HORS SÉRIE - HORS SÉRIE - HORS SÉRIE - HORS SÉRIE - HORS SÉRIE

GNU LINUX MAGAZINE / FRANCE

France Métro : 6,40€ - DOM 6,95€ - BEL : 7,30€ - LUX : 7,30€ - PORT. CONT. : 7,30€ - CH : 13FS - CAN : 12$ - MAR : 65DH

L 15066 - 29 H - F: 6,40 € - RD

Mars / Avril 2007

HORS SÉRIE N°29

BSD ACTE I

Découvrez des systèmes en Logiciel libre qui n'ont rien à envier à GNU/Linux.

UTILISATION DES PORTS FREEBSD

Maîtrisez l'installation/désinstallation d'applicatifs et la mise à jour du système au travers d'outils comme CVSup, Csup ou Portsnap.

GESTION DES VOLUMES LOGIQUES

Découvrez GEOM pour gérer vos unités de stockage, faire du RAID1, créer des volumes réseau, chiffrer les données ou encore monter des grappes (RAID5).

FIREWALL, FILTRAGE, QOS

Oubliez Netfilter/iptables et faites connaissance avec la simplicité, la richesse et les fonctionnalités du PacketFilter (PF) des *BSD.

DÉVELOPPEMENT NOYAU

Partez à la découverte du développement kernel pour FreeBSD et créez votre premier module.

PROGRAMMATION WIFI SOUS NETBSD

Développez des codes accédant au matériel 802.11 depuis l'espace utilisateur.

OPENBSD ET IPSEC

Configurez et déployez un VPN IPSec en toute simplicité.

LOAD BALANCING ET HAUTE DISPONIBILITÉ

Utilisez PF et CARP pour la mise en œuvre de solutions de répartition de charge en HA.

ROUTAGE ET HAUTE DISPONIBILITÉ

Initiez-vous au routage OSPF/BGP avec OpenBSD.

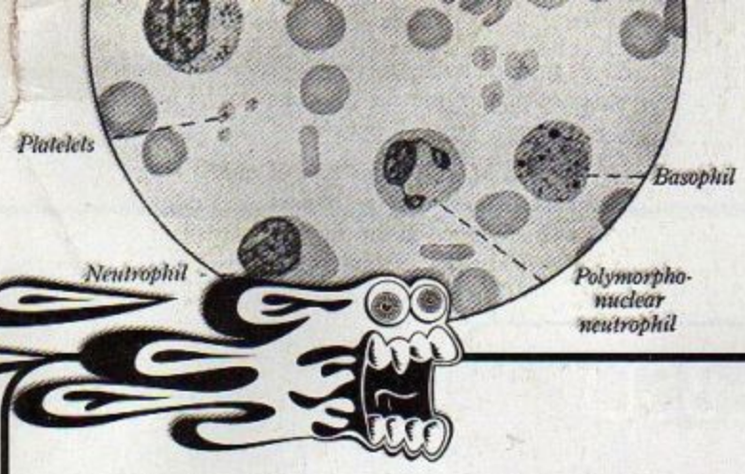
FreeBSD

OpenBSD

NetBSD®

GRUIK CODERS UNITED 1996 · 2006

Administration et développement sur systèmes UNIX



Hérésie ! Horreur ! Malédiction !

Vous ne rêvez pas ! Vous tenez dans les mains un numéro hors série de **GNU/Linux** *Magazine France* consacré aux ***BSD**. Cachez ce magazine et protégez-le des regards indiscrets ! Il s'agit sans doute du premier artefact impie de l'histoire du Logiciel libre. Ses origines sont obscures, mais nous connaissons ceux qui ont contribué à sa création. Il s'agit d'un groupe de cabalistes et autres sorcières (oui, un homme comme une femme peut être une sorcière), connu sous l'acronyme GCU. Ce groupe est également appelé tantôt « la honte du Logiciel libre » (nous ne savons pas si un dépôt de marque est en cours), par certaines formes de vie que je ne prendrai pas la peine de qualifier ou classifier ici.

Quoi qu'il en soit, dans les pages qui vont suivre et sur un ton léger très caractéristique, vous trouverez des explications d'une grande technicité dispensées par des auteurs qualifiés. Profitons-en pour préciser le niveau requis pour la lecture de ce qui suit : ce contenu est destiné principalement aux utilisateurs GNU/Linux ou éventuellement d'autres UNIX. « Utilisateur » dans le sens « qui utilise et exploite », et non « qui a installé et qui clique ». Vous ne trouverez donc pas d'explications superflues sur l'utilisation de base d'un système de type UNIX, d'un *shell* ou des outils de base. Tout ceci doit être connu et maîtrisé.

Ah oui ! Un dernier petit conseil : une malédiction est associée à ce manuscrit. Sous certaines conditions, la lecture des pages dans un ordre donné provoque un état d'hypnose. État durant lequel vous risquez d'invoquer des démons (en baskets, par exemple) et procéder à d'autres rites étranges (comme vous envelopper dans une serviette orange pour faire... des choses). Il est également possible que vous installiez un FreeBSD, OpenBSD ou NetBSD. La rédaction ne saurait être tenue responsable de ces effets.

Enfin, il se peut également qu'une envie subite de communiquer avec d'autres personnes vous tenaille. Sans vous en rendre compte (à ce stade, il est fort probable que vous soyez déjà possédé), il est possible que vous arriviez sur le canal `#gcu` du réseau IRC Freenode. Si tel est le cas, utilisez le peu de raison qui vous reste pour visiter **http://wiki.gcu.info/doku.php?id=de_la_bienseance_sur_le_canal_irc**.

Sur ce, je vous laisse profiter de ces 80 pages de pur bonheur, non sans vous avoir annoncé qu'une suite est d'ores et déjà planifiée et prendra la forme d'un autre hors-série *BSD.

Denis Bodor

PS : Je tiens à remercier, très sérieusement et très officiellement, toute l'équipe GCU-Squad pour le sérieux et la qualité du travail effectué (organisation, rédaction, relecture).

SOMMAIRE:

USER

SÉCURITÉ

ADMINISTRATION

DÉVELOPPEMENT

Linux Magazine France Hors Série
est édité par **Diamond Editions**
B.P. 20142 - 67603 Sélestat Cedex

Tél. : 03 88 58 02 08
Fax : 03 88 58 02 09

E-mail :
cial@ed-diamond.com

Service commercial :
abo@ed-diamond.com

Site :
www.ed-diamond.com

Directeur de publication :
Arnaud Metzler

PRINTED IN Germany / Imprimé en allemagne / Dépôt légal : 3e Trimestre 1998 / N° ISSN : 1291-78 34 / Commission Paritaire : 09 03 K78 976 / Périodicité : Bimestrielle / Prix de vente : 6,40 Euros

RÉDACTION

Rédacteur en chef :
Denis Bodor

Conception graphique :
Kathrin Troeger

Responsable publicité :
Véronique Wilhelm
Tél. : 03 88 58 02 08

Service abonnement :
Tél. : 03 88 58 02 08

Relecture :
Dominique Grosse

Impression : VPM DRUCK / www.vpm-druck.de

Distribution France :
(uniquement pour les dépositaires de presse)

MLP Réassort :
Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04

Service des ventes : Distri-médias :
Tél. : 05 61 72 76 24

nico, hr, mat, twisla

CE QUI M'A DÉROUTÉ SOUS FREEBSD LA PREMIÈRE FOIS

1. L'OS EN LUI-MÊME

FreeBSD est un *Operating System* complet. FreeBSD n'est pas un noyau agrémenté d'une collection de logiciels plus ou moins empaquetés autour, empaquetage fait par qui veut bien selon sa logique plus ou moins tordue et ses outils plus ou moins efficaces. FreeBSD est beau. FreeBSD est grand. FreeBSD dispose de sa propre `libc`. FreeBSD possède ses propres outils UNIX comme `sed`, `awk`, `vi`, `cut`, `tr`, `ed`, `sh`. FreeBSD n'utilise pas les *autotools*. FreeBSD utilise `make`(1). FreeBSD te rendra soyeux comme un pull en poils de bouquetins. FreeBSD est composé d'un noyau et d'un système de base, appelé dorénavant « *basesystem* ». FreeBSD est cohérent dans son approche d'UNIX. FreeBSD a le même comportement quelle que soit la version de FreeBSD que tu utiliseras. FreeBSD est libre. FreeBSD est sécurisé. FreeBSD est taillé pour le réseau. FreeBSD est facile à installer. FreeBSD est multitâche, *multi-thread*, multi-CPU. FreeBSD est compatible POSIX. Amen.

2. LE FS (SLICES + PARTITIONS)

Le premier contact avec un OS se fait généralement à l'installation. Et là, FreeBSD diffère de GNU/Linux sur quelques points, notamment la gestion des systèmes de fichiers. En effet, FreeBSD dispose d'une « couche » en plus, qui permet de partitionner plus finement un disque. Cette couche s'appelle un *slice*, et on utilise généralement un slice qui couvre tout le disque. Néanmoins, on peut être assez dérouté par le nommage entraîné par les slices. Un slice est une partition MS-DOS encore appelée « partition primaire ». C'est une spécificité des plateformes compatibles PC.

Exemple :

```
/dev/da0s1a   738M    45M   634M     7%    /
/dev/da0s1f   984M   110M   795M    12%    /tmp
/dev/da0s1g    29G    10G    16G    39%    /usr
/dev/da0s1e   984M   703M   202M    78%    /var
/dev/da0s1h    35G    29G   3.3G    90%    /data/zone1
/dev/da1s1e   275G   138G   115G    54%    /data/zone2
```

Décortiquons cela ensemble. Une entrée concernant une partition sous FreeBSD aura cette allure : `/dev/daXsYz`

- `da` : nom d'un disque SCSI standard (`man 4 da`). Dans le cas d'un disque IDE, on aurait `ad` à la place (`man 4 ad`).
- `X` : numéro du périphérique dans l'ordre du BIOS.
- `Y` : numéro du slice.
- `z` : lettre de la partition.

Les slices sont numérotés dans l'ordre des partitions DOS MBR présentes sur le disque. Il est à noter que FreeBSD ne reconnaît que les partitions primaires. Lorsque `z` est la lettre « c », cela représente le slice où FreeBSD est installé.

On peut également noter que les slices sont une spécificité du monde x86 (eh oui, sur SPARC, il n'y a pas de partitions DOS) et que si le seul OS est FreeBSD, on n'a même pas besoin de slices.

3. LE NOM DES PÉRIPHÉRIQUES (FXP/EM/TOUSSA VS ETH)

Autre point déroutant pour une personne jouant avec le réseau : les noms des périphériques. Sous Linux, les cartes réseau « filaires » s'appellent `ethX` quelle que soit la carte. Sous BSD, le nom du périphérique est lié au driver : `fxp` pour les cartes Intel, `rl` pour les Realtek, `ste` pour les DLink... Assez perturbant au premier abord, ce système permet de savoir plus facilement sur quelle carte réseau (physique) on agit. Lié au `ifconfig` de BSD (cf. plus bas), cela se révèle un outil de diagnostic parfois salvateur (essayez sur une machine avec 6 interfaces réseau et vous serez convaincu).

4. LA SÉPARATION NETTE SYSTÈME DE BASE/PORTS

La séparation assez nette entre le système de base et les ports a une conséquence qui peut se révéler assez consommatrice de temps. Sous GNU/Linux, les configurations sont (dans 95% des cas) placées dans `/etc`, or FreeBSD possède 2 endroits pour stocker les configurations : `/etc` pour ce qui concerne le système de base (`/etc/ssh` par exemple) et `/usr/local/etc` pour ce qui concerne les ports (`/usr/local/etc/postfix` par exemple). Il en va de même pour les scripts `rc` avec `/etc/rc.d` et `/usr/local/etc/rc.d`.

5. LES PORTS/SYSTÈMES DE PAQUETAGE

Pour installer un logiciel sous FreeBSD, on utilise un système appelé « port ». De nombreux systèmes pour les exploiter existent (utilisation de base, `portupgrade`, `portmaster`) et permettent de choisir plus ou moins facilement entre les différentes versions des logiciels. De même, compiler ses programmes permet de choisir certaines options et d'en abandonner d'autres (les utilisateurs de Gentoo ne seront pas trop perturbés, puisque le système qu'ils utilisent est une adaptation des ports BSD). Mais cela se révèle parfois à double tranchant (un serveur LDAP qui ne démarre plus parce qu'on a activé une option liée aux threads et vous met votre authentification en l'air est un bon exemple). Mais les barbus ont pensé à ça, et les informations concernant les mises à jour se trouvent dans `/usr/ports/UPDATING` qu'il faut systématiquement consulter avant toute mise à jour.

6. LES VERSIONS

FreeBSD dispose de deux principaux axes de développement liés à son architecture : le basesystem et les ports. Chacun possède des versions, marquant une étape dans le développement.

Le système de base se voit ainsi attribuer 3 tags intéressants (pour nous) : RELENG_X_Y, RELENG_X et HEAD. RELENG_X_Y est fait pour les gens qui désirent avoir une version précise de FreeBSD avec les patches de sécurité inclus et les correctifs affectant la stabilité du système. `X` désigne le numéro de version majeure et `Y` le numéro de version mineure.

Exemple : RELENG_5_4 va installer la version 5.4 de FreeBSD ainsi que tous les patches de sécurité, ce qui donnera un 5.4-RELEASE-p11. Prendre un tag RELENG_X_Y est l'assurance d'avoir un système stable et surtout dénué de difficultés de mises à jour.

RELENG_6 va vous installer la version 6.x de FreeBSD la plus récente. C'est la branche appelée « STABLE », et lorsque le développement de FreeBSD fait que FreeBSD passe d'une version mineure à une autre, vous y allez aussi. Les auteurs de cet article utilisent STABLE sur des machines en production et n'ont pas rencontré de problèmes lors des mises à jour. Bien entendu, -STABLE bénéficie aussi des mises à jour de sécurité.

HEAD enfin, est la branche de développement de FreeBSD, aussi appelée « CURRENT ». C'est bien entendu la branche la plus instable, étant donné que c'est elle qui contient toutes les dernières nouveautés du développement. Cette branche est déconseillée pour de la production.

Passons aux ports. Les ports possèdent aussi des tags qui, à vrai dire, ne servent que pour identifier les versions ayant servies à générer tous les *packages* fournis avec une *release* officielle de FreeBSD, et qui ne sont jamais mis à jour. On ne va donc s'intéresser qu'à HEAD (alias CURRENT alias « . ») qui sert au quotidien. CURRENT intègre toutes les dernières modifications sur les ports que les développeurs de FreeBSD auront faites, ainsi que les correctifs de sécurité. Il est plus que recommandé de lire `/usr/ports/UPDATING` avant de faire une mise à jour d'un port. Par exemple, le port d'OpenLDAP a récemment subi une transformation et utilise maintenant BerkeleyDB en version 4.4 pour son *backend* au lieu de la version 4.3. Il est indiqué dans `/usr/ports/UPDATING` comment faire une sauvegarde, puis une restauration de sa base de données dans le cas où quelque chose se passerait mal pendant la mise à jour.

7. LA MISE À JOUR

Justement, la mise à jour. Il faut avant tout mettre à jour les sources du système. On peut voir cette opération comme l'équivalent du `apt-get update` de Debian, tandis que la mise à jour en elle même sera le `apt-get dist-upgrade`, qui se fera via recompilation dans `/usr/src`, ou utilisation de `sysinstall`(8) pour l'*upgrade* binaire du basesystem. La page de manuel `build`(7) de FreeBSD récapitule bien la procédure de mise à jour (recompilation du basesystem). La mise à jour des sources se fait généralement via l'outil `cvsup`, mais on lui préfèrera son successeur `csup` (écrit en C et inclus dans le basesystem depuis peu de temps), ainsi que `portsnap`(8) pour les ports. Il est également recommandé de lire `/usr/src/UPDATING` avant de tenter quoi que ce soit.

La mise à jour binaire et/ou source des ports se fait, elle, via un outil tiers comme `portupgrade`(1) ou `portmaster`(8). Il faut bien entendu avoir mis à jour au préalable les sources des ports. Voir pour tout ceci l'article traitant de ce sujet dans le présent magazine.

8. LES DIFFÉRENCES

8.1 LA PHILOSOPHIE TRÈS CARRÉE

FreeBSD, c'est un peu comme un sergent instructeur de film américain. C'est carré, bien rasé et ça a un t-shirt près du corps. Quand on installe un port, il faut spécifier (via la modification de `/etc/rc.conf`) si le logiciel doit être lancé au démarrage. Le système ne le fera pas pour nous. Pas de scripts de *post-install* qui vont tout configurer pour nous. Par contre, une documentation efficace est systématiquement fournie. Les exemples sont commentés, et généralement concrets (`/usr/local/share/doc` et `/usr/local/share/examples` pour les ports : enlever le `local` pour le basesystem).

8.2 LE SYSTÈME D'INIT

Le système d'init des BSD est quelque chose de complètement différent de l'init des SystemV.

Nous allons rapidement nous rappeler comment démarre un Unix System V, puis nous allons ensuite voir quelles sont les différences.

8.2.1 L'init System V

Sous Linux (qui est un clone d'UNIX System V), comme vous le savez, `/sbin/init` est le premier processus lancé par le noyau à la fin de sa phase de boot. Il se met alors à lire le fichier `/etc/inittab`, et démarre des *daemons*, des *process* de *login*, s'occupe de mettre la machine en réseau et effectue quelques tâches secondaires.

Voici un exemple d'`inittab`, que nous allons commenter ensemble.

```
id:2:initdefault:
```

Cette ligne indique que le *runlevel* par défaut est le runlevel 2. Sous Debian (la distribution dont nous avons extrait l'`inittab`), c'est le niveau d'exécution qui correspond au multi-utilisateur avec réseau, mais sans X. Le runlevel considéré est toujours placé dans le deuxième champ.

```
si::sysinit:/etc/init.d/rcS
```

Cette ligne indique que le script `rcS` est lancé une fois au démarrage de la machine. On voit qu'il n'y a pas de numéro de runlevel, pour la bonne raison qu'il est inutile ici.

```
l0:0:wait:/etc/init.d/rc 0
l1:1:wait:/etc/init.d/rc 1
l2:2:wait:/etc/init.d/rc 2
l3:3:wait:/etc/init.d/rc 3
l4:4:wait:/etc/init.d/rc 4
l5:5:wait:/etc/init.d/rc 5
l6:6:wait:/etc/init.d/rc 6
# Normally not reached, but fallthrough
# in case of emergency.
z6:6:respawn:/sbin/sulogin
```

Ces lignes indiquent quelle action effectuer suivant le runlevel désiré. On voit que c'est le script `/etc/init.d/rc` qui est appelé avec l'argument `n`, égal au runlevel désiré. `/etc/init.d/rc` s'occupe de lancer tous les daemons comme décrit plus bas.

```
1:2345:respawn:/sbin/getty 38400 tty1
2:23:respawn:/sbin/getty 38400 tty2
3:23:respawn:/sbin/getty 38400 tty3
```

On voit ici qu'init s'occupe également de la mise en place des terminaux virtuels. Chaque script de gestion de daemon est placé dans `/etc/rc.d/init.d`. Pour savoir si le daemon doit être lancé au démarrage de la machine ou pas, des liens symboliques sont créés dans `/etc/rcX.d/YnnDaemon.sh`, où X est le runlevel, Y la lettre « S » ou « K », et `nn` la priorité.

Il y a plusieurs niveaux d'exécution (appelés « runlevels ») dans un Unix SystemV, et le niveau par défaut est indiqué dans une des lignes du fichier `inittab`. X représente ce niveau. En général, 0 représente l'arrêt de la machine, 1 est le mode mono-utilisateur, 2 est le mode multi-utilisateur (sans réseau sous les RedHat-like, avec sous les Debian-like), 3 et 4 ne sont pas toujours utilisés, 5 est le mode multi-utilisateur graphique en réseau et 6 est le reboot. Ensuite, vient la lettre qui détermine si le service doit être lancé (« S », comme Start) ou arrêté (« K », comme Kill). Le numéro qui vient ensuite est un numéro codé sur deux chiffres, qui détermine l'ordre dans lequel les services seront lancés (S20exim4 sera par exemple lancé avant S59ssh).

On voit déjà là que c'est un sacré bordel à gérer. Des distributions proposent des scripts pour améliorer cela, comme RedHat avec « service », ou Debian avec `update-rc.d`(8). Malheureusement, ce ne sont pas des outils unifiés, que l'on peut retrouver sur n'importe quelle distribution. D'autre part, toutes les distributions n'utilisent pas le même système d'init, les scripts ne sont pas tous à la même place, etc. La seule chose que l'on est sûr de retrouver n'importe où est `ln`(1).

8.2.2 L'init BSD

Sous BSD, la phase de démarrage est beaucoup plus simple. Il y a deux sous-types de démarrage : la vénérable école, représentée par OpenBSD, et la nouvelle école, fondée par NetBSD, puis rejointe par FreeBSD et DragonFlyBSD.

Sous OpenBSD, `init`(8) se contente de lancer le script `/etc/rc`, qui s'occupe de tout initialiser. Il prend sa configuration dans le fichier `/etc/rc.conf` et exécute optionnellement `/etc/rc.local` (pour des ajouts locaux à la machine). Ce grand script monolithique est lancé en cas de boot multi-utilisateur. Ceci est magnifiquement expliqué dans les splendides pages de manuel OpenBSD, que vous retrouverez en ligne sur le site **http://www.openbsd.org/cgi-bin/man.cgi?query=init&sektion=8**.

NetBSD et ses disciples ont un peu pris le meilleur de chaque monde (la simplicité de l'init BSD et la souplesse de l'init SystemV), sans les défauts de chaque (la rigidité de l'init BSD, la porcherie de l'init SystemV). Sous NetBSD, `/etc/rc` n'est plus un script monolithique comme sous NetBSD, mais est un tout petit morceau de script qui va s'occuper de lancer les sous-scripts présents dans `/etc/rc.d`. Vous allez nous dire que cela ressemble nettement à un init SysV, mais point du tout ! Les choses reprochables à l'init SysV ont disparu ! Le bricolage avec les liens symboliques a disparu, car la configuration se fait bêtement dans un fichier texte `/etc/rc.conf` et indique si un service doit être lancé ou pas.

```
user@netbsd % grep ssh /etc/rc.conf
sshd=YES
user@netbsd % grep sendmail /etc/rc.conf
sendmail=NO
```

D'autre part, la gestion de l'ordre de lancement des services est effectuée par un système s'appelant `rcorder`(8).

```
# PROVIDE: named
# REQUIRE: SERVERS
# BEFORE:  DAEMON
# KEYWORD: chrootdir
```

Cet exemple, extrait de /etc/rc.d/named, nous montre que le service named(8) dépend des « SERVERS », fournit le service named, doit être lancé avant les DAEMON et utilise chrootdir.

Pour voir dans quel ordre les daemons peuvent être lancés, vous pouvez utiliser la commande /sbin/rcorder -s nostart /etc/rc.d/*. Nous indiquons « peuvent être lancés », car ils se lanceront si et seulement si nous l'avons indiqué dans /etc/rc.conf. Précisons aussi que c'est **toujours** le fichier /etc/rc.conf qui doit être édité afin de déterminer si un service se lance ou pas, que le service soit un service présent dans le basesystem ou dans les ports. Il reste à noter que sous OpenBSD, il faut éditer le fichier /etc/rc.conf.local et non pas le fichier /etc/rc.conf, qui, lui, contient les réglages par défaut (et est susceptible d'être écrasé lors d'une mise à jour). Le fichier contenant les réglages par défaut des daemons est /etc/default/rc.conf sous NetBSD et FreeBSD.

8.3 LES OUTILS QUI CHANGENT (ROUTE/NETSTAT, IPTABLES/PF)

Le changement le plus flagrant entre les systèmes Linux et BSD se situe au niveau de ce qu'on utilise le plus : les programmes d'administration. En effet, certaines commandes existent toujours, mais n'ont plus la même syntaxe ou le même usage. Assez perturbant par exemple, l'affichage de la table de routage du noyau : là où, sous GNU/Linux, on tape route -n, il faut taper netstat -rn -f inet pour l'équivalent sous BSD. Notons que, sous OpenBSD, vous pouvez utiliser route -n show pour afficher les tables de routage.

Le nombre de commandes qui changent n'est pas mirobolant, mais la force de l'habitude n'a pas que du bon. Avoir un guru des systèmes BSD à ses cotés est l'idéal pour cette période de transition, sinon Google est un bon pote. À noter, un outil qui prend du galon lorsqu'on passe de Linux à FreeBSD : ifconfig permet par exemple de voir l'état du médium (connecté ou pas), de configurer le mode de transmission, et aussi (et surtout, dirais-je), de configurer les interfaces *wifi* aussi bien que les interfaces filaires. ifconfig permet aussi de lister les interfaces clonables avec le *switch* -C, de voir les stations wifi attachées lorsque l'interface est en mode *access point* et de scanner le réseau wifi à la recherche de points d'accès en mode *monitor*.

Exemple avec une interface cuivre :

```
user@freebsd % /sbin/ifconfig -m bge0
bge0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        options=1b<RXCSUM,TXCSUM,VLAN_MTU,VLAN_HWTAGGING>
        capabilities=1b<RXCSUM,TXCSUM,VLAN_MTU,VLAN_HWTAGGING>
        inet6 fe80::204:76ff:fef0:4ele%bge0 prefixlen 64 scopeid 0x3
        inet 10.0.0.1 netmask 0xffffff00 broadcast 10.0.0.255
        ether 00:04:76:33:e:1
        media: Ethernet autoselect (1000baseTX <full-duplex>)
        status: active
        supported media:
                media autoselect
                media 1000baseTX mediaopt full-duplex
                media 1000baseTX
                media 100baseTX mediaopt full-duplex
                media 100baseTX
                media 10baseT/UTP mediaopt full-duplex
                media 10baseT/UTP
                media none
```

Exemple avec une interface wifi :

```
user@openbsd % /sbin/ifconfig -m ral0
ral0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        lladdr 00:13:d3:d6:e6:e6
        media: IEEE802.11 OFDM54 mode 11g hostap (autoselect mode 11g hostap)
        status: active
        ieee80211: nwid OpenLAND chan 2 bssid 00:13:d3:6d:6e:e6 100dBm
        supported media:
                media autoselect
                media autoselect mediaopt ibss
                media autoselect mediaopt hostap
                media autoselect mediaopt monitor
                [...]
                media OFDM54
                media OFDM54 mediaopt ibss
                media OFDM54 mediaopt hostap
                media OFDM54 mediaopt monitor
        inet 10.0.1.254 netmask 0xffffff00 broadcast 10.0.1.255
        inet6 fe80::213:d3ff:fe6d:6ee6%ral0 prefixlen 64 scopeid 0x2
user@netbsd % /sbin/ifconfig -C
bridge vlan stf gif gre tun tap strip sl pppoe ppp lo
```

8.4 EN PARLANT DE RÉSEAU...

Le réseau sous les BSD se configure différemment : sous FreeBSD, il se configure dans /etc/rc.conf (avec des variables ifconfig_ifX, voire le man de rc.conf(5)) et sous NetBSD/OpenBSD, dans de simples fichiers plats /etc/ifconfig.ifX et /etc/hostname.ifX. De même, la route par défaut se place dans /etc/mygate et le nom d'hôte dans /etc/hostname. N'est-ce pas beau ?

9. CE QUI ME PLAÎT

9.1 LA STABILITÉ

Les services essentiels d'une entreprise (comme le DNS, les bases de données ou même des routeurs) ne peuvent pas être interrompus de façon intempestive. Pas question de les rebooter parce que Gérard a décidé d'installer le SNMP dessus. Même si avoir un gros *uptime* ne rend pas les cheveux brillants et n'attire pas les *guris*, il est essentiel que la disponibilité des services soit maximale (Cf. les articles de messieurs aflab & HR dans

ce même magazine). La qualité du code et sa robustesse ont prouvé que FreeBSD est une excellente base pour l'architecture d'un système informatique (Netcraft ne vous dira pas le contraire).

9.2 PF

PackerFilter (PF pour les intimes) a failli faire son apparition dans les outils qui changent, mais il est tellement beau et puissant qu'il DOIT (comme un MUST de RFC) être cité dans ce qui nous fait aimer FreeBSD. Pour tout ce qui est histoire et utilisation de PF, se rapporter à l'article de Gaston. PF, c'est avant tout une syntaxe claire comme de l'eau, qui se rapproche vraiment du langage naturel :

```
rdr on $ext1 proto {tcp, udp} \
        from any to $ip_pub port 22 -> $box port 22
```

Refaire de la syntaxe `iptables` après avoir joué avec du PF, c'est de la torture, OUI MONSIEUR ! De plus, PF offre une foultitude de fonctionnalités conviviales, comme le *binat* qui pourra vous aider si vous avez une architecture un peu tordue.

