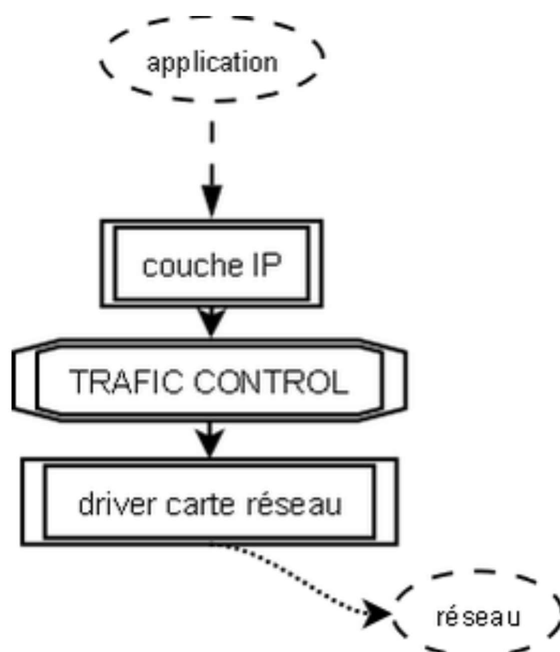
1. Traffic Control, les bases

Traffic Control travaille sur les paquets sortant du noyau. Il n'a pas, initialement, pour objectif de contrôler le trafic des paquets entrants. Cette portion de code du noyau se situe entre la couche IP et le pilote du matériel qui transmet sur le réseau. On est donc très bas dans les couches. En réalité, c'est Traffic Control qui est constamment en charge de transmettre au driver de la carte réseau le paquet à envoyer.

Cela signifie en fait que le module TC - le scheduler de paquet - est en permanence activé dans le noyau, même quand vous ne pensez pas l'utiliser.

Par défaut, ce scheduler maintient une *queue* (prononcer kiou, une file d'attente) similaire à FIFO, dans laquelle le premier paquet entré est donc le premier sorti.



La base de TC est la *Queuing Discipline* (*qdisc*) qui représente la politique de *scheduling* appliquée à une queue. Il existe différentes *qdisc*. Comme pour le scheduling processeur, on retrouve les méthodes FIFO, FIFO à plusieurs files, FIFO avec hash et round robin (SFQ). On a également un système *Token Bucket Filter* (TBF) qui attribue des jetons (*tokens*) à une qdisc pour en limiter le débit (pas de token = pas de transmission = on

attend d'avoir un jeton disponible). Cette dernière politique a ensuite été étendue à un TBF hiérarchique, le HTB (*Hierarchical Token Bucket*). Les politiques que nous allons étudier ici sont TBF, qui pose les fondamentaux, SFQ et HTB. Nous allons également jeter un coup d'œil à la politique par défaut que, tout Monsieur Jourdain que nous sommes, nous utilisons sans le savoir : pfifo_fast.

1.1 Premier contact

Jean-Kevin est pressé, il n'a pas de temps à perdre, et tout de suite maintenant, il doit limiter la bande passante sortante de son serveur web à 200 kbits par secondes (25 ko/s). Au diable la théorie, on y reviendra plus tard, mettons tout de suite les mains dans le cambouis.

La mécanique que nous allons mettre en place est simple. Nous allons utiliser une règle Netfilter pour marquer les paquets qui nous intéressent. Ensuite, nous allons fournir à TC une politique qui s'appliquera sur les paquets contenant la marque définie. C'est parti.

1.2 Netfilter MARK

Netfilter permet d’interagir directement avec la structure représentant un paquet dans le noyau. Cette structure, le **sk_buff**, possède un champ **__u32 nfmark** que l’on va renseigner et qui sera lu par le filtre de TC pour sélectionner la classe de destination du paquet.

La règle iptables suivante va appliquer la marque '80' sur les paquets sortant (chaîne OUTPUT) ayant pour port source le port 80 :

iptables -t mangle -A OUTPUT -o eth0 -p tcp --sport 80 -j MARK --set-mark 80

On peut vérifier que cette règle est bien appliquée aux paquets sortants en visualisant les statistiques de Netfilter.

iptables -L OUTPUT -t mangle -v

Chain OUTPUT (policy ACCEPT 74107 packets, 109M bytes)

pkts bytes target prot opt in out source   destination

73896 109M MARK   tcp -- any eth0 anywhere anywhere   tcp spt:www MARK xset 0x50/0xffffffff

1.3 Deux classes dans un arbre

Le binaire **/sbin/tc** est compris dans le package **iproute**. Un simple **aptitude** suffit à l'installer, s'il ne l'est pas déjà.

Nous allons créer un arbre dont la racine appliquera la politique HTB. Cet arbre va contenir deux classes : une pour notre trafic marqué, l'autre pour tout le reste et qui sera donc considérée par défaut.

tc qdisc add dev eth0 root handle 1: htb default 20

tc class add dev eth0 parent 1:0 classid 1:10 htb rate 200kbit ceil 200kbit prio 1 mtu 1500

tc class add dev eth0 parent 1:0 classid 1:20 htb rate 1024kbit ceil 1024kbit prio 2 mtu 1500

Les deux classes filles sont raccrochées à la racine. Ces classes possèdent un débit garantie (*rate*) et un débit maximal opportuniste (*ceil*). Si la bande passante n'est pas utilisée, alors une classe pourra monter son débit jusqu'à la valeur de ceil. Sinon, c'est la valeur de rate qui s'applique.

Cela veut dire que la somme des valeurs de rate doit correspondre à la bande passante disponible. Dans le cas d'un upload ADSL classique chez un fournisseur correct, cela sera d'environ 1024 kbits (dans le meilleur des cas, éloignement du DSLAM, etc.).

Nous avons maintenant d'un côté un arbre de contrôle de trafic et d'un autre côté du marquage de paquets. Il reste donc à relier les deux. Cela est fait avec les règles de filtrage de TC. Ces règles sont très simples. On dit à TC de regarder (*handle*) les paquets portant la marque 80 et de les envoyer

(*fw flowid*) à la classe correspondante. Un point important toutefois, un filtre doit être rattaché à la racine « *root* » de l'arbre. Sinon, il n'est pas pris en compte.

tc filter add dev eth0 parent 1:0 protocol ip prio 1 handle 80 fw flowid 1:10

Faisons maintenant le test avec netcat, on ouvre un port en écoute qui renvoie des zéros. C'est basique et parfait pour tester notre politique. On lance donc :

nc -l -p 80 < /dev/zero

Et sur une autre machine, on lance un telnet vers le port 80 de la machine en écoute. L'outil **iptraf** permet de visualiser la connexion en cours, et surtout, son débit (voir figure 2).

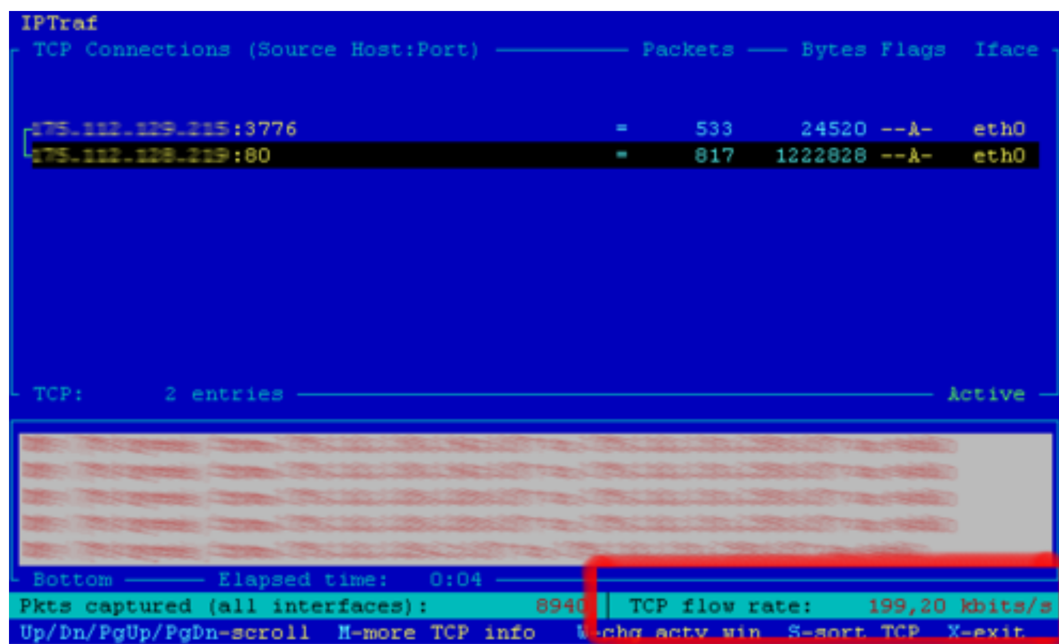


Figure 2 : débit QOS simple

Comme on le voit dans l'encadré rouge, en bas à droite, le débit de la connexion est de 199,20 kbps. On s'approche beaucoup des 200 kbps, la précision dépendant de quelques paramètres que nous allons étudier.

Si l'on teste une connexion du même type sur un autre port, on verra un débit limité à 1024 kbps, ce qui correspond au débit de la classe par défaut

qui s'applique à tous les paquets non marqués.

2. De la pratique à la théorie

Après ce premier contact, il est temps de revenir aux fondamentaux de la QoS sous Linux. L'objectif de ce chapitre est de préparer le terrain à la mise en place d'un jeu de règles plus évolué afin d'avoir un contrôle plus fin sur les débits des connexions. Mais pour cela, il faut bien comprendre les mécanismes internes du scheduling de paquets.

2.1 Vingt mille lieues sous le code

Le code de TC est contenu dans le répertoire **net/sched** des sources du noyau. Le noyau dissocie les flux entrants – ingress – des flux sortants - egress -. Or, comme on l'a dit plus tôt, c'est le module TC du noyau qui a la responsabilité de traiter le chemin egress.

La figure 3 représente le chemin suivi par un paquet dans le noyau, quand il y rentre – ingress - et quand il en part – egress -. Si l'on se concentre sur un paquet sortant, on voit qu'il arrive par la couche 4 (TCP/UDP/...), puis entre dans la couche IP, qui n'est pas représentée ici. En fait, les chaînes Netfilter OUTPUT et POSTROUTING font parties de la couche IP et sont intercalées entre les fonctions de manipulation IP (création des *headers*, fragmentation, etc.). À la sortie de la table mangle de la chaîne POSTROUTING, le paquet est transmis à la file d'attente egress, et c'est là que TC commence son travail.

