HOWTO du routage avancé et du contrôle de trafic sous 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>

Laurent Foucher

Traducteur

<foucher@gch.iut-tlse3.fr>

Philippe Latu

Traducteur/Relecteur

<philippe.latu@linux-france.org>

Guillaume Allègre

Relecteur

<Guillaume.Allegre@imag.fr>

Historique des versions

Version $Revision: 1.1.1.1 $	$Date: 2003/01/03 02:38:54 $
DocBook Edition	

Une approche pratique d'iproute2, de la mise en forme du trafic et un peu de netfilter.

Table des matières

Chapitre 1. Dédicace

Ce document est dédié à beaucoup de gens ; dans ma tentative de tous me les rappeler, je peux en citer quelques−uns :

- Rusty Russell
- Alexey N. Kuznetsov
- La fine équipe de Google
- L'équipe de Casema Internet

Chapitre 2. Introduction

Bienvenue, cher lecteur.

Ce document a pour but de vous éclairer sur la manière de faire du routage avancé avec Linux 2.2/2.4. Méconnus par les utilisateurs, les outils standard de ces noyaux permettent de faire des choses spectaculaires. Les commandes comme **route** et **ifconfig** sont des interfaces vraiment pauvres par rapport à la grande puissance potentielle d'iproute2.

J'espère que ce HOWTO deviendra aussi lisible que ceux de Rusty Russell, très réputé (parmi d'autres choses) pour son netfilter.

Vous pouvez nous contacter en nous écrivant à l'équipe HOWTO. Cependant, postez, s'il vous plaît, vos questions sur la liste de diffusion (voir la section correspondante) pour celles qui ne sont pas directement liées à ce HOWTO.

Avant de vous perdre dans ce HOWTO, si la seule chose que vous souhaitez faire est de la simple mise en forme de trafic, allez directement au chapitre *Autres possibilités*, et lisez ce qui concerne CBQ.init.

2.1. Conditions de distribution et Mise en garde

Ce document est distribué dans l'espoir qu'il sera utile et utilisé, mais SANS AUCUNE GARANTIE ; sans même une garantie implicite de qualité légale et marchande ni aptitude à un quelconque usage.

En un mot, si votre dorsale STM−64 est tombée ou distribue de la pornographie à vos estimés clients, cela n'est pas de notre faute. Désolé.

Copyright (c) 2001 par Bert Hubert, Gregory Maxwell et Martijn van Oosterhout, Remco van Mook, Paul B. Schroeder et autres. Ce document ne peut être distribué qu'en respectant les termes et les conditions exposés dans la Open Publication License, v1.0 ou supérieure (la dernière version est actuellement disponible sur http://www.opencontent.org/openpub/).

Copiez et distribuez (vendez ou donnez) librement ce document, dans n'importe quel format. Les demandes de corrections et/ou de commentaires sont à adresser à la personne qui maintient ce document.

Il est aussi demandé que, si vous publiez cet HOWTO sur un support papier, vous en envoyiez des exemplaires aux auteurs pour une << relecture critique >> :−)

2.2. Connaissances préalables

Comme le titre l'implique, ceci est un HOWTO << avancé >>. Bien qu'il ne soit pas besoin d'être un expert réseau, certains pré−requis sont nécessaires.

Voici d'autres références qui pourront vous aider à en apprendre plus :

Rusty Russell's networking−concepts−HOWTO
: Très bonne introduction, expliquant ce qu'est un réseau, et comment on le connecte à d'autres réseaux.

Linux Networking−HOWTO (ex Net−3 HOWTO)
: Excellent document, bien que très bavard. Il vous apprendra beaucoup de choses qui sont déjà configurées si vous êtes capable de vous connecter à Internet. Il peut éventuellement être situé à `/usr/doc/HOWTO/NET-HOWTO.txt`, mais peut également être trouvé en ligne

2.3. Ce que Linux peut faire pour vous

Une petite liste des choses qui sont possibles :

- Limiter la bande passante pour certains ordinateurs
- Limiter la bande passante VERS certains ordinateurs
- Vous aider à partager équitablement votre bande passante
- Protéger votre réseau des attaques de type Déni de Service
- Protéger Internet de vos clients
- Multiplexer plusieurs serveurs en un seul, pour l'équilibrage de charge ou une disponibilité améliorée
- Restreindre l'accès à vos ordinateurs
- Limiter l'accès de vos utilisateurs vers d'autres hôtes

* Faire du routage basé sur l'ID utilisateur (eh oui !), l'adresse MAC, l'adresse IP source, le port, le type de service, l'heure ou le contenu.

Peu de personnes utilisent couramment ces fonctionnalités avancées. Il y a plusieurs raisons à cela. Bien que la documentation soit fournie, la prise en main est difficile. Les commandes de contrôle du trafic ne sont pratiquement pas documentées.

2.4. Notes diverses

Il y a plusieurs choses qui doivent être notées au sujet de ce document. Bien que j'en ai écrit la majeure partie, je ne veux vraiment pas qu'il reste tel quel. Je crois beaucoup à l'Open Source, je vous encourage donc à envoyer des remarques, des mises à jour, des corrections, etc. N'hésitez pas à m'avertir des coquilles ou d'erreurs pures et simples. Si mon anglais vous paraît parfois peu naturel, ayez en tête, s'il vous plaît, que l'anglais n'est pas ma langue natale. N'hésitez pas à m'envoyer vos suggestions [NdT : en anglais !]

Si vous pensez que vous êtes plus qualifié que moi pour maintenir une section ou si vous pensez que vous pouvez écrire et maintenir de nouvelles sections, vous êtes le bienvenu. La version SGML de ce HOWTO est disponible via CVS. J'envisage que d'autre personnes puissent travailler dessus.

Pour vous aider, vous trouverez beaucoup de mentions FIXME (NdT : A CORRIGER). Les corrections sont toujours les bienvenues. Si vous trouvez une mention FIXME, vous saurez que vous êtes en territoire inconnu. Cela ne veut pas dire qu'il n'y a pas d'erreurs ailleurs, faites donc très attention. Si vous avez validé quelque chose, faites−nous le savoir, ce qui nous permettra de retirer la mention FIXME.

Je prendrai quelques libertés tout au long de cet HOWTO. Par exemple, je pars de l'hypothèse d'une connexion Internet à 10 Mbits, bien que je sache très bien que cela ne soit pas vraiment courant.

2.5. Accès, CVS et propositions de mises à jour

L'adresse canonique de cet HOWTO est Ici.

Nous avons maintenant un CVS en accès anonyme disponible depuis le monde entier. Cela est intéressant pour plusieurs raisons. Vous pouvez facilement télécharger les nouvelles versions de ce HOWTO et soumettre des mises à jour.

En outre, cela permet aux auteurs de travailler sur la source de façon indépendante, ce qui est une bonne chose aussi.

```
$ export CVSROOT=:pserver:anon@outpost.ds9a.nl:/var/cvsroot
$ cvs login
CVS password: [enter 'cvs' (sans les caractères ')]
$ cvs co 2.4routing
cvs server: Updating 2.4routing
U 2.4routing/2.4routing.sgml
```

Si vous repérez une erreur ou voulez ajouter quelque chose, faites le en local, exécutez **`cvs diff −u`**, et envoyez−nous le résultat.

Un fichier Makefile est fourni pour vous aider à créer des fichiers PostScript, dvi, pdf, html et texte. Vous pouvez avoir à installer les docbook, docbook−utils, ghostscript et tetex pour obtenir tous les formats de sortie.

2.6. Liste de diffusion

Les auteurs reçoivent de plus en plus de courriers électroniques à propos de cet HOWTO. Vu l'intérêt de la communauté, il a été décidé la mise en place d'une liste de diffusion où les personnes pourront discuter du routage avancé et du contrôle de trafic. Vous pouvez vous abonner à la liste ici.

Il devra être noté que les auteurs sont très hésitants à répondre à des questions qui n'ont pas été posées sur la liste. Nous aimerions que la liste devienne une sorte de base de connaissance. Si vous avez une question, recherchez, s'il vous plaît, d'abord dans l'archive, et ensuite postez−là dans la liste de diffusion.

2.7. Plan du document

Nous allons essayer de faire des manipulations intéressantes dès le début, ce qui veut dire que tout ne sera pas expliqué en détail tout de suite. Veuillez passer sur ces détails, et accepter de considérer qu'ils deviendront clairs par la suite.

Le routage et le filtrage sont deux choses distinctes. Le filtrage est très bien documenté dans le HOWTO de Rusty, disponible ici :

- Rusty's Remarkably Unreliable Guides

Nous nous focaliserons principalement sur ce qu'il est possible de faire en combinant netfilter et iproute2.

Chapitre 3. Introduction à iproute2

3.1. Pourquoi iproute2 ?

La plupart des distributions Linux et des UNIX utilisent couramment les vénérables commandes **arp**, **ifconfig** et **route**. Bien que ces outils fonctionnent, ils montrent quelques comportements inattendus avec les noyaux Linux des séries 2.2 et plus. Par exemple, les tunnels GRE font partie intégrante du routage de nos jours, mais ils nécessitent des outils complètement différents.

Avec iproute2, les tunnels font partie intégrante des outils.

Les noyaux Linux des séries 2.2 et plus ont un sous−système réseau complètement réécrit. Ce nouveau codage de la partie réseau apporte à Linux des performances et des fonctionnalités qui n'ont pratiquement pas d'équivalent parmi les autres systèmes d'exploitation. En fait, le nouveau logiciel de filtrage, routage et de classification possède plus de fonctionnalités que les logiciels fournis sur beaucoup de routeurs dédiés, de pare−feu et de produits de mise en forme (*shaping*) du trafic.

Dans les systèmes d'exploitation existants, au fur et à mesure que de nouveaux concepts réseau apparaissaient, les développeurs sont parvenus à les greffer sur les structures existantes. Ce travail constant d'empilage de couches a conduit à des codes réseau aux comportements étranges, un peu comme les langues humaines. Dans le passé, Linux émulait le mode de fonctionnement de SunOS, ce qui n'était pas l'idéal.

La nouvelle structure d'iproute2 a permis de formuler clairement des fonctionnalités impossibles à implémenter dans le sous−système réseau précédent.

3.2. Un tour d'horizon d'iproute2

Linux possède un système sophistiqué d'allocation de bande passante appelé Contrôle de trafic (*Traffic Control*). Ce système supporte différentes méthodes pour classer, ranger par ordre de priorité, partager et limiter le trafic entrant et sortant.

Nous commencerons par un petit tour d'horizon des possibilités d'iproute2.

3.3. Prérequis

Vous devez être sûr que vous avez installé les outils utilisateur (NdT : userland tools, par opposition à la partie << noyau >> d'iproute2). Le paquet concerné s'appelle iproute sur RedHat et Debian. Autrement, il peut être trouvé à `ftp://ftp.inr.ac.ru/ip-routing/iproute2-2.2.4-now-ss??????.tar.gz`.

Vous pouvez aussi essayer iproute2−current.tar.gz pour la dernière version.

Certains éléments d'iproute vous imposent l'activation de certaines options du noyau. Il devra également être noté que toutes les versions de RedHat jusqu'à la version 6.2 incluse n'ont pas les fonctionnalités du contrôle de trafic dans le noyau par défaut.

RedHat 7.2 contient tous les éléments par défaut.

Soyez également sûr que vous avez le support *netlink*, même si vous devez choisir de compiler votre propre noyau ; iproute2 en a besoin.

3.4. Explorer votre configuration courante

Cela peut vous paraître surprenant, mais iproute2 est déjà configuré ! Les commandes courantes **ifconfig** et **route** utilisent déjà les appels système avancés d'iproute2, mais essentiellement avec les options par défaut (c'est−à−dire ennuyeuses).

L'outil **ip** est central, et nous allons lui demander de nous montrer les interfaces.

3.4.1. ip nous montre nos liens

```
[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
```

La sortie peut varier, mais voici ce qui est affiché pour mon routeur NAT (NdT : translation d'adresse) chez moi. J'expliquerai seulement une partie de la sortie, dans la mesure où tout n'est pas directement pertinent.

La première interface que nous voyons est l'interface *loopback*. Bien que votre ordinateur puisse fonctionner sans, je vous le déconseille. La taille de MTU (unité maximum de transmission) est de 3924 octets, et *loopback* n'est pas supposé être mis en file d'attente, ce qui prend tout son sens dans la mesure où cette interface est le fruit de l'imagination de votre noyau.

Je vais passer sur l'interface *dummy* pour l'instant, et il se peut qu'elle ne soit pas présente sur votre ordinateur. Il y a ensuite mes deux interfaces physiques, l'une du côté de mon modem câble, l'autre servant mon segment ethernet à la maison. De plus, nous voyons une interface *ppp0*.

Notons l'absence d'adresses IP. Iproute déconnecte les concepts de << liens >> et << d'adresses IP >>. Avec l'*IP aliasing*, le concept de l'adresse IP canonique est devenu, de toute façon, sans signification.

ip nous montre bien, cependant, l'adresse MAC, l'identifiant matériel de nos interfaces ethernet.

3.4.2. ip nous montre nos adresses 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
```

Cela contient plus d'informations : **ip** montre toutes nos adresses, et à quelles cartes elles appartiennent. *inet* signifie Internet (IPv4). Il y a beaucoup d'autres familles d'adresses, mais elles ne nous concernent pas pour le moment.

Examinons l'interface *eth0* de plus près. Il est dit qu'elle est reliée à l'adresse internet *10.0.0.1/8*. Qu'est–ce que cela signifie ? Le /8 désigne le nombre de bits réservés à l'adresse réseau. Il y a 32 bits, donc il reste 24 bits pour désigner une partie de notre réseau. Les 8 premiers bits de *10.0.0.1* correspondent à *10.0.0.0*, notre adresse réseau, et notre masque de sous–réseau est *255.0.0.0*.

Les autres bits repèrent des machines directement connectées à cette interface. Donc, *10.250.3.13* est directement disponible sur *eth0*, comme l'est *10.0.0.1* dans notre exemple.

Avec *ppp0*, le même concept existe, bien que les nombres soient différents. Son adresse est *212.64.94.251*, sans masque de sous–réseau. Cela signifie que vous avez une liaison point à point et que toutes les adresses, à l'exception de *212.64.94.251*, sont distantes. Il y a cependant plus d'informations. En effet, on nous dit que de l'autre côté du lien, il n'y a encore qu'une seule adresse, *212.64.94.1*. Le /32 nous précise qu'il n'y a pas de << bits réseau >>.

Il est absolument vital que vous compreniez ces concepts. Référez–vous à la documentation mentionnée au début de ce HOWTO si vous avez des doutes.

Vous pouvez aussi noter *qdisc*, qui désigne la gestion de la mise en file d'attente (*Queueing Discipline*). Cela deviendra vital plus tard.

3.4.3. ip nous montre nos routes

Nous savons maintenant comment trouver les adresses *10.x.y.z*, et nous sommes capables d'atteindre *212.64.94.1*. Cela n'est cependant pas suffisant, et nous avons besoin d'instructions pour atteindre le monde. L'Internet est disponible via notre connexion PPP, et il se trouve que *212.64.94.1* est prêt à propager nos paquets à travers le monde, et à nous renvoyer le résultat.

```
[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
```

Cela se comprend de soi−même. Les 4 premières lignes donnent explicitement ce qui était sous−entendu par **`ip address show`**, la dernière ligne nous indiquant que le reste du monde peut être trouvé via *212.64.94.1*, notre passerelle par défaut. Nous pouvons voir que c'est une passerelle à cause du mot << via >>, qui nous indique que nous avons besoin d'envoyer les paquets vers *212.64.94.1*, et que c'est elle qui se chargera de tout.

En référence, voici ce que l'ancien utilitaire **route** nous propose :

```
[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 est le Protocole de Résolution d'Adresse (*Address Resolution Protocol*). Il est décrit dans le RFC 826. ARP est utilisé par une machine d'un réseau local pour retrouver l'adresse matérielle (la localisation) d'une autre machine sur le même réseau. Les machines sur Internet sont généralement connues par leur nom auquel correspond une adresse IP. C'est ainsi qu'une machine sur le réseau *foo.com* est capable de communiquer avec une autre machine qui est sur le réseau *bar.net*. Une adresse IP, cependant, ne peut pas vous indiquer la localisation physique de la machine. C'est ici que le protocole ARP entre en jeu.

Prenons un exemple très simple. Supposons que j'aie un réseau composé de plusieurs machines, dont la machine *foo* d'adresse IP *10.0.0.1* et la machine *bar* qui a l'adresse IP *10.0.0.2*. Maintenant, *foo* veut envoyer un **ping** vers *bar* pour voir s'il est actif, mais *foo* n'a aucune indication sur la localisation de *bar*. Donc, si *foo* décide d'envoyer un **ping** vers *bar*, il a besoin d'envoyer une requête ARP. Cette requête ARP est une façon pour *foo* de crier sur le réseau << Bar (10.0.0.2) ! Où es−tu ? >>. Par conséquent, toutes les machines sur le réseau entendront *foo* crier, mais seul *bar* (*10.0.0.2*) répondra. *Bar* enverra une réponse ARP directement à *foo* ; ce qui revient à dire : << Foo (10.0.0.1) ! je suis ici, à l'adresse 00:60:94:E:08:12 >>. Après cette simple transaction utilisée pour localiser son ami sur le réseau, *foo* est capable de communiquer avec *bar* jusqu'à ce qu'il (le cache ARP de *foo*) oublie où bar est situé (typiquement au bout de 15 minutes sur Unix).

Maintenant, voyons comment cela fonctionne. Vous pouvez consulter votre cache (table) ARP (*neighbor*) comme ceci :

```
[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
```

Comme vous pouvez le voir, ma machine *espa041* (*9.3.76.41*) sait où trouver *espa042* (*9.3.76.42*) et *espagate* (*9.3.76.1*). Maintenant, ajoutons une autre machine dans le cache 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
```

```
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
```

Par conséquent, lorsque *espa041* a essayé de contacter *espa043*, l'adresse matérielle de *espa043* (sa localisation) a alors été ajoutée dans le cache ARP. Donc, tant que la durée de vie de l'entrée correspondant à *espa043* dans le cache ARP n'est pas dépassée, *espa041* sait localiser *espa043* et n'a plus besoin d'envoyer de requête ARP.

Maintenant, effaçons *espa043* de notre cache 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
```

Maintenant, *espa041* a à nouveau oublié la localisation d'*espa043* et aura besoin d'envoyer une autre requête ARP la prochaine fois qu'il voudra communiquer avec lui. Vous pouvez aussi voir ci−dessus que l'état d'*espagate* (*9.3.76.1*) est passé en *stale*. Cela signifie que la localisation connue est encore valide, mais qu'elle devra être confirmée à la première transaction avec cette machine.

Chapitre 4. Règles − bases de données des politiques de routage

Si vous avez un routeur important, il se peut que vous vouliez satisfaire les besoins de différentes personnes, qui peuvent être traitées différemment. Les bases de données des politiques de routage vous aident à faire cela, en gérant plusieurs ensembles de tables de routage.

Si vous voulez utiliser cette fonctionnalité, assurez−vous que le noyau est compilé avec les options *IP : Advanced router* et *IP : policy routing*.

Quand le noyau doit prendre une décision de routage, il recherche quelle table consulter. Par défaut, il y a trois tables. L'ancien outil **route** modifie les tables principale (*main*) et locale (*local*), comme le fait l'outil **ip** (par défaut).

Les règles par défaut :

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

Ceci liste la priorité de toutes les règles. Nous voyons que toutes les règles sont appliquées à tous les paquets (*from all*). Nous avons vu la table *main* précédemment, sa sortie s'effectuant avec **`ip route ls`**, mais les tables *local* et *default* sont nouvelles.

Si nous voulons faire des choses fantaisistes, nous pouvons créer des règles qui pointent vers des tables différentes et qui nous permettent de redéfinir les règles de routage du système.

Pour savoir exactement ce que fait le noyau en présence d'un assortiment de règles plus complet, référez−vous à la documentation ip−cref d'Alexey [NdT : dans le paquetage iproute2 de votre distribution].

4.1. Politique de routage simple par l'adresse source

Prenons encore une fois un exemple réel. J'ai 2 modems câble, connectés à un routeur Linux NAT (*masquerading*). Les personnes habitant avec moi me paient pour avoir accès à Internet. Supposons qu'un de mes co−locataires consulte seulement hotmail et veuille payer moins. C'est d'accord pour moi, mais il utilisera le modem le plus lent.

Le modem câble << rapide >> est connu sous *212.64.94.251* et est en liaison PPP avec *212.64.94.1*. Le modem câble << lent >> est connu sous diverses adresses IP : *212.64.78.148* dans notre exemple avec un lien vers *195.96.98.253*.

La table locale :

```
[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
```

Il y a beaucoup de choses évidentes, mais aussi des choses qui ont besoin d'être précisées quelque peu, ce que nous allons faire. La table de routage par défaut est vide.

Regardons la table principale (*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
```

Maintenant, nous générons une nouvelle règle que nous appellerons *John*, pour notre hypothétique co−locataire. Bien que nous puissions travailler avec des nombres IP purs, il est plus facile d'ajouter notre table dans le fichier `/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
```

Maintenant, tout ce qu'il reste à faire est de générer la table *John*, et de vider le cache des routes :

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

Et voilà qui est fait. Il ne reste plus, comme exercice laissé au lecteur, qu'à implémenter cela dans **ip−up**.

Chapitre 5. GRE et autres tunnels

Il y a trois sortes de tunnels sous Linux : l'IP dans un tunnel IP, le tunnel GRE et les tunnels qui existent en dehors du noyau (comme PPTP, par exemple).

5.1. Quelques remarques générales à propos des tunnels :

Les tunnels peuvent faire des choses très inhabituelles et vraiment sympas. Ils peuvent aussi absolument tout détraquer si vous ne les avez pas configurés correctement. Ne définissez pas votre route par défaut sur un tunnel, à moins que vous ne sachiez *EXACTEMENT* ce que vous faites.

De plus, le passage par un tunnel augmente le poids des en−têtes (*overhead*), puisqu'un en−tête IP supplémentaire est nécessaire. Typiquement, ce surcoût est de 20 octets par paquet. Donc, si la taille maximum de votre paquet sur votre réseau (MTU) est de 1500 octets, un paquet qui est envoyé à travers un tunnel sera limité à une taille de 1480 octets. Ce n'est pas nécessairement un problème, mais soyez sûr d'avoir bien étudié la fragmentation et le réassemblage des paquets IP quand vous prévoyez de relier des réseaux de grande taille par des tunnels. Et bien sûr, la manière la plus rapide de creuser un tunnel est de creuser des deux côtés.

5.2. IP dans un tunnel IP

Ce type de tunnel est disponible dans Linux depuis un long moment. Il nécessite deux modules, **ipip.o** et **new_tunnel.o**.

Disons que vous avez trois réseaux : 2 réseaux internes A et B, et un réseau intermédiaire C (ou disons Internet). Les caractéristiques du réseau A sont :

```
réseau 10.0.1.0
masque de sous−réseau 255.255.255.0
routeur  10.0.1.1
```

Le routeur a l'adresse *172.16.17.18* sur le réseau C.

et le réseau B :

```
réseau 10.0.2.0
masque de sous−réseau 255.255.255.0
routeur  10.0.2.1
```

Le routeur a l'adresse *172.19.20.21* sur le réseau C.

En ce qui concerne le réseau C, nous supposerons qu'il transmettra n'importe quel paquet de A vers B et vice−versa. Il est également possible d'utiliser l'Internet pour cela.

Voici ce qu'il faut faire :

Premièrement, assurez−vous que les modules soient installés :

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

Ensuite, sur le routeur du réseau A, faites la chose suivante :

```
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
```

et sur le routeur du réseau 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
```

Et quand vous aurez terminé avec votre tunnel :

```
ifconfig tunl0 down
```

Vite fait, bien fait. Vous ne pouvez pas transmettre les paquets de diffusion (*broadcast*), ni le trafic IPv6 à travers un tunnel IP−IP. Vous ne pouvez connecter que deux réseaux IPv4 qui, normalement, ne seraient pas capables de se << parler >>, c'est tout. Dans la mesure où la compatibilité a été conservée, ce code tourne depuis un bon moment, et il reste compatible depuis les noyaux 1.3. Le tunnel Linux IP dans IP ne fonctionne pas avec d'autres systèmes d'exploitation ou routeurs, pour autant que je sache. C'est simple, ça marche. Utilisez−le si vous le pouvez, autrement utilisez GRE.

5.3. Le tunnel GRE

GRE est un protocole de tunnel qui a été à l'origine développé par Cisco, et qui peut réaliser plus de choses que le tunnel IP dans IP. Par exemple, vous pouvez aussi transporter du trafic multi−diffusion (*multicast*) et de l'IPv6 à travers un tunnel GRE.

Dans Linux, vous aurez besoin du module ip_gre.o.

5.3.1. Le tunnel IPv4

Dans un premier temps, intéressons−nous au tunnel IPv4 :

Disons que vous avez trois réseaux : 2 réseaux internes A et B, et un réseau intermédiaire C (ou disons internet).

Les caractéristiques du réseau A sont :

```
réseau 10.0.1.0
masque de sous-réseau 255.255.255.0
routeur  10.0.1.1
```

Le routeur a l'adresse *172.16.17.18* sur le réseau C. Appelons ce réseau *neta*.

Et pour le réseau B :

```
réseau 10.0.2.0
masque de sous-réseau 255.255.255.0
routeur  10.0.2.1
```

Le routeur a l'adresse *172.19.20.21* sur le réseau C. Appelons ce réseau *netb*.

En ce qui concerne le réseau C, nous supposerons qu'il transmettra n'importe quels paquets de A vers B et vice−versa. Comment et pourquoi, on s'en moque.

Sur le routeur du réseau A, nous faisons la chose suivante :

```
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
```

Discutons un peu de cela. Sur la ligne 1, nous avons ajouté un périphérique tunnel, que nous avons appelé *netb* (ce qui est évident, dans la mesure où c'est là que nous voulons aller). De plus, nous lui avons dit d'utiliser le protocole GRE (*mode gre*), que l'adresse distante est *172.19.20.21* (le routeur de l'autre côté), que nos paquets << tunnelés >> devront être générés à partir de *172.16.17.18* (ce qui autorise votre serveur à avoir plusieurs adresses IP sur le réseau C et ainsi vous permet de choisir laquelle sera utilisée pour votre tunnel) et que le champ TTL de vos paquets sera fixé à 255 (*ttl 255*).

La deuxième ligne active le périphérique.

Sur la troisième ligne, nous avons donné à cette nouvelle interface l'adresse *10.0.1.1*. C'est bon pour de petits réseaux, mais quand vous commencez une exploitation minière (*BEAUCOUP* de tunnels !), vous pouvez utiliser une autre gamme d'adresses IP pour vos interfaces << tunnel >> (dans cet exemple, vous pourriez utiliser *10.0.3.0*).

Sur la quatrième ligne, nous positionnons une route pour le réseau B. Notez la notation différente pour le masque de sous−réseau. Si vous n'êtes pas familiarisé avec cette notation, voici comment ça marche : vous écrivez le masque de sous−réseau sous sa forme binaire, et vous comptez tous les 1. Si vous ne savez pas comment faire cela, rappelez−vous juste que *255.0.0.0* est /8, *255.255.0.0* est /16 et *255.255.255.0* est /24. Et *255.255.254.0* est /23, au cas où ça vous intéresserait.

Mais arrêtons ici, et continuons avec le routeur du réseau 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
```

Et quand vous voudrez retirer le tunnel sur le routeur A :

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

Bien sûr, vous pouvez remplacer *netb* par *neta* pour le routeur B.

5.3.2. Le tunnel IPv6

Voir la section 6 pour une courte description des adresses IPv6.

À propos des tunnels.

Supposons que vous ayez le réseau IPv6 suivant, et que vous vouliez le connecter à une dorsale IPv6 (6bone) ou à un ami.

