u32 match ip
  protocol 1 0xFF flowid 10:100
```

15.4. Priorizado de tráfico interactivo

Si están saliendo muchos datos por su enlace, o entrando que para el caso es lo mismo, y está intentando hacer mantenimiento vía telnet o ssh, puede que la cosa no vaya muy bien. Los otros paquetes están bloqueando nuestras pulsaciones. ¿No sería maravilloso si hubiera una manera de que nuestros paquetes interactivois se colasen por entre el tráfico masivo? ¡Linux puede hacerlo!

Como antes, necesitamos gestionar el tráfico que va en ambas direcciones. Evidentemente, esto funciona mejor si hay máquinas Linux a ambos lados del enlace, aunque otros UNIX son capaces de hacerlo. Consulte a su gurú local de Solaris/BSD al respecto.

El organizador pfifo_fast estándar tiene 3 "bandas" diferentes. El tráfico del a banda 0 se transmite antes, tras lo cual se toma en consideración el tráfico de las bandas 1 y 2. ¡Es vital que nuestro tráfico interactivo vaya a la banda 0!

Hemos adaptado esto desvergonzadamente directamente del ipchains HOWTO (que pronto estará obsoleto):

Hay cuatro bits en la cabecera IP que se usan rara vez, llamados los bits de Tipo de Servicio (TOS). Afectan la manera en que se trata un paquete; los cuatro bits son "Retraso Mínimo", "Transferencia Máxima", "Fiabilidad Máxima" y "Coste Mínimo". Sólo se permite activar uno de estos bits. Rob van Nieuwkerk, el autor del código de ipchains que trabaja con el TOS, lo dice así:

```
El "Mínimum Delay" es especialmente importante para mí. Lo activo para
paquetes "interactivos" en mi router (Linux) de salida. Estoy tras un
enlace por módem a 33k6. Linux prioriza los paquetes en tres colas. De
esta manera, consigo un rendimiento interactivo aceptable mientras
realizo
descargas masivas al mismo tiempo.
```

El uso más común es marcar las conexiones de telnet y control de ftp con "Retraso Mínimo", y la de datos de FTP a "Transferencia Máxima". Esto debería hacerse de la siguiente manera, en el router de salida:

```
# iptables -A PREROUTING -t mangle -p tcp --sport telnet \
  -j TOS --set-tos Minimize-Delay
# iptables -A PREROUTING -t mangle -p tcp --sport ftp \
  -j TOS --set-tos Minimize-Delay
# iptables -A PREROUTING -t mangle -p tcp --sport ftp-data \
  -j TOS --set-tos Maximize-Throughput
```

Lo hecho hasta ahora sólo sirve para los datos que vienen de la máquina remota hacia la suya. Para los que van en el otro sentido parece que ya está hecho, ya que telnet, ssh y amigos activan el campo TOS automáticamente en los paquetes que salen.

Si tuviera una aplicación que no lo hace, siempre puede controlarlo con netfilter. En la máquina local:

```
# iptables -A OUTPUT -t mangle -p tcp --dport telnet \
  -j TOS --set-tos Minimize-Delay
# iptables -A OUTPUT -t mangle -p tcp --dport ftp \
  -j TOS --set-tos Minimize-Delay
# iptables -A OUTPUT -t mangle -p tcp --dport ftp-data \
  -j TOS --set-tos Maximize-Throughput
```

15.5. Caché transparente de web usando netfilter, iproute2, ipchains y squid

Esta sección nos la envió el lector Ram Narula de Internet para la Educación (Tailandia).

La técnica habitual para conseguir esto en Linux probablemente es usar ipchains DESPUES de asegurarse que el tráfico de "salida" del puerto 80(web) se dirige al servidor que ejecuta el squid.

Hay 3 métodos comunes para asegurarse de que el tráfico "saliente" del puerto 80 se envía al servidor que ejecuta el squid, y aquí vamos a introducir una cuarta.

Que el router pasarela lo haga por tí.

> Si puede decirle a su pasarela que haga que los paquetes que salen hacia el puerto 80 sean enviados hacia la dirección IP del servidor squid.
>
> PERO

Esto añadiría una carga adicional al router y algunos router comerciales podrían no soportarlo siquiera.

Usar un switch de capa 4.

Los switchs de capa 4 pueden gestionar esto sin problemas.

PERO

El coste de este equipamiento suele ser muy alto. Un switch de capa 4 típico normalmente costará más que el router típico + un buen servidor linux.

Usar un servidor caché como pasarela de la red.

Puede forzar que TODO el tráfico pase por el servidor caché.

PERO

Esto es muy arriesgado porque Squid utiliza gran cantidad de CPU y podría acabar ralentizando el rendimiento de la red entera o que el propio servidor se caiga, y nadie de la red podrá acceder a Internet si esto ocurre.

Linux+router NetFilter.

Con NetFilter se puede implementar otra técnica que usa NetFilter para "marcar" los paquetes con puerto de destino 80 e iproute2 para encaminar los paquetes marcados al servidor Squid.

```
|----------------|
| Implementación |
|----------------|

 Direcciones usadas
 10.0.0.1 naret (Servidor NetFilter)
 10.0.0.2 silom (Servidor Squid)
 10.0.0.3 donmuang (Router conectado a Internet)
 10.0.0.4 kaosarn (otro servidor de la red)
 10.0.0.5 RAS
 10.0.0.0/24 red principal
 10.0.0.0/19 red total

|---------------|
|Diagrama de red|
|---------------|

Internet
|
donmuang
|
------------hub/switch----------
|        |            |       |
naret   silom        kaosarn  RAS etc.
```

Primero, hagamos pasar todo el tráfico a través de naret asegurándonos de que es la pasarela por defecto excepto para silom. La pasarela por defecto de silom será donmuang (10.0.0.3) o crearíamos un bucle de tráfico.

(todos los servidores de mi red tienen como pasarela por defecto a 10.0.0.1, que era la anterior dirección IP del router donmuang, así que lo que he hecho es cambiar la dirección IP de donmuang a 10.0.0.3 y darle a naret la dirección ip 10.0.0.1)

```
Silom
-----
-configurar squid e ipchains
```

Configuramos el servidor Squid de silom, y nos aseguramos de que soporte proxy/caché transparente; el puerto por defecto suele ser 3218, de manera que se tiene que redirigir localmente todo el tráfico hacia el puerto 80 al puerto 3128. Esto se puede hacer con ipchains con lo siguiente:

```
silom# ipchains -N allow1
silom# ipchains -A allow1 -p TCP -s 10.0.0.0/19 -d 0/0 80 -j REDIRECT
3128
silom# ipchains -I input -j allow1
```

O, en el idioma de netfilter:

```
silom# iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 80 -j
REDIRECT --to-port 3128
```

(nota: podría tener otras entradas también)

Si desea más información sobre configuración del servidor Squid, por favor, remítase a la página de las FAQ de Squid en http://squid.nlanr.net).

Nos aseguramos de que este servidor tiene activo el reenvío de ip (forwarding) y de que su pasarela por defecto es el router donmuang (NO naret).