RÉFÉRENCES

- `rc.d-NG` expliqué par le barbu l'ayant conçu :
  http://www.mewburn.net/luke/papers/rc.d.pdf

- Le `rc`(8) *old-school* :
  http://www.openbsd.org/faq/faq10.html#rc

- `rc.d-NG` sous FreeBSD :
  http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/configtuning-rcd.html

- Schéma du boot d'OpenBSD :
  http://www.formation.ssi.gouv.fr/stages/documentation/architecture_securisee/images/bsd-boot.png

- Autres :
  http://www.onlamp.com/pub/a/bsd/2004/11/11/FreeBSD_Basics.html

  http://www.onlamp.com/pub/a/bsd/2005/01/13/FreeBSD_Basics.html

  http://www.over-yonder.net/~fullermd/rants/bsd4linux/bsd4linux1.php

bedis pour GCU, bedis@bedis.eu

GESTION AVANCÉE DES PORTS DANS FREEBSD

Tout d'abord, qu'est qu'un port ? On pourrait le définir ainsi : un « port » est un logiciel « porté » sur FreeBSD. En gros, un port est composé de différents éléments permettant d'installer ledit logiciel sur FreeBSD. Ces différents éléments sont :

- une liste de sites où récupérer les sources ;
- des patches à appliquer au code source pour le rendre compatible avec FreeBSD ;
- des directives de compilation ;
- les dépendances vis-à-vis d'autres ports

Tout le système des ports de FreeBSD (et des ports OpenBSD, ainsi que pkgsrc) sont organisés autour des Makefiles. Ce sont les Makefiles qui donnent les directives énumérées ci-dessus. Exemple :

```
% cat /usr/ports/graphics/pornview/Makefile
[...]
PORTNAME=       pornview
PORTVERSION=    0.2.0.p.1
[...]
CATEGORIES=     graphics
MASTER_SITES=   ${MASTER_SITE_SOURCEFORGE}
[...]
MAINTAINER=     dinoex[at]FreeBSD.org
COMMENT=        PornView is an image viewer/manager

LIB_DEPENDS=    exif.12:${PORTSDIR}/graphics/libexif

USE_XLIB=       yes
USE_GNOME=      gdkpixbuf gnomehack gnomeprefix
GNU_CONFIGURE=  yes
[...]
```

On a ainsi, successivement, le nom du port, sa version, dans quelle(s) catégorie(s) il est classé, le site où aller le télécharger (dans ce cas, Sourceforge), le mainteneur du port (celui qui s'occupe de le mettre à jour quand c'est nécessaire), une description courte d'une ligne, et des dépendances sur des bibliothèques et/ou des outils GNU.

Cela permet d'utiliser un outil que tout le monde connaît (make) et ainsi de faciliter l'utilisation des ports. En effet, installer un logiciel comme Gnome, qui présente des dizaines de dépendances, qui elles-mêmes possèdent encore des dépendances, est aussi simple que de taper :

```
% cd /usr/ports/x11/gnome2 && sudo make install clean
```

Un port se présente sous la forme de quelques fichiers textes. L'ensemble des ports est en fait une arborescence regroupant les ports en catégories. Par exemple www, mail, java...

En général, sous FreeBSD, on retrouve l'arborescence de ports dans /usr/ports.
Dans l'article, on parlera d'arbre, pour désigner cette arborescence.

Sous FreeBSD, un port peut aussi se présenter sous la forme d'une archive contenant le port compilé ainsi que des directives d'installation. On nomme cette archive un *package*. Un package est donc, grossièrement, la version compilée d'un port, comme un .rpm ou un .deb l'est d'un .tar.gz (OUI, ce sont des analogies grossières, voire vulgaires). Ici, nous traiterons principalement de la compilation de sources.

1. GESTION DE L'ARBORESCENCE

1.1 MAKE.CONF

La toute première chose avant de manipuler les ports est de configurer le fichier de configuration de make : /etc/make.conf. Voici quelques-unes des principales options que vous pouvez utiliser :

```
# Habitez-vous aux états-unis?
# (si vrai, empêchera certains ports de se compiler,
# notemment ceux relatifs à la crypto.)
USA_RESIDENT=   false

# Architecture de votre machine.
# NB le "?=" signifie qu'un port peut outrepasser
# cette valeur.
CPU_TYPE?=      i686

# Liste des langues et encodages dans lesquels on
# installera la documentation
DOC_LANG=       en_US.ISO8859-1 fr_FR.ISO8859-1

# Ne pas compiler tout ce qui concerne X
# (utile pour un serveur...)
NO_X=           true
WITHOUT_X11=    true

# Version de certains packages à utiliser
WANT_BDB_VER=44
WANT_OPENLDAP_VER=23
WANT_PGSQL_VER=81
```

Pour une liste exhaustive des options avec leur explication, consultez le fichier /usr/share/examples/etc/make.conf.

1.2 RÉCUPÉRER LES PORTS

On considère que vous avez déjà un FreeBSD installé, mais que celui-ci est tout vierge. En tant que *root*, exécutez sysinstall :

➘ menu ***Configure : Do post-install configuration of FreeBSD,***

➘ puis ***Distributions : Install additional distribution sets.***

Allez sur la ligne « *[ ] ports The FreeBSD Ports collection* », appuyez sur [ESPACE]. Validez. Choisissez votre miroir et c'est parti !

Maintenant, exécutez un petit `ls -l /usr/ports`... On va vous présenter trois manières de mettre à jour votre arbre des ports, en fonction de la version de FreeBSD que vous avez :

➘ CVSup, qui marche sur tous les Freebsd ;

➘ Csup, qui est livré avec FreeBSD 6.2, mais que l'on peut installer à partir de l'arbre des ports ;

➘ Portsnap, qui est livré avec FreeBSD 6.0, mais que l'on peut installer à partir de l'arbre des ports (oui, le chat se mord la queue).

1.2.1 CVSup

CVSup est l'outil qui va vous permettre de mettre à jour votre arborescence locale, en la comparant à un serveur.

Installer CVSup

Allez à la racine de votre arbre et recherchez CVSup :

```
# cd /usr/ports
# make search name=cvsup | grep -A 1 Port
Port:   cvsup-16.1h_2
Path:   /usr/ports/net/cvsup
--
Port:   cvsup-mirror-1.3_5
Path:   /usr/ports/net/cvsup-mirror
--
Port:   cvsup-without-gui-16.1h_2
Path:   /usr/ports/net/cvsup-without-gui
--
Port:   cvsupchk-19990209
Path:   /usr/ports/net/cvsupchk
--
Port:   fastest_cvsup-0.2.9_3
Path:   /usr/ports/sysutils/fastest_cvsup
```

On va utiliser `cvsup-without-gui-16.1h_2`. Reste plus qu'à l'installer tout simplement :

```
# cd /usr/ports/net/cvsup-without-gui
# make install clean
```

Configuration de CVSup

Créer un répertoire pour accueillir nos fichiers de configuration de CVSup et récupérer le fichier de configuration par défaut :

```
# mkdir /usr/local/etc/cvsup
# cd /usr/local/etc/cvsup
# cp /usr/share/example/cvsup/ports-supfile .
# chmod 644 /usr/local/etc/cvsup/ports-supfile
```

Grâce à ce fichier, on va indiquer à CVSup un serveur sur lequel se synchroniser, quelle branche des ports récupérer et quelles catégories de ports.

➘ `*default host=CHANGE_THIS.FreeBSD.org` : Définit le serveur sur lequel se synchroniser. Une liste des serveurs est disponible sur **http://www.freebsd.org/doc/handbook/cvsup.html#CVSUP-MIRRORS** (Cf. le chapitre « *Speedy* CVSup » pour choisir le miroir le plus rapide...).

➘ `*default base=/var/db` : Définit l'emplacement où CVSup va stocker les informations sur la collection de ports transférée sur votre machine. L'emplacement sera, dans ce cas, `/var/db/sup`.

➘ `*default prefix=/usr` : Définit l'emplacement où CVSup va stocker les fichiers demandés. Dans ce cas, CVSup va stocker les fichiers dans `/usr/ports`.

➘ `*default release=cvs tag=.` : Définit la version de la collection de port à récupérer. Dans ce cas, on prend la toute dernière version. En remplaçant le point final par `RELENG_6_1`, on va récupérer la collection correspondant à FreeBSD 6.1. (en pratique, je ne connais personne qui utilise autre chose que `-HEAD` -- mat)

➘ `*default delete use-rel-suffix` : Autorise CVSup à supprimer des fichiers dans la collection de ports locale. En gros, permet de construire en local une arborescence identique à celle du serveur. `use-rel-suffix` indique à CVSup d'ajouter un suffixe construit à partir de la version et du tag ajoutés au nom de chaque fichier qu'il maintient. (plus efficace)

➘ `*default compress` : Les échanges de fichiers du serveur vers le client CVSup seront compressés.

! ATTENTION

Utile pour les connexions peu rapides, mais inutile sur un LAN ou si vous avez une grosse connexion internet.

Maintenant, il ne reste plus qu'à savoir quelle partie de la collection récupérer. Pour aller vite, on peut utiliser `ports-all`. Ceci permettra de récupérer TOUTE la collection.

Si vous préférez ne récupérer que les catégories qui vous intéressent, utilisez une liste de la sorte :

```
ports-base
ports-archivers
ports-audio
ports-converters
ports-databases
ports-devel
ports-dns
ports-editors
ports-french
ports-ftp
...
```

! ATTENTION

Conservez toujours `ports-base`, si vous ne voulez pas rencontrer de problèmes lors de futures compilations !

Mettre à jour la collection de ports
Simple, il suffit d'exécuter CVSup en appelant le fichier de configuration :

```
# cvsup /usr/local/etc/cvsup/ports-supfile
[...]
 Checkout ports/audio/amarok/files/patch-configure
 Edit ports/audio/amarok/pkg-descr
 Edit ports/audio/amarok/pkg-message
 Edit ports/audio/amarok/pkg-plist
 Edit ports/audio/anthem/Makefile
 Edit ports/audio/arts/Makefile
[...]
```

1.2.2 Csup

Csup est un autre outil qui va vous permettre de mettre à jour votre arborescence locale, en la comparant à un serveur. Les deux différences qu'il a avec CVSup à ma connaissance sont qu'il est écrit en C (et pas dans un langage d'extra-terrestre) et surtout qu'il est présent dans le *basesystem* de FreeBSD depuis FreeBSD 6.2 (donc cela évite l'installation d'un package supplémentaire).

En revanche, Csup parle le protocole de CVSup. Il se connecte donc à un serveur CVSup, et utilise **strictement** le même fichier de configuration que CVSup. La description nous dit que « Csup est une réécriture du client de mise à jour CVSup faite dans le langage C ». N'est-ce pas amour, tout cela ?

Installer Csup

```
% ls -l /usr/bin/csup
-r-xr-xr-x  1 root  wheel  86244 Dec  9 03:09 /usr/bin/csup*
```

Oh, mais que de magie ! Csup est déjà présent parmi nous mes frères ! (Comme dit ci-dessus). Il n'y a plus qu'à suivre les autres étapes présentes pour CVSup, c'est-à-dire : « Configuration de CVSup » et « Mettre à jour la collection de ports ». Bien entendu, à tous les endroits où il est indiqué « CVSup », il faut remplacer par « Csup » !

1.2.3 Portsnap

Présentation (honteusement repompée du Handbook)
Portsnap est un système de distribution sécurisée du catalogue des logiciels portés sur FreeBSD. Approximativement chaque heure, un « instantané » (*snapshot*) du catalogue des logiciels portés est généré, rassemblé et signé de manière chiffrée. Les fichiers résultants sont alors distribués par l'intermédiaire du protocole HTTP.

Tout comme CVSup, Portsnap utilise un modèle de mise à jour de type *pull* : le catalogue des logiciels portés packagé et signé est placé sur un serveur Web qui attend les requêtes des clients. Les utilisateurs doivent soit exécuter manuellement `portsnap`(8) pour télécharger les mises à jour, soit configurer `cron`(8) pour un téléchargement régulier et automatique des mises à jour.

Pour des raisons techniques, Portsnap ne met pas à jour le catalogue des logiciels portés directement dans le répertoire `/usr/ports` ; le logiciel travaille plutôt par défaut sur une version compressée de l'arborescence des logiciels portés dans le répertoire `/var/db/portsnap`. Cette copie compressée est ensuite utilisée pour mettre à jour le catalogue des logiciels portés.

Première utilisation de Portsnap
Au premier lancement de la commande, il va chercher la totalité de l'arbre des ports (sous forme compressée) sur un des serveurs Portsnap le mettant à disposition. Cela est effectué avec la commande `portsnap fetch`.
Une fois que l'arbre des ports a été récupéré, vous devez le mettre en place dans `/usr/ports`. `portsnap extract`.

Utilisations suivantes (mise à jour de l'arbre des ports)
Très simple, `portsnap fetch update`.

! ATTENTION

Il ne faut pas exécuter cette commande lorsque des ports sont en train d'être compilés.

1.2.4 Speedy CVSup

Dans les ports, il y a ce magnifique petit outil : `fastest_cvsup`. Il va permettre de trouver les 3 serveurs CVSup les plus rapides vis-à-vis de notre serveur. On l'installe comme suit.
Si on a déjà installé `portupgrade` :

```
$ sudo portinstall sysutils/fastest_cvsup
```

Sinon :

```
# cd /usr/ports/sysutils/fastest_cvsup
# sudo make install clean
```

Et maintenant, on l'exécute comme suit :

```
$ fastest_cvsup -c fr
[...]
>>  Speed Daemons:
    - 1st: cvsup2.fr.freebsd.org
    - 2nd: cvsup4.fr.freebsd.org
    - 3rd: cvsup8.fr.freebsd.org
```

Et voilà ! Il ne reste plus qu'à choisir le serveur le plus rapide dans vos fichiers de configuration !

2. GESTION AVANCÉE DES PORTS

2.1 INSTALLATION DE PORTUPGRADE

Allez dans `/usr/ports/sysutils/portupgrade`, puis

```
# make install clean
```

Gardez les options par défaut. `portupgrade` va nous installer un ensemble d'outils qui vont nous faciliter la maintenance des ports.

2.2 LE FICHIER PKGTOOLS.CONF

Toute la configuration de `portupgrade` se fait dans le fichier `/usr/local/etc/pkgtools.conf`.

2.2.1 Variables d'environnement

La collection de ports utilise quelques variables, habituellement définies dans `make.conf`. Les outils fournis par `portupgrade` n'utiliseront pas ces variables, mais celles définies dans le fichier `pkgtools.conf`. Voici les variables par défaut :

```
# Emplacement de la collection de ports
ENV['PORTSDIR'] ||= '/usr/ports'
# Emplacement des packages
ENV['PACKAGES'] ||= ENV['PORTSDIR'] + '/packages'
# Répertoire regroupant tous les packages
ENV['PKG_PATH'] ||= ENV['PACKAGES'] + '/All'
# Où sauvegarder les packages
ENV['PKG_BACKUP_DIR'] ||= ENV['PKG_PATH']
```

2.2.2 Choix des catégories

Il est possible de signaler aux outils d'ignorer certaines catégories de ports, grâce à la variable `IGNORE_CATEGORIES` :

```
IGNORE_CATEGORIES = [
   'chinese',
   'german',
   'hebrew',
   'japanese',
   'korean',
   'russian',
   'ukrainian',
   'vietnamese',
   ]
```

Si vous utilisez cette variable, vous devrez remettre à jour la base des ports avec la commande `portsdb -Ufu`.

! ATTENTION

C'est long à générer. Pour aller plus vite, exécutez simplement (après un `cvsup`) : `portsdb -F`. Cela va récupérer la base des ports sur le net plutôt que d'avoir à la regénérer entièrement !

2.2.3 Ports à ne pas mettre à jour

`portupgrade` nous offre la possibilité de ne pas mettre à jour certains ports. Ça peut permettre de gagner du temps, surtout pour les *desktops* où l'on pourra se passer de recompiler tout Xorg ou OpenOffice...

```
HOLD_PKGS = [
   'xorg-*',
   'openoffice*',
   ]
```

2.2.4 Packages plutôt que ports

Il est possible de donner une liste de ports pour lesquels on souhaite (voire on veut uniquement) installer la version packagée :

```
# Liste de ports que l'on souhaite gérer via un package,
# mais si pas possible on accepte de compiler la version
# des ports.
USE_PKGS = [
   'firefox*',
   ]
# Liste de ports que l'on souhaite gérer en mode package
# exclusivement.
USE_PKGS_ONLY = [
   'openoffice*',
   'gimp*',
   ]
```

2.2.5 Ports alternatifs

Avec `pkgtools.conf`, il est possible de redéfinir les dépendances d'un port. Par exemple, si un port dépend d'un autre port A, on peut le forcer à utiliser B à la place de A.

```
ALT_PKGDEP = {
  # Utiliser le port OpenLDAP 2.3 à la place du port
  # OpenLDAP 2.2
  'openldap22*' => 'openldap23*',

  # La même chose que ci-dessus, mais en utilisant
  # l'emplacement du port
  'net/openldap22-client' => 'net/openldap23-client',
  'net/openldap22-server' => 'net/openldap23-server',

  # Si vous avez installé apache directement depuis
  # les sources :
  'apache' => :delete,
}
```

2.2.6 Options des ports

Lors de l'installation de `portupgrade`, on a vu que l'on pouvait définir des options à la compilation. C'est très pratique, pour *tuner* au maximum les applications en place, et n'avoir que ce dont on a besoin. Seul inconvénient, lorsque l'on a 150 ports installés, il faut retenir chacune des options que l'on a décidé d'installer... `pkgtools.conf` nous permet de nous affranchir de cela grâce à la variable `MAKE_ARGS`.

```
MAKE_ARGS = {
   'mail/p5-Mail-SpamAssassin' => [
      'WITH_AS_ROOT=yes',
      'WITHOUT_DOMAINKEYS=yes',
      'WITHOUT_MYSQL=yes',
      'WITH_PGSQL=yes',
      'WITH_RAZOR=yes',
      'WITHOUT_SPF_QUERY=yes',
      'WITHOUT_RELAY_COUNTRY=yes',
      'WITH_TOOLS=yes',
   ],
}
```

2.2.7 Gestion des services

Sur un serveur, un port peut installer un service qui tourne sous forme de démon (Apache, ldap, MySQL, etc.). `pkgtools.conf` va nous permettre de configurer des commandes à exécuter lors d'un *upgrade* de port.

```
# L'upgrade consiste à désinstaller l'ancienne
# version du port
# Ici, on demande d'arrêter apache avant de le
# désinstaller :
BEFOREDEINSTALL = {
  # Arrêter apache
  # localbase() retourne le répertoire où sont
  # installé les ports.
  'www/apache*' => localbase() + '/sbin/apachectl -k stop',

  # Arrêter automatiquement tout les ports
  # qui ont un script rc activé
  '*' => proc { |origin|
    cmd_stop_rc(origin)
  },
}

# Ensuite, l'upgrade installe la nouvelle version
# du port
AFTERINSTALL = {
  # Démarrer apache
  'www/apache*' => localbase() + '/sbin/apachectl -k start',

  # Démarrer automatiquement tous les port
  # qui ont un script rc activé
  '*' => proc { |origin|
    cmd_start_rc(origin)
  },
}
```

2.3 RECHERCHER UN PORT

Allez dans `/usr/ports` et utilisez une commande du style `make search name=PORT`. Exemple :

```
# make search name=cvsup
Port:   cvsup-16.1h_2
Path:   /usr/ports/net/cvsup
Info:   General network file distribution system optimized\
        for CVS (GUI version)Maint:  jdp@FreeBSD.org
B-deps: expat-2.0.0_1 ezm3-1.2 fontconfig-2.3.2_3,1\
        freetype2-2.1.10_3 libdrm-2.0_1 pkgconfig-0.20\
        xorg-libraries-6.9.0
R-deps: expat-2.0.0_1 fontconfig-2.3.2_3,1\
        freetype2-2.1.10_3 libdrm-2.0_1 pkgconfig-0.20\
        xorg-libraries-6.9.0
WWW:    http:\/\/www.cvsup.org/
[...]
```

On peut aussi rechercher par mot clé en utilisant `make search key=KEYWORD`.

2.4 INSTALLER UN PORT

Pour installer un port, nous allons utiliser la commande `ortinstall`. Principales options :

- `-e` ou `--emit-summaries` : génère un rapport après chaque port installé
- `-f` ou `--force` : force la mise à jour, même si le port est en version inférieure ou égale au port déjà installé ou s'il est défini dans la variable `HOLD_PKGS`.
- `-i` ou `--interactive` : mode interactif.
- `-k` ou `--keep-going` : continue, même si l'installation d'un port échoue.
- `-l FILE` ou `--result-file FILE` : fichier où écrire le résultat des opérations.
- `-m` ou `--make-args` : spécifier des arguments à passer à chaque commande `make`.
- `-M` ou `--make-env` : spécifier des variables d'environnement.
- `-n` ou `--noexecute` : n'installe ni ne met à jour aucun port. Montre juste ce qui va être fait.
- `-p` ou `--package` : crée le package du port.
- `-P` ou `--use-packages` : utilise un package au lieu d'un port, quand c'est possible.
- `-PP` ou `--use-packages-only` : n'utilise jamais le port, mais uniquement le package.
- `-q` ou `--noconfig` : ne lit pas le fichier de configuration `pkgtools.conf`.
- `-r` ou `--recursive` : agit sur tous les ports dépendants du port donné.
- `-R` ou `--upward-recursive` : agit sur tous les ports qui requièrent le port donné.
- `-s` ou `--sudo` : exécute `sudo` sur les commandes le nécessitant.
- `-u` ou `--uninstall-shlibs` : supprime les bibliothèques partagées obsolètes.
- `-v` ou `--verbose` : mode bavard.
- `-y` ou `--yes` : répond `yes` à toutes les questions.

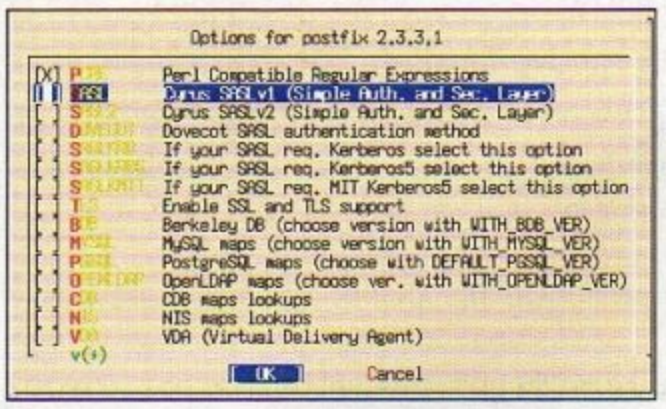
Cas d'école : installation de Postfix :

1 Aller dans le répertoire du port :

```
# cd /usr/ports/mail/postfix
```

2 Voir quelles sont les options de configuration disponibles :

```
# make config
```



NOTE

On peut aussi utiliser un `make showconfig`, mais y'a pas les couleurs...

3 Compléter la variable `MAKE_ARGS` du fichier `pkgtools.conf` avec les options désirées ou non :

```
MAKE_ARGS = {
   'mail/postfix' => [
     'WITH_PCRE=yes',
     'WITHOUT_SASL=yes',
     'WITH_SASL2=yes',
[...]
    ],
}
```

4 Lancer l'installation

```
sudo portinstall mail/postfix
---> Installing 'postfix-2.3.3,1' from a port (mail/postfix)
---> Building '/usr/ports/mail/postfix' with make flags: \
  WITH_PCRE=1 WITHOUT_SASL=1 WITHOUT_SASL2=1 WITH_TLS=1
```

Notez la prise en compte des variables positionnées dans le fichier `pkgtools.conf`.

2.5 SUIVI DES MISES À JOUR

Une fois que vous avez installé vos logiciels et que votre serveur est en production, il faut le maintenir à jour.

2.5.1 Maintien de l'arbre des ports à jour

Très simple, placez la commande `/usr/local/bin/cvsup /usr/local/etc/cvsup/ports-supfile` dans un *cron*. Après chaque mise à jour, vous devrez mettre à jour le fichier `INDEX` (base des ports disponibles). C'est possible de le faire grâce à `portsdb` :

- `portsdb -U` : génère le fichier en analysant l'arbre complet. Déconseillé, car long et coûteux en ressources.
- `portsdb -Fu` : récupère le fichier `INDEX` du net, puis met à jour le `INDEX.db`.

Ce qui nous donne :

```
22 12 * * * /usr/local/bin/cvsup /usr/local/etc/cvsup/\
          ports-supfile 2>&1 >>/dev/null
22 34 * * * /usr/local/sbin/portsdb -Fu 2>&1 >>/dev/null
```

Compter 30 minutes environ pour le CVSup.

2.5.2 Recherche des ports à mettre à jour

Pour savoir quels ports ne sont pas *up to date* sur notre serveur, on va utiliser la commande `portversion`. Principales options :

- `-l CAR` ou `--limit CAR` : inclut uniquement les packages avec le statut CHARS
- `-v` ou `--verbose` : active le mode bavard
- `-x CHAINE` ou `--exclude CHAINE` : n'effectue pas de comparaison pour les ports dont les noms *matchent* la CHAINE.

Utilisation : ne lister que les ports qui ne sont pas à jour :

```
portversion -v -l\<
ezm3-1.2        <  needs updating (port has 1.2_1)
gmake-3.80_2    <  needs updating (port has 3.81_1)
postfix-2.3.3,1 <  needs updating (port has 2.3.4,1)
```

Il ne reste plus qu'à coller le tout dans une *crontab*, avec un `| mail` :

```
22 44 * * * /usr/local/sbin/portversion -v -l\< | \
          /usr/bin/mail admin@domaine.tld
```

Et voilà. Nous venons de voir comment maintenir à jour la base des ports et configurer un agent qui nous remontera chaque port ou package à mettre à jour.

2.6 MISE À JOUR D'UN PORT

Maintenant que notre arborescence des ports se maintient à jour, nous allons pouvoir mettre à jour les ports. Nous allons utiliser la commande `portupgrade`. Les options sont les mêmes que pour `portinstall`.

REMARQUE

A la racine de votre arborescence des ports (traditionnellement `/usr/ports`), il y a un fichier `UPDATING`, qu'il est de bon ton de consulter avant chaque mise à jour.

Ce fichier contient les problèmes pouvant survenir suite à une mise à jour, et surtout une méthode pour les régler.

Fonctionnement : grossièrement, `portupgrade` va récupérer les sources et compiler. Si tout se passe bien, alors il va désinstaller le port actuellement en production et installer celui fraîchement compilé.

ATTENTION

Si le port en question installe un démon, pensez à le signaler dans `pkgtools.conf`, directives `AFTERINSTALL` et `BEFOREDEINSTALL`, afin que `portupgrade` l'arrête, désinstalle l'ancienne version, installe la nouvelle version et redémarre le service.

Dans le chapitre précédent, nous avons vu que notre Postfix n'est pas à jour. Nous allons donc le mettre à jour. Premièrement, éditez `pkgtools.conf`, directive `BEFOREDEINSTALL` :

```
# Stop postfix
'mail/postfix*' => localbase() + '/etc/rc.d/postfix stop',
```

Directive `AFTERINSTALL` :

```
# Start postfix
'mail/postfix*' => localbase() + '/etc/rc.d/postfix start',
```

Puis exécutez `portupgrade` :

```
# portupgrade -r postfix-2.3.3,1
[...]
---> Building '/usr/ports/mail/postfix' with make flags: \
   WITH_PCRE=1 WITHOUT_SASL=1 WITHOUT_SASL2=1 WITH_TLS=1
[...]
---> Executing a command before deinstalling 'mail/
postfix':\
   /usr/local/etc/rc.d/postfix stop
postfix/postfix-script: stopping the Postfix mail system
[...]
---> Executing a post-install command for 'mail/postfix':\
   /usr/local/etc/rc.d/postfix start
postfix/postfix-script: starting the Postfix mail system
```

Et voilà Postfix démarré et fonctionnel. Temps de *downtime* du service, quelques secondes, et le package a été recompilé avec les mêmes options qu'à l'installation.