Presque tous les équipements utilisent une queue pour scheduler le trafic egress. Le noyau dispose d'algorithmes permettant de manipuler ces queues, ce sont les fameuses *queuing disciplines* (FIFO, SFQ,HTB, ...). Le travail de TC va être d'appliquer ces queuing disciplines à la file egress pour sélectionner un paquet à envoyer à l'équipement.

TC va travailler avec la structure **sk_buff** qui représente un paquet dans le noyau, et non pas sur le paquet directement. **sk_buff**est une structure partagée et accessible partout dans le noyau, nul besoin de copie en mémoire. Cette méthode est plus flexible et beaucoup plus rapide car

sk_buffcontient toutes les informations nécessaires à la manipulation du paquet, et l'on évite ainsi de copier les headers et le payload d'une zone à une autre de la mémoire, ce qui ruinerait les performances.

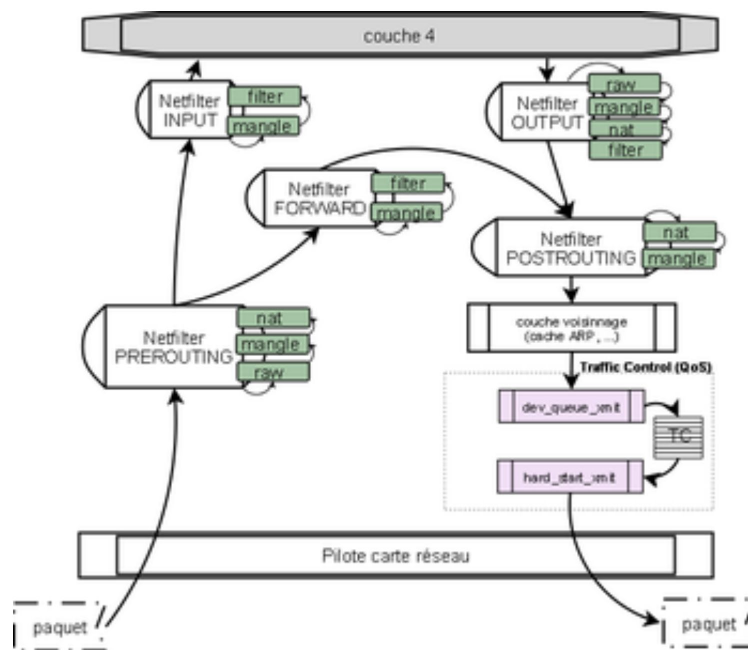


Figure 3 : Netfilter et TC dans la pile réseau du noyau

Notre paquet est donc mis à disposition de TC. Le scheduler de TC va se réveiller régulièrement pour sélectionner un paquet à transmettre.

La majeure partie du travail est lancée par la fonction**dev_queue_xmit** (définie dans **net/core/dev.c**), qui ne reçoit qu'une structure **sk_buff** en entrée. Cette structure va contenir toutes les informations nécessaires. **skb->dev** contient, par exemple, un pointeur vers l'équipement réseau de sortie.

dev_queue_xmit s'assure que le paquet est prêt à être transmis sur le réseau, que la fragmentation est compatible avec les capacités de la carte réseau et que les *checksums* ont été calculés (notons au passage qu'un périphérique peut calculer les checksums lui-même, auquel cas le contrôle est contourné). Une fois ces contrôles effectués, et si l'équipement dispose d'une queuedans**skb->dev->qdisc**, la structure **sk_buff** est ajoutée dans cette queue (via la fonction **enqueue**) et la fonction **qdisc_run** est appelée pour sélectionner un paquet à transmettre.

Cela signifie que le paquet qui vient d'être ajouté ne sera peut-être pas

celui immédiatement transmis, mais on est toutefois certain qu'il est prêt pour transmission ultérieure au moment où il est ajouté dans la queue.

À chaque périphérique est attachée une queue discipline (qdisc) racine. C'est ce que nous avons fait précédemment lors de la création de la qdisc racine via la commande :

tc qdisc add dev eth0 root handle 1: htb default 20

qui signifie en fait : « attache une qdisc au device eth0, qui possédera l'identifiant 1, utilisera l'algorithme HTB et enverra par défaut les paquets vers la classe 20 ».

On retrouve donc cette qdisc dans notre structure **sk_buff** au pointeur **skb->dev->qdisc**. On y trouvera également les fonctions de mise en file d'attente et de retrait de la file d'attente après scheduling via les pointeurs **skb->dev->qdisc->enqueue()**et **skb->dev->qdisc->dequeue()**. C'est cette dernière fonction **dequeue()** qui aura la responsabilité de transmettre la structure **sk_buff** du paquet au périphérique pour transmission.

La qdisc racine peut avoir des feuilles, sous forme de classes, qui peuvent ensuite appeler d'autres qdisc afin de construire une arborescence.

2.2 Un peu de discipline

Tout ce chemin pour en arriver là ! J'espère que vous êtes toujours vivant. Pour ceux qui veulent approfondir le vaste/immense sujet, la lecture de *Understanding Linux Network Internals*, de Christian Benvenuti, chez O'reilly, s'avérera des plus enrichissante.

Nous avons donc une fonction **dequeue()**dont le rôle va être de sortir le prochain paquet à transmettre de la qdisc d'un périphérique. Pour cela, cette fonction se base sur des algorithmes d'ordonnancement que nous allons donc maintenant étudier, puis mettre en œuvre.

Il existe deux types d'algorithmes : Classless et Classful. Les algorithmes

Classful contiennent des qdisc pouvant elles-mêmes contenir des classes, comme nous l'avons fait dans notre premier exemple avec HTB. Les algorithmes Classless ne permettent pas cela et sont donc bien plus simples (quoique).

2.2.1 pfifo_fast

Commençons petit. **pfifo_fast** est l'algorithme d'ordonnancement par défaut lorsqu'aucun autre n'est défini. Autrement dit, c'est celui qu'utilisent 99,9% des Linuxiens. Il est classless et un poil plus évolué qu'un FIFO basique, puisqu'il implémente trois « bandes » en parallèles. Ces bandes sont numérotées 0, 1 et 2. Tant que 0 ne sera pas vide, 1 ne sera pas traitée, puis ce sera le tour de 2.

Le noyau se base sur le champ *Type Of Service*du *header* IP du paquet pour sélectionner la bande dans laquelle insérer le paquet (enfin...sa structure **sk_buff**).

Cet algorithme est défini dans **net/sched/sch_generic.c** et représenté en figure 4.

La longueur des bandes, soit le nombre de paquets qu'elles peuvent contenir, est définie en dehors de TC. Cela fait partie des paramètres du périphérique réseau que l'on peut paramétrer via **ifconfig**et le paramètre**txqueuelen**. On peut également visualiser la valeur courante dans **/sys** :

```
# cat /sys/class/net/eth0/tx_queue_len
```

Cela signifie que, par défaut, 1000 paquets pourront être mis en file d'attente dans chacune des bandes. Passée cette limite, le noyau commence à détruire des paquets. Heureusement, cela n'arrive jamais car la pile TCP s'assure d'adapter sa vitesse d'émission (mécanisme de *TCP slow start*), mais sur des débits importants (en gigabits, par exemple), il peut être intéressant de monter cette limite à 10 000, voire 100 000, paquets.

Enfin, comme cette discipline est classless, il est impossible d'ajouter une

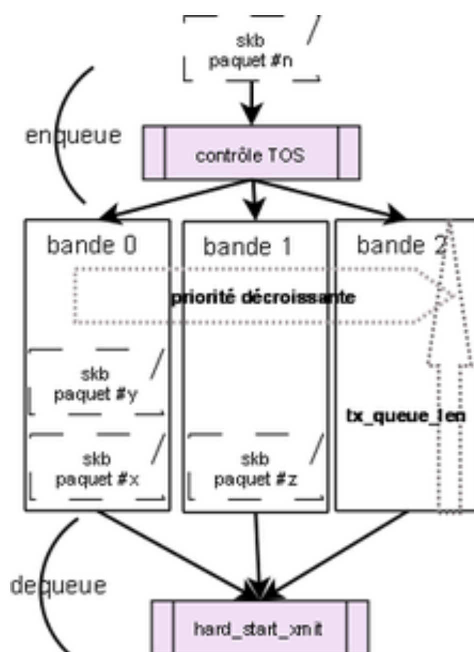
autre discipline après celle-ci.

2.2.2 Stochastic Fairness Queuing

Sous le doux nom de « mise en file d'attente équitable et aléatoire » se cache en fait une technique permettant d'offrir un partage raisonnablement équitable de la bande passante entre plusieurs connexions.

Le principe de SFQ est de prendre une empreinte (*hash*) du paquet en se basant sur ses headers et de classer ensuite ce paquet dans un seau (*bucket*) choisi entre 1 et 1024. Les seaux sont ensuite vidés les uns après les autres suivant le principe du *Round Robin*.

L'intérêt est donc qu'aucun paquet n'aura la priorité sur un autre et qu'une connexion ne pourra pas s'approprier toute la bande passante. La répartition sera presque toujours équitable, mais bien sûr, il y a des limitations. La plus importante est que le hash peut être le même pour plusieurs connexions et que ces dernières seront donc mises dans le même bucket. C'est pour cela que SFQ modifie l'algorithme de hash à fréquence régulière (par défaut, toutes les 10 secondes).



Le code de SFQ est dans **net/sched/sch_sfq.c**. Il définit en dur les valeurs du nombre de buckets (**SFQ_HASH_DIVISOR** à 1024) et de la profondeur des buckets (**SFQ_DEPTH** à 128). Cette dernière valeur

détermine le nombre de paquets pouvant être mis dans chaque bucket, on pourra donc mettre au total 1024*128 = 131072 paquets en file d'attente dans SFQ si tous les buckets sont pleins (cas idéal à faible probabilité, puisqu'il faudrait que toutes les connexions aient des hashs différents).