```
Naret
-----
-configurar iptables e iproute2
-desactivar los mensajes icmp REDIRECT (si es necesario)
```

1. "Marca" con el valor 2 los paquetes con destino al puerto 80
2. `naret# iptables -A PREROUTING -i eth0 -t mangle -p tcp --dport 80 \`
3. `-j MARK --set-mark 2`

4. Configuramos iproute2 para que envíe los paquetes con la marca 2 a silom
5. `naret# echo 202 www.out >> /etc/iproute2/rt_tables`

```
6.   naret# ip rule add fwmark 2 table www.out
7.   naret# ip route add default via 10.0.0.2 dev eth0 table www.out
8.   naret# ip route flush cache
```

Si donmuang y naret están en la misma subred entonces naret no debería enviar mensajes icmp REDIRECT. En este caso, deberíamos desactivar los icmp REDIRECT así:

```
naret# echo 0 > /proc/sys/net/ipv4/conf/all/send_redirects
naret# echo 0 > /proc/sys/net/ipv4/conf/default/send_redirects
naret# echo 0 > /proc/sys/net/ipv4/conf/eth0/send_redirects
```

Hemos terminado el montaje, comprobemos la configuración

```
En naret:

naret# iptables -t mangle -L
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
MARK       tcp  --  anywhere             anywhere           tcp
dpt:www MARK set 0x2

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination

naret# ip rule ls
0:      from all lookup local
32765:  from all fwmark        2 lookup www.out
32766:  from all lookup main
32767:  from all lookup default

naret# ip route list table www.out
default via 203.114.224.8 dev eth0

naret# ip route
10.0.0.1 dev eth0  scope link
10.0.0.0/24 dev eth0  proto kernel  scope link  src 10.0.0.1
127.0.0.0/8 dev lo  scope link
default via 10.0.0.3 dev eth0

(hay que asegurarse de que silom aparece en una de las líneas
anteriores,
en este caso es la línea con 10.0.0.0/24)

|-------|
|-HECHO-|
|-------|
```

15.5.1. Diagrama de flujo del tráfico tras la implementación

```
|-----------------------------------------------------|
|Diagrama de flujo del tráfico tras la implementación|
|-----------------------------------------------------|

INTERNET
/\
||
\/
-----------------router donmuang---------------------
/\                                /\         ||
||                                ||         ||
||                                \/         ||
naret                            silom       ||
*tráfico hacia el puerto 80==========>(caché)      ||
/\                                ||         ||
||                                \/         \/
\\==================================kaosarn, RAS, etc.
```

Tengan en cuenta que la red es asimétrica ya que hay un salto extra en la ruta general de salida.

```
Este es el camino seguido por los paquetes que atraviesan la red desde
kaosarn hacia y desde Internet.

Tráfico web/http:
consulta http de kaosarn->naret->silom->donmuang->internet
respuestas http desde Internet->donmuang->silom->kaosarn

consultas no web/http (ej: telnet):
datos salientes de kaosarn->naret->donmuang->internet
datos entrantes de Internet->donmuang->kaosarn
```

15.6. Sortear los problemas de Path MTU Discovery con configuraciones de MTU por ruta

Generalmente, Internet funciona mejor enviando tráfico masivo si se usan paquetes grandes. Cada paquete implica una decisión de encaminamiento, y cuando se envía un fichero de 1 megabyte, podemos estar hablando de alrededor de 700 paquetes si se envían del mayor tamaño posible, o 4000 si se usa el valor pequeño por defecto.

Sin embargo, no todas las partes de Internet soportan 1460 bytes de carga por paquete. Por tanto, se hace necesario probar hasta encontrar el mayor paquete que "encaje", para optimizar una conexión.

Este proceso se denomina «Descubrimiento de MTU de la ruta» (Path MTU Discovery), siendo MTU la «Maximum Transfer Unit».

Cuando un router encuentra un paquete que es demasiado grande para enviarlo de una sola pieza, Y que está marcado con el bit «Don't Fragment», devuelve un mensaje ICMP que indica que se vio obligado a descartar un paquete por esta causa. La máquina que lo envió reacciona a esto enviando paquetes más pequeños, y repitiendo puede encontrar el tamaño de paquete óptimo para una conexión sobre determinada ruta.

Esto funcionó bien hasta que los hooligans que se dedican a interferir en las comunicaciones descubrieron la Internet. Esto hizo que los administradores bloqueasen o ajustasen el tráfico ICMP en un intento equivocado de mejorar la seguridad o robustez de su servicio de Internet.

Lo que ocurre ahora es que el Path MTU Discovery funciona cada vez peor y falla en determinadas rutas, lo que lleva a sesiones TCP/IP extrañas que mueren al cabo de un tiempo.

Aunque no tengo pruebas de esto, dos sitios con los que solía tener este problema tenían Alteon Acedirectors delante de los sistemas afectados (quizá alguien con más conocimientos pueda darme pistas de por qué sucede esto).

15.6.1. Solución

Cuando encuentre sitios que sufren este problema, puede desactivar el Path MTU discovery configurándolo manualmente. Koos van den Hout, con pequeños cambios, escribe:

El siguiente problema: configuro la mtu/mru de mi línea dedicada con ppp a 296 porque sólo es una 33k6 y no puedo influenciar el encolado en el otro extremo. A 296, la respuesta a una pulsación de tecla está dentro de un margen razonable.

Y por mi lado tengo un masqrouter con Linux (por supuesto).

Recientemente, separé el "servidor" y el "router", de manera que la mayoría de mis aplicaciones funcionan en una máquina diferente a la que hace el encaminamiento.

Entonces tuve problemas al entrar en irc. ¡Pánico! Escarbando un poco averigué que sí conectaba a irc, e incluso aparecía como conectado allí, pero no recibía el motd. Busqué lo que podía estar mal y me acordé que ya había tenido problemas relacionados con la MTU para acceder a ciertos sitios web, ya que no tenía problemas para acceder cuando ponía la MTU a 1500, y el problema sólo aparecía cunado la ponía a 296. Como los servidores de irc bloquean casi cualquier tipo de tráfico que no sea necesario para su operación inmediata, también bloquean icmp.

Conseguí convencer a los operadores de un servidor web de que ésta era la causa del problema, pero los operadores del servidor de irc no iban arreglarlo.

De manera que tenía que asegurarme de que el tráfico enmascarado empezaba con la mtu más baja del enlace externo. Pero quería que mi tráfico ethernet local tuviera una mtu normal (para cosas como el tráfico nfs).

Solucion:

```
ip route add default via 10.0.0.1 mtu 296
```

(siendo 10.0.0.1 la pasarela por defecto, la dirección interna del router enmascarador)

En general, es posible saltarse el PMTU Discovery configurando rutas específicas. Por ejemplo, si sólo da problemas cierta subred, esto podría ayudar:

```
ip route add 195.96.96.0/24 via 10.0.0.1 mtu 1000
```

15.7. Sortear los problemas de Path MTU Discovery con MSS Clamping (para usuarios de ADSL, cable, PPPoE y PPtP)

Como explicamos anteriormente, el Path MTU Discovery ya no funciona bien. Si sabe de hecho que algún salto en algún lado de su red tiene una MTU limitada (<1500), no puede depender de PMTU Discovery para encontrarlo.

Aparte de la MTU, hay otra manera más de establecer el tamaño máximo de un paquete, denominada Tamaño Máximo de Segmento (Maximum Segment Size.). Este es un campo en la parte de Opciones TCP de un paquete SYN.

Los núcleos Linux recientes, y unos pocos controladores PPPoE (en especial el excelente de Roaring Penguin), permiten la posibilidad de «fijar el MSS».

Lo bueno de esto es que estableciendo el valor del MSS, le está diciendo inequívocamente al otro extremo «nunca intentes siquiera mandarme paquetes más grandes que este valor». No se necesita tráfico ICMP para que esto funcione.

Lo malo es que obviamente es un "hack" (rompe el "end to end" modificando los paquetes). Dicho esto, usamos este truco en varios sitios, y funciona de maravilla.

Para que esto funcione necesitaremos al menos iptables-1.2.1a y Linux 2.4.3 o mayor. La línea de órdenes es:

```
# iptables -A FORWARD -p tcp --tcp-flags SYN,RST SYN -j TCPMSS  --
clamp-mss-to-pmtu
```

Esto calcula el MSS adecuado para su enlace. Si nos sentimos valientes, o pensamos que tenemos más idea, podemos hacer algo como esto:

```
# iptables -A FORWARD -p tcp --tcp-flags SYN,RST SYN -j TCPMSS --set-
mss 128
```

Esto establece el MSS de los paquetes SYN que pasen a 128. Uselo si tiene VoIP con paquetes pequeños, y grandes paquetes de http están provocando que sus llamadas se entrecorten.

15.8. El acondicionador de tráfico definitivo: baja latencia, envíos y descargas rápidos

Nota: este script ha sido actualizado recientemente ¡y anteriormente sólo funcionaba con los clientes Linux de su red! De manera que quizá quiera actualizarlo, si tiene máquinas Windows o Mac en la red, y ha comprobado que no son capaces de descargar más rápido mientras los otros están enviando datos.

He intentado crear el santo grial:

Mantener una latencia baja todo el tiempo para el tráfico interactivo

> Esto significa que descargar o enviar ficheros no debería molestar a SSH o incluso a telnet. Estas son las cosas más importantes, y es lentísimo trabajar incluso con una latencia de 200ms.

Permitir la "navegación" a velocidades razonables mientras se envía o descarga información

> Incluso pensando que http es tráfico "masivo", el resto de tráfico no debería ahogarlo demasiado.

Asegurarse de que los envíos no dañan las descargas, y viceversa

> Se observa a menudo el fenómeno de que el tráfico de subida destruye la velocidad de descarga.

Resulta que todo esto es posible, al coste de una pequeña cantidad de ancho de banda. La razón de que los envíos, descargas y ssh se maten entre ellos es la presencia de grandes colas en muchos dispositivos domésticos de acceso como los módem de cable y DSL.

La siguiente sección explica en profundidad lo que causa los retrasos, y cómo podemos corregirlo. Puede saltárselo tranquilamente y pasar directamente al script si no le importa cómo se realiza esta magia.

15.8.1. Por qué no funciona bien por defecto

Los ISP saben que se les compara sólamente según lo rápido que puede descargar la gente. Aparte del ancho de banda disponible, la velocidad de descarga viene muy influenciada por la pérdida de paquetes, que obstaculiza seriamente el rendimiento de

TCP/IP. Las colas grandes pueden ayudar a evitar la pérdida de paquetes, y acelera las descargas. De manera que los ISP configuran colas grandes.

Sin embargo, estas grandes colas dañan la interactividad. Una pulsación tiene que pasar primero por la cola de envío, lo que puede tardar varios segundos (!) y llegar a la máquina remota. Entonces se muestra, lo que genera un paquete de vuelta, que debe atravesar otra cola, situada en el ISP, antes de aparecer en la pantalla.

Esta Cómo le explica cómo cambiar y procesar la cola de varias maneras, pero tristemente, no todas estas colas están a nuestro alcance. La cola del ISP está completamente fuera de nuestros límites, mientras que la cola de envío probablemente reside en el cable módem o en el dispositivo DSL. Puede ser o no que haya posibilidad de configurarla. Lo más probable es que no.

De manera que, ¿qué sigue? Ya que no podemos controlar ninguna de esas colas, debemos eliminarlas, y trasladarlas a nuestro router Linux. Por suerte, esto es posible.

Limitar la velocidad de envío

> Limitando nuestra velocidad de envío a poco menos de la verdadera tasa disponible, no creamos colas en nuestro módem. La cola se ha trasladado al Linux.

Limitar la velocidad de descarga

> Esto tiene algo más de truco, ya que no podemos influenciar realmente lo rápido que internet nos entrega los datos. Sin embargo, podemos descartar paquetes que vengan demasiado rápido, lo que hace que TCP/IP frene justo hasta la tasa que deseamos. Como no deseamos descartar tráfico innecesariamente, configuraremos el tamaño de la "ráfaga" que permitiremos a altas velocidades.

Ahora, una vez hecho esto, hemos eliminado totalmente la cola de descarga (exceptuando pequeñas ráfagas), y ganado la capacidad de gestionar la cola de envío con todo el potencial que ofrece Linux.

Lo que queda por hacer es asegurarnos de que el tráfico interactivo pase a primera línea de la cola de subida. Para asegurarnos de que los envíos no obstaculizan las descargas, también pasaremos los paquetes ACK a primera línea. Esto es lo que normalmente produce la gran reducción de velocidad cuando se genera tráfico masivo en ambas direcciones. Las confirmaciones (ACKnowledgements) del tráfico de descarga deben competir con el tráfico de envío, y quedan retrasados en el proceso.

Si hacemos todo esto, obtenemos las siguientes mediciones usando una conexión ADSL excelente de xs4all en Holanda:

```
Latencia de base:
round-trip min/avg/max = 14.4/17.1/21.7 ms

Sin condicionador de tráfico, mientras descargamos:
round-trip min/avg/max = 560.9/573.6/586.4 ms
```

```
Sin condicionador de tráfico, mientras enviamos:
round-trip min/avg/max = 2041.4/2332.1/2427.6 ms

Con el condicionador, durante un envío a 220kbit/s:
round-trip min/avg/max = 15.7/51.8/79.9 ms

Con el condicionador, durante una descarga a 850kbit/s:
round-trip min/avg/max = 20.4/46.9/74.0 ms

Al enviar, las descargas se producen aproximadamente al 80% de la
velocidad disponible. Los envíos alrededor de al 90%. La latencia
salta
entonces a 850ms, y aún intentamos averiguar por qué.
```

Lo que pueda esperar de este script depende mucho de la velocidad real del enlace de salida. Cuando se envía a toda velocidad, siempre habrá un único paquete por delante de sus pulsaciones. Este es el límite inferior de latencia que puede alcanzar (divida el MTU por la velocidad de subida para calcularlo). Los valores normales serán algo más altos que esto. ¡Reduzca la MTU para obtener mejores efectos!

Ahora, dos versiones del script, una con el excelente HTB de Devik, y la otra con el CBQ que viene en cada núcleo de Linux, al contrario que HTB. Hemos probado ambos, y funcionan bien.

15.8.2. El script (CBQ)

Funciona en todos los núcleos. Dentro de la qdisk CBQ colocamos dos Stochastic Fairness Queues que se aseguran de que varios flujos masivos no se ahoguen uno a otro.

El tráfico de descarga se ajusta usando un filtro tc que contenga un Token Bucket Filter.

Puede mejorar este script añadiendo «bounded» a la línea que comienza con «tc class add .. classid 1:20». Si disminuyó la MTU, ¡reduzca también los números allot y avpkt!

```
#!/bin/bash

# The Ultimate Setup For Your Internet Connection At Home
#
#
# Set the following values to somewhat less than your actual download
# and uplink speed. In kilobits
DOWNLINK=800
UPLINK=220
DEV=ppp0

# clean existing down- and uplink qdiscs, hide errors
tc qdisc del dev $DEV root    2> /dev/null > /dev/null
tc qdisc del dev $DEV ingress 2> /dev/null > /dev/null

###### uplink

# install root CBQ

tc qdisc add dev $DEV root handle 1: cbq avpkt 1000 bandwidth 10mbit
```

```
# shape everything at $UPLINK speed - this prevents huge queues in
your
# DSL modem which destroy latency:
# main class

tc class add dev $DEV parent 1: classid 1:1 cbq rate ${UPLINK}kbit \
allot 1500 prio 5 bounded isolated

# high prio class 1:10:

tc class add dev $DEV parent 1:1 classid 1:10 cbq rate ${UPLINK}kbit \
   allot 1600 prio 1 avpkt 1000

# bulk and default class 1:20 - gets slightly less traffic,
#  and a lower priority:

tc class add dev $DEV parent 1:1 classid 1:20 cbq rate
$[9*$UPLINK/10]kbit \
   allot 1600 prio 2 avpkt 1000

# both get Stochastic Fairness:
tc qdisc add dev $DEV parent 1:10 handle 10: sfq perturb 10
tc qdisc add dev $DEV parent 1:20 handle 20: sfq perturb 10

# start filters
# TOS Minimum Delay (ssh, NOT scp) in 1:10:
tc filter add dev $DEV parent 1:0 protocol ip prio 10 u32 \
      match ip tos 0x10 0xff  flowid 1:10

# ICMP (ip protocol 1) in the interactive class 1:10 so we
# can do measurements & impress our friends:
tc filter add dev $DEV parent 1:0 protocol ip prio 11 u32 \
        match ip protocol 1 0xff flowid 1:10

# To speed up downloads while an upload is going on, put ACK packets
in
# the interactive class:

tc filter add dev $DEV parent 1: protocol ip prio 12 u32 \
   match ip protocol 6 0xff \
   match u8 0x05 0x0f at 0 \
   match u16 0x0000 0xffc0 at 2 \
   match u8 0x10 0xff at 33 \
   flowid 1:10

# rest is 'non-interactive' ie 'bulk' and ends up in 1:20

tc filter add dev $DEV parent 1: protocol ip prio 13 u32 \
   match ip dst 0.0.0.0/0 flowid 1:20

########## downlink #############
# slow downloads down to somewhat less than the real speed  to prevent
# queuing at our ISP. Tune to see how high you can set it.
# ISPs tend to have *huge* queues to make sure big downloads are fast
#
# attach ingress policer:

tc qdisc add dev $DEV handle ffff: ingress

# filter *everything* to it (0.0.0.0/0), drop everything that's
# coming in too fast:
```

```
tc filter add dev $DEV parent ffff: protocol ip prio 50 u32 match ip
src \
   0.0.0.0/0 police rate ${DOWNLINK}kbit burst 10k drop flowid :1
```

Si desea que ppp ejecute este script al conectar, cópielo en /etc/ppp/ip-up.d.

Si las últimas dos líneas dan error, ¡actualice la herramienta tc a una versión más moderna!

15.8.3. El script (HTB)

El siguiente script alcanza todas nuestras metas usando la maravillosa cola HTB (vea el capítulo relevante). ¡Vale la pena parchear el núcleo!

