Mathieu BIVERT, François CHAPUIS,
Calypso PETIT, Sophie VALENTIN

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

Tuteur : Dino LOPEZ

3 février 2012

Table des matières

1 Introduction

Tous les réseaux de type Ethernet nécessitent un contrôle de congestion. En effet, l'augmentation du trafic, et la distribution non-équitable des ressources entre les usagers, peuvent conduire dans le meilleur des cas à un ralentissement des communications et dans le pire des cas à des *congestion collapses*. La clé du succès de l'Internet est la mise en place du contrôle de congestion.

Une famille de protocoles de contrôle de congestion est l'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'ERN. Un tel module établira des statistiques sur les paquets ERN[1] 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 øeuvre pour l'élaboration des statistiques et leurs résultats. Enfin, nous conclurons sur ce qui a été fait et ce qu'il reste à faire.

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 meilleur chemin[2] 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 ERN. Un routeur ERN a besoin de connaître :

- l'Input Traffic Rate ;
- le Qsize ;
- l'Output Capacity Link (déjà connu).

Compte tenu de la courte durée du projet et du nombre important de concepts à assimiler (programmation de modules, gestion de la mémoire, mécanismes de concurrence, structure du noyau, ...), nous nous sommes focalisés sur une statistique particulière : l'occupation réelle instantanée de la file d'attente, 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 Packets, I) et le nombre de paquets sortants par cette interface (Output Packets, O). Le tableau sera de la forme donnée par le tableau 1.

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

FIG. 1 – Statistiques requises par le protocole

En comptabilisant I et O, on pourra déterminer Qsize à tout moment. Notons que la statistique Qsize est associée à une interface de sortie mais tient compte des paquets entrants par toutes les interfaces.

2.2 Netfilter

Netfilter est la première piste que nous avons décidé d'explorer pour parvenir à calculer Qsize puisque nous avons vu qu'un routeur ERN a été développé en utilisant Netfilter [2]. Netfilter est un framework

[1] Un paquet ERN est en fait un paquet TCP avec des champs additionnels (dans les options TCP). Ces champs additionnels sont : le DrdRate (Desired Rate), le RTT (Round-trip delay time) et le cwnd (valeur courante de la fenêtre de congestion)

[2] Le meilleur chemin est défini par les politiques de routage. Il peut s'agir du chemin le plus court, le plus fiable ou le plus sécurisé.

permettant l'interception des paquets grâce à des hooks : ce sont des points d'accroche dans le noyau [3]. Lorsqu'un événement réseau se produit (entrée d'un paquet par exemple), une fonction de callback associée à ce type d'événement est appelée. À chaque hook, Netfilter permet d'accepter les paquets ou de s'en débarrasser. Pour le protocole IPv4, on compte cinq hooks. Ces derniers sont organisés comme sur la figure 2.

