Création d'un framework pour des routeurs avec support pour notification de vitesse explicite (kernel space)
Tuteur : Dino LOPEZ

Mathieu BIVERT, François CHAPUIS, Calypso PETIT, Sophie VALENTIN

2 février 2012

Table des matières

1 Introduction

L'essor de réseaux de grande envergure nécessite la mise en place de protocoles de contrôle de congestion. En effet, l'augmentation du traffic, et la distribution non-équitable des ressources entre les usagers, peuvent conduire à un ralentissement des communications. Une famille de protocoles de contrôle de congestion est l'IP-ERN (Explicit Rate Notification) [1]. Ils reposent sur les informations envoyées/calculées par des routeurs, comme le taux d'émission optimal. Dans le cadre de notre projet, nous devons élaborer un module noyau à destination d'un routeur supportant l'IP-ERN. Un tel module établiera des statistiques sur les paquets TCP qui traversent le routeur. Dans un premier temps, nous allons préciser la problématique du sujet et l'outil principal utilisé. Puis dans un second temps, nous présenterons les solutions mises en oeuvre pour l'élaboration des statistiques. Enfin, nous terminerons en faisant le bilan des résultats établis.

2 Description du problème

2.1 Contexte

Un routeur est une machine qui assure l'acheminement des données, d'un réseau informatique à un autre. Son rôle consiste à diriger les paquets d'une interface réseau vers une autre par le chemin le plus rapide et selon les règles définies dans la table de routage. Les interfaces réseaux peuvent être assimilées à des portes par lesquelles les données entrent et sortent du routeur, et la table de routage à une façon de diriger les paquets à travers ces portes. Il s'agit ici de calculer des statistiques nécessaires pour le protocole IP-ERN. Compte tenu de la courte durée du projet et du nombre important de concepts à assimiler (programmation de modules, management de mémoire, mécanismes de concurrence, structure du noyau, ...), nous nous sommes focalisés sur une statistique particulière : la taille de la file d'attente interne appelée Qsize.

En utilisant l'architecture de Netfilter, nous devons renseigner un tableau de statistiques comportant, pour chaque interface, le nombre de paquets entrants à destination de cette interface (Input Traffic Rate, I) et le nombre de paquets sortants (Output Traffic Rate, O) par cette interface. La quantité de donnée pouvant circuler sera donnée par l'Output Link Capacity (C). Le tableau sera de la forme donnée par 1.

Nom de l'interface	I	O	C	Q = I-O
eth0	...	...	...	...
eth1	...	...	...	...
...	...	...	...	...
eth$n$	...	...	...	...

FIG. 1 – Statistiques requises par le protocole

En comptabilisant l'Input trafic Rate et l'Output trafic Rate, on pourra déterminer la taille de la file d'attente (Qsize).

2.2 Netfilter

Netfilter est la première piste que nous avons décidé d'explorer pour parvenir à calculer Qsize puisque c'était l'outil à utiliser selon le sujet du projet. Netfilter est un framework permettant l'interception des paquets grâce à des hooks : ce sont des points d'accroche dans le noyau [2]. Lorsqu'un événement réseau se produit (entrée d'un paquet par exemple), une fonction de callback associée à ce type d'évènement est appelée. A chaque hook, Netfilter permet d'accepter les paquets ou de s'en débarasser. Pour le protocole IPv4, on compte cinq hooks. Ces derniers sont organisés comme sur la figure 2.

Les paquets entrent dans le routeur par le point d'entrée, après avoir subi des tests au niveau de la carte réseau. Ils traversent un premier hook, appelé *NF_IP_PRE_ROUTING*. Ensuite, après consultation de la table de routage, les paquets à destination du routeur sont dirigés vers un processus interne et passent par le deuxième hook, *NF_IP_LOCAL_IN*. Quant aux autres paquets, ils sont dirigés vers une interface de sortie et traversent successivement le troisième hook, *NF_IP_FORWARD* puis le quatrième,