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

Votre adresse IPv4 est *172.16.17.18*, et le routeur 6bone a une adresse 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
```

Voyons cela de plus près. Sur la première ligne, nous avons créé un périphérique tunnel appelé *sixbone*. Nous lui avons affecté le mode *sit* (qui est le tunnel IPv6 sur IPv4) et lui avons dit où l'on va (*remote*) et d'où l'on vient (*local*). *TTL* est configuré à son maximum : *255*. Ensuite, nous avons rendu le périphérique actif (*up*). Puis, nous avons ajouté notre propre adresse réseau et configuré une route pour *3ffe::/15* à travers le tunnel.

Les tunnels GRE constituent actuellement le type de tunnel préféré. C'est un standard qui est largement adopté, même à l'extérieur de la communauté Linux, ce qui constitue une bonne raison de l'utiliser.

5.4. Tunnels dans l'espace utilisateur

Il y a des dizaines d'implémentations de tunnels à l'extérieur du noyau. Les plus connues sont bien sûr PPP et PPTP, mais il y en a bien plus (certaines propriétaires, certaines sécurisés, d'autres qui n'utilisent pas IP), qui dépassent le cadre de ce HOWTO.

Chapitre 6. Tunnel IPv6 avec Cisco et/ou une dorsale IPv6 (6bone)

Par Marco Davids <marco@sara.nl>

NOTE au mainteneur :

En ce qui me concerne, ce tunnel IPv6−IPv4 n'est pas, par définition, un tunnel GRE. Vous pouvez réaliser un tunnel IPv6 sur IPv4 au moyen des périphériques tunnels GRE (tunnels GRE *N'IMPORTE QUOI* vers IPv4), mais le périphérique utilisé ici (*sit*) ne permet que des tunnels IPv6 sur IPv4, ce qui est quelque chose de différent.

6.1. Tunnel IPv6

Voici une autre application des possibilités de tunnels de Linux. Celle−ci est populaire parmi les premiers adeptes d'IPv6 ou les pionniers si vous préférez. L'exemple pratique décrit ci−dessous n'est certainement pas la seule manière de réaliser un tunnel IPv6. Cependant, c'est la méthode qui est souvent utilisée pour réaliser un tunnel entre Linux et un routeur Cisco IPv6 et l'expérience m'a appris que c'est ce type d'équipement que beaucoup de personnes ont. Dix contre un que ceci s'appliquera aussi pour vous ;−).

De petites choses à propos des adresses IPv6 :

Les adresses IPv6 sont, en comparaison avec les adresses IPv4, vraiment grandes : 128 bits contre 32 bits. Et ceci nous fournit la chose dont nous avons besoin : beaucoup, beaucoup d'adresses IP : 340,282,266,920,938,463,463,374,607,431,768,211,465 pour être précis. A part ceci, IPv6 (ou IPng génération suivante (*Next Generation*)) est supposé fournir des tables de routage plus petites sur les routeurs des dorsales Internet, une configuration plus simple des équipements, une meilleure sécurité au niveau IP et un meilleur support pour la Qualité de Service (QoS).

Un exemple : *2002:836b:9820:0000:0000:0000:836b:9886*

Ecrire les adresses IPv6 peut être un peu lourd. Il existe donc des règles qui rendent la vie plus facile :

- Ne pas utiliser les zéros de tête, comme dans IPv4 ;
- Utiliser des double−points de séparation tous les 16 bits ou 2 octets ;
- Quand vous avez beaucoup de zéros consécutifs, vous pouvez écrire *::*. Vous ne pouvez, cependant, faire cela qu'une seule fois par adresse et seulement pour une longueur de 16 bits.

L'adresse *2002:836b:9820:0000:0000:0000:836b:9886* peut être écrite *2002:836b:9820::836b:9886*, ce qui est plus amical.

Un autre exemple : l'adresse *3ffe:0000:0000:0000:0000:0000:34A1:F32C* peut être écrite *3ffe::20:34A1:F32C*, ce qui est beaucoup plus court.

IPv6 a pour but d'être le successeur de l'actuel IPv4. Dans la mesure où cette technologie est relativement récente, il n'y a pas encore de réseau natif IPv6 à l'échelle mondiale. Pour permettre un développement rapide, la dorsale IPv6 (6bone) a été introduite.

Les réseaux natifs IPv6 sont interconnectés grâce à l'encapsulation du protocole IPv6 dans des paquets IPv4, qui sont envoyés à travers l'infrastructure IPv4 existante, d'un site IPv6 à un autre.

C'est dans cette situation que l'on monte un tunnel.

Pour être capable d'utiliser IPv6, vous devrez avoir un noyau qui le supporte. Il y a beaucoup de bons documents qui expliquent la manière de réaliser cela. Mais, tout se résume à quelques étapes :

- Récupérer une distribution Linux récente, avec une glibc convenable.
- Récupérer alors les sources à jour du noyau.

Si tout cela est fait, vous pouvez alors poursuivre en compilant un noyau supportant l'IPv6 :

- Aller dans `/usr/src/linux` et tapez :
- **`make menuconfig`**
- Choisir *Networking Options*
- Sélectionner *The IPv6 protocol*, *IPv6: enable EUI−64 token format*, *IPv6: disable provider based addresses*

ASTUCE :Ne compiler pas ces options en tant que module. Ceci ne marchera souvent pas bien.

En d'autres termes, compilez IPv6 directement dans votre noyau. Vous pouvez alors sauvegarder votre configuration comme d'habitude et entreprendre la compilation de votre noyau.

ASTUCE: Avant de faire cela, modifier votre Makefile comme suit : *EXTRAVERSION = −x ; −−> ; EXTRAVERSION = −x−IPv6*

Il y a beaucoup de bonnes documentations sur la compilation et l'installation d'un noyau. Cependant, ce document ne traite pas de ce sujet. Si vous rencontrez des problèmes à ce niveau, allez et recherchez dans la documentation des renseignements sur la compilation du noyau Linux correspondant à vos propres spécifications.

Le fichier `/usr/src/linux/README` peut constituer un bon départ. Après avoir réalisé tout ceci, et redémarré avec votre nouveau noyau flambant neuf, vous pouvez lancer la commande **/sbin/ifconfig −a** et noter un nouveau périphérique *sit0*. SIT signifie Transition Simple d'Internet (*Simple Internet Transition*). Vous pouvez vous auto−complimenter : vous avez maintenant franchi une étape importante vers IP, la prochaine génération ;−)

Passons maintenant à l'étape suivante. Vous voulez connecter votre hôte ou peut−être même tout votre réseau LAN à d'autres réseaux IPv6. Cela pourrait être la dorsale IPv6 << 6bone >> qui a été spécialement mise en place dans ce but particulier.

Supposons que vous avez le réseau IPv6 suivant : *3ffe:604:6:8::/64* et que vous vouliez le connecter à une dorsale IPv6 ou à un ami. Notez, s'il vous plaît, que la notation sous–réseau */64* est similaire à celle des adresses IPv4.

Votre adresse IPv4 est *145.100.24.181* et le routeur 6bone a l'adresse IPv4 *145.100.1.5*.

```
# ip tunnel add sixbone mode sit remote 145.100.1.5 [local 145.100.24.181 ttl 225]
# ip link set sixbone up
# ip addr add 3FFE:604:6:7::2/96 dev sixbone
# ip route add 3ffe::0/15 dev sixbone
```

Discutons de ceci. Dans la première ligne, nous avons créé un périphérique appelé *sixbone*. Nous lui avons donné l'attribut *sit* (*mode sit*) (qui est le tunnel IPv6 dans IPv4) et nous lui avons dit où aller (*remote*) et d'où nous venions (*local*). *TTL* est configuré à son maximum, 255.

Ensuite, nous avons rendu le périphérique actif (*up*). Après cela, nous avons ajouté notre propre adresse réseau et configuré une route pour *3ffe::/15* (qui est actuellement la totalité du 6bone) à travers le tunnel. Si la machine sur laquelle vous mettez en place tout ceci est votre passerelle IPv6, ajoutez alors les lignes suivantes :

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

En dernière instruction, radvd est un démon d'annonce comme zebra qui permet de supporter les fonctionnalités d'autoconfiguration d'IPv6. Recherchez le avec votre moteur de recherche favori. Vous pouvez vérifier les choses comme ceci :

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

Si vous arrivez à avoir radvd tournant sur votre passerelle IPv6 et que vous démarrez une machine avec IPv6 sur votre réseau local, vous serez ravi de voir les bénéfices de l'autoconfiguration 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
```

Vous pouvez maintenant configurer votre serveur de noms pour les adresses IPv6. Le type *A* a un équivalent pour IPv6 : *AAAA*. L'équivalent de *in–addr.arpa* est : *ip6.int*. Il y a beaucoup d'informations disponibles sur ce sujet.

Il y a un nombre croissant d'applications IPv6 disponibles, comme le shell sécurisé, telnet, inetd, le navigateur Mozilla, le serveur web Apache et beaucoup d'autres. Mais ceci est en dehors du sujet de ce document de routage ;–).

Du côté Cisco, la configuration ressemblera à ceci :

```
!
interface Tunnel1
description IPv6 tunnel
no ip address
no ip directed-broadcast
ipv6 enable
ipv6 address 3FFE:604:6:7::1/96
```

```
tunnel source Serial0
tunnel destination 145.100.24.181
tunnel mode ipv6ip
!
ipv6 route 3FFE:604:6:8::/64 Tunnel1
```

Si vous n'avez pas un Cisco à votre disposition, essayez un des prestataires de tunnel IPv6 disponible sur Internet. Ils sont prêts à configurer leur Cisco avec un tunnel supplémentaire pour vous, le plus souvent au moyen d'une agréable interface web. Cherchez *ipv6 tunnel broker* avec votre moteur de recherche favori.

Chapitre 7. IPsec : IP sécurisé à travers l'Internet

FIXME: Pas de rédacteur. En attendant, voir: Le projet FreeS/WAN. Cerberus du NIST est une autre implémentation de IPSec pour Linux. Cependant, leurs pages web n'ont pas été mis à jour depuis plus d'un an et leur version a tendance à être très nettement à la traîne par rapport aux dernières versions du noyau. USAGI, une implémentation alternative d'IPv6 pour Linux, inclue également une partie IPSec, mais qui n'est que pour IPv6.

Chapitre 8. Routage multidistribution (*multicast*)

FIXME: Pas de rédacteur !

Le Multicast−HOWTO est (relativement) ancien. De ce fait, il peut être imprécis ou induire en erreur à certains endroits.

Avant que vous ne puissiez faire du routage multidistribution, le noyau Linux a besoin d'être configuré pour supporter le type de routage multidistribution que vous voulez faire. Ceci, à son tour, exige une décision quant au choix du protocole de routage multidistribution que vous vous préparez à utiliser. Il y a essentiellement quatre types << communs >> de protocoles : DVMRP (la version multidistribution du protocole RIP unicast), MOSPF (la même chose, mais pour OSPF), PIM−SM (*Protocol Independant Multicasting − Sparse Mode*) qui suppose que les utilisateurs de n'importe quel groupe de multidistribution sont dispersés plutôt que regroupés) et PIM−DM (le même, mais *Dense Mode*) qui suppose qu'il y aura un regroupement significatif des utilisateurs d'un même groupe de multidistribution.

On pourra noter que ces options n'apparaissent pas dans le noyau Linux. Ceci s'explique par le fait que le protocole lui−même est géré par une application de routage, comme Zebra, mrouted ou pind. Cependant, vous devez avoir une bonne idée de ce que vous allez utiliser, de manière à sélectionner les bonnes options dans le noyau.

Pour tout routage multidistribution, vous avez forcément besoin de sélectionner les options *multicasting* et *multicasting routing*. Ceci est suffisant pour DVMRP et MOSPF. Dans le cas de PIM, vous devez également valider les options *PIMv1* ou *PIMv2* suivant que le réseau que vous connectez utilise la version 1 ou 2 du protocole PIM.

Une fois que tout cela a été réalisé, et que votre nouveau noyau a été compilé, vous verrez au démarrage que IGMP est inclus dans la liste des protocoles IP. Celui−ci est un protocole permettant de gérer les groupes multidistribution. Au moment de la rédaction, Linux ne supportait que les versions 1 et 2 de IGMP, bien que la version 3 existe et ait été documentée. Ceci ne va pas vraiment nous affecter dans la mesure où IGMPv3 est encore trop récent pour que ses fonctionnalités supplémentaires soient largement utilisées. Puisque IGMP s'occupe des groupes, seules les fonctionnalités présentes dans la plus simple version de IGMP gérant un groupe entier seront utilisées. IGMPv2 sera utilisé dans la plupart des cas, bien que IGMPv1 puisse encore être rencontré.

Jusque−là, c'est bon. Nous avons activé la multidistribution. Nous devons dire au noyau de l'utiliser concrètement. Nous allons donc démarrer le routage. Ceci signifie que nous ajoutons un réseau virtuel de multidistribution à la table du routeur :

```
ip route add 224.0.0.0/4 dev eth0
```

(En supposant bien sûr, que vous diffusez à travers eth0 ! Remplacez−le par le périphérique de votre choix, si nécessaire.)

Maintenant, dire à Linux de transmettre les paquets...

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

Arrivé ici, il se peut que vous vous demandiez si ceci va faire quelque chose. Donc, pour tester notre connexion, nous pinguons le groupe par défaut, *224.0.0.1*, pour voir si des machines sont présentes. Toutes les machines du réseau local avec la multidistribution activée *DEVRAIENT* répondre, et aucune autre. Vous remarquerez qu'aucune des machines qui répondent ne le fait avec l'adresse IP *224.0.0.1*. Quelle surprise ! :) Ceci est une adresse de groupe (une << diffusion >> pour les abonnés) et tous les membres du groupe répondront avec leur propre adresse, et non celle du groupe.

```
ping −c 2 224.0.0.1
```

Maintenant, vous êtes prêt à faire du vrai routage multidistribution. Bien, en supposant que vous ayez deux réseaux à router l'un vers l'autre.

(A continuer !)

Chapitre 9. Gestionnaires de mise en file d'attente pour l'administration de la bande passante

Quand je l'ai découvert, cela m'a *VRAIMENT* soufflé. Linux 2.2 contient toutes les fonctionnalités pour la gestion de la bande passante, de manière comparable à un système dédié de haut niveau.

Linux dépasse même ce que l'ATM et le Frame peuvent fournir.

Afin d'éviter toute confusion, voici les règles utilisées par **tc** pour la spécification de la bande passante :

```
mbps = 1024 kbps = 1024 * 1024 bps => byte/s (octets/s)
mbit = 1024 kbit => kilo bit/s.
mb = 1024 kb = 1024 * 1024 b => byte (octet)
mbit = 1024 kbit => kilo bit.
```

En interne, les nombres sont stockés en bps et b.

Mais **tc** utilise l'unité suivante lors de l'affichage des débits :

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

9.1. Explication sur les files d'attente et la gestion de la mise en file d'attente

Avec la mise en file d'attente, nous déterminons la manière dont les données sont *ENVOYEES*. Il est important de comprendre que nous ne pouvons mettre en forme que les données que nous transmettons.

Avec la manière dont Internet travaille, nous n'avons pas de contrôle direct sur ce que les personnes nous envoient. C'est un peu comme votre boîte aux lettres (physique !) chez vous. Il n'y a pas de façon d'influencer le nombre de lettres que vous recevez, à moins de contacter tout le monde.

Cependant, l'Internet est principalement basé sur TCP/IP qui possède quelques fonctionnalités qui vont pouvoir nous aider. TCP/IP n'a pas d'aptitude à connaître les performances d'un réseau entre deux hôtes. Il envoie donc simplement des paquets de plus en plus rapidement (<< *slow start* >>) et quand des paquets commencent à se perdre, il ralentit car il n'a plus la possibilité de les envoyer. En fait, c'est un peu plus élégant que cela, mais nous en dirons plus par la suite.

C'est comme si vous ne lisiez que la moitié de votre courrier en espérant que vos correspondants arrêteront de vous en envoyer. À la différence que ça marche sur Internet :−)

Si vous avez un routeur et que vous souhaitez éviter que certains hôtes de votre réseau aient des vitesses de téléchargement trop grandes, vous aurez besoin de mettre en place de la mise en forme de trafic sur l'interface *INTERNE* de votre routeur, celle qui envoie les données vers vos propres ordinateurs.

Vous devez également être sûr que vous contrôlez le goulot d'étranglement de la liaison. Si vous avez une carte réseau à 100Mbit et un routeur avec un lien à 256kbit, vous devez vous assurer que vous n'envoyez pas plus de données que ce que le routeur peut manipuler. Autrement, ce sera le routeur qui contrôlera le lien et qui mettra en forme la bande passante disponible. Nous devons pour ainsi dire << être le propriétaire de la file d'attente >> et être le lien le plus lent de la chaîne. Heureusement, c'est facilement réalisable.

9.2. Gestionnaires de mise en file d'attente simples, sans classes

Comme nous l'avons déjà dit, la gestion de mise en file d'attente permet de modifier la façon dont les données sont envoyées. Les gestionnaires de mise en file d'attente sans classes sont ceux qui, en gros, acceptent les données et qui ne font que les réordonner, les retarder ou les jeter.

Ils peuvent être utilisés pour mettre en forme le trafic d'une interface sans aucune subdivision. Il est primordial que vous compreniez cet aspect de la mise en file d'attente avant de continuer sur les gestionnaires de mise en files d'attente basés sur des classes contenant d'autres gestionnaires de mise en file d'attente.

Le gestionnaire le plus largement utilisé est de loin *pfifo_fast*, qui est celui par défaut. Ceci explique aussi pourquoi ces fonctionnalités avancées sont si robustes. Elles ne sont rien de plus << qu'une autre file d'attente >>.

Chacune de ces files d'attente a ses forces et ses faiblesses. Toutes n'ont peut−être pas été bien testées.

9.2.1. *pfifo_fast*

Cette file d'attente, comme son nom l'indique : premier entré, premier sorti (*First In First Out*), signifie que les paquets ne subissent pas de traitements spéciaux. En fait, ce n'est pas tout à fait vrai. Cette file d'attente a trois << bandes >>. A l'intérieur de chacune de ces bandes, des règles FIFO s'appliquent. Cependant, tant qu'il y a un paquet en attente dans la bande 0, la bande 1 ne sera pas traitée. Il en va de même pour la bande 1 et la bande 2.

Le noyau prend en compte la valeur du champ Type de Service des paquets et prend soin d'insérer dans la bande 0 les paquets ayant le bit << délai minimum >> activé.

Ne pas confondre ce gestionnaire de mise en file d'attente sans classes avec celui basé sur des classes PRIO ! Bien qu'ils aient des comportements similaires, *pfifo_fast* ne possède pas de classes et vous ne pourrez pas y ajouter de nouveaux gestionnaires avec la commande **tc**.

9.2.1.1. Paramètres & usage

Vous ne pouvez pas configurer le gestionnaire *pfifo_fast*, dans la mesure où c'est celui par défaut. Voici sa configuration par défaut :

priomap

Détermine comment les priorités des paquets sont reliées aux bandes, telles que définies par le noyau. La relation est établie en se basant sur l'octet TOS du paquet, qui ressemble à ceci :

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

Les quatre bits TOS (le champ TOS) sont définis comme suit :

```
Binaire Décimal   Signification
-----------------------------------------
1000    8         Minimise le Délai (Minimize delay) (md)
0100    4         Maximalise le Débit (Maximize throughput) (mt)
0010    2         Maximalise la Fiabilité (Maximize reliability) (mr)
0001    1         Minimalise le Coût Monétaire (Minimize monetary cost) (mmc)
0000    0         Service Normal
```

Comme il y a 1 bit sur la droite de ces quatre bits, la valeur réelle du champ TOS est le double de la valeur des bits TOS. **`tcpdump -v -v`** fournit la valeur de tout le champ TOS, et non pas seulement la valeur des quatre bits. C'est la valeur que l'on peut voir dans la première colonne du tableau suivant :

```
TOS     Bits  Signification                     Priorité Linux    Bande
------------------------------------------------------------------------
0x0     0     Service Normal                    0 Best Effort     1
0x2     1     Minimise le Coût Monétaire (mmc)  1 Filler          2
0x4     2     Maximalise la Fiabilité (mr)      0 Best Effort     1
0x6     3     mmc+mr                            0 Best Effort     1
0x8     4     Maximalise le Débit (mt)          2 Masse           2
0xa     5     mmc+mt                            2 Masse           2
0xc     6     mr+mt                             2 Masse           2
0xe     7     mmc+mr+mt                         2 Masse           2
0x10    8     Minimise le Délai (md)            6 Interactive     0
0x12    9     mmc+md                            6 Interactive     0
0x14    10    mr+md                             6 Interactive     0
0x16    11    mmc+mr+md                         6 Interactive     0
0x18    12    mt+md                             4 Int. Masse      1
0x1a    13    mmc+mt+md                         4 Int. Masse      1
0x1c    14    mr+mt+md                          4 Int. Masse      1
0x1e    15    mmc+mr+mt+md                      4 Int. Masse      1
```

[NdT : par flux de masse (*bulk flow*), il faut entendre << gros flot de données transmises en continu >> comme un transfert FTP. A l'opposé, un flux interactif (*interactive flow*), correspond à celui généré par des requêtes SSH].

Beaucoup de nombres. La seconde colonne contient la valeur correspondante des quatre bits TOS, suivi de leur signification. Par exemple, 15 représente un paquet voulant un coût monétaire minimal, une fiabilité maximum, un débit maximum *ET* un délai minimum. J'appellerai ceci un << paquet Hollandais >>.

La quatrième colonne liste la manière dont le noyau Linux interprète les bits TOS, en indiquant à quelle priorité ils sont reliés.

La dernière colonne montre la carte des priorités par défaut. Sur la ligne de commande, la carte des priorités ressemble à ceci :

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

Ceci signifie , par exemple, que la priorité 4 sera reliée à la bande numéro 1. La carte des priorités vous permet également de lister des priorités plus grandes (> 7) qui ne correspondent pas à une relation avec le champ TOS, mais qui sont configurées par d'autres moyens.

Le tableau suivant provenant de la RFC 1349 (à lire pour plus de détails) indique comment les applications devraient configurer leurs bits TOS pour fonctionner correctement :

```
TELNET                    1000           (minimise le délai)
FTP
        Contrôle          1000           (minimise le délai)
        Données           0100           (maximalise le débit)

TFTP                      1000           (minimise le délai)

SMTP
        phase de commande 1000           (minimise le délai)
        phase DATA        0100           (maximalise le débit)

Domain Name Service
        requête UDP       1000           (minimise le délai)
        requête TCP       0000
        Transfert de Zone 0100           (maximalise le débit)

NNTP                      0001           (minimise le coût monétaire)

ICMP
        Erreurs           0000
        Requêtes          0000 (presque)
        Réponses         <même chose que requête> (presque)
```

txqueuelen

La longueur de cette file d'attente est fournie par la configuration de l'interface, que vous pouvez voir et configurer avec **ifconfig** et **ip**. Pour configurer la longueur de la file d'attente à *10*, exécuter : **`ifconfig eth0 txqueuelen 10`**

Vous ne pouvez pas configurer ce paramètre avec **tc** !

9.2.2. Filtre à seau de jetons (*Token Bucket Filter*)

Le *Token Bucket Filter* (TBF) est un gestionnaire de mise en file d'attente simple. Il ne fait que laisser passer les paquets entrants avec un débit n'excédant pas une limite fixée administrativement. L'envoi de courtes rafales de données avec un débit dépassant cette limite est cependant possible.

TBF est très précis, et peu gourmand du point de vue réseau et processeur. Considérez−le en premier si vous voulez simplement ralentir une interface !

L'implémentation TBF consiste en un tampon (seau), constamment rempli par des éléments virtuels d'information appelés jetons, avec un débit spécifique (débit de jeton). Le paramètre le plus important du tampon est sa taille, qui correspond au nombre de jetons qu'il peut stocker.

Chaque jeton entrant laisse sortir un paquet de données de la file d'attente de données et ce jeton est alors supprimé du seau. L'association de cet algorithme avec les deux flux de jetons et de données, nous conduit à trois scénarios possibles :

- Les données arrivent dans TBF avec un débit *EGAL* au débit des jetons entrants. Dans ce cas, chaque paquet entrant a son jeton correspondant et passe la file d'attente sans délai.
- Les données arrivent dans TBF avec un débit *PLUS PETIT* que le débit des jetons. Seule une partie des jetons est supprimée au moment où les paquets de données sortent de la file d'attente, de sorte que les jetons s'accumulent jusqu'à atteindre la taille du tampon. Les jetons libres peuvent être utilisés pour envoyer des données avec un débit supérieur au débit des jetons standard, si de courtes rafales de données arrivent.
- Les données arrivent dans TBF avec un débit *PLUS GRAND* que le débit des jetons. Ceci signifie que le seau sera bientôt dépourvu de jetons, ce qui provoque l'arrêt de TBF pendant un moment. Ceci s'appelle << une situation de dépassement de limite >> (*overlimit situation*). Si les paquets continuent à arriver, ils commenceront à être éliminés.

Le dernier scénario est très important, car il autorise la mise en forme administrative de la bande passante disponible pour les données traversant le filtre.

L'accumulation de jetons autorise l'émission de courtes rafales de données sans perte en situation de dépassement de limite, mais toute surcharge prolongée causera systématiquement le retard des paquets, puis leur rejet.

Notez que, dans l'implémentation réelle, les jetons correspondent à des octets, et non des paquets.

9.2.2.1. Paramètres & usage

Même si vous n'aurez probablement pas besoin de les changer, TBF a des paramètres. D'abord, ceux toujours disponibles sont :

limit or latency
: *Limit* est le nombre d'octets qui peuvent être mis en file d'attente en attendant la disponibilité de jetons. Vous pouvez également indiquer ceci d'une autre manière en configurant le paramètre *latency*, qui spécifie le temps maximal pendant lequel un paquet peut rester dans TBF. Ce dernier paramètre prend en compte la taille du seau, le débit, et s'il est configuré, le débit de crête (*peakrate*).

burst/buffer/maxburst
: Taille du seau, en octets. C'est la quantité maximale, en octets, de jetons dont on disposera simultanément. En général, plus les débits de mise en forme sont importants, plus le tampon doit être grand. Pour 10 Mbit/s sur plateforme Intel, vous avez besoin d'un tampon d'au moins 10 kilo−octets si vous voulez atteindre la limitation configurée !

  Si votre tampon est trop petit, les paquets pourront être rejetés car il arrive plus de jetons par top d'horloge que ne peut en contenir le tampon.

mpu
: Un paquet de taille nulle n'utilise pas une bande passante nulle. Pour ethernet, la taille minimale d'un paquet est de 64 octets. L'Unité Minimale de Paquet (*Minimun Packet Unit*) détermine le nombre minimal de jetons à utiliser pour un paquet.

rate
: Le paramètre de la vitesse. Voir les remarques au−dessus à propos des limites !

Si le seau contient des jetons et qu'il est autorisé à se vider, alors il le fait par défaut avec une vitesse infinie. Si ceci vous semble inacceptable, utilisez les paramètres suivants :

peakrate
: Si des jetons sont disponibles et que des paquets arrivent, ils sont immédiatement envoyés par défaut ; et pour ainsi dire à << la vitesse de la lumière >>. Cela peut ne pas vous convenir, spécialement si vous avez un grand seau.

  Le débit de crête (*peak rate*) peut être utilisé pour spécifier la vitesse à laquelle le seau est autorisé à se vider. Si tout se passe comme écrit dans les livres, ceci est réalisé en libérant un paquet, puis en attendant suffisamment longtemps, pour libérer le paquet suivant. Le temps d'attente est calculé de manière à obtenir un débit égal au débit de crête.

  Cependant, étant donné que la résolution du minuteur (*timer*) d'UNIX est de 10 ms et que les paquets ont une taille moyenne de 10 000 bits, nous sommes limités à un débit de crête de 1mbit/s !

mtu/minburst
: Le débit de crête de 1Mb/s ne sert pas à grand chose si votre débit habituel est supérieur à cette valeur. Un débit de crête plus élevé peut être atteint en émettant davantage de paquets par top du minuteur, ce qui a pour effet de créer un second seau.

  Ce second *bucket* ne prend par défaut qu'un seul paquet, et n'est donc en aucun cas un seau.

  Pour calculer le débit de crête maximum, multipliez le *mtu* que vous avez configuré par 100 (ou plus exactement par HZ, qui est égal à 100 sur Intel et à 1024 sur Alpha).

9.2.2.2. Configuration simple

Voici une configuration simple, mais *très* utile :

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

Pourquoi est−ce utile ? Si vous avez un périphérique réseau avec une grande file d'attente, comme un modem DSL ou un modem câble, et que le dialogue se fasse à travers une interface rapide, comme une interface ethernet, vous observerez que télécharger vers l'amont (*uploading*) dégrade complètement l'interactivité.