2.7 DÉSINSTALLATION D'UN PORT

Pour désinstaller un port, nous allons utiliser `pkg_deinstall`. Il est capable de gérer les dépendances. Principales options :

- `-d` ou `--rmdir` : supprime les répertoires vides.
- `-f` ou `--force` : force la suppression d'un port.
- `-i` ou `--interractive` : demande confirmation avant chaque suppression.
- `-n` ou `--noexexute` : montre ce que la commande ferait sans l'exécuter.
- `-r` ou `--recursive` : supprime le package et toutes ses dépendances.
- `-R` ou `--upward-recursive` : supprime le package et les packages requis pour installer celui-ci.
- `-v` ou `--verbose` : mode bavard.

Veuillez noter les différences :

```
# pkg_deinstall -n ruby
--->  Deinstalling 'ruby-1.8.5_3,1'
--->  Listing the results (+:done / -:ignored / *:skipped /\
                         !:failed)
        + ruby-1.8.5_3,1
--->  Packages processed: 1 done, 0 ignored, 0 skipped and \
                         0 failed

# pkg_deinstall -n -r ruby
--->  Deinstalling 'libchk-1.9'
--->  Deinstalling 'portupgrade-2.1.3.3_1,2'
--->  Deinstalling 'ruby18-bdb-0.5.9_2'
--->  Deinstalling 'ruby-1.8.5_3,1'
--->  Listing the results (+:done / -:ignored / *:skipped /\
                         !:failed)
        + libchk-1.9
        + portupgrade-2.1.3.3_1,2
        + ruby18-bdb-0.5.9_2
        + ruby-1.8.5_3,1
--->  Packages processed: 4 done, 0 ignored, 0 skipped and\
                         0 failed

# pkg_deinstall -n -R ruby
--->  Deinstalling 'ruby-1.8.5_3,1'
--->  Listing the results (+:done / -:ignored / *:skipped /\
                         !:failed)
        + ruby-1.8.5_3,1
--->  Packages processed: 1 done, 0 ignored, 0 skipped and\
                         0 failed
```

2.8 NETTOYAGE DE L'ARBRE DES PORTS

Avec toutes ces installations, il est possible que certains fichiers obsolètes encombrent notre disque dur. Nous allons utiliser `portsclean` pour nettoyer tout ça. Principales options :

- `-C` : supprime les répertoires de compilation (`work` dans chaque répertoire d'un port).
- `-D` : supprime les archives sources qui ne sont pas référencées dans l'arbre.
- `-DD` : supprime les archives sources des ports qui ne sont pas installés.
- `-P` : supprime les archives de packages obsolètes.
- `-PP` : supprime toutes les archives de packages.

Nous allons procéder à un petit nettoyage comme suit :

```
$ sudo portsclean -C -DD -PP
```

3. SÉCURITÉ

3.1 AUDIT DES PORTS

Sur des serveurs en production, on ne peut pas tous les jours compiler les ports qui ne sont pas à la dernière version. Par contre, il est très important de savoir si une faille de sécurité a été découverte dans un des ports installés sur la machine.

On va utiliser `security/portaudit` pour cela. Principales options de `portaudit` :

- `-a` : teste tous les ports installés.
- `-d` : affiche la date de création de la DB.
- `-F` : récupère la dernière version de la DB.
- `-v` : mode bavard.

Une fois installé, exécutez :

```
# /usr/local/sbin/portaudit -Fda
```

Cela va récupérer la dernière DB contenant les vulnérabilités et effectuer un test. Par ailleurs, Portaudit s'exécutera une fois par nuit et son rapport sera envoyé avec le mail « *security* » du système. Voici un exemple de vulnérabilité :

```
Affected package: php5-5.1.6
Type of problem: php -- open_basedir Race Condition
 Vulnerability.
Reference:
<http://www.FreeBSD.org/ports/portaudit/\
edabe438-542f-11db-a5ae-00508d6a62df.html>

1 problem(s) in your installed packages found.
```

Si j'essaye de mettre à jour ce port malgré la faille, je vais avoir un message d'erreur :

```
sudo portupgrade -r php5-5.1.6
--->  Upgrading 'php5-5.1.6' to 'php5-5.1.6_1' (lang/php5)
[...]
=> php -- open_basedir Race Condition Vulnerability.
   Reference:
<http://www.FreeBSD.org/ports/portaudit/\
edabe438-542f-11db-a5ae-00508d6a62df.html>
=> Please update your ports tree and try again.
* Error code 1
Stop in /usr/ports/lang/php5.
```

Vous aurez beau mettre à jour vos ports, si la version disponible est vulnérable, cela ne changera rien et l'installation/mise à jour du port sera bloquée. Pour passer outre cela, il faut utiliser :

```
sudo portupgrade -r -m -DDISABLE_VULNERABILITIES php5
```

LIEN

http://www.FreeBSD.org/ports

Emile « iMil » Heitor, GCU canal historique, pour GLMF

NETBSD DANS LA POCHE

Combien de fois vous êtes-vous retrouvé chez tonton Louis ou beau-père Michel, fiers de leur nouvelle bête de course équipée d'un accès DSL, tétanisé devant un écran bleu pastel arborant en guise de *login* un jeune en skateboard ou une pièce d'échiquier ? Ne vous êtes-vous pas maudit à ce moment, alors que vous alliez passer le week-end entier à cet endroit, de ne pas avoir pris dans vos bagages un *live CD* quelconque, un OS muni d'un shell digne de ce nom, ainsi que de tous les outils indispensables à un week-end de repos ? Et vous avez, dans votre poche, cette clé USB, pleine à craquer de mp3 – libres évidemment –, de photos de vacances et autres captures vidéo de la dernière *Solutions Linux*. Il restait pourtant dans cette clé 64 mégas octets inutilisés, 64 mégas dans lesquels vous auriez pu caser un UNIX minimal, un diamant qui serait muni d'un OpenSSH, d'un Irssi et d'un Links, que vous aurait présenté un Ion 3 à peine configuré.

Voici ce que cet article vous propose : disposer à tout instant du couteau suisse de l'administrateur système UNIX, un NetBSD minimal, mais fonctionnel, qui se logera dans un coin de votre clé USB, sans pour autant la rendre inexploitable par d'autres systèmes. Le but de l'opération sera d'obtenir une clé amorçable, qui n'utilisera pas le système de fichiers de la clé, mais une image brute, générée à l'aide de l'outil `dd`(1), qu'un noyau affublé d'un RAMDISK ira *mounter* comme *root filesystem*.

Pour construire notre OS minimal, nous utiliserons, sur une machine NetBSD 3.1, des outils présents dans le *basesystem*, mais également Grub, disponible dans `pkgsrc` (`pkgsrc/sysutils/grub`). Enfin, Qemu, installable depuis `pkgsrc/emulators/qemu`, nous servira à tester nos travaux. Cet article s'appuie sur un projet sur lequel j'ai travaillé en 2006 : the NetBSD LiveKey (**http://imil.net/nlk/**). J'utiliserai donc tout naturellement quelques scripts écrits pour l'occasion qui nous simplifieront grandement la tâche.

NOTE

Quelques notions des systèmes BSD, et particulièrement NetBSD, sont nécessaires pour la bonne compréhension de cet article. Si toutefois ces notions vous faisaient défaut, vous trouverez probablement de l'aide dans le fabuleux NetBSD Guide à cette adresse **http:www.netbsd.org/guide/en/** ou en version française (mais un peu dépassée) à cette adresse :**http:www.mclink.it/personal/MG2508/nbsdfra/netbsd.html**.

1. CRÉATION DU NOYAU

La première étape de notre périple va consister à créer un noyau muni d'un RAMDISK. Pour cela, nous allons compiler un noyau NetBSD de manière habituelle (voir **http://netbsd.org/Documentation/kernel/#how_to_build_a_kernel**), en nous assurant que les options suivantes sont bien activées/renseignées :

```
# Nous allons utiliser UNION filesystem pour mounter
# les partitions "réelles" sur le ramdisk
file-system     UNION

# vnd est le device permettant de mounter des images,
# notre root filesystem sera une image
pseudo-device   vnd             4
# et pour stocker encore un peu plus de données,
# nous utiliserons une image compressée
options         VND_COMPRESSION

# Les directives de création et d'espace nécessaire
# au RAMDISK
options         MEMORY_DISK_HOOKS
options         MEMORY_DISK_IS_ROOT
options         MEMORY_DISK_SERVER=0
options         MEMORY_DISK_ROOT_SIZE=32768
options         MEMORY_RBFLAGS=0
```

Procédons donc à la compilation de ce noyau custom (on considère que le fichier de configuration utilisé est `NBUSB`) :

```
# pwd
/usr/src/sys/arch/i386/conf
# config NBUSB
# cd ../compile/NBUSB
# make depend && make
```

Si tout s'est bien déroulé, vous devriez maintenant disposer d'un noyau nommé `netbsd` à cet endroit de l'arborescence.

Il nous faut maintenant affubler ce noyau anormalement gros d'un *root filesystem* minimal, qui constituera le pivot de notre mini-OS. Afin de simplifier l'opération, nous allons peupler ce RAMDISK de quelques outils compilés statiquement, que nous piocherons dans le répertoire `/rescue` de notre NetBSD souche. Pour peupler ce disque d'amorçage du système, nous devons réfléchir aux différentes étapes : 1. obligatoires dans une séquence de *boot*, 2. dont nous allons avoir besoin dans le cas spécifique qui nous occupe, à savoir, utiliser autre chose que le système de fichiers initial.

Nous savons que le premier processus qui sera appelé est `init`(8), classiquement situé dans `/sbin`. Créons donc une fausse arborescence munie de cette ébauche d'OS :

```
# mkdir -p /home/pinpin/nbusb/fakeroot/sbin
# cp /rescue/init /home/pinpin/nbusb/fakeroot/sbin
```

init(8), sur un système héritier de 4.0BSD, exécute le script /etc/rc, qui habituellement s'occupe de démarrer différents services à l'aide des divers /etc/rc.*. Nous allons utiliser rc(8) de manière beaucoup plus basique. En effet, cet /etc/rc sera exécuté dans le contexte d'un boot singulier, et c'est à lui que reviendra la charge d'effectuer le pivot entre le RAMDISK et l'image root contenant les moult outils qui motivent cette tâche. Voici un exemple de fichier fakeroot/etc/rc minimal et commenté :

```
# debut de /etc/rc

echo "Initialisation du système..."

export PATH=/sbin:/bin:/usr/sbin:/usr/bin
# on exporte la variable qui contient le point
# de montage de la clé
KEYMOUNTPOINT=/mnt
# puis une variable contenant les données utiles
KEYFILESPATH=${KEYMOUNTPOINT}/nbusb
# enfin, le nom de l'image que nous créerons plus tard
IMGNAME=nbusb.zimg

sleep 2 # on attend la fin de l'affichage

echo
echo "-----------------------------------------"
echo " . Stage 2: montage de l'image disque"
echo "-----------------------------------------"
echo
echo "Nous allons monter le système de fichier VFAT."
echo -n "Entrez le device sur lequel se trouve " \
        "l'image root (wd0a, wd0e, sd0e...): "
read dev

# 1ere passe, on monte la clé avec le système
# de fichiers VFAT
echo "Montage de /dev/${dev}"
mount_msdos /dev/${dev} ${KEYMOUNTPOINT}

# Nous avons maintenant la main sur le filesystem de la clé,
# nous allons pouvoir créer le device virtuel vnd qui sera
# associé au fichier image
vnconfig -z vnd0 ${KEYFILESPATH}/${IMGNAME}
# /!\ issu de man vnconfig :
#
#     -z      Assume that regular file is compressed
#             disk image in cloop2 format, and
#             configures it read-only.
#                       ^^^^^^^^^

# 2eme passe, nous pouvons maintenant monter / issu
# du device vnd
mount_ffs -o ro,union /dev/vnd0a / >/dev/null 2>&1

# puis on monte le reste du filesystem en Memory
# FileSystem (vnd compressé et en lecture seule)
echo "montages mfs"
mount_mfs -s 262144 swap /tmp
mount_mfs -s 12000 swap /dev
mount_mfs -s 262144 swap /var
mount_mfs -s 131072 swap /etc
mount_mfs -s 262144 swap /home
mount_mfs -s 262144 swap /usr/pkg
sleep 1

echo "Création des devices..."
cd /dev && sh MAKEDEV all

echo
echo "-----------------------------------------"
echo " . Stage 3: multiutilisateur"
echo "-----------------------------------------"
echo

# on appelle maintenant le "vrai" rc
sh /etc/rc

# fin de /etc/rc
```

Ce rc fait appel à différents exécutables que nous devons donc copier sur notre clé : sh (sans lui, point de rc !) et sleep dans fakeroot/bin, puis mount_msdos, mount_ffs, mount_mfs et vnconfig dans fakeroot/sbin. Copiez également les bibliothèques desquelles dépendent ces outils, typiquement tout /lib, /usr/lib/libc.so* ainsi que /usr/lib/libutil.so.* et évidemment /libexec/ld.elf_so.

Maintenant que nous disposons de la structure de fichiers initiale, il nous faut la greffer à notre noyau. Nous allons pour cela utiliser les outils makefs(8) et mdsetimage(8). Ces outils servent respectivement à générer une image à partir d'une structure de fichiers et à copier cette image dans le RAMDISK du noyau.

Créons une image de 16 mégaoctets du filesystem basique créé plus haut. Nous choisissons évidemment FFS comme type de système de fichiers :

```
# makefs -s 16m -t ffs md.img fakeroot
```

Copions maintenant cette image dans notre RAMDISK :

```
# mdsetimage netbsd md.img
```

Afin de ne gaspiller aucun octet, compressons ce noyau :

```
# gzip -9 netbsd && mv netbsd.gz netbsd
```

Votre noyau est fin prêt.

2. CRÉATION DU ROOT FILESYSTEM

Muni de son système initial, notre noyau va dérouler, init(8) va invoquer rc qui va tenter de monter l'image nbusb.zimg. Ce fichier image n'est rien de plus qu'un disque virtuel créé à l'aide de dd(1), que nous allons peupler comme une simple partition.

Créons donc un disque vide de 128 mégaoctets :

```
# dd if=/dev/zero of=nbusb.img count=262144 bs=512
```

Associons-le à un *device* virtuel de type `vnd` :

```
# vnconfig -c -v /dev/vnd0 nbusb.img
```

On génère, comme pour un véritable disque, un label :

```
# cat > nbusb.disktab << EOF
  miniroot:\
  :ty=floppy:se#512:nt#1:rm#3600:ns#262144:nc#1:\
  :pa#262144:oa#0:ba#4096:fa#512:ta=4.2BSD:\
  :pb#262144:ob#0:\
  :pc#262144:oc#0:
  EOF
```

On inscrit ce label :

```
# disklabel -w -f nbusb.disktab /dev/vnd0 nbusb
```

Puis, on construit le système de fichiers :

```
# newfs -m 0 /dev/vnd0a
```

Il est désormais possible de monter ce disque :

```
# mount /dev/vnd0a /mnt
```

Reste à peupler ce disque avec les *packages* de base de votre choix. Pour cette petite démonstration, nous nous contenterons des sets `base` et `etc`, que vous pourrez télécharger ici : **ftp://ftp.netbsd.org/pub/NetBSD/NetBSD-3.1/i386/binary/sets/**. Remplissons notre faux disque des packages fraîchement récupérés :

```
# cd /mnt
# tar zxvfp /home/pinpin/tmp/base.tgz
# tar zxvfp /home/pinpin/tmp/etc.tgz
```

Notre image dispose maintenant d'un système minimal, mais parfaitement fonctionnel. Reste à lui apporter quelques petites modifications afin de le rendre exploitable ; comme placer la variable `rc_configured` à `YES` dans `etc/rc.conf`, éventuellement ajouter un utilisateur ou en tout cas s'assurer qu'on pourra passer root. Pourquoi pas spécifier un nom d'hôte dans `/etc/myname` ou configurer quelques variables d'environnement dans l'`/etc/profile`.

Notre disque créé et peuplé, il nous reste à le compresser afin qu'il occupe le moins de place possible sur notre clé. Ceci est réalisé à l'aide de la commande `vndcompress`(1) :

On démonte notre disque virtuel :

```
# umount /mnt
```

On le désassocie du device `vnd0` :

```
# vnconfig -u vnd0
```

Et on compresse l'image :

```
# vndcompress nbusb.img nbimg.zimg
```

Nous avons spécifié dans le fichier `rc` du RAMDISK que l'image se situera dans le répertoire `nbusb/` sur la clé au format VFAT. C'est donc dans ce répertoire que nous allons copier notre image :

```
# mount_msdos /dev/sd0e /mnt
```

`sd0e` représente le device associé à la partition VFAT de notre clé :

```
# mkdir /mnt/nbusb
# cp netbsd /mnt/nbusb
# cp nbusb.zimg /mnt/nbusb
```

Reste à rendre notre clé amorçable grâce à Grub :

```
# grub-install --root-directory=/mnt /dev/sd0
```

Et créer un `menu.lst` idoine :

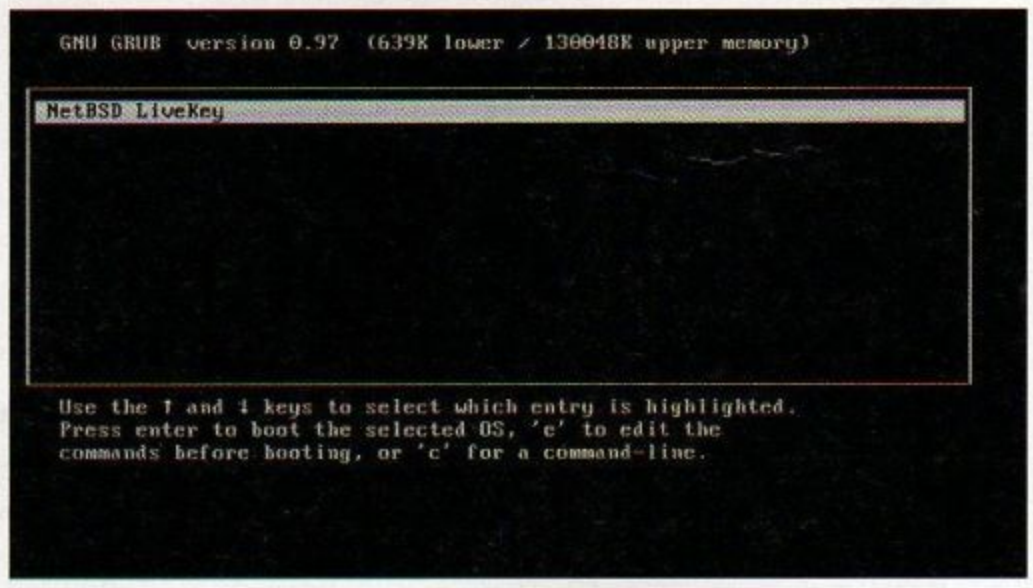
```
# cat > /mnt/boot/grub/menu.lst << EOF
title NetBSD LiveKey
root (hd0,0)
kernel --type=netbsd /nbusb/netbsd
boot
EOF
```

C'est terminé !

Vous pouvez maintenant tester votre NetBSD-light grâce, par exemple, à l'excellent Qemu en utilisant la syntaxe suivante :

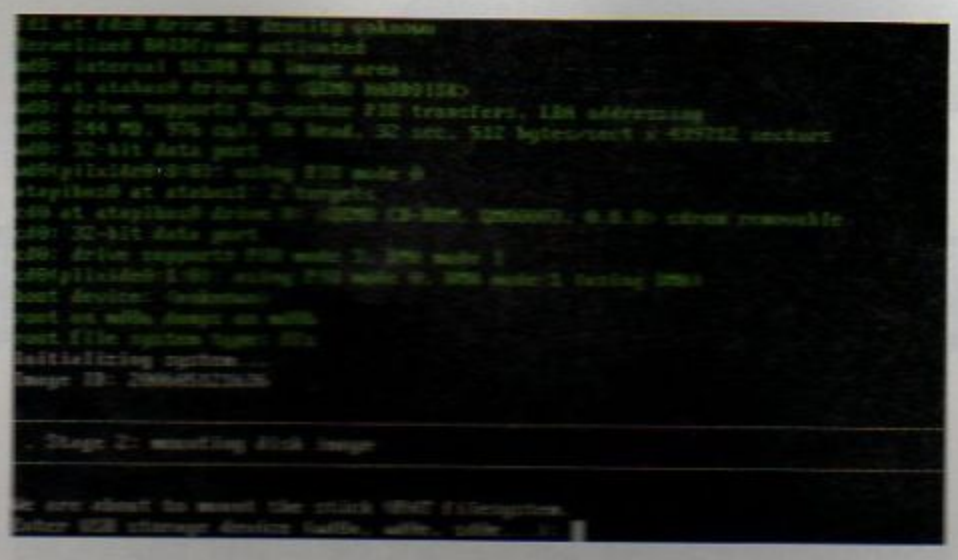
```
# qemu /dev/sda
```

Si tout se passe bien, vous devriez admirer un magnifique menu Grub.



! ATTENTION

Pour Qemu, le disque source sera exploité comme un disque IDE. Aussi, lorsque `rc` demandera quel device utiliser comme root filesystem, c'est bien wd0X (ou X=a, e...) qu'il faudra entrer, *a contrario* d'une vraie utilisation lors de laquelle le noyau NetBSD associera votre clé à un device de type sdXY.



À la fin de la séquence de boot, vous devriez normalement vous trouver devant un TTY qui vous présente fièrement sa demande de *login*. Y'a plus qu'à.

```
Checking for botched superblock upgrades: done.
Starting file system checks:
Setting tty flags.
Setting sysctl variables:
Starting network.
Hostname: livekey
IPv6 mode: host
Configuring network interfaces:.
Building databases...
wsdisplay0: screen 1 added (80x25, vt100 emulation)
wsdisplay0: screen 2 added (80x25, vt100 emulation)
wsdisplay0: screen 3 added (80x25, vt100 emulation)
wsdisplay0: screen 4 added (80x25, vt100 emulation)
Mounting all filesystems...
Creating a.out runtime link editor directory cache.
Checking quotas: done.
/etc/rc: WARNING: No swap space configured!
Starting virecover.
Starting local daemons:.
Updating motd.
Thu May 11 17:31:51 UTC 2006

NetBSD/i386 (livekey) (console)

login:
```

3. AU REVOIR PRÉSIDENT

L'article se finit ici, mais vous avez certainement compris que c'est également ici que démarrent les insomnies, car il ne tient qu'à vous de créer autant de configurations de NetBSD « embarquées » que vous le souhaitez, de faire le ménage dans le basesystem, de retoucher, réécrire, repenser le `rc` basique !

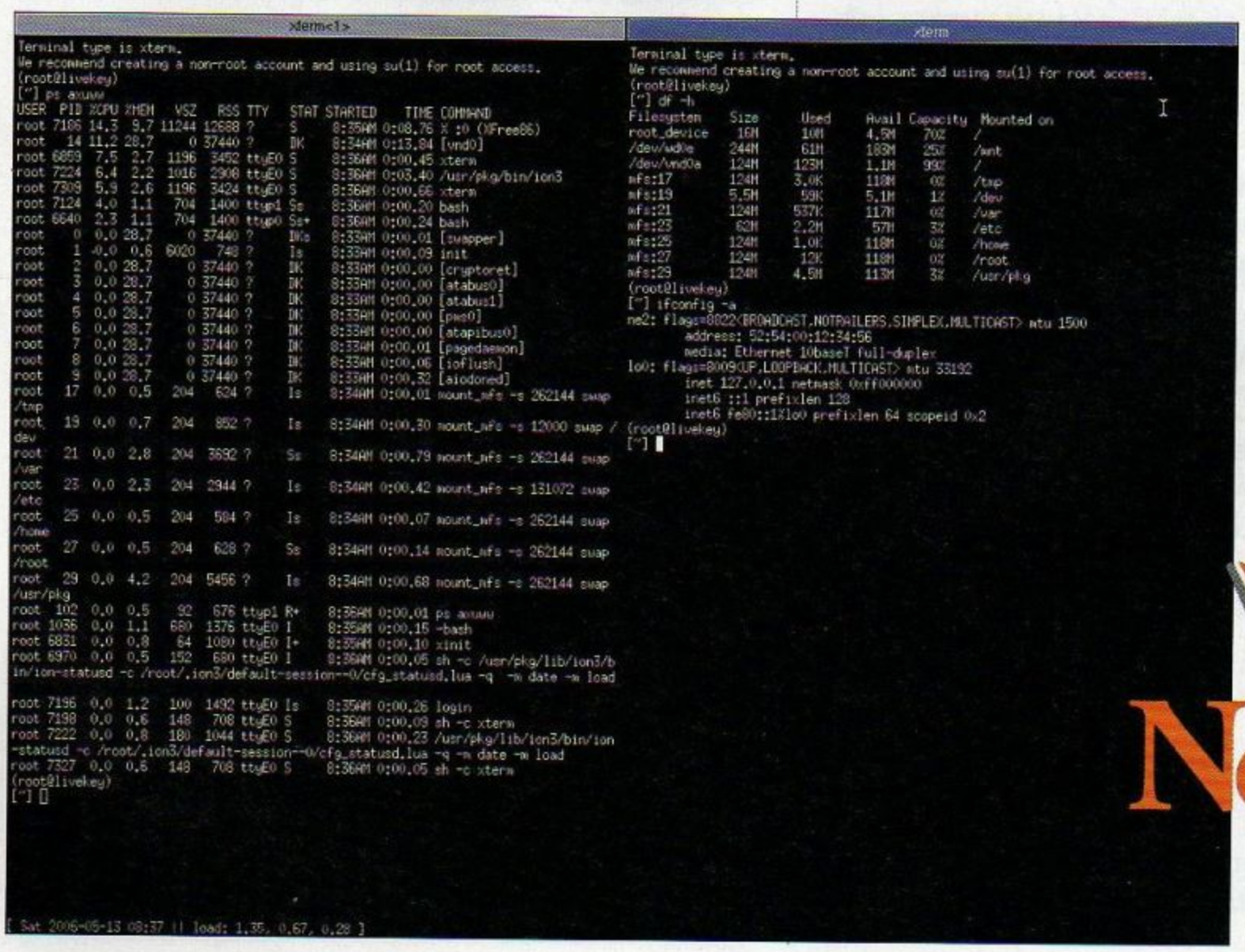
De plus, il reste dans notre image basique plusieurs dizaines de mégas inutilisés. Faites donc une petite sélection de packages (FTP : **ftp.netbsd.org/pub/NetBSD/packages/3.0/i386/**), puis à l'aide de `pkg_add -p`, installez-les dans votre environnement minimal. Attention, l'image compressée est en lecture seule. Il faudra travailler sur l'image décompressée puis regénérer le `.zimg` à l'aide de VNDcompress.

Vous trouverez quelques ajouts, ainsi que des scripts automatisant les étapes que nous avons vues à l'adresse suivante : **http://imil.net/nlk/**. L'`/etc/rc` de ce projet utilise par exemple un autre répertoire sur la clé afin de pouvoir modifier des fichiers de configuration classiques sans devoir reconstruire la clé. Il est également capable d'installer au boot des packages qui se trouvent dans le répertoire `livekey/` et il mounte `/root` en lecture écriture sur un répertoire situé sur la clé. Bien des améliorations sont imaginables, et il est sans aucun doute possible d'accélérer entre autres le processus de boot. Et comme d'habitude, commentaires et contributions sont les bienvenus :)

RÉFÉRENCES

Quelques liens connexes :

- **http://pkgsrc.se/sysutils/mklivecd**
- **http://www.yazzy.org/docs/NetBSD/netbsd-livecd.txt**
- **http://wiki.netbsd.se/index.php/How_to_build_your_own_NetBSD_LiveCD**
- **http://www.wifibsd.org/**



Landry « gaston » Breuil (gaston -at- gcu -dot- info), pour #gcu canal poisson-qui-pique, gruikifiying since '05

PF POUR LES NULS

Maman, j'ai peur d'internet, y'a que des pirates pédophiles nazis et des blogueurs du web 3.2alpha, ou « comment jouer au videur de boîte de nuit avec les paquets réseau ?». Miam, ça c'est de l'intro !

1. KESSESSÉ ?

1.1 UNE PETITE PAGE D'HISTOIRE

PF est né d'une histoire classique dans le monde du Logiciel libre, un problème de licence. A l'origine, OpenBSD utilisait depuis la création du monde (événement connu sous le nom de *la-grande-décision-de-théo*) un système de *firewalling*/filtrage nommé « IP Filter » (*aka* IPF), écrit par Darren Reed **[1]**. Cependant, un changement **[2] [3] [4]** dans la licence d'IPF, ainsi que plusieurs désaccords sur son architecture, aboutissent à l'éviction **[5]** de celui-ci d'OpenBSD peu après la sortie de la version 2.9 de l'OS. Cet événement donna par ailleurs aussi lieu au *grand-audit-des-licences-terreuses*.