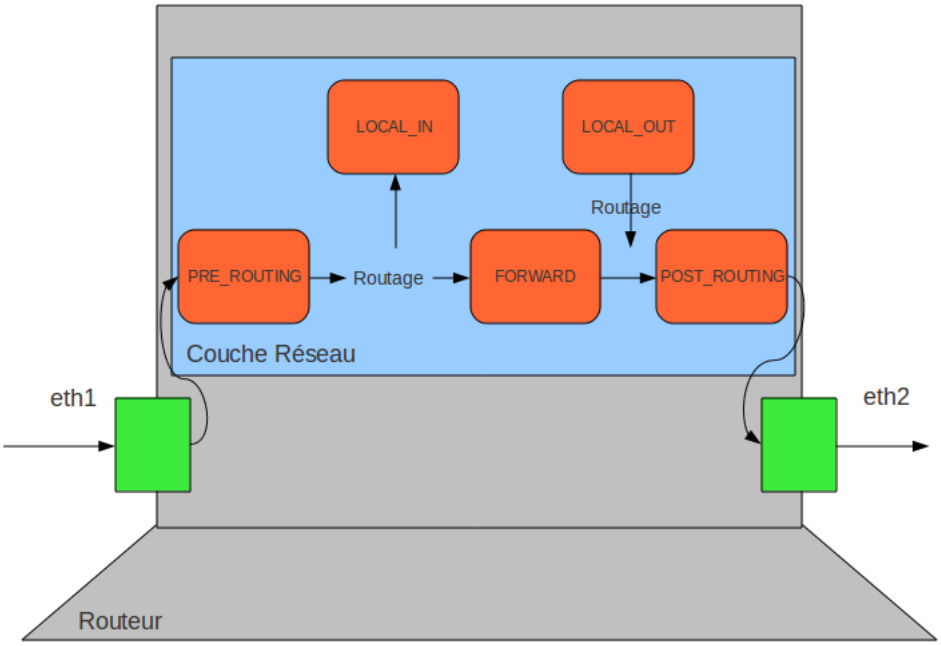

FIG. 2 – Position des hooks Netfilter

Les paquets entrent dans le routeur par l'interface réseau eth1 et subissent 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, *NF_IP_POST_ROUTING*, avant de sortir du routeur par l'interface réseau eth2. 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 [4]. 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.

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[5].

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 récepteur.

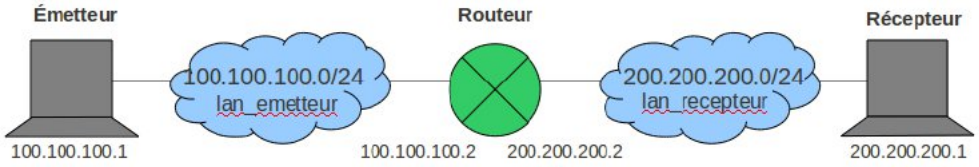

FIG. 3 – Topologie du réseau

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 [6]. Nous créons donc deux réseaux locaux comme dans le tableau 4.

Nom du réseau	Adresse réseau	Masque 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 essaie de *pinger* le récepteur depuis l'émetteur (figure 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 TCP. Par conséquent, nous avons cherché un outil utilisant le protocole 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 récepteur. Par défaut *iperf* utilise le protocole TCP.

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

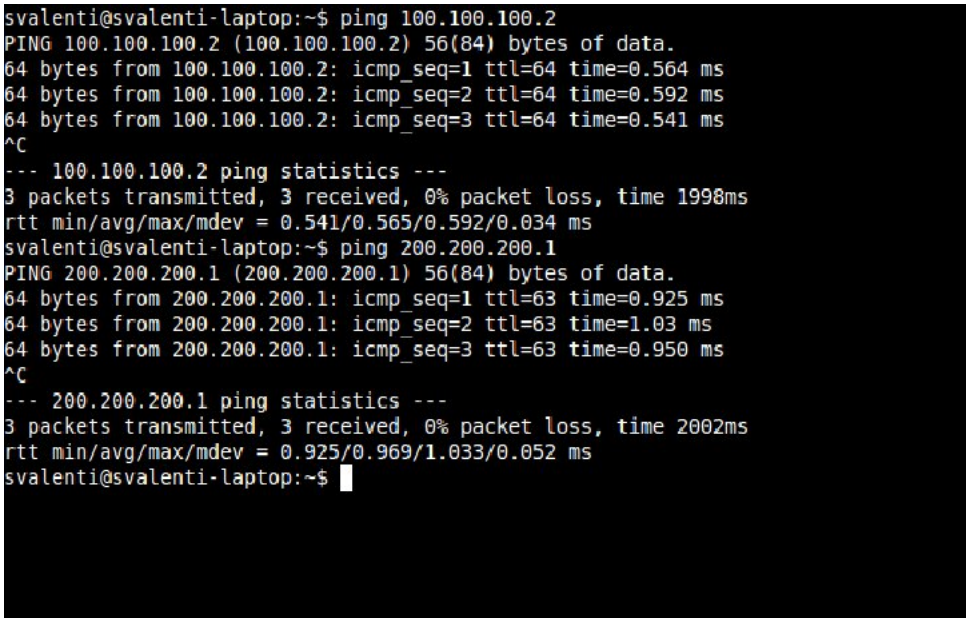

FIG. 5 – Émetteur et récepteur peuvent communiquer

```
# iperf -s
```

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

```
# 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, ...).

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

À présent, nous écrivons un module utilisant Netfilter. Les fonctions de callback données par les hooks permettent d'accéder à un *struct sk_buff* [7]. 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, la valeur de I de l'interface de sortie est incrémenté et, dans le second hook, la valeur de O 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 Packets par l'Output Packets.

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 [8]. Le noyau fournit des spinlocks, jouant le même rôle, mais dans un contexte d'interruptions. [9]

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'abord chercher la taille de la file 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 figure 6.