[NdT : *uploading* désigne une opération qui consiste à transférer des données ou des programmes stockés dans un ordinateur local vers un ordinateur distant à travers un réseau. La traduction officielle pour ce terme est << téléchargement vers l'amont >>. On parle alors de voie montante. Le *downloading* désigne l'opération inverse (transfert d'un hôte distant vers l'ordinateur local) et est traduit par << téléchargement >> ou << téléchargement vers l'aval >>. On parle alors de la voie descendante.]

Le téléchargement vers l'amont va en effet remplir la file d'attente du modem. Celle−ci est probablement *ENORME* car cela aide vraiment à obtenir de bon débit de téléchargement vers l'amont. Cependant, ceci n'est pas forcément ce que voulez. Vous ne voulez pas forcément avoir une file d'attente importante de manière à garder l'interactivité et pouvoir encore faire des choses pendant que vous envoyez des données.

La ligne de commande au−dessus ralentit l'envoi de données à un débit qui ne conduit pas à une mise en file d'attente dans le modem. La file d'attente réside dans le noyau Linux, où nous pouvons lui imposer une taille limite.

Modifier la valeur 220kbit avec votre vitesse de lien *REELLE* moins un petit pourcentage. Si vous avez un modem vraiment rapide, augmenter un peu le paramètre *burst*.

9.2.3. Mise en file d'attente stochastiquement équitable (*Stochastic Fairness Queueing*)

Stochastic Fairness Queueing (SFQ) est une implémentation simple de la famille des algorithmes de mise en file d'attente équitable. Cette implémentation est moins précise que les autres, mais elle nécessite aussi moins de calculs tout en étant presque parfaitement équitable.

Le mot clé dans SFQ est conversation (ou flux), qui correspond principalement à une session TCP ou un flux UDP. Le trafic est alors divisé en un grand nombre de jolies files d'attente FIFO : une par conversation. Le trafic est alors envoyé dans un tourniquet, donnant une chance à chaque session d'envoyer leurs données tour à tour.

Ceci conduit à un comportement très équitable et empêche qu'une seule conversation étouffe les autres. SFQ est appelé << Stochastic >> car il n'alloue pas vraiment une file d'attente par session, mais a un algorithme qui divise le trafic à travers un nombre limité de files d'attente en utilisant un algorithme de hachage.

A cause de ce hachage, plusieurs sessions peuvent finir dans le même seau, ce qui peut réduire de moitié les chances d'une session d'envoyer un paquet et donc réduire de moitié la vitesse effective disponible. Pour empêcher que cette situation ne devienne importante, SFQ change très souvent son algorithme de hachage pour que deux sessions entrantes en collision ne le fassent que pendant un nombre réduit de secondes.

Il est important de noter que SFQ n'est seulement utile que dans le cas où votre interface de sortie est vraiment saturée ! Si ce n'est pas le cas, il n'y aura pas de files d'attente sur votre machine Linux et donc, pas d'effets. Plus tard, nous décrirons comment combiner SFQ avec d'autres gestionnaires de mise en files d'attente pour obtenir le meilleur des deux mondes.

Configurer spécialement SFQ sur l'interface ethernet qui est en relation avec votre modem câble ou votre routeur DSL est vain sans d'autres mises en forme du trafic !

9.2.3.1. Paramètres & usage

SFQ est presque configuré de base :

perturb
: Reconfigure le hachage une fois toutes les *pertub* secondes. S'il n'est pas indiqué, le hachage se sera jamais reconfiguré. Ce n'est pas recommandé. 10 secondes est probablement une bonne valeur.

quantum
: Nombre d'octets qu'un flux est autorisé à retirer de la file d'attente avant que la prochaine file d'attente ne prenne son tour. Par défaut, égal à la taille maximum d'un paquet (MTU). Ne le configurez pas en−dessous du MTU !

9.2.3.2. Configuration simple

Si vous avez un périphérique qui a une vitesse identique à celle du lien et un débit réel disponible, comme un modem téléphonique, cette configuration aidera à promouvoir l'équité :

```
# 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)
```

Le nombre *800c* est un descripteur (*handle*) automatiquement assigné et *limit* signifie que *128* paquets peuvent attendre dans la file d'attente. Il y a *1024* << seaux de hachage >> disponibles pour la comptabilité, *128* pouvant être actifs à la fois (pas plus de paquets ne conviennent dans la file d'attente). Le hachage est reconfiguré toutes les 10 secondes.

9.3. Conseils pour le choix de la file d'attente

Pour résumer, ces files d'attente simples gèrent le trafic en réordonnant, en ralentissant ou en supprimant les paquets.

Les astuces suivantes peuvent vous aider à choisir la file d'attente à utiliser. Elles mentionnent certaines files d'attente décrites dans le chapitre *Gestionnaires de mise en file d'attente avancés*.

- Pour simplement ralentir le trafic sortant, utilisez le *Token Bucket Filter*. Il convient bien pour les énormes bandes passantes, si vous paramètrez en conséquence le seau.
- Si votre lien est vraiment saturé et que vous voulez être sûr qu'aucune session ne va accaparer la bande passante vers l'extérieur, utilisez le *Stochastical Fairness Queueing*.
- Si vous avez une grande dorsale et que vous voulez savoir ce que vous faîtes, considérez *Random Early Drop* (voir le chapitre *Gestionnaires de mise en file d'attente avancés*).
- Pour << mettre en forme >> le trafic entrant qui n'est pas transmis, utilisez la réglementation Ingress (*Ingress Policier*). La mise en forme du flux entrant est appelée << réglementation >> (*policing*) et non << mise en forme >> (*shaping*).
- Si vous transmettez le trafic, utilisez TBF sur l'interface vers laquelle vous transmettez les données. Si vous voulez mettre en forme un trafic pouvant sortir par plusieurs interfaces, alors le seul facteur commun est l'interface entrante. Dans ce cas, utilisez la réglementation Ingress.
- Si vous ne voulez pas mettre en forme le trafic, mais que vous vouliez voir si votre interface est tellement chargée qu'elle a dû mettre en file d'attente les données, utilisez la file d'attente *pfifo* (pas *pfifo_fast*). Elle n'a pas de bandes internes, mais assure le comptage de la taille de son accumulateur.
- Finalement, vous pouvez aussi faire de la << mise en forme sociale >>. La technologie n'est pas toujours capable de réaliser ce que vous voulez. Les utilisateurs sont hostiles aux contraintes techniques. Un mot aimable peut également vous aider à avoir votre bande passante correctement divisée !

9.4. Terminologie

Pour comprendre correctement des configurations plus compliquées, il est d'abord nécessaire d'expliquer quelques concepts. A cause de la complexité et de la relative jeunesse du sujet, beaucoup de mots différents sont utilisés par les personnes mais ils signifient en fait la même chose.

Ce qui suit est lâchement inspiré du texte `draft−ietf−diffserv−model−06.txt`, *An Informal Management Model for Diffserv Routers*. Il peut être trouvé à l'adresse http://www.ietf.org/internet−drafts/draft−ietf−diffserv−model−04.txt.

Lisez−le pour les définitions strictes des termes utilisés.

Gestionnaire de mise en file d'attente (Queueing Discipline)
: Un algorithme qui gère la file d'attente d'un périphérique, soit pour les données entrantes (*ingress*), soit pour les données sortantes (*egress*).

Gestionnaire de mise en file d'attente sans classes (Classless qdisc)
: Un gestionnaire de mise en file d'attente qui n'a pas de subdivisions internes configurables.

Gestionnaire de mise en file d'attente basé sur des classes (Classful qdisc)
: Un gestionnaire de mise en file d'attente basé sur des classes contient de multiples classes. Chacune de ces classes contient un gestionnaire de mise en file d'attente supplémentaire, qui peut encore être

basé sur des classes, mais ce n'est pas obligatoire. Si l'on s'en tient à la définition stricte, *pfifo_fast* *EST* basé sur des classes, dans la mesure où il contient trois bandes, qui sont en fait des classes. Cependant, d'un point de vue des perspectives de configuration pour l'utilisateur, il est sans classes dans la mesure où ces classes ne peuvent être modifiées avec l'outil **tc**.

Classes
: Un gestionnaire de mise en file d'attente peut avoir beaucoup de classes, chacune d'elles étant internes au gestionnaire. Chacune de ces classes peut contenir un gestionnaire de mise en file d'attente réel.

Classificateur (Classifier)
: Chaque gestionnaire de mise en file d'attente basé sur des classes a besoin de déterminer vers quelles classes il doit envoyer un paquet. Ceci est réalisé en utilisant le classificateur.

Filtre (Filter)
: La classification peut être réalisée en utilisant des filtres. Un filtre est composé d'un certain nombre de conditions qui, si elles sont toutes vérifiées, satisfait le filtre.

Ordonnancement (Scheduling)
: Un gestionnaire de mise en file d'attente peut, avec l'aide d'un classificateur, décider que des paquets doivent sortir plus tôt que d'autres. Ce processus est appelé ordonnancement (*scheduling*), et est réalisé par exemple par le gestionnaire *pfifo_fast* mentionné plus tôt. L'ordonnancement est aussi appelé << reclassement >> (*reordering*), ce qui peut prêter à confusion.

Mise en forme (Shaping)
: Le processus qui consiste à retarder l'émission des paquets sortants pour avoir un trafic conforme à un débit maximum configuré. La mise en forme est réalisée sur *egress*. Familièrement, rejeter des paquets pour ralentir le trafic est également souvent appelé Mise en forme.

Réglementation (Policing)
: Retarder ou jeter des paquets dans le but d'avoir un trafic restant en dessous d'une bande passante configurée. Dans Linux, la réglementation ne peut que jeter un paquet, et non le retarder dans la mesure où il n'y a pas de << file d'attente d'entrée >> (*ingress queue*).

Work−Conserving
: Un gestionnaire de mise en file d'attente *work−conserving* délivre toujours un paquet s'il y en a un de disponible. En d'autres termes, il ne retarde jamais un paquet si l'adaptateur réseau est prêt à l'envoyer (dans le cas du gestionnaire *egress*).

non−Work−Conserving
: Quelques gestionnaire de mise en files d'attente, comme par exemple le *Token Bucket Filter*, peuvent avoir besoin de maintenir un paquet pendant un certain temps pour limiter la bande passante. Ceci signifie qu'ils refusent parfois de libérer un paquet, bien qu'ils en aient un de disponible.

Maintenant que nous avons défini notre terminologie, voyons où tous ces élements sont situés.

```
              Programmes Utilisateurs
                   ^
                   |
   +---------------+-------------------------------------------+
   |               Y                                           |
   |    -------> Pile IP                                       |
   |   |              |                                        |
   |   |              Y                                        |
   |   |              Y                                        |
   |   ^              |                                        |
   |   |  / ----------> Transmission ->                        |
   |   ^ /                           |                         |
   |   |/                            Y                         |
   |   |                             |                         |
   |   ^                             Y            /-qdisc1-\   |
   |   |                          Classificateur /--qdisc2--\  |
--->->Gestionnaire de mise        de sortie      ---qdisc3---- |  ->
   |  en file d'attente           (Egress)       \__qdisc4__/  |
   |  d'entrée (Ingress)                          \-qdiscN_/   |
   |                                                           |
   +-----------------------------------------------------------+
```

Merci à Jamal Hadi Salim pour cette représentation ascii.

Le grand rectangle représente le noyau. La flèche la plus à gauche représente le trafic du réseau entrant dans votre machine. Celui−ci alimente alors le gestionnaire de mise en file d'attente Ingress qui peut appliquer des filtres à un paquet, et décider de le supprimer. Ceci est appelé << réglementation >> (*Policing*).

Ce processus a lieu très tôt, avant d'avoir beaucoup parcouru le noyau. C'est par conséquent un très bon endroit pour rejeter au plus tôt du trafic, sans pour autant consommer beaucoup de ressources CPU.

Si le paquet est autorisé à continuer, il peut être destiné à une application locale et, dans ce cas, il entre dans la couche IP pour être traité et délivré à un programme utilisateur. Le paquet peut également être transmis sans entrer dans une application et dans ce cas, être destiné à *egress*. Les programmes utilisateurs peuvent également délivrer des données, qui sont alors transmises et examinées par le classificateur *Egress*.

Là, il est examiné et mis en file d'attente vers un certain nombre de gestionnaire de mise en file d'attente. Par défaut, il n'y a qu'un seul gestionnaire *egress* installé, *pfifo_fast*, qui reçoit tous les paquets. Ceci correspond à << la mise en file d'attente >> (*enqueueing*).

Le paquet réside maintenant dans le gestionnaire de mise en file d'attente, attendant que le noyau le réclame pour le transmettre à travers l'interface réseau. Ceci correspond au << retrait de la file d'attente >> (*dequeueing*).

Le schéma ne montre que le cas d'un seul adaptateur réseau. Les flèches entrantes et sortantes du noyau ne doivent pas être trop prises au pied de la lettre. Chaque adaptateur réseau a un gestionnaire d'entrée et de sortie.

9.5. Gestionnaires de file d'attente basés sur les classes

Les gestionnaires de mise en file d'attente basés sur des classes sont très utiles si vous avez différentes sortes de trafic qui doivent être traités différemment. L'un d'entre eux est appelé CBQ, pour *Class Based Queueing*. Il est si souvent mentionné que les personnes identifient les gestionnaires de mise en file d'attente basés sur des classes uniquement à CBQ, ce qui n'est pas le cas.

CBQ est le mécanisme le plus ancien, ainsi que le plus compliqué. Il n'aura pas forcément les effets que vous recherchez. Ceci surprendra peut−être ceux qui sont sous l'emprise de << l'effet Sendmail >>, qui nous enseigne qu'une technologie complexe, non documentée est forcément meilleure que toute autre.

Nous évoquerons bientôt, plus à propos, CBQ et ses alternatives.

9.5.1. Flux à l'intérieur des gestionnaires basés sur des classes & à l'intérieur des classes

Quand le trafic entre dans un gestionnaire de mise en file d'attente basé sur des classes, il doit être envoyé vers l'une de ses classes ; il doit être << classifié >>. Pour déterminer que faire d'un paquet, les élements appelés << filtres >> sont consultés. Il est important de savoir que les filtres sont appelés de l'intérieur d'un gestionnaire, et pas autrement !

Les filtres attachés à ce gestionnaire renvoient alors une décision que le gestionnaire utilise pour mettre en file d'attente le paquet vers l'une des classes. Chaque sous−classe peut essayer d'autres filtres pour voir si de nouvelles instructions s'appliquent. Si ce n'est pas le cas, la classe met le paquet en file d'attente dans le gestionnaire de mise en file d'attente qu'elle contient.

En plus de contenir d'autres gestionnaires, la plupart des gestionnaires de mise en file d'attente basés sur des classes réalisent également de la mise en forme. Ceci est utile pour réaliser à la fois l'ordonnancement (avec SFQ, par exemple) et le contrôle de débit. Vous avez besoin de ceci dans les cas où vous avez une interface à haut débit (ethernet, par exemple) connectée à un périphérique plus lent (un modem câble).

Si vous n'utilisez que SFQ, rien ne devait se passer, dans la mesure où les paquets entrent et sortent du routeur sans délai : l'interface de sortie est de loin beaucoup plus rapide que la vitesse réelle de votre liaison ; il n'y a alors pas de files d'attente à réordonnancer.

9.5.2. La famille des gestionnaires de mise en file d'attente : racines, descripteurs, descendances et parents

Chaque interface à << un gestionnaire de mise en file d'attente racine >> de sortie (*egress root qdisc*). Par défaut, le gestionnaire de mise en file d'attente sans classes mentionné plus tôt *pfifo_fast*. Chaque gestionnaire peut être repéré par un descripteur (*handle*), qui pourra être utilisé par les prochaines déclarations de configuration pour se référer à ce gestionnaire. En plus du gestionnaire de sortie, une interface peut également avoir un gestionnaire d'entrée (*ingress*), qui réglemente le trafic entrant.

Ces descripteurs sont constitués de deux parties : un nombre majeur et un nombre mineur. Il est habituel de nommer le gestionnaire racine *1:*, ce qui est équivalent à *1:0*. Le nombre mineur d'un gestionnaire de mise en file d'attente est toujours 0.

Les classes doivent avoir le même nombre majeur que leur parent.

9.5.2.1. Comment les filtres sont utilisés pour classifier le trafic

Pour récapituler, une hiérarchie typique pourrait ressembler à ceci :

```
                  racine 1:
                      |
                    _1:1_
                   /  |  \
                  /   |   \
                 /    |    \
               10:   11:   12:
              /   \       /   \
           10:1  10:2   12:1  12:2
```

Mais ne laissez pas cet arbre vous abuser ! Vous ne devriez *pas* imaginer le noyau être au sommet de l'arbre et le réseau en−dessous, ce qui n'est justement pas le cas. Les paquets sont mis et retirés de la file d'attente à la racine du gestionnaire, qui est le seul élément avec lequel le noyau dialogue.

Un paquet pourrait être classifié à travers une chaîne suivante :

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

Le paquet réside maintenant dans la file d'attente du gestionnaire attaché à la classe 12:2. Dans cet exemple, un filtre a été attaché à chaque noeud de l'arbre, chacun choisissant la prochaine branche à prendre. Cela est réalisable. Cependant, ceci est également possible :

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

Dans ce cas, un filtre attaché à la racine a décidé d'envoyer le paquet directement à *12:2*.

9.5.2.2. Comment les paquets sont retirés de la file d'attente et envoyés vers le matériel

Quand le noyau décide qu'il doit extraire des paquets pour les envoyer vers l'interface, le gestionnaire racine *1:* reçoit une requête *dequeue*, qui est transmise à *1:1* et qui, à son tour, est passée à *10:*, *11:* et *12:*, chacune interrogeant leurs descendances qui essaient de retirer les paquets de leur file d'attente. Dans ce cas, le noyau doit parcourir l'ensemble de l'arbre, car seul *12:2* contient un paquet.

En résumé, les classes << emboîtées >> parlent *uniquement*à leur gestionnaire de mise en file d'attente parent ; jamais à une interface. Seul la file d'attente du gestionnaire racine est vidée par le noyau !

Ceci a pour résultat que les classes ne retirent jamais les paquets d'une file d'attente plus vite que ce que leur parent autorise. Et c'est exactement ce que nous voulons : de cette manière, nous pouvons avoir SFQ dans une classe interne qui ne fait pas de mise en forme, mais seulement de l'ordonnancement, et avoir un gestionnaire de mise en file d'attente extérieur qui met en forme le trafic.

9.5.3. Le gestionnaire de mise en file d'attente PRIO

Le gestionnaire de mise en file d'attente ne met pas vraiment en forme le trafic ; il ne fait que le subdiviser en se basant sur la manière dont vous avez configuré vos filtres. Vous pouvez considérer les gestionnaires PRIO comme une sorte de super *pfifo_fast* dopé, où chaque bande est une classe séparée au lieu d'une simple FIFO.

Quand un paquet est mis en file d'attente dans le gestionnaire PRIO, une classe est choisie en fonction des filtres que vous avez donnés. Par défaut, trois classes sont créées. Ces classes contiennent par défaut de purs gestionnaires de mise en file d'attente FIFO sans structure interne, mais vous pouvez les remplacer par n'importe quels gestionnaires disponibles.

Chaque fois qu'un paquet doit être retiré d'une file d'attente, la classe *:1* est d'abord testée. Les classes plus élevées ne sont utilisées que si aucune des bandes plus faibles n'a pas fourni de paquets.

Cette file d'attente est très utile dans le cas où vous voulez donner la priorité à certains trafics en utilisant toute la puissance des filtres **tc** et en ne se limitant pas seulemenent aux options du champ TOS. Il peut également contenir n'importe quel gestionnaire de mise en file d'attente, tandis que *pfifo_fast* est limité aux gestionnaires simples FIFO.

Puisqu'il ne met pas vraiment en forme, on applique le même avertissement que pour SFQ. Utilisez PRIO seulement si votre lien physique est vraiment saturé ou intégrez−le à l'intérieur d'un gestionnaire de mise en file d'attente basé sur des classes qui réalisent la mise en forme. Ce dernier cas est valable pour pratiquement tous les modems−câbles et les périphériques DSL.

En termes formels, le gestionnaire de mise en file d'attente PRIO est un ordonnanceur *Work−Conserving*.

9.5.3.1. Paramètres PRIO & usage

Les paramètres suivants sont reconnus par **tc** :

bands
: Nombre de bandes à créer. Chaque bande est en fait une classe. Si vous changez ce nombre, vous devez également changer :

priomap
: Si vous ne fournissez pas de filtres **tc** pour classifier le trafic, le gestionnaire PRIO regarde la priorité *TC_PRIO* pour décider comment mettre en file d'attente le trafic.

: Ceci fonctionne comme le gestionnaire de mise en file d'attente *pfifo_fast* mentionné plus tôt. Voir la section correspondante pour plus de détails.

Les bandes sont des classes et sont appelées par défaut majeur:1 à majeur:3. Donc, si votre gestionnaire de mise en file d'attente est appelé *12:*, **tc** filtre le trafic vers *12:1* pour lui accorder une plus grande priorité.

Par itération, la bande 0 correspond au nombre mineur 1, la bande 1 au nombre mineur 2, etc ...

9.5.3.2. Configuration simple

Nous allons créer cet arbre :

```
     racine 1: prio
         /   |   \
       1:1  1:2  1:3
        |    |    |
       10:  20:  30:
       sfq  tbf  sfq
bande   0    1    2
```

Le trafic de masse ira vers *30:* tandis que le trafic interactif ira vers *20:* ou *10:*.

Les lignes de commande :

```
# tc qdisc add dev eth0 root handle 1: prio
## Ceci crée *instantanément* les classes 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
```

Regardons maintenant ce que nous avons créé :

```
# 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)
```

Comme vous pouvez le voir, la bande 0 a déjà reçu du trafic, et un paquet a été envoyé pendant l'exécution de cette commande !

Nous allons maintenant générer du trafic de masse avec un outil qui configure correctement les options TOS, et regarder de nouveau :

```
# 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)
```

Comme vous pouvez le voir, tout le trafic a été envoyé comme prévu vers le descripteur *30:*, qui est la bande de plus faible priorité. Maintenant, pour vérifier que le trafic interactif va vers les bandes de plus grande priorité, nous générons du trafic interactif :

```
# 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)
```

Ca a marché. Tout le trafic supplémentaire a été vers *10:*, qui est notre gestionnaire de plus grande priorité. Aucun trafic n'a été envoyé vers les priorités les plus faibles, qui avaient reçu au préalable tout le trafic venant de notre **scp**.

9.5.4. Le célèbre gestionnaire de mise en file d'attente CBQ

Comme dit avant, CBQ est le gestionnaire de mise en file d'attente disponible le plus complexe, celui qui a eu le plus de publicité, qui est le moins compris et qui est probablement le plus farceur lors de sa mise au point. Ce n'est pas parce que les auteurs sont mauvais ou incompétents, loin de là, mais l'algorithme CBQ n'est pas remarquablement précis et il ne correspond pas vraiment à la façon dont Linux fonctionne.

En plus d'être basé sur des classes, CBQ sert également à la mise en forme de trafic et c'est sur cet aspect qu'il ne fonctionne pas très bien. Il travaille comme ceci : si vous essayez de mettre en forme une connexion de 10mbit/s à 1mbits/s, le lien doit être inactif 90% du temps. Si ce n'est pas le cas, nous devons limiter le taux de sorte qu'il *soit* inactif 90% du temps.

Ceci est assez dur à mesurer et c'est pour cette raison que CBQ déduit le temps d'inactivité du nombre de microsecondes qui s'écoulent entre les requêtes de la couche matérielle pour avoir plus de données. Cette combinaison peut être utilisée pour évaluer si le lien est chargé ou non.

Ceci est plutôt léger et l'on arrive pas toujours à des résultats convenables. Par exemple, qu'en est−il de la vitesse de liaison réelle d'une interface qui n'est pas capable de transmettre pleinement les données à 100mbit/s, peut−être à cause d'un mauvais pilote de périphérique ? Une carte réseau PCMCIA ne pourra jamais atteindre 100mbit/s à cause de la conception du bus. De nouveau, comment calculons−nous le temps d'inactivité ?

Cela devient même pire quand on considère un périphérique réseau "pas−vraiment−réel" comme *PPP Over Ethernet* ou *PPTP over TCP/IP*. La largeur de bande effective est dans ce cas probablement déterminée par l'efficacité des tubes vers l'espace utilisateur, qui est énorme.

Les personnes qui ont effectué des mesures ont découvert que CBQ n'est pas toujours très exact, et parfois même, très éloigné de la configuration.

Cependant, il marche bien dans de nombreuses circonstances. Avec la documentation fournie ici, vous devriez être capable de le configurer pour qu'il fonctionne bien dans la plupart des cas.

9.5.4.1. Mise en forme CBQ en détail

Comme dit précédemment, CBQ fonctionne en s'assurant que le lien est inactif juste assez longtemps pour abaisser la bande passante réelle au débit configuré. Pour réaliser cela, il calcule le temps qui devrait s'écouler entre des paquets de taille moyennne.

En cours de fonctionnement, le temps d'inactivité effectif (*the effective idletime*) est mesuré en utilisant l'algorithme EWMA (*Exponential Weighted Moving Average*), qui considère que les paquets récents sont exponentiellement plus nombreux que ceux passés. La charge moyenne UNIX (*UNIX loadaverage*) est calculée de la même manière.

Le temps d'inactivité calculé est soustrait à celui mesuré par EWMA et le nombre résultant est appelé *avgidle*. Un lien parfaitement chargé a un *avgidle* nul : un paquet arrive à chaque intervalle calculé.

Une liaison surchargée a un *avgidle* négatif et s'il devient trop négatif, CBQ s'arrête un moment et se place alors en dépassement de limite (*overlimit*).

Inversement, un lien inutilisé peut accumuler un *avgidle* énorme, qui autoriserait alors des bandes passantes infinies après quelques heures d'inactivité. Pour éviter cela, *avgidle* est borné à *maxidle*.

En situation de dépassement de limite, CBQ peut en théorie bloquer le débit pour une durée équivalente au temps qui doit s'écouler entre deux paquets moyens, puis laisser passer un paquet et bloquer de nouveau le débit. Regardez cependant le paramètre *minburst* ci−dessous.

Voici les paramètres que vous pouvez spécifier pour configurer la mise en forme :

avpkt
: Taille moyenne d'un paquet mesurée en octets. Nécessaire pour calculer *maxidle*, qui dérive de *maxburst*, qui est spécifié en paquets.

bandwidth
: La bande passante physique de votre périphérique nécessaire pour les calculs du temps de non utilisation (*idle time*).

cell
: La durée de transmission d'un paquet n'augmente pas nécessairement de manière linéaire en fonction de sa taille. Par exemple, un paquet de 800 octets peut être transmis en exactement autant de temps qu'un paquet de 806 octets. Ceci détermine la granularité. Cette valeur est généralement positionnée à *8*, et doit être une puissance de deux.

maxburst
: Ce nombre de paquets est utilisé pour calculer *maxidle* de telle sorte que quand *avgidle* est égal à *maxidle*, le nombre de paquets moyens peut être envoyé en rafale avant que *avgidle* ne retombe à 0. Augmentez−le pour être plus tolérant vis à vis des rafales de données. Vous ne pouvez pas configurer *maxidle* directement, mais seulement via ce paramètre.

minburst
: Comme nous l'avons déjà indiqué, CBQ doit bloquer le débit dans le cas d'un dépassement de limite. La solution idéale est de le faire pendant exactement le temps d'inutilisation calculé, puis de laisser passer un paquet. Cependant, les noyaux UNIX ont généralement du mal à prévoir des événements plus courts que 10 ms, il vaut donc mieux limiter le débit pendant une période plus longue, puis envoyer *minburst* paquets d'un seul coup et dormir pendant une durée de *minburst*.

  Le temps d'attente est appelé *offtime*. De plus grandes valeurs de *minburst* mènent à une mise en forme plus précise dans le long terme, mais provoquent de plus grandes rafales de données pendant des périodes de quelques millisecondes.

minidle

Si *avgidle* est inférieur à 0, nous sommes en dépassement de limite et nous devons attendre jusqu'à ce que *avgidle* devienne suffisamment important pour envoyer un paquet. Pour éviter qu'une brusque rafale de données n'empêche le lien de fonctionner pendant une durée prolongée, *avgidle* est remis à *minidle* s'il atteint une valeur trop basse.