OpenBSD se retrouva alors sans système de filtrage de paquets. C'est à ce moment qu'un semi-dieu, nommé Daniel-le-bienveillant (aussi connu sous le surnom de « l'homme aux longs cheveux qui venait du pays des horloges », ou plus communément dhartmei@) entreprit **[6] [7] [8]**, à partir d'un burin, un marteau et un bloc de granit, la création de ce qui allait devenir PacketFilter, un puissant système de filtrage de paquets avec suivi d'états (*stateful packet filter*) à la syntaxe claire comme de l'eau de roche. PF fut intégré avec la bénédiction de Théo-le-magnifique dans OpenBSD dès la version 3.0, et depuis a été porté dans la douleur **[9]** sur NetBSD 2.0, et a aussi remplacé **[10]** IPF dans FreeBSD à partir de la version 5.3 et dans DragonFlyBSD à partir de la version 1.2. PF *roulaize over da world* et *oune* IPF. Oups, désolé, je m'emporte, refermons la page historique ici.

1.2 ALORS DONC, ÇA FAIT QUOI CET ENGIN ?

PF est donc un filtre de paquets réseau fonctionnant en espace noyau, c'est-à-dire qu'il se place au niveau de la pile réseau et inspecte tous les paquets entrants et sortants de la machine, et en fonction de règles de filtrage va prendre la décision de :

- bloquer silencieusement le paquet (dans l'espace, personne n'entend les air-lutins crier) ;
- rejeter le paquet en envoyant un TCP RST à l'expéditeur (j'ai dit : « PAS DE BASKETS @#! ») ;
- laisser passer le paquet, en le marquant éventuellement au fer rouge pour post-traitement (Alors ça j'y mets là, ça j'y colle ici..).

PF utilise en plus de ce filtrage une table d'états des connexions établies, permettant de laisser passer les paquets d'une connexion existante sans qu'aucun traitement ne leur soit appliqué. Cette table permet aussi de faire des vérifications sur les numéros de séquence TCP, de vérifier qu'un ICMP *echo reply* correspond à un ICMP *echo request*...

Enfin, PF permet aussi (oui, madame Michu, ce n'est pas tout) de faire de la NAT (-ation ***dzim-boum*** calembour "- -" Mat`). Vous en aviez rêvé, vous allez pouvoir partager votre connexion internet avec tout votre réseau local en faisant de la traduction d'adresses (oui, je sais, ça vient de mon réseau local en adresse privée, mais il faut que ça aille fissa vers le grand nain Ternet et que ça revienne aussi vite à son maî-maître), faire de la redirection de ports, séduire votre voisine (hep, chérie, t'as vu tous les poils que j'ai mis dans cette règle de filtrage ? HaoOOon whoaaaa, prends-moi toute !!) et bien d'autres choses encore...

Pour faire tout cela, PF utilise un simple fichier texte pour sa configuration, par défaut `/etc/pf.conf`. Ce fichier contient une liste de règles. Nous reviendrons sur sa syntaxe après un intermède *trollifère*.

1.3 ET PAR RAPPORT À IPTABLES, ÇA VA LEVERAGER MON OUTSOURCING ?

Alors évidemment, je l'attendais la grande question. Oui, évidemment, PF surpasse Iptables, PF enfonce Iptables dans une taupinière et lui met des coups de bêche. Je ne sais même pas pourquoi je devrais me justifier. Objectivement :

- Sa syntaxe de configuration est beaucoup plus concise et logique : par exemple, pour rediriger ce qui arrive sur le port 80 extérieur vers le port 8080 d'une machine sur le réseau local, la règle PF serait :

```
rdr on $ext proto tcp from any to $ext port 80 -> \
    192.168.40.1 port 8080
```

L'équivalent Iptables :

```
iptables -t nat -A PREROUTING -p tcp -i eth0 --dport 80 \
    -j DNAT --to 192.168.0.2:8080
iptables -A FORWARD -p tcp -i eth0 -d 192.168.40.1 \
    --dport 8080 -j ACCEPT
```

- Il s'interfaçe nativement avec des outils de haute disponibilité (CARP/Pfsync) et de routage (OpenBGPD/OpenOSPFD).
- Il permet nativement de faire de la gestion de bande passante (QoS) avec `AltQ`.
- Il propose un proxy FTP, pendant en plus mieux bien de `ip_conntrack_ftp` chez NetFilter/Iptables.
- On peut faire de la création de règles à la connexion d'un utilisateur via `authpf`, utile pour forcer les utilisateurs à s'authentifier avant d'autoriser le trafic à passer. (Hop, je me connecte à distance sur mon firewall. Tant que je suis connecté, mon trafic perso passe. Hop, je me déconnecte, ça revient à la normale.)
- On peut faire de la reconnaissance d'OS (je redirige les Solaris vers tel serveur, les Windows XP vers Dave Null..).
- Tous les changements de règles au niveau du moteur de filtrage sont des opérations atomiques.
- dhartmei@ il a plein de poils, alors que les codeurs Iptables, ils en ont une toute petite. NDLR : ce commentaire n'engage que l'auteur de l'article :)
- et j'en passe. En gros, ce sont deux projets ayant le même but, mais qui se basent sur des conceptions et idées différentes. Chacun voit midi à sa porte, YMMV, toussa. T'façon, j'y connais keudalle en Iptables.

La seule différence notable au niveau de la configuration par rapport à Iptables se trouve dans l'ordre d'interprétation des règles : par défaut dans PF, la dernière règle qui correspond au paquet emporte la décision (« *last matching rule wins* »), alors que pour Iptables l'analyse des règles s'arrête à la première correspondant au paquet. Cependant, nous verrons que le mot clé `quick` permet de passer outre ce comportement.

Par la suite, mes exemples prendront comme hypothèse que PF est sur une machine fonctionnant comme une passerelle entre internet (interface réseau externe) et un réseau local (interface réseau interne), mais, bien évidemment, on peut utiliser PF sur une machine personnelle avec une seule interface réseau, et aussi sur un serveur avec 3/4/2142 interfaces, comme par exemple une zouliiie Soekris. Par flemme, j'utiliserai aussi l'anglicisme « *matcher* une règle », lorsqu'un paquet correspond aux critères d'une règle. Oui, traitez-moi de feignant, j'aime ça.

2. COMMENT QUE ÇA MARCHE LE BOUZIN ?

2.1 QUOI, UN BÊTE FICHIER TEXTE POUR LE CONFIGURER ?

La configuration de PF se fait donc via `/etc/pf.conf`, dont la syntaxe est simple : les lignes vides sont ignorées, les lignes commençant par un `#` sont des commentaires et `$variable` est remplacé par la valeur de `variable`. Ce fichier est principalement (je ne rentre volontairement pas dans les détails) composé de 5 sections.

2.1.1 Les macros et listes

On peut définir des macros pour remplacer des variables revenant souvent, ou par souci de clarification : ports, noms d'interfaces, adresses IP...

```
if_ext="rl0"
webserver="192.168.1.10"
```

Les listes permettent, quant à elles, de rassembler des critères dans une variable :

```
block out on fxp0 proto tcp from \
    { 192.168.0.1, 10.5.32.6 } to any port { 22 80 }
```

Cette liste sera « déroulée » lors du chargement de la règle correspondante, donnant ici lieu à quatre règles avec chacune des combinaisons des valeurs des listes. La virgule entre les valeurs d'une liste est facultative.

On peut, bien évidemment, mixer les macros et les listes, et écrire de jolies choses :

```
if_ext="rl0"
trusted = "{ 192.168.1.2 192.168.5.36 }"
#...
pass in on $if_ext inet proto tcp from \
    { 10.10.0.0/24 $trusted } to port 22
```

2.1.2 Les tables

Elles permettent de stocker un grand nombre d'adresses (50 ou 50000, même combat), et sont ensuite utilisées directement dans les règles de filtrage/NAT/redirection. La recherche d'une adresse dans une table en mémoire est beaucoup plus rapide, et moins coûteuse en temps processeur/mémoire que la recherche dans un jeu de règles correspondant chacune à une valeur d'une liste d'adresses.

Le mot-clé `const` est utilisé lorsqu'on veut que la table ne puisse pas être modifiée, et `persist` indique à PF de ne pas supprimer une table non référencée par une règle. L'avantage d'une table non `const` par rapport aux listes est que l'on peut ajouter/supprimer à la volée des adresses ou sous-réseaux dans une table, utile pour bloquer un certain temps l'adresse d'un *spammeur* ou d'un *script-kiddie* par exemple, ou gérer la redirection vers un ensemble de serveurs faisant de la haute disponibilité.

Enfin, cerise sur le gâteau, on peut initialiser une table avec un fichier contenant une liste d'adresses.

```
table <privateip> const \
    { 192.168.0.0/16, 172.16.0.0/12, 10.0.0.0/8 }
table <spammers> persist file "/etc/spammers"
#...
block in on fxp0 from { <privateip>, <spammers> } to any
...
```

```
root@spud[~]#pfctl -t spammers -T add 218.70.0.0/16
root@spud[~]#pfctl -t spammers -T delete 218.70.5.12
```

Les tables, c'est VIE, AMOUR et POIL QUI BRILLE aflab.

2.1.3 Les règles de NAT

Basiquement, la NAT fait de la traduction d'adresses, c'est-à-dire qu'on modifie les paquets provenant d'un réseau en adressage privé pour qu'ils puissent transiter sur Internet. PF va remplacer l'adresse source (privée) dans un paquet sortant par l'adresse publique de la passerelle, et garder dans une table de correspondance la paire ip:port source + ip:port destination. Ainsi, quand on reçoit la réponse correspondant à ce paquet sortant, on remplace l'adresse destination (qui était notre adresse publique) par l'adresse privée de la machine ayant initié la connexion. Ce mécanisme marche bien entendu pour TCP, UDP ou ICMP, même si ces deux derniers sont des protocoles sans état. C'est bon, j'ai perdu personne ?

Tout d'abord, pour activer le *forwarding* de paquet, il faut mettre le sysctl `net.inet.ip.forwarding` à 1, via la commande `sysctl` ou le fichier `/etc/sysctl.conf` si on veut que le changement soit persistant. De plus, il ne faut pas oublier que les paquets vont passer à travers le filtre de paquets après avoir été modifiés, sauf si l'on utilise le mot-clé `pass`. Et ceci s'applique aussi à RDR que nous verrons juste après.

La syntaxe générique d'une règle de NAT est globalement la suivante :

```
nat [pass] on interface [address_family] from src_addr \
    [port src_port] to dst_addr [port dst_port] -> ext_addr
```

Je décortique ces hiéroglyphes : ce qui est entre crochets est facultatif, ce qui est en bleu est variable.

- `nat` => c'est une règle de NAT.
- `pass` => le paquet est NATé, et envoyé directement sans passer par le filtre de paquets.
- `on interface` => le paquet est arrivé sur telle interface réseau (rl0, ne3, $ext_if, ppp0…)
- `address_family` => inet ou inet6, c'est un détail qui peut avoir son importance.
- `from src_addr` => le paquet provient de telle adresse. Ici, pour l'adresse, on peut spécifier plein de trucs :
  - une adresse simple en notation IPv4/IPv6 : 10.246.200.1
  - une plage d'adresses en notation CIDR : 162.168.10.0/24
  - un nom de domaine, il sera résolu par PF lors du chargement des règles.
  - une interface réseau : elle correspondra à toutes les adresses qui lui sont attribuées.
  - une table ou une liste, comme expliqué un peu plus haut.
  - n'importe laquelle de ces notations, précédée d'un `!` pour signifier la négation.
  - et finalement le mot-clé magique `any`, pour n'importe quelle adresse.
- `port src_port` => si on veut NATer uniquement un certain port ou une plage de ports... rarement utilisé.
- `to dst_addr` => le paquet est destiné à telle adresse. Mêmes possibilités que pour `src_addr`.
- `-> ext_addr` => remplacer l'adresse source par cette adresse. Le retour se fera tout seul par magie. Et si cette adresse change (attribuée par DHCP), on peut ici spécifier le nom de l'interface réseau entre parenthèses (rl0), et l'adresse sera automatiquement mise à jour dans la règle. Ca, c'est **hooOOaaNnn**.

Donc, pour partager simplement votre connexion internet avec tout votre réseau local, il suffit d'une seule règle, par exemple (ne3 est ma carte réseau côté local, 10.246.200.0/24 est la notation CIDR de mon réseau local, et rl0 la carte réseau côté externe, branchée sur un modem en IP dynamique) :

```
nat on rl0 inet from ne3 : network to any -> (rl0)
```

C'est-ty-pas beau ? Vous vous rappelez le bordel que c'est pour faire la même chose avec Iptables ?

2.1.4 Les règles de redirection de paquets

RDR est ici le petit frère caché de la NAT, à savoir qu'il fait tout l'inverse de son grand-frère : il prend les paquets arrivant de l'extérieur de la passerelle, et les redirige vers une machine du réseau local. Ça peut être très pratique quand on a un serveur web sur une machine de son réseau local, et qu'on veut qu'il soit directement accessible de l'extérieur, mais aussi dans des exemples plus pragmatiques comme « faire marcher le transfert de fichier avec Jabber », ou « rediriger un certain trafic venant de la Freebox vers mon Laptop pour pouvoir se 'loutrifier' devant la télé au pieu ». Oui, je sais. Honte, flemme, toussa.

Maintenant, le comment-kon-fait. La syntaxe générique d'une règle de RDR est globalement la suivante :

```
rdr [pass] on interface [address_family] from src_addr
    [port src_port] to dst_addr [port dst_port]
    -> int_addr [port int_port]
```

C'est la même syntaxe que pour la NAT, à quelques petits détails près, du genre « je redirige ce qui était à destination de `dst_addr:dst_port` vers `int_addr:int_port`, et, comme pour la NAT, le retour se fera tout seul comme par magie.

Hop hop, un petit exemple, disons que je veux accéder de l'extérieur au service SSH d'un serveur de mon réseau local, mais en gardant l'accès SSH à ma passerelle. Je choisis un autre port extérieur (mettons 34522, oui j'aime me compliquer la vie à trouver des numéros de ports que je suis sûr d'oublier au moment où j'aurais besoin de m'en souvenir) et je redirige ce qui arrive sur ce port vers mon serveur (`$external` est le nom de mon interface réseau externe, et `$diane` est l'adresse privée du serveur sur le réseau local).

```
rdr pass on $external proto tcp to port 35422 -> $diane \
    port ssh
# et j'accède au serveur de l'extérieur par :
ssh -p 35422 my.gateway.info
```

Au passage, vous remarquerez que j'ai nommé directement le port ssh, et non 22. Oui, on peut utiliser tous les petits noms de `/etc/services` à chaque fois qu'on veut désigner un port.

Bon évidemment, mal foutu, tout ce genre de bricolage peut avoir des conséquences dramatiques sur la sécurité de votre réseau local, car la machine vers laquelle on redirige ce trafic (qui n'a pas forcément de firewall, ELLE) est directement accessible de l'extérieur sur ce port. Vous iriez vous balader tout nu dans la neige un 31 décembre avec 5 grammes d'alcool dans le sang ? Moi, non (Quoique, bref, *__ahem__*, passons !). Généralement, on met les machines auxquelles on peut accéder de l'extérieur de cette manière dans une DMZ, avec une politique de filtrage bien pensée. Bref, on fait attention à ce qu'on fait, **rognnntudju** !

D'ailleurs, houba houba hop, passons à la susdite politique de filtrage.

2.1.5 Les règles de filtrage

Comme dit précédemment, les règles de filtrage sont évaluées de façon séquentielle de la première à la dernière (du haut vers le bas dans les fichiers de règles utilisés). Ce qui veut dire que chaque paquet va être évalué par chaque règle, et la dernière règle correspondant au paquet emporte la décision (`block` ou `pass`). Au contraire, si on utilise le mot-clef `quick`, l'évaluation s'arrête dès qu'une règle correspond au paquet. La première règle (implicite) est un « tout laisser passer » de sorte que si aucune règle n'est applicable à un paquet, celui-ci est accepté, c'est pourquoi généralement la première règle explicite est un `block all`. On va pas laisser rentrer n'importe qui non plus.

Allez hop, comme pour les autres, on déshabille la syntaxe générale d'une règle :

```
action [direction] [log] [quick] [on interface] \
    [address_family] [proto protocol] \
    [from src_addr [port src_port]] \
    [to dst_addr [port dst_port]] \
    [flags tcp_flags] [state]
```

Bon, quoi de neuf là-dedans qu'on n'ait pas déjà vu... ?

- `action` => au choix, `block` ou `pass`. J'ai pas besoin de vous faire un dessin ? Ah si, un détail, la politique pour les paquets bloqués sera `drop` ou `return` en fonction de l'option `block-policy`. Par défaut, c'est `drop`.
- `direction` => `in` ou `out`, si on veut filtrer le trafic entrant ou sortant de l'interface. Si on ne met rien, la règle sera évaluée pour les deux sens.
- `log` => si ce flag est présent, on enregistre la décision prise pour cette règle concernant le paquet. Pour analyser ceci, `pflog` sera notre ami. Nous ferons sa connaissance un peu plus loin.
- `quick` => j'en ai déjà parlé, si ce flag est aussi présent et que le paquet matche la règle, il ne sera plus analysé/trituré, et la décision prise par cette règle sera notre dernier mot, Jean-Pierre.
- `proto protocol` => un protocole de niveau 4, généralement `tcp`, `udp` ou `icmp`, mais on peut aussi rencontrer n'importe quel protocole de niveau 4 référencé dans `/etc/protocols`. Si on veut faire un peu d'*obfuscation*, on peut même l'appeler par son petit numéro même.
- `port dst_port` => petite précision que j'avais oubliée tout à l'heure, emporté par mon élan fougueux, on peut ici spécifier un intervalle complexe de ports avec les opérateurs <, <=, >=, >, <>, <> et :, voyez avec l'ami `man pf.conf` pour plus de détails. Vive les hiéroglyphes, merci Spontex.
- `flags tcp_flags_check/mask` => on peut spécifier des vérifications supplémentaires sur les drapeaux d'un paquet TCP, par exemple pour traiter les ouvertures de session TCP. On utilise souvent `flags S/SA` que je traduirais par « cette règle s'applique sur les paquets TCP qui ont, sur les deux drapeaux SYN et ACK, seulement SYN positionné ». Si les 2 drapeaux sont positionnés, le paquet ne matchera pas la règle. Pour les autres flags, RTFM un peu.
- `state` => ici, on utilise généralement deux possibilités : `keep state` ou `synproxy state`. La première est utilisée quand on veut créer une entrée dans la table des états de connexions lorsqu'un paquet matche la règle, et appliquer la même politique aux paquets suivants prenant part à la connexion. Tous ces paquets sont donc rattachés à cette entrée, et on peut aussi du coup vérifier que la séquence des paquets TCP est bien respectée. `synproxy state` est utilisé quand on veut que PF joue le rôle d'un proxy TCP pour l'établissement d'une connexion. Dans ce cas, PF va traiter la demande en lieu et place du destinataire et ne transférera qu'ensuite les paquets à ce dernier. Aucun paquet n'est transmis au destinataire avant que le client n'ait terminé l'échange initial. Cette technique permet de protéger le destinataire des attaques de type TCP SYN flood, lorsqu'on demande un grand nombre d'ouvertures de connexions en vue de provoquer un déni de service.

Ça commence à devenir un peu indigeste tout ça, donc on va voir tout de suite deux exemples simples de règles :

```
pass in quick on $external inet proto tcp from any to any \
    port { http, https, smtp, imaps } flags S/SA keep state
```

En français, j'autorise tous les paquets TCP/IPv4 arrivant sur l'interface externe à destination des ports **http/https/smtp/imaps** à passer. Je vérifie que ce sont des ouvertures de connexion TCP, j'enregistre leur état dans la table, et j'arrête l'analyse de ces paquets sur cette règle (`quick`). Plutôt simple, non ?

```
block in log on $external from \
    { 192.168.0.0/16, 172.16.0.0/12, 10.0.0.0/8 } to any
```

Ici, je bloque les paquets arrivant sur l'interface externe avec une adresse source privée (et donc non *routable*), et j'enregistre les informations du paquet bloqué. Ça permet de parer certaines attaques de type *spoofing*, où on envoie des paquets usurpés dans le but d'induire en erreur les équipements réseau (pour plus d'informations sur cette technique, se référer à la RFC 2827 **[11]**)

Bon évidemment, il y a encore pleins d'options, de détails et de particularités (comme les ancres, le `scrubbing`, l'`antispoofing`...) dont je n'ai pas parlé (et que je maîtrise pas, un peu d'humilité n'a jamais fait de mal). Pour plus d'informations, il faudra se reporter à la documentation officielle de `pf.conf` **[12]**. Attention, ça peut faire mal au crâne.

2.2 J'EN FAIS QUOI DE CE PF.CONF MAINTENANT ?

L'interface principale côté utilisateur de PF est `pfctl`. Avant toute chose, il faut « activer » le *device* PF dans le noyau, soit manuellement via `pfctl -e`, soit en ajoutant `pf=YES` dans le fichier de configuration global du système lu lors du démarrage de la machine (généralement `/etc/rc.conf.local`). Si on veut faire de la NAT/RDR, ne pas oublier d'activer cette option dans le noyau via la commande `sysctl net.inet.ip.forwarding=1`, ou en ajoutant/décommentant la ligne `net.inet.ip.forwarding=1` dans `/etc/sysctl.conf` pour rendre ce changement persistant.

Tout d'abord, on va se pencher sur les principales options de `pfctl` permettant de configurer PF :

- `-d` : désactive PF, l'opposé de `-e`. Une fois PF désactivé, plus rien n'est filtré/natté, les paquets transitent directement dans la pile réseau du noyau.
- `-f fichier` : le fichier de configuration à charger. (`/etc/pf.conf` par défaut)
- `-n` : ne pas réellement charger les règles, faire juste une analyse syntaxique du fichier.
- `-o` : activer les options d'optimisations. PF va réordonner les règles et supprimer les doublons pour optimiser le jeu de règles.

Ensuite, les options permettant de modifier à la volée l'état de PF :

- `-T` (`kill/flush/add/delete/show/test/..`) : utilisé conjointement avec `-t table`, permet de manipuler une table : la supprimer, la vider, ajouter une adresse, en supprimer une, l'afficher, vérifier qu'une adresse est dans la table... Exemple : `pfctl -t blocked-hosts -T show` va m'afficher les adresses de toutes les machines ayant été ajoutées dans la table `blocked-hosts`, déclarée un peu plus haut dans mon `/etc/pf.conf`.
- `-F` (`nat/rules/state/Tables/..`) : remet respectivement à zéro les règles de NAT, les règles de filtrage, les états des connexions ouvertes, ou les tables. Pratique, si on veut faire un peu de ménage, remettre à zéro des compteurs ou des connexions, désactiver la NAT, supprimer toutes les entrées de toutes les tables, etc..
- `-k` (`host/network`) : Permet de tuer toutes les entrées dans la table d'état concernant les connexions venant d'une machine/d'un réseau. Si on utilise deux fois cette option, on supprime les états des connexions venant de la première adresse vers la deuxième. Exemple : `pfctl -k 192.168.1.0/24 -k 172.16.0.0/16` supprimera tous les états des connexions entre ces deux sous-réseaux.

Enfin, l'option permettant d'obtenir beaucoup d'informations sur le statut de PF, j'ai nommé `-s modifieur`, à utiliser conjointement avec `-r` si on veut que PF fasse des *reverse-dns lookup* pour les adresses qu'il affiche. Les valeurs les plus intéressantes pour `modifieur` sont `rules`, `nat`, `state`, et `info`, pour afficher respectivement les règles de filtrage chargées en mémoire, celles de NAT, les connexions ouvertes, et des statistiques globales sur PF (`all` permettra d'afficher tout ce que PF a à nous dire, tu vas les cracher ces infos 'spèce de balance @"!@#%*$!?). Hop hop, quelques exemples valent mieux qu'un long blabla :

```
root@spud[~]#pfctl -s nat
nat on ne3 inet from 10.206.210.0/24 to any
    -> 82.245.152.88
rdr pass on ne3 inet proto udp from 212.27.38.253 to any
    -> 10.206.210.174
rdr pass on ne3 inet proto tcp from any to any port = 1720
    -> 10.206.210.175
....


root@spud[~]#pfctl -s rules
block drop in quick from <blocked-hosts> to any
block drop all
pass quick on lo all
pass quick on rl0 all
pass in quick on ne3 inet proto tcp from any to any
    port = www flags S/SA keep state
pass in quick on ne3 inet proto tcp from any to any
    port = https flags S/SA keep state
pass in quick on ne3 inet proto tcp from any to any
    port = smtp flags S/SA keep state
pass in quick on ne3 inet proto tcp from any to any
    port = imaps flags S/SA keep state
```

```
pass out quick on ne3 inet proto tcp all flags S/SA
    keep state
pass out quick on ne3 inet proto udp all keep state
pass out quick on ne3 inet proto icmp all keep state
....

root@spud[~]#pfctl -r -s state
all tcp network.org:24680 -> anthony.freenode.net:6667
    ESTABLISHED:ESTABLISHED
all tcp renton.network.org:39832 -> network.org:62328
    -> jabber.freenet.de:5222  ESTABLISHED:ESTABLISHED
all tcp renton.network.org:33719 -> network.org:62794
    -> 205.188.7.181:5190   ESTABLISHED:ESTABLISHED
all tcp renton.network.org:55564 -> network.org:56919
    -> zone6.gcu-squad.org:80   FIN_WAIT_2:FIN_WAIT_2
all tcp renton.network.org:55565 -> network.org:62487
    -> zone6.gcu-squad.org:80   FIN_WAIT_2:FIN_WAIT_2
all tcp network.org:80
    <- sek76-2-82-245-144-239.fbx.proxad.net:1143
    FIN_WAIT_2:FIN_WAIT_2
all tcp network.org:80
    <- sek76-2-82-245-144-239.fbx.proxad.net:1161
    TIME_WAIT:TIME_WAIT
all tcp network.org:13150 -> pop.free.fr:110
    TIME_WAIT:TIME_WAIT
all tcp network.org:11892 -> firewall-services.com:995
    ESTABLISHED:ESTABLISHED
all udp network.org:26191 -> dns1.proxad.net:53
    MULTIPLE:MULTIPLE
all tcp network.org:33799 -> network.org:60594
    -> anelis.isima.fr:22   ESTABLISHED:ESTABLISHED
....

root@spud[~]#pfctl -s info
Status: Enabled for 99 days 08:19:47          Debug: Urgent

Interface Stats for ne3              IPv4             IPv6
  Bytes In                   110259968736                0
  Bytes Out                   99321680976              352
  Packets In
    Passed                      174547989                0
    Blocked                       1075543                0
  Packets Out
    Passed                      168492615                1
    Blocked                        26376                4

State Table                         Total             Rate
  current entries                       9
  searches                      691697504            80.6/s
  inserts                         8159941             1.0/s
  removals                        8159932             1.0/s
....
```

Pour toutes ces commandes, on peut utiliser `-v` pour rendre PF un peu plus bavard, voire `-vv` et même `-g` et `-x` pour le passer en mode debug, auquel cas il devient aussi saoulant qu'un vieil oncle ayant pris un peu trop d'apéros lors d'un repas de famille. L'option `-q` pourra alors permettre de lui faire fermer un peu sa gu...
Bien sûr, je n'ai pas la prétention de vous expliquer tout `pfctl`, la documentation de référence sera toujours la page de manuel officielle **[13]**. Voilà, avec ça je pense que vous êtes en mesure de passer aux exemples complets. Attention, interrogation écrite après cette section.

3. J'AI RIEN COMPRIS, DESSINE-MOI UN MOUTON

Tout d'abord, un premier exemple simple, voici notre cahier des charges (f34r d4 8uZzW0rD) :

- Nous avons une machine connectée à internet via une interface réseau en IP fixe, nommons-la `bge0` pour l'exemple.
- Elle doit être accessible de l'extérieur en ssh à partir de trois machines dites « de confiance ».
- Son serveur Apache doit être accessible à tous.
- On veut qu'elle réponde aux *pings* de l'extérieur.
- On veut pouvoir accéder à internet à partir de cette machine, pratique pour lire de la doc (ou pompazer de la bonne stuffaize (**kh**)).
- Enfin, on bloquera silencieusement tous les autres paquets.