L'outil **TC** nous permet de voir les options paramétrables de SFQ :

Usage: ... sfq [limit NUMBER] [perturb SECS] [quantum BYTES]

- **limit** : la taille d'un bucket, peut être réduite en dessous de **SFQ_DEPTH** mais pas augmentée. Si vous essayez de mettre une valeur supérieure, elle sera simplement ignorée;

- **perturb**: la fréquence, en seconde, de mise à jour de la fonction de hash ;

- **quantum**: représente le nombre d'octets que le Round Robin pourra dépiler avant de passer au bucket suivant. **quantum** doit être, au minimum, égal au MTU (*Maximum Transmission Unit*, la quantité maximale d'octets que peut contenir une trame) du périphérique (généralement 1500 octets sur Ethernet). Dans le cas contraire, tous les paquets de taille supérieure à **quantum** ne pourront être envoyés, car leur taille sera trop élevée et ils resteront donc bloqués dans le bucket, bouchant ce dernier au passage.

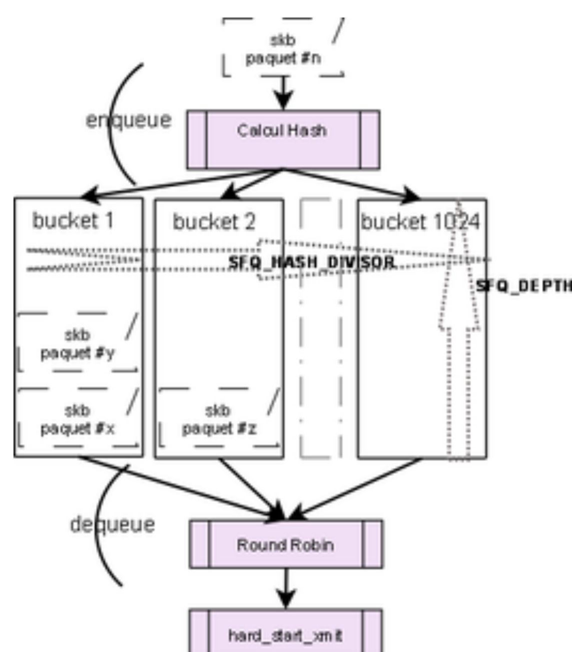


Figure 5 : Stochastic Fairness Queuing

2.2.2.1 Le hashing dans SFQ

Le hash est calculé par la fonction **sfq_hash** sur les headers IP destination, IP Source, le protocole de couche 4 et les ports de couche 4 (si le paquet n'est pas un fragment IP). Le tout est mixé avec un nombre aléatoire regénéré toutes les « perturb »secondes.

Si l'on fait le test avec une connexion partant de 126.255.254.240:8080 vers 175.112.129.215:2146 en TCP, l'algorithme donne :

/*IP Destination en hexa*/

/* 06 est le numéro du protocole TCP dans le header IP*/

/* le paquet ip n'est pas un fragment, donc on prend en compte les ports TCP*/

/* 1f900862 = Port Source et Port Dest*/

/* algorithme de Jenkins avec quelques golden numbers supplémentaires */

h = jhash(h1, h2, perturbation)

Cela fournit un hash « h » final sur 32 bits, qui sera utilisé par SFQ pour sélectionner le bucket de destination. Comme la valeur de perturbation est renouvelée à fréquence régulière, les paquets d'une connexion peuvent être redirigés vers des buckets différents si la connexion dure assez longtemps.

Cela peut également signifier qu'un cas de déséquençage existe si deux paquets d'une même connexion sont mis dans deux buckets différents et que le Round Robin vide le bucket 2 avant le bucket 1. Le protocole TCP va corriger cela grâce aux numéros de séquence, mais pour UDP, par exemple, cela ne sera pas le cas et le paquet 2 sera traité avant le paquet 1 par le destinataire. Le cas est tout de même limité, mais cela peut être gênant sur un flux de *remote syslog*, par exemple.

Pour améliorer la flexibilité du système de hash, il est possible de modifier,

via les filtres de TC, les champs sur lesquels le hash est calculé. On y reviendra plus tard pour déterminer quels champs sont pertinents, mais cela ressemble à :

tc qdisc add dev eth0 root handle 1: sfq perturb 10 quantum 3000 limit 64

tc filter add dev eth0 parent 1:0 protocol ip handle 1 \

flow hash keys src,dst divisor 1024

Le filtre ci-dessus modifie simplement le hash pour ne prendre en compte que les adresses source et destination. Le *divisor* représente le nombre de buckets. On pourrait donc créer un hash qui ne travaille que sur 10 buckets (divisor=10) en prenant en compte l'IP de destination.

Et pour en terminer avec SFQ, comme cette discipline est classless, il est impossible d'ajouter une autre discipline après celle-ci.

2.2.3 Token Bucket Filter

Jusque-là, nous avons étudié des algorithmes qui ne permettent pas de contrôler la bande passante. SFQ et **PFIFO_FAST** permettent essentiellement de fluidifier le trafic, voire de le catégoriser un peu, mais pas de contrôler son débit.

En fait, tout le problème du contrôle du débit consiste à compter les paquets pour savoir à quel moment les envoyer. Comme cette méthode est particulièrement difficile à implémenter (maintien d'état, utilisation de mémoire, etc.), les développeurs ont traité le problème différemment.

L'idée est la suivante : plutôt que de compter les paquets (ou les bits des paquets, c'est pareil), l'algorithme *Token Bucket Filter* envoie à fréquence régulière un jeton (*token*) dans un seau (*bucket*). Quand un paquet arrive et qu'il souhaite sortir, il est autorisé à passer si suffisamment de jetons sont disponibles. Sinon, il attend.

En fait, ce ne sont pas les paquets mais leur taille - les bits - qui est prise en

compte. Prenons un exemple : admettons qu'un paquet de 8000 bits (1000 octets) souhaite sortir, il arrive dans l'algorithme TBF qui vérifie le contenu du bucket. Si 8000 tokens sont disponibles, TBF les consument et transmet le paquet au périphérique. Sinon, il attend que les 8000 tokens soient disponibles.

La fréquence d'arrivage des tokens, et donc de remplissage du bucket, est fixe suivant le débit (*rate*) que l'on souhaite donner a la qdisc. C'est sur ce paramètre que l'essentiel du contrôle du débit s'effectue.

Une autre particularité de TBF est de permettre des *bursts*. En effet, si aucun paquet n'arrive, le bucket continuera tout de même de se remplir. Une fois qu'il est plein, il arrête de se remplir.

Si une rafale de paquets arrivent alors que le bucket est plein, ils pourront vider le bucket d'un coup et passer immédiatement : c'est un burst. Ainsi, plus le bucket est grand, plus les bursts seront importants et la valeur de burst détermine la taille du bucket.

Cela peut poser des problèmes dans le cas où on utilise des buckets très grands. C'est pour cela que l'implémentation prévoit un moyen de limiter l'impact de ces bursts, au moyen d'un second bucket. Ce second bucket est plus petit, par défaut, il est de taille égale au MTU (1500 octets en Ethernet), soit un seul paquet en fait. Ce second bucket se remplit à la vitesse *peakrate*, qui va donc définir la vitesse maximale des bursts.

On a donc les paramètres suivants :

- peakrate > rate, le second bucket doit se remplir plus vite que le premier (pour autoriser les bursts). Si peakrate est infini, alors TBF se comporte comme si le second bucket n'existait pas et videra tout le bucket principal d'un coup si nécessaire ;

- burst > MTU, la taille du bucket principal doit être bien plus grande que celle du bucket de limitation des bursts, sinon ce dernier ne sert à rien et les bursts sont impossibles.

Donc, en clair et pour résumer, quand tout fonctionne normalement, les paquets sont mis en queue et passent à la vitesse de rate. Si des tokens sont disponibles et que des paquets arrivent, ils peuvent être envoyés plus rapidement à la vitesse de peakratejusqu'à ce que le bucket principal soit vide. Tout cela est représenté en figure 6. Le code source est lisible dans le fichier **net/sched/sch_tbf.c**, la fonction vraiment intéressante étant **tbf_dequeue**.

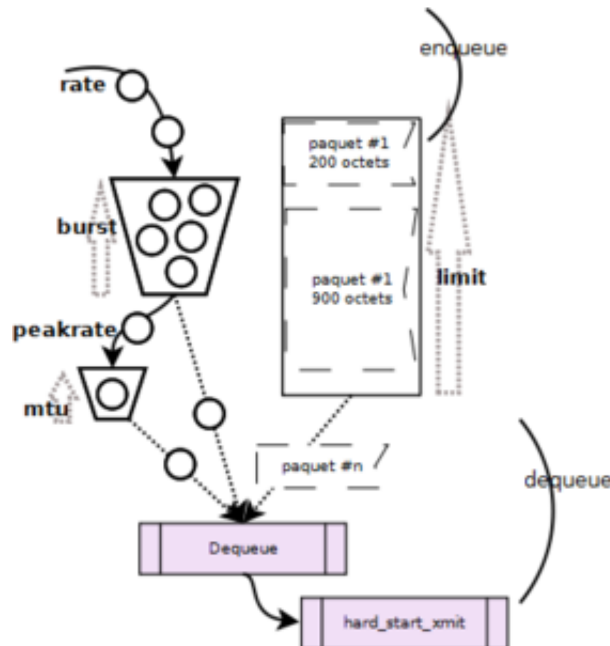


Figure 6 : Token Bucket Filter

Les options paramétrables de TBF sont disponibles via **TC** :

Usage: ... tbf limit BYTES burst BYTES[/BYTES] rate KBPS [mtu BYTES[/BYTES]]

[peakrate KBPS] [latency TIME] [overhead BYTES] [linklayer TYPE]

On retrouve les paramètres **burst**, **rate**, **mtu** et **peakrate**. **limit** est la taille de la queue de paquets. **latency** est une autre manière de calculer la valeur de limit, en précisant une latence (le temps d'envoi) qui en déduit le nombre de paquets pouvant attendre en file (calcul qui prend en compte les valeurs de burst, rate, peakrateet MTU).Concernant **overhead** et **linklayer**, c'est encore une autre histoire... que nous allons étudier maintenant.

2.2.3.1 ADSL et ATM, le boeuf qui se croyait grenouille

Si les réseaux locaux sont, pour 99,999% d'entre eux, basés sur Ethernet, les liaisons ADSL, une fois sorties de la box, sont pour la plupart réalisées sur ATM. Or la particularité d'ATM est de découper les gros paquets en tout petits, plus nombreux et fortement multiplexés. Ainsi, une trame de 1500 octets va se voir découpée en une trentaine de paquets de... 53 octets. Et dans ces 53 octets, seuls 48 proviennent du paquet d'origine, le reste étant des headers spécifiques à ATM.