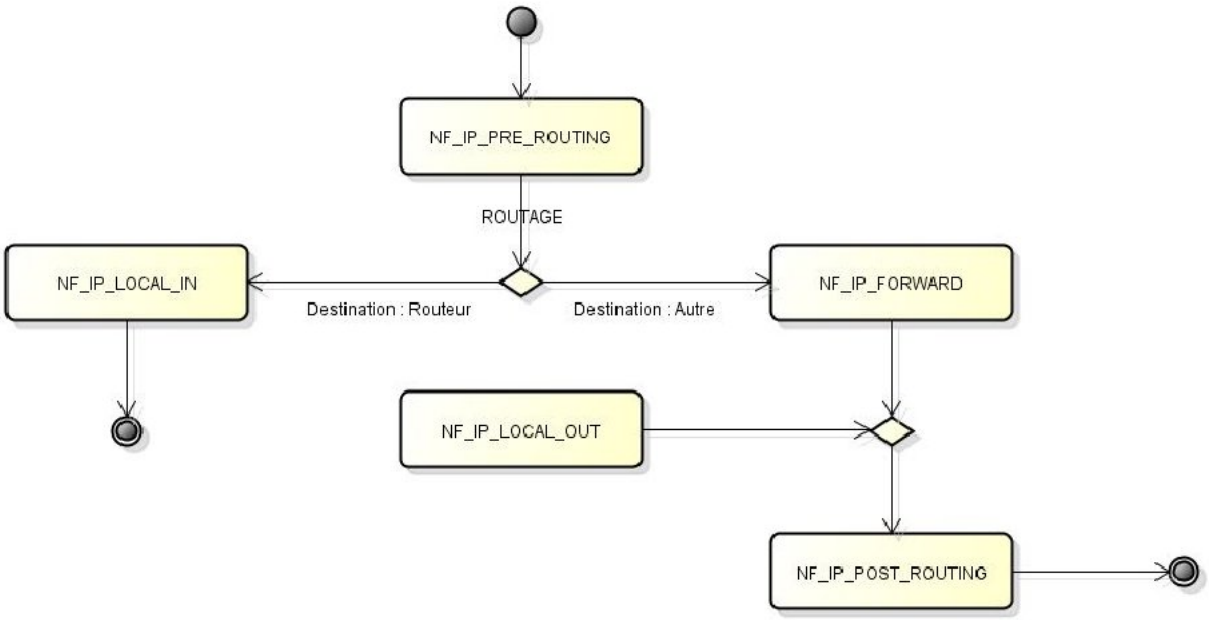

FIG. 2 – Position des hooks Netfilter

NF_IP_POST_ROUTING, avant de sortie du routeur. Les paquets peuvent aussi être créés par le routeur lui-même, ils passent alors par le hook *NF_IP_LOCAL_OUT*.

Netfilter va nous permettre, grâce aux hooks, d'ajouter des fonctionnalités au routeur sans toucher directement au code du noyau Linux. Afin d'utiliser Netfilter, il faut programmer un module en mode noyau. Les modules sont des morceaux de code écrits en langage C, pouvant être ajoutés ou retirés du noyau dynamiquement [3]. Ils apportent de nouvelles fonctionnalités au noyau. Dans notre cas, le module fournira des fonctions dites de callback qui seront automatiquement appelées lorsqu'un paquet traversera le hook associé à la fonction. Ces fonctions serviront à calculer les statistiques que nous devons fournir durant ce projet. En ce qui concerne la programmation des modules en mode noyau sous Linux, elle est vraiment différente de celle en mode utilisateur. En effet, les bibliothèques sont plus restreintes et le débogage plus ardu, car le moindre accès mémoire invalide peut rendre le système inutilisable. Les détails techniques concernant ce type de programmation sont évoqués en annexe [numéro de l'annexe].

3 Environnement de tests

3.1 Architecture

Comme nous l'avons dit ci-dessus, la programmation en mode noyau peut avoir un grave impact sur le fonctionnement de l'ordinateur en cas de bug. De ce fait, nous avons décidé d'utiliser des machines virtuelles. Celles-ci vont également nous permettre de mettre aisément en place les interfaces réseaux nécessaires aux tests.La mise en place des machines virtuelles se fera grâce au logiciel VirtualBox.

Dans un premier temps, on souhaite mettre en place une architecture réseau simple, constituée de trois machines, la figure 3 indiquant la topologie réseau suivie :

- un émetteur ;
- un routeur ;
- et un récépteur.