```
[  543.299399] FORWARD
[  543.299402] PostIn: eth1 | PostOut: eth2
[  543.299403] InputPkts: 121566
[  543.299405] POST_ROUTING
[  543.299406] PostIn: <NULL> | PostOut: eth2
[  543.299407] OutPkt: 121566
[  543.299408] Qsize: 0
[  543.304527]
[  543.304530] FORWARD
[  543.304534] PostIn: eth2 | PostOut: eth1
[  543.304536] InputPkts: 28096
[  543.304538] POST_ROUTING
[  543.304539] PostIn: <NULL> | PostOut: eth1
[  543.304540] OutPkt: 28096
[  543.304541] Qsize: 0
[  543.304859]
[  543.304860] FORWARD
[  543.304862] PostIn: eth1 | PostOut: eth2
[  543.304864] InputPkts: 121567
[  543.304865] POST_ROUTING
[  543.304866] PostIn: <NULL> | PostOut: eth2
[  543.304868] OutPkt: 121567
[  543.304869] Qsize: 0
```

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 netem [10] avec la commande *tc* (traffic control) [11] [12]. 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. Dans la pratique, la capacité des interface (C) est passée en paramètre au chargement du module. Voici les rapports d'iperf en figure 7

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 et qu'on n'a jamais deux appels consécutifs au hook *NF_IP_FORWARD*. Les paquets sont donc traités complètement : la file d'attente 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 adresses IP fixes dans le code.

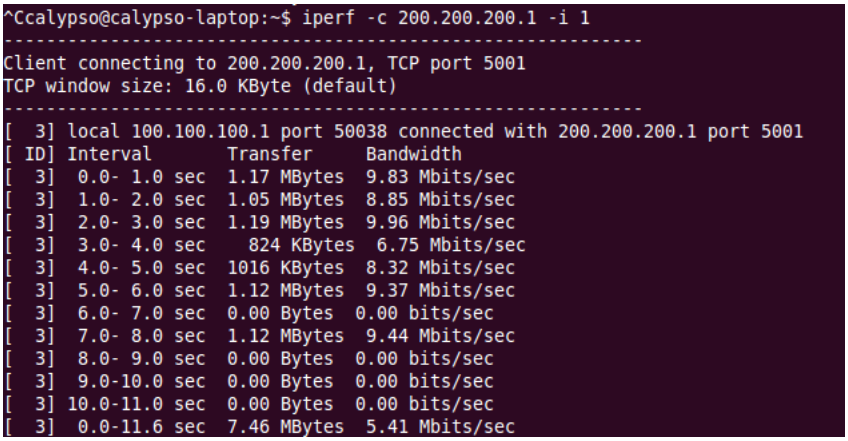

FIG. 7 – Rapport de bande passante d'iperf

Nous obtenons finalement le même résultat 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. Effectivement, pour une interface de sortie donnée on connaît le nombre de paquets transmis par cette interface. Il faut se placer maintenant sur chacune des autres interfaces, et comptabiliser le nombre de paquets entrants à destination de notre interface de sortie. Cela est possible à condition d'effectuer un routage des paquets pour connaître leur destination.

Cependant, les statistiques sur les cartes réseaux ne permettent pas de différencier les paquets TCP des autres paquets, il faudrait donc compter ceux-ci séparemment. De plus, le routage évoqué est assez lourd. Il nous faut donc aller un peu plus loin dans le code.

5 Analyse de la file d'attente au niveau de 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 dans la couche Réseau du modèle OSI. Par conséquent, nous allons tenter de l'analyser 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 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 [13] ou ingress queue, dans la couche Liaison. 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 essayé 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 s'est avéré être plus compliqué que ce que nous pensions. Et nous n'avons pas voulu perdre de temps

sur ce point puisque nos recherches nous ont appris que dans le cadre d'un routeur ERN, seul le taux d'occupation de la file d'attente entre en jeu. De telles files ne nous intéressent donc pas.

5.2 Les files d'attente après 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 [14] [15]. À 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 enfile 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, et peu pratique : un module a le gros avantage de ne pas être dépendant des sources du noyau.

5.2.2 La compilation du noyau Linux

Pour pouvoir compiler un noyau Linux [16], il nous a fallu télécharger son code source. Tout d'abord, nous avons téléchargé 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 inutilisables un certain nombre de machines virtuelles malgré nos tentatives de leur allouer davantage 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.