Alors quel est le problème ? Eh bien, en réalisant une QoS avec limitation du débit sur des liens ethernet, on calcule une bande passante qui ne prend pas en compte cet overhead due aux headers du protocole ATM. Une trame ethernet faisant initialement 1500 octets sera envoyée en 32 trames ATM pour une taille totale de (32*5)+1500 = 1660 octets, soit une augmentation d'un peu plus de 10%. En clair, on dispose finalement de 10% de bande passante de moins qu'on ne le croit (c'est une moyenne qui dépend du type de paquet, de sa taille, etc.).

Jesper Dangaard Brouer a réalisé son Master sur ce sujet et a produit des patchs pour le noyau et l'outil TC. Ces patchs implémentent les paramètres **overhead** et **linklayer**, qui permettent d'inclure le critère de liaison ATM dans le calcul de la QoS.

- **overhead** représente la quantité d'octets ajoutée par les headers ATM (ici, donc, 5 octets) ;

- **linklayer** définit le type, compris entre {**ethernet**, **atm**, **adsl**} (**atm** et **adsl** signifient la même chose, un overhead de 5 octets).

Au final, on peut créer une qdisc de type TBF sur eth0 via la commande suivante :

tc qdisc add dev eth0 root tbf rate 1mbit burst 10k latency 30ms linklayer atm

tc -s qdisc show dev eth0

qdisc tbf 8005: root rate 1000Kbit burst 10Kb lat 30.0ms

Sent 738 bytes 5 pkt (dropped 0, overlimits 0 requeues 0)

rate 0bit 0pps backlog 0b 0p requeues 0

Et en positionnant le paramètre linklayer à **atm**, on dit à TBF de prendre en compte un overhead de 5 octets et donc de calculer le débit en prévoyant le multiplexage ATM qui va intervenir plus tard, en dehors de Linux. C'est une manière élégante d'éviter de surcharger la file d'attente interne de sa box ADSL. Il faut toutefois faire attention à la manipulation de ce paramètre et ne pas le positionner trop haut dans l'arbre, comme nous allons le voir dans l'implémentation.

2.2.3.2 Les limites de TBF

TBF permet un contrôle très précis du débit d'une qdisc. Toutefois, il impose que tous les paquets de cette qdisc soient dans une seule file d'attente et risque de bloquer l'émission de paquets plus petits si un gros paquet est devant (cas de la file d'attente en figure 6).

On pourrait optimiser la bande passante en laissant passer le paquet de 200 octets avant celui de 900 octets, mais cela reviendrait à réordonnancer l'émission des paquets et on retomberait dans les travers de SFQ.

L'autre solution serait de faire un filtrage plus fin, avec plusieurs qdisc, et de répartir la bande passante entre ces qdisc via des filtres. Pourquoi ne pas les faire également communiquer entre elles pour qu'elles puissent s'emprunter de la bande passante ?

Ca y est, nous venons d'inventer le principe de *classful qdisc* !

2.2.4 Hierarchical Token Bucket

Le Seau à Jetons Hiérarchique, HTB, est une implémentation améliorée de TBF pour introduire la notion de classe. Chaque classe est en fait une qdisc TBF améliorée. Les classes sont reliées entre elles sont la forme d'un arbre, avec une racine et des feuilles.

HTB introduit des notions particulièrement utiles pour la gestion de la bande passante, comme la notion de priorité, l'emprunt de bande passante ou la possibilité de brancher une qdisc de niveau inférieur qui soit différente de TBF, comme un **pfifo_fast** ou un SFQ. Cela va nous être utile dans l'implémentation de nos règles.

Chaque classe HTB va avoir en permanence l'un des trois statuts suivant :

1. **HTB_CANT_SEND** : la classe ne peut ni émettre, ni emprunter ;

2. **HTB_MAY_BORROW** : ne peut pas émettre, mais peut essayer d'emprunter ;

3. **HTB_CAN_SEND** : la classe peut émettre.

Prenons un exemple simple, représenté en figure 7, d'un arbre composé d'une classe root (*handle* 1) et de deux classes filles (*handles* 10 et 20).

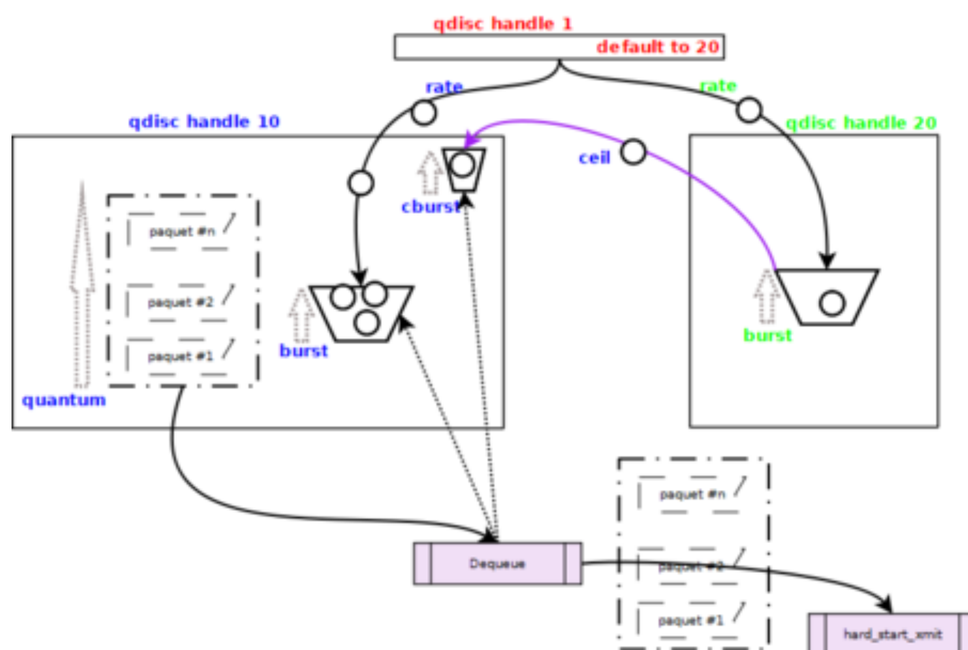
10 a un débit (*rate)* à 200 kbit, un emprunt (*ceil)* à 400 kbit et une priorité (*prio*) à 1. 20 a un rate a 200 kbit, un ceil a 200 kbit, ce qui signifie que cette classe ne peut pas emprunter de bande passante, et une priorité à 2.

Un groupe de paquets arrive dans TC, il est marqué du flag 10 et va donc être traité dans la qdisc 10. Le bucket de la classe 10 ne contient pas suffisamment de tokens pour envoyer le premier paquet, la classe va donc essayer d'emprunter des tokens à la classe 20. Toutefois, elle va tenter d'emprunter plus de tokens que nécessaire pour envoyer le premier paquet. Cela est paramétré par la variable **quantum**.

Un quantum est la quantité maximale de bits que la classe va tenter d'envoyer d'un seul coup avant de rendre la main. C'est le même quantum que celui que nous avons vu avec SFQ. Plus la valeur de quantum est proche du MTU, plus le *scheduling* sera précis. À l'inverse, plus la valeur de quantum est grande et plus une classe sera privilégiée : elle pourra envoyer beaucoup de bits d'un seul coup et emprunter de grandes quantités à ses voisines, quitte à les pénaliser.

Pourquoi les pénaliser ? Car lorsque l'emprunt a lieu, **TC** ne sait pas si des paquets sont en attente dans les autres classes. Il va donc vider les buckets de ces classes, qui ne pourront potentiellement plus envoyer leurs propres paquets ensuite.

En pratique, on ne manipule pas directement la valeur de quantum. Une valeur r2q (quantum = rate / r2q) est disponible pour cela. Par défaut, r2q vaut 10, donc pour rate valant 200 kbit, quantum vaudra 20kbit. Pour les débits faibles ou importants, il faudra se soucier de la valeur de r2q afin de s'assurer que quantum n'est ni trop grand (trop de paquets partent d'un coup) ni trop petit (impossible d'envoyer les paquets dont la taille est supérieure à quantum). En revanche, **r2q** doit être positionné au niveau de la qdisc racine, il faut donc prévoir son calcul pour toutes les qdisc impactées.



Hierarchical Token Bucket

Evidemment, la notion hiérarchique de HTB implique que les débits additionnés des classes filles (aussi appelées feuilles, *leaves*) ne dépassent pas le débit de la classe parente.

2.2.4.1 Hystérésis

L'hystérésis est un effet de bord introduit par une optimisation de HTB. Pour décharger le CPU, HTB ne recalculait pas, dans une ancienne version,

le contenu des buckets à fréquence suffisante, ce qui pouvait induire une surutilisation de la bande passante par certaines classes.

Ce problème à été corrigé depuis et un paramètre existe afin d'utiliser ou non le mécanisme d'optimisation (qui peut s'avérer nécessaire sur d'importants débits). Comme ce n'est pas notre cas ici, il faut vérifier que la valeur d'hystérésis est positionnée à zéro :

```
# cat /sys/module/sch_htb/parameters/htb_hysteresis
```

C'est tout pour HTB, on va revenir sur les paramètres internes plus tard, dans l'implémentation.

2.2.5 L'influence de la fréquence des timers

Une petite précision importante, qui concerne TC en général. Le scheduler de TC est prévu pour se réveiller sur appel du noyau, sa fréquence de réveil est donc contrainte par la fréquence de réveil du scheduler de processus du noyau. Cela se paramètre en dehors de TC dans la variable **HZ**. Sur les systèmes anciens, ou parfois par choix sur les serveurs, cette valeur est positionnée à 100, ce qui signifie que les processus sont reschedulés toutes les 1/100 = 10ms.

Pour TC, cela a un impact direct sur la latence induite aux paquets, car si un paquet arrive en file au même moment qu'un *tick* du noyau est émis, il devra attendre 10 ms, le prochain tick, pour être traité et envoyé au périphérique. Ce problème est connu et c'est entre autres pour cette raison qu'ont été intégrés des *timers* haute résolution dans le noyau (**hrtimers**, pour les curieux).

Une autre solution est de positionner la valeur de HZ à 1000 afin de réduire cette latence à 1ms, mais d'aucuns vous diront que cela peut être une mauvaise idée sur un serveur.