Pour cela nous utilisons trois machines virtuelles lancées sur la même machine hôte. Comme un routeur sert à faire transiter des données d'un réseau à un autre, deux réseaux sont nécessaires ici : un sur lequel se trouve l'émetteur et un pour le récepteur. VirtualBox permet de créer des réseaux locaux et les différentes machines virtuelles peuvent appartenir au réseau local que l'on souhaite [4]. Nous créons

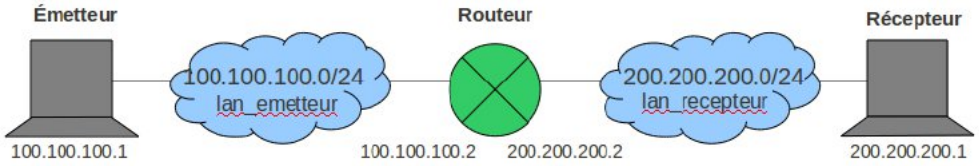

FIG. 3 – Topologie du réseau

donc deux réseaux locaux comme dans la figure 4.

Nom du réseau	CIDR	Masque de sous-réseau
lan_emetteur	100.100.100.0/24	255.255.255.0
lan_recepteur	200.200.200.0/24	255.255.255.0

FIG. 4 – Les deux réseaux

Sur chaque machine virtuelle, il faut configurer les interfaces réseaux et la table de routage. Ici l'émetteur peut atteindre le routeur. De même, le récepteur peut atteindre le routeur. En revanche, le routeur sert de passerelle entre l'émetteur et le récepteur. Le routeur doit simplement être connecté aux deux réseaux. Cela est possible car il dispose de plusieurs interfaces réseaux appelées eth0, eth1, eth2... Chacune des deux interfaces du routeur doit donc être connectée au bon réseau :

```
# ifconfig eth1 100.100.100.2 netmask 255.255.255.0
# ifconfig eth2 200.200.200.2 netmask 255.255.255.0
```

Enfin, il ne faut pas oublier d'activer le transfert de paquets :

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

Du côté de l'émetteur, l'interface eth0 est reliée à *lan_emetteur*, et il faut passer par 100.100.100.2 (c'est-à-dire l'interface eth1 du routeur) pour transmettre des paquets à destination du réseau lan_recepteur :

```
# ifconfig eth0 100.100.100.1 netmask 255.255.255.0
# route add -net 200.200.200.0 netmask 255.255.255.0 gw 100.100.100.2 eth0
```

Côté récepteur, la configuration est analogue à celle de l'émetteur :

```
# ifconfig eth0 200.200.200.1 netmask 255.255.255.0
# route add -net 100.100.100.0 netmask 255.255.255.0 gw 200.200.200.2 eth0
```

3.2 Test de connectivité

En guise de vérification, on essaye de *pinger* le récépteur depuis l'émetteur (fig.5) :

La commande *ping* n'est pas suffisante pour envoyer des paquets dans le cadre de notre projet car elle utilise le protocole ICMP et non le protocole IP. Par conséquent, nous avons cherché un outil utilisant les protocoles IP et TCP. La commande *iperf* permet d'envoyer entre un client et un serveur des paquets TCP ou UDP. Nous l'avons donc utilisé entre l'émetteur et le récépteur. Par défaut *iperf* utilise le protocole TCP.

Côté récépteur, on lance *iperf* en mode serveur :

```
# iperf -s
```

et côté émetteur, on envoie des paquets sur le récépteur :

```
# iperf -c 200.200.200.1
```

L'option *-i* d'*iperf* peut-être utilisée pour fixer l'intervalle de temps entre les rapports d'envois de paquets (mesure bande passante, ...).