La valeur *minidle* est spécifiée en microsecondes négatives : 10 signifie alors que *avgidle* est borné à −10µs.

mpu
: Taille minumum d'un paquet. Nécessaire car même un paquet de taille nulle est encapsulé par 64 octets sur ethernet et il faut donc un certain temps pour le transmettre. CBQ doit connaître ce paramètre pour calculer précisément le temps d'inutilisation.

rate
: Débit du trafic sortant du gestionnaire. Ceci est le << paramètre de vitesse >> !

En interne, CBQ est finement optimisé. Par exemple, les classes qui sont connues pour ne pas avoir de données présentes dans leur file d'attente ne sont pas interrogées. Les classes en situation de dépassement de limite sont pénalisées par la diminution de leur priorité effective. Tout ceci est très habile et compliqué.

9.5.4.2. Le comportement *CBQ classful*

En plus de la mise en forme, en utilisant les approximations *idletime* mentionnées ci−dessus, CBQ peut également agir comme une file d'attente PRIO dans le sens où les classes peuvent avoir différentes priorités. Les priorités de plus faible valeur seront examinées avant celles de valeur plus élevées.

Chaque fois qu'un paquet est demandé par la couche matérielle pour être envoyé sur le réseau, un processus *weighted round robin* (WRR) démarre en commençant par les classes de plus faibles priorités.

Celles−ci sont regroupées et interrogées si elles ont des données disponibles. Après qu'une classe ait été autorisée à retirer de la file d'attente un nombre d'octets, la classe de priorité suivante est consultée.

Les paramètres suivants contrôlent le processus WRR :

allot
: Quand le CBQ racine reçoit une demande d'envoi de paquets sur une interface, il va essayer tous les gestionnaires internes (dans les classes) tour à tour suivant l'ordre du paramètre *priority*. A chaque passage, une classe ne peut envoyer qu'une quantité limitée de données. Le paramètre *allot* est l'unité de base de cette quantité. Voir le paramètre *weight* pour plus d'informations.

prio
: CBQ peut également agir comme un périphérique PRIO. Les classes internes avec les priorités les plus faibles sont consultées en premier et, aussi longtemps qu'elles ont du trafic, les autres classes ne sont pas examinées.

weight
: Le paramètre *weight* assiste le processus *Weighted Round Robin*. Chaque classe a tour à tour la possibilité d'envoyer ses données. Si vous avez des classes avec des bandes passantes significativement plus importantes, il est logique de les autoriser à envoyer plus de données à chaque tour que les autres.

Vous pouvez utiliser des nombres arbitraires dans la mesure où CBQ additionne tous les paramètres *weight* présents sous une classe et les normalise. La règle empirique qui consiste à prendre *rate/10* semble fonctionner correctement. Le paramètre *weight* normalisé est multiplié par le paramètre *allot* pour déterminer la quantité de données à envoyer à chaque tour.

Notez, s'il vous plaît, que toutes les classes à l'intérieur d'une hiérarchie CBQ doivent avoir le même nombre majeur !

9.5.4.3. Paramètres CBQ qui déterminent le partage & le prêt du lien

En plus de purement limiter certains trafics, il est également possible de spécifier quelles classes peuvent emprunter de la bande passante aux autres classes ou, réciproquement, prêter sa bande passante.

isolated/ sharing
: Une classe qui est configurée avec *isolated* ne prêtera pas sa bande passante à ses classes enfants. Utilisez ceci si vous avez sur votre lien deux agences concurrentes ou qui ne s'apprécient pas et qui ne veulent pas se prêter gratuitement de la bande passante.

  Le programme de contrôle **tc** connait également *sharing*, qui agit à l'inverse du paramètre *isolated*.

bounded/ borrow
: Une classe peut aussi être bornée (*bounded*), ce qui signifie qu'elle n'essaiera pas d'emprunter de la bande passante à ses classes enfants. **tc** connait également *borrow*, qui agit à l'inverse de *bounded*.

Une situation typique pourrait être le cas où vous avez deux agences présentes sur votre lien qui sont à la fois *isolated* et *bounded*. Ceci signifie qu'elles sont strictement limitées à leur débit et qu'elles ne prêteront pas aux autres leur bande passante.

A l'intérieur de ces classes d'agence, il pourrait y avoir d'autres classes qui sont autorisées à échanger leur bande passante.

9.5.4.4. Configuration simple

Cette configuration limite le trafic d'un serveur web à 5 mbit et le trafic smtp à 3 mbit. Il est souhaitable qu'ils n'occupent pas plus de 6 mbit à eux deux. Nous avons une carte réseau à 100 mbit et les classes peuvent s'emprunter mutuellement de la bande passante.

```
# 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
```

Cette partie installe la racine et la classe *1:0* habituelle. La classe *1:1* est bornée, la bande passante totale ne pourra donc pas excéder 6 mbit.

Comme dit avant, CBQ a besoin de *NOMBREUX* paramètres. Tous ces paramètres sont cependant expliqués au−dessus. La configuration HTB correspondante est beaucoup plus simple.

```
# 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
```

Ce sont nos deux classes. Notez comment nous avons configuré la valeur du paramètre *weight* en fonction du paramètre *rate*. La bande passante de l'ensemble des deux classes ne pourra jamais dépasser 6 mbit. En fait, les identifieurs de classe (*classid*) doivent avoir le même numéro majeur que le parent CBQ !

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

Les deux classes ont par défaut un gestionnaire de mise en file d'attente FIFO. Nous les remplaçons par une file d'attente SFQ de telle sorte que chaque flux de données soit traité de manière égale.

```
# 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
```

Ces commandes, directement attachées à la racine, envoient le trafic vers le bon gestionnaire de mise en file d'attente.

Notez que nous utilisons **`tc class add`** pour *CREER* des classes à l'intérieur d'un gestionnaire de mise en file d'attente, et que nous utilisons **`tc qdisc add`** pour véritablement configurer ces classes.

Vous vous demandez peut−être ce qui arrive au trafic qui n'est classifié par aucune des deux règles. Dans ce cas, les données seront traitées à l'intérieur de *1:0*, et le débit ne sera pas limité.

Si le trafic smtp+web tente de dépasser la limite de 6 mbit/s, la bande passante sera divisée selon le paramètre *weight*, donnant 5/8 du trafic au serveur web et 3/8 au serveur smtp.

Avec cette configuration, vous pouvez également dire que le trafic du serveur web sera au minimum de 5/8 * 6 mbit = 3.75 mbit.

9.5.4.5. D'autres paramètres CBQ : *split* & *defmap*

Comme précisé avant, un gestionnaire de mise en file d'attente basé sur des classes doit appeler des filtres pour déterminer dans quelle classe un paquet sera mis en file d'attente.

En plus d'appeler les filtres, CBQ offre d'autres options : *defmap* & *split*. C'est plutôt compliqué à comprendre et, de plus, ce n'est pas vital. Mais, étant donné que ceci est le seul endroit connu où *defmap* & *split* sont correctement expliqués, je vais faire de mon mieux.

Etant donné que nous voulons le plus souvent réaliser le filtrage en ne considérant que le champ TOS, une syntaxe spéciale est fournie. Chaque fois que CBQ doit trouver où le paquet doit être mis en file d'attente, il vérifie si le noeud est un noeud d'aiguillage (*split node*). Si c'est le cas, un de ses sous−gestionnaire a indiqué son souhait de recevoir tous les paquets configurés avec une certaine priorité. Celle ci peut être dérivée du champ TOS ou des options des sockets positionnées par les applications.

Les bits de priorités des paquets subissent un OU logique avec le champ *defmap* pour voir si une correspondance existe. En d'autres termes, c'est un moyen pratique de créer un filtre très rapide, qui ne sera actif que pour certaines priorités. Un *defmap* de *ff* (en hexadécimal) vérifiera tout tandis qu'une valeur de *0* ne vérifiera rien. Une configuration simple aidera peut−être à rendre les choses plus claires :

```
# 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
```

Préambule standard de CBQ. Je n'ai jamais pris l'habitude de la quantité de nombres nécessaires !

Le paramètre *defmap* se réfère aux bits TC_PRIO qui sont définis comme suit :

```
TC_PRIO..          Num  Correspond à TOS
-------------------------------------------------
BESTEFFORT         0    Maximalise la Fiabilité
FILLER             1    Minimalise le Coût
BULK               2    Maximalise le Débit (0x8)
INTERACTIVE_BULK   4
INTERACTIVE        6    Minimise le Délai (0x10)
CONTROL            7
```

Les nombres TC_PRIO.. correspondent aux bits comptés à partir de la droite. Voir la section *pfifo_fast* pour plus de détails sur la façon dont les bits TOS sont convertis en priorités.

Maintenant, les classes interactive et de masse :

```
# 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 gestion de mise en file d'attente d'aiguillage (*split qdisc*) est *1:0* et c'est à ce niveau que le choix sera fait. *C0* correspond au nombre binaire *11000000* et *3F* au nombre binaire *00111111*. Ces valeurs sont choisies de telle sorte qu'à elles deux, elles vérifient tous les bits. La première classe correspond aux bits 6 & 7, ce qui est équivalent aux trafics << interactif >> et de << contrôle >>. La seconde classe correspond au reste.

Le noeud *1:0* possède maintenant la table suivante :

```
priorité        envoyer à
0               1:3
1               1:3
2               1:3
3               1:3
4               1:3
5               1:3
6               1:2
7               1:2
```

Pour d'autres amusements, vous pouvez également donner un << masque de changement >> qui indique exactement les priorités que vous souhaitez changer. N'utilisez ceci qu'avec la commande **`tc class change`**. Par exemple, pour ajouter le trafic *best effort* à la classe *1:2*, nous devrons exécuter ceci :

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

La carte des priorités au niveau de *1:0* ressemble maintenant à ceci :

```
priorité        envoyer à
0               1:2
1               1:3
2               1:3
3               1:3
4               1:3
5               1:3
6               1:2
7               1:2
```

FIXME: **`tc class change`** n'a pas été testé, mais simplement vu dans les sources.

9.5.5. Seau de jetons à contrôle hiérarchique (*Hierarchical Token Bucket*)

Martin Devera(<devik>) réalisa à juste titre que CBQ est complexe et qu'il ne semble pas optimisé pour de nombreuses situations classiques. Son approche hiérarchique est bien adaptée dans le cas de configurations où il y a une largeur de bande passante fixée à diviser entre différents éléments. Chacun de ces éléments aura une bande passante garantie, avec la possibilité de spécifier la quantité de bande passante qui pourra être empruntée.

HTB travaille juste comme CBQ, mais il n'a pas recourt à des calculs de temps d'inoccupation pour la mise en forme. A la place, c'est un *Token Bucket Filter* basé sur des classes, d'où son nom. Il n'a que quelques paramètres, qui sont bien documentés sur ce site.

Au fur et à mesure que votre configuration HTB se complexifie, votre configuration s'adapte bien. Avec CBQ, elle est déjà complexe même dans les cas simples ! HTB ne fait pas encore partie du noyau standard, mais cela devrait bientôt être le cas !

Si vous êtes sur le point de mettre à jour votre noyau, considérez HTB coûte que coûte.

9.5.5.1. Configuration simple

Fonctionnellement presque identique à la configuration simple CBQ présentée ci−dessus :

```
# 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
```

L'auteur recommande SFQ sous ces classes :

```
# 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
```

Ajouter les filtres qui dirigent le trafic vers les bonnes classes :

```
# 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
```

Et, c'est tout. Pas de vilains nombres non expliqués, pas de paramètres non documentés.

HTB semble vraiment merveilleux. Si *10:* et *20:* ont atteint tous les deux leur bande passante garantie et qu'il en reste à partager, ils l'empruntent avec un rapport de 5:3, comme attendu.

Le trafic non classifié est acheminé vers *30:*, qui a une petite bande passante, mais qui peut emprunter tout ce qui est laissé libre. Puisque nous avons choisi SFQ en interne, on hérite naturellement de l'équité.

9.6. Classifier des paquets avec des filtres

Pour déterminer quelle classe traitera un paquet, la << chaîne de classificateurs >> est appelée chaque fois qu'un choix a besoin d'être fait. Cette chaîne est constituée de tous les filtres attachés aux gestionnaires de mise en file d'attente basés sur des classes qui doivent prendre une décision.

On reprend l'arbre qui n'est pas un arbre :

```
                    root 1:
                      |
                    _1:1_
                   /  |  \
                  /   |   \
                 /    |    \
               10:   11:   12:
              /   \       /   \
           10:1  10:2   12:1  12:2
```

Quand un paquet est mis en file d'attente, l'instruction appropriée de la chaîne de filtre est consultée à chaque branche. Une configuration typique devrait avoir un filtre en *1:1* qui dirige le paquet vers *12:* et un filtre en *12:* qui l'envoie vers *12:2*.

Vous pourriez également avoir ce dernier filtre en *1:1*, mais vous pouvez gagner en efficacité en ayant des tests plus spécifiques plus bas dans la chaîne.

A ce propos, vous ne pouvez pas filtrer un paquet << vers le haut >>. Donc, avec HTB, vous devrez attacher tous les filtres à la racine !

Encore une fois, les paquets ne sont mis en file d'attente que vers le bas ! Quand ils sont retirés de la file d'attente, ils montent de nouveau, vers l'interface. Ils ne tombent *PAS* vers l'extrémité de l'arbre en direction de l'adaptateur réseau !

9.6.1. Quelques exemples simples de filtrage

Comme expliqué dans le chapitre *Filtres avancés pour la classification des paquets*, vous pouvez vraiment analyser n'importe quoi en utilisant une syntaxe très compliquée. Pour commencer, nous allons montrer comment réaliser les choses évidentes, ce qui heureusement est plutôt facile.

Disons que nous avons un gestionnaire de mise en file d'attente PRIO appelé *10:* qui contient trois classes, et que nous voulons assigner à la bande de plus haute priorité tout le trafic allant et venant du port 22. Les filtres seraient les suivants :

```
# 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
```

Qu'est–ce que cela signifie ? Cela dit : attacher à *eth0*, au noeud *10:* un filtre *u32* de priorité 1 qui analyse le port de destination ip 22 et qui l'envoie vers la bande *10:1*. La même chose est répétée avec le port source 80. La dernière commande indique que si aucune correspondance n'est trouvée, alors le trafic devra aller vers la bande *10:2*, la plus grande priorité suivante.

Vous devez ajouter *eth0* ou n'importe laquelle de vos interfaces, car chaque interface possède un espace de nommage de ses descripteurs qui lui est propre.

Pour sélectionner une adresse IP, utilisez ceci :

```
# 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
```

Ceci dirige le trafic allant vers *4.3.2.1* et venant de *1.2.3.4* vers la file d'attente de plus haute priorité, tandis que le reste ira vers la prochaine plus haute priorité.

Vous pouvez rassembler ces deux vérifications pour récupérer le trafic venant de *1.2.3.4* avec le port source 80 :

```
# 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. Toutes les commandes de filtres dont vous aurez normalement besoin

La plupart des commandes présentées ici commencent avec le préambule suivant :

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

Ils sont appelés filtres *u32* qui analysent *N'IMPORTE QUELLE* partie d'un paquet.

Sur l'adresse source/destination
: Masque pour la source *match ip src 1.2.3.0/24* et masque pour la destination *match ip dst 4.3.2.0/24*. Pour analyser un hôte simple, employez /32 ou omettez le masque.

Sur le port source/destination, tous les protocoles IP
: Source: *match ip sport 80 0xffff* et destination : *match ip dport ?? 0xffff*

Sur le protocole ip (tcp, udp, icmp, gre, ipsec)
: Utilisez les nombres définis dans `/etc/protocols`, par exemple 1 pour icmp : *match ip protocol 1 0xff*.

Sur fwmark
: Vous pouvez marquer les paquets avec ipchains, par exemple, et voir cette marque préservée lors du routage à travers les interfaces. Ceci est vraiment utile pour mettre uniquement en forme le trafic sur *eth1* et venant de *eth0*, par exemple. La syntaxe est la suivante :

```
# tc filter add dev eth1 protocol ip parent 1:0 prio 1 handle 6 fw flowid 1:1
```

Notez que ce n'est pas une correspondance *u32* !

Vous pouvez positionner une marque comme ceci :

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

Le nombre 6 est arbitraire.

Si vous ne voulez pas assimiler la syntaxe complète de **tc filter**, utilisez juste **iptables** et apprenez seulement la sélection basée sur fwmark.

Sur le champ TOS
: Pour sélectionner le trafic interactif, délai minimum :

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

Utilisez *0x08 0xff* pour le trafic de masse.

Pour plus de commandes de filtrage, voir le chapitre *Filtres avancés pour la classification des paquets*.

Chapitre 10. Équilibrage de charge sur plusieurs interfaces

Il existe plusieurs manières pour le faire. Une des plus faciles et des plus directes est TEQL (*True (or Trivial) Link Equalizer*. Comme la plupart des éléments en relation avec la gestion de file d'attente, l'équilibrage de charge est bidirectionnel. Les deux équipements terminaux du lien ont besoin de participer pour obtenir une efficacité optimale.

Imaginez la situation suivante :

```
                 +-------+   eth1   +-------+
                 |       |==========|       |
 'réseau 1' -----|   A   |          |   B   |---- 'réseau 2'
                 |       |==========|       |
                 +-------+   eth2   +-------+
```

A et *B* sont des routeurs dont nous supposerons qu'ils fonctionnent avec Linux pour le moment. Si le trafic va du réseau 1 vers le réseau 2, le routeur *A* a besoin de distribuer les paquets sur les deux liens allant vers *B*. Le routeur *B* a besoin d'être configuré pour l'accepter. On retrouve la même chose dans le sens inverse, pour les paquets allant du réseau 2 vers le réseau 1. Le routeur *B* a besoin d'envoyer les paquets à la fois sur *eth1* et *eth2*.

La répartition est faite par un périphérique TEQL, comme ceci (cela ne pouvait pas être plus simple) :

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

N'oubliez pas la commande **ip link set up** !

Ceci a besoin d'être fait sur les deux hôtes. Le périphérique *teql0* est basiquement un distributeur tourniquet au dessus de *eth1* et *eth2* pour l'envoi des paquets. Aucune donnée n'arrive jamais à travers un périphérique *teql*, mais les données apparaîssent sur *eth1* et *eth2*.

Nous n'avons pour le moment que les périphériques et nous avons également besoin d'un routage correct. L'une des possibilités pour réaliser cela est d'assigner un réseau */31* sur chacun des liens, ainsi que sur le périphérique *teql0* :

FIXME: Avons nous besoin de quelque chose comme *nobroadcast* ? Un */31* est trop petit pour contenir une adresse réseau et une adresse de diffusion. Si cela ne marche pas comme prévu, essayez un */30*, et ajustez les adresses IP. Vous pouvez même essayer sans attribuer d'adresses à *eth1* et *eth2*.

Sur le routeur 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
```

Sur le routeur 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
```

Le routeur *A* devrait maintenant être capable de lancer un **ping** vers *10.0.0.1*, *10.0.0.3* et *10.0.0.5* à travers les deux liens physiques et le périphérique << égalisé >>. Le routeur *B* devrait maintenant être capable de lancer un **ping** vers *10.0.0.0*, *10.0.0.2* et *10.0.0.4* à travers les liens.

Si cela marche, le routeur *A* peut prendre *10.0.0.5* comme route vers le réseau 2 et le routeur *B 10.0.0.4* comme route vers le réseau 1. Pour le cas particulier où le réseau 1 est votre réseau personnel et où le réseau 2 est l'Internet, le routeur *A* peut prendre *10.0.0.5* comme passerelle par défaut.

10.1. Avertissement

Rien n'est aussi simple qu'il y paraît. Les interfaces *eth1* et *eth2* sur les deux routeurs *A* et *B* ne doivent pas avoir la fonction de filtrage par chemin inverse activée. Dans le cas contraire, ils rejeteront les paquets destinés à des adresses autres que les leurs :

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

Il y a un sérieux problème avec le réordonnancement des paquets. Supposons que six paquets aient besoin d'être envoyés de *A* vers *B*. Par exemple, *eth1* peut traiter les paquets 1, 3 et 5 et *eth2* les paquets 2, 4 et 6. Dans un monde idéal, le routeur *B* devrait recevoir ces paquets dans l'ordre 1, 2, 3, 4, 5, 6. Mais il est plus probable que le noyau les reçoive comme ceci : 2, 1, 4, 3, 6, 5. Ce problème va perturber TCP/IP. Alors qu'il n'y a pas de problèmes pour les liens transportant différentes sessions TCP/IP, vous ne serez pas capable de regrouper plusieurs liens et obtenir par ftp un simple fichier beaucoup plus rapidement, à moins que le système d'exploitation envoyant ou recevant ne soit Linux. En effet, celui−ci n'est pas facilement perturbé par de simples réordonnancements.

Cependant, l'équilibrage de charge est une bonne idée pour de nombreuses applications.

Chapitre 11. Netfilter et iproute − marquage de paquets

Jusqu'à maintenant, nous avons vu comment iproute travaille, et netfilter a été mentionné plusieurs fois. Vous ne perdrez pas votre temps à consulter Rusty's Remarkably Unreliable Guides. Le logiciel Netfilter peut être trouvé ici.

Netfilter nous permet de filtrer les paquets ou de désosser leurs en−têtes. Une de ses fonctionnalités particulières est de pouvoir marquer un paquet avec un nombre, grâce à l'option *−−set−mark*.

Comme exemple, la commande suivante marque tous les paquets destinés au port 25, en l'occurrence le courrier sortant.

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

Disons que nous avons plusieurs connexions, une qui est rapide (et chère au mégaoctet) et une qui est plus lente, mais avec un tarif moins élevé. Nous souhaiterions que le courrier passe par la route la moins chère.

Nous avons déjà marqué le paquet avec un "1" et nous allons maintenant renseigner la base de données de la politique de routage pour qu'elle agisse sur ces paquets marqués.

```
# echo 201 mail.out >> /etc/iproute2/rt_tables
# ip rule add fwmark 1 table mail.out
# ip rule ls
```

```
0:      from all lookup local
32764:  from all fwmark        1 lookup mail.out
32766:  from all lookup main
32767:  from all lookup default
```

Nous allons maintenant générer la table mail.out avec une route vers la ligne lente, mais peu coûteuse.

```
# /sbin/ip route add default via 195.96.98.253 dev ppp0 table mail.out
```

Voilà qui est fait. Il se peut que nous voulions mettre en place des exceptions, et il existe de nombreux moyens pour le faire. Nous pouvons modifier la configuration de netfilter pour exclure certains hôtes ou nous pouvons insérer une règle avec une priorité plus faible qui pointe sur la table principale pour nos hôtes faisant exception.

Nous pouvons aussi utiliser cette fonctionnalité pour nous conformer aux bits TOS en marquant les paquets avec différents types de service et les nombres correspondants. On crée ensuite les règles qui agissent sur ces types de service. De cette façon, on peut dédier une ligne RNIS aux connexions interactives.

Inutile de le dire, cela marche parfaitement sur un hôte qui fait de la translation d'adresse (NAT), autrement dit du *masquerading*.

IMPORTANT : Nous avons reçu une information selon laquelle MASQ et SNAT entrent en conflit avec le marquage de paquets. Rusty Russell l'explique dans ce courrier.

Désactivez le filtrage de chemin inverse pour que cela fonctionne correctement.

Note : pour marquer les paquets, vous aurez besoin de valider quelques options du noyau :

```
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/?]
```

Voir aussi Section 15.5 dans le chapitre *Recettes de cuisine*.

Chapitre 12. Filtres avancés pour la (re−)classification des paquets

Comme expliqué dans la section sur les gestionnaires de mise en file d'attente basés sur des classes, les filtres sont nécessaires pour classifier les paquets dans n'importe laquelle des sous−files d'attente. Ces filtres sont appelés à l'intérieur des gestionnaires de mise en file d'attente basés sur des classes.

Voici une liste incomplète des classificateurs disponibles :

fw
: Base la décision sur la façon dont le pare−feu a marqué les paquets. Ceci peut être un passage facile si vous ne voulez pas apprendre la syntaxe **tc** liée aux filtres. Voir le chapitre sur les gestionnaires de mise en file d'attente pour plus de détails.

u32
: Base la décision sur les champs à l'intérieur du paquet (c'est−à−dire l'adresse IP source, etc.)

route
: Base la décision sur la route que va emprunter le paquet.

rsvp, rsvp6
: Route les paquets en se basant sur RSVP. Seulement utile sur les réseaux que vous contrôlez. Internet ne respecte pas RSVP.

tcindex
: Utilisé par le gestionnaire de file d'attente *DSMARK*. Voir la section DSMARK.

Notez qu'il y a généralement plusieurs manières de classifier un paquet. Cela dépend du système de classification que vous souhaitez utiliser.

Les classificateurs acceptent en général quelques arguments communs. Ils sont listés ici pour des raisons pratiques :

protocol
: Le protocole que ce classificateur acceptera. Généralement, on n'acceptera que le trafic IP. Exigé.

parent
: Le descripteur auquel ce classificateur est attaché. Ce descripteur doit être une classe déjà existante. Exigé.

prio
: La priorité de ce classificateur. Les plus petits nombres seront testés en premier.

handle
: Cette référence a plusieurs significations suivant les différents filtres.

Toutes les sections suivantes supposeront que vous essayez de mettre en forme le trafic allant vers *HostA*. Ces sections supposeront que la classe racine a été configurée sur *1:* et que la classe vers laquelle vous voulez envoyer le trafic sélectionné est *1:1*.

12.1. Le classificateur *u32*

Le filtre *u32* est le filtre le plus avancé dans l'implémentation courante. Il est entièrement basé sur des tables de hachage, ce qui le rend robuste quand il y a beaucoup de règles de filtrage.

Dans sa forme la plus simple, le filtre *u32* est une liste d'enregistrements, chacun consistant en deux champs : un sélecteur et une action. Les sélecteurs, décrits ci−dessous, sont comparés avec le paquet IP traité jusqu'à la première correspondance, et l'action associée est réalisée. Le type d'action le plus simple serait de diriger le paquet vers une classe CBQ définie.

La ligne de commande du programme **tc filter**, utilisée pour configurer le filtre, consiste en trois parties : la spécification du filtre, un sélecteur et une action. La spécification du filtre peut être définie comme :

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

Le champ *protocol* décrit le protocole sur lequel le filtre sera appliqué. Nous ne discuterons que du cas du protocole *ip*. Le champ *preference* (*priority* peut être utilisé comme alternative) fixe la priorité du filtre que l'on définit. C'est important dans la mesure où vous pouvez avoir plusieurs filtres (listes de règles) avec des priorités différentes. Chaque liste sera scrutée dans l'ordre d'ajout des règles. Alors, la liste avec la priorité la plus faible (celle qui a le numéro de préférence le plus élevé) sera traitée. Le champ *parent* définit le sommet de l'arbre CBQ (par ex. *1:0*) auquel le filtre doit être attaché.

Les options décrites s'appliquent à tous les filtres, pas seulement à *u32*.

12.1.1. Le sélecteur U32

Le sélecteur U32 contient la définition d'un modèle, qui sera comparé au paquet traité. Plus précisément, il définit quels bits doivent correspondre dans l'en−tête du paquet, et rien de plus, mais cette méthode simple est très puissante. Jetons un oeil sur l'exemple suivant, directement tiré d'un filtre assez complexe réellement existant :

```
# tc filter parent 1: protocol ip pref 10 u32 fh 800::800 order 2048 key ht 800 bkt 0 flowid 1:3 \
  match 00100000/00ff0000 at 0
```

Pour l'instant, laissons de côté la première ligne ; tous ces paramètres décrivent les tables de hachage du filtre. Focalisons−nous sur la ligne de sélection contenant le mot−clé *match*. Ce sélecteur fera correspondre les en−têtes IP dont le second octet sera *0x10* (*0010*). Comme nous pouvons le deviner, le nombre *00ff* est le masque de correspondance, disant au filtre quels bits il doit regarder. Ici, c'est *0xff*, donc l'octet correspondra si c'est exactement 0x10. Le mot−clé *at* signifie que la correspondance doit démarrer au décalage spécifié (en octets) − dans notre cas, c'est au début du paquet. Traduisons tout cela en langage humain : le paquet correspondra si son champ Type de Service (TOS) a le bit << faible délai >> positionné. Analysons une autre règle :

```
# tc filter parent 1: protocol ip pref 10 u32 fh 800::803 order 2051 key ht 800 bkt 0 flowid 1:3 \
  match 00000016/0000ffff at nexthdr+0
```

L'option *nexthdr* désigne l'en−tête suivant encapsulé dans le paquet IP, c'est à dire celui du protocole de la couche supérieure. La correspondance commencera également au début du prochain en−tête. Elle devrait avoir lieu dans le deuxième mot de 32 bits de l'en−tête. Dans les protocoles TCP et UDP, ce champ contient le port de destination du paquet. Le nombre est donné dans le format big−endian, c'est−à−dire les bits les plus significatifs en premier. Il faut donc lire 0x0016 comme 22 en décimal, qui correspond au service SSH dans le cas de TCP. Comme vous le devinez, cette correspondance est ambiguë sans un contexte, et nous en discuterons plus loin.