```
# cp /boot/config-2.6.32 .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.

À 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 la fonction dev_queue_xmit()

En sortie, il existe une politique de gestion de files d'attente appelée qdisc. Il s'agit d'une structure comportant entre autre 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. Nous avons aussi vu que la fonction *dequeue* est appelée indirectement par *dev_hard_start_xmit()* Cependant, le paquet peut très bien être dépilé de *qdisc*, mais ne pas être envoyé. Il est donc impératif de se placer à un niveau inférieur.

5.2.4 Un second essai en utilisant la fonction dev_hard_start_xmit()

Comme on l'a dit précédemment, à chaque fois que la fonction *dev_queue_xmit()* est appelée, un paquet est mis en bufferisation. Nous avons vu que le décrément au niveau du pointeur de fonction *dequeue* ne fonctionnait pas : en fait, le paquet est dépilé à la sortie de la couche transport, alors que la décrémentation devrait plutôt être réalisée une fois le paquet mis sur le réseau. Par conséquent, nous avons cherché une fonction mettant le paquet dans la file d'attente de la carte réseau. Il s'agit de la fonction *ndo_start_xmit()* appelée dans *dev_hard_start_xmit()*.

En fait, chaque driver implémente une fonction permettant l'envoi de paquets sur le réseau. Une solution naïve consisterait à chercher le code du bon driver, et à y placer la décrémentation. Mais alors le code ne fonctionnerait plus qu'avec un seul driver, à moins de tous les modifier.

Les fonctions permettant de manipuler une carte Ethernet sont sous Linux abstraites 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 le driver e1000[3], 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 appelée, plus ou moins directement, et qu'elle contient un appel à *ndo_start_xmit()* qui lancera la transmission réelle des paquets. La décrémentation de la taille de la file peut donc être effectuée ici.

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

[3]Driver très répandu, utilisé par la carte Intel. Il est présent notamment sur VirtualBox.

la fin de *dev_queue_xmit()*, on voit bien qu'elle est à *NET_XMIT_DROP* au lieu de *NET_XMIT_SUCCESS* à chaque fois que Qsize n'est pas décrémenté en sortie :

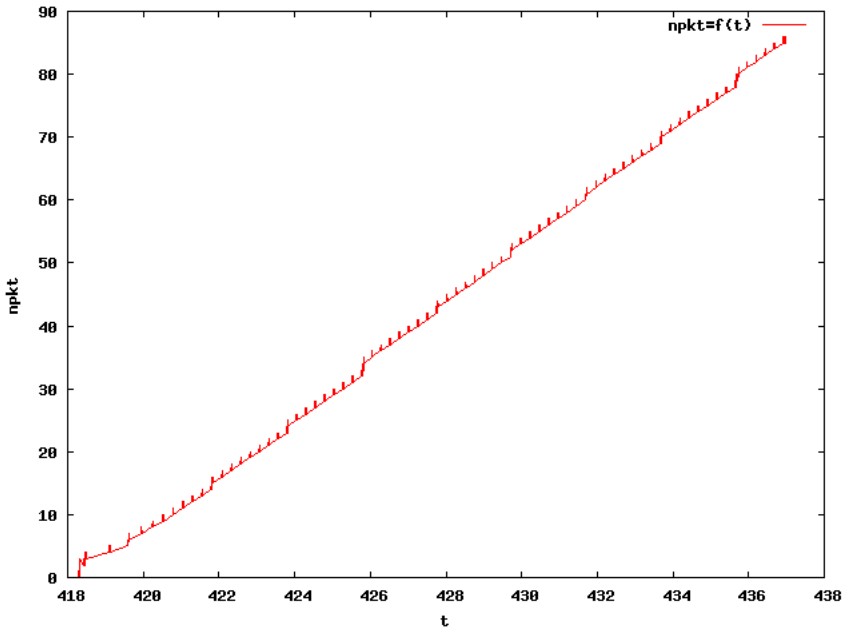

FIG. 8 – Après réglage avec tc(1)

On pourrait très bien décrémenter Qsize à chaque paquet droppé : cela permettrait de réinitialiser le compteur plutôt que de le laisser grandir à chaque paquet non-envoyé, et cela resterait logique, dans le sens où le paquet n'est plus dans la file d'attente.