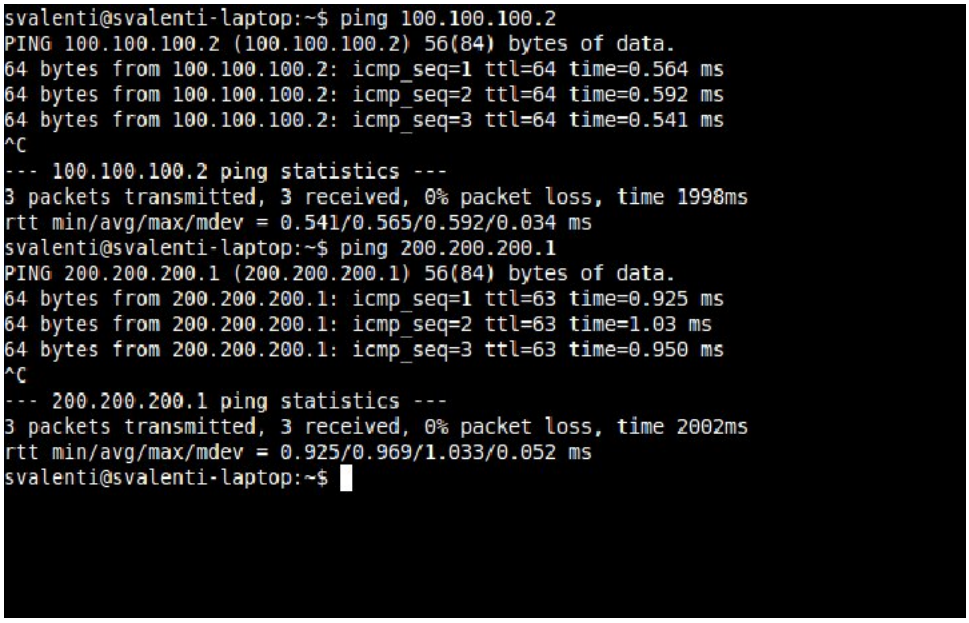

Fig. 5 – Émetteur et récépteur peuvent communiquer

4 Calcul de la taille de la file d'attente avec Netfilter

A présent, nous écrivons un module utilisant Netfilter. Les fonctions de callback données par les hooks permettent d'accéder à un *struct sk_buff* [5]. Cette structure représente un paquet dans le noyau Linux. Il est ainsi possible de récupérer les informations du datagramme telles que le protocole ou encore l'adresse IP de destination.

Les fonctions de callback fournissent également deux structures de type *struct net_device* : une pour l'interface réseau d'entrée et une autre pour l'interface réseau de sortie. Avec une telle structure, on peut connaître le nom de l'interface grâce au champ *net_device.name*.

Cependant, ces structures ne sont pas toujours remplies par le noyau : cela dépend du hook dans lequel nous nous trouvons. En effet, dans le hook *NF_IP_PRE_ROUTING*, seule l'interface réseau d'entrée du paquet est connue. Dans le hook *NF_IP_POST_ROUTING*, c'est le contraire : seule l'interface de sortie est renseignée. En revanche, le hook *NF_IP_FORWARD* connait les deux interfaces. Pour déterminer où se trouve la file d'attente, nous devons travailler avec deux hooks : le premier hook doit se trouver en amont de l'emplacement supposé de la file d'attente et le second en aval. Dans le premier hook, l'input trafic rate de l'interface de sortie est incrémenté et, dans le second hook, l'output trafic rate de l'interface de sortie est incrémenté.

Après chaque opération, on met à jour la taille de la file d'attente de l'interface de sortie en soustrayant l'Input Traffic Rate par l'Output Traffic Rate.

Les deux hooks pouvant accéder aux mêmes données en même temps, il est indispensable de mettre en place un mécanisme de synchronisation. Les hooks étant déclenchés par des interruptions, il est impossible d'utiliser des mutex ou encore des sémaphores [6]. Le noyau fournit des spinlocks, jouant le même rôle, mais dans un contexte d'interruptions. [7]

Les hooks concernant les paquets qui traversent le routeur sont : *NF_IP_PRE_ROUTING*, *NF_IP_FORWARD* et *NF_IP_POST_ROUTING*. Parmi ces hooks, seuls *NF_IP_FORWARD* et *NF_IP_POST_ROUTING* contiennent l'information sur l'interface de sortie dans la structure *net_device*. Par conséquent, nous allons tout d'abords chercher la taille de la pile entre *NF_IP_FORWARD* et *NF_IP_POST_ROUTING*.

4.1 *NF_IP_FORWARD* et *NF_IP_POST_ROUTING*

Pour commencer, les vitesses maximales des interfaces eth1 et eth2 sont configurées de la même façon. Ainsi, on s'attend à ce que la sortie des paquets soit fluide : nous ne nous attendons pas à la création d'une file d'attente interne.L'exécution du module utilisant ces deux hooks donne l'extrait de trace en

fig.6.