2.2.6 Respirez...

Ouf ! Bon, la tartine est un peu sèche et pas facile à avaler, je vous

l'accorde. D'ailleurs, je ne m'attends pas à ce que vous la mangiez d'un seul coup, en ayant tout compris. La mise en application des notions vues précédemment va aider à la compréhension des algorithmes, mais pour bien comprendre le sujet, il faut également passer du temps dans les sources du noyau, dans la documentation LARTC, linux-ip.net ou encore le mémoire de JD Brouer. Et puis surtout dans la console. Et si vous en voulez encore, il en reste beaucoup à couvrir. Je n'ai parlé ni de GRED ni de HFSC...

3. Une configuration complète

Nous allons donc implémenter une politique de QoS afin d'optimiser la bande passante montante d'une connexion ADSL. Les principes étudiés ici s'appliquent tout aussi bien à un réseau domestique qu'à un réseau dit « professionnel », pour monter un boîtier de QoS dans une infrastructure d'entreprise. Il faudra juste accepter qu'au final, notre *traffic shaper* ne fera pas de lumière et n'aura pas un joli logo, ce qui peut gêner même les décideurs les plus aguerris...

L'arbre de QoS est présenté en figure 8. Le trafic est divisé en 6 classes :

1. Interactivité : la classe ayant la priorité la plus basse (donc la plus forte) est réservée aux flux nécessitant une grande interactivité (faible latence), comme la VoIP ou les résolutions DNS.

2. TCP ACKs : les accusés de réception TCP sont intéressants à prioriser si l'on souhaite garantir une bonne réactivité des flux descendants. Si l'on part du principe qu'une connexion ADSL n'est pas utilisée que pour faire de l'hébergement, mais également pour naviguer sur le Net, cela peut être utile.

3. SSH : un faible débit, mais garanti pour pouvoir intervenir à distance même en cas de forte charge.

4. HTTP/HTTPS : les flux web bénéficient d'une importante bande passante garantie et sont ensuite redécoupés en sous-classes pour, par exemple, réaliser une répartition plus fine par site (par *virtual host*).

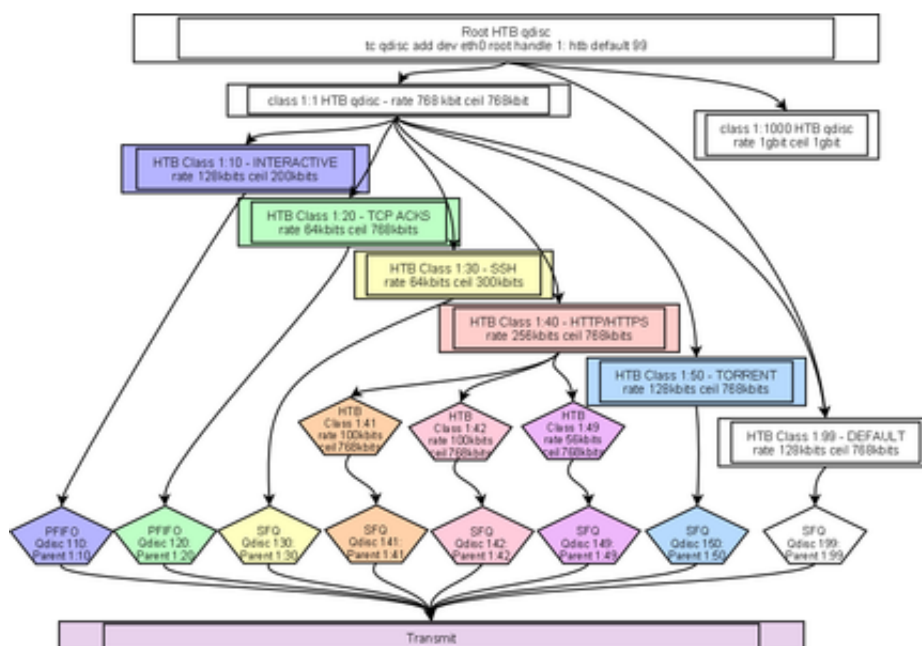
5. TORRENT : le protocole BitTorrent est intéressant car difficile à classifier. On va donc devoir utiliser des fonctionnalités avancées de Netfilter pour travailler dessus.

6. DEFAULT : la classe dans laquelle tombent tous les flux qui ne rentrent ailleurs.

De plus, en sortie de chaque classe, un qdisc classless est ajouté. Il s'agit de PFIFO pour les flux interactifs, pour éviter le réordonnancement des paquets UDP, et pour les TCP ACKs, et de SFQ pour tous les autres, afin de garantir une répartition au mieux de la bande passante entre les connexions.

Voyons maintenant comment mettre en place cet arbre sous Linux. J'utilise, pour les tests, une Debian Squeeze en 2.6.26 avec iproute basé sur la version du 25 juillet 2008 (2.6.26 également, ce package doit correspondre au noyau). La configuration devrait toutefois fonctionner sur toutes les distributions modernes.

Subtilité : TC utilise les deux notations de débits, en bits par seconde et en octets par seconde. La notation en kilobits utilise le paramètre kbit, alors que la notation en kilooctets utilise le paramètre kbps. Il ne faut pas les confondre, ici on utilise les kbit.



3.1 Planter son arbre

Tout d'abord, nettoyons toute qdisc existante :

tc qdisc del dev eth0 root

La première étape est de créer la qdisc racine, celle qui sera directement accrochée au périphérique (dans **skb->dev->qdisc**). Cette qdisc est de type HTB et ne comporte pas de paramètre de débit. On indique uniquement la valeur de r2q, réduite pour améliorer la gestion des faibles débits. Cette qdisc va envoyer par défaut les paquets dans la classe numérotée 99.

tc qdisc add dev eth0 root handle 1: htb default 99 r2q 5

On va maintenant créer une première branche, directement sous la racine, qui va appliquer le débit global de l'*uplink*. On ne positionne pas le paramètre **linklayer atm** sur cette branche, car on va mettre ce paramètre directement sur les classes. Dans le cas contraire, la valeur d'overhead serait calculée à plusieurs reprises (sur la branche et sur les classes) et additionnée, et on aurait alors une surestimation de l'overhead et une baisse de la bande passante utilisée.

tc class add dev eth0 parent 1:0 classid 1:1 htb rate 768kbit ceil 768kbit

En bonus, une étape non obligatoire, mais si votre Linux est également utilisé sur le réseau local, qui sera très probablement en gigabits, il est important de créer une seconde branche dans laquelle passeront les paquets à destination du réseau local. On accroche cette branche directement à la racine et on positionne son débit à 1 gbit.

tc class add dev eth0 parent 1:0 classid 1:1000 htb rate 1gbit ceil 1gbit

tc filter add dev eth0 parent 1:0 protocol ip prio 1000 handle 1000 fw flowid 1:1000

3.2 Interactive – prio 1

La classe interactive, comme toutes les classes à destination d'Internet qui

vont suivre, est définie avec le paramètre **linklayer atm**. Ce paramètre, que nous avons vu avec TBF, mais qui s'applique également à HTB, va permettre d'améliorer le calcul du débit.

Le burst est positionné à 5 kilooctets, ce qui devrait permettre de laisser passer les requêtes DNS d'un seul coup.

#---------------interactive

tc class add dev eth0 parent 1:1 classid 1:10 htb rate 128kbit ceil 200kbit burst 5k prio 1 linklayer atm

On accroche ensuite une qdisc de type pfifo avec une limite à 1000 paquets. On pourrait aussi utiliser bfifo à la place, qui mesure la limite non pas en paquets (p) mais en octets (bytes).

#--------------------------sub interactive: pfifo

tc qdisc add dev eth0 parent 1:10 handle 110: pfifo limit 1000

On va également créer un filtre, basique, pour envoyer les paquets portant la marque 10 vers la valeur de **classid** 10. Comme vu au tout début de l'article, les filtres sont directement attachés à la qdisc racine.

tc filter add dev eth0 parent 1:0 protocol ip prio 1 handle 10 fw flowid 1:10

3.3 TCP ACKs – prio 2

Dans le cas où le serveur Linux fait uniquement de l'hébergement, la priorisation des paquets **ACK** n'est pas une nécessité. Mais si votre serveur réalise des téléchargements régulièrement, ou agit comme passerelle Internet, alors c'est nécessaire.

On crée donc une classe ayant un débit de 64 kbit et pouvant emprunter. Grosso modo, le débit montant des paquets **ACK** représente 5% du débit descendant, on pourra donc télécharger un fichier à environ 1280 kbit par seconde sans souffrir de ralentissement. Évidemment, si l'emprunt est possible, on pourra monter jusqu'au débit maximum, soit 15360 kbits/s.

Ce trafic n'étant pas *bursty* du tout (un **ACK** fait environ 46 octets et le flux sera régulier en cas de téléchargement), on positionne une valeur de burst à 300 octets.

Entre parenthèses, ce petit calcul permet également de comprendre pourquoi les débits uplink des lignes ADSL sont figés à 1024 kbit/s. Ce n'est pas pour permettre l'hébergement, c'est simplement pour pouvoir profiter d'une ligne descendante de 20 mbit/s en TCP.

De même que précédemment, on positionne ici une qdisc en sortie de type pfifo avec une limite à 1000 et un filtre qui renvoie les paquets marqués 20 vers la **classid** 20,

#----------------tcp acks

tc class add dev eth0 parent 1:1 classid 1:20 htb rate 64kbit ceil 768kbit burst 300 prio 2 linklayer atm

#-------------------------sub tcp acks: pfifo

tc qdisc add dev eth0 parent 1:20 handle 120: pfifo limit 1000

tc filter add dev eth0 parent 1:0 protocol ip prio 2 handle 20 fw flowid 1:20

3.4 SSH – prio 3

Vous imaginez-vous que MacGyver sorte sans son couteau suisse ? Il ferait comment, ensuite, pour rentrer chez lui, démarrer sa voiture ou payer son boulanger ?

Ici, c'est pareil, avoir un accès SSH qui ne rame pas est indispensable quand on est en vacances et que le serveur se fait flooder. Le trafic SSH n'est pas bursty, les paquets sont émis à fréquences régulières, un bucket de grande taille n'est donc pas nécessaire.

En revanche, on mettra ici, et comme sur toutes les classes suivantes, une qdisc de type SFQ pour garantir un partage équitable du débit entre les connexions concurrentes.

tc class add dev eth0 parent 1:1 classid 1:30 htb rate 64kbit ceil 300kbit burst 2k prio 3 linklayer atm

#---------------------------sub ssh: sfq

tc qdisc add dev eth0 parent 1:30 handle 130: sfq perturb 10

tc filter add dev eth0 parent 1:0 protocol ip prio 3 handle 30 fw flowid 1:30

3.5 HTTP/HTTPS – prio 4

Pour le trafic web, on va mettre en place une hiérarchie de classes spécifiques. L'idée est, d'une part, de permettre le burst pour laisser passer rapidement les réponses aux requêtes venant d'Internet et, d'autre part, de réaliser une classification par virtual host afin de pouvoir estimer la bande passante de Site1 par rapport à Site2.