```
#!/bin/bash

# The Ultimate Setup For Your Internet Connection At Home
#
#
# Set the following values to somewhat less than your actual download
# and uplink speed. In kilobits
DOWNLINK=800
UPLINK=220
DEV=ppp0

# clean existing down- and uplink qdiscs, hide errors
tc qdisc del dev $DEV root    2> /dev/null > /dev/null
tc qdisc del dev $DEV ingress 2> /dev/null > /dev/null

###### uplink

# install root HTB, point default traffic to 1:20:

tc qdisc add dev $DEV root handle 1: htb default 20

# shape everything at $UPLINK speed - this prevents huge queues in
your
# DSL modem which destroy latency:

tc class add dev $DEV parent 1: classid 1:1 htb rate ${UPLINK}kbit
burst 6k

# high prio class 1:10:

tc class add dev $DEV parent 1:1 classid 1:10 htb rate ${UPLINK}kbit \
   burst 6k prio 1

# bulk & default class 1:20 - gets slightly less traffic,
# and a lower priority:

tc class add dev $DEV parent 1:1 classid 1:20 htb rate
$[9*$UPLINK/10]kbit \
   burst 6k prio 2

# both get Stochastic Fairness:
tc qdisc add dev $DEV parent 1:10 handle 10: sfq perturb 10
tc qdisc add dev $DEV parent 1:20 handle 20: sfq perturb 10

# TOS Minimum Delay (ssh, NOT scp) in 1:10:
```

```
tc filter add dev $DEV parent 1:0 protocol ip prio 10 u32 \
      match ip tos 0x10 0xff  flowid 1:10

# ICMP (ip protocol 1) in the interactive class 1:10 so we
# can do measurements & impress our friends:
tc filter add dev $DEV parent 1:0 protocol ip prio 10 u32 \
        match ip protocol 1 0xff flowid 1:10

# To speed up downloads while an upload is going on, put ACK packets
in
# the interactive class:

tc filter add dev $DEV parent 1: protocol ip prio 10 u32 \
   match ip protocol 6 0xff \
   match u8 0x05 0x0f at 0 \
   match u16 0x0000 0xffc0 at 2 \
   match u8 0x10 0xff at 33 \
   flowid 1:10

# rest is 'non-interactive' ie 'bulk' and ends up in 1:20


########## downlink #############
# slow downloads down to somewhat less than the real speed  to prevent
# queuing at our ISP. Tune to see how high you can set it.
# ISPs tend to have *huge* queues to make sure big downloads are fast
#
# attach ingress policer:

tc qdisc add dev $DEV handle ffff: ingress

# filter *everything* to it (0.0.0.0/0), drop everything that's
# coming in too fast:

tc filter add dev $DEV parent ffff: protocol ip prio 50 u32 match ip
src \
   0.0.0.0/0 police rate ${DOWNLINK}kbit burst 10k drop flowid :1
```

Si desea que ppp ejecute este script al conectar, cópielo en /etc/ppp/ip-up.d.

Si las últimas dos líneas dan error, ¡actualice la herramienta tc a una versión más moderna!

15.9. Limitar la tasa a una única máquina o máscara de red

Aunque esto se describe con muchísimo detalle en algún otro lado en las páginas de manual, a menudo nos hacen esta pregunta y por suerte hay una respuesta sencilla que no necesita de una comprensión profunda del control de tráfico.

Este script de tres líneas hace el truco:

```
          tc qdisc add dev $DEV root handle 1: cbq avpkt 1000 bandwidth
10mbit
```

```
        tc class add dev $DEV parent 1: classid 1:1 cbq rate 512kbit
\
        allot 1500 prio 5 bounded isolated