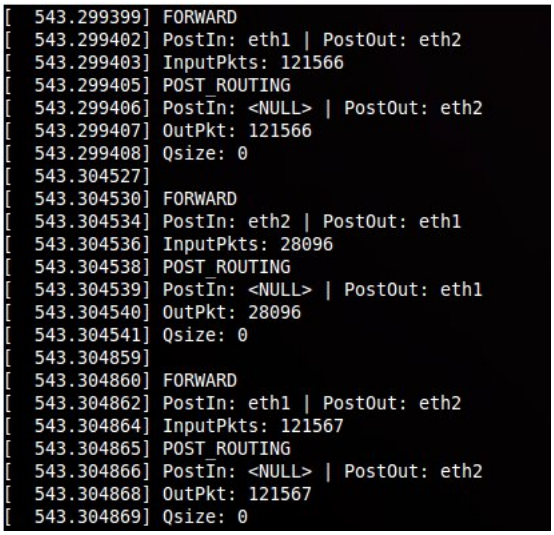

FIG. 6 – Trace obtenue entre FORWARD et POST

Après examination entière de la trace, on constate que Qsize pour eth1 et Qsize pour eth2 sont constamment égaux à zéro. On peut tout de même vérifier que le nombre de paquets entrants est incrémenté correctement à chaque exécution de la fonction de callback de *NF_IP_FORWARD*. Quant au nombre de paquets sortants, il est incrémenté correctement à chaque exécution de la fonction de callback du hook *NF_IP_POST_ROUTING*.

À présent, on configure la bande passante maximale de l'interface eth2 afin que la vitesse maximale soit de 10 Mbits/seconde. Pour cela, on utilise la commande *tc* (traffic control) [8] [9]. Cette commande va agir sur le fonctionnement des files d'attente en sortie d'une interface donnée.

```
# tc qdisc add dev eth2 root handle 1: tbf rate 10mbit buffer 1600 limit 3000
# tc qdisc add dev eth2 parent 1: handle 10: netem delay 3ms
```

Pour limiter la bande passante, on utilise un Token Bucket Filter (tbf) en précisant qu'on ne peut pas dépasser un débit de 10Mbits/seconde.

Après avoir ainsi configuré la vitesse maximale, nous nous attendons à obtenir une file d'attente non nulle. Les résultats des tests effectués nous montrent que la taille de la file d'attente est toujours nulle, ce qui prouve que la file d'attente interne ne se trouve pas entre les hooks *NF_IP_FORWARD* et *NF_IP_POST_ROUTING*.

Il nous faut donc regarder entre deux autres hooks : *NF_IP_PRE_ROUTING* et *NF_IP_POST_ROUTING*.

4.2 *NF_IP_PRE_ROUTING* et *NF_IP_POST_ROUTING*

L'interface de sortie n'étant pas disponible dans le hook *NF_IP_PRE_ROUTING*, il va nous falloir effectuer le routage à la main pour déterminer à quelle interface de sortie le paquet est destiné. Notre objectif étant de chercher où se trouve la file, nous nous contentons d'un routage manuel, avec des adresse IP fixes dans le code.

Nous obtenons finalement le même résultats que précédemment. Il semble donc qu'il soit impossible d'utiliser Netfilter pour obtenir les statistiques.

4.3 Lecture des statistiques sur les cartes réseaux

Grâce à une fonction du noyau, il est possible d'accéder à des statistiques concernant les interfaces. La fonction *dev_get_stats* définie dans *linux/netdevice.h* fournit :

- le nombre de paquets (ou octets) reçus sur une interface ;
- le nombre de paquets (ou octets) transmis par une interface ;

– le nombre d'erreurs ;
– et le nombre de paquets "dropped".

Ces statistiques pourraient nous aider à calculer Qsize. Cependant, elles ne permettent pas de différencier les paquets TCP des autres paquets. Il nous faut donc aller un peu plus loin dans le code.

5 Localisation des files d'attente dans la couche liaison

Après tous les tests que nous avons effectué précédemment, nous pouvons affirmer que la file d'attente contenant les paquets qui n'ont pas encore été traités ne se situe pas au niveau du routeur, c'est-à-dire dans la couche Réseau du modèle OSI. Par conséquent, nous allons tenter de la localiser dans la couche inférieure du modèle OSI : la couche Liaison. Voici le schéma représentant les différentes phases du traitement d'un paquet :

Schéma !