Ayant compris tout cela, nous trouverons le sélecteur suivant très facile à lire : *match c0a80100/ffffff00 at 16*. Ce que nous avons ici, c'est une correspondance de trois octets au 17ème octet, en comptant à partir du début de l'en−tête IP. Cela correspond aux paquets qui ont une adresse de destination quelconque dans le réseau *192.168.1/24*. Après avoir analysé les exemples, nous pouvons résumer ce que nous avons appris.

12.1.2. Sélecteurs généraux

Les sélecteurs généraux définissent le modèle, le masque et le décalage qui seront comparés au contenu du paquet. En utilisant les sélecteurs généraux, vous pouvez rechercher des correspondances sur n'importe quel bit de l'en−tête IP (ou des couches supérieures). Ils sont quand même plus difficiles à écrire et à lire que les sélecteurs spécifiques décrits ci−dessus. La syntaxe générale des sélecteurs est :

```
match [ u32 | u16 | u8 ] PATTERN MASK [ at OFFSET | nexthdr+OFFSET]
```

Un des mots−clés *u32*, *u16* ou *u8* doit spécifier la longueur du modèle en bits. *PATTERN* et *MASK* se rapporteront à la longueur définie par ce mot−clé. Le paramètre *OFFSET* est le décalage, en octets, pour le démarrage de la recherche de correspondance. Si le mot−clef *nexthdr+* est présent, le décalage sera relatif à l'en−tête de la couche réseau supérieure.

Quelques exemples :

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

Un paquet correspondra à cette règle si sa << durée de vie >> (TTL) est de 64. TTL est le champ démarrant juste après le 8ème octet de l'en−tête 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: Il a été montré que cette syntaxe ne marche pas correctement.

Utilisez ceci pour déterminer la présence du bit ACK sur les paquets d'une longueur inférieure à 64 octets :

```
## Vérifie la présence d'un ACK,
## protocol IP 6,
## longueur de l'entête IP 0x5(mots de 32 bits),
## longueur total IP 0x34 (ACK + 12 octets d'options TCP)
## TCP ack actif (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
```

Seuls les paquets TCP sans charge utile et avec le bit ACK positionné vérifieront cette règle. Ici, nous pouvons voir un exemple d'utilisation de deux sélecteurs, le résultat final étant un ET logique de leur résultat. Si nous jetons un coup d'oeil sur un schéma de l'en−tête TCP, nous pouvons voir que le bit ACK est le second bit (*0x10*) du 14ème octet de l'en−tête TCP (*at nexthdr+13*). Comme second sélecteur, si nous voulons nous compliquer la vie, nous pouvons écrire *match u8 0x06 0xff at 9* à la place du sélecteur spécifique *protocol tcp*, puisque 6 est le numéro du protocole TCP, spécifié au 10ème octet de l'en−tête IP. D'un autre côté, dans cet exemple, nous ne pourrons pas utiliser de sélecteur spécifique pour la première correspondance, simplement parce qu'il n'y a pas de sélecteur spécifique pour désigner les bits TCP ACK.

12.1.3. Les sélecteurs spécifiques

La table suivante contient la liste de tous les sélecteurs spécifiques que les auteurs de cette section ont trouvés dans le code source du programme **tc**. Ils rendent simplement la vie plus facile en accroissant la lisibilité de la configuration du filtre.

FIXME: emplacement de la table − la table est dans un fichier séparé "selector.html"

FIXME: C'est encore en Polonais :−(FIXME: doit être "sgmlisé"

Quelques exemples :

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

FIXME: tcp dst match ne fonctionne pas comme décrit ci−dessous :

La règle ci−dessus correspondra à des paquets qui ont le champ TOS égal à *0x10*. Le champ TOS commence au deuxième octet du paquet et occupe 1 octet, ce qui nous permet d'écrire un sélecteur général équivalent : *match u8 0x10 0xff at 1*. Cela nous donne une indication sur l'implémentation du filtre *u32*. Les règles spécifiques sont toujours traduites en règles générales, et c'est sous cette forme qu'elles sont stockées en mémoire par le noyau. Cela amène à une autre conclusion : les sélecteurs *tcp* et *udp* sont exactement les mêmes et c'est la raison pour laquelle vous ne pouvez pas utiliser un simple sélecteur *match tcp dst 53 0xffff* pour désigner un paquet TCP envoyé sur un port donné. Ce sélecteur désigne aussi les paquets UDP envoyés sur ce port. Vous devez également spécifier le protocole avec la règle suivante :

```
# 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. Le classificateur *route*

Ce classificateur filtre en se basant sur les informations des tables de routage. Quand un paquet passant à travers les classes en atteint une qui est marquée avec le filtre *route*, il divise le paquet en se basant sur l'information de la table de routage.

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

Ici, nous ajoutons un classificateur *route* sur le noeud parent *1:0*, avec la priorité *100*. Quand un paquet atteint ce noeud (ce qui arrive immédiatement, puisqu'il est racine), il consulte la table de routage et si une entrée de la table correspond, il envoie le paquet vers la classe donnée et lui donne une priorité de *100*. Ensuite, vous ajoutez l'entrée de routage appropriée pour finalement activer les choses.

L'astuce ici est de définir *realm* en se basant soit sur la destination, soit sur la source. Voici la façon de procéder :

```
# ip route add Host/Network via Gateway dev Device realm RealmNumber
```

Par exemple, nous pouvons définir notre réseau de destination *192.168.10.0* avec le nombre *realm* égal à *10* :

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

Quand on ajoute des filtres *route*, on peut utiliser les nombres *realm* pour représenter les réseaux ou les hôtes et spécifier quelle est la correspondance entre les routes et les filtres.

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

La règle ci–dessus indique que les paquets allant vers le réseau *192.168.10.0* correspondent à la classe *1:10*.

Le filtre *route* peut aussi être utilisé avec les routes sources. Par exemple, il y a un sous–réseau attaché à notre routeur Linux sur *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
```

Ici, le filtre spécifie que les paquets venant du réseau *192.168.2.0* (*realm 2*) correspondront à la classe *1:2*.

12.3. Les filtres de réglementation (*Policing filters*)

Pour réaliser des configurations encore plus compliquées, vous pouvez avoir des filtres qui analysent le trafic à hauteur d'une certaine bande passante. Vous pouvez configurer un filtre pour qu'il cesse complètement l'analyse de tout le trafic au–dessus d'un certain débit ou pour qu'il n'analyse pas la bande passante dépassant un certain débit.

Ainsi, si vous décidez de réglementer à 4mbit/s, mais qu'un trafic de 5mbit/s est présent, vous pouvez cesser d'analyser l'ensemble des 5mbit/s ou seulement cesser d'analyser le 1 mbit/s supplémentaire et envoyer 4 mbit/s à la classe correspondante.

Si la bande passante dépasse le débit configuré, vous pouvez rejeter un paquet, le reclassifier ou voir si un autre filtre y correspond.

12.3.1. Techniques de réglementation

Il y a essentiellement deux façons de réglementer. Si vous avez compilé le noyau avec *Estimators*, celui−ci peut mesurer plus ou moins pour chaque filtre le trafic qui est passé. Ces estimations ne sont pas coûteuses en temps CPU, étant donné qu'il ne compte que 25 fois par seconde le nombre de données qui sont passées, et qu'il calcule le débit à partir de là.

L'autre manière utilise encore le *Token Bucket Filter* qui réside à l'intérieur du filtre cette fois. Le TBF analyse seulement le trafic *A HAUTEUR* de la bande passante que vous avez configurée. Si cette bande passante est dépassée, seul l'excès est traité par l'action de dépassement de limite configurée.

12.3.1.1. Avec l'estimateur du noyau

Ceci est très simple et il n'y a qu'un seul paramètre : *avrate*. Soit le flux demeure sous *avrate* et le filtre classifie le trafic vers la classe appropriée, soit votre débit le dépasse et l'action indiquée par défaut, la << reclassification >>, est réalisée dans ce cas.

Le noyau utilise l'algorithme EWMA pour votre bande passante, ce qui la rend moins sensible aux courtes rafales de données.

12.3.1.2. Avec le *Token Bucket Filter*

Utilisez les paramètres suivants :

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

Ceux−ci se comportent la plupart du temps de manière identique à ceux décrits dans la section *Filtre à seau de jetons*. Notez cependant que si vous configurez le *mtu* du filtre de réglementation TBF trop bas, aucun paquet ne passera et le gestionnaire de mise en file d'attente de sortie TBF ne fera que les ralentir.

Une autre différence est que la réglementation ne peut que laisser passer ou jeter un paquet. Il ne peut pas le retenir dans le but de le retarder.

12.3.2. Actions de dépassement de limite (*Overlimit actions*)

Si votre filtre décide qu'un dépassement de limite est atteint, il peut mettre en oeuvre des << actions >>. Actuellement, trois actions sont disponibles :

continue
: Provoque l'arrêt de l'analyse du filtre, bien que d'autres filtres aient la possibilité de le faire.

drop
: Ceci est une option très féroce qui supprime simplement le trafic excédant un certain débit. Elle est souvent employée dans le *Ingress policer* et a des utilisations limitées. Par exemple, si vous avez un serveur de noms qui s'écroule s'il traite plus de 5mbit/s de paquets, alors, vous pourrez dans ce cas utiliser un filtre d'entrée pour être sûr qu'il ne traitera jamais plus de 5mbit/s.

Pass/OK
: Transmettre le trafic. Peut être utilisé pour mettre hors service un filtre compliqué, tout en le laissant

en place.

reclassify
: Permet le plus souvent une reclassification vers *Best Effort*. Ceci est l'action par défaut.

12.3.3. Exemples

Le seul vrai exemple connu est mentionné dans la section *Protéger votre machine des inondations SYN*.

FIXME: Si vous avez déjà utilisé ceci, partagez s'il vous plaît votre expérience avec nous.

12.4. Filtres hachés pour un filtrage massif très rapide

Si vous avez besoin de milliers de règles, par exemple, dans le cas où vous avez beaucoup de clients ou d'ordinateurs, tous avec des spécifications QoS différentes, vous pourrez constater que le noyau passe beaucoup de temps à analyser toutes ces règles.

Par défaut, tous les filtres résident dans une grande chaîne qui est analysée par ordre décroissant des priorités. Si vous avez 1000 règles, 1000 contrôles peuvent être nécessaires pour déterminer ce qu'il faut faire d'un paquet.

La vérification irait plus vite s'il y avait 256 chaînes avec chacune quatre règles et si vous pouviez répartir les paquets sur ces 256 chaînes, afin que la bonne règle soit présente.

Ceci est rendu possible par le hachage. Imaginons que vous ayez sur votre réseau 1024 clients avec des modems câble, avec des adresses IP allant de *1.2.0.0* à *1.2.3.255*, et que chacun doit avoir un classement particulier, par exemple << pauvre >>, << moyen >> et << bourrage >>. Cela vous ferait alors 1024 règles, dans le genre :

```
# 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
```

Pour aller plus vite, nous pouvons utiliser la dernière partie de l'adresse IP comme << clé de hachage >>. Nous obtenons alors 256 tables, la première ressemblant à ceci :

```
# 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 suivante commence comme ceci :

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

De cette manière, seules quatre recherches au plus sont nécessaires et deux en moyenne.

La configuration est plutôt compliquée, mais elle en vaut vraiment la peine du fait des nombreuses règles. Nous créons d'abord un filtre racine, puis une table avec 256 entrées :

```
# 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: u32 divisor 256
```

Nous ajoutons maintenant des règles dans la table précédemment créée :

```
# 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
```

Ceci est l'entrée 123, qui contient les correspondances pour *1.2.0.13*, *1.2.1.123*, *1.2.2.123* et *1.2.3.123* qui les envoient respectivement vers *1:1*, *1:2*, *1:3* et *1:2*. Notez que nous devons spécifier notre seau de hachage en hexadécimal, *0x7b* pour *123*.

Nous créons ensuite un << filtre de hachage >>qui dirige le trafic vers la bonne entrée de la table de hachage :