Sans effectuer de réglages, les paquets ne sont pas droppés, et, selon la configuration de la carte, nous observons soit un Qsize toujours à 0, soit un Qsize grandissant à l'occasion, mais se réduisant par la suite :

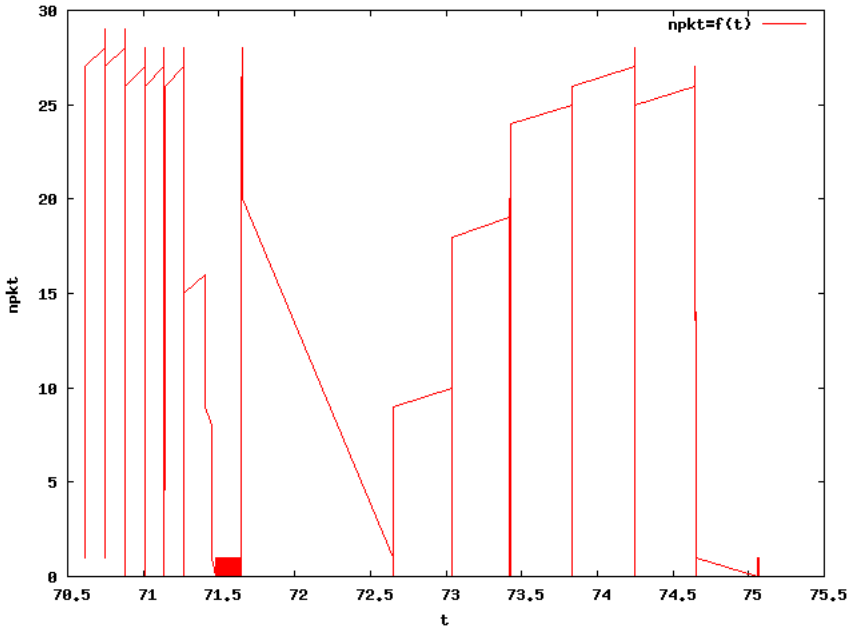

FIG. 9 – Sans réglage

7 Conclusion

Durant ce projet, nous avons fourni une analyse de la taille de la file d'attente réelle instantanée au sein d'un routeur ERN. Pour y parvenir, nous avons exploré différentes solutions avant d'aboutir à la solution finale utilisant la couche Liaison. Nos recherches nous ont fait appréhender un domaine jusqu'ici inconnu, celui de la recherche. Elles nous ont également permis de développer nos connaissances en réseau, en programmation en mode noyau et en virtualisation. Nos résultats sont exploitables par les chercheurs. Néanmoins, une statistique telle que l'Input Traffic Rate devra être établie pour compléter le développement du framework de notre routeur. Pour obtenir la vitesse, en sachant que l'on a déjà le nombre de paquets entrants, il faudrait utiliser un timer.

A Compléments sur la programmation de modules Linux

A.1 Organisation générale

Un module contient toujours deux fonctions respectivement exécutées au chargement et au déchargement du module :

```
int
init_module(void)
{
    ...
}

void
cleanup_module(void)
{
    ...
}
```

Les fonctions C « classiques » ne sont plus disponibles, et des équivalents en mode noyau viennent les remplacer : par exemple, *printf()* est remplacée par *printk()*.

Pour charger/décharger un noyau, on peut utiliser au choix *insmod(8)* et *rmmod(8)* ou encore *modprobe(8)* pour charger et *modprobe -r* pour décharger.

A.2 Mise en place d'un hook Netfilter

Un hook pour Netfilter est une fonction

```
unsigned int fun(unsigned int, struct sk_buff *, const struct net_device *,
const struct net_device *, int (*)(struct sk_buff *));
```

Elle est appelée dans un contexte d'interruptions, à chaque fois qu'un paquet passe par le hook correspondant. Sont passés en arguments :

struct sk_buff *skb le paquet ;

const struct net_device *in, *out les interfaces d'entrées/sorties ;