HTTP est un protocole bursty par excellence. Pour calculer la valeur de burst, il faut regarder plus en détail ce que cela signifie. Le fonctionnement requête/réponse de HTTP signifie que le client va envoyer une première requête, disons un **GET /**, et que le serveur va renvoyer la page par défaut, souvent « index.html ». Le client va alors parser le contenu de la page et lancer les requêtes correspondant aux éléments de la page afin de l'afficher.

Notre objectif ici est de permettre au serveur d'envoyer cette première page très rapidement, d'un seul coup. Or en faisant le test d'une connexion sans cache sur un site *dotclear* 2 classique (via *firebug*), la réponse à la première requête fait 7 Ko. Sur des sites plus importants, il n'est pas rare de voir des pages de 150 ou 200 Ko. En mettant donc un burst à 20 Ko, on assure à notre petit hébergement maison une bonne réactivité, mais en pénalisant également les autres classes pour une durée limitée : un burst de 160 kbits (20 Ko) prendra 1/5 de seconde à émettre (20ms), auquel il faut ajouter un tick max de 10ms, pour un total de 30ms (sans compter la latence de la liaison). Cela reste raisonnable mais pourra dégrader la qualité de service de la voix sur IP, par exemple.

Petit détail, les priorités des sous-classes sont les mêmes que celles de la branche, soit 4. Toutes les sous-classes seront donc traitées de la même façon en termes de partage de bande passante.

```
tc class add dev eth0 parent 1:1 classid 1:40 htb rate 256kbit ceil 768kbit burst 2k prio 4

#---------------------------http/s sub 1

tc class add dev eth0 parent 1:40 classid 1:41 htb rate 100kbit ceil 768kbit burst 2k prio 4 linklayer atm

#--------------------------------------------sub http1: sfq

tc qdisc add dev eth0 parent 1:41 handle 141: sfq perturb 10

tc filter add dev eth0 parent 1:0 protocol ip prio 4 handle 41 fw flowid 1:41

#---------------------------http/s sub 2

tc class add dev eth0 parent 1:40 classid 1:42 htb rate 100kbit ceil 768kbit burst 2k prio 4 linklayer atm

#--------------------------------------------sub http2: sfq

tc qdisc add dev eth0 parent 1:42 handle 142: sfq perturb 10

tc filter add dev eth0 parent 1:0 protocol ip prio 5 handle 42 fw flowid 1:42

#---------------------------http/s sub 9

tc class add dev eth0 parent 1:40 classid 1:49 htb rate 56kbit ceil 768kbit burst 2k prio 4 linklayer atm

#--------------------------------------------sub http 9: sfq

tc qdisc add dev eth0 parent 1:49 handle 149: sfq perturb 10

tc filter add dev eth0 parent 1:0 protocol ip prio 6 handle 49 fw flowid 1:49
```

3.6 Torrent – prio 5

Pourquoi parler de BitTorrent ici ? Ce n'est pas réellement un service hébergé et d'aucuns argueront que je fais ici l'apologie d'une utilisation non conforme à la réglementation. Que nenni ! Ce protocole m'intéresse uniquement pour son côté « chaotique » : il a, en effet, une fâcheuse tendance à créer des connexions entre des machines sur des ports aléatoires. Difficile donc de le faire rentrer dans une classe particulière. Et si, du côté de TC, la configuration n'a rien de nouveau ou de compliqué, c'est du côté de Netfilter, plus tard, que l'on utilisera des petites fonctionnalités sympathiques.

BitTorrent est un protocole fluide, on positionne donc une faible valeur de burst.

tc class add dev eth0 parent 1:1 classid 1:50 htb rate 128kbit ceil 768kbit burst 2k prio 5 linklayer atm

#---------------------------sub ssh: sfq

tc qdisc add dev eth0 parent 1:50 handle 150: sfq perturb 10

tc filter add dev eth0 parent 1:0 protocol ip prio 7 handle 50 fw flowid 1:50

3.7 La classe par défaut – prio 6

Et finalement, la classe par défaut. Un simple burst à 2 k et une bande passante garantie à 128 kbit pour être certain de ne pas pénaliser les applications environnantes.

tc class add dev eth0 parent 1:1 classid 1:99 htb rate 128kbit ceil 768kbit burst 2k prio 5 linklayer atm

#---------------------------sub ssh: sfq

tc qdisc add dev eth0 parent 1:99 handle 199: sfq perturb 10

tc filter add dev eth0 parent 1:0 protocol ip prio 99 handle 99 fw flowid

1:99

3.8 Admirer le résultat

Une fois que toutes ces règles sont chargées dans le noyau, on peut visualiser le résultat avec TC.

Pour visualiser les qdisc (commutateur **-s** pour les stats ou **-d** pour le détail) :

tc -d qdisc show dev eth0

qdisc htb 1: root r2q 5 default 99 direct_packets_stat 0 ver 3.17

qdisc pfifo 110: parent 1:10 limit 1000p

qdisc pfifo 120: parent 1:20 limit 1000p

qdisc sfq 130: parent 1:30 limit 127p quantum 1514b flows 127/1024 perturb 10sec

qdisc sfq 141: parent 1:41 limit 127p quantum 1514b flows 127/1024 perturb 10sec

qdisc sfq 142: parent 1:42 limit 127p quantum 1514b flows 127/1024 perturb 10sec

qdisc sfq 149: parent 1:49 limit 127p quantum 1514b flows 127/1024 perturb 10sec

qdisc sfq 150: parent 1:50 limit 127p quantum 1514b flows 127/1024 perturb 10sec

qdisc sfq 199: parent 1:99 limit 127p quantum 1514b flows 127/1024 perturb 10sec

Pour visualiser les classes (de même, **-s** ou **-d**) :

tc -s class show dev eth0

class htb 1:1 root rate 768000bit ceil 768000bit burst 1599b cburst 1599b

Sent 30660 bytes 231 pkt (dropped 0, overlimits 0 requeues 0)

rate 896bit 1pps backlog 0b 0p requeues 0

lended: 9 borrowed: 0 giants: 0

tokens: 15137 ctokens: 15137

class htb 1:99 parent 1:1 leaf 199: prio 5 rate 128000bit ceil 768000bit burst 2Kb cburst 1499b

Sent 30660 bytes 231 pkt (dropped 0, overlimits 0 requeues 0)

rate 896bit 1pps backlog 0b 0p requeues 0

lended: 222 borrowed: 9 giants: 0

tokens: 115295 ctokens: 13641

On retrouve ici toutes les informations que l'on a décrit plus tôt dans les algorithmes, comme les quantités de tokens disponibles (**ctokens** représente les tokens empruntables) ou encore le débit actuel de la classe.

3.9 MARKer son terrain

La QoS est en place, à part que, pour le moment, tous les paquets sont envoyés vers la classe 99 (la classe par défaut). En effet, on a bien une QoS et des filtres mais comme aucune marque n'est pour le moment appliquée, les paquets tombent toujours dans le cas « je ne sais pas où l'envoyer ».

Ce marquage des paquets, nous allons l'effectuer dans Netfilter. Mais où exactement dans Netfilter ? Comme vu en figure 3, tout au début, on dispose en réalité de 5 chaînes comportant chacune entre 2 et 4 tables. Or la toute dernière table de la dernière chaîne traversée avant transmission des paquets à TC est la table **MANGLE** de la chaîne **POSTROUTING**. C'est donc là que nous allons effectuer le marquage.

Dans la mesure du possible, on va également tenter de limiter au maximum le travail de Netfilter. Inutile de marquer **tous** les paquets, puisque Netfilter dispose d'un moteur de suivi de connexions (le *conntrack*). On va donc marquer un seul paquet de chaque connexion (quand le suivi est possible, comme avec TCP) et restaurer cette marque sur tous les paquets de la connexion. Ainsi, on en marquera un seul et la marque sera transposée par le noyau directement.

Netfilter exporte des informations dans le pseudo-système de fichier **/proc** qui permettent de visualiser le travail de conntrack. En faisant un **cat /proc/net/ip_conntrack**, on peut ainsi contrôler l'état des connexions (NEW, ESTABLISHED, ...), mais également la marque appliquée à une connexion, via la valeur « mark ». Attention toutefois, ce champ « mark » représente une marque de connexion, donc établie via la cible **CONNMARK**, et non pas une marque de paquet qui serait établie via la cible **MARK** (cette dernière n'est visualisable que dans la structure **sk_buff** correspondante, donc interne au noyau uniquement). Dans l'exemple ci-dessous, on voit une connexion portant la marque 49.

cat /proc/net/ip_conntrack|grep ESTABLISHED|grep "dport=80"

tcp 6 431999 ESTABLISHED src=175.112.129.215 dst=175.112.128.154 sport=1520 dport=80 packets=36450 bytes=1623170 src=175.112.128.154 dst=175.112.129.215 sport=80 dport=1520 packets=45874 bytes=68304451 [ASSURED] **mark=49** secmark=0 use=16

Ceci étant dit, nous pouvons commencer. D'abord, un peu de nettoyage :

Dans un premier temps, on peut commencer avec une règle simple : celle du réseau local. Admettons que ce réseau soit 192.168.1.0/24, la règle suivante va marquer tous les paquets à destination du LAN, sans distinction aucune.

Tous les paquets vers le LAN sont marqués à 1000

iptables -t mangle -A POSTROUTING -o eth0 -d 192.168.1.0/24 -j CONNMARK --set-mark 1000

Ensuite, les paquets de la classe interactive (VoIP, DNS et consorts). Libre à vous d'étendre la liste des ports.

Paquets de la classe interactive marqués à 10

iptables -t mangle -A POSTROUTING -o eth0 -p udp -m multiport --sports 53,123,161:162 -j CONNMARK --set-mark 10

iptables -t mangle -A POSTROUTING -o eth0 -p udp --dport 53 -j CONNMARK --set-mark 10

Le SSH n'est pas plus compliqué. On applique un masque pour ne marquer que les paquets **SYN**, **ACK** (le deuxième du *handshake* TCP) et on propagera la marque ensuite.