```
# 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:
```

Ok, certains nombres doivent être expliqués. La table de hachage par défaut est appelée *800::* et tous les filtres démarrent de là. Nous sélectionnons alors l'adresse source qui est en position 12, 13, 14 et 15 dans l'en−tête IP, et indiquons que seule la dernière partie nous intéresse. Ceci est envoyé vers la table de hachage *2:* qui a été créée plus tôt.

C'est plutôt compliqué, mais cela marche en pratique et les performances seront époustouflantes. Notez que cet exemple pourrait être amélioré pour que chaque chaîne contienne un filtre, ce qui représenterait le cas idéal !

Chapitre 13. Paramètres réseau du noyau

Le noyau utilise de nombreux paramètres qui peuvent être ajustés en différentes circonstances. Bien que, comme d'habitude, les paramètres par défaut conviennent à 99% des installations, nous ne pourrions pas appeler ce document << HOWTO avancé >> sans en dire un mot.

Les éléments intéressants sont dans `/proc/sys/net`, jetez−y un oeil. Tout ne sera pas documenté ici au départ, mais nous y travaillons.

En attendant, vous pouvez jeter un oeil dans les sources du noyau Linux et lire le fichier `Documentation/filesystems/proc.txt`. La plupart des fonctionnalités y sont expliquées.

13.1. Filtrage de Chemin Inverse (*Reverse Path Filtering*)

Par défaut, les routeurs routent tout, même les paquets qui visiblement n'appartiennent pas à votre réseau. Un exemple courant est l'espace des adresses IP privées s'échappant sur Internet. Si vous avez une interface avec une route pour *195.96.96.0/24* dessus, vous ne vous attendrez pas à voir arriver des paquets venant de *212.64.94.1*.

Beaucoup d'utilisateurs veulent désactiver cette fonctionnalité. Les développeurs du noyau ont permis de le faire facilement. Il y a des fichiers dans `/proc` où vous pouvez ordonner au noyau de le faire pour vous. La méthode est appelée << Filtrage par Chemin Inverse >> (*Reverse Path Filtering*). Pour faire simple, si la réponse à ce paquet ne sort pas par l'interface par laquelle il est entré, alors c'est un paquet << bogué >> et il sera ignoré.

Les instructions suivantes vont activer cela pour toutes les interfaces courantes et futures.

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

En reprenant l'exemple du début, si un paquet arrivant sur le routeur Linux par *eth1* prétend venir du réseau Bureau+FAI, il sera éliminé. De même, si un paquet arrivant du réseau Bureau prétend être de quelque part à l'extérieur du pare−feu, il sera également éliminé.

Ce qui est présenté ci−dessus est le filtrage de chemin inverse complet. Le paramétrage par défaut filtre seulement sur les adresses IP des réseaux directement connectés. Ce paramétrage par défaut est utilisé parce que le filtrage complet échoue dans le cas d'un routage asymétrique (où il y a des paquets arrivant par un chemin et ressortant par un autre, comme dans le cas du trafic satellite ou si vous avez des routes dynamiques (bgp, ospf, rip) dans votre réseau. Les données descendent vers la parabole satellite et les réponses repartent par des lignes terrestres normales).

Si cette exception s'applique dans votre cas (vous devriez être au courant), vous pouvez simplement désactiver le `rp_filter` sur l'interface d'arrivée des données satellite. Si vous voulez voir si des paquets sont éliminés, le fichier `log_martians` du même répertoire indiquera au noyau de les enregistrer dans votre syslog.

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

FIXME: Est−ce que la configuration des fichiers dans .../conf/{default,all} suffit ? − martijn

13.2. Configurations obscures

Bon, il y a beaucoup de paramètres qui peuvent être modifiés. Nous essayons de tous les lister. Voir aussi une documentation partielle dans `Documentation/ip−sysctl.txt`.

Certaines de ces configurations ont des valeurs par défaut différentes suivant que vous répondez *Yes* ou *No* à la question *Configure as router and not host* lors de la compilation du noyau.

13.2.1. ipv4 générique

En remarque générale, les fonctionnalités de limitation de débit ne fonctionnent pas sur l'interface *loopback*. N'essayez donc pas de les tester localement. Les limites sont exprimées en << tic−tac >> (*jiffies*) et elles utilisent obligatoirement le *Token Bucket Filter* mentionné plus tôt.

[NdT : le terme *jiffies* désigne un mouvement régulier, faisant référence au << tic−tac >> d'une horloge. Dans le noyau lui−même, une variable globale nommée *jiffies* est incrémenté à chaque interruption d'horloge]

Le noyau a une horloge interne qui tourne à *HZ* impulsions (ou *jiffies*) par seconde. Sur intel, *HZ* est la plupart du temps de égale à *100*. Donc, configurer un fichier `*_rate` à, disons 50, autorise 2 paquets par seconde. Le *Token Bucket Filter* est également configuré pour autoriser une rafale de données de 6 paquets au plus, si suffisamment de jetons ont été gagnés.

Plusieurs éléments de la liste suivante proviennent du fichier `/usr/src/linux/Documentation/networking/ip−sysctl.txt`, écrit par Alexey Kuznetsov `<kuznet@ms2.inr.ac.ru>` et Andi Kleen `<ak@muc.de>`.

/proc/sys/net/ipv4/icmp_destunreach_rate
: Si le noyau décide qu'il ne peut pas délivrer un paquet, il le rejetera et enverra à la source du paquet un ICMP notifiant ce rejet.

/proc/sys/net/ipv4/icmp_echo_ignore_all
: N'agit en aucun cas comme écho pour les paquets. Ne configurez pas ceci par défaut. Cependant, si vous êtes utilisé comme relais dans une attaque de Déni de Services, cela peut être utile.

/proc/sys/net/ipv4/icmp_echo_ignore_broadcasts [Utile]
: Si vous pinguez l'adresse de diffusion d'un réseau, tous les hôtes sont censés répondre. Cela permet de coquettes attaques de déni de service. Mettez cette valeur à 1 pour ignorer ces messages de diffusion.

/proc/sys/net/ipv4/icmp_echoreply_rate
: Le débit auquel les réponses echo sont envoyées aux destinataires.

/proc/sys/net/ipv4/icmp_ignore_bogus_error_responses
: Configurer ceci pour ignorer les erreurs ICMP d'hôtes du réseau réagissant mal aux trames envoyées vers ce qu'ils perçoivent comme l'adresse de diffusion.

/proc/sys/net/ipv4/icmp_paramprob_rate
: Un message ICMP relativement peu connu, qui est envoyé en réponse à des paquets qui ont des en−têtes IP ou TCP erronés. Avec ce fichier, vous pouvez contrôler le débit auquel il est envoyé.

/proc/sys/net/ipv4/icmp_timeexceed_rate
: Voici la célèbre cause des << étoiles Solaris >> dans traceroute. Limite le nombre de messages ICMP Time Exceeded envoyés.

/proc/sys/net/ipv4/igmp_max_memberships
: Nombre maximal de sockets igmp (multidistribution) en écoute sur l'hôte. FIXME: Est−ce vrai ?

/proc/sys/net/ipv4/inet_peer_gc_maxtime
: FIXME : Ajouter une petite explication sur le stockage des partenaires internet (inet peer) ? Intervalle de temps minimum entre deux passages du ramasse−miettes. Cet intervalle est pris en compte lors d'une faible (voire inexistante) utilisation du *pool*. Mesuré en *jiffies*. [NdT : Le *pool* désigne ici la liste des adresses IP des partenaires internet.]

/proc/sys/net/ipv4/inet_peer_gc_mintime
: Intervalle de temps minimum entre deux passages du ramasse−miettes. Cet intervalle est pris en compte lors d'une utilisation intensive du *pool*. Mesuré en *jiffies*.

/proc/sys/net/ipv4/inet_peer_maxttl
: Durée de conservation maximale des enregistrements. Les entrées non utilisées expireront au bout de cet intervalle de temps (c'est−à−dire quand le nombre d'entrées dans le *pool* est très petit). Mesuré en *jiffies*.

/proc/sys/net/ipv4/inet_peer_minttl
: Durée de conservation minimale des enregistrements. Devrait être suffisante pour prendre en compte le temps de vie des fragments sur l'hôte qui doit réassembler les paquets. Cette durée minimale est garantie si le nombre d'éléments dans le *pool* est inférieur au seuil fixé par *inet_peer_threshold*.

/proc/sys/net/ipv4/inet_peer_threshold
: Taille approximative de l'espace de stockage des partenaires internet. A partir de ce seuil, les entrées sont effacées. Ce seuil détermine la durée de vie des entrées, ainsi que les intervalles de temps entre deux déclenchements du ramasse−miettes. Plus il y a d'entrées, plus le temps de vie est faible et plus l'intervalle du ramasse−miettes est faible.

`/proc/sys/net/ipv4/ip_autoconfig`
: Ce fichier contient la valeur 1 si l'hôte a reçu sa configuration IP par RARP, BOOTP, DHCP ou un mécanisme similaire. Autrement, il contient la valeur zéro.

`/proc/sys/net/ipv4/ip_default_ttl`
: Durée de vie (TTL) des paquets. Fixer à la valeur sûre de 64. Augmentez−la si vous avez un réseau immense, mais pas << pour s'amuser >> : les boucles sans fin d'un mauvais routage sont plus dangereuses si le TTL est élevé. Vous pouvez même envisager de diminuer la valeur dans certaines circonstances.

`/proc/sys/net/ipv4/ip_dynaddr`
: Vous aurez besoin de positionner cela si vous utilisez la connexion à la demande avec une adresse d'interface dynamique. Une fois que votre interface a été configurée, toutes les sockets TCP locaux qui n'ont pas eu de paquets de réponse seront retraitées pour avoir la bonne adresse. Cela résout le problème posé par une connexion défectueuse ayant configuré une interface, suivie par une deuxième tentative réussie (avec une adresse IP différente).

`/proc/sys/net/ipv4/ip_forward`
: Le noyau doit−il essayer de transmettre les paquets ? Désactivé par défaut.

`/proc/sys/net/ipv4/ip_local_port_range`
: Intervalle des ports locaux pour les connexions sortantes. En fait, assez petit par défaut, 1024 à 4999.

`/proc/sys/net/ipv4/ip_no_pmtu_disc`
: Configurez ceci si vous voulez désactiver la découverte du MTU de chemin, une technique pour déterminer le plus grand MTU possible sur votre chemin. Voir aussi la section sur la découverte du MTU de chemin dans le chapitre *Recettes de cuisine*.

`/proc/sys/net/ipv4/ipfrag_high_thresh`
: Mémoire maximum utilisée pour réassembler les fragments IP. Quand *ipfrag_high_thresh* octets de mémoire sont alloués pour cela, le gestionnaire de fragments rejettera les paquets jusqu'à ce que *ipfrag_low_thresh* soit atteint.

`/proc/sys/net/ipv4/ip_nonlocal_bind`
: Configurez ceci si vous voulez que vos applications soient capables de se lier à une adresse qui n'appartient pas à une interface de votre système. Ceci peut être utile quand votre machine est sur un lien non−permanent (ou même permanent). Vos services sont donc capables de démarrer et de se lier à une adresse spécifique quand votre lien est inactif.

`/proc/sys/net/ipv4/ipfrag_low_thresh`
: Mémoire minimale utilisée pour réassembler les fragments IP.

`/proc/sys/net/ipv4/ipfrag_time`
: Temps en secondes du maintien d'un fragment IP en mémoire.

`/proc/sys/net/ipv4/tcp_abort_on_overflow`
: Une option booléenne contrôlant le comportement dans le cas de nombreuses connexions entrantes. Quand celle−ci est activée, le noyau envoie rapidement des paquets RST quand un service est surchargé.

`/proc/sys/net/ipv4/tcp_fin_timeout`
: Temps de maintien de l'état *FIN−WAIT−2* pour un socket dans le cas où il a été fermé de notre côté. Le partenaire peut être défectueux et ne jamais avoir fermé son côté ou même mourir de manière inattendue. La valeur par défaut est de 60 secondes. La valeur usuelle utilisée dans le noyau 2.2 était de 180 secondes. Vous pouvez la remettre, mais rappelez vous que si votre machine a un serveur WEB surchargé, vous risquez de dépasser la mémoire avec des kilotonnes de sockets morts. Les sockets *FIN−WAIT2* sont moins dangereux que les sockets *FIN−WAIT1* parce qu'ils consomment au maximum 1,5K de mémoire, mais ils ont tendance à vivre plus longtemps. Cf *tcp_max_orphans*.

`/proc/sys/net/ipv4/tcp_keepalive_time`
: Durée entre l'envoi de deux messages *keepalive* quand l'option *keepalive* est activée. Par défaut : 2 heures.

`/proc/sys/net/ipv4/tcp_keepalive_intvl`
: A quelle fréquence les sondes sont retransmises lorsqu'il n'y a pas eu acquittement de sonde. Par défaut : 75 secondes.

`/proc/sys/net/ipv4/tcp_keepalive_probes`

Combien de sondes TCP *keepalive* seront envoyées avant de décider que la connexion est brisée. Par défaut : 9. En multipliant par *tcp_keepalive_intvl*, cela donne le temps pendant lequel un lien peut être actif sans donner de réponses après l'envoi d'un *keepalive*.

`/proc/sys/net/ipv4/tcp_max_orphans`
: Nombre maximum de sockets TCP qui ne sont pas reliés à un descripteur de fichier utilisateur, géré par le système. Si ce nombre est dépassé, les connexions orphelines sont immédiatement réinitialisées et un avertissement est envoyé. Cette limite existe seulement pour prévenir des attaques de déni de services simples. Vous ne devez pas compter sur ceci ou diminuer cette limite artificiellement, mais plutôt l'augmenter (probablement après avoir augmenté la mémoire) si les conditions du réseau réclament plus que cette valeur par défaut et régler vos services réseau pour qu'ils détruisent sans tarder ce type d'état. Laissez−moi vous rappeler encore que chaque orphelin consomme jusqu'à environ 64K de mémoire non *swappable*.

`/proc/sys/net/ipv4/tcp_orphan_retries`
: Combien d'essais avant de détruire une connexion TCP, fermée par notre côté. La valeur par défaut de 7 correspond à un temps d'environ 50s à 16 min suivant le RTO. Si votre machine supporte un serveur Web, vous pouvez envisager de baisser cette valeur, dans la mesure où de tels sockets peuvent consommer des ressources significatives. Cf *tcp_max_orphans*.

`/proc/sys/net/ipv4/tcp_max_syn_backlog`
: Nombre maximum de requêtes d'une connexion mémorisée, qui n'avait pas encore reçu d'accusé de réception du client connecté. La valeur par défaut est de 1024 pour des systèmes avec plus de 128 Mo de mémoire et 128 pour des machines avec moins de mémoire. Si un serveur souffre de surcharge, essayez d'augmenter ce nombre. Attention ! Si vous positionnez une valeur supérieure à 1024, il serait préférable de changer *TCP_SYNQ_HSIZE* dans le fichier `include/net/tcp.h` pour garder *TCP_SYNQ_HSIZE*16 <= tcp_max_syn_backlog* et de recompiler de noyau.

`/proc/sys/net/ipv4/tcp_max_tw_buckets`
: Nombre maximum de sockets *timewait* gérées par le système simultanément. Si ce nombre est dépassé, le socket *timewait* est immédiatement détruit et un message d'avertissement est envoyé. Cette limite n'existe que pour prévenir des attaques de déni de services simples. Vous ne devez pas diminuer cette limite artificiellement, mais plutôt l'augmenter (probablement après avoir augmenté la mémoire) si les conditions du réseau réclament plus que cette valeur par défaut.

`/proc/sys/net/ipv4/tcp_retrans_collapse`
: Compatibilité bug à bug avec certaines imprimantes défectueuses. Tentative d'envoi de plus gros paquets lors de la retransmission pour contourner le bug de certaines piles TCP.

`/proc/sys/net/ipv4/tcp_retries1`
: Combien d'essais avant de décider que quelque chose est erroné et qu'il est nécessaire d'informer de cette suspicion la couche réseau. La valeur minimale du RFC est de 3. C'est la valeur par défaut ; elle correspond à un temps d'environ 3 sec à 8 min suivant le RTO.

`/proc/sys/net/ipv4/tcp_retries2`
: Combien d'essais avant de détruire une connexion TCP active. Le RFC 1122 précise que la limite ne devrait pas dépasser 100 secondes. C'est un nombre trop petit. La valeur par défaut de 15 correspond à un temps de environ 13 à 30 minutes suivant le RTO.

`/proc/sys/net/ipv4/tcp_rfc1337`
: Ce booléen active un rectificatif pour << l'assassinat hasardeux des time−wait dans tcp >>, décrit dans le RFC 1337. S'il est activé, le noyau rejette les paquets RST pour les sockets à l'état de *time−wait*. Par défaut : 0

`/proc/sys/net/ipv4/tcp_sack`
: Utilise un ACK sélectif qui peut être utilisé pour signifier que des paquets spécifiques sont manquant. Facilite ainsi une récupération rapide.

`/proc/sys/net/ipv4/tcp_stdurg`
: Utilise l'interprétation du RFC *Host Requirements* du champ TCP pointeur urgent. La plupart des hôtes utilisent la vieille interprétation BSD. Donc, si vous activez cette option, il se peut que Linux ne communique plus correctement avec eux. Par défaut : FALSE (FAUX)

`/proc/sys/net/ipv4/tcp_syn_retries`
: Nombre de paquets SYN que le noyau enverra avant de tenter l'établissement d'une nouvelle

connexion.

`/proc/sys/net/ipv4/tcp_synack_retries`
: Pour ouvrir l'autre côté de la connexion, le noyau envoie un SYN avec un ACK superposé (*piggyback*), pour accuser réception du SYN précédemment envoyé. C'est la deuxième partie de la poignée de main à trois voies (*threeway handshake*). Cette configuration détermine le nombre de paquets SYN+ACK à envoyer avant que le noyau n'abandonne la connexion.

`/proc/sys/net/ipv4/tcp_timestamps`
: Les estampillages horaires sont utilisés, entre autres, pour se protéger du rebouclage des numéros de séquence. On peut concevoir qu'un lien à 1 gigabit puisse de nouveau rencontrer un numéro de séquence précédent avec une valeur hors−ligne parcequ'il était d'une génération précédente. L'estampillage horaire permet de reconnaître cet << ancien paquet >>.

`/proc/sys/net/ipv4/tcp_tw_recycle`
: Mise en place du recyclage rapide des sockets *TIME−WAIT*. La valeur par défaut est 1. Celle−ci ne devrait pas être changée sans le conseil/demande d'experts techniques.

`/proc/sys/net/ipv4/tcp_window_scaling`
: TCP/IP autorise normalement des fenêtres jusqu'à une taille de 65535 octets. Pour des réseaux vraiment rapides, cela peut ne pas être assez. Les options *windows scaling* autorisent des fenêtres jusqu'au gigaoctet, ce qui est adapté pour les produits à grande bande passante.

13.2.2. Configuration des périphériques

DEV peut désigner soit une interface réelle, soit *all*, soit *default*. *Default* change également les paramètres des interfaces qui seront créées par la suite.

`/proc/sys/net/ipv4/conf/DEV/accept_redirects`
: Si un routeur décide que vous l'utilisez à tort (c'est−à−dire qu'il a besoin de ré−envoyer votre paquet sur la même interface), il vous enverra un message *ICMP Redirect*. Cela présente cependant un petit risque pour la sécurité, et vous pouvez le désactiver ou utiliser les redirections sécurisées.

`/proc/sys/net/ipv4/conf/DEV/accept_source_route`
: Plus vraiment utilisé. On l'utilisait pour être capable de donner à un paquet une liste d'adresses IP à visiter. Linux peut être configuré pour satisfaire cette option IP.

`/proc/sys/net/ipv4/conf/DEV/bootp_relay`
: Accepte les paquets avec une adresse source 0.b.c.d et des adresses destinations qui ne correspondent ni à cet hôte, ni au réseau local. On suppose qu'un démon de relais BOOTP interceptera et transmettra de tels paquets.

  La valeur par défaut est 0, puique cette fonctionnalité n'est pas encore implémentée (noyau 2.2.12).

`/proc/sys/net/ipv4/conf/DEV/forwarding`
: Active ou désactive la transmission IP sur cette interface.

`/proc/sys/net/ipv4/conf/DEV/log_martians`
: Voir la section sur le *Filtrage de Chemin Inverse*.

`/proc/sys/net/ipv4/conf/DEV/mc_forwarding`
: Si vous faites de la transmission multidistribution (*multicast*) sur cette interface.

`/proc/sys/net/ipv4/conf/DEV/proxy_arp`
: Si vous configurez ceci à 1, cet interface répondra aux requêtes ARP pour les adresses que le noyau doit router. Peut être très utile si vous mettez en place des << pseudo−ponts ip >>. Prenez bien garde d'avoir des masques de sous−réseau corrects avant d'activer cette option. Faites également attention que le rp_filter agisse aussi sur le requêtes ARP !

`/proc/sys/net/ipv4/conf/DEV/rp_filter`
: Voir la section sur le *Filtrage de Chemin Inverse*.

`/proc/sys/net/ipv4/conf/DEV/secure_redirects`
: Accepte les messages de redirection ICMP seulement pour les passerelles indiquées dans la liste des passerelles par défaut. Activé par défaut.

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

Active la possibilité d'envoyer les messages de redirections mentionnées ci−dessus.

`/proc/sys/net/ipv4/conf/DEV/shared_media`
: Si cela n'est pas activé, le noyau ne considère pas que différents sous−réseaux peuvent communiquer directement sur cette interface. La configuration par défaut est *Yes*.

`/proc/sys/net/ipv4/conf/DEV/tag`
: FIXME: à remplir

13.2.3. Politique de voisinage

DEV peut désigner soit une interface réelle, soit *all*, soit *default*. *Default* change également les paramètres des interfaces qui seront créées par la suite.

`/proc/sys/net/ipv4/neigh/DEV/anycast_delay`
: Valeur maximum du délai aléatoire de réponse exprimé en *jiffies* (1/100 sec) aux messages de sollicitation des voisins. N'est pas encore implémenté (Linux ne possède pas encore le support *anycast*).

`/proc/sys/net/ipv4/neigh/DEV/app_solicit`
: Détermine le nombre de requêtes à envoyer au démon ARP de l'espace utilisateur. Utilisez 0 pour désactiver.

`/proc/sys/net/ipv4/neigh/DEV/base_reachable_time`
: Une valeur de base utilisée pour le calcul du temps aléatoire d'accès comme spécifié dans le RFC2461.

`/proc/sys/net/ipv4/neigh/DEV/delay_first_probe_time`
: Délai avant de tester pour la première fois si le voisin peut être atteint. (voir *gc_stale_time*)

`/proc/sys/net/ipv4/neigh/DEV/gc_stale_time`
: Détermine la fréquence à laquelle on doit vérifier les vieilles entrées ARP. Si une entrée est obsolète, elle devra de nouveau être résolue (ce qui est utile quand une adresse IP a été attribuée à une autre machine). Si *ucast_solicit* est supérieur à 0, alors on essaie d'abord d'envoyer un paquet ARP directement à l'hôte connu. Si cela échoue, et que *mcast_solicit* est supérieur à 0, alors une requête ARP est multidiffusée.

`/proc/sys/net/ipv4/neigh/DEV/locktime`
: Une entrée ARP n'est remplacée par une nouvelle que si l'ancienne est au moins présente depuis *locktime*. Cela évite trop d'écriture dans le cache.

`/proc/sys/net/ipv4/neigh/DEV/mcast_solicit`
: Nombre maximum d'essais consécutifs pour une sollicitation *multicast*.

`/proc/sys/net/ipv4/neigh/DEV/proxy_delay`
: Temps maximum (le temps réel est aléatoire et compris entre 0 et *proxytime*) avant de répondre à une requête ARP pour laquelle nous avons une entrée de proxy ARP. Peut être utilisé dans certains cas pour se prémunir des inondations réseaux.

`/proc/sys/net/ipv4/neigh/DEV/proxy_qlen`
: Longueur maximale de la file d'attente du temporisateur de cache arp en attente (Voir *proxy_delay*).

`/proc/sys/net/ipv4/neigh/DEV/retrans_time`
: Le temps, exprimé en *jiffies* (1/100 sec), entre deux requêtes ARP. Utilisé pour la résolution d'adresses et pour déterminer si un voisin est inaccessible.

`/proc/sys/net/ipv4/neigh/DEV/ucast_solicit`
: Nombre maximum de requêtes ARP unicast.

`/proc/sys/net/ipv4/neigh/DEV/unres_qlen`
: Longueur maximum de la file d'attente pour la requête ARP en cours : le nombre de paquets qui sont acceptés des autres couches pendant la résolution ARP d'une adresse.

Internet QoS: Architectures and Mechanisms for Quality of Service, Zheng Wang, ISBN 1−55860−608−4
: Livre traitant des sujets liés à la qualité de service. Bien pour comprendre les concepts de base.

13.2.4. Configuration du routage

/proc/sys/net/ipv4/route/error_burst
: Ces paramètres sont utilisés pour limiter le nombre de messages d'avertissement écrits dans le journal du noyau par le code de routage. Plus le paramètre *error_burst* est grand, moins il y aura de messages. *Error_burst* contrôle le moment où les messages seront supprimés. Les configurations par défaut se limitent à un message d'avertissement toutes les cinq secondes.

/proc/sys/net/ipv4/route/error_cost
: Ces paramètres sont utilisés pour limiter le nombre de messages d'avertissement écrits dans le journal du noyau par le code de routage. Plus le paramètre *error_cost* est grand, moins il y aura de messages. *error_burst* contrôle le moment où les messages seront jetés. Les configurations par défaut se limitent à un message d'avertissement toutes les cinq secondes.

/proc/sys/net/ipv4/route/flush
: L'écriture dans ce fichier provoque la vidange du cache du routage.

/proc/sys/net/ipv4/route/gc_elasticity
: Valeurs qui contrôlent la fréquence et le comportement de l'algorithme *garbage collection* du cache de routage. Ceci peut être important en cas de défaut. Au moins *gc_timeout* secondes s'écouleront avant que le noyau ne passe à une autre route si la précédente n'est plus opérationnelle. Configuré par défaut à 300. Diminuez cette valeur si vous voulez passer plus rapidement ce type de problème.

  Voir aussi ce message par Ard van Breemen.

/proc/sys/net/ipv4/route/gc_interval
: Voir `/proc/sys/net/ipv4/route/gc_elasticity`.

/proc/sys/net/ipv4/route/gc_min_interval
: Voir `/proc/sys/net/ipv4/route/gc_elasticity`.

/proc/sys/net/ipv4/route/gc_thresh
: Voir `/proc/sys/net/ipv4/route/gc_elasticity`.

/proc/sys/net/ipv4/route/gc_timeout
: Voir `/proc/sys/net/ipv4/route/gc_elasticity`.

/proc/sys/net/ipv4/route/max_delay
: Délai d'attente pour la vidange du cache du routage.

/proc/sys/net/ipv4/route/max_size
: Taille maximum du cache de routage. Les vieilles entrées seront purgées quand le cache aura atteint cette taille.

/proc/sys/net/ipv4/route/min_adv_mss
: FIXME: à remplir

/proc/sys/net/ipv4/route/min_delay
: Délai d'attente pour vider le cache de routage.

/proc/sys/net/ipv4/route/min_pmtu
: FIXME: à remplir

/proc/sys/net/ipv4/route/mtu_expires
: FIXME: à remplir

/proc/sys/net/ipv4/route/redirect_load
: Facteurs qui déterminent si plus de redirections ICMP doivent être envoyées à un hôte spécifique. Aucune redirection ne sera envoyée une fois que la limite de charge (*load limit*) ou que le nombre maximum de redirections aura été atteint.

/proc/sys/net/ipv4/route/redirect_number
: Voir `/proc/sys/net/ipv4/route/redirect_load`.

/proc/sys/net/ipv4/route/redirect_silence
: Temporisation pour les redirections. Au dela de cette période, les redirections seront de nouveau envoyées, même si elles ont été stoppées parce que la charge ou le nombre limite a été atteint.

Chapitre 14. Gestionnaires de mise en file d'attente avancés & moins communs

Si vous constatez que vous avez des besoins qui ne sont pas gérés par les files d'attente citées précédemment, le noyau contient quelques autres files d'attente plus spécialisées mentionnées ici.

14.1. *bfifo/pfifo*

Ces files d'attente sans classes sont plus simples que *pfifo_fast* dans la mesure où elles n'ont pas de bandes internes, tout le trafic étant vraiment équivalent. Elles ont cependant l'avantage important de réaliser des statistiques. Donc, même si vous n'avez pas besoin de mise en forme ou de donner une priorité, vous pouvez employer ce gestionnaire pour déterminer l'arriéré (*backlog*) de votre interface.

pfifo mesure en paquets et *bfifo* en octets.

14.1.1. Paramètres & usage

limit
: Spécifie la taille de la file d'attente. Mesurée en octets pour *bfifo* et en paquets pour *pfifo*. Par défaut, correspond à des paquets de taille égale au paramètre *txqueuelen* de l'interface (voir le chapitre pfifo_fast) ou *txqueuelen*mtu* octets pour *bfifo*.

14.2. Algorithme Clark−Shenker−Zhang (CSZ)

Ceci est si théorique que même Alexey (l'auteur principal de CBQ) prétend ne pas le comprendre. De son propre avis :

> David D. Clark, Scott Shenker and Lixia Zhang *Supporting Real−Time Applications in an Integrated Services Packet Network: Architecture and Mechanism*.
>
> Comme je le comprends, l'idée principale est de créer des flux WFQ pour chaque service garanti et d'allouer le reste de la bande passante au flux factice, appelé *flow−0*. Le Flux−0 inclut le trafic de service prédictif et le trafic *best−effort*. Il est traité par un ordonnanceur de priorité qui alloue la bande passante de plus grande priorité aux services prédictifs, et le reste aux paquets *best−effort*.
>
> Notez que dans CSZ, les flux ne sont *PAS* limités à leur bande passante. On suppose que le flux a passé le contrôle d'admission à la frontière du réseau QoS et qu'il n'a pas besoin de mises en forme supplémentaires. N'importe quelles autres tentatives pour améliorer le flux ou pour le mettre en forme grâce à un seau de jetons lors d'étapes intermédiaires introduiront des retards non désirés et augmenteront la gigue.
>
> A l'heure actuelle, CSZ est le seul ordonnanceur qui fournit un véritable service garanti. Les autres implémentations (incluant CBQ) n'assurent pas un délai garanti et rendent la gigue aléatoire.
>
> Ne semble pas actuellement un bon candidat à utiliser, à moins que vous n'ayez lu et compris l'article mentionné.

14.3. *DSMARK*

Esteve Camps

`<marvin@grn.es>`
`<esteve@hades.udg.es>`

Ce texte est un extrait de ma thèse sur le *support QoS dans Linux*, Septembre 2000.

Documents sources :

- Draft−almesberger−wajhak−diffserv−linux−01.txt.
- Exemples de la distribution iproute2.
- White Paper−QoS protocols and architectures et Foires Aux Questions IP QoS, les deux par *Quality of Service Forum*.

14.3.1. Introduction

Avant tout, il serait préférable de lire les RFC écrits sur ce sujet (RFC2474, RFC2475, RFC2597 et RFC2598) sur le site web du groupe de travail DiffServ IETF et sur le site web de Werner Almesberger (Il a écrit le code permettant le support des Services Différenciés sous Linux).

14.3.2. A quoi *DSMARK* est−il relié ?

DSMARK est un gestionnaire de mise en file d'attente qui offre les fonctionnalités dont ont besoin les services différenciés (*Differentiated Services*) (également appelés DiffServ ou tout simplement DS). *DiffServ* est l'une des deux architectures actuelles de la Qualité de Services (QoS : *Quality of Services*) (l'autre est appelée *services intégrés* (*Integrated Services*). Elle se base sur la valeur du champ DS contenu dans l'en−tête IP du paquet.

Une des premières solutions dans IP pour offrir des niveaux de qualité de services était le champ *Type de Service* (octet TOS) de l'en−tête IP. En modifiant la valeur de ce champ, nous pouvions choisir un niveau élevé/faible du débit, du délai ou de la fiabilité. Cependant, cela ne fournissait pas une flexibilité suffisante pour les besoins de nouveaux services (comme les applications temps réel, les applications interactives et autres). Par la suite, de nouvelles architectures sont apparues. L'une d'elle a été *DiffServ* qui a gardé les bits TOS et les a renommés champ DS.

14.3.3. Guide des services différenciés

Les services différenciés sont orientés groupes. Cela signifie que nous ne savons rien des flux (ce sera le but des services intégrés (*integrated Services*)). Nous connaissons par contre les agrégations de flux et nous adopterons des comportements différents suivant l'agrégation à laquelle appartient le paquet.

Quand un paquet arrive à un noeud frontalier (noeud d'entrée du domaine DiffServ) et entre dans un domaine DiffServ, nous devrons avoir une politique, une mise en forme et/ou un marquage de ces paquets (le marquage fait référence à la mise en place d'une valeur dans le champ DS. Comme on le ferait pour des vaches :−)). Ce sera cette marque/valeur que les noeuds internes de votre domaine DiffServ regarderons pour déterminer quel comportement ou niveau de qualité de service appliquer.

Comme vous pouvez le déduire, les Services Différenciés impliquent un domaine sur lequel toutes les règles DS devront être appliquées. Vous pouvez raisonner de la façon suivante : << Je classifierai tous les paquets entrant dans mon domaine. Une fois qu'ils seront entrés dans mon domaine, ils seront soumis aux règles que ma classification impose et chaque noeud traversé appliquera son niveau de qualité de service >>.

En fait, vous pouvez appliquer vos propres politiques dans vos domaines locaux, mais des *autorisations au niveau service* devront être considérées lors de la connexion à d'autres domaines DS.

En ce moment, vous vous posez peut–être beaucoup de questions. DiffServ est plus vaste que ce que j'ai expliqué. En fait, vous pouvez comprendre que je ne peux pas résumer plus de trois RFC en 50 lignes :–).

14.3.4. Travailler avec *DSMARK*

Comme le spécifie la bibliographie concernant DiffServ, nous différencions les noeuds frontaliers et les noeuds intérieurs. Ce sont deux éléments importants dans le chemin qu'emprunte le trafic. Les deux réalisent une classification quand un paquet arrive. Le résultat peut être utilisé à différents endroits lors du processus DS avant que le paquet ne soit libéré vers le réseau. Cela est possible car le nouveau code DiffServ fournit une structure appelée *sk_buff*, incluant un nouveau champ appelé *skb–>tcindex*. Ce champ mémorisera le résultat de la classification initiale et pourra être utilisé à plusieurs reprises dans le traitement DS.

La valeur *skb–>tc_index* sera initialement configurée par le gestionnaire de mise en file d'attente *DSMARK*. Cette valeur sera extraite du champ DS de l'en–tête IP de tous les paquets reçus. En outre, le classificateur *cls_tcindex* lira tout ou une partie de la valeur *skb–>tcindex* et l'utilisera pour sélectionner les classes.

Mais, avant tout, regardons la commande **qdisc DSMARK** et ses paramètres :

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

Que signifient ces paramètres ?

- *indices* : taille de la table des couples (masque,valeur). La valeur maximum est *2^n*, où *n>=0*.
- *default_index* : index d'entrée par défaut de la table si aucune correspondance n'est trouvée.
- *set_tc_index* : indique au gestionnaire *DSMARK* de récupérer le champs DS et de l'enregistrer dans *skb–>tc_index*.

Regardons *DSMARK* procéder.

14.3.5. Comment *SCH_DSMARK* travaille.

Ce gestionnaire de mise en file d'attente réalisera les étapes suivantes :

- Si vous avez déclaré l'option *set_tc_index* dans la commande **qdisc**, le champ DS est récupéré et mémorisé dans la variable *skb–>tc_index*.
- Le classificateur est invoqué. Celui–ci sera exécuté et retournera un identificateur de classe (*class ID*) qui sera enregistré dans la variable *skb–>tc_index*. Si aucun filtre correspondant n'est trouvé, nous considérons l'option *default_index* comme étant l'identificateur de classe à enregistrer. Si, ni *set_tc_index*, ni *default_index* n'ont été déclarés, alors les résultats peuvent être non prédictifs.
- Après avoir été envoyé dans le gestionnaire de file d'attente interne, où le résultat du filtre peut être réutilisé, l'identificateur de classe retourné par le gestionnaire est stocké dans la variable *skb–>tc_index*. Cette valeur sera utilisée plus tard pour indexer la table masque–valeur. Le résultat de l'opération suivante sera assigné au paquet :

```
 Nouveau_champ_DS = ( Ancien_champ_DS & masque ) | valeur
```

- La nouvelle valeur résultera donc d'un ET logique entre les valeurs du champ_DS et du masque, suivi d'un OU logique avec le paramètre valeur. Regardez la figure suivante pour comprendre tout ce processus :

```
                         skb->ihp->tos
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - >
     |                                                       |     ^
     | -- Si vous déclarez set_tc_index, nous configurons    |     |  <-----Peut changer
     |    la valeur DS dans la variable skb->tc_index        |     |O       le champ DS
     |                                                      A|     |R
   +-|-+      +------+    +---+-+  File d'attente-+     +---N|-----|----+
   | | |      |filtre|--->|   | |-->  . . .  -->| |     |   D|     |    |
   | | |----->|  tc  |--->|   | |   interne     | |---->|    v     |    |
   | | |      |index |--->| | | +---------------+ |   ---->(masque,valeur)|
-->| O |      +------+    +-|-+--------------^----+  /  |  (.  ,  .)    |
   | | |          ^         |                |       |  |  (.  ,  .)    |
   | | +----------|---------|----------------|-------|--+  (.  ,  .)    |
   | | sch_dsmark |         |                |       |                  |
   +-|------------|---------|----------------|-------|------------------+
     |            |         | <- tc_index -> |       |
     |            |(lecture)|   peut changer |       |  <--------------Index de la table
     |            |         |                |       |                    des couples
     v            |         v                v       |                    (masque,valeur)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - ->
                         skb->tc_index
```

Comment faire le marquage ? Il suffit de modifier le masque et la valeur associés à la classe que vous voulez marquer. Regardez la ligne de code suivante :

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

Cela modifie le couple (masque,valeur) dans la table de hachage, et re−marque les paquets appartenant à la classe 1:1. Vous devez "changer" ces valeurs en raison des valeurs par défaut que le couple (masque, valeur) obtient initialement (voir le tableau ci−dessous).

Nous allons maintenant expliquer comment le filtre TC_INDEX travaille, et comment il s'intègre dans tout cela. En outre, le filtre TC_INDEX peut être utiliser dans des configurations autres que celles incluant les services DS.

14.3.6. Le filtre TC_INDEX

Voici la commande de base pour déclarer un filtre TC_INDEX :

```
... tcindex [ hash SIZE ] [ mask MASK ] [ shift SHIFT ]
            [ pass_on | fall_through ]
            [ classid CLASSID ] [ police POLICE_SPEC ]
```

Ensuite, nous montrons l'exemple utilisé pour expliquer le mode opératoire de TC_INDEX. Soyez attentif aux mots en gras : 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 # Classe du trafic 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 # Gestionnaire de file d'attente fifo pour le trafic 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* (Ce code n'est pas complet. Ce n'est qu'un extrait de l'exemple EFCBQ inclus dans la distribution iproute2).

Avant tout, supposons que nous recevons un paquet marqué comme EF. Si vous lisez le RFC2598, vous verrez que DSCP recommande une valeur de 101110 pour le trafic EF. Cela signifie que le champ DS sera égal à 10111000 (rappelez− vous que les bits les moins significatifs de l'octet TOS ne sont pas utilisés dans DS) ou 0xb8 en notation hexadécimale.

```
              FILTRE
              TC INDEX
   +---+      +--------+   +---+-+    +------+                 +-+    +-------+
   |   |      |        |   |   | |    |FILTRE|  +-+    +-+     | |    |       |
   |   |----->| MASK   | -> |   | | -> |HANDLE|->| |    | | ->  | | -> |       |
   |   |  .   | =0xfc  |   |   | |    |0x2E  |  | +----+ |     | |    |       |
   |   |  .   |        |   |   | |    +------+  +--------+     | |    |       |
   |   |  .   |        |   |   | |                             | |    |       |
-->|   |  .   | SHIFT  |   |   | |                             | |    |       |-->
   |   |  .   | =2     |   |   | +-----------------------------+ |    |       |
   |   |      |        |   |   |       CBQ 2:0                   |    |       |
   |   |      +--------+   +---+---------------------------------+    |       |
   |   |                                                              |       |
   |   +--------------------------------------------------------------+       |
   |                          DSMARK 1:0                                      |
   +--------------------------------------------------------------------------+
```

Le paquet arrive alors avec la valeur du champ DS configurée à 0xb8. Comme je l'ai expliqué auparavant, le gestionnaire de mise en file d'attente dsmark, identifié par 1:0 dans cet exemple, récupère le champ DS et l'enregistre dans la variable skb−>tc_index. L'étape suivante consistera à associer un filtre à ce gestionnaire de mise en file d'attente (la seconde ligne dans cet exemple). Les opérations suivantes seront réalisées :

```
Valeur1 = skb->tc_index & MASK
Clé = Valeur1 >> SHIFT
```

Dans cet exemple, MASK=0xFC et SHIFT=2.