Typiquement, l'enregistrement du module se fait dans la fonction *init_module()* :

```
static struct nf_hook_ops nfho;

...

int
init_module(void)
{
    nfho.hook     = fun;             /* handler function */
    nfho.hooknum  = NF_INET_FORWARD; /* emplacement du hook */
    nfho.pf       = PF_INET;
    nfho.priority = NF_IP_PRI_FIRST;   /* priorité */

    nf_register_hook(&nfho);
}
```

et le déchargement dans *cleanup_module()* :

```
void cleanup_module()
{
    nf_unregister_hook(&nfho);
}
```

A.3 Synchronisation (spinlock)

Les spinlocks sont un mécanisme de synchronisation utilisable dans un contexte d'interruptions, comme c'est le cas pour les hooks Netfilter. Nous nous sommes ici contentés de les utiliser comme de simples verrous, ce que le code suivant résume :

```
#include <linux/spinlock.h>
static DEFINE_SPINLOCK(lock);

{
    spin_lock_irq(&lock);    /* verouiller */
    /* code protégé */
    spin_unlock_irq(&lock);  /* dévérouiller */
}
```

D'autres mécanismes comme les sémaphores ou les mutex sont utilisables dans des cas moins particuliers.

A.4 Module Netfilter

```
/* Netfilter module for computing Qsize */

#ifndef __KERNEL__
        #define __KERNEL__
#endif
#ifndef MODULE
        #define MODULE
#endif

#include <linux/version.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/init.h>
#include <linux/skbuff.h>
#include <linux/tcp.h>
#include <asm/io.h>
#include <linux/inet.h>
#include <linux/vmalloc.h>
#include <net/ip.h>

#include <net/route.h>

#include <linux/spinlock.h>

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,16)
#include <linux/in.h>
#include <linux/ip.h>
#endif

#define NB_INTFS 3
#define NB_STATS 4

/* Args */
static int outputRate_eth1 = 1000;
```

```
static int outputRate_eth2 = 10;
module_param(outputRate_eth1, int, 0);
module_param(outputRate_eth2, int, 0);

static DEFINE_SPINLOCK(lock);

static int stats_packets[NB_INTFS][NB_STATS] = { { 0 } };

/* This is the structure we shall use to register our function */
static struct nf_hook_ops nfho_forward;
static struct nf_hook_ops nfho_post;
static struct nf_hook_ops nfho_pre;

/* Name of the interface we want to drop packets from */
struct sk_buff *sock_buff;
struct iphdr *ip_header;
struct tcphdr *tcp_header;

inline int hash_index(const char* name_intf) {
    return name_intf[strlen(name_intf) - 1] - '0';
}

/* Hook function: FORWARD */
unsigned int hook_func_forward(unsigned int hooknum, struct sk_buff *skb, const struct net_device *i

    int index;
    sock_buff = skb; // Ok, useless but ...

    if (!sock_buff) {
        return NF_ACCEPT;
    }
    ip_header = (struct iphdr *) skb_network_header (sock_buff);
    if (!ip_header) {
        return NF_ACCEPT;
    }
    if (ip_header->protocol != 6) {
        return NF_ACCEPT;
    }
    tcp_header = (struct tcphdr *) skb_transport_header (sock_buff);
    if (!tcp_header) { return NF_ACCEPT; }

    return NF_ACCEPT;
}

/* Hook function: POST */
unsigned int hook_func_post(unsigned int hooknum, struct sk_buff *skb, const struct net_device *in,

    int index;
    sock_buff = skb; // Ok, useless but ...

    if (!sock_buff) {
        return NF_ACCEPT; }
    ip_header = (struct iphdr *) skb_network_header (sock_buff);
    if (!ip_header) {
        return NF_ACCEPT;
    }
```

```
    if (ip_header->protocol != 6) {
        return NF_ACCEPT;
    }
    tcp_header = (struct tcphdr *) skb_transport_header (sock_buff);