Hop, toutes ces règles se traduisent par ce `/etc/pf.conf` :

```
# Notre interface réseau
iface = "bge0"
# Les machines auxquelles on fait confiance,
# elles seront autorisées à se connecter via ssh.
trusted_hosts = "{ 131.25.4.12, 88.12.74.5, 207.124.20.9 }"

# On ne s'occupe pas de l'interface de loopback, utilisée
# par plusieurs services internes à la machine.
set skip on lo

# On active la normalisation de paquets en entrée. Pf
# va réassembler les paquets fragmentés et faire des
# vérifications supplémentaires dessus.
scrub in all

# Par défaut, on bloque tout les paquets.
block all

# On autorise les paquets icmp de type echo request pour
# les pings venant de l'extérieur, et echo reply/time
# exceeded/destination unreachable pour les réponses aux
# pings que l'on avait initié vers l'extérieur.
pass in inet proto icmp from any to $iface icmp-type \
    { echoreq, echorep, timex, unreach }

# On autorise les connexions au serveur Apache
# et on les enregistre dans la table d'état.
pass in inet proto tcp from any to $iface port www \
    flags S/SA keep state

# On autorise les connexions au service sshd, uniquement \
# en provenance des machines auxquelles on fait confiance.
pass in inet proto tcp from $trusted_hosts to $iface \
    port ssh flags S/SA keep state

# Enfin on autorise tout le trafic sortant.
pass out inet proto tcp  from $iface to any flags S/SA \
    keep state
pass out inet proto { udp, icmp } from $iface to any \
    keep state
```

Un coup de `pfctl -e -f /etc/pf.conf` et voilà, avec cette configuration notre machine est protégée ! Certaines règles sont assez strictes (généralement, on met `all` à la place de `from any to $iface`, on laisse passer tous les paquets ICMP, on utilise `quick` à outrance...), mais le filtrage que l'on désirait est en place. Faites une pause, et réfléchissez à ce qu'il aurait fallu faire pour obtenir la même chose avec Iptables. Un peu plus mieux bien, non ?

Mais je suppose que ça ne vous suffit pas (bande de petits coquins, il vous en faut toujours plus.). Passons donc à un exemple un peu plus évolué.

4. ET SI J'AI ENVIE D'EN COLLER SUR LES MURS ET AU PLAFOND ?

Donc oui, maintenant on en vient à l'exemple que vous attendez tous. Comment faire une passerelle tout en un qui protège notre réseau local des attaques extérieures, permette de sortir couvert sur l'intairnaitte (si si, Roger et Lucette, ils l'appellent comme ça) plein de méchants, et tant qu'à faire fasse marcher des trucs convis qui utilisent des ports exotiques. Non, elle ne fera pas le café, sauf si vous achetez l'adaptateur kivabien.

On récapitule ce qu'on veut en bon françois des familles :

- Notre passerelle (`renton`) dispose de deux interfaces, `rl0` côté internet et `ne3` côté réseau local.
- La première possède une IP fixe (comme chez la plupart des fournisseurs d'accès internet, encore faut-il être dégroupé...)
- La seconde interface côté réseau local est configurée avec l'adresse 192.168.1.1 sur le réseau local en 192.168.1.0/24.
- On protégera tout ce beau monde de l'extérieur, et comme on est parano, on fera un peu de *logging*.
- On veut que les machines de notre réseau local aient accès à internet sans limitation.
- Une machine particulière (`diane`) devra être accessible via ssh et HTTPS. (Oui, j'ai la flemme de la mettre dans une DMZ, *blame on me*.)
- On voudra accéder au multiposte de Free sur une machine (`sickboy`), c'est juste un flux RTSP/UDP classique.
- Notre machine principale (`tommy`) aimerait bien faire marcher des trucs clicka-compliants, comme le transfert de fichiers via Jabber.
- Enfin, on aimerait bien pouvoir accéder à notre passerelle de l'extérieur en ssh, des fois qu'une urgence se présente.

Vous êtes prêt ? Allez hop, on se jette dans un nouveau `/etc/pf.conf` !

```
# /etc/pf.conf, seconde édition
# - disclaimer - si cette configuration fait fondre
# votre passerelle, je décline toute responsabilité.

# Déclaration des interfaces réseau
ext="rl0"
int="ne3"

# Déclaration des ports à ne pas logguer
# (5900 est le port utilisé par VNC)
ports_not_logged = "{ netbios-ssn, microsoft-ds, \
    epmap, ms-sql-s, 5900}"

# Déclaration des hôtes sur mon réseau local
diane = "192.168.1.2"
tommy = "192.168.1.20"
sickboy = "192.168.1.21"

# Déclaration du port utilisé à l'extérieur pour accèder
# au sshd de diane
ssh_diane = "65322"

set skip on lo
scrub in all

# Tout d'abord, la règle principale de NAT pour le réseau
# local. Ici, je ne mets pas 'pass' car après j'autorise
# explicitement tout le trafic sortant. Le suffixe :network
# est utilisé pour dire 'le sous-réseau correspondant à
# l'adresse de cette interface'
nat on $ext from $int:network to any -> $ext

# Ensuite, les règles de redirection. J'ai mis le mot-clé
# 'pass' pour ne pas faire de filtrage supplémentaire sur
# ces connexions, sinon il aurait fallu rajouter les ports
# correspondants dans la règle de filtrage sur le trafic
# venant de l'extérieur.

# Le flux rtsp du multiposte free est de l'udp provenant
# de freeplayer.freebox.fr
rdr pass on $ext proto udp from 212.27.38.253 -> $sickboy

# On redirige le port 8010 utilisé par Jabber pour
# l'ouverture des transferts de fichiers entrants
rdr pass on $ext proto tcp to port 8010 -> $tommy

# Enfin on redirige le ssh (sur le port particulier utilisé
# à l'extérieur) et le https vers diane
rdr pass on $ext proto tcp to port $ssh_diane \
    -> $diane port ssh
rdr pass on $ext proto tcp to port https -> $diane

# Petite nouveauté, on active l'antispoofing sur l'interface
# externe : cette règle bloquera les paquets venant de
# l'extérieur essayant d'utiliser frauduleusement notre
# adresse pour passer à travers le filtre.
antispoof for $ext

# On ne filtre pas les paquets sur l'interface interne.
pass quick on $int

# Maintenant, les règles de filtrage...
# Par défaut, on bloque et on loggue tout les paquets
# venant de l'extérieur.
block in on $ext log all
```

```
# On n'a pas envie de remplir nos logs en 5 minutes avec les
# quelques vers bien connus qui trainent sur internet
# donc on bloque certains paquets sans les logguer.
block in on $ext inet proto tcp from any to any \
    port $ports_not_logged

# On autorise les pings en provenance de l'extérieur
pass in on $ext inet proto icmp from any to any icmp-type \
    { echorep, echoreq, timex, unreach }

# On autorise le ssh venant de l'extérieur
pass in on $ext inet proto tcp from any to any port ssh \
    flags S/SA keep state

# On autorise tout le trafic sortant (le trafic NATé du \
# réseau local passera par ces règles)
pass out on $ext inet proto tcp all flags S/SA keep state
pass out on $ext inet proto { udp, icmp } all keep state
```

Alors, heureux ? Ça vous paraît pas déjà plus clair avec quelques exemples ? Évidemment, il y aurait encore beaucoup à dire sur le sujet, mais je vous laisserai le plaisir de découvrir toutes les subtilités jouissives de la configuration d'un PF à 5h30 du matin, les yeux rougis par l'écran, les mains rendues tremblantes par la surdose de café (ou de bière, belge de préférence), l'aube dardant ses rayons d'argent par la fenêtre, alors que vous vous dites « C'est beau, une ville, la nuit. ». Rhalala, quelle poésie, on dirait du Pinpin.
Tiens, je m'apprêtais à prendre congé de vous à regret, mais je m'aperçois que nous avons utilisé un mot-clé dont je n'ai pas encore parlé en long et en large : `log`. Houba Hop, fissa, *next section* !

5. ET COMMENT JE FAIS JOUJOU À LA NSA ?

Donc, quand PF a envie de cafter un peu, il va envoyer des informations sous un format binaire (histoire que ça soit plus rigolo, c'est du PCAP/TCP*dump* standard) sur une pseudo-interface réseau (`pflog0`), et un de ses bons potes, j'ai nommé `pflogd`, va stocker tout ça dans le fichier `/var/log/pflog`. Puis, nous aurons la joie de faire la connaissance d'un autre larron, nommé `tcpdump`, qui se chargera de traduire toutes ces informations en français académique. Si, si, je vous le jure, presque académique.

Tout d'abord, il faut activer/lancer le démon `pflogd`. Normalement, il doit se lancer tout seul si PF est activé au démarrage de la machine. Si ce n'est pas le cas : `ifconfig pflog0 up && pflogd`.

On vérifie que tout marche bien :

```
root@spud[~]#ps waux | grep pflog
root     29220  0.0  0.1   448   396 ??  Is    10Sep06
    0:00.02 pflogd: [priv] (pflogd)
_pflogd  13046  0.0  0.1   512   280 ??  S     10Sep06
    12:36.73 pflogd: [running] -s 116 -f /var/log/pflog
    (pflogd)
root@spud[~]#ifconfig pflog0
pflog0: flags=141<UP,RUNNING,PROMISC> mtu 33224
```

Maintenant que PF cause sur `pflog0` lorsqu'un paquet matche une règle où le mot-clé `log` est utilisé, passons à `tcpdump`. On peut l'utiliser selon deux modes :

- interactif : `tcpdump -i pflog0`. Il ira directement lire ce qui passe en direct sur `pflog0`, `pflogd` sera donc inutile.
- passif : `tcpdump -r /var/log/pflog`. Il lira ce qui a été enregistré par `pflogd` dans son fichier de sortie.

Un exemple sera peut-être plus parlant :

```
root@spud[~]#tcpdump -qea -ttt -i pflog0
Dec 24 18:56:42.832166 rule 39/(match) block in on ne3:
    dslb-088-073-124-180.pools.arcor-ip.net.46554
    > my.network.org.2967: [|tcp] (DF)
Dec 24 18:56:46.077820 rule 39/(match) block in on ne3:
    dslb-088-073-124-180.pools.arcor-ip.net.46554
    > my.network.org.2967: [|tcp] (DF)
Dec 24 18:59:48.917072 rule 39/(match) block in on ne3:
    62.160.212.130.4376 > my.network.org.5900: [|tcp] (DF)
```

On peut aussi passer une expression à `tcpdump` pour qu'il filtre sa sortie selon des critères précis :

```
root@spud[~]#tcpdump -ttt -r /var/log/pflog port 80 and \
    host 192.168.1.2
```

Enfin, `tcpdump` comprend aussi la syntaxe de configuration de PF. On peut donc lui demander des choses de ce style :

```
root@spud[~]#tcpdump -ttt -i pflog0 inbound and action \
    pass and on wi0
```

Avec cette commande, il n'affichera que les paquets autorisés à passer, logués et entrants sur l'interface `wi0`.

Évidemment, `tcpdump` propose des dizaines d'options intéressantes, comme l'affichage hexadécimal des paquets filtrés, la reconnaissance de système d'exploitation, et bien d'autres encore. Pour toutes ces possibilités qui font mal au crâne, la page de manuel **[14]** sera la référence (qui a dit « comme d'habitude » ?).

Tiens, d'ailleurs, parlons vite fait de cette fameuse reconnaissance de système d'exploitation, aka « *passive os fingerprinting* ». PF utilise pour cela des parties de `p0f` **[15]**, un outil écrit par Michal Zalewski, encore un hacker de génie venu des pays de l'Est.

Je ne m'étendrai pas sur ce sujet, sachez juste qu'il existe des techniques pointues d'analyse de paquets d'ouvertures de session TCP, de calcul de temps de réponse, de vaudou thaïlandais et de chamanisme à base d'incantations telles que « LE CODE EST LIBRE, LE CODE EST BEAU » ou « LA BOOOONNNE PAROOOOLE », tous ces gris-gris permettent d'avoir une bonne idée du système d'exploitation qu'utilise la machine avec laquelle on communique.

PF peut utiliser ces techniques à deux niveaux : on peut ajouter le mot-clé `os` à une règle de filtrage, auquel cas un paquet ne matchera la règle que si l'analyse du paquet laisse à penser que l'OS distant est celui que l'on désire filtrer. Exemple :

```
block in on $ext proto tcp from any \
    os {"Windows 95", "Windows 98"} to any port smtp
```

Cette règle permettra de bloquer un certain nombre de « spammeurs à l'insu de leur plein gré », pauvres machines tournant sous un OS bancal et vérolé jusqu'à la moelle. On peut obtenir la liste des systèmes d'exploitation que PF « reconnaît » avec la commande `pfctl -so`.

Enfin, on peut utiliser cette technique avec `tcpdump`, l'option `-o` permettra d'activer la détection de l'OS lors de la lecture d'un log.

```
root@spud[~]#tcpdump -o -r /var/log/pflog
08:57:29.102478 bas3-montreal02-1096688079.dsl.bell.ca.4996
    > my.network.org.54587: S (src OS: Windows XP SP1,
    Windows 2000 SP4) 1417722298:1417722298(0) win 65535
    <mss 1440,nop,nop,sackOK> (DF)
09:53:06.654419 host198-214.pool8248.interbusiness.it.3793
    > my.network.org.2967: S (src OS: Windows XP SP1,
    Windows 2000 SP2+) 2867956627:2867956627(0) win 16384
    <mss 1460,nop,nop,sackOK> (DF)
```

Bien évidemment, ces techniques consomment du CPU, ne sont pas infaillibles, car on peut toujours bricoler les paramètres de sa pile TCP/IP pour essayer de se faire passer pour quelqu'un d'autre, mais, dans ce domaine, `p0f` fait partie des meilleurs outils existants avec `nmap`. Si vous voulez en savoir plus, demandez « *google os fingerprinting* » à Pinpin.

6. A QUI JE DEMANDE MOOOOORE ?

6.1 PAR ICI LA SORTIE, N'OUBLIEZ PAS LE GUIDE

Voilà, j'espère avoir fait le tour de la question avec vous et que ce voyage au cœur d'un filtre de paquets réseau aura été intéressant. Vous devriez maintenant disposer des bases de PF, à vous d'en apprendre plus et d'utiliser ces connaissances à bon escient. Que la force soit avec vous, toussa !

Pour des choses un peu plus velues qui causent de haute disponibilité, équilibrage de charge et montage avancé, lisez maintenant la prose de messieurs ***aigr***flab et asher.

LINQSES

- FAQ officielle PF sur http://www.openbsd.org/faq/pf/fr/index.html
- Pf chez dhartmei@ sur http://www.benzedrine.cx/pf.html
- Pf par dhartmei@ sur http://undeadly.org/, #1
- Pf par dhartmei@ sur http://undeadly.org/, #2
- Pf par dhartmei@ sur http://undeadly.org/, #3
- PF sur wikipedia http://fr.wikipedia.org/wiki/Packet_Filter
- Tutoriel 1 : Peter Hansteen sur http://home.nuug.no/~peter/pf/en/
- Tutoriel 2 : Peter Matulis sur http://www2.papamike.ca:8082/tutorials/pub/obsd_fw.html

BOOKIN'S

The OpenBSD PF Packet Filter Book sur http://www.reedmedia.net/books/pf-book/

Building firewalls with OpenBSD and PF sur https://https.openbsd.org/cgi-bin/order.eu?B01=1&B01%2b=Add

RÉFÉRENCES

[1] http://coombs.anu.edu.au/~avalon/

[2] http://marc.theaimsgroup.com/?l=ipfilter&m=99103632211327&w=2

[3] http://archives.neohapsis.com/archives/openbsd/2001-05/3215.html

[4] http://www.newsforge.com/article.pl?sid=01/06/06/169245

[5] http://archives.neohapsis.com/archives/openbsd/cvs/2001-05/1236.html

[6] http://kerneltrap.org/node/477

[7] http://www.onlamp.com/pub/a/bsd/2004/04/15/pf_developers.html

[8] http://www.onlamp.com/pub/a/bsd/2004/05/06/pf_developers.html

[9] http://marc.theaimsgroup.com/?t=105691343700002&r=2&w=2

[10] http://kerneltrap.org/node/627

[11] http://www.ietf.org/rfc/rfc2827.txt

[12] http://www.openbsd.org/cgi-bin/man.cgi?query=pf.conf

[13] http://www.openbsd.org/cgi-bin/man.cgi?query=pfctl

[14] http://www.openbsd.org/cgi-bin/man.cgi?query=tcpdump

[15] http://lcamtuf.coredump.cx/p0f.shtml

Arnaud Coston<alf@gcu-squad.org>

IPSEC SOUS OPENBSD 4.0

Pffff IPSec la blague... Saviez-vous que dans le monde sérieux des entreprises qui gèrent des succursales, des partenaires et des tiers mainteneurs, IPSec est une réalité ? Peut-être que pour vous ce n'est pas le cas. D'autres solutions d'interconnexions sont pour vous plus abordables, plus simples à mettre en place. Peut-être aussi que pour vous IPsec est réservé à une élite... Ce n'est pas faux... IPSec est peut-être obscur, les solutions sont souvent chères ou difficiles à mettre en place, voire les deux à la fois. Mais ce qui est certain, c'est que c'est une solution propre, bien ficelée et surtout interopérable, aujourd'hui indispensable pour interconnecter son réseau avec d'autres. Avec OpenBSD, vous avez maintenant la possibilité de mettre en place simplement une solution au top. Allez, suis-moi maraud, on va leur montrer de quel bois on se chauffe à ces Juniper, Cisco et autres Enterasys.

1. RÉCUPÉRATION, INSTALLATION ET COMPILATION D'IPSEC POUR OPENBSD

Ne cherchez pas, il n'y a rien à faire en dehors d'installer OpenBSD sur la plate-forme de son choix (x86, amd64, macppc, zaurus, sparc...). IPSec est supporté de base. Simple non ?

2. DIRECTEMENT DANS LE VIF DU SUJET QUI BRÛLE

Premier contact avec IPSec sur OpenBSD 4.0 :

```
$ man -k ipsec
ipsec (4) - IP Security Protocol
ipsec.conf (5) - IPsec configuration file
ipsecctl (8) - control flows for IPsec
sasyncd (8) - IPsec SA synchronization daemon for
              failover gateways
```

... de bon augure n'est-ce pas ?

3. IPSEC FWWWWWWWOU FLASHBACK !

IPSec, quand on n'est pas ingénieur, ça fait peur. En fait, ça fait peur même aux ingénieurs. IPsec est un protocole imaginé par des collèges de normalisation qui ont voulu se faire plaisir : le résultat, c'est un machin complexe, sûrement trop complexe, qui oblige à être vigilant <... suite du blabla classique qui accompagne IPsec...>. Mais bon, dans ce monde merveilleux des lutins IP, il n'y a pas encore de meilleure solution **[1]**.

OpenBSD a ainsi été le premier système d'exploitation à inclure de base IPSec en 1997. En effet, en cette deuxième partie des années 1990, les législations concernant les moyens de chiffrement et leur exportation étaient relativement sévères, en particulier aux États-Unis. Le projet OpenBSD basé au Canada et déjà très intéressé par l'intégration de ce genre d'outil a tout naturellement occupé cette place.

Mais, presque dix ans après, quoi de neuf ? Eh bien, pas grand-chose, en dehors d'une bonne interopérabilité entre les différentes implémentations libres ou commerciales. D'habitude, c'est là que je me mange : « ouais, c'est mort, maintenant il y a mieux, il y a OpenVPN (lecteur, insère ici ta solution de VPN SSL préférée) ». Ok, donc je disais l'interopérabilité... (là, généralement, les plus relous continuent avec OpenVPN et leur client multiplateforme). Et là je dis : « stop, ton OpenVPN tu le connectes comment aujourd'hui au Juniper ou au *checkpoint* de la Megacorp avec laquelle tu dois travailler ? ».

Bon et sinon IPSec...

```
man 4 ipsec
```

« Ipsec, c'est une paire de... protocoles, l'ESP (*Encapsulating Security Payload*) et l'AH (*Authentication Header*) qui fournissent les services pour sécuriser IP : la confidentialité, l'intégrité, l'authenticité, la protection... ». Après cela, on en sait un peu plus de ce que sont « AH », « ESP », les « SA », les « SPI », le mode tunnel, le mode transport, les « *Lifetimes* ». On découvre `isakmpd`(8) et aussi un exemple de configuration qui conceptualise un peu tout ça avec la mise en place d'un VPN (*Virtual Private Network* ou Réseau Privé Virtuel). Mais rassurez-vous, lecteur, pour le moment la seule chose que l'on doit retenir, c'est simplement que l'on veut mettre en place ce VPN.

4. UN VPN IPSEC EN 5 MIN ?

Qu'est-ce qu'un VPN ? D'après Wikipédia, un VPN est un réseau virtuel, car il relie deux réseaux « physiques » (réseaux locaux) par une liaison non fiable (par exemple, Internet) et privée, car seuls les ordinateurs des réseaux locaux, de part et d'autre du VPN, peuvent « voir » les données.

Avec IPSec, on peut monter un VPN avec le protocole ESP en mode tunnel. Par contre, pour éviter toute confusion

et pour que les choses soient bien claires, quand on parle ici de monter un VPN IPSec, on parle bien de mettre en place une solution pour interconnecter deux réseaux, c'est-à-dire une solution de passerelle à passerelle.

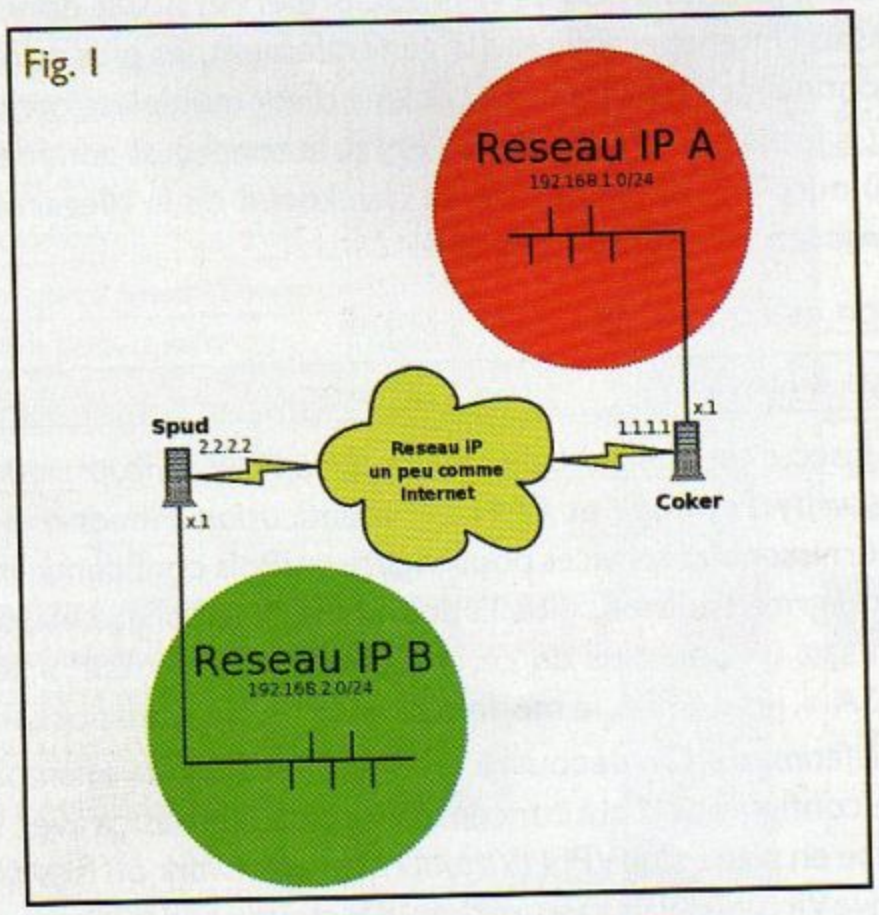
Depuis la version 3.8 d'OpenBSD, une commande magique a fait son apparition : `ipsecctl`(8). Elle a gentiment écarté l'ancienne commande `ipsecadm` qui vient de disparaître du système. L'idée d'`ipsecctl` est de configurer un VPN IPSec de la même manière que l'on configure un *firewall* `pf` avec `pfctl`... tout un programme. Résultat : aujourd'hui, grâce à OpenBSD, on peut mettre IPSec entre toutes les mains.

Ce que j'ai oublié de dire, c'est que si vous voulez m'accompagner dans les travaux pratiques qui suivent, il ne vous faudra non pas installer un OpenBSD, mais installer deux machines sous OpenBSD. J'en vois qui ricanent déjà : « ah, elle est belle l'intéroperabilité ».

4.1 LE PLAN D'ATTAQUE

L'idée est de faire communiquer les deux réseaux IP « A » et « B » à travers les passerelles « Coker » et « Spud », connues respectivement avec les adresses IP « 1.1.1.1 » et « 2.2.2.2 » sur un réseau ressemblant étrangement à Internet, mais en bien plus dangereux encore.

Fig. 1



4.2 PREMIÈRE ÉTAPE : CONFIGURONS IPSEC

Sur Coker :

```
# cat> /etc/ipsec.conf
ike esp from 192.168.1.0/24 to 192.168.2.0/24 peer 2.2.2.2
```

Sérieusement, pour les utilisateurs de `pf`, cela ne vous rappelle-t-il pas quelque chose ?

Sur Spud :

```
# cat> /etc/ipsec.conf
ike esp from 192.168.2.0/24 to 192.168.1.0/24 peer 1.1.1.1
```

4.3 DEUXIÈME ÉTAPE : INSTALLONS DE BONS SECRETS

Pas de VPN sans secrets... Un des meilleurs alliés d'IPSec, c'est le certificat numérique X509 et OpenBSD ne déroge pas à cette règle. Toutefois, tout le monde ne disposant pas de certificats X509, la solution de repli était en général d'utiliser de bons vieux mots de passe (ou secret partagé ou *PreShared Key*). OpenBSD propose cependant en plus une solution intermédiaire.

A l'installation du système, un couple de clefs publique/privée a été généré dans le répertoire `/etc/isakmpd/private/`. L'idée est de distribuer la clef publique, un peu comme on a l'habitude de le faire pour SSH, aux machines avec lesquelles nous allons monter des connexions IPSec. Bref, on oublie tout de suite les mises en place de tunnels avec des mots de passe comme « test » et « toto » sous le seul prétexte que cela est plus simple et qu'il ne s'agit que d'un test... Quoique même ça, avec `ipsecctl`, c'est encore plus simple...

Donc, copions le fichier `/etc/isakmpd/private/local.pub` de Spud vers le fichier `/etc/isakmpd/pubkeys/ipv4/2.2.2.2` sur Coker et de copier le fichier `/etc/isakmpd/private/local.pub` de Coker vers le fichier `/etc/isakmpd/pubkeys/ipv4/1.1.1.1` sur Spud.

La démarche est identique avec des FQDN plutôt que des adresses IP (exemple avec **aflab.est.une.loutre.gcu.info** ou **hr.assume.gcu-squad.org**). On aurait ainsi utilisé le fichier `/etc/isakmpd/pubkeys/fqdn/aflab.est.une.loutre.gcu.info` sur Coker. Il faudra cependant aussi le spécifier dans `/etc/ipsec.conf` à l'aide des mots clefs « srcid » et « dstid » pour qu'`ipsecctl` retrouve ses petits (`ike esp from 192.168.1.0/24 to 192.168.2.0/24 peer 2.2.2.2 srcid aflab.est.une.loutre.gcu.info dstid hr.assume.gcu-squad.org`).

4.4 TROISIÈME ÉTAPE : ON LANCE IPSEC

Sur Coker :

```
# isakmpd -K
# ipsecctl -f /etc/ipsec.conf
```

Sur Spud :

```
# isakmpd -K
# ipsecctl -f /etc/ipsec.conf
```

Maintenant, il ne reste plus qu'à monter et tester le VPN à coup de `ping`(8) entre les réseaux 192.168.1.0/24 et 192.168.2.0/24. On peut aussi vérifier l'état des tunnels sur chacune des passerelles avec `ipsecctl` et/ou `netstat`.