```
Valeur1 = 10111000 & 11111100 = 10111000
Clé = 10111000 >> 2 = 00101110 -> 0x2E en hexadécimal
```

La valeur retournée correspondra à un identificateur de filtre du gestionnaire de file d'attente interne (dans l'exemple, identifier par 2:0). Si un filtre avec cet identificateur (id) existe, les conditions de contrôle et de performance seront vérifiées (au cas où le filtre inclurait ceci) et l'identificateur de classe sera retourné (dans notre exemple, classid 2:1) et stocké dans la variable skb−>tc_index.

Si aucun filtre avec cet identificateur n'est trouvé, le résultat dépendra de la déclaration de l'option fall_through. Si tel est le cas, la valeur Clé est retournée comme identificateur de classe. Si cela n'est pas le cas, une erreur est retournée et le traitement continue avec les filtres restant. Faites attention si vous utilisez l'option fall_through ; ceci ne peut être fait que si une relation existe entre les valeurs de la variable skb−>tc_index et les identificateurs de classe.

Les derniers paramètres à commenter sont hash et pass_on. Le premier est relié à la taille de la table de hachage. Pass_on sera utilisé pour indiquer d'essayer le filtre suivant dans le cas où aucun identificateur de classe égal au résultat du filtre ne serait trouvé. L'action par défaut est fall_through (regarder la table suivante).

Finalement, regardons quelles sont les valeurs possibles pour la configuration de tous ces paramètres TCINDEX :

```
Nom TC                  Valeur          Défaut
-----------------------------------------------------------------
Hash                    1...0x10000     Dépendant de l'implémentation
Mask                    0...0xffff      0xffff
Shift                   0...15          0
Fall through / Pass_on  Flag            Fall_through
Classid                 Major:minor     Rien
Police                  .....           Rien
```

Ce type de filtre est très puissant. Il est nécessaire d'explorer toutes les possibilités. En outre, ce filtre n'est pas seulement utilisé dans les configurations DiffServ. Vous pouvez l'utiliser comme n'importe quel autre filtre.

Je vous recommande de regarder les exemples DiffServ inclus dans la distribution iproute2. Je vous promets que j'essaierai de compléter ce texte dès que possible. Tout ce que j'ai expliqué est le résultat de nombreux tests. Merci de me dire si je me suis trompé quelque part.

14.4. Gestionnaire de mise en file d'attente d'entrée (*Ingress qdisc*)

Tous les gestionnaires de mise en file d'attente dont nous avons discuté jusqu'ici sont des gestionnaires de sortie. Chaque interface peut également avoir un gestionnaire de mise en file d'attente d'entrée qui n'est pas utilisé pour envoyer les paquets à l'extérieur du périphérique réseau. Au lieu de cela, il vous autorise à appliquer des filtres tc aux paquets entrants par l'interface, indépendamment de s'ils ont une destination locale ou s'ils sont destinés à être transmis.

Etant donné que les filtres tc contiennent une implémentation complète du Filtre à Seau de Jetons, et qu'ils sont également capables de s'appuyer sur l'estimation du flux fourni par le noyau, il y a beaucoup de fonctionnalités disponibles. Ceci vous permet de réglementer le trafic entrant de façon efficace, avant même qu'il n'entre dans la pile IP.

14.4.1. Paramètres & usage

Le gestionnaire de mise en file d'attente d'entrée ne nécessite pas de paramètres. Il diffère des autres gestionnaires dans le fait qu'il n'occupe pas la racine du périphérique. Attachez−le comme ceci :

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

Ceci vous autorise à avoir d'autres gestionnaires de sortie sur votre périphérique en plus du gestionnaire d'entrée.

Pour un exemple inventé sur la façon dont le gestionnaire d'entrée peut être utilisé, voir le chapitre Recettes de cuisine.

14.5. *Random Early Detection* (RED)

Ce chapitre est conçu comme une introduction au routage de dorsales (backbones). Ces liaisons impliquent souvent des bandes passantes supérieures à 100 mégabits/s, ce qui nécessite une approche différente de celle de votre modem ADSL à la maison.

Le comportement normal des files d'attente de routeurs est appelé "tail−drop" (NdT : élimine le reste). Le "tail−drop" consiste à mettre en file d'attente un certain volume de trafic et à éliminer tout ce qui déborde. Ce n'est pas du tout équitable et cela conduit à des retransmissions de synchronisation. Quand une retransmission de synchronisation a lieu, la brusque rafale de rejet d'un routeur qui a atteint sa limite entraînera une rafale de retransmissions retardée qui inondera à nouveau le routeur congestionné.

Dans le but d'en finir avec les congestions occasionnelles des liens, les routeurs de dorsales intègrent souvent des files d'attente de grande taille. Malheureusement, bien que ces files d'attente offrent un bon débit, elles peuvent augmenter sensiblement les temps de latence et entraîner un comportement très saccadé des connexions TCP pendant la congestion.

Ces problèmes avec le "tail−drop" deviennent de plus en plus préoccupants avec l'augmentation de l'utilisation d'applications hostiles au réseau. Le noyau Linux nous offre la technique RED, abréviation de Random Early Detect ou détection précoce directe.

RED n'est pas la solution miracle à tous ces problèmes. Les applications qui n'intègrent pas correctement la technique de "l'exponential backoff" obtiennent toujours une part trop grande de bande passante. Cependant, avec la technique RED elles ne provoquent pas trop de dégâts sur le débit et les temps de latence des autres connexions.

RED élimine statistiquement des paquets du flux avant qu'il n'atteigne sa limite "dure" (hard). Sur une dorsale congestionnée, cela entraîne un ralentissement en douceur de la liaison et évite les retransmissions de synchronisation. La technique RED aide aussi TCP à trouver une vitesse "équitable" plus rapidement : en permettant d'éliminer des paquets plus tôt, il conserve une file d'attente plus courte et des temps de latence mieux contrôlés. La probabilité qu'un paquet soit éliminé d'une connexion particulière est proportionnelle à la bande passante utilisée par cette connexion plutôt qu'au nombre de paquets qu'elle envoie.

La technique RED est une bonne gestion de file d'attente pour les dorsales, où vous ne pouvez pas vous permettre le coût d'une mémorisation d'état par session qui est nécessaire pour une mise en file d'attente vraiment équitable.

Pour utiliser RED, vous devez régler trois paramètres : Min, Max et burst. Min est la taille minimum de la file d'attente en octets avant que les rejets n'aient lieu, Max est le maximum "doux" (soft) en−dessous duquel l'algorithme s'efforcera de rester, et burst est le nombre maximum de paquets envoyés "en rafale".

Vous devriez configurer Min en calculant le plus grand temps de latence acceptable pour la mise en file d'attente, multiplié par votre bande passante. Par exemple, sur mon lien ISDN à 64 Kbits/s, je voudrais avoir un temps de latence de base de mise en file d'attente de 200 ms. Je configure donc Min à 1600 octets (= 0,2 x 64000 / 8). Imposer une valeur Min trop petite va dégrader le débit et une valeur Min trop grande va dégrader le temps de latence. Sur une liaison lente, choisir un coefficient Min petit ne peut pas remplacer une réduction du MTU pour améliorer les temps de réponse.

Vous devriez configurer Max à au moins deux fois Min pour éviter les synchronisations. Sur des liens lents avec de petites valeurs de Min, il peut être prudent d'avoir Max quatre fois plus grand que Min ou plus.

Burst contrôle la réponse de l'algorithme RED aux rafales. Burst doit être choisi plus grand que min/avpkt (paquet moyen). Expérimentalement, j'ai trouvé que (min+min+max)/(3*avpkt) marche bien.

De plus, vous devez configurer limit et avpkt. Limit est une valeur de sécurité : s'il y a plus de Limit octets dans la file, RED reprend la technique "tail−drop". Je choisis une valeur typique égale à 8 fois Max. Avpkt devrait être fixé à la taille moyenne d'un paquet. 1000 fonctionne correctement sur des liaisons Internet haut débit ayant un MTU de 1500 octets.

Lire l'article sur la file d'attente RED par Sally Floyd et Van Jacobson pour les informations techniques.

14.6. Generic Random Early Detection

Il n'y a pas grand chose de connu sur GRED. Il ressemble à GRED avec plusieurs files d'attente internes, celles−ci étant choisies en se basant sur le champ tcindex de Diffserv. Selon une diapositive trouvée ici, il possède les capacités *Distributed Weighted RED* de Cisco, ainsi que les capacités RIO de Clark.

Chaque file d'attente virtuelle peut avoir ses propres "Drop Parameters".

FIXME: Demandez à Jamal or Werner de nous en dire plus

14.7. Emulation VC/ATM

Ceci est l'effort principal de Werner Almesberger pour vous autoriser à construire des circuits virtuels au−dessus des sockets TCP/IP. Le circuit virtuel est un concept venant de la théorie des réseaux ATM.

Pour plus d'informations, voir la page ATM sur Linux.

14.8. Weighted Round Robin (WRR)

Ce gestionnaire de mise en file d'attente n'est pas inclus dans les noyaux standards, mais peut être téléchargée à partir de http://wipl−wrr.dkik.dk/wrr/. Ce gestionnaire de mise en file d'attente n'a été testé qu'avec les noyaux 2.2, mais marchera probablement également avec les noyaux 2.4/2.5.

La file d'attente WRR partage la bande passante entre ses classes en utilisant la technique du tourniquet pondéré. Ceci est similaire à la file d'attente CBQ qui contient des classes sur lesquelles l'on peut associer arbitrairement des files d'attente. Toutes les classes qui ont suffisamment de demandes obtiendront la bande passante proportionnellement au poids associé des classes. Les poidss peuvent être configurés manuellement en utilisant le programme *tc*. Ils peuvent également être configurés pour décroître automatiquement pour les classes transférant moins de données.

La file d'attente a un classificateur intégré qui assigne les paquets venant ou allant vers différentes machines à différentes classes. On peut utiliser soit l'adresse MAC soit l'adresse IP de l'adresse source ou de destination. L'adresse MAC ne peut cependant être utilisée que quand la boite Linux est un pont ethernet. Les classes sont automatiquement assignées aux machines en fonction des paquets vus.

Ce gestionnaire de mise en file d'attente peut être très utile au site comme les résidences étudiantes où des individus sans liens particuliers partagent une connexion Internet. Un ensemble de scripts pour configurer un tel cas de figure pour ce genre de site est proposé dans la distribution WRR.

Chapitre 15. Recettes de cuisine

Cette section contient des << recettes de cuisine >> qui peuvent vous aider à résoudre vos problèmes. Un livre de cuisine ne remplace cependant pas une réelle compréhension, essayez donc d'assimiler ce qui suit.

15.1. Faire tourner plusieurs sites avec différentes SLA (autorisations)

Vous pouvez faire cela de plusieurs manières. Apache possède un module qui permet de le supporter, mais nous montrerons comment Linux peut le faire pour d'autres services. Les commandes ont été reprises d'une présentation de Jamal Hadi, dont la référence est fournie ci−dessous.

Disons que nous avons deux clients avec : http, ftp et du streaming audio. Nous voulons leur vendre une largeur de bande passante limitée. Nous le ferons sur le serveur lui−même.

Le client *A* doit disposer d'au moins 2 mégabits, et le client *B* a payé pour 5 mégabits. Nous séparons nos clients en créant deux adresses IP virtuelles sur notre serveur.

```
# ip address add 188.177.166.1 dev eth0
# ip address add 188.177.166.2 dev eth0
```

C'est à vous d'associer les différents serveurs à la bonne adresse IP. Tous les démons courants supportent cela.

Nous pouvons tout d'abord attacher une mise en file d'attente CBQ à *eth0* :

```
# tc qdisc add dev eth0 root handle 1: cbq bandwidth 10Mbit cell 8 avpkt 1000 \
  mpu 64
```

Nous créons ensuite les classes pour nos clients :

```
# 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
```

Nous ajoutons les filtres pour nos deux classes :

```
##FIXME: Pourquoi cette ligne, que fait-elle ? Qu'est-ce qu'un
diviseur ?
##FIXME: Un diviseur est lié à une table de hachage et au nombre de
seaux -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
```

Et voilà qui est fait.

FIXME: Pourquoi pas un filtre token bucket ? Y a t−il un retour par défaut à *pfifo_fast* quelque part ?

15.2. Protéger votre machine des inondations SYN

D'après la documentation iproute d'Alexey adaptée à netfilter. Si vous utilisez ceci, prenez garde d'ajuster les nombres avec des valeurs raisonnables pour votre système.

Si vous voulez protéger tout un réseau, oubliez ce script, qui est plus adapté à un hôte seul.

Il apparaît que la toute dernière version de l'outil iproute2 est nécessaire pour que ceci fonctionne avec le noyau 2.4.0.

```
#! /bin/sh -x
#
# script simple utilisant les capacités de Ingress.
# Ce script montre comment on peut limiter le flux entrant des SYN.
# Utile pour la protection des TCP-SYN. Vous pouvez utiliser IPchains
# pour bénéficier de puissantes fonctionnalités sur les SYN.
#
# chemins vers les divers utilitaires
```

```
# À changer en fonction des vôtres
#
TC=/sbin/tc
IP=/sbin/ip
IPTABLES=/sbin/iptables
INDEV=eth2
#
# marque tous les paquets SYN entrant à travers $INDEV avec la valeur 1
############################################################
$iptables -A PREROUTING -i $INDEV -t mangle -p tcp --syn \
  -j MARK --set-mark 1
############################################################
#
# installe la file d'attente ingress sur l'interface associée
############################################################
$TC qdisc add dev $INDEV handle ffff: ingress
############################################################
#
# Les paquets SYN ont une taille de 40 octets (320 bits), donc trois SYN
# ont une taille de 960 bits (approximativement 1Kbit) ; nous limitons donc
# les SYNs entrants à 3 par seconde (pas vraiment utile, mais sert à
# montrer ce 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:

#supprime la file d'attente ingress
#$TC qdisc del $INDEV ingress
```

15.3. Limiter le débit ICMP pour empêcher les dénis de service

Récemment, les attaques distribuées de déni de service sont devenues une nuisance importante sur Internet. En filtrant proprement et en limitant le débit de votre réseau, vous pouvez à la fois éviter de devenir victime ou source de ces attaques.

Vous devriez filtrer vos réseaux de telle sorte que vous n'autorisiez pas les paquets avec une adresse IP source non−locale à quitter votre réseau. Cela empêche les utilisateurs d'envoyer de manière anonyme des cochonneries sur Internet.

La limitation de débit peut faire encore mieux, comme vu plus haut. Pour vous rafraîchir la mémoire, revoici notre diagramme ASCII :

```
[Internet] ---<E3, T3, n'importe quoi>--- [routeur Linux] --- [Bureau+FAI]
                                         eth1          eth0
```

Nous allons d'abord configurer les parties pré−requises :

```
# 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 vous avez des interfaces de 100 Mbits ou plus, ajustez ces nombres. Maintenant, vous devez déterminer combien de trafic ICMP vous voulez autoriser. Vous pouvez réaliser des mesures avec tcpdump, en écrivant les résultats dans un fichier pendant un moment, et regarder combien de paquets ICMP passent par votre réseau. Ne pas oublier d'augmenter la longueur du "snapshot". Si la mesure n'est pas possible, vous pouvez consacrer par exemple 5% de votre bande passante disponible. Configurons notre classe :

```
# 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
```

Cela limite le débit à 100 Kbits sur la classe. Maintenant, nous avons besoin d'un filtre pour assigner le trafic ICMP à cette classe :

```
# tc filter add dev eth0 parent 10:0 protocol ip prio 100 u32 match ip
  protocol 1 0xFF flowid 10:100
```

15.4. Donner la priorité au trafic interactif

Si beaucoup de données arrivent à votre lien ou en partent, et que vous essayez de faire de la maintenance via telnet ou ssh, cela peut poser problème : d'autres paquets bloquent vos frappes clavier. Cela ne serait−il pas mieux si vos paquets interactifs pouvaient se faufiler dans le trafic de masse ? Linux peut faire cela pour vous.

Comme précédemment, nous avons besoin de manipuler le trafic dans les deux sens. Evidemment, cela marche mieux s'il y a des machines Linux aux deux extrémités du lien, bien que d'autres UNIX soient capables de faire la même chose. Consultez votre gourou local Solaris/BSD pour cela.

Le gestionnaire standard pfifo_fast a trois "bandes" différentes. Le trafic de la bande 0 est transmis en premier, le trafic des bandes 1 et 2 étant traité après. Il est vital que votre trafic interactif soit dans la bande 0 ! Ce qui suit est adapté du (bientôt obsolète) Ipchains−HOWTO :

Il y a quatre bits rarement utilisés dans l'en−tête IP, appelés bits de Type de Service (TOS). Ils affectent la manière dont les paquets sont traités. Les quatre bits sont "Délai Minimum", "Débit Maximum", "Fiabilité Maximum" et "Coût Minimum". Seul un de ces bits peut être positionné. Rob van Nieuwkerk, l'auteur du code TOS−mangling dans ipchains, le configure comme suit :

```
Le "Délai Minimum" est particulièrement important pour moi. Je le
positionne à 1 pour les paquets interactifs sur mon routeur (Linux)
qui envoie le trafic vers l'extérieur. Je suis derrière un modem à
33,6 Kbps. Linux répartit les paquets dans trois files
d'attente. De cette manière, j'obtiens des performances acceptables
pour le trafic interactif tout en téléchargeant en même temps.
```

L'utilisation la plus commune est de configurer les connexions telnet et ftp à "Délai Minimum" et les données FTP à "Débit Maximum". Cela serait fait comme suit, sur mon routeur :

```
# 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
```

En fait, cela ne marche que pour les données venant d'un telnet extérieur vers votre ordinateur local. Dans l'autre sens, ça se fait tout seul : telnet, ssh, et consorts configurent le champ TOS automatiquement pour les paquets sortants.

Si vous avez un client incapable de le faire, vous pouvez toujours le faire avec netfilter. Sur votre machine locale :

```
# 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. Cache web transparent utilisant netfilter, iproute2, ipchains et squid

Cette section a été envoyée par le lecteur Ram Narula de "Internet for Education" (Internet pour l'éducation) (Thailande).

La technique habituelle pour réaliser ceci dans Linux est probablement l'utilisation d'ipchains APRES s'être assuré que le trafic sortant du port 80 (web) est routé à travers le serveur faisant fonctionner squid.

Il y a 3 méthodes communes pour être sûr que le trafic sortant du port 80 est routé vers le serveur faisant fonctionner squid et une quatrième est introduite ici.

La passerelle le fait.
: Si vous pouvez dire à votre passerelle que les paquets sortants à destination du port 80 doivent être envoyés vers l'adresse IP du serveur squid.

  MAIS

  Ceci amènerait une charge supplémentaire sur le routeur et des routeurs commerciaux peuvent même ne pas supporter ceci.

Utiliser un commutateur Couche 4.

Les commutateurs Couche 4 peuvent manipuler ceci sans aucun problème.

MAIS

Le coût pour un tel équipement est en général très élevé. Typiquement, un commutateur couche 4 coûte normalement plus cher qu'un serveur classique + un bon serveur linux.

Utiliser le serveur cache comme passerelle réseau
: Vous pouvez forcer TOUT le trafic à travers le serveur cache

MAIS

Ceci est plutôt risqué dans la mesure où Squid utilise beaucoup de ressources CPU, ce qui peut conduire à une baisse des performances de tout le réseau. Le serveur peut également ne plus fonctionner et personne sur le réseau ne pourra accéder à Internet si cela a lieu.

Routeur Linux+NetFilter.
: En utilisant Netfilter, une autre technique peut être implémentée. Celle−ci consiste à utiliser Netfilter pour "marquer" les paquets à destination du port 80 et à utiliser iproute2 pour router les paquets "marqués" vers le serveur Squid.

```
|----------------|
| Implémentation |
|----------------|

 Adresses utilisées
 10.0.0.1 naret (serveur NetFilter)
 10.0.0.2 silom (serveur Squid)
 10.0.0.3 donmuang (routeur connecté à Internet)
 10.0.0.4 kaosarn (un autre serveur sur le réseau)
 10.0.0.5 RAS
 10.0.0.0/24 réseau principal
 10.0.0.0/19 réseau total

|----------------|
|Schéma du réseau|
|----------------|

Internet
|
donmuang
|
------------hub/commutateur----------
|        |             |       |
naret   silom        kaosarn  RAS etc.
```

Premièrement, faire en sorte que tout le trafic passe par naret en étant sûr que c'est la passerelle par défaut, à l'exception de silom. La passerelle par défaut de silom doit être donmuang (10.0.0.3) ou ceci créerait une boucle du trafic web.

(Tous les serveurs sur mon réseau avaient 10.0.0.1 comme passerelle par défaut qui était l'ancienne adresse du routeur donmuang. Cela m'a conduit à attribuer 10.0.0.3 comme adresse IP à donmuang et à donner 10.0.0.1 comme adresse IP à naret.)

```
Silom
-----
-configurer squid et ipchains
```

Pour la configuration du serveur Squid sur silom, soyez sûr que celui−ci supporte le cache/proxy transparent (transparent caching/proxying). Le port par défaut pour squid est en général 3128. Tout le trafic pour le port 80 doit donc être redirigé localement vers le port 3128. Ceci peut être fait en utilisant ipchains comme suit :

```
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
```

Ou, avec netfilter:

```
silom# iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 80 -j REDIRECT --to-port 3128
```

(note: vous pouvez avoir également d'autres entrées)

Pour plus d'informations sur la configuration du serveur Squid, se référer à la page FAQ Squid sur http://squid.nlanr.net).

Soyez sûr que l''ip forwarding" est actif sur votre serveur et que la passerelle par défaut pour ce serveur est donmuand (PAS naret).

```
Naret
-----
- configurer squid et ipchains
- désactiver les messages icmp REDIRECT (si besoin)
```

1. "Marquer" les paquets à destination du port 80 avec la valeur 2

   ```
   naret# iptables -A PREROUTING -i eth0 -t mangle -p tcp --dport 80 \
    -j MARK --set-mark 2
   ```

2. Configurer iproute2 de sorte qu'il route les paquets avec la marque 2 vers silom

   ```
   naret# echo 202 www.out >> /etc/iproute2/rt_tables
   naret# ip rule add fwmark 2 table www.out
   naret# ip route add default via 10.0.0.2 dev eth0 table www.out
   naret# ip route flush cache
   ```

   Si donmuang et naret sont sur le même réseau, naret ne doit pas envoyer de messages icmp REDIRECT. Ceux−ci doivent être désactivés par :

   ```
   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
```

La configuration est terminée, vérifions−la.

```
Sur 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

(soyez sûr que silom appartiens à l'une des lignes ci-dessus. Dans ce cas,
c'est la ligne avec 10.0.0.0/24)

|------|
|-FAIT-|
|------|
```

15.5.1. Schéma du trafic après l'implémentation

```
|----------------------------------------|
|Schéma du trafic après l'implémentation|
|----------------------------------------|

INTERNET
/\
||
\/
-----------------routeur donmuang---------------------
/\                                    /\          ||
||                                    ||          ||
```

```
||                                      \/         ||
naret                                  silom       ||
*trafic à destination du port 80=====>(cache)      ||
/\                                      ||         ||
||                                      \/         \/
\\==================================kaosarn, RAS, etc.
```

Noter que le réseau est asymétrique car il y a un saut supplémentaire sur le chemin sortant.

```
Voici le cheminement d'un paquet traversant le réseau de kaosarn allant et
venant d'Internet.

Pour le trafic web/http :
requête http kaosarn->naret->silom->donmuang->Internet
réponse http de Internet->donmuang->silom->kaosarn

Pour les requêtes non web/http (par ex. telnet) :
données sortantes kaosarn->naret->donmuang->Internet
données entrantes d'Internet->donmuang->kaosarn
```

15.6. Circonvenir aux problèmes de la découverte du MTU de chemin en configurant un MTU par routes

Pour envoyer de grande quantité de données, Internet fonctionne généralement mieux quand de grands paquets sont utilisés. Chaque paquet implique une décision de routage. Le transfert d'un fichier de 1Mo peut entraîner soit l'envoi de 700 paquets, en maximisant la taille des paquets, soit de 4000 paquets en utilisant la plus petite taille possible.

Cependant, tous les éléments d'Internet ne supportent pas une capacité utile (payload) de 1460 octets par paquet. Il est de plus nécessaire d'essayer de trouver le plus grand paquet qui "conviendra" le mieux, dans le but d'optimiser la connexion.

Ce processus est appelé "Découverte du MTU de chemin", où MTU signifie 'Maximum Transfert Unit' (Unité de transfert maximum).

Quand un routeur rencontre un paquet qui est trop gros pour être envoyé en un seul morceau, ET que celui−ci a été marqué avec le bit "Don't Fragment", il retourne un message ICMP indiquant qu'il a été obligé de rejeter le paquet. L'hôte émetteur prend acte de cette indication en envoyant des paquets plus petits et, par itération, peut trouver la taille optimum du paquet pour une connexion à travers un chemin particulier.

Ceci fonctionnait correctement jusqu'à ce que Internet soit découvert par des vandales qui s'efforcent de perturber les communications. Ceci a conduit les administrateurs à, soit bloquer, soit mettre en forme le trafic ICMP lors d'essais malencontreux dans le but d'améliorer la sécurité ou la robustesse de leurs services Internet.

La conséquence, maintenant, est que la découverte du MTU de chemin fonctionne de moins en moins bien, et échoue pour certaines routes, conduisant à d'étranges sessions TCP/IP qui tombent peu de temps après.

Bien que je n'aie pas de preuves de ceci, deux sites avec qui j'avais l'habitude d'avoir des problèmes faisaient fonctionner à la fois Alteon et Acedirectors avant les systèmes affectés. Peut−être quelqu'un avec plus de connaissances peut fournir des indices quant à la raison de ce qui se passe.

15.6.1. Solution

Quand vous rencontrez des sites qui présentent ce problème, vous pouvez désactiver la découverte du MTU de chemin en le configurant manuellement. Koos van den Hout a à peu près écrit :

> Le problème suivant : j'ai configuré le mtu et mru de ma ligne dédiée fonctionnant avec ppp à 296 dans la mesure où le débit est de seulement 33k6 et que je ne peux pas influencer la file d'attente de l'autre côté. A 296, la réponse à l'appui d'une touche intervient dans un délai raisonnable.
>
> Et, de mon côté, j'ai un routeur avec translation d'adresse (masquage) fonctionnant (bien sûr) sous Linux.
>
> Récemment, j'ai séparé le serveur du routeur de sorte que la plupart des applications fonctionnent sur une machine différente de celle qui réalise le routage.
>
> J'ai alors eu des problèmes en me connectant sur l'irc. Grosse panique ! Je vous assure que certains essais trouvaient que j'étais connecté à l'irc, me montrant même comme connecté sur l'irc mais je ne recevais pas le "motd" (message of the day, message du jour) de l'irc. J'ai vérifié ce qui pouvait être erroné et ai noté que j'avais déjà eu des soucis liés au MTU en contactant certains sites web. Je n'avais aucun souci pour les atteindre quand le MTU était à 1500, le problème n'apparaissant que lorsque le MTU était configuré à 296. Puisque les serveurs irc bloquent tout le trafic dont il n'ont pas besoin pour leurs opérations immédiates, ils bloquent aussi l'icmp.
>
> J'ai manoeuvré pour convaincre les responsables d'un serveur web que ceci était la cause d'un problème, mais les responsables du serveur irc n'avaient pas l'intention de réparer ceci.
>
> Donc, je devais être sûr que le trafic masqué sortant partait avec le mtu faible du lien externe. Mais, je voulais que le trafic ethernet local ait le MTU normal (pour des choses comme le trafic nfs).
>
> Solution :
>
> ```
> ip route add default via 10.0.0.1 mtu 296
> ```
>
> (10.0.0.1 étant ma passerelle par défaut, l'adresse interne de mon routeur masquant)

En général, il est possible d'outrepasser la découverte du MTU de chemin en configurant des routes spécifiques. Par exemple, si seuls certains réseaux posent problèmes, ceci devrait aider :

```
ip route add 195.96.96.0/24 via 10.0.0.1 mtu 1000
```

15.7. Circonvenir aux problèmes de la découverte du MTU de chemin en imposant le MSS (pour les utilisateurs de l'ADSL, du câble, de PPPoE & PPtP)

Comme expliqué au−dessus, la découverte du MTU de chemin ne marche pas aussi bien que cela devrait être. Si vous savez qu'un saut de votre réseau a un MTU limité (<1500), vous ne pouvez pas compter sur la découverte du MTU de chemin pour le découvrir.

Outre le MTU, il y a encore un autre moyen de configurer la taille maximum du paquet, par ce qui est appelé le MSS (Maximum Segment Size, Taille Maximum du Segment). C'est un champ dans les options TCP du paquet SYN.

Les noyaux Linux récents, et quelques pilotes de périphérique pppoe (notamment, l'excellent Roaring Penguin) implémente la possibilité de 'fixer le MSS'.

Le bon côté de tout ceci est que, en positionnant la valeur MSS, vous dîtes à l'hôte distant de manière équivoque "n'essaie pas de m'envoyer des paquets plus grands que cette valeur". Aucun trafic ICMP n'est nécessaire pour faire fonctionner cela.

Malheureusement, c'est de la bidouille évidente −− ça détruit la propriété «bout−en−bout» de la connexion en modifiant les paquets. Ayant dit cela, nous utilisons cette astuce dans beaucoup d'endroit et cela fonctionne comme un charme.

Pour que tout ceci fonctionne, vous aurez besoin au moins de iptables−1.2.1a et de Linux 2.4.3 ou plus. La ligne de commande basique est :

```
# iptables -A FORWARD -p tcp --tcp-flags SYN,RST SYN -j TCPMSS  --clamp-mss-to-pmtu
```

Ceci calcule le MSS approprié pour votre lien. Si vous vous sentez courageux ou que vous pensez être le mieux placé pour juger, vous pouvez aussi faire quelque chose comme ceci :