    if (!tcp_header) {
        return NF_ACCEPT;
    }

    printk(KERN_CRIT "POST");
    printk(KERN_CRIT "PostIn: %s | PostOut: %s", in->name, out->name);

    index = hash_index(out->name);

    spin_lock_irq(&lock);
    stats_packets[index][1]++; // O++
    stats_packets[index][2] = stats_packets[index][0] - stats_packets[index][1]; // qsize = I - O
    printk(KERN_CRIT "OutPkt: %d\n", stats_packets[index][1]);
    printk(KERN_CRIT "Qsize: %d\n", stats_packets[index][2]);
    spin_unlock_irq(&lock);

    return NF_ACCEPT;
}

/* Hook function: PRE */
unsigned int hook_func_pre(unsigned int hooknum, struct sk_buff *skb, const struct net_device *in, c

    int rc = 51;
    int index = 42;
    __be32 src, dst;
    u8 tos;

    // Ok, useless but ...
    sock_buff = skb;

    if (!sock_buff) {
        return NF_ACCEPT;
    }
    ip_header = (struct iphdr *) skb_network_header (sock_buff);
    if (!(ip_header)) {
        return NF_ACCEPT;
    }
    if (ip_header->protocol != 6) {
        return NF_ACCEPT;
    }
    tcp_header = (struct tcphdr *) skb_transport_header (sock_buff);
    if (!tcp_header) {
        return NF_ACCEPT;
    }

    printk(KERN_CRIT "\nPRE");
    printk(KERN_CRIT "PreIn: %s | PreOut: %s", in->name, out->name);

    // Tests routage propre
    src = ip_header->saddr;
    dst = ip_header->daddr;
    tos = ip_header->tos;
```

```
    rc = ip_route_input(skb, dst, src, tos, in);
    printk(KERN_CRIT "Route: %d", rc);

    if (ip_header->daddr == 29935816) { // Target: 200.200.200.1 via eth2
        index = hash_index("eth2");
    } else if (ip_header->daddr == 23356516) { // Target: 100.100.100.1 via eth1
        index = hash_index("eth1");
    }
    if (index == 42) {
        return NF_ACCEPT;
    }

    spin_lock_irq(&lock);
    stats_packets[index][0]++;
    //stats_packets[index][2] = stats_packets[index][0] - stats_packets[index][1]; // qsize = I - O
    printk(KERN_CRIT "InputPkts: %d\n", stats_packets[index][0]);
    //printk(KERN_CRIT "Qsize: %d\n", stats_packets[index][2]);
    spin_unlock_irq(&lock);

    return NF_ACCEPT;
}

/* Initialisation routine */
int init_module()
{

    /* Fill in our hook structure */
    nfho_forward.hook     = hook_func_forward;         /* Handler function */
    nfho_forward.hooknum  = NF_INET_FORWARD; /* NF_IP_FORWARD */
    nfho_forward.pf       = PF_INET;
    nfho_forward.priority = NF_IP_PRI_FIRST;   /* Make our function first */

    /* Fill in our hook structure */
    nfho_post.hook     = hook_func_post;         /* Handler function */
    nfho_post.hooknum  = NF_INET_POST_ROUTING; /* NF_IP_FORWARD */
    nfho_post.pf       = PF_INET;
    nfho_post.priority = NF_IP_PRI_FIRST;   /* Make our function first */

    /* Fill in our hook structure */
    nfho_pre.hook     = hook_func_pre;         /* Handler function */
    nfho_pre.hooknum  = NF_INET_PRE_ROUTING; /* NF_IP_FORWARD */
    nfho_pre.pf       = PF_INET;
    nfho_pre.priority = NF_IP_PRI_FIRST;   /* Make our function first */

    printk(KERN_CRIT "Go ahead\n");
    nf_register_hook(&nfho_forward);
    nf_register_hook(&nfho_post);
    nf_register_hook(&nfho_pre);

    //stats_packets[1][1] = outputRate_eth1;
    //stats_packets[2][1] = outputRate_eth2;

    return 0;
}
/* Cleanup routine */
void cleanup_module()
```

```
{
    nf_unregister_hook(&nfho_forward);
    nf_unregister_hook(&nfho_post);
    nf_unregister_hook(&nfho_pre);
}

MODULE_LICENSE("GPL");
```