5.1 Les files d'attente avant le routage

Comme on peut le voir sur ce schéma issu de nos recherches, à la reception des paquets, ces derniers sont stockés dans une file d'attente appelée backlog [10] ou ingress queue, dans la couche Liaison. Il en existe une par processeur. Lorsqu'un paquet entre, une interruption appelée *NET_RX_SOFTIRQ* est levée. On peut choisir de traiter cette interruption grâce à un handler qui est une fonction appelée automatiquement lorsque l'interruption est levée. Nous avons donc essayer d'incrémenter le nombre de paquets dans ce handler et de continuer à le décrémenter dans le hook *NF_IP_POST_ROUTING*. Malheureusement, le traitement des interruptions en mode noyau est avéré être plus compliqué que ce que nous pensions. Et nous n'avons pas voulu perdre de temps sur ce point puisque, par ailleurs, les recherches que nous avons fait nous ont appris que ces files d'attente sont liées à la vitesse de traitement des paquets [11] [12]. De telles files ne nous intéressent donc pas car nous nous intéressons uniquement aux files d'attente liées aux vitesses d'émission des paquets.

5.2 Les files d'attente après le routage

5.2.1 File d'attente entre les fonctions dev_queue_xmit et hard_start_xmit

Après le passage dans les hooks Netfilter, le paquet traverse la couche Liaison [13] [14]. A ce niveau, il y a une mise en file d'émission pour chaque interface de sortie. Cette file d'émission existe entre la fonction noyau *dev_queue_xmit* et la fonction *hard_start_xmit*. La première fonction empile le paquet lorsqu'il passe de la couche Réseau à la couche Liaison. La deuxième fonction a pour but d'indiquer au noyau que les paquets ont été transmis à la carte réseau. Nous pouvons donc incrémenter le nombre de paquets au niveau de la fonction *dev_queue_xmit*, qui prend en compte l'interface de sortie.

Il n'est pas possible d'écrire un code modulaire pour ce type de manipulation. Nous sommes contraints de modifier le code du noyau. Cela implique la compilation d'un nouveau noyau, tâche assez lourde.

5.2.2 La compilation du noyau Linux

Pour pouvoir compiler un noyau Linux [15], il nous a fallu télécharger son code source. Tout d'abord, nous avons télécharger sur le web une des versions les plus récentes. Mais, avec cette version, nous n'avions pas le bon fichier de configuration. De ce fait, la compilation du noyau prenait énormément d'espace sur le disque dur de la machine virtuelle et cela a rendu inutilisable un certain nombre de machines virtuelles malgré nos tentatives de leur allouer d'avantage d'espace disque. Cette étape nous a fait perdre beaucoup de temps. Finalement, nous avons téléchargé un autre noyau moins récent grâce à la commande :

```
apt_get install linux-source-2.6.32
```

Nous avons choisi la version 2.6.32 car c'est celle que nous avions sur nos machines, cela nous a permis de récupérer le fichier de configuration et la compilation prend beaucoup moins de place. Après avoir téléchargé le noyau, il y a plusieurs étapes pour le compiler. Le téléchargement nous procure un fichier compressé, il faut donc le décompresser :

```
# tar xvjf linux-2.6.32.tar.bz2
```

Une fois la décompression effectuée, il faut placer le fichier de configuration trouvé dans /boot dans le dossier contenant le noyau et le renommer en .config. Ensuite, il faut créer l'image binaire du noyau qui servira pour booter sur notre nouveau noyau :

```
# make bzImage
```

Les modules, qui comprennent notamment les pilotes utiles pour l'utilisation du clavier et de la souris par exemple, peuvent être compilés et installés :

```
# make modules
# make modules_install
```

Enfin, on peut compiler et installer le noyau lui-même :

```
# make
# make install
```

Pour pouvoir booter sur ce noyau ainsi installé, il faut taper la commande :

```
# update-grub
```

et redémarrer l'ordinateur.

A chaque modification d'un des fichiers sources du noyau, il faut refaire le make et le make install.

5.2.3 Un premier essai dans dev_queue_xmit

En sortie, il existe une politique de gestion de files d'attente appelée qdisc. Il s'agit d'une structure comportant deux pointeurs sur fonction enqueue et dequeue. Il existe une qdisc par interface de sortie. Nous avons donc cherché à connaître le contenu effectif en paquets TCP de la file d'attente. Il est clair que l'incrément du nombre de paquets TCP doit se faire au début du corps de la fonction *dev_queue_xmit* quand un appel à la fonction pointée par enqueue est fait. En revanche, nous avons vu que la fonction *hard_start_xmit* est implémentée au niveau des drivers. Cela implique que l'implémentation de la fonction diffère selon la carte réseau. Pour pallier ce problème, nous voulons décrémenter le nombre de paquets à un niveau supérieur : par exemple au niveau de l'appel à la fonction pointée par dequeue. Cependant, le paquet peut très bien êtredépilé de *qdisc*, mais ne pas être envoyé. Il est donc impératif de se placer à un niveau inférieur.