```
# iptables -A FORWARD -p tcp --tcp-flags SYN,RST SYN -j TCPMSS --set-mss 128
```

Ceci configure le MSS du paquet SYN à 128. Utilisez ceci si vous avez de la voix sur IP (VoIP) avec de tous petits paquets, et de grands paquets http qui provoquent des coupures dans vos communications vocales.

15.8. Le Conditionneur de Trafic Ultime : Faible temps de latence, Téléchargement vers l'amont et l'aval rapide

Note : Ce script a récemment été mis à niveau et il ne marchait avant qu'avec les clients Linux de votre réseau ! Vous devriez donc le mettre à jour si vous avez des machines Windows ou des Macs dans votre réseau qui n'étaient pas capables de télécharger plus rapidement pendant que d'autres étaient en train de télécharger vers l'amont.

J'ai essayé de créer le Saint Graal :

Maintenir à tout moment un faible temps de latence pour le trafic interactif
: Ceci signifie que la récupération ou la transmission de fichiers ne doivent pas perturber SSH ou même telnet. Ceci est la chose la plus importante, car même un temps de latence de 200ms est important pour pouvoir travailler confortablement.

Autoriser le 'surf' à des vitesses raisonnables pendant que l'on télécharge vers l'amont ou vers l'aval

Même si http est un trafic de masse, les autres trafics ne doivent pas trop le noyer.

Etre sûr que le téléchargement vers l'amont ne va pas faire du tort aux téléchargements vers l'aval et aux autres éléments autour

Le principal phénomène observé est la forte réduction de la vitesse de téléchargement vers l'aval quand il y a du trafic montant.

Il s'avère que tout ceci est possible, au prix d'une minuscule réduction de la bande passante. La présence de grandes files d'attente sur les dispositifs d'accès domestiques, comme le câble ou les modems DSL, explique pourquoi les téléchargements vers l'amont, vers l'aval et ssh se pénalisent mutuellement.

La prochaine partie explique en profondeur ce qui provoque les retards, et comment nous pouvons les corriger. Vous pouvez sans danger la passer et aller directement au script si vous vous fichez de la façon dont la magie opère.

15.8.1. Pourquoi cela ne marche t−il pas bien par défaut ?

Les FAI savent que leurs performances ne sont seulement jugées que sur la vitesse à laquelle les personnes peuvent télécharger vers l'aval. En plus de la bande passante disponible, la vitesse de téléchargement est lourdement influencée par la perte des paquets, qui gêne sérieusement les performances de TCP/IP. Les grandes files d'attente peuvent aider à prévenir la perte des paquets, et augmenter les téléchargements. Les FAI configurent donc de grandes files d'attente.

Ces grandes files d'attente endommagent cependant l'interactivité. Une frappe doit d'abord parcourir la file d'attente du flux montant, ce qui peut prendre plusieurs secondes, et aller jusqu'au hôte distant. Elle est alors traitée, conduisant à un paquet de retour qui doit traverser la file d'attente du flux descendant, localisée chez votre FAI, avant qu'elle n'apparaisse sur l'écran.

Cet HOWTO nous enseigne plusieurs manières pour modifier et traiter la file d'attente mais, malheureusement, toutes les files d'attente ne nous sont pas accessibles. Les files d'attente du FAI sont sans limites et la file d'attente du flux montant réside probablement dans votre modem câble ou votre périphérique DSL. Il se peut que vous soyiez capable ou non de le configurer. La plupart du temps, ce ne sera pas le cas.

Et après ? Etant donné que nous ne pouvons pas contrôler ces files d'attente, elles doivent disparaître et être transférées sur notre routeur Linux. Heureusement, ceci est possible.

Limiter la vitesse de téléchargement vers l'amont (upload)

En limitant notre vitesse de téléchargement vers l'amont à une vitesse légèrement plus faible que la vitesse réelle disponible, il n'y a pas de files d'attente mises en place dans notre modem. La file d'attente est maintenant transférée vers Linux.

Limiter la vitesse de téléchargement vers l'aval (download)

Ceci est légèrement plus rusé dans la mesure où nous ne pouvons pas vraiment influencer la vitesse à laquelle Internet nous envoie les données. Nous pouvons cependant rejetter les paquets qui arrivent trop vite, ce qui provoque le ralentissement de TCP/IP jusqu'au débit désiré. Parceque que nous ne voulons pas supprimer inutilement du trafic, nous configurons une vitesse de rafale ('burst') plus grande.

Maintenant que nous avons fait ceci, nous avons éliminé totalement la file d'attente du flux descendant (sauf pour de courtes rafales de données), et obtenu la possibilité de gérer la file d'attente du flux montant avec toute la puissance Linux.

Il nous reste à nous assurer que le trafic interactif se retrouve au début de la file d'attente du flux montant. Pour être sûr que le flux montant ne va pas pénaliser le flux descendant, nous déplaçons également les paquets ACK au début de la file d'attente. C'est ce qui provoque normalement un énorme ralentissement quand du trafic de masse est généré dans les deux sens. Les paquets ACK du trafic descendant rentrent en concurrence avec le trafic montant et sont donc ralentis.

Si nous avons fait tout ceci, nous obtenons les mesures suivantes en utilisant l'excellente connexion ADSL de xs4all, en Hollande :

```
Temps de latence de base :
round-trip min/avg/max = 14.4/17.1/21.7 ms

Sans le conditionneur de trafic, lors d'un télechargement vers l'aval :
round-trip min/avg/max = 560.9/573.6/586.4 ms

Sans le conditionneur de trafic, lors d'un télechargement vers l'amont :
round-trip min/avg/max = 2041.4/2332.1/2427.6 ms

Avec le conditionneur, lors d'un télechargement vers l'amont à 220kbit/s :
round-trip min/avg/max = 15.7/51.8/79.9 ms

Avec le conditionneur, lors d'un télechargement vers l'aval à 850kbit/s :
round-trip min/avg/max = 20.4/46.9/74.0 ms

Lors d'un téléchargement vers l'amont, les téléchargements vers l'aval s'effectuent à environ
80 % de la vitesse maximale disponible et 90% pour les téléchargements vers
l'amont. Le temps de latence augmente alors jusqu'à 850 ms ; je n'ai pas encore
déterminé la raison de ce phénomène.
```

Ce que vous pouvez attendre de ce script dépend largement de votre vitesse de lien réelle. Quand vous téléchargez vers l'amont à pleine vitesse, il y aura toujours un paquet devant votre frappe de clavier. Ceci est la limite basse de votre temps de latence. Pour la calculer, divisez votre MTU par la vitesse du flux montant. Les valeurs classiques seront un peu plus élevées que ca. Diminuez votre MTU pour un meilleur effet !

Voici deux versions de ce script, une avec l'excellent HTB de Devik, et l'autre avec CBQ qui est présent dans chaque noyau Linux, contrairement à HTB. Les deux ont été testés et marchent correctement.

15.8.2. Le script (CBQ)

Marche avec tous les noyaux. A l'intérieur du gestionnaire de mise en file d'attente CBQ, nous plaçons deux SFQ pour être sûr que de multiples flux de masse ne vont pas mutuellement se pénaliser.

Le trafic descendant est réglementé en utilisant un filtre tc contenant un Token Bucket Filter.

Vous pourriez améliorer ce script en ajoutant 'bounded' aux lignes qui démarrent avec 'tc class add .. classid 1:20'. Si vous avez diminué votre MTU, diminuez aussi les nombres allot & avpkt !

```
#!/bin/bash

# La configuration ultime pour votre connexion Internet domestique
#
# Configurez les valeurs suivantes avec des valeurs légèrement inférieures que
# vos vitesses de flux montant et descendant. Exprimé en kilobits.
DOWNLINK=800
UPLINK=220
DEV=ppp0

```

```
# Nettoie les gestionnaires de sortie et d'entrés, cache les erreurs
tc qdisc del dev $DEV root    2> /dev/null > /dev/null
tc qdisc del dev $DEV ingress 2> /dev/null > /dev/null

###### Flux montant (uplink)

# installe CBQ à la racine

tc qdisc add dev $DEV root handle 1: cbq avpkt 1000 bandwidth 10mbit

# Le trafic est mis en forme à une vitesse de $UPLINK. Ceci évite
# d'énormes files d'attente dans votre modem DSL qui pénalisent le temps de
# latence.
# Classe principale

tc class add dev $DEV parent 1: classid 1:1 cbq rate ${UPLINK}kbit \
allot 1500 prio 5 bounded isolated

# classe de priorité supérieure 1:10:

tc class add dev $DEV parent 1:1 classid 1:10 cbq rate ${UPLINK}kbit \
   allot 1600 prio 1 avpkt 1000

# la classe par défaut et pour le trafic de masse 1:20. Reçoit légèrement
# moins que le trafic et a une priorité plus faible :
# 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

# Les deux sont gérées par SFQ :
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

# Démarrage des filtres
# le bit Délai Minimum du champ TOS (ssh, PAS scp) est dirigé vers
# 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) est dirigé vers la classe interactive 1:10 de telle
# sorte que nous pouvons réaliser des mesures et impressionner nos
# amis :
tc filter add dev $DEV parent 1:0 protocol ip prio 11 u32 \
        match ip protocol 1 0xff flowid 1:10

# Pour accélérer les téléchargements vers l'aval lors de la présence d'un
# flux montant, les paquets ACK sont placés dans la classe
# interactive :

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

# Le reste est considéré 'non-interactif' cad 'de masse' et fini dans 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

########## Flux descendant (downlink) #############
# Ralentir le flux descendant à une valeur légèrement plus faible que votre
# vitesse rélle de manière à éviter la mise en file d'attente chez notre
```

```
# FAI. Faites des tests pour voir la vitesse maximum à laquelle vous pouvez
# le configurer. Les FAI ont tendance à avoir *d'énormes* files d'attente
# pour s'assurer de la rapidité des gros téléchargements.
#
# attache la réglementation d'entrée (ingress policer) :

tc qdisc add dev $DEV handle ffff: ingress

# Filtre *tout* (0.0.0.0/0), rejette tout ce qui arrive trop
# rapidement :

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 vous voulez que ce script soit exécuté par ppp à la connexion, copiez−le dans /etc/ppp/ip−up.d.

Si les deux dernières lignes conduisent à une erreur, mettez à jour l'outil tc avec la dernière version !

15.8.3. Le script (HTB)

Le script suivant nous permet d'atteindre tous nos buts en utilisant la merveilleuse file d'attente HTB. Voir le chapitre correspondant. Cela vaut la peine de mettre à jour votre noyau !

```
#!/bin/bash

# La configuration ultime pour votre connexion Internet domestique
#
# Configurez les valeurs suivantes avec des valeurs légèrement inférieures que
# vos vitesses de flux montant et descendant. Exprimé en kilobits.
DOWNLINK=800
UPLINK=220
DEV=ppp0

#Nettoie les gestionnaires de sortie et d'entrés, cache les erreurs
tc qdisc del dev $DEV root    2> /dev/null > /dev/null
tc qdisc del dev $DEV ingress 2> /dev/null > /dev/null

###### Flux montant (uplink)

# installe HTB à la racine, le trafic ira par défaut vers 1:20 :

tc qdisc add dev $DEV root handle 1: htb default 20

# Le trafic est mis en forme à une vitesse de $UPLINK. Ceci évite
# d'énormes files d'attente dans votre modem DSL qui pénalisent le temps de
# latence.

tc class add dev $DEV parent 1: classid 1:1 htb rate ${UPLINK}kbit burst 6k

# la classe de haute priorité 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

# Les deux sont gérées par SFQ :
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
```

```
# le bit Délai Minimum du champ TOS (ssh, PAS scp) est dirigé vers
# 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) est dirigé vers la classe interactive 1:10 de telle
# sorte que nous pouvons réaliser des mesures et impressionner nos
# amis :
tc filter add dev $DEV parent 1:0 protocol ip prio 10 u32 \
        match ip protocol 1 0xff flowid 1:10

# Pour accélérer les téléchargements vers l'aval lors de la présence d'un
# flux montant, les paquets ACK sont placés dans la classe
# interactive :

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

# Le reste est considéré 'non-interactif' cad 'de masse' et fini dans 1:20


########## Flux descendant (downlink) #############
# Ralentir le flux descendant à une valeur légèrement plus faible que votre
# vitesse rélle de manière à éviter la mise en file d'attente chez notre
# FAI. Faites des tests pour voir la vitesse maximum à laquelle vous pouvez
# le configurer. Les FAI ont tendance à avoir *d'énormes* files d'attente
# pour s'assurer de la rapidité des gros téléchargements.
#
# attache la réglementation d'entrée (ingress policer) :

tc qdisc add dev $DEV handle ffff: ingress

# Filtre *tout* (0.0.0.0/0), rejette tout ce qui arrive trop
# rapidement :

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 vous voulez que ce script soit exécuté par ppp à la connexion, copiez−le dans /etc/ppp/ip−up.d.

Si les deux dernières lignes conduisent à une erreur, mettez à jour l'outil tc avec la dernière version !

Chapitre 16. Construire des ponts et des pseudo−ponts avec du Proxy ARP

Les ponts sont des périphériques qui peuvent être installés dans un réseau sans aucune reconfiguration. Un commutateur réseau est basiquement un pont multi−ports. Un pont est souvent un commutateur avec 2 ports. Cependant, Linux supporte très bien plusieurs interfaces dans un pont, le conduisant à fonctionner comme un vrai commutateur.

Les ponts sont souvent déployés quand on est confronté à un réseau défaillant qui a besoin d'être réparé sans aucune modification. Dans la mesure où un pont est un équipement de niveau 2, la couche sous la couche IP, les routeurs et serveurs ne sont pas conscients de son existence. Ceci signifie que vous pouvez bloquer ou modifier certains paquets de manière transparente ou mettre en forme le trafic.

Un autre élément positif est qu'un pont peut souvent être remplacé par un câble croisé ou un hub quand il tombe en panne.

L'aspect négatif est que la mise en place d'un pont peut engendrer beaucoup de confusion, à moins qu'il ne soit très bien configuré. Le pont n'apparaît pas dans les traceroute, mais pourtant des paquets disparaissent sans raison ou sont changés en allant d'un point A à un point B. Vous devriez également vous demander si une organisation qui "ne veut rien changer" fait le bon choix.

Le *pont Linux 2.4* est documenté sur cette page.

16.1. Etat des ponts et iptables

Au moment de Linux 2.4.14, le pont et iptables ne se "voient" pas l'un l'autre sans une aide. Si vous "pontez" les paquets de eth0 à eth1, ils ne "passent" pas par iptables. Ceci signifie que vous ne pouvez pas faire de filtrage, de translation d'adresse (NAT), de dessossage ou quoique ce soit d'autres.

Il y a plusieurs projets continuant de corriger ceci, le meilleur étant celui de l'auteur du code du pont Linux 2.4, Lennert Buytenhek. il nous a récemment informé qu'à partir de bridge−nf 0.0.2 (voir l'url ci−dessus), le code est stable et utilisable dans un environnement de production. Il demande maintenant aux responsables du noyau si et comment la mise à jour peut être ajoutée. Rester branché !

Nous comptons que ce problème soit bientôt résolu.

16.2. Pont et mise en forme

Ca marche comme dans les réclames. Soyez sûr du côté attribué à chaque interface. Autrement, il se peut que vous mettiez en forme le trafic sortant au niveau de votre interface interne, ce qui ne marchera pas. Utilisez tcpdump si nécessaire.

16.3. Pseudo−pont avec du Proxy−ARP

Si vous voulez juste implémenter un pseudo−pont, allez jusqu'à la section "Implémentez−le". Cependant, il est sage de lire un peu la façon dont il fonctionne en pratique.

Un pseudo−pont travaille de manière un peu différente. Par défaut, un pont transmet les paquets sans les altérer d'une interface à une autre. Il ne regarde que l'adresse matérielle des paquets pour déterminer où ils doivent aller. Ceci signifie que vous pouvez "pontez" un trafic que Linux ne comprend pas, aussi longtemps qu'il y a une adresse matérielle.

Un "pseudo−pont" travaille différemment et ressemble plus à un routeur caché qu'à un pont. Mais, comme un pont, il a un impact faible sur l'architecture du réseau.

Le fait qu'il ne soit pas un pont présente l'avantage que les paquets traversent réellement le noyau, et peuvent être filtrés, modifiés, redirigés ou reroutés.

Un pont réel peut également réaliser ces tours de force, mais il a besoin d'un code spécial, comme le Ethernet Frame Diverter ou la mise à jour mentionnée au−dessus.

Un autre avantage d'un pseudo−pont est qu'il ne transmet pas les paquets qu'il ne comprend pas, nettoyant ainsi votre réseau de beaucoup de cochonneries. Dans le cas où vous auriez besoin de ces cochonneries (comme les paquets SAP ou Netbeui), utilisez un vrai pont.

16.3.1. ARP & Proxy−ARP

Quand un hôte veut dialoguer avec un autre hôte sur le même segment physique, il envoie un paquet du Protocole de Résolution d'Adresse (ARP) qui, en simplifiant quelque peu, est lu comme ceci : "Qui a 10.0.0.1, le dire à 10.0.0.7". En réponse à ceci, 10.0.0.1 renvoie un petit paquet "ici".

10.0.0.7 envoie alors des paquets à l'adresse matérielle mentionnée dans le paquet "ici". Il met dans un cache cette adresse matérielle pour un temps relativement long et, après l'expiration du cache, repose sa question.

Quand on construit un pseudo−pont, on configure le pont pour qu'il réponde à ces paquets ARP, les hôtes du réseau envoyant alors leurs paquets au pont. Le pont traite alors ces paquets et les envoie à l'interface adaptée.

Donc, en résumé, quand un hôte d'un côté du pont demande l'adresse matérielle d'un hôte se situant de l'autre côté, le pont répond avec un paquet qui dit "transmets le moi".

De cette façon, tout le trafic de données est transmis à la bonne place et il traverse toujours le pont.

16.3.2. Implémentez−le

Les versions anciennes du noyau linux permettait de faire du proxy ARP uniquement à une granularité sous réseaux. Ainsi, pour configurer un pseudo−pont, il fallait spécifier les bonnes routes vers les deux côtés du pont, et également créer les règles proxy−ARP correspondantes. C'était pénible, déjà par la quantité de texte qu'il fallait taper, puis parce qu'il était facile de se tromper et créer des configurations erronées, où le pont répondait à des requêtes pour des réseaux qu'il ne savait pas router.

Avec Linux 2.4 (et peut−être bien le 2.2), cette possibilité a été retirée et a été remplacée par une option dans le répertoire /proc, appelée "proxy−arp". La procédure pour construire un pseudo−pont est maintenant :

1. Assigner une adresse à chaque interface, la "gauche" et la "droite"
2. Créer des routes pour que votre machine connaisse quels hôtes résident à gauche et quels hôtes résident à droite
3. Activer le proxy−ARP sur chaque interface echo 1 > /proc/sys/net/ipv4/conf/ethL/proxy_arp echo 1 > /proc/sys/net/ipv4/conf/ethR/proxy_arp où L et R désignent les numéros de l'interface du côté gauche (Left) et de celle du côté droit (Right)

N'oubliez pas également d'activer l'option ip_forwarding ! Quand on convertit un vrai pont, il se peut que vous trouviez cette option désactivée dans la mesure où il n'y en a pas besoin pour un pont.

Une autre chose que vous devriez considérer lors de la conversion est que vous aurez besoin d'effacer le cache arp des ordinateurs du réseau. Le cache arp peut contenir d'anciennes adresses matérielles du pont qui ne sont plus correctes.

Sur un Cisco, ceci est réalisé en utilisant la commande 'clear arp−cache' et, sous linux, en utilisant 'arp −d ip.adresse'. Vous pouvez aussi attendre l'expiration manuelle du cache, ce qui peut être plutôt long.

Il se peut que vous découvriez également que votre réseau était mal configuré si vous avez/aviez l'habitude de spécifier les routes sans les masques de sous−réseau. Dans le passé, certaines versions de **route** pouvaient correctement deviner le masque ou, au contraire, se tromper sans pour autant vous le notifier. Quand vous faites du routage chirurgical comme décrit plus haut, il est *vital* que vous vérifiez vos masques de sous−réseau.

Chapitre 17. Routage Dynamique − OSPF et BGP

Si votre réseau commence à devenir vraiment gros ou si vous commencez à considérer Internet comme votre propre réseau, vous avez besoin d'outils qui routent dynamiquement vos données. Les sites sont souvent reliés entre eux par de multiples liens, et de nouveaux liens surgissent en permanence.

L'Internet utilise la plupart du temps les standards OSPF et BGP4 (rfc1771). Linux supporte les deux, par le biais de *gated* et *zebra*.

Ce sujet est pour le moment hors du propos de ce document, mais laissez−nous vous diriger vers des travaux de référence :

Vue d'ensemble :

Cisco Systems Cisco Systems Designing large−scale IP Internetworks

Pour OSPF :

Moy, John T. "OSPF. The anatomy of an Internet routing protocol" Addison Wesley. Reading, MA. 1998.

Halabi a aussi écrit un très bon guide sur la conception du routage OSPF, mais il semble avoir été effacé du site Web de Cisco.

Pour BGP :

Halabi, Bassam "Internet routing architectures" Cisco Press (New Riders Publishing). Indianapolis, IN. 1997.

Il existe aussi

Cisco Systems

Using the Border Gateway Protocol for Interdomain Routing

Bien que les exemples soient spécifiques à Cisco, ils sont remarquablement semblables au langage de configuration de Zebra :−)

Chapitre 18. Autres possibilités

Ce chapitre est une liste des projets ayant une relation avec le routage avancé et la mise en forme du trafic sous Linux. Certains de ces liens mériteraient des chapitres spécifiques, d'autres sont très bien documentés, et n'ont pas besoin de HOWTO en plus.

Implémentation VLAN 802.1Q pour Linux (site)
: VLAN est une façon très sympa de diviser vos réseaux d'une manière plus virtuelle que physique. De bonnes informations sur les VLAN pourront être trouvées ici. Avec cette implémentation, votre boite Linux pourra dialoguer VLAN avec des machines comme les Cisco Catalyst, 3Com: {Corebuilder, Netbuilder II, SuperStack II switch 630}, Extreme Ntwks Summit 48, Foundry: {ServerIronXL, FastIron}.

Implémentation alternative VLAN 802.1Q pour Linux(site)
: Une implémentation alternative de VLAN pour Linux. Ce projet a démarré suite au désaccord avec l'architecture et le style de codage du projet VLAN 'établi', avec comme résultat une structure de l'ensemble plus clair. Mise à jour : a été inclus dans le noyau 2.4.14 (peut−être dans le 2.4.13).

Un bon HOWTO à propos des VLAN peut être trouvé ici.

Mise à jour : a été inclus dans le noyau à partir de la version 2.4.14 (peut−être 13).

Serveur Linux Virtuel (Linux Virtual Server)(site)

Ces personnes sont très talentueuses. Le Serveur Virtuel Linux est un serveur à haute disponibilité, hautement évolutif, construit autour d'une grappe (cluster) de serveurs, avec un équilibreur de charge tournant sur le système d'exploitation Linux. L'architecture du cluster est transparente pour les utilisateurs finaux, qui ne voient qu'un simple serveur virtuel.

En résumé, que vous ayez besoin d'équilibrer votre charge ou de contrôler votre trafic, LVS aura une manière de le faire. Certaines de leurs techniques sont positivement diaboliques !. Par exemple, ils permettent à plusieurs machines d'avoir une même adresse IP, mais en désactivant l'ARP dessus. Seule la machine LVS qui a, elle, l'ARP actif, décide de l'hôte qui manipulera le paquet entrant. Celui−ci est envoyé avec la bonne adresse MAC au serveur choisi. Le trafic sortant passe directement par le routeur, et non par la machine LVS, qui, par conséquent n'a pas besoin de voir vos 5Gbit/s de données allant sur Internet. Cette machine LVS ne peut alors pas être un goulot d'étranglement.

L'implémentation de LVS nécessite une mise à jour pour les noyaux 2.0 et 2.2, alors qu'un module Netfilter est disponible dans le 2.4. Il n'y a donc pas besoin de mise à jour pour cette version du noyau. Le support 2.4 est encore en développement. Battez−vous donc avec et envoyez vos commentaires ou vos mises à jour.

CBQ.init (site)

Configurer CBQ peut être un peu intimidant, spécialement si votre seul souhait est de mettre en forme le trafic d'ordinateurs placés derrière un routeur. CBQ.init peut vous aider à configurer Linux à l'aide d'une syntaxe simplifiée.

Par exemple, si vous voulez que tous les ordinateurs de votre réseau 192.168.1.0/24 (sur eth1 10 Mbits) aient leur vitesse de téléchargement limitée à 28 Kbits, remplissez le fichier de configuration de CBQ.init avec ce qui suit :

```
DEVICE=eth1,10Mbit,1Mbit
RATE=28Kbit
WEIGHT=2Kbit
PRIO=5
RULE=192.168.1.0/24
```

Utiliser simplement ce programme si le 'comment et pourquoi' ne vous intéresse pas. Nous utilisons CBQ.init en production et il marche très bien. On peut même faire des choses plus avancées, comme la mise en forme dépendant du temps. La documentation est directement intégrée dans le script, ce qui explique l'absence d'un fichier README.

Scripts faciles de mise en forme Chronox(site)

Stephan Mueller (smueller@chronox.de) a écrit deux scripts utiles, "limit.conn" et "shaper". Le premier vous permet de maîtriser une session de téléchargement, comme ceci :

```
# limit.conn −s SERVERIP −p SERVERPORT −l LIMIT
```

Il fonctionne avec Linux 2.2 et 2.4.

Le second script est plus compliqué et peut être utilisé pour mettre en place des files d'attente différentes basées sur les règles iptables. Celles−ci sont utilisées pour marquer les paquets qui sont alors mis en forme.

Implémentation du Protocole Redondant Routeur Virtuel (site)

Ceci est purement pour la redondance. Deux machines avec leurs propres adresses IP et MAC créent une troisième adresse IP et MAC virtuelle. Bien que destiné à l'origine uniquement aux routeurs, qui ont besoin d'adresses MAC constantes, cela marche également pour les autres serveurs.

La beauté de cette approche est l'incroyable facilité de la configuration. Pas de compilation de noyau ou de nécessité de mise à jour, tout se passe dans l'espace utilisateur.

Lancer simplement ceci sur toutes les machines participant au service :

```
# vrrpd −i eth0 −v 50 10.0.0.22
```

Et vous voilà opérationnel ! 10.0.0.22 est maintenant géré par l'un de vos serveurs, probablement le premier à avoir lancé le démon vrrp. Déconnectez maintenant cet ordinateur du réseau et très rapidement, l'adresse 10.0.0.22 et l'adresse MAC seront gérées par l'un des autres ordinateurs.

J'ai essayé ceci et il a été actif et opérationnel en 1 minute. Pour une raison étrange, ma passerelle par défaut a été supprimée. Cependant, l'option −n permet de prévenir cela.

Voici une défaillance en "direct" :

```
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
```

Pas *un* paquet ping n'a été perdu ! Après 4 paquets, j'ai déconnecté mon P200 du réseau, et mon 486 a pris le relais, ce qui est visible par l'augmentation du temps de latence.

Chapitre 19. Lectures supplémentaires

http://snafu.freedom.org/linux2.2/iproute−notes.html

Contient beaucoup d'informations techniques, et de commentaires sur le noyau.

http://www.davin.ottawa.on.ca/ols/

Transparents de Jamal Hadi Salim, un des auteurs du contrôleur de trafic de Linux.

http://defiant.coinet.com/iproute2/ip−cref/

Version HTML de la documentation LaTeX d'Alexeys ; explique une partie d'iproute2 en détails.

http://www.aciri.org/floyd/cbq.html

Sally Floyd a une bonne page sur CBQ, incluant ses publications originales. Aucune n'est spécifique à Linux, mais il y a un travail de discussion sur la théorie et l'utilisation de CBQ. Contenu très technique, mais une bonne lecture pour ceux qui sont intéressés.

Differentiated Services on Linux

This document par Werner Almesberger, Jamal Hadi Salim et Alexey Kuznetsov. Décrit les fonctions DiffServ du noyau Linux, entre autres les gestionnaires de mise en file d'attente TBF, GRED, *DSMARK* et le classificateur tcindex.

http://ceti.pl/~ekravietz/cbq/NET4_tc.html
: Un autre HOWTO, en polonais ! Vous pouvez cependant copier/coller les lignes de commandes, elles fonctionnent de la même façon dans toutes les langues. L'auteur travaille en collaboration avec nous et sera peut être bientôt un auteur de sections de cet HOWTO.

IOS Committed Access Rate
: Des gens de Cisco qui ont pris la louable habitude de mettre leur documentation en ligne. La syntaxe de Cisco est différente mais les concepts sont identiques, sauf qu'on fait mieux, et sans matériel qui coûte autant qu'une voiture :−)

TCP/IP Illustrated, volume 1, W. Richard Stevens, ISBN 0−201−63346−9
: Sa lecture est indispensable si vous voulez réellement comprendre TCP/IP, et de plus elle est divertissante.

Chapitre 20. Remerciements

Notre but est de faire la liste de toutes les personnes qui ont contribué à ce HOWTO, ou qui nous ont aidés à expliquer le fonctionnement des choses. Alors qu'il n'existe pas actuellement de tableau d'honneur Netfilter, nous souhaitons saluer les personnes qui apportent leur aide.

- Juanjo 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@prosys.com.pl>`
- 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>
- 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>
- 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>
- 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>