Sur Coker :

```
# ipsecctl  -s all
FLOWS:
flow esp in from 192.168.1.0/24 to 192.168.2.0/24
        peer 2.2.2.2 srcid 1.1.1.1/32 dstid 2.2.2.2/32 type use
flow esp out from 192.168.2.0/24 to 192.168.1.0/24
```

```
        peer 2.2.2.2 srcid 1.1.1.1/32 dstid 2.2.2.2/32 type require
SAD:
esp tunnel from 1.1.1.1 to 2.2.2.2 spi 0x9ce94c96 auth hmac-sha2-256 enc aes \
      authkey 0xa274d5679c4992715f0667e9a8288a3cbb1807aa0a3995f63e15d830469b8fd8 \
      enckey 0xf287e759aca7fd6f32401693e7aa8c45
esp tunnel from 2.2.2.2 to 1.1.1.1 spi 0x2af0ed2b auth hmac-sha2-256 enc aes \
      authkey 0x309e0330883fe13a2f17be33951537b85c4fa09e103c208170ce2e6abbc76f9f \
      enckey 0x4cf48044e643be9fcb29b88dd50afdd6
```

Ou encore...

```
# netstat -rnf encap
Routing tables
Encap:
Source       Port Destination Port Proto SA(Address/Proto/Type/Direction)
192.168.1/24 0    192.168.2/24   0    0  2.2.2.2/esp/use/in
192.168.2/24 0    192.168.1/24   0    0  2.2.2.2/esp/require/out
```

4.5 DERNIÈRE ÉTAPE : ON GARDE TOUT CELA EN MÉMOIRE

Oui on va quand même s'arranger pour qu'au prochain démarrage, le VPN soit monté automatiquement, parce que, bon, il ne faudrait pas avoir à refaire tout ça...

Donc sur Spud et Coker, on fait :

```
# cat> /etc/rc.conf.local
isakmpd_flags="-K"
ipsec=YES
```

Bravo, c'est terminé, vous avez mis en place un VPN IPsec en cinq minutes entre deux passerelles OpenBSD et en utilisant, pour les puristes, IKE avec des clefs asymétriques : alors vous trouvez ça encore compliqué IPSec ?

5. IPSECCTL ET ISAKMPD, COPAINS COMME COCHONS

Là, vous allez me dire que c'est un peu normal que ça marche aussi facilement entre deux OpenBSD. On peut faire la même chose presque aussi simplement avec n'importe quelle autre solution IPSec. Eh bien, c'est que vous avez manqué l'essentiel...

Revenons un peu sur `ipsecctl` et notre ligne d'`ipsec.conf`.

```
ike esp from 192.168.1.0/24 to 192.168.2.0/24 peer 2.2.2.2
```

Déjà, on aurait pu mettre en place un tunnel manuel, en utilisant à la place d'« ike » le mot clef « flow ». Cependant, je ne serais pas en ce moment en train de vous expliquer la magie s'opérant entre `ipsecctl` et `isakmpd`...

Pour la littérature... `isaknpd`(8), `isaknpd.conf`(5) et `idaknpd.policy`(5).

Bon, alors rapidement, IKE est un protocole d'échange de clefs utilisé aujourd'hui par IPSec. `isakmpd`, c'est le nom du démon qui gère IKE sur OpenBSD, et `isakmpd.conf` et `isakmpd.policy` sont les fichiers utilisés pour le configurer.

Le mot clef « ike » dans le fichier `ipsec.conf` indique à `ipsecctl` qu'il doit gérer le démon `isakmpd`. `ipsecctl` et `ipsec.conf` deviennent alors une véritable couche d'abstraction aux fichiers de configuration `isakmpd.conf` et `isakmpd.policy`, qui, soit dit en passant, en ont bien besoin. Notez que c'est aussi pour cette raison que nous avons démarré `isakmpd` avec l'option `-K`. C'est elle qui indique à `isakmpd` que comme `ipsecctl` mène la danse, il n'a pas besoin de charger quoi que ce soit.

Cependant, tous les paramètres d'`isakmpd` ne sont pas (encore ?) gérés par `ipsecctl`. Donc, dans certains cas, il faut encore travailler simultanément avec `ipsecctl` et un `isakmpd.conf` minimal.

Mais, en attendant, comparons...

Une configuration d'`isakmpd` qui ressemblait, avant, à ça...

```
[Phase 1]
2.2.2.2=                ISAKMP-peer-B
[Phase 2]
Connections=            IPsec-A-B
[ISAKMP-peer-B]
Phase=                  1
Address=                2.2.2.2
Configuration=          Default-main-mode
Authentication=         pipinestunefemme
[IPsec-A-B]
Phase=                  2
ISAKMP-peer=            ISAKMP-peer-B
Configuration=          Default-quick-mode
Local-ID=               Net-A
Remote-ID=              Net-B
[Net-A]
ID-type=                IPV4_ADDR_SUBNET
Network=                192.168.1.0
Netmask=                255.255.255.0
[Net-B]
ID-type=                IPV4_ADDR_SUBNET
Network=                192.168.2.0
Netmask=                255.255.255.0
[Default-main-mode]
EXCHANGE_TYPE=          ID_PROT
Transforms=             AES-SHA
[Default-quick-mode]
EXCHANGE_TYPE=          QUICK_MODE
Suites=                 QM-ESP-AES-SHA-PFS-SUITE
```

...ressemble maintenant à ça :

```
ike esp from 192.168.1.0/24 to 192.168.2.0/24 \
peer 2.2.2.2 main auth hmac-sha1 \
enc aes psk "pinpinestunefemme"
```

C'est quand même plus compréhensible... Imaginons que l'on ait des centaines de tunnels à gérer comme c'est souvent le cas sur des équipements centraux, eh bien, on appréciera tout de suite l'amélioration apportée par l'outil. C'est un peu comme, et loin de moi l'idée de lancer un troll puant, comparer des jeux de règles Netfilter à des jeux de règles `pf`. Je suis sûrement un peu idiot, mais j'ai toujours du mal à comprendre les jeux de règles Netfilter.

Et si, toutefois, on a des doutes sur le travail réalisé par `ipsecctl` ou que l'on a beaucoup d'affinités avec l'ancienne

méthode, on a la possibilité de visualiser nos jeux de règles IPSec dans un format isakmpd.conf :

```
# ipsecctl -vf /etc/ipsec.conf
C set [Phase 1]:2.2.2.2=peer-2.2.2.2 force
C set [peer-2.2.2.2]:Phase=1 force
C set [peer-2.2.2.2]:Address=2.2.2.2 force
C set [peer-2.2.2.2]:Authentication=pinpinestunefemme force
C set [peer-2.2.2.2]:Configuration=mm-2.2.2.2 force
C set [mm-2.2.2.2]:EXCHANGE_TYPE=ID_PROT force
C add [mm-2.2.2.2]:Transforms=AES-SHA force
C set [IPsec-192.168.1.0/24-192.168.2.0/24]:Phase=2 force
C set [IPsec-192.168.1.0/24-192.168.2.0/24]:ISAKMP-peer=peer-2.2.2.2 force
C set [IPsec-192.168.1.0/24-192.168.2.0/24]:
        Configuration=qm-192.168.1.0/24-192.168.2.0/24 force
C set [IPsec-192.168.1.0/24-192.168.2.0/24]:Local-ID=lid-192.168.1.0/24 force
C set [IPsec-192.168.1.0/24-192.168.2.0/24]:Remote-ID=rid-192.168.2.0/24 force
C set [qm-192.168.1.0/24-192.168.2.0/24]:EXCHANGE_TYPE=QUICK_MODE force
C set [qm-192.168.1.0/24-192.168.2.0/24]:
        Suites=QM-ESP-AES-SHA2-256-PFS-SUITE force
C set [lid-192.168.1.0/24]:ID-type=IPV4_ADDR_SUBNET force
C set [lid-192.168.1.0/24]:Network=192.168.1.0 force
C set [lid-192.168.1.0/24]:Netmask=255.255.255.0 force
C set [rid-192.168.2.0/24]:ID-type=IPV4_ADDR_SUBNET force
C set [rid-192.168.2.0/24]:Network=192.168.2.0 force
C set [rid-192.168.2.0/24]:Netmask=255.255.255.0 force
C add [Phase 2]:Connections=IPsec-192.168.1.0/24-192.168.2.0/24
```

Maintenant vous avez remarqué que, dans ce précédent exemple, la configuration est plus compliquée que celle que nous avons utilisée pour monter notre VPN en cinq minutes : nous y avons en particulier précisé les algorithmes à utiliser. Car, en effet, si par défaut on ne précise pas certains paramètres, ipsecctl va en choisir de solides pour nous (par exemple AES-256, SHA2-256).

C'est très bête, tout le monde aurait pu le faire et pourtant, au bout de dix ans, c'est le genre d'idée qui change tout. Je rajouterai qu'en plus ça rentre complètement dans la philosophie d'OpenBSD : mais vraiment, pourquoi n'ont-ils pas fait ça avant ?

6. ET SI ÇA NE MARCHE PAS ?

Justement, je parlais d'huîtres... On a de la chance, de base sous UNIX et en particulier sous OpenBSD, on a plein de trucs pour débugger les problèmes de tuyauterie IPSec.

```
# isakpmd -K -d -DA=50
```

Avec -d, on positionne isakmpd en *debug*, -DA=XX permet de faire varier le nombre et la qualité des informations à afficher.

```
# ipsecctl -s all -v
```

Le -v permet de détailler.

```
# netstat -rnf encap
```

```
# tcpdump -s1500 -nvvve -i fxp0 port 500 or port 4500 or esp
```

500 pour voir le protocole IKE, esp pour voir l'ESP et 4500 pour voir IPSec encapsulé dans UDP (NAT-T).

ou encore

```
# tcpdump -s1500 -nvvve -i enc0
```

enc0 pour rappel, c'est la pseudo interface liée à IPSec. Cette commande permet donc de regarder ce qui se passe à l'intérieur du tunnel.

```
# ping -I 192.168.1.99 192.168.3.99
```

Le -I peut être extrêmement pratique pour vérifier l'état du tunnel directement depuis la passerelle.

```
/usr/src/sbin/isakmpd/DESIGN-NOTES
```

On y trouve de bonnes informations sur l'implémentation d'isakmpd dans OpenBSD (il faut cependant avoir récupéré les sources). De quoi se retourner plusieurs fois le cerveau...

7. UNE CERISE SUR UN GÂTEAU : SASYNCD

Un jour de mars 2005 est tombé ceci dans le CVS du projet OpenBSD : Log Message: Directory /cvs/src/usr.sbin/sasyncd added to the repository.sasyncd ? Oui sasyncd(8), voyons... le démon de synchronisation d'IPSec.

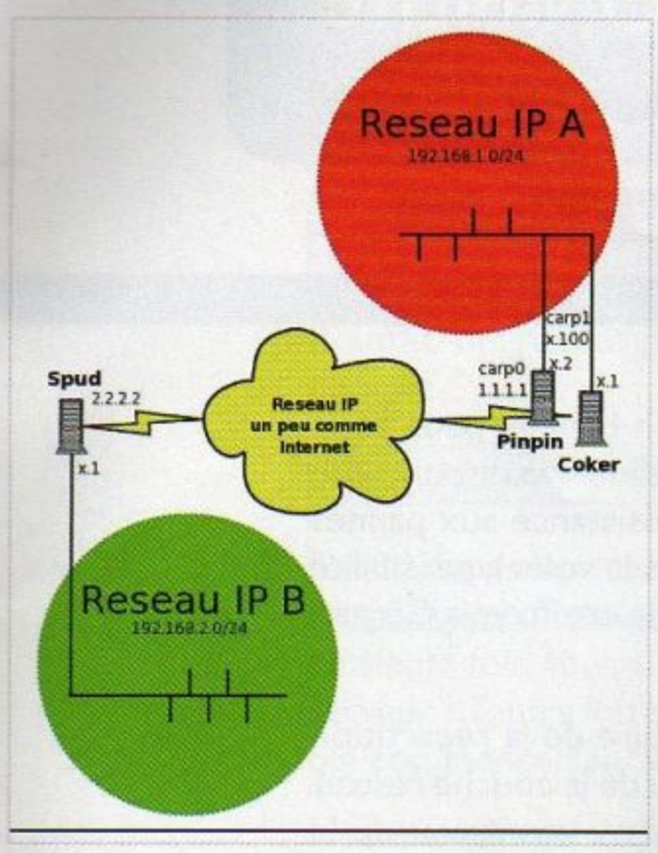
Pourquoi synchroniser IPSec ? Pour la même raison que l'on synchronise les états d'un firewall, pour éviter de se retrouver avec des utilisateurs déconnectés et des applications plantées si quelque chose de triste arrive à notre passerelle IPSec. Mais, à vrai dire, synchroniser les informations des tunnels IPSec actifs (les « SA » ou « *Security Association* ») ne sert pas à grand-chose tout seul. En revanche, synchroniser quand on utilise en parallèle un mécanisme de haute disponibilité réseau comme CARP(4), cela devient une tuerie.

Je ne m'attarderai pas trop longtemps sur CARP. Cependant, pour les lecteurs qui ne connaissent pas encore ce protocole, il s'agit d'un mécanisme qui permet de partager une même adresse IP virtuelle entre plusieurs équipements sur un même LAN (ou réseau local). Un des équipements voit son interface CARP désignée, en fonction de paramètres, comme « MASTER » et les interfaces CARP des autres équipements se positionnent en « BACKUP ». C'est l'équipement possédant l'interface CARP à l'état « MASTER », qui utilisera l'interface IP virtuelle associée. Enfin, des événements externes peuvent avoir des effets sur les interfaces CARP et ainsi entraîner l'élection d'un nouveau « MASTER ».

En mars 2005, il était cependant encore un peu tôt, et c'est seulement avec OpenBSD 4.0 que sasyncd est devenu réellement intéressant, plus robuste et plus agile. Avec l'arrivée « du compteur de dégradation » de CARP (carpdemote) contrôlable en *userland*, sasyncd a gagné un peu de pouvoir. Par exemple, en mode préemptif (preempt), une interface CARP ne rebasculera à l'état « MASTER » qu'après avoir complètement synchronisé ses « SA » **[2][3][4]**. Il est devenu agile comme un pfsync, je vous dis !

Renseignons-nous un peu... sasyncd(8) et sasycnd.conf(5).

7.1 LE NOUVEAU PLAN



Nous allons tester CARP et `sasyncd` dans un scénario très basique en s'appuyant sur notre TP de tout à l'heure. Nous allons ajouter la machine Pinpin à notre configuration, machine qui devra travailler de concert avec Coker en partageant des interfaces CARP en mode préemptif et en synchronisant ses SA avec cette dernière. Notez que l'interface 1.1.1.1 est devenue une interface virtuelle CARP.

7.2 CONFIGURONS CARP

Sur Coker :

```
# sysctl -w net.inet.carp.preempt=1
# ifconfig carp0 1.1.1.1 netmask 255.255.255.0 \
vhid 1 pass pinpinestamour carpdev sis0 advskew 120

# ifconfig carp1 192.168.1.100 netmask 255.255.255.0 \
vhid 2 pass pinpinestbonheur carpdev sis1 advskew 120
```

Je saute volontairement la configuration des autres interfaces pour me concentrer sur les aspects spécifiques de `sasyncd`.

Sur Pinpin :

```
# sysctl -w net.inet.carp.preempt=1
# ifconfig carp0 1.1.1.1 netmask 255.255.255.0 \
vhid 1 pass pinpinestamour carpdev sis0 advskew 150

# ifconfig carp1 192.168.1.100 netmask 255.255.255.0 \
vhid 2 pass pinpinestbonheur carpdev sis1 advskew 150
```

7.3 CONFIGURONS SASYNCD

Sur Coker :

```
# cat> /etc/sasyncd.conf
interface carp0
peer 192.168.1.2
sharedkey b341aa065c3850edd6a61e150d6a5fd3

# chmod 0600 /etc/sasyncd.conf
```

Sur Pinpin :

```
# cat> /etc/sasyncd.conf
interface carp0
peer 192.168.1.1
sharedkey b341aa065c3850edd6a61e150d6a5fd3

# chmod 0600 /etc/sasyncd.conf
```

7.4 LANÇONS LE BIDULE

Sur les deux machines :

```
# sasyncd
```

Pensez qu'il faudra relancer `isakmpd`, s'il a été arrêté depuis. Pensez aussi à partager les règles `ipsec.conf` entre les deux machines. De même, assurez-vous d'avoir bien installé le couple de clefs plublique/privée de Coker sur Pinpin, sinon la passerelle en face va nous snober. Enfin, il faut aussi partager toutes les clefs publiques identifiant les autres passerelles.

7.5 VERDICT : COUPABLE

Si tout s'est bien passé sur Pinpin et Coker, on peut voir ça en double :

```
# ipsecctl -sa
FLOWS:
flow esp out from 192.168.1.0/24 to 192.168.2.0/24
        peer 2.2.2.2 srcid 1.1.1.1/32 dstid 2.2.2.2/32
        type require
flow esp in from 192.168.2.0/24 to 192.168.1.0/24
        peer 2.2.2.2 srcid 1.1.1.1/32 dstid 2.2.2.2/32
        type use

SAD:
esp tunnel from 2.2.2.2 to 1.1.1.1 spi 0x90a7a6a \
        auth hmac-sha2-256 enc aes \
        authkey 0x073367ddb3d7ccb3c196005322b96586eb6b5c2ba1371291b896a88056ac24a8 \
        enckey 0x04d53fa17ceb40907bddd456af9d8dca
esp tunnel from 1.1.1.1 to 2.2.2.2 spi 0xa11dcb8c auth \
        hmac-sha2-256 enc aes \
        authkey 0xbf73e23d962391252e4a6e0532eea2ea1de7db3ac1b8b1cf90adcf2fb5f866b9 \
        enckey 0x42c6161bc9d802674ab5f34f819f89d5
```

Bref, il n'y a plus qu'à couper le fil...

CONCLUSION

`ipsecctl` est l'outil qui manquait et qui aurait peut-être dû arriver plus tôt. Mon avis est que c'est une bonne nouvelle pour IPSec, qui avait bien besoin d'être dépoussiéré et remis au goût du jour, (lire « simplifié »). Maintenant, il faut espérer qu'`ipsecctl` prenne le même chemin que `pf`, ou, rêvons encore un peu plus, qu'OpenSSH soit porté sur d'autres plateformes.

Avec `sasyncd`, c'est bien plus le travail d'horloger qui est ici intéressant. En fait, tout s'emboîte à merveille dans OpenBSD, de `bgpd` à `ospdf`, de `pf` à `pfsync` en passant par CARP. `sasyncd` n'est ici qu'une pièce supplémentaire au puzzle. Cependant, il a aussi l'inconvénient de complexifier quelque peu la configuration de la passerelle IPSec. Mais franchement, avouez que `sasyncd`, c'est quand même le genre de fonctionnalité qui vous fait regarder certains équipements réseau hors de prix comme des bouses infâmes...

RÉFÉRENCES

[1] http://www.schneier.com/paper-ipsec.html

[2] http:www.undeadly.org/cgi?action=article&sid=20060621160000

[3] http:www.oreillynet.com/pub/a/sysadmin/2006/10/26/openbsd-40.html?page=1

[4] http://www.openbsd.org/faq/pf/fr/carp.html

aflab pour GCU Canal Hystérique

RÉPARTITION DE CHARGE ET HAUTE DISPONIBILITÉ SUR LES OS *BSD (PARTIE 1)

OU COMMENT FAIRE FAIRE DE LA BALANÇOIRE À VOS PETITS LUTINS

1. AVERTISSEMENT

Il se peut qu'au cours de cet article vos lutins décident de s'envoyer en l'air, GCU ne pourra être tenu responsable de la perte de l'un d'eux.

Nous partons du principe que vous avez déjà utilisé PF et nos chers *BSD.

2. INTRODUCTION

Toi aussi, tu es en plein *leverage* de ton architecture *middleware* pour l'installation de ton nouveau *progiciel* de *e-learning*. Alors, pour le déploiement de cette nouvelle application *web2.0 compliant* tes managers te demandent d'améliorer la *Load scalability* et, si tu n'y arrives pas, tu vas perdre le *leadership* de ta *team*.

MAIS, heureusement, tu as lu le dernier 01 et tu as vu les mots *LOAD BALANCING* et *HIGH AVAILABILITY*... Eh bien oui, dans cet article nous allons t'apprendre à faire tout ça à l'aide de tes petits doigts et des merveilleux outils que nos barbus de chez *BSD and Co. nous ont mis à disposition.

Certes, cela ne va pas provoquer l'effet d'un *cululingus* sur tes managers, mais je t'assure que tes cheveux vont briller.

3. LES CONCEPTS (OUI GERARD, IL FAUT CONCEPTUALISER)

Gérard, je sais que tu te demandes ce qu'est la haute disponibilité. Eh bien, je vais te le dire. C'est un « concept » visant à mettre en œuvre un ensemble de mesures permettant d'améliorer et de garantir une très forte disponibilité du service. L'ensemble des mesures nécessaires à l'amélioration de la garantie de disponibilité passe par des actions au niveau du matériel, du système, du réseau etc.

Nous allons nous intéresser à quelques techniques disponibles, grâce aux outils présents sur nos chers systèmes *BSD, visant à améliorer la disponibilité au niveau réseau.

La répartition de charge ou load balancing (pour briller en société) est une technique permettant de répartir la charge sur une ferme de serveurs ou sur un pool de serveurs (toujours pour briller en société) et donc toujours dans la continuité. Alors oui, Gérard, tu te demandes, mais à quoi ça sert ? Eh bien, je peux te le dire, ça sert tout simplement à améliorer la disponibilité de ton service : une plus grande résistance aux pannes (ajout et suppression de serveurs à la volée), possibilité de répondre à plus de clients à la fois, etc. Tu vois Gérard, que c'est pas inutile !

Donc Gérard, je t'explique. Pour faire de la répartition de charge, tu peux le faire au niveau de la couche réseau, au niveau de la couche transport ou au niveau de la couche application (dis-moi Gérard, tu as bien écouté les cours de réseaux ou tu me fais des yeux de merlan frit par pur plaisir ?).

De plus, la répartition de charge sur tes serveurs peut se faire à l'aide de différents algorithmes : le plus connu est l'algorithme « *round robin* » *aka* je balance un coup à l'un un coup à l'autre, aka aussi la balançoire à lutins. Ce qui signifie qu'à chaque connexion, il est probable que tu n'interroges pas le même serveur. Si, pour une raison ou pour une autre, tu dois mettre en œuvre une persistance pour que l'utilisateur tombe sur le même serveur, tu devras utiliser un algorithme de type « hash ».

Passons aux choses sérieuses...

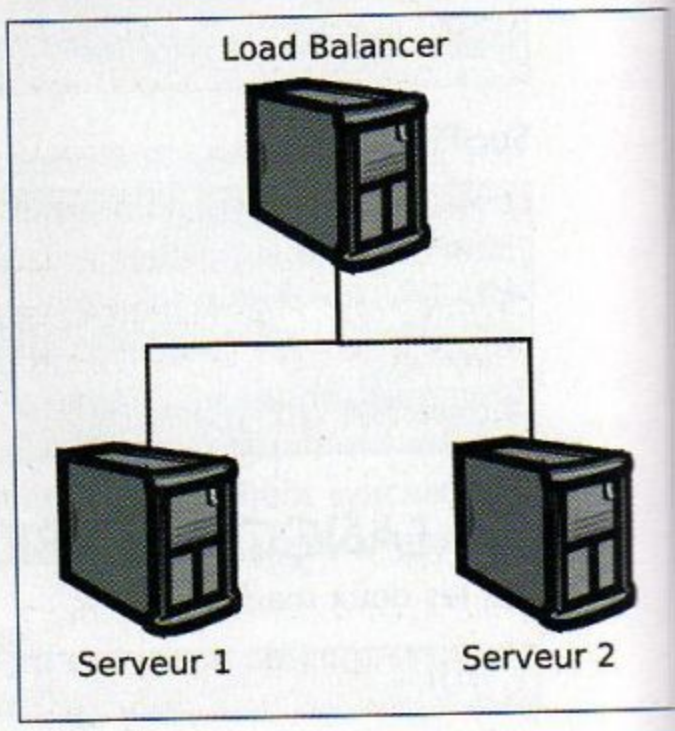
4. LES CHOSES SÉRIEUSES (PART ONE)

Voilà, Gérard, tu as donc 2 serveurs WEB et tu te dis : « Mais comment je vais pouvoir faire du load balancing sans acheter un boîtier à très très cher ? » Eh bien, Gérard, j'ai la solution ! Je te propose d'utiliser les fonctionnalités de notre magnifique *firewall* que nous aimons tous c'est-à-dire PF *aka Packet Filter*, pour ceux qui n'ont pas lu les pages précédentes.

Liste des courses avant de commencer :

- 1 machine avec 2 cartes réseau faisant office de load balancer ;
- 2 machines avec une carte réseau faisant office de serveur web.

Voyons donc à quoi ressemble notre architecture :



Nous allons configurer PF pour lui dire de rediriger chaque requête sur un des serveurs :

```
web = "{ 192.168.0.2 192.168.0.3 }"

rdr pass on $ext_if proto tcp from any to $ext_if \
        port 80 -> $web round-robin
```

Il suffit de recharger PF pour que ce soit pris en compte.

```
# pfctl -f /path/to/pf.conf -F nat
```

Il suffit donc maintenant d'appeler dans ton joli navigateur **http://ip_externe_load_balancer/** et, si tu recharges plusieurs fois, tu vas tomber une première fois sur ton serveur 1, l'autre fois sur ton serveur 2. C'est la balançoire qui commence, Gérard !

Notons que ceci est aussi possible avec IPF, mais bon, PF c'est quand même vachement mieux.

5. OUI, MAIS SAI KOI LES ALGORITHMES DISPONIBLES ???? (PART TWO)

Voila Gérard ! Je savais que tu allais me le demander. Eh bien, PF implémente 4 types d'algorithmes pour la répartition de charge :

- **bitmask**, algorithme qui se base sur la plage d'adresses IP du pool de serveurs et l'adresse IP du client ;
- **random**, algorithme permettant de rediriger chaque connexion de façon aléatoire sur le pool de serveurs ;
- **round-robin**, algorithme permettant de rediriger de façon séquentielle chaque connexion ;
- **source-hash**, basé sur l'adresse IP du client permettant d'obtenir une persistance : un même client sera toujours redirigé vers le même serveur.

Le seul algorithme qui répartit la charge de façon équitable est l'algorithme round-robin. En effet, si tu as 10 clients qui se connectent et 5 serveurs, chaque serveur recevra 2 connexions, ce qui n'est pas garanti avec les autres.

Une subtilité supplémentaire avec les algorithmes random et round-robin. Il est possible de rajouter l'option `sticky-address` qui permet d'avoir une pseudo-persistance. Le client tombera sur le même serveur tant que PF aura des informations concernant cette session dans ses tables.

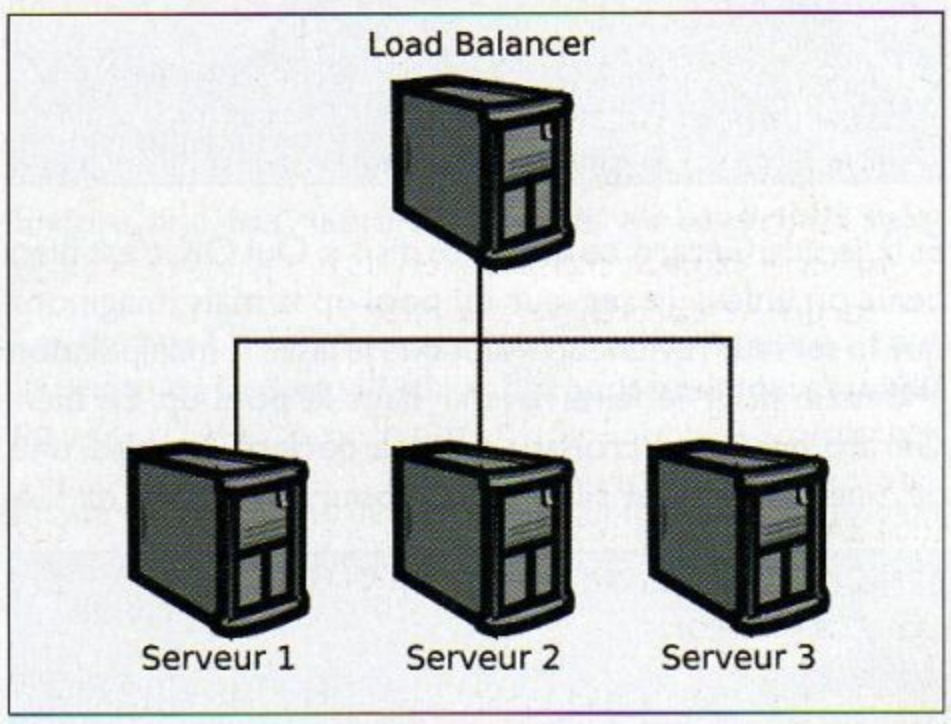
6. OUI, MAIS, SI ÇA PLANTE ? (PART THREE)

Oui, Gérard, tu as raison, si ça plante ? Eh ben, pour le moment, si un de nos serveurs web est en panne pour une raison ou une autre (par exemple, parce que ton prestataire venu d'une grande SSII ne connaît rien à l'administration, pourtant il est expert !), eh bien, le load balancer (le répartiteur de charge, je te vois venir toi le défenseur de la langue française) continuera à envoyer des demandes à ce serveur. Le service sera donc fortement perturbé, puisqu'une fois sur deux le client tombera sur le mauvais serveur.