A.5 Fonctions modifiées du noyau

```
--- /usr/src/linux/net/core/dev.c.orig    2012-02-03 10:57:27.000000000 +0100
+++ /usr/src/linux/net/core/dev.c    2012-02-03 10:59:16.000000000 +0100
@@ -135,6 +135,13 @@

 #include "net-sysfs.h"

+#include <linux/spinlock.h>
+
+static DEFINE_SPINLOCK(lock);
+
+static int npkt = 0;
+
+
 /* Instead of increasing this, you should create a hash table. */
 #define MAX_GRO_SKBS 8

@@ -2001,6 +2008,7 @@
 {
     const struct net_device_ops *ops = dev->netdev_ops;
     int rc = NETDEV_TX_OK;
+    struct iphdr *iph;

     if (likely(!skb->next)) {
         if (!list_empty(&ptype_all))
@@ -2047,10 +2055,22 @@
             }
         }

+        printk(KERN_CRIT "\t (NON GSO)\n");
         rc = ops->ndo_start_xmit(skb, dev);
         trace_net_dev_xmit(skb, rc);
-        if (rc == NETDEV_TX_OK)
+        if (rc == NETDEV_TX_OK) {
+            iph = (struct iphdr *)skb_network_header(skb);
+            if (iph->protocol == 6) {
+                spin_lock_irq(&lock);
+                printk(KERN_CRIT "OUT: %d\n", npkt--);
+                spin_unlock_irq(&lock);
+            }
+            printk(KERN_CRIT "\t\ttransmission ok!\n");
             txq_trans_update(txq);
+        }
+        else {
+            printk(KERN_CRIT "rc: %d\n", rc);
+        }
```

```
         return rc;
     }

@@ -2068,9 +2088,11 @@
         if (dev->priv_flags & IFF_XMIT_DST_RELEASE)
             skb_dst_drop(nskb);

+        printk(KERN_CRIT "\t (GSO)\n");
         rc = ops->ndo_start_xmit(nskb, dev);
         trace_net_dev_xmit(nskb, rc);
         if (unlikely(rc != NETDEV_TX_OK)) {
+            printk(KERN_CRIT "\t\ttransmission ok!\n");
             if (rc & ~NETDEV_TX_MASK)
                 goto out_kfree_gso_skb;
             nskb->next = skb->next;
@@ -2250,6 +2272,7 @@
     struct netdev_queue *txq;
     struct Qdisc *q;
     int rc = -ENOMEM;
+    struct iphdr *iph;

     /* Disable soft irqs for various locks below. Also
      * stops preemption for RCU.
@@ -2262,8 +2285,17 @@
 #ifdef CONFIG_NET_CLS_ACT
     skb->tc_verd = SET_TC_AT(skb->tc_verd, AT_EGRESS);
 #endif
+
+    iph = (struct iphdr *)skb_network_header(skb);
+    if (iph->protocol == 6) {
+        spin_lock_irq(&lock);
+        printk(KERN_CRIT "IN %d\n", npkt++);
+        spin_unlock_irq(&lock);
+    }
+
     trace_net_dev_queue(skb);
     if (q->enqueue) {
+        printk(KERN_CRIT "device has queue! __dev_xmit_skb()");
         rc = __dev_xmit_skb(skb, q, dev, txq);
         goto out;
     }
@@ -2281,6 +2313,7 @@
        Either shot noqueue qdisc, it is even simpler 8)
      */
     if (dev->flags & IFF_UP) {
+        printk(KERN_CRIT "device has no queue!");
         int cpu = smp_processor_id(); /* ok because BHs are off */

         if (txq->xmit_lock_owner != cpu) {
@@ -2292,6 +2325,7 @@

             if (!netif_tx_queue_stopped(txq)) {
                 __this_cpu_inc(xmit_recursion);
+                printk(KERN_CRIT "\tcalling dev_hard_start_xmit()\n");
                 rc = dev_hard_start_xmit(skb, dev, txq);
                 __this_cpu_dec(xmit_recursion);
```

```
                if (dev_xmit_complete(rc)) {
@@ -2320,6 +2354,12 @@
     kfree_skb(skb);
     return rc;
 out:
+    if (rc == NET_XMIT_DROP) {
+        printk(KERN_CRIT "packet dropped!\n");
+    }
+    if (rc < 0) {
+        printk(KERN_CRIT "rc < 0 in dev_queue_xmit");
+    }
     rcu_read_unlock_bh();
     return rc;
 }
```

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] S. Jain, Y. Zhang, and D. Loguinov, "A generic traffic conditioningmodel for differentiated services."

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

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

[5] `https://www.virtualbox.org/`. VirtualBox.

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

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

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

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

[10] `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem/`. Netem, The Linux Foundation.

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

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

[13] L. Lin, T. Jiang, and J. Lo, "Towards experimental evaluation of explicit congestion control."

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

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

[16] `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/