        tc filter add dev $DEV parent 1: protocol ip prio 16 u32 \
        match ip dst 195.96.96.97 flowid 1:1
```

La primera línea instala una clase basada en la cola de la interfaz, y le dice al núcleo que debe asumir para los cálculos que la interfaz es de 10mbit. Si se equivoca con esto, no pasa nada malo. Pero dejarlo correctamente hará todo más preciso.

La segunda línea crea una clase de 512kbit con algunos valores por defecto razonables. Si desea detalles, lea las páginas de manual de cbq y Capítulo 9.

La última línea indica qué tráfico debería ir a la clase ajustada. El tráfico que no sea captado por esta regla NO será ajustado. Si quiere captar cosas más complicadas (subredes, puertos de origen o destino), lea Sección 9.6.2.

Si cambia cualquier cosa y quiere reiniciar el script, ejecuta "tc qdisc del dev $DEV root" para borrar la configuración existente.

Se puede mejorar el script añadiendo la última línea opcional "tc qdisc add dev $DEV parent 1:1 sfq perturb 10". Vea Sección 9.2.3 si desea detalles sobre la forma de hacerlo.

Capítulo 16. Hacer bridges y pseudo-bridges con Proxy ARP

Los bridges son dispositivos que se pueden instalar en una red sin tener que reconfigurar. Un switch de red es básicamente un bridge de muchos puertos. Un bridge a menudo es un switch de 2 puertos. Linux, sin embargo, soporta varias interfaces en un bridge, lo que le convierte en un verdadero switch.

A menudo se usan bridges cuando nos encontramos con una red mal configurada que debe ser arreglada sin alteraciones. Como un bridge es un dispositivo de capa 2, una capa por debajo de IP, los router y los servidores no se percatan de su existencia. Esto significa que puede bloquear o modificar ciertos paquetes de una manera transparente, o hacer ajustes.

Otra cosa buena es que a menudo se puede reemplazar un bridge con un cable cruzado o un hub, en caso de que se estropee.

La mala noticia es que un bridge puede causar una gran confusión a menos que esté muy bien documentado. No aparece en las trazas de ruta, pero algunos paquetes pueden desaparecer o cambiar entre los puntos A y B («¡esta red está EMBRUJADA!»). También debería preguntarse si una organización que «no quiere cambiar nada» está haciendo lo correcto.

El bridge de Linux 2.4/2.5 está documentado en esta página.

16.1. Estado del bridging e iptables

Hasta Linux 2.4.20, el bridging e iptables no se "ven" uno a otro sin ayuda. Si se puentean los paquetes de eth0 a eth1, no "pasan" por iptables. Esto significa que no podrá filtrarlos, hacer NAT, modificarlos, o lo que sea. En Linux 2.5.45 y superiores, esto está corregido.

Puede que haya visto mencionar "ebtables", que es otro proyecto más (puede hacer cosas realmente bestiales con MACNAT y "brouting"). Da mucho miedo.

16.2. Bridging y ajustes (shaping)

Funciona tal como sugiere. Asegúrese de averiguar en qué lado está cada interfaz, o puede acabar haciendo ajustes al tráfico saliente en la interfaz interna, que no es lo que se quiere. Use tcpdump en caso necesario.

16.3. Pseudo-bridges con Proxy-ARP

Si simplemente quiere implementar un Pseudo-bridge, sáltese un par de secciones hasta "Implementándolo", pero es interesante leer un poco sobre la manera en que funciona en la práctica.

Un Pseudo-bridge funciona de manera ligeramente diferente. Por defecto, un bridge pasa los paquetes inalterados de una interfaz a la otra. Sólo comprueba las direcciones hardware de los paquetes para determinar a dónde va qué. Esto a su vez significa que se puede hacer bridge sobre tráfico que Linux no entienda, siempre y cuándo tenga una dirección hardware que sí comprenda.

Un "Pseudo-bridge" funciona ligeramente diferente y es más un router oculto que un bridge, pero al igual que un bridge, tiene poco impacto en el diseño de la red.

Una ventaja de que no sea un bridge recae en el hecho de que los paquetes pasan realmente por el kernel, y se pueden filtrar, cambiar, redirigir y reencaminar.

Un bridge real también puede realizar estas tareas, pero necesita código especial como el Ethernet Frame Diverter, o el parche mencionado anteriormente.

Otra ventaja de un pseudo-bridge es que no pasa paquetes que no entienda (limpiando la red de esta manera de un montón de porquería). En casos en que necesite esta porquería (como paquetes SAP o Netbeui), use un bridge de verdad.

16.3.1. ARP y Proxy-ARP

Cuando una máquina desea hablar con otra en la mismo segmento físico de red, envía un paquete del Address Resolution Protocol, que, simplificando, es algo así como «quien tenga 10.0.0.1, que se lo diga a 10.0.0.7». En respuesta a esto, 10.0.0.1 responde con un breve paquete «aquí».

10.0.0.7 envía entonces paquetes a la dirección de hardware mencionada en el paquete «aquí». Hace caché de esta dirección hardware durante un tiempo relativamente largo, y tras que expire, vuelve a lanzar la pregunta.

Cuando creamos un Pseudo-bridge, le indicamos que conteste a estos paquetes ARP, lo que hace que las máquinas de la red envíen sus paquetes al bridge. Entonces el bridge procesa los paquetes, y los envía a la interfaz relevante.

De manera que, en breve, siempre que una máquina a un lado del bridge pregunta por la dirección de una máquina en el otro lado, el bridge responde con un paquete que dice «envíamelo a mí».

De esta manera, todo el tráfico se transmite al sitio correcto, y siempre pasa por el bridge.

16.3.2. Implementándolo

En tiempos más oscuros, se podía instruir al núcleo de Linux para realizar "proxy-ARP" para cualquier subred. De manera que, para configurar un pseudo-bridge, tendría tanto que especificar las rutas correctas hacia ambos lados del bridge COMO crear reglas proxy-ARP correspondientes. Esto es malo porque precisa escribir mucho, pero también porque le permite cometer con facilidad errores que harán que el bridge responda a consultas ARP para redes a las que no sabe cómo llegar.

Con Linux 2.4/2.5 (y posiblemente 2.2), se ha retirado esta posibilidad y se sustituye con una marca en el directorio /proc, denominada "proxy_arp". Por tanto, el procedimiento para construir un pseudo-bridge, es:

1. Asignar una dirección IP a ambas interfaces, la "izquierda" y la "derecha".
2. Crear rutas de manera que la máquina sepa qué máquinas residen a la izquierda y cuales a la derecha.
3. Ponga en marcha en ambas int Turn on proxy-ARP on both interfaces, echo 1 > /proc/sys/net/ipv4/conf/ethI/proxy_arp, echo 1 > /proc/sys/net/ipv4/conf/ethD/proxy_arp, siendo L y R los números de las interfaces de la izquierda y derecha respectivamente.

Además, ¡no debe olvidar activar el indicador de ip_forwarding! Al pasar a ser un bridge verdadero, podría ver que este indicador está inactivo, ya que no es necesario al hacer un puenteo.

Otra cosa que podría observar al dar este paso es que necesita borrar la caché arp de los computadores de la red (la caché arp puede contener direcciones antiguas pre-bridge que ya no son correctas).

En un Cisco, esto se hace usando la orden "clear arp-cache", y en Linux, utilice "arp -d ip.address". También puede esperar a que la caché expire de forma automática, pero puede tardar un buen rato.

Puede acelerar esto usando la maravillosa herramienta "arping", que en muchas distribuciones es parte del paquete "iputils". Usando "arping" se puede enviar muchos mensajes ARP no solicitados para actualizar las caché arp remotas.

¡Esta es una técnica muy poderosa que también usan los "sombreros negros" (black hats) para subvertir el rutado!

> En Linux 2.4, ¡puede ser que necesite ejecutar "echo 1 > /proc/sys/net/ipv4/ip_nonlocal_bind" antes de poder enviar paquetes ARP no solicitados!

También puede descubrir que la red estaba mal configurada si tiene/tenía el hábito de especificar rutas sin máscaras de red. Me explico, podría ser que en el pasado, algunas versiones de route hayan acertado con la máscara de red correcta, o no, sin que usted se haya dado cuenta. Cuando haga encaminamiento quirúrgico como el descrito anteriormente, ¡es *vital* que compruebe las máscaras de red!

Capítulo 17. Encaminamiento dinámico - OSPF y BGP

Una vez que su red empiece a hacerse realmente grande, o empiece a considerar "la Internet" como su red, necesitará herramientas que encaminen los datos de forma dinámica. A menudo los sitios están conectados unos con otros mediante enlaces múltipls, y aparecen otros de cuando en cuando.

La Internet prácticamente ha estandarizado OSPF y BGP4 (rfc1771). Linux soporta ambos, mediante gated y zebra

Aunque en estos momentos no se encuentra en el ámbito de este document, nos gustaría señalarla las obras definitivas:

Introducción:

Cisco Systems Designing large-scale IP Internetworks

Para OSPF:

Moy, John T. "OSPF. The anatomy of an Internet routing protocol" Addison Wesley. Reading, MA. 1998.

Halabi también ha escrito una buena guía al diseño de encaminamiento con OSPF, pero parece que la han eliminado del sitio web de Cisco.

Para BGP:

Halabi, Bassam "Internet routing architectures" Cisco Press (New Riders Publishing). Indianapolis, IN. 1997.

también

Cisco Systems

Using the Border Gateway Protocol for interdomain routing

Aunque los ejemplos son específics de Cisco, son notablemente similares al lenguaje de configuración de Zebra :-)

Capítulo 18. Otras posibilidades

Este capítulo es una lista de proyectos que tienen que ver con el encaminamiento y el control de tráfico avanzados en Linux. Algunos de estos enlaces merecerían capítulos completos por sí solos, mientras que otros están muy bien documentados por sí mismos, y no necesitan más Cómos.

Implementación para Linux de 802.1Q VLAN (sitio)

> Las VLAN son una manera muy guay de separar sus redes de una manera más virtual que física. Puede encontrar buena información sobre las VLAN aquí. Con esta implementación, puede hacer que su máquina Linux hable VLAN con máquinas como Cisco Catalyst, 3Com: {Corebuilder, Netbuilder II, SuperStack II switch 630}, Extreme Ntwks Summit 48, Foundry: {ServerIronXL, FastIron}.
>
> Aquí puede encontrar un gran Cómo sobre VLAN.
>
> Actualización: incluido en el núcleo desde 2.4.14 (quizá 13).

Implementación alternativa para Linux de 802.1Q VLAN (sitio)

> Implementación alternativa de VLAN para Linux. Este proyecto empezó por un desacuerdo con la arquitectura y el estilo de programación del proyecto "establecido" de VLAN, lo que ha resultado en un diseño general más limpio.

Linux Virtual Server (sitio)

> Esta gente es brillante. El Linux Virtual Server es un servidor de alta escalabilidad y disponibilidad construido sobre un clúster de servidores reales, con el equilibrador de carga ejecutándose en el sistema operativo Linux. La arquitectura del clúster es transparente al usuario final. Este sólo verá un único servidor virtual.
>
> En breve, siempre que necesite equilibrio de carga, a cualquier nivel de tráfico, LVS es el camino a seguir. ¡Algunas de sus técnicas son verdaderamente malvadas! Por ejemplo, permiten que varias máquinas tengan la misma

dirección IP en el mismo segmento, pero desactivándoles ARP. Sólo la máquina LVS hace ARP, y decide cual de las máquinas posteriores deberá gestionar un paquete entrante, enviándoselo directamente a la dirección MAC correcta. El tráfico saliente se envía directamente al router, y no a través de la máquina LVS, que por tanto no necesita ver sus 5Gbit de contenido saliendo hacia el mundo, y por tanto no será un cuello de botella.

El LVS se implementa como un parche para el núcleo en Linux 2.0 y 2.2, y como módulo para Netfilter en 2.4/2.5, ¡de manera que no necesitará parches! El soporte para 2.4 aún se encuentra en principio de desarrollo, de manera que pruébelo y proporcione realimentación o parches.

CBQ.init (sitio)

Configurar CBQ puede ser un poco intimidante, especialmente si todo lo que desea es ajustar el tráfico de algunos computadores hacia un router. CBQ.init puede ayudarle a configurar Linux con una sintaxis simplificada.

Por ejemplo, si desea que todos los computadores de su subred 192.168.1.0/24 (en una eth1 10mbit) tengan limitada la velocidad de descarga a 28kbit/s, ponga esto en el fichero de configuración de CBQ.init:

```
DEVICE=eth1,10Mbit,1Mbit
RATE=28Kbit
WEIGHT=2Kbit
PRIO=5
RULE=192.168.1.0/24
```

Use este programa siempre que no le interese el "cómo y por qué". Estamos usando CBQ.init en producción y funciona muy bien. La documentación está inscrita en el script, lo cual explica por qué no encontrará un README.

Chronox easy shaping scripts (site)

Stephan Mueller (smueller@chronox.de) wrote two useful scripts, 'limit.conn' and 'shaper'. The first one allows you to easily throttle a single download session, like this:

```
# limit.conn -s SERVERIP -p SERVERPORT -l LIMIT
```

It works on Linux 2.2 and 2.4/2.5.

The second script is more complicated, and can be used to make lots of different queues based on iptables rules, which are used to mark packets which are then shaped.

Virtual Router Redundancy Protocol implementation (site)

This is purely for redundancy. Two machines with their own IP address and MAC Address together create a third IP Address and MAC Address, which is

virtual. Originally intended purely for routers, which need constant MAC addresses, it also works for other servers.

The beauty of this approach is the incredibly easy configuration. No kernel compiling or patching required, all userspace.

Just run this on all machines participating in a service:

```
# vrrpd -i eth0 -v 50 10.0.0.22
```

And you are in business! 10.0.0.22 is now carried by one of your servers, probably the first one to run the vrrp daemon. Now disconnect that computer from the network and very rapidly one of the other computers will assume the 10.0.0.22 address, as well as the MAC address.

I tried this over here and had it up and running in 1 minute. For some strange reason it decided to drop my default gateway, but the -n flag prevented that.

This is a 'live' fail over:

```
64 bytes from 10.0.0.22: icmp_seq=3 ttl=255 time=0.2 ms
64 bytes from 10.0.0.22: icmp_seq=4 ttl=255 time=0.2 ms
64 bytes from 10.0.0.22: icmp_seq=5 ttl=255 time=16.8 ms
64 bytes from 10.0.0.22: icmp_seq=6 ttl=255 time=1.8 ms
64 bytes from 10.0.0.22: icmp_seq=7 ttl=255 time=1.7 ms
```

Not *one* ping packet was lost! Just after packet 4, I disconnected my P200 from the network, and my 486 took over, which you can see from the higher latency.

Capítulo 19. Further reading

http://snafu.freedom.org/linux2.2/iproute-notes.html

Contains lots of technical information, comments from the kernel

http://www.davin.ottawa.on.ca/ols/

Slides by Jamal Hadi Salim, one of the authors of Linux traffic control

http://defiant.coinet.com/iproute2/ip-cref/

HTML version of Alexeys LaTeX documentation - explains part of iproute2 in great detail

http://www.aciri.org/floyd/cbq.html

Sally Floyd has a good page on CBQ, including her original papers. None of it is Linux specific, but it does a fair job discussing the theory and uses of CBQ. Very technical stuff, but good reading for those so inclined.

Differentiated Services on Linux

This document by Werner Almesberger, Jamal Hadi Salim and Alexey Kuznetsov describes DiffServ facilities in the Linux kernel, amongst which are TBF, GRED, the DSMARK qdisc and the tcindex classifier.

http://ceti.pl/~kravietz/cbq/NET4_tc.html

Yet another HOWTO, this time in Polish! You can copy/paste command lines however, they work just the same in every language. The author is cooperating with us and may soon author sections of this HOWTO.

IOS Committed Access Rate

From the helpful folks of Cisco who have the laudable habit of putting their documentation online. Cisco syntax is different but the concepts are the same, except that we can do more and do it without routers the price of cars :-)

Docum experimental site(site)

Stef Coene is busy convincing his boss to sell Linux support, and so he is experimenting a lot, especially with managing bandwidth. His site has a lot of practical information, examples, tests and also points out some CBQ/tc bugs.

TCP/IP Illustrated, volume 1, W. Richard Stevens, ISBN 0-201-63346-9

Required reading if you truly want to understand TCP/IP. Entertaining as well.

Capítulo 20. Acknowledgements

It is our goal to list everybody who has contributed to this HOWTO, or helped us demystify how things work. While there are currently no plans for a Netfilter type scoreboard, we do like to recognize the people who are helping.

- Junk Alins

  <juanjo@mat.upc.es>

- Joe Van Andel
- Michael T. Babcock

  <mbabcock@fibrespeed.net>

- Christopher Barton

  <cpbarton%uiuc.edu>

- Ard van Breemen

  <ard%kwaak.net>

- Ron Brinker

  <service%emcis.com>

- ?ukasz Bromirski

  <l.bromirski@mr0vka.eu.org>

- Lennert Buytenhek

  <buytenh@gnu.org>

- Esteve Camps

  <esteve@hades.udg.es>

- Stef Coene

  <stef.coene@docum.org>

- Don Cohen

  <don-lartc%isis.cs3-inc.com>

- Jonathan Corbet

  <lwn%lwn.net>

- Gerry N5JXS Creager

  <gerry%cs.tamu.edu>

- Marco Davids

  <marco@sara.nl>

- Jonathan Day

  <jd9812@my-deja.com>

- Martin aka devik Devera

  <devik@cdi.cz>

- Stephan "Kobold" Gehring

  <Stephan.Gehring@bechtle.de>

- Jacek Glinkowski

  <jglinkow%hns.com>

- Andrea Glorioso

  <sama%perchetopi.org>

- Nadeem Hasan

  <nhasan@usa.net>

- Erik Hensema

  <erik%hensema.xs4all.nl>

- Vik Heyndrickx

  <vik.heyndrickx@edchq.com>

- Spauldo Da Hippie

  <spauldo%usa.net>

- Koos van den Hout

  <koos@kzdoos.xs4all.nl>

- Stefan Huelbrock <shuelbrock%datasystems.de>
- Alexander W. Janssen <yalla%ynfonatic.de>
- Gareth John <gdjohn%zepler.org>
- Dave Johnson

<dj@www.uk.linux.org>

- Martin Josefsson <gandalf%wlug.westbo.se>
- Andi Kleen <ak%suse.de>
- Andreas J. Koenig <andreas.koenig%anima.de>
- Pawel Krawczyk <kravietz%alfa.ceti.pl>
- Amit Kucheria <amitk@ittc.ku.edu>
- Edmund Lau <edlau%ucf.ics.uci.edu>
- Philippe Latu <philippe.latu%linux-france.org>
- Arthur van Leeuwen <arthurvl%sci.kun.nl>
- Jose Luis Domingo Lopez

  <jdomingo@24x7linux.com>

- Jason Lunz <j@cc.gatech.edu>
- Stuart Lynne <sl@fireplug.net>
- Alexey Mahotkin <alexm@formulabez.ru>
- Predrag Malicevic <pmalic@ieee.org>
- Patrick McHardy <kaber@trash.net>
- Andreas Mohr <andi%lisas.de>
- Andrew Morton <akpm@zip.com.au>
- Wim van der Most
- Stephan Mueller <smueller@chronox.de>
- Togan Muftuoglu <toganm%yahoo.com>
- Chris Murray <cmurray@stargate.ca>
- Patrick Nagelschmidt <dto%gmx.net>
- Ram Narula <ram@princess1.net>
- Jorge Novo <jnovo@educanet.net>
- Patrik <ph@kurd.nu>
- P?l Osgy?ny <oplab%westel900.net>
- Lutz Preßler <Lutz.Pressler%SerNet.DE>
- Jason Pyeron <jason%pyeron.com>
- Rusty Russell <rusty%rustcorp.com.au>
- Mihai RUSU <dizzy%roedu.net>
- Jamal Hadi Salim <hadi%cyberus.ca>
- Ren? Serral <rserral%ac.upc.es>
- David Sauer <davids%penguin.cz>
- Sheharyar Suleman Shaikh <sss23@drexel.edu>
- Stewart Shields <MourningBlade%bigfoot.com>
- Nick Silberstein <nhsilber%yahoo.com>
- Konrads Smelkov <konrads@interbaltika.com>
- William Stearns

  <wstearns@pobox.com>

- Andreas Steinmetz <ast%domdv.de>
- Jason Tackaberry <tack@linux.com>
- Charles Tassell <ctassell%isn.net>
- Glen Turner <glen.turner%aarnet.edu.au>
- Tea Sponsor: Eric Veldhuyzen <eric%terra.nu>
- Song Wang <wsong@ece.uci.edu>

- Lazar Yanackiev

  <Lyanackiev%gmx.net>