5.2.4 Wrapper pour les drivers ethernet (npkt–)

Les fonctions permettant de manipuler une carte ethernet sont sous linux abstraite par une structure : ajouter un driver revient donc à déclarer une structure et à remplir les différents champs, avec les fonctions adaptées à la carte. La structure *net_device_ops*, déclarée dans *linux/netdevice.h* joue ce rôle :

```
struct net_device_ops {
    int (*ndo_init)(struct net_device *dev);
    void (*ndo_uninit)(struct net_device *dev);
    int (*ndo_open)(struct net_device *dev);
    int (*ndo_stop)(struct net_device *dev);
    netdev_tx_t (*ndo_start_xmit) (struct sk_buff *skb, struct net_device *dev);
    ...
};
```

Et un driver, par exemple e1000, l'utilise ainsi :

```
static const struct net_device_ops e1000_netdev_ops = {
    .ndo_open               = e1000_open,
    .ndo_stop               = e1000_close,
    .ndo_start_xmit         = e1000_xmit_frame,
    .ndo_get_stats          = e1000_get_stats,
    ...
};
```

La fonction *dev_hard_start_xmit*, située dans *net/core/dev.c*, est chargée de transmettre le paquet à la carte. Après quelques tests à l'aide de *printk* bien placé, nous avons constaté que dans tous les cas, la fonction est appellée, plus ou moins directement, et qu'elle contient un appel à *ndo_start_xmit* qui lancera la transmission réelle des paquets. La décrementation de la taille de la pile peut donc être effectuée ici, plutôt que de l'être dans le noyau.

6 Résultats

En réglant la vitesse et la taille du buffer avec tc, on peut observer une augmentation de qsize : les paquets rentrent, mais ne sortent jamais. L'hypothèse que nous avons par la suite vérifiée est que ces paquets sont droppés par la carte : en regardant la valeur de retour (*rc*) à la fin de *dev_queue_xmit*, on voit bien qu'elle est à *NET_XMIT_DROP* à chaque fois que qsize n'est pas décrémenté en sortie :

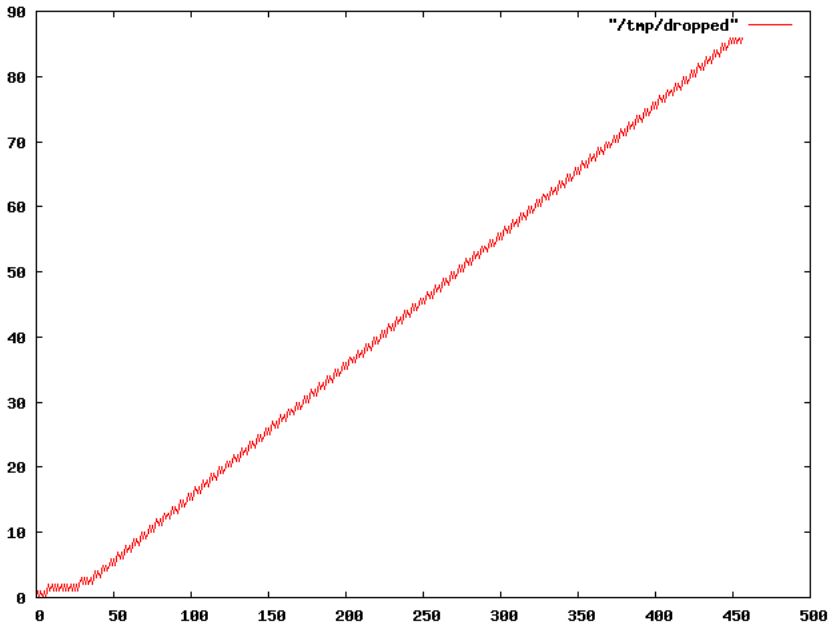

FIG. 7 – Après réglage avec tc(1)