Paquets du serveur SSH local marqués à 30

iptables -t mangle -A OUTPUT -o eth0 -p tcp --tcp-flags SYN,ACK SYN,ACK --sport 22 -j CONNMARK --set-mark 30

3.9.1 Marquer les Virtual Hosts

Le serveur web maintenant. Celui-la est plus compliqué car on va trier sur le virtual host. Pour faire ce tri, il faut regarder la requête qui vient du navigateur du client et faire une correspondance sur le header HTTP « Host: » qui va contenir le vhost de destination.

Pour trier sur le champ **Host**, il faut cette fois positionner une règle sur les flux entrants, quand la requête HTTP arrive du client. On va utiliser le module *string* d'iptables, comme suit :

classe 41, vhost "www.toto.com"

iptables -t mangle -A INPUT -i eth0 -p tcp --dport 80 -m string --string "Host: www.toto.com" --algo bm ! -s 192.168.1.0/24 -j CONNMARK --set-mark 41

classe 42, vhost "wiki.linuxwall.info"

iptables -t mangle -A INPUT -i eth0 -p tcp --dport 80 -m string --string "Host: wiki.linuxwall.info" --algo bm ! -s 192.168.1.0/24 -j CONNMARK --set-mark 42

On va ainsi marquer les connexions dès leur entrée dans le système, avant même traitement par le serveur web (ce qui signifie également que ce n'est pas faisable sur les flux HTTPS). La marque sera propagée sur la connexion, et bien que la règle sortante marque tous les paquets à 49, c'est bien la marque 41 ou 42 qui sera appliquée et conservée.

Il semblerait que les prochaines versions de ce module supportent le *string matching* sans prise en compte de la casse (paramètre **–icase**), mais ce dernier ne fonctionnait pas sur la version du noyau utilisée pour les tests.

Il faut également prendre en compte le fait que ce matching ne peut être fait qu'une fois la connexion TCP établie et la requête HTTP du client envoyée, on va donc devoir attendre le 4ème paquet pour voir le champ **Host**.

Cela a un impact sur la règle générale qui suit, qui ne doit pas s'appliquer avant que les règles précédentes n'aient eu le temps de vérifier le champ **Host**. Dans le cas contraire, c'est la marque générale qui s'appliquerait à toutes les connexions.

Pour pallier cela, on va s'assurer que la règle générique ne s'applique pas tant que le flag **SYN** est présent.

De même, on va utiliser le module **connmark** afin de vérifier que les paquets sur le point d'être marqués ne le sont pas déjà. La combinaison de ces deux règles va nous permettre de ne marquer que les paquets qui ne rentrent pas dans les classes précédentes.

Paquets du serveur web marqués à 49

iptables -t mangle -A OUTPUT -o eth0 -p tcp --tcp-flags SYN,ACK ACK --sport 80 -m connmark --mark 0 ! -d 192.168.1.0/24 -j CONNMARK --set-mark 49

iptables -t mangle -A OUTPUT -o eth0 -p tcp --tcp-flags SYN,ACK ACK --sport 443 -m connmark --mark 0 ! -d 192.168.1.0/24 -j CONNMARK --set-mark 49

On reviendra sur le champ **tcp-flags** un peu plus loin, pour expliquer son fonctionnement.

3.9.2 Marquer le BitTorrent

Iptables dispose encore une fois d'un module nous permettant de résoudre le problème des connexions BitTorrent. Si l'on ne peut pas marquer les connexions car on ne sait pas sur quelles paires de ports elles vont s'ouvrir, il nous reste une possibilité : marquer le propriétaire de la socket.

Dans le cas présent, le logiciel BitTorrent qui va être utilisé est Rtorrent. Or nous allons lancer ce démon sous l'utilisateur rtorrent (via un **sudo -u rtorrent /usr/bin/rtorrent**).

Il faut donc, du côté d'iptables, utiliser le module **xt_owner** pour filtrer sur le propriétaire de la socket, de la façon suivante :

flux de l'utilisateur rtorrent marqués à 50

iptables -t mangle -A OUTPUT -o eth0 -p tcp --tcp-flags SYN SYN -m owner --uid-owner 1014 ! -d 192.168.1.0/24 -j CONNMARK --set-mark 50

Il faut simplement récupérer l'UID de l'utilisateur rtorrent dans le fichier **/etc/passwd**.

Cette technique, pratique pour marquer des flux sans avoir à suivre les paquets, impose toutefois que le démon **rtorrent** soit lancé sur la machine réalisant la QoS. Cela ne fonctionne pas en mode passerelle, car iptables ne connaît alors pas le propriétaire de la socket ayant émis les paquets.

Et pour le reste ? On ne fait rien, la qdisc racine envoyant par défaut les flux restants vers la classe 99.

Ensuite, on doit ajouter la règle qui propage les marques depuis les

paquets vers les connexions via le module **conntrack**.

Propagation des marks par conntrack sur les connexions

iptables -t mangle -A POSTROUTING -j CONNMARK --restore-mark

Et enfin, après la règle de propagation des connmark, on va placer la règle de marquage des paquets TCP ACKs. On place cette règle en tout dernier car elle va certainement matcher au moins un paquet par connexion TCP, et donc si on la place avant la propagation des marques, elle forcera toutes les connexions à porter la marque 20.

Paquets de la classe tcp acks marqués à 20

iptables -t mangle -A POSTROUTING -p tcp --tcp-flags URG,ACK,PSH,RST,SYN,FIN ACK -m length --length 40:64 -j MARK --set-mark 20

Petite subtilité ici : le masque **tcp-flags** qui peut être déroutant. Ce masque prend deux listes en paramètres. La première liste **URG,ACK,PSH,RST,SYN,FIN** contient les flags TCP que l'on va regarder. La seconde liste **ACK** contient les flags TCP qui doivent être positionnés à 1 pour que la règle fonctionne, imposant également que les autres flags soient à 0. Ainsi, notre règle signifie « regarde tous les flags et ne prends en compte que ceux dont seul le flag **ACK** est activé ». On ignore ainsi les paquets qui ne sont pas des acquittements. Mais il reste un second contrôle à effectuer, sur la taille du paquet cette fois. En effet, TCP utilise le paquet **ACK** pour envoyer des données, et positionne les flags **PUSH**, **ACK** pour indiquer à la pile distante qu'elle peut envoyer les paquets en couche applicative. Ici, en plus de matcher le flag **ACK**, il faut regarder la taille du paquet afin de vérifier qu'il ne transporte pas de données. C'est ce que l'on fait avec le module **limit**, qui ne prendra en compte que les paquets dont la taille annoncée dans le header *Total Length* (header IP) a une valeur comprise entre 40 et 64 octets. La marge de 24 octets permet de couvrir la présence potentielle des options TCP parfois positionnées à la fin des headers TCP.

Et voilà, nous avons maintenant une politique de QoS fiable et performante, et des règles iptables pour trier nos connexions vers les classes déterminées.

Vous pouvez très facilement tester ces règles avec netcat d'un côté et telnet de l'autre. On peut également visualiser l'état des classes et qdiscvia la ligne de commandes TC, mais il faut admettre que tout cela manque de saveur. Alors la prochaine et dernière étape, celle qui impressionne la voisine, c'est la visualisation de l'impact de la QoS. Et pour cela, il va falloir faire un peu de Perl.

4. Grapher tout cela avec RRDTool et un peu de Perl

Un classique pour grapher une utilisation réseau, c'est de coupler RRDTool avec Perl. Pour une utilisation industrielle, il est possible d'interfacer TC avec SNMP, ce qui pourrait être le sujet d'un article suivant. Pour le moment, on cherche à faire simple.

4.1 Créer la base avec RRDTool

Il va nous falloir installer le package RRDTool avant de commencer :

aptitude install rrdtool

Et l'on va créer une base RRD, de son petit nom **tcgraph.rrd**, qui va recevoir toutes les minutes les statistiques d'utilisation des classes depuis un programme Perl.

Le script ci-dessous permet de créer la base RRD :

RRASET 1 keep 1 record for 1 minute for 7 days (10080 records)

RRASET 2 keep 1 record for 60 minutes for 60 days (1440 records)

RRASET 3 keep 1 record for 720 minutes for 366 days (732 records)

rrdtool create tcgraph.rrd --start now --step 60 \

DS:interactive:COUNTER:120:0:786432 \

DS:tcp_acks:COUNTER:120:0:786432 \

DS:ssh:COUNTER:120:0:786432 \

DS:web_sub1:COUNTER:120:0:786432 \

DS:web_sub2:COUNTER:120:0:786432 \

DS:web_gen:COUNTER:120:0:786432 \

DS:torrent:COUNTER:120:0:786432 \

DS:default:COUNTER:120:0:786432 \

RRA:AVERAGE:0.5:1:10080 \

RRA:AVERAGE:0.5:60:1440 \

RRA:AVERAGE:0.5:720:732 \

Cette commande va nous créer un fichier **tcgraph.rrd**, préparé pour recevoir des compteurs (des entiers) dans 8 catégories, qui sont nos 8 classes de trafic (3 pour les flux web).

Les DS définissent les colonnes de la base et reprennent donc nos classes. Pour le détail des valeurs, voir la documentation officielle. Il faut spécialement faire attention à la dernière valeur 786432, qui représente le débit maximum de la connexion en bits par seconde.

Les RRA permettent de conserver les données de la façon suivante :

1. Premier niveau de RRA conserve une entrée par minute pendant 7 jours (10080 entrées) ;

2. Deuxième niveau de RRA conserve une entrée par 60 minutes pendant 60 jours (1440 entrées) ;

3. Troisième niveau de RRA conserve une entrée par 720 minutes pendant 366 jours (732 entrées).

Nous allons déplacer le fichier **tcgraph.rrd** vers le répertoire**/var/www/tcgraph** créé pour l'occasion.

4.2 Parser les statistiques de Traffic Control en Perl

Ensuite, il nous faut un script Perl qui va tourner en tâche de fond et parser toutes les minutes les statistiques de TC, puis injecter le résultat dans le fichier **tcgraph.rrd**.