Mais là encore une fois, Gérard, pour toi, j'ai une solution pas chère et facile à mettre en œuvre.

Dans l'article « PF pour les nuls » (quoi tu ne l'as pas lu ????), tu as entendu parler des tables. Je sais que tu n'es pas convaincu de la puissance des tables et pourtant je vais te montrer qu'elles vont nous permettre d'ajouter et de retirer des serveurs à la volée.

Voila donc notre architecture un peu plus complexe (**je ris**) :



Commençons par notre configuration PF :

```
table <web_up> persist {192.168.0.2,192.168.0.3,192.168.0.4}
table <web_down> persist { }

rdr pass on $ext_if proto tcp from any to $ext_if \
        port 80 -> <web_up> round-robin
```

Le mot `persist`, quand on déclare une table, est très important, car cela va nous permettre de manipuler cette table même une fois que les règles sont chargées.

Il nous suffit de recharger ensuite notre fichier de configuration.

```
# pfctl -f /path/to/pf.conf -F nat
```

Je vois bien ton air dubitatif ; tu te dis : « mais ça gère pas le *failover* toussaaaa, oui mais regarde » !

```
# pfctl -t web_up  -Ts
```

Et là, ô, magie, tu vois apparaître la liste de tes serveurs qui sont dans le pool « up ».

De même, si tu tapes :

```
# pfctl -t web_down  -Ts
```

Là, normalement, c'est vide ou alors Gérard tu as sauté une étape.

Donc, imagine maintenant que tu fasses un contrôle sur tes serveurs (toutes les x minutes, par exemple) et quand le test échoue sur l'un d'entre eux tu l'enlèves de ton pool de serveurs sur lequel les requêtes sont envoyées :

```
#!/bin/sh

PING="/sbin/ping -c 2 -t 2"

if ! WEB_UP=`pfctl -t web_up -Ts`; then
        echo "la table web_up n'existe pas"
        exit 1
fi

for IP in $WEB_UP; do
        if $PING $IP >/dev/null 2>&1; then
                echo "$IP est up"
        else
                echo "$IP est down"
                pfctl -t web_up -Td $IP >/dev/null 2>&1
        fi
done
```

Et là, je sais, Gérard, ce que tu te dis : « Oui OK, c'est bien beau, on enlève le serveur du pool up », mais imaginons que le serveur revient up, il faut que je fasse la manipulation à la main pour le faire revenir dans le pool up. Eh bien, Gérard pourquoi crois-tu qu'on a déclaré 2 tables, une *up*, une *down* ? Tout simplement, pour gérer tout ça !

```
#!/bin/sh

PING="/sbin/ping -c 2 -t 2"

if ! WEB_UP=`pfctl -t web_up -Ts`; then
        echo "la table web_up n'existe pas"
        exit 1
fi

if ! WEB_DOWN=`pfctl -t web_down -Ts`; then
        echo "la table web_down n'existe pas"
        exit 1
fi

for IP in $WEB_UP; do
        if $PING $IP >/dev/null 2>&1; then
                echo "$IP est up"
        else
                echo "$IP est down"
                pfctl -t web_up -Td $IP >/dev/null 2>&1
                pfctl -t web_down -Ta $IP >/dev/null >2>&1
        fi
done

for IP in $WEB_DOWN; do
        if $PING $IP >/dev/null 2>&1; then
                echo "$IP est maintenant up"
                pfctl -t web_down -Td $IP >/dev/null >2>&1
                pfctl -t web_up -Ta $IP >/dev/null 2>&1
        else
                echo "$IP toujours down"
        fi
done
```

Alors oui, je sais, tu vas me dire qu'un test *ping*, ça suffit pas ! Tu as bien raison, mais j'ai confiance en toi pour nous coder une série de tests qui vont te permettre de certifier que ton serveur est bien down.

Il est donc possible d'ajouter des serveurs ou d'en retirer à la volée, ce qui permet quand tu ajoutes une nouvelle machine dans ton pool de ne pas perturber le service en rechargeant de façon intempestive ta configuration.

Je sais ce que tu te dis : « c'est beau ». Oui, ça l'est et j'espère t'avoir convaincu de l'utilité des tables.

7. LA HONTE, IL PARLE DE HAUTE DISPONIBILITÉ, MAIS SI SON LOAD BALANCER PLANTE ! (PART FOUR)

Je te voyais venir là, tu as vu de suite le point faible de l'architecture ; en effet, si notre load balancer tombe en panne, le service est complètement interrompu.

C'est dans ces moments que tu te dis que la vie est bien faite quand même, parce que nos chers amis (barbus) de chez OpenBSD ont pensé à tout. Ils ont développé CARP (*Common Address Redundancy Protocol*, une alternative libre aux protocoles VRRP, HSRP et FSRP qui sont soumis à des brevets).

Donc, comme son nom l'indique et comme tu as pu le deviner, cela va nous permettre de partager une IP « publique » entre 2 machines : un load balancer maître et un load balancer esclave. Comme ça, en cas de défaillance du load balancer maître, le load balancer esclave s'attribuera l'adresse IP « publique » : le service pourra continuer à fonctionner.

La liste des courses avant de commencer :

- 2 load balancer avec 2 cartes réseau ;
- 3 serveurs avec une carte réseau (chacun, hein !) ;
- des câbles réseau ;
- des *switchs* ;
- de l'alcool[1]
- de la drogue[1]
- un truc à manger.

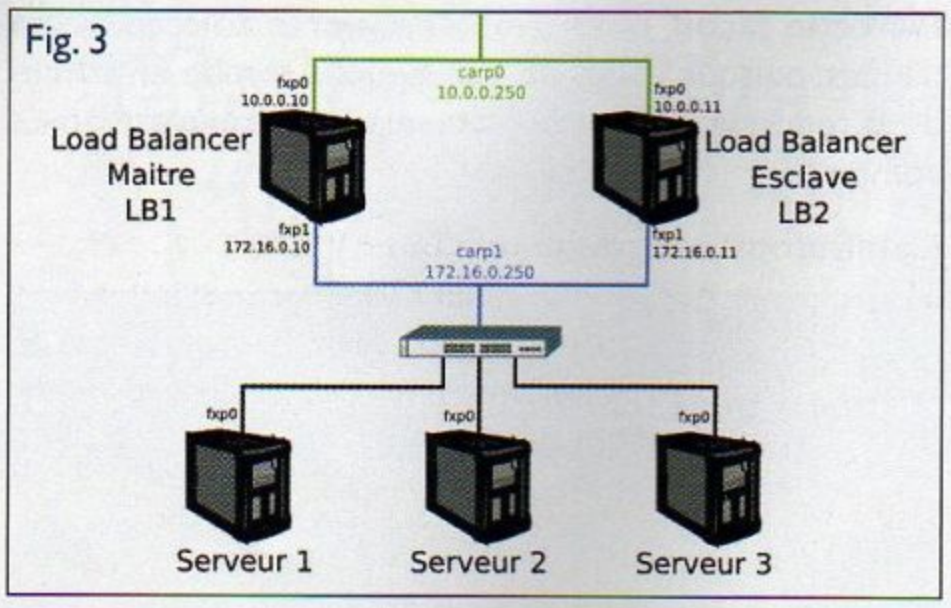
Voilà donc l'architecture que nous allons mettre en place (voir Fig. 3).

Donc, dans cet exemple, 10.0.0.0/24 est notre plage d'adresses « publique », la plage 172.16.0.0/24 étant notre plage d'adresses dite « privée », sur laquelle nos serveurs sont connectés.

Nos 2 load balancers ont donc sur chacune de ces plages une adresse IP réelle (RIP) et une adresse IP virtuelle (VIP) qu'ils vont se partager.

[1] NDA : Non, c'est pour rire, c'est mal !

Fig. 3



LB1 (Load Balancer maître) possède :

- l'adresse IP « publique » : 10.0.0.10 (fxp0) ;
- l'adresse IP « privée » : 172.16.0.10 (fxp1)

LB2 (Load Balancer esclave) possède :

- l'adresse IP « publique » : 10.0.0.11 (fxp0) ;
- l'adresse IP « privée » : 172.16.0.11 (fxp1)

Côté réseau « public », la VIP partagée sera **10.0.0.250**.

Côté réseau « privé », la VIP partagée sera **172.16.0.250**.

Alors Gérard, tu peux te demander pourquoi, côté réseau privé, nous avons besoin d'une VIP : tout simplement parce que nos load balancers font aussi office de passerelles pour nos serveurs. Il faut donc que l'adresse IP de la passerelle ne change pas pour que les serveurs puissent continuer à communiquer vers l'extérieur.

Configurons CARP sur LB1 :

```
# ifconfig carp0 create
# ifconfig carp1 create

# ifconfig carp0 vhid 1 advskew 1 \
        pass insère_ici_un_password 10.0.0.250
# ifconfig carp1 vhid 2 advskew 1 \
        pass insère_ici_un_autre_password 172.16.0.250
```

Configurons CARP sur LB2 :

```
# ifconfig carp0 create
# ifconfig carp1 create

# ifconfig carp0 vhid 1 advskew 100 \
        pass insère_ici_un_password 10.0.0.250
# ifconfig carp1 vhid 2 advskew 100 \
        pass insère_ici_un_autre_password 172.16.0.250
```

Tu as sans doute remarqué la différence de valeur entre LB1 et LB2 pour la directive `advskew`. Cela nous permet de dire que LB1 sera le maître par rapport à LB2 ; il aura donc la priorité sur LB2.

Notons aussi que `insère_ici_un_password` et `insère_ici_un_autre_password` doivent être identiques sur les 2 machines.

Il faut penser à activer le mode `preempt` aussi.

```
# sysctl -w net.inet.carp.preempt=1
```

La définition du mode `preempt` et de `advskew` est très importante pour permettre à LB1 de redevenir maître, sinon, en cas de défaillance de celui-ci, LB2 gardera la main indéfiniment, jusqu'à que ce tu fasses la manipulation nécessaire pour que LB1 redevienne maître. Alors que là, c'est automatique. Si !

Notons aussi, qu'au besoin, il faut autoriser le protocole CARP sur ton firewall :

```
pass quick on fxp0 proto carp keep state
pass quick on fxp1 proto carp keep state
```

À rajouter dans ton `pf.conf`.

8. OUI, MAIS ! (PART FIVE)

Je te vois, Gérard, tu es en train de te dire : « OUI, MAIS SI LB1 PLANTE, OK, LB2 PREND LA MAIN, MAIS IL A TOUT PERDU ET DONC SAI NUL. »

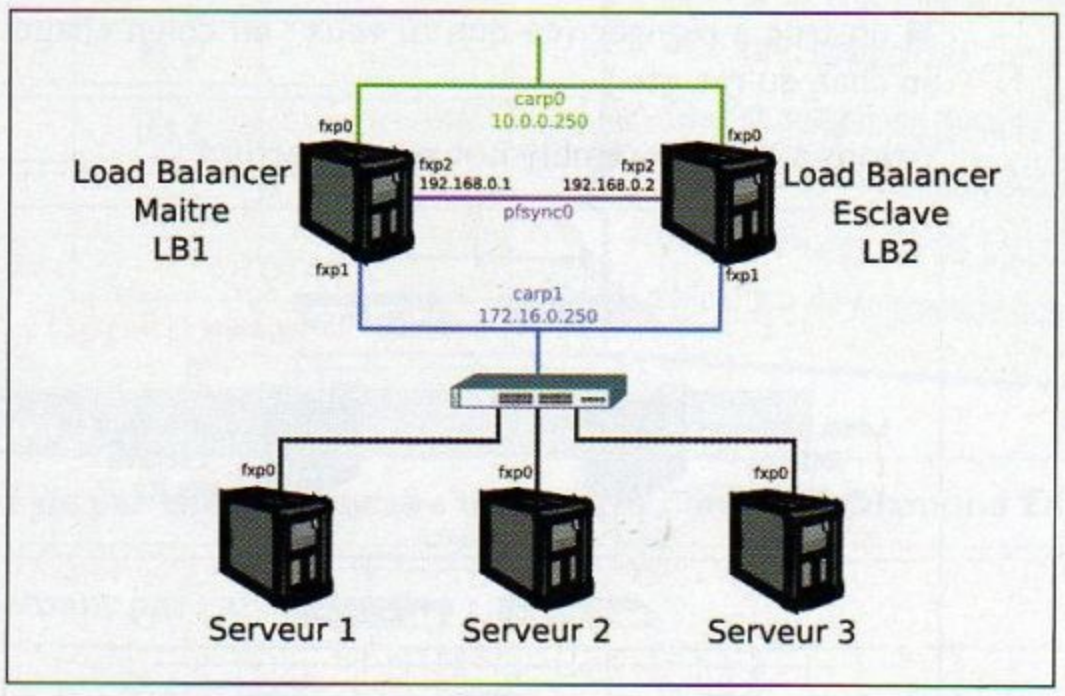
Encore une fois, Gérard, j'ai envie de te dire : « **tais-toi** ».

Les barbus (ils font forts ces barbus !) ont pensé à tout. Ils ont mis à ta disposition un joli petit jouet qui s'appelle Pfsync.

Pfsync est un pseudo *device* réseau qui se charge d'ausculter les changements d'état du firewall et de les envoyer sur le réseau.

Par défaut, Pfsync fait du *multicast* et sans authentification. C'est pourquoi, il est préférable d'utiliser un réseau séparé pour éviter qu'un gentil pirate ne déclare aussi son Pfsync. Notons qu'il existe l'option `syncpeer` permettant de spécifier à Pfsync d'envoyer en unicast à un seul serveur Pfsync (améliorant ainsi la sécurité).

Donc, on reprend la même architecture que tout à l'heure et on l'améliore :



Par rapport à tout à l'heure nous avons rajouté 1 carte réseau sur chaque load balancer : ils ont donc un réseau dédié pour les échanges pfsync.

Configurons LB1 :

```
# ifconfig fxp2 192.168.0.1 netmask 255.255.255.0
# ifconfig pfsync0 syncdev fxp2
# ifconfig pfsync0 up
```

Configurons LB2 :

```
# ifconfig fxp2 192.168.0.2 netmask 255.255.255.0
# ifconfig pfsync0 syncdev fxp2
# ifconfig pfsync0 up
```

Au besoin, il faut autoriser le protocole Pfsync sur ton firewall :

```
pass quick on fxp2 proto pfsync
```

... dans ton `pf.conf`.

9. OH OUI, J'EN VEUX ENCORE ! (PART SIX)

Gérard, tu es gourmand, mais j'aime ça. Ta soif de connaissance me plaît.

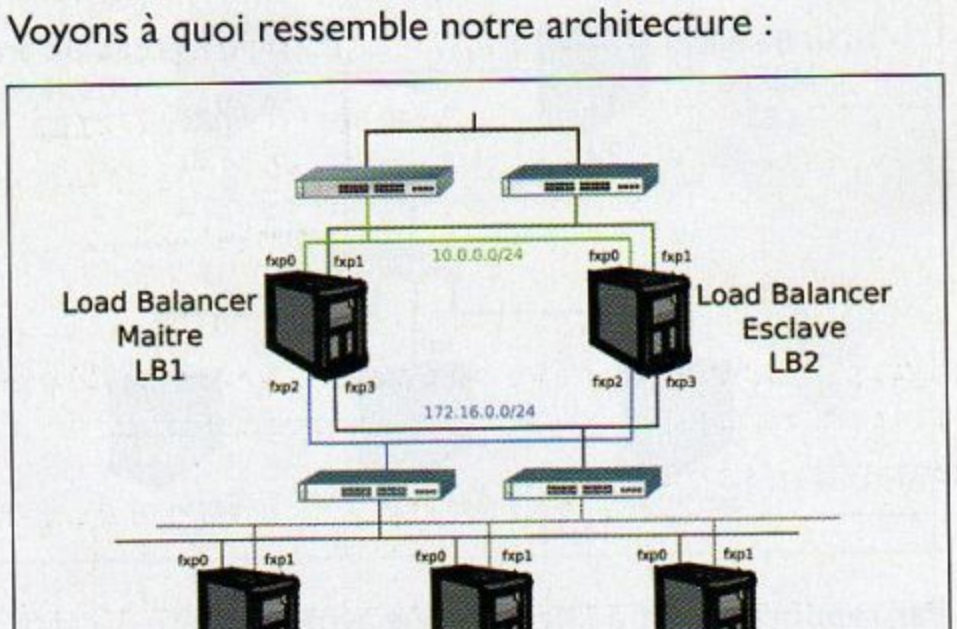
Je vais maintenant te parler d'une nouvelle technique : *interface bonding* (rien à voir avec le *bondage* Gérard, tu as vraiment l'esprit mal placé) ou agrégation d'interface. Je vois bien tes yeux briller Gérard.

Cette technique te permet tout simplement de faire l'agrégation de 2 liens réseau, ce qui offre une grande tolérance aux pannes. En effet, si un lien réseau tombe, le reste du trafic bascule vers l'autre lien réseau.

Voilà la liste des courses :

- 2 load balancers avec 4 interfaces réseau ;
- 2 serveurs avec 2 interfaces réseau ;
- 4 switchs ;
- des câbles réseau ;
- une photo de Pinpin ;
- de l'alcool[1] ;
- de la drogue[1] ;
- un truc à manger (ce que tu veux : un chien chaud, du chat, du rat, etc.).

Voyons à quoi ressemble notre architecture :



De cette façon, nous avons une forte tolérance aux pannes, puisque, si un de nos switchs tombe en panne, il est redondé et donc le « chemin » réseau n'est jamais rompu.

Configurons les agrégats sur LB1 :

```
# ifconfig fxp0 up
# ifconfig fxp1 up
# ifconfig trunk0 trunkproto failover \
        trunkport fxp0 trunkport fxp1 \
        10.0.0.10 netmask 255.255.255.0

# ifconfig fxp2 up
# ifconfig fxp3 up
# ifconfig trunk1 trunkproto failover \
        trunkport fxp2 trunkport fxp3 \
        172.16.0.10 netmask 255.255.255.0
```

Gérard tu as donc compris que `trunk0` est l'agrégat de l'interface fxp0 et fxp1 et que `trunk1` est l'agrégat de fxp2 fxp3.

Le mot clé `failover` permet de spécifier que l'ensemble du trafic passera par fxp0 et, en cas de défaillance de ce lien, le trafic passera par fxp1. De même, avec avec fxp2 et fxp3.

Configurons les agrégats sur LB2 :

```
# ifconfig fxp0 up
# ifconfig fxp1 up
# ifconfig trunk0 trunkproto failover \
        trunkport fxp0 trunkport fxp1 \
        10.0.0.11 netmask 255.255.255.0

# ifconfig fxp2 up
# ifconfig fxp3 up
# ifconfig trunk1 trunkproto failover \
        trunkport fxp2 trunkport fxp3 \
        172.16.0.11 netmask 255.255.255.0
```

Et ainsi de suite...

Il ne te reste plus qu'à appliquer l'ensemble des techniques que tu as vues tout au long de cet article sur tes nouvelles interfaces virtuelles.

Note Gérard que cette manipulation n'est possible qu'à partir d'OpenBSD 3.9.

POUR CONCLURE

Voilà, si tu n'as pas compris quelque chose dans cet article, je n'ai qu'une chose à te dire : « **RTFM** ».

Ah oui, Gérard, tu es invité à mon mariage avec Spud.

Pour les sources d'informations :

- **http://openbsd.org/faq/pf/**
- **http://openbsd.org/faq/faq6.html#CARP**
- **http://www.countersiege.com/doc/pfsync-carp/**
- **http://team.gcu-squad.org/~aflab/papers/**
- **http://www.gcu.info/**

Stefan « hr » Berder <hr@bonz.org>

ROUTAGE DYNAMIQUE ET HAUTE DISPONIBILITÉ (PARTIE 2)

OU METS TES LUTINS AU TRICOT

AVERTISSEMENT

Il faut toujours faire attention quand on manie des objets pointus ou tranchants, donc ne courez pas avec cet article dans les mains et ne le laissez pas à la portée des enfants en bas âge. Ah, et une dernière chose, surtout ne nourrissez jamais cet article après minuit, on ne sait pas ce qu'il pourrait advenir.

Il est préférable d'avoir une certaine connaissance des protocoles de routage dynamique OSPF **[1]** et BGP **[2]** et une bonne maîtrise d'OpenBSD. La lecture de l'article précédent « Répartition de charge et Haute disponibilité sur les OS *BSD (Partie 1) » est conseillée. Nous aborderons des concepts qui y sont brillamment expliqués </prosel>. Ayant travaillé avec des routeurs sensibles, j'ai modifié les IP, AS et noms des machines publiques.

1. INTRODUCTION

Ah, te voilà Gérard, c'était bien ton cours précédent ? Je vois que tu as encore des étoiles plein les yeux et un sourire béat sur le visage. Tu en veux encore j'ai l'impression, j'ai tout de suite senti que tu étais un goulu. On va rassasier cette faim d'apprendre avec un peu plus de haute disponibilité. Il ne faut jamais contrarier le curieux, tu le sais ça, mon petit Gérard.

Donc, nous allons nous pencher, mais pas trop, sur les notions de *multihoming* **[3]** et de protocoles de routage dynamique. On va y aller pas à pas, comme si on apprenait le vélo avec les petites roues d'abord. On va commencer par parler rapidement d'OSPF et BGP et ensuite on installera un environnement propice à la mise en place de notre système de haute disponibilité. Tu m'écoutes Gérard ?

Les aspects de sécurisation de ces protocoles ne seront pas abordés, donc on ne parlera pas de chiffrement des sessions ou de règles *firewall* pour protéger les accès aux sessions BGP.

1.1 OSPF

L'OSPF (*Open Shortest Path First*) est un protocole IGP (*Interior Gateway Protocol*) **[4]** d'annonce de route qui permet de transmettre les états des interfaces (*link-state*) d'un routeur à d'autres routeurs. La transmission se fait en *multicast* et uniquement sur changement de topologie. Les routeurs connectés en OSPF peuvent donc se faire une représentation du réseau sous forme d'un graphe et choisir le chemin le moins coûteux en utilisant l'algorithme de Dijkstra.

1.2 BGP

Le BGP (*Border Gateway Protocol*) permet d'échanger des routes agrégées (adresse + *netmask*) entre routeurs. Cet échange se fait en TCP sur le port 179. L'échange se fait généralement entre AS (*Autonomous System*) et uniquement de voisin à voisin. On parle d'eBGP lorsque l'échange se fait entre AS différents et d'iBGP au sein d'un même AS. Les tables de routes échangées permettent ensuite de créer la table de routage du routeur BGP grâce à un algorithme décisionnel.

1.3 OPENBGPD ET OPENOSPFD

Le projet OpenBSD **[7]** nous gratifie de 2 excellents daemons implémentant les protocoles de routage OSPF **[5]** et BGP-4 **[6] [10]** tels que décrits dans leurs RFC. Ces services assurent donc la collecte et l'algorithme de décision qui permet de construire la table de routage du *kernel*. La base de connaissance des routes apprises via ces 2 protocoles est appelée RIB (*Routing Information Base*) et la table de routage est appelée FIB (*Forwarding Information Base*). Ces 2 serveurs tournent avec séparation de privilège. Le processus *root* servant uniquement à ouvrir le port privilégié en BGP et à modifier la table de routage du noyau. Pour une présentation exhaustive du projet, tu peux consulter l'excellente présentation de Henning Brauer **[11]**, mon petit Gérard.

2. INSTALLATION DE BASE

Pour commencer, Gérard, il te faut 2 systèmes OpenBSD fraîchement installés. Je te fais confiance. Moi, j'ai fait le choix d'OpenBSD 4.0 que tu trouveras dans toutes les bonnes crémeries.

Ces 2 serveurs doivent avoir 3 interfaces réseau physiques chacun. Nous allons les configurer comme sur le schéma 1, page suivante.

Il y a donc plusieurs choses à configurer sur chaque serveur :

- une adresse vers le domaine public ;
- une adresse vers le domaine privé que l'on utilise comme adresse dite « d'administration » ;

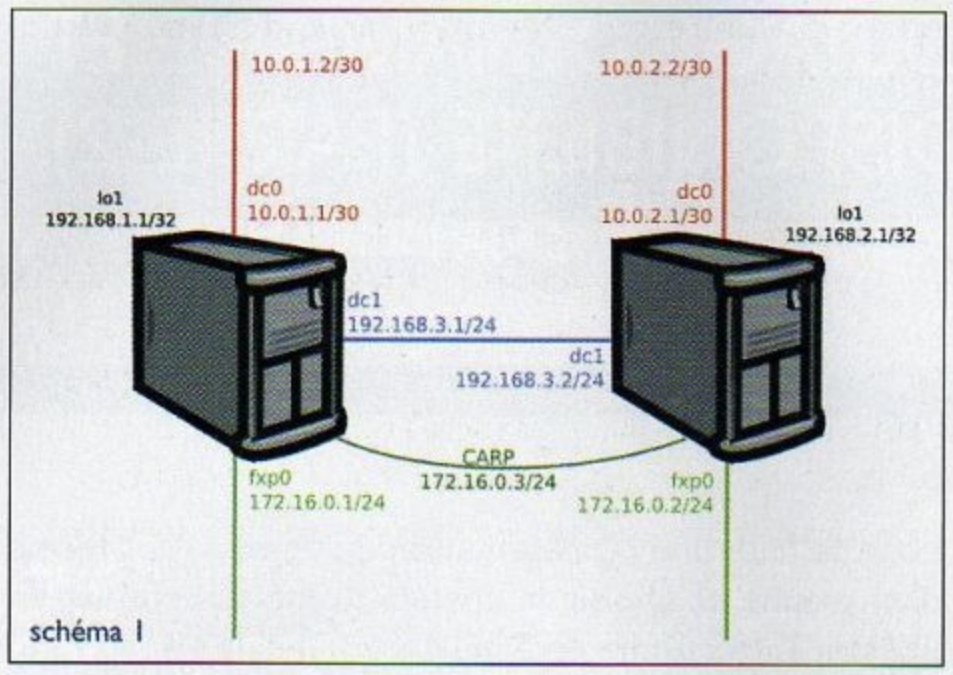


schéma 1

- une adresse « flottante » qui sera associée aux 2 interfaces privées grâce à CARP **[8]** ;
- une adresse de *back-to-back* qui connectera les 2 serveurs ;
- une *loopback* ;

Les 2 serveurs s'appellent « openbgp01 » et « openbgp02 ». Les adresses choisies sont les suivantes :

- openbgp01 :
  - → adresse publique : `10.0.1.1/30` ;
  - → adresse privée : `172.16.0.1/24` ;
  - → adresse de loopback : `192.168.1.1/32` ;
  - → adresse de back-to-back : `192.168.3.1/24` ;
  - → passerelle par défaut : `10.0.1.2/30`.
- openbgp02 :
  - → adresse publique : `10.0.2.1/30` ;
  - → adresse privée : `172.16.0.2/24` ;
  - → adresse de loopback : `192.168.2.1/32` ;
  - → adresse de back-to-back : `192.168.3.2/24` ;
  - → passerelle par défaut : `10.0.2.2/30`.

L'adresse flottante utilisée sera `172.16.0.3/24`. C'est un peu long tout ça, mais plus c'est long, ben… plus c'est long à configurer.

2.1 MATÉRIEL

Les machines utilisées comme routeurs dans cet article sont des PC équipés de 2 processeurs PIII 800 et de 2 Go de RAM sur une carte mère ASUS CUR-DLS. Les interfaces réseau sont le contrôleur Ethernet de type Intel EtherExpress PRO/100 intégré à la carte mère et une carte 4 ports Ethernet D-LINK DFE-570TX.