Sans effectuer de réglages, les paquets ne sont pas droppés, et, selon des facteurs indeterminés (vitesse carte ? TODO), nous observons soit un qsize toujours à 0, soit un qsize grandissant à l'occasion, mais se réduisant par la suite :

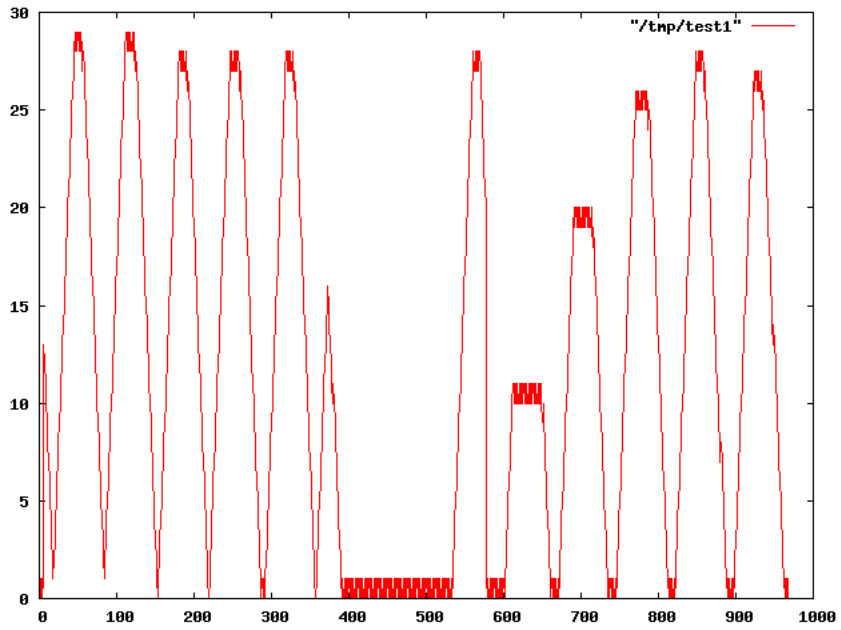

FIG. 8 – Sans réglage

Références

[1] D.-M. LOPEZ-PACHECO, *Propositions for a robust and inter-operable eXplicit Control Protocol on heterogeneous high speed networks*. PhD thesis, Ecole Normale Supérieure de Lyon, 2008.

[2] `http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO-3.html`. Architecture de Netfilter.

[3] `http://linux.die.net/lkmpg/index.html`. The Linux Kernel Module Programming Guide.

[4] `http://www.virtualbox.org/manual/ch06.html`. Gestion du réseau avec VirtualBox.

[5] `http://vger.kernel.org/~davem/skb.html`. La structure sk_buff.

[6] `http://stackoverflow.com/questions/7427402/some-question-about-kernel-object`. Mutexes and semaphores in interrupt context.

[7] `http://www.linuxgrill.com/anonymous/fire/netfilter/kernel-hacking-HOWTO-5.html`. Locking (mutex, spinlocks).

[8] `http://wiki.linuxwall.info/doku.php/fr:ressources:dossiers:networking:traffic_control/`. Voyage au centre du noyau : Traffic Control, la QoS.

[9] A. KELLER, "Manual tc packet filtering and netem," 2006.

[10] J. L. Longsong Lin, Tianji Jiang, "A generic traffic conditioningmodel for differentiated services."

[11] `http://ftp.gnumonks.org/pub/doc/packet-journey-2.4.html`. The journey of a packet through the linux 2.4 network stack.

[12] C. Benvenuti, "Understanding linux network internals."

[13] `http://download.intel.com/design/intarch/PAPERS/323704.pdf`. Linux Network Stack.

[14] `http://gicl.cs.drexel.edu/people/sevy/network/Linux_network_stack_walkthrough.html`. Linux Network Stack Functions.

[15] `http://ftp.traduc.org/doc-vf/gazette-linux/html/2005/111/lg111-C.html`. Compilation du noyau Linux.

Autre documentation consultée :

- Commande mii-tool (configuration vitesse) : http ://www.cyberciti.biz/faq/linux-change-the-speed-and-duplex-settings-of-an-ethernet-card/
- Source du noyau : http ://lxr.linux.no/
- COMPLETER

A Compléments sur la programmation de modules Linux

A.1 Organisation générale

A.2 Mise en place d'un hook netfilter

A.3 Synchronisation (spinlock)

A.4 Compilation d'un noyau Linux