Pour cela, il va nous falloir installer deux modules Perl : celui pour passer le programme en mode « daemon » et la bibliothèque RRD :

```
# aptitude install librrds-perl libproc-daemon-perl
```

Le code source ci-dessous sera ensuite à exécuter :

```
my $rrdfile = "/var/www/tcgraph/tcgraph.rrd";

my $logfile = "/var/www/tcgraph/tcgraph.log";

  my %valuelist = %classlist;

# récupère les stats de la ligne de commande

  open(TCSTAT,"tc -s class show dev eth0 |")

    || die "could not open tc command line";

    foreach my $class (keys %classlist){

      if ($_ =~ /\:$class parent/){

        my $nextline = <TCSTAT>;

        my @splitline = split(/ /,$nextline);

# on récupère une valeur en octets que
```

```
# l'on convertit en bits

$valuelist{$class} = $splitline[2]*8;

# on eleve cette classe de la liste à parser

delete $classlist{$class};

# injection dans RRD

my $updateline =
time().":$valuelist{'10'}:$valuelist{'20'}:$valuelist{'30'}:$valuelist{'41'}:$valuelist{'42'}:$valuelist{'49'}:$valuelist{'50'}:$valuelist{'99'}";

RRDs::update $rrdfile, "$updateline";

# on conserve une copie dans un fichier texte

open(TCGLOG,">>$logfile");

print TCGLOG "$updateline\n";

# dors pendant une minute
```

On peut lancer ce programme directement depuis la ligne de commandes, le module **Proc::Daemon** se chargera de fermer les descripteurs et de le rattacher à init.

```
# ps -edf|grep tcparsestat

root     10888     1 0 05:32 ?        00:00:00 perl tcparsestat.pl
```

Un petit tour dans le fichier **/var/www/tcgraph/tcgraph.log** nous permet de vérifier que le parsing fonctionne bien. Vous pouvez comparer les valeurs avec celles retournées par la commande **tc -s class show dev eth0** pour contrôler.

```
1265430749:10080:16029120:1544:0:0:0:0:151976
```

1265430809:10080:16064328:1544:0:0:0:0:151976

4.3 Générer les Graphs

La dernière étape consiste à générer les graphes depuis la base **tcgraph.rrd**. Cela est fait via les outils **rrdtool** et le script qui suit. Il est un peu lourd et difficile à lire car il faut définir la charte graphique de chaque classe.

Ce script prend deux paramètres. Le premier est le nom du fichier image de destination et le second est la période de temps en secondes à grapher. On peut rapidement générer un graphe des dernières 30 minutes de la façon suivante :

sh buildtcgraph.sh current.png -1800

qui va nous générer un fichier **current.png** des dernières 30 minutes d'activité.

Le code du script est ci-dessous :

rrdtool graph $1 --start $2 --title "Traffic Control on Uplink" \

--width 450 --vertical-label "bits/s" \

--color BACK#333333 --color SHADEA#000000 --color SHADEB#000000 \

--color CANVAS#000000 --color GRID#999999 --color MGRID#666666 \

--color FONT#CCCCCC --color FRAME#333333 \

DEF:interactive="/var/www/tcgraph/tcgraph.rrd":interactive:AVERAGE \

AREA:interactive#ffe400:interactive:STACK \

GPRINT:interactive:MAX:"max = %6.2lf%Sbps" \

```
GPRINT:interactive:LAST:"last = %6.2lf%Sbps" \

GPRINT:interactive:AVERAGE:"avg = %6.2lf%Sbps" \

COMMENT:\\s COMMENT:\\s \

DEF:tcp_acks="/var/www/tcgraph/tcgraph.rrd":tcp_acks:AVERAGE \

AREA:tcp_acks#b535ff:tcp_acks:STACK \

GPRINT:tcp_acks:MAX:"max = %6.2lf%Sbps" \

GPRINT:tcp_acks:LAST:"last = %6.2lf%Sbps" \

GPRINT:tcp_acks:AVERAGE:"avg = %6.2lf%Sbps" \

COMMENT:\\s COMMENT:\\s \

DEF:ssh="/var/www/tcgraph/tcgraph.rrd":ssh:AVERAGE \

AREA:ssh#1b7b16:ssh:STACK \

GPRINT:ssh:MAX:"max = %6.2lf%Sbps" \

GPRINT:ssh:LAST:"last = %6.2lf%Sbps" \

GPRINT:ssh:AVERAGE:"avg = %6.2lf%Sbps" \

COMMENT:\\s COMMENT:\\s \

DEF:web_sub1="/var/www/tcgraph/tcgraph.rrd":web_sub1:AVERAGE \

AREA:web_sub1#2a93ff:web_sub1:STACK \

GPRINT:web_sub1:MAX:"max = %6.2lf%Sbps" \

GPRINT:web_sub1:LAST:"last = %6.2lf%Sbps" \

GPRINT:web_sub1:AVERAGE:"avg = %6.2lf%Sbps" \

COMMENT:\\s COMMENT:\\s \
```

```
DEF:web_sub2="/var/www/tcgraph/tcgraph.rrd":web_sub2:AVERAGE \

AREA:web_sub2#b2ec00:web_sub2:STACK \

GPRINT:web_sub2:MAX:"max = %6.2lf%Sbps" \

GPRINT:web_sub2:LAST:"last = %6.2lf%Sbps" \

GPRINT:web_sub2:AVERAGE:"avg = %6.2lf%Sbps" \

COMMENT:\\s COMMENT:\\s \

DEF:web_gen="/var/www/tcgraph/tcgraph.rrd":web_gen:AVERAGE \

AREA:web_gen#ec0000:web_gen:STACK \

GPRINT:web_gen:MAX:"max = %6.2lf%Sbps" \

GPRINT:web_gen:LAST:"last = %6.2lf%Sbps" \

GPRINT:web_gen:AVERAGE:"avg = %6.2lf%Sbps" \

COMMENT:\\s COMMENT:\\s \

DEF:torrent="/var/www/tcgraph/tcgraph.rrd":torrent:AVERAGE \

AREA:torrent#4dfd:torrent:STACK \

GPRINT:torrent:MAX:"max = %6.2lf%Sbps" \

GPRINT:torrent:LAST:"last = %6.2lf%Sbps" \

GPRINT:torrent:AVERAGE:"avg = %6.2lf%Sbps" \

COMMENT:\\s COMMENT:\\s \

DEF:default="/var/www/tcgraph/tcgraph.rrd":default:AVERAGE \

AREA:default#ffffff:default:STACK \

GPRINT:default:MAX:"max = %6.2lf%Sbps" \
```

```
GPRINT:default:LAST:"last = %6.2lf%Sbps" \

GPRINT:default:AVERAGE:"avg = %6.2lf%Sbps" \
```

4.4 Des améliorations

Utiliser ce type de script est contraignant car il faut s'assurer qu'il soit lancé régulièrement (viacrontab), que l'on stocke les images, que l'on a une page HTML pour les afficher, etc. On peut faire mieux, par exemple, en modifiant le script CGI de Mailgraph pour afficher nos courbes. Je ne rentrerais pas dans le détail de la modification de Mailgraph ici, faute de place, mais le résultat est disponibles sur le wiki de linuxwall sous le nom de TCGraph.

L'idée est que le CGI va recalculer les images à chaque rechargement de page. A moins d'avoir constamment le nez sur les courbes (OK, je plaide coupable), on va donc réduire la charge liée à la création des images et se donner la possibilité de rafraîchir ces dernières plus souvent.

Et pour finir

Le meilleur moment de l'article, c'est celui où l'on découvre enfin que sa chaise a un dossier et que l'on peut s'enfoncer dedans en contemplant le résultat d'une belle implémentation de QoS. L'objectif de cet article n'était pas de fournir un script clé en main, car chacun a des besoins différents, mais de fournir les données pour comprendre et créer sa propre QoS. Il reste pourtant des points à approfondir, en particulier concernant des algorithmes comme HFSC, dont l'intérêt ne dépasse pas encore la complexité, ce qui explique peut-être que la seule documentation disponible (ou presque) soit le code source de l'implémentation dans le noyau.

Et pour finir, la cerise sur le gâteau, c'est la courbe en figure 9, que vous visualiserez sur votre machine une fois le travail terminé. On y retrouve, en simultané, des requêtes DNS en pagaille, du *download* qui implique la priorisation des TCP ACKs, des flux SSH, des flux web sur les trois sous-classes et un flux de zéros (netcat d'un côté, telnet de l'autre) dans la classe

par défaut.

Comme attendu, chacun dispose de son débit garanti et l'emprunt de débit se fait par ordre de priorité.

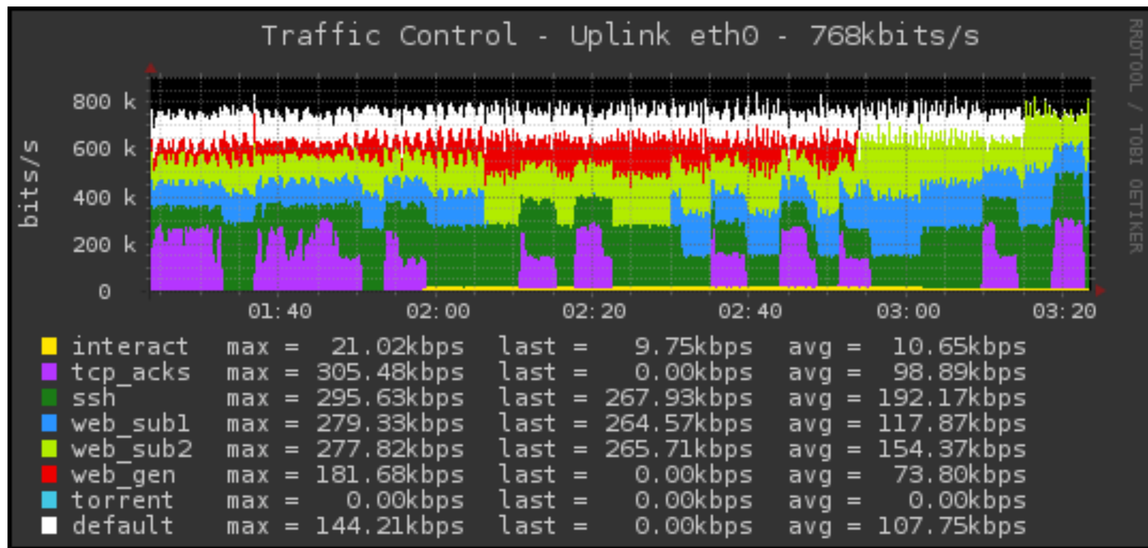


Figure 9 : Tcgraph en action

Remerciements

Un grand merci à David Bigot et Xavier Skapin pour leurs relectures et commentaires des plus pertinents.