2.2 RÉSEAU

Cela donne une configuration de ce type :

```
openbgp01# # ifconfig
fxp0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        inet 172.16.0.1 netmask 0xffffff00 broadcast 172.16.0.255
dc0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        inet 10.0.1.1 netmask 0xfffffffc broadcast 10.0.1.3
dc1: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        inet 192.168.3.1 netmask 0xffffff00 broadcast 192.168.3.255
lo1: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 33224
        groups: lo
        inet 192.168.1.1 netmask 0xffffffff

openbgp02# ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 33224
        inet 127.0.0.1 netmask 0xff000000
fxp0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        inet 172.16.0.2 netmask 0xffffff00 broadcast 172.16.0.255
dc0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        inet 10.0.2.1 netmask 0xfffffffc broadcast 10.0.2.3
dc1: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        inet 192.168.3.2 netmask 0xffffff00 broadcast 192.168.3.255
lo1: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 33224
        groups: lo
        inet 192.168.2.1 netmask 0xffffffff
```

2.3 ROUTAGE

Il faut que nos serveurs se comportent comme des routeurs, il faut donc activer le *forwarding* IP via `sysctl` et faire en sorte que la configuration soit présente en cas de redémarrage en la mettant dans `/etc/sysctl.conf`.

```
# sysctl -w net.inet.ip.forwarding=1
```

2.4 CARP ET ÇA FLOTTE

On met en place l'adresse flottante sur nos 2 serveurs grâce à CARP. On décide qu'openbgp01 sera *master* et devra récupérer ce rôle en cas de retour après panne. Vérifions que CARP est bien activé sur nos serveurs :

```
# sysctl net.inet.carp.allow
net.inet.carp.allow=1
```

On peut aussi éventuellement activer les logs d'erreurs via la variable `net.inet.carp.log`.

```
openbgp01# ifconfig carp0 create
openbgp01# ifconfig carp0 vhid 1 advskew 1 pass \
  _0p3nBGP carpdev fxp0 172.16.0.3 netmask 255.255.255.0
openbgp02# ifconfig carp0 create
openbgp02# ifconfig carp0 vhid 1 advskew 100 pass \
  _0p3nBGP carpdev fxp0 172.16.0.3 netmask 255.255.255.0
openbgp01# ifconfig carp0

carp0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        lladdr 00:00:5e:00:01:01
        carp: MASTER carpdev fxp0 vhid 1 advbase 1 advskew 1
        groups: carp
        inet6 fe80::200:5eff:fe00:101%carp0 prefixlen 64 scopeid 0xb
        inet 172.16.0.3 netmask 0xffffff00 broadcast 172.16.0.255
openbgp02# ifconfig carp0

carp0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        lladdr 00:00:5e:00:01:01
        carp: BACKUP carpdev fxp0 vhid 1 advbase 1 advskew 100
        groups: carp
        inet6 fe80::200:5eff:fe00:101%carp0 prefixlen 64 scopeid 0xb
        inet 172.16.0.3 netmask 0xffffff00 broadcast 172.16.0.255
```

openbgp01 est bien master dès le départ, donc tout est ok. On fait en sorte que cette machine récupère

toujours le statut de master grâce à la variable `net.inet.carp.preempt`.

```
openbgp01# sysctl -w net.inet.carp.preempt=1
```

Si tout fonctionne correctement, on colle tout ça dans `/etc/sysctl.conf` et `/etc/hostname.carp0`.

2.5 IFSTATED

On pourrait associer `ifstated` **[9]** à cette configuration pour faire basculer l'adresse flottante sur openbgp02 lorsque des problèmes se posent sur l'interface publique d'openbgp01, mais c'est exactement le résultat que nous allons obtenir par la suite avec OSPF et BGP.

2.5 UN INSTANT KODAK

Comme le but ici est quand même de s'intéresser au routage, on va prendre une jolie photo des tables de routage. Mets-toi juste à côté Gérard.

```
openbgp01# netstat -nrf inet
Routing tables

Internet:
Destination  Gateway           Flags Refs  Use   Mtu Interface
default      10.0.1.2          UGS      1  831     -  dc0
10.0.1.0/30  link#2            UC       1    0     -  dc0
10.0.1.2     00:10:db:ff:20:80 UHLc     1    0     -  dc0
127/8        127.0.0.1         UGRS     0    0 33224  lo0
127.0.0.1    127.0.0.1         UH       0    0 33224  lo0
192.168.1.1  192.168.1.1       UH       0    0 33224  lo1
192.168.3/24 link#3            UC       0    0     -  dc1
172.16.0/24  link#1            UC       0    0     -  fxp0
172.16.0.3   172.16.0.3        UH       0    0     -  carp0
224/4        127.0.0.1         URS      0    0 33224  lo0
```

```
openbgp02# netstat -nrf inet
Routing tables

Internet:
Destination  Gateway           Flags Refs  Use   Mtu Interface
default      10.0.2.2          UGS      1  831     -  dc0
10.0.2.0/30  link#2            UC       1    0     -  dc0
10.0.2.2     00:10:db:ff:20:80 UHLc     1    0     -  dc0
127/8        127.0.0.1         UGRS     0    0 33224  lo0
127.0.0.1    127.0.0.1         UH       0    0 33224  lo0
192.168.2.1  192.168.2.1       UH       0    0 33224  lo1
192.168.3/24 link#3            UC       0    0     -  dc1
172.16.0/24  link#1            UC       0    0     -  fxp0
172.16.0.3   172.16.0.3        UH       0    0     -  carp0
224/4        127.0.0.1         URS      0    0 33224  lo0
```

Pour information, la commande `route -n show -inet` donne exactement le même résultat sous OpenBSD.

3. VENEZ-EN AU FAIT MON VIEUX !

Eh oui, c'est vrai, on tourne autour du pot, hein, et à force de tourner, eh bien, on en perd l'équilibre, c'est assez désolant. Donc, on y va, on installe les *daemons* qui nous intéressent et on se lance dans la bataille. Gérard, c'est pas parce que tu apprends que tu peux te permettre de rester à me regarder comme un flan, on se bouge mon vieux.

3.1 INSTALLATION

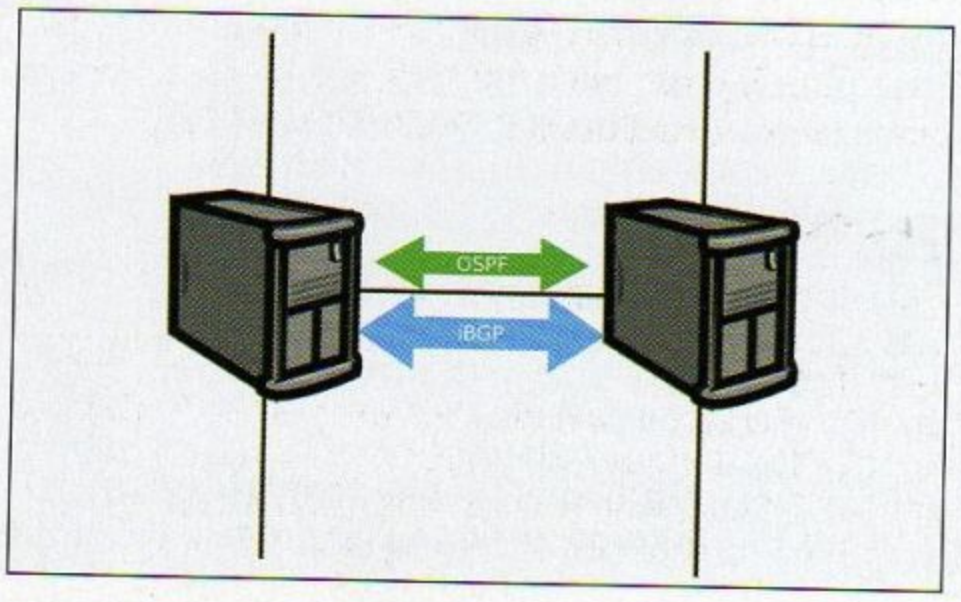
On installe donc les 2 daemons qui vont nous permettre de mettre en place notre routage dynamique. Cette installation se fait sur les 2 routeurs. Sur un OpenBSD 4.0, `openbgpd` et `openospfd` sont déjà installés dans le système de base. On reproduit la manipulation d'installation au cas où ils ne seraient pas installés sur votre système.

```
# mkdir src
# cd src
# wget ftp://ftp.crans.org/pub/OpenBSD/OpenBGPD/openbgpd-4.0.tgz
# tar zxf openbgpd-4.0.tgz
# cd bgpd
# make && make install
[...]
# cd ../bgpctl
# make && make install
[...]
# wget ftp://ftp.crans.org/pub/OpenBSD/OpenBGPD/openospfd-4.0.tgz
# tar zxf openospfd-4.0.tgz
# cd ospfd
# make && make install
[...]
# cd ../ospfctl
# make && make install
[...]
```

Et c'est fini. C'est d'une simplicité déconcertante mon ami. Je vous sens tout baba.

3.2 IL FAUT TOUJOURS COMMENCER TRANQUILLE : OSPF ET IBGP

Là, bien solide sur ses appuis, tu vois Gérard, la souplesse du cuisseau. C'est très important ça, la souplesse, donc tranquille, détendu et souple, mais vif tout de même, on n'est pas des fruits de mer voyons. On va commencer par monter une session OSPF pour annoncer les interfaces locales connectées et les *nexthops* connectés. Cette annonce permettra de construire une table de routage comprenant la *loopback* de chaque routeur, ce qui est important pour la suite. Ensuite, la session iBGP rentre en piste et nous permet de redistribuer les routes existant sur un routeur vers son voisin. Celui-ci décidera ensuite de la table de routage qu'il doit appliquer en connaissance de cause.



3.2.1 OSPF

Le protocole OSPF servi par le fameux `ospfd` est configuré par `/etc/ospfd.conf` :

```
loopback="192.168.1.1"
router-id $loopback
area 0.0.0.1 {
        interface dc1
        interface lo1
        interface dc0 {
                passive
        }
}
```

La macro `loopback` doit avoir pour valeur l'adresse IP de la loopback du serveur concerné. On crée une `area` avec un identifiant quelconque (ici `0.0.0.1`) et on demande à ce bloc d'annoncer l'interface publique, l'interface privée et la loopback. En outre, on précise que l'interface publique ne doit ni émettre, ni recevoir de paquets OSPF, grâce à l'option `passive`. La configuration doit être identique sur les 2 routeurs.

Je lance le daemon `ospfd` en mode non détaché et verbeux pour voir ce qu'il se passe et éventuellement détecter des erreurs. Au lancement sur openbgp01, on obtient :

```
openbgp01# ospfd -d -v

loopback = "192.168.1.1"
startup
orig_asext_lsa: 10.0.1.2/32 age 0
start_spf_timer: IDLE -> DELAY
orig_rtr_lsa: area 0.0.0.1
orig_rtr_lsa: stub net, interface dc0
orig_rtr_lsa: stub net, interface dc1
if_fsm: event UP resulted in action START and changing
  state for interface dc0 from DOWN to DOWN
if_fsm: interface lo1, event UP not expected in state LOOP
orig_rtr_lsa: area 0.0.0.1
orig_rtr_lsa: stub net, interface dc0
orig_rtr_lsa: stub net, interface dc1
if_fsm: event UP resulted in action START and changing
  state for interface dc1 from DOWN to WAIT
spf_calc: calculation started, area ID 0.0.0.1
spf_calc: calculation ended, area ID 0.0.0.1
spf_start_holdtimer: DELAY -> HOLD
spf_timer: state HOLD -> IDLE
```

Quand on lance le daemon sur openbgp02, voici ce qu'`ospfd` nous dit sur openbgp01 :

```
openbgp01nbr_fsm: event HELLO_RECEIVED
  resulted in action START_INACTIVITY_TIMER and changing
  state for neighbor ID 192.168.2.1 from DOWN to INIT
nbr_fsm: event 2_WAY_RECEIVED resulted in action EVAL
  and changing state for neighbor ID 192.168.2.1
  from INIT to EXSTA
if_act_elect: interface dc1 old dr 192.168.3.1 new dr
  192.168.3.1, old bdr none new bdr 192.168.3.2
orig_rtr_lsa: area 0.0.0.1
orig_rtr_lsa: stub net, interface dc0
orig_rtr_lsa: stub net, interface dc1
if_fsm: event NEIGHBORCHANGE resulted in action ELECT and
  changing state for interface dc1 from DR to DR
nbr_fsm: event NEGOTIATION_DONE resulted in action SNAPSHOT
   and changing state for neighbor ID 192.168.2.1
  from EXSTA to SNAP
if_act_elect: interface dc1 old dr 192.168.3.1 new
  dr 192.168.3.1, old bdr 192.168.3.2 new bdr 192.168.3.2
if_fsm: event NEIGHBORCHANGE resulted in action ELECT and
  changing state for interface dc1 from DR to DR
nbr_fsm: event SNAPSHOT_DONE resulted in action SNAPSHOT_DONE
  and changing state for neighbor ID 192.168.2.1
  from SNAP to EXCHG
nbr_fsm: event EXCHANGE_DONE resulted in action EXCHANGE_DONE
  and changing state for neighbor ID 192.168.2.1
  from EXCHG to LOAD
start_spf_timer: IDLE -> DELAY
orig_rtr_lsa: area 0.0.0.1
orig_rtr_lsa: stub net, interface dc0
orig_rtr_lsa: transit net, interface dc1
nbr_fsm: event LOADING_DONE resulted in action NOTHING and
  changing state for neighbor ID 192.168.2.1 from LOAD to FULL
spf_calc: calculation started, area ID 0.0.0.1
spf_calc: w id 192.168.2.1 type 1 has
    no link to v id 192.168.3.1 type 2
spf_calc: calculation ended, area ID 0.0.0.1
spf_start_holdtimer: DELAY -> HOLD
start_spf_timer: HOLD -> HOLDQUEUE
spf_timer: HOLDQUEUE -> DELAY
spf_calc: calculation started, area ID 0.0.0.1
spf_calc: calculation ended, area ID 0.0.0.1
spf_start_holdtimer: DELAY -> HOLD
spf_timer: state HOLD -> IDLE
```

La session est d'abord négociée entre les 2 routeurs. Ensuite, on observe que le routeur apprend les interfaces que son voisin lui annonce. Il annonce d'ailleurs que l'interface `dc0` est dans un statut « *stub* », ce qui signifie qu'elle n'est pas considérée comme connectée. Le serveur finit par calculer l'arbre des chemins pour ensuite prendre ses décisions de routage. La convergence est tout de même un peu lente, mais ça prend effet en 20 secondes environ. Il convient aussi de remarquer que les développeurs d'OpenOSPFd sont de grossiers personnages qui traitent mes interfaces de moignons (stub).

```
openbgp01# ospfctl show database


                Router Link States (Area 0.0.0.1)

Link ID         Adv Router      Age  Seq#       Checksum
192.168.1.1     192.168.1.1     79   0x80000002 0x886a
192.168.2.1     192.168.2.1     80   0x80000002 0xbc31

                Net Link States (Area 0.0.0.1)

Link ID         Adv Router      Age  Seq#       Checksum
192.168.3.2     192.168.2.1     80   0x80000001 0x9011

                Type-5 AS External Link States

Link ID         Adv Router      Age  Seq#       Checksum
10.0.1.2        192.168.1.1     123  0x80000001 0xb088
10.0.2.2        192.168.2.1     120  0x80000001 0x9f97
```

```
openbgp01# ospfctl show rib

Destination     Nexthop     Path Type  Type    Cost Uptime
192.168.2.1     192.168.3.2 Intra-Area Router  10   01:58:40
10.0.2.0/30     192.168.3.2 Intra-Area Network 20   01:58:40
192.168.2.1/32  192.168.3.2 Intra-Area Network 10   01:58:40
192.168.3.0/24  192.168.3.1 Intra-Area Network 10   01:58:47

openbgp01# netstat -nrf inet
Routing tables

Internet:
Destination     Gateway           Flags Refs Use   Mtu Interface
default         10.0.1.2          UGS      2 831     -  dc0
10.0.1.0/30     link#2            UC       1   0     -  dc0
10.0.1.2        00:10:db:ff:20:80 UHLc     1   0     -  dc0
10.0.2.0/30     192.168.3.2       UG2      1   0     -  dc1
127/8           127.0.0.1         UGRS     0   0 33224  lo0
127.0.0.1       127.0.0.1         UH       0   0 33224  lo0
192.168.1.1     192.168.1.1       UH       0   0 33224  lo1
192.168.2.1/32  192.168.3.2       UG2      0   0     -  dc1
192.168.3/24    link#3            UC       1   0     -  dc1
192.168.3.2     00:80:c8:c9:9a:72 UHLc     2  28     -  dc1
172.16.0/24     link#1            UC       0   0     -  fxp0
172.16.0.3      172.16.0.3        UH       0   0     -  carp0
224/4           127.0.0.1         URS      0   0 33224  lo0
```

Mais regardez-moi ça, vous savez ce que c'est ? Ce sont de belles routes vers openbgp02 :

- une route pour 10.0.2.0/30 via le back-to-back ;
- une route pour 192.168.2.1/32 toujours via le back-to-back ;
- la correspondance IP/adresse mac pour cette adresse de back-to-back.

Maintenant qu'on est sûr que le protocole a correctement négocié, on peut relancer `ospfd` en mode détaché sur les 2 routeurs et le rajouter dans le `/etc/rc.conf`.

```
ospfd=YES
ospfd_flags=NO
```

3.2.2 iBGP

Allez Gérard, on s'attaque à l'iBGP maintenant. Tu vas voir, c'est beau tout plein ! Maintenant qu'on connaît les loopbacks du routeur d'à côté, on va monter la session iBGP qui nous permettra en cas de problème de rediriger le trafic via le back-to-back. C'est un premier pas vers un système sécurisé en réseau. La configuration de `bgpd` via le fichier `/etc/bgpd.conf` est la suivante :

```
peer_ibgp="192.168.2.1"
loopback="192.168.1.1"
as_ibgp="65001"

# global configuration
AS 65001
listen on $loopback
network connected

# iBGP
group "IBGP" {
        remote-as $as_ibgp
        neighbor $peer_ibgp {
                descr "lo openbgp02"
                local-address $loopback
        }
        announce all
}
```

Tu vas voir, on va faire exactement comme tout à l'heure. On lance d'abord `bgpd` en mode non détaché et verbeux sur openbgp01 :

```
openbgp01# bgpd -d -v
peer_ibgp = "192.168.2.1"
loopback = "192.168.1.1"
as_ibgp = "65001"
startup
route decision engine ready
listening on 192.168.1.1
session engine ready
neighbor 192.168.2.1 (lo openbgp02):
  state change None -> Idle, reason: None
neighbor 192.168.2.1 (lo openbgp02):
  state change Idle -> Connect, reason: Start
neighbor 192.168.2.1 (lo openbgp02):
  socket error: Connection refused
neighbor 192.168.2.1 (lo openbgp02):
  state change Connect -> Active, reason:
  Connection open failed
```

Et maintenant, on le lance sur openbgp02 et on attend qu'ils se mettent d'accord et qu'ils soient vraiment contents :

```
neighbor 192.168.2.1 (lo openbgp02):
  state change Active -> OpenSent,
  reason: Connection opened
neighbor 192.168.2.1 (lo openbgp02):
  state change OpenSent -> OpenConfirm,
  reason: OPEN message received
neighbor 192.168.2.1 (lo openbgp02):
  state change OpenConfirm -> Established,
  reason: KEEPALIVE message received
nexthop 192.168.2.1 now valid: via 192.168.3.2
```

Le serveur BGP sur openbgp01 confirme qu'il connaît le voisin openbgp02 et qu'il reçoit les informations via le back-to-back. Voyons maintenant tout ce que ça a pu générer sur les 2 routeurs. D'abord l'état des sessions BGP :

```
openbgp01# bgpctl show summary
Neighbor     AS    MsgRcvd MsgSent OutQ Up/Down  State/PrefixRcvd
lo openbgp02 65001       6       5    0 00:01:18      1

openbgp02# bgpctl show summary
Neighbor     AS    MsgRcvd MsgSent OutQ Up/Down  State/PrefixRcvd
lo openbgp01 65001      10       9    0 00:02:11      3
```

Une seule route a été échangée avec openbgp02. Par contre, openbgp02 a appris des routes de la part d'openbgp01.

Nous allons voir, par la suite, quelles sont ces routes et leur signification.

Ensuite, les tables de routes apprises via BGP sur openbgp01 et openbgp02 :

```
openbgp01# bgpctl show rib
flags: * = Valid, > = Selected, I = via IBGP, A = Announced
origin: i = IGP, e = EGP, ? = Incomplete

flags destination    gateway     lpref   med aspath origin
AI*>  10.0.1.0/30    0.0.0.0       100     0 i
I*>   10.0.2.0/30    192.168.2.1   100     0 i
AI*>  192.168.3.0/24 0.0.0.0       100     0 i
AI*>  172.16.0.0/24  0.0.0.0       100     0 i
```

Le routeur openbgp01 annonce (A) toutes ses routes apprises via OSPF (I), à l'exception de la route qui annonce le chemin vers internet via openbgp02.

```
openbgp02# bgpctl show rib
flags: * = Valid, > = Selected, I = via IBGP, A = Announced
origin: i = IGP, e = EGP, ? = Incomplete

flags destination    gateway     lpref   med aspath origin
I*>   10.0.1.0/30    192.168.1.1   100     0 i
AI*>  10.0.2.0/30    0.0.0.0       100     0 i
I*>   192.168.3.0/24 192.168.1.1   100     0 i
AI*   192.168.3.0/24 0.0.0.0       100     0 i
I*>   172.16.0.0/24  192.168.1.1   100     0 i
AI*   172.16.0.0/24  0.0.0.0       100     0 i
```

La différence entre les 2 tables RIB est un choix fait par OpenBGPd, a priori basé sur le `router-id` de chaque routeur. Ne l'ayant pas spécifié, OpenBGPd a fait son choix seul. Le `router-id` constitue le dernier critère de décision dans l'algorithme de construction de la table de routage par BGP.

```
openbgp01# netstat -nrf inet
Routing tables

Internet:
Destination    Gateway           Flags Refs Use   Mtu  Interface
default        10.0.1.2          UGS      2 831     -  dc0
10.0.1.0/30    link#2            UC       1   0     -  dc0
10.0.1.2       00:10:db:ff:20:80 UHLc     1   0     -  dc0
10.0.2.0/30    192.168.3.2       UG2      0   0     -  dc1
127/8          127.0.0.1         UGRS     0   0 33224  lo0
127.0.0.1      127.0.0.1         UH       0   0 33224  lo0
192.168.1.1    192.168.1.1       UH       0   0 33224  lo1
192.168.2.1/32 192.168.3.2       UG2      1  28     -  dc1
192.168.3/24   link#3            UC       1   0     -  dc1
192.168.3.2    00:80:c8:c9:9a:72 UHLc     2  15     -  dc1
172.16.0/24    link#1            UC       0   0     -  fxp0
172.16.0.3     172.16.0.3        UH       0   0     -  carp0
224/4          127.0.0.1         URS      0   0 33224  lo0
```

Aucune modification de la table de routage sur openbgp01, puisque toutes les routes locales sont « *connected* ». Si un problème survenait, de nouvelles décisions de routage seraient prises sur la base des routes connues. Il est à noter que la route vers le sous-réseau public 10.0.2.0/30 est connue à la fois via OSPF et via BGP.

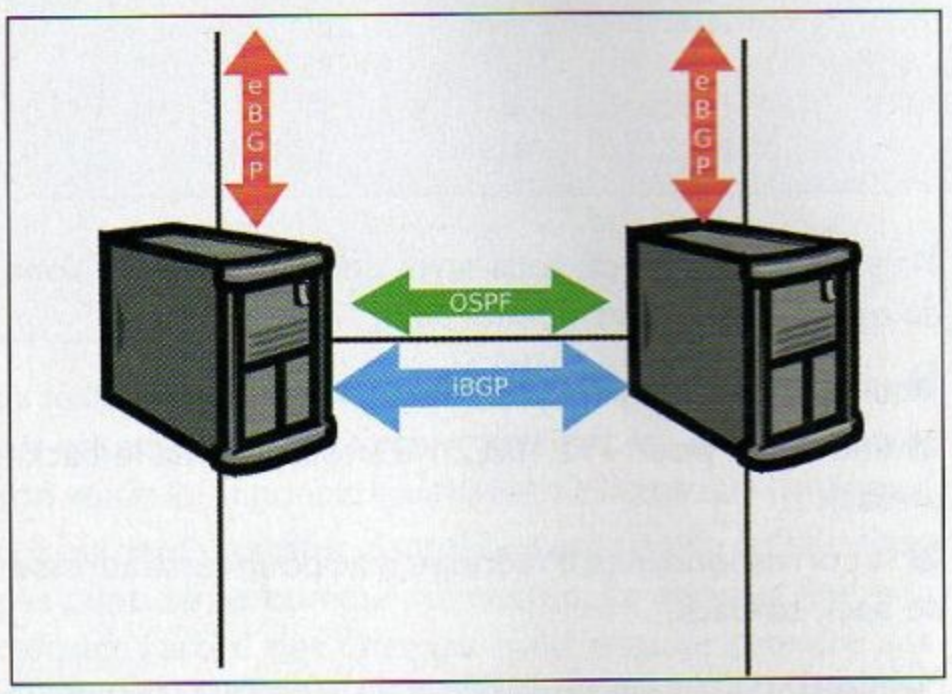
Le résultat est satisfaisant, donc on peut prévoir le lancement de `bgpd` au démarrage via `/etc/rc.conf`.

```
bgpd=YES
bgpd_flags=NO
```

Gérard, mon petit, je sens que tu veux crier victoire. Tu te dis qu'il n'y a rien de tel que d'aller chez Azzedine Alaia ou même de s'acheter des sous-pulls chez Yohji Yamamoto.

3.3 ET LÀ ON EST CHAUD, ALORS ON SE LANCE : OSPF, IBGP ET EBGP

Excuse-moi de te dire ça, mon pauvre Gérard, mais tu confonds un peu tout, tu fais un amalgame entre la coquetterie et la classe. Tu es fou ! La vraie classe, c'est de monter une session BGP avec un autre AS et d'apprendre Internet.



On rajoute maintenant une session eBGP avec un routeur d'un autre AS. La configuration du daemon `bgpd`, sur openbgp01 uniquement pour le moment, donne :

```
peer_ebgp="10.0.1.2"
public="10.0.1.1"
as_ebgp="65002"

listen on $public
network connected

# eBGP
group "EBGP" {
        remote-as $as_ebgp
        neighbor $peer_ebgp {
                descr "peer eBGP"
                local-address $public
        }
        announce none
}
```

Ce rajout au fichier de configuration `/etc/bgpd.conf` est à peu de chose près identique à la configuration iBGP. La grande différence est que le numéro d'AS du voisin est différent du numéro d'AS d'openbgp01. Il est également précisé qu'on ne va annoncer aucune route vers le

routeur BGP voisin. On ne fera qu'apprendre les routes qu'il veut bien nous envoyer.

On peut recharger le fichier de configuration via la commande `bgpctl reload` et on ne va pas s'en priver.

```
[...openbgp01...]
rereading config
peer_ibgp = "192.168.2.1"
peer_ebgp = "10.0.1.2"
loopback = "192.168.1.1"
public = "10.0.1.1"
as_ibgp = "65001"
as_ebgp = "65002"
nexthop 192.168.2.1 now valid: via 192.168.3.2
RDE reconfigured
listening on 10.0.1.1
SE reconfigured
neighbor 10.0.1.2 (peer eBGP):
  state change None -> Idle,
  reason: None
neighbor 10.0.1.2 (peer eBGP):
  state change Idle -> Connect,
  reason: Start
neighbor 10.0.1.2 (peer eBGP):
  state change Connect -> OpenSent,
  reason: Connection opened
neighbor 10.0.1.2 (peer eBGP):
  state change OpenSent -> OpenConfirm,
  reason: OPEN message received
neighbor 10.0.1.2 (peer eBGP):
  received notification: error in OPEN message,
  unsupported capability
neighbor 10.0.1.2 (peer eBGP):
  received "unsupported capability"
  notification without data part,
  disabling capability announcements altogether
neighbor 10.0.1.2 (peer eBGP):
  state change OpenConfirm -> Idle,
  reason: NOTIFICATION received
neighbor 10.0.1.2 (peer eBGP):
  state change Idle -> Connect,
  reason: Start
neighbor 10.0.1.2 (peer eBGP):
  state change Connect -> OpenSent,
  reason: Connection opened
neighbor 10.0.1.2 (peer eBGP):
  state change OpenSent -> OpenConfirm,
  reason: OPEN message received
neighbor 10.0.1.2 (peer eBGP):
  state change OpenConfirm -> Established,
  reason: KEEPALIVE message received
```

Le *peer* eBGP envoie des messages de confirmation. Les deux routeurs se mettent d'accord sur les options qu'ils supportent l'un et l'autre et ouvrent la session.

```
[...openbgp02...]
nexthop 10.0.1.2 now valid: via 192.168.3.1
```

Le serveur openbgp02 nous signale, quant à lui, qu'il a appris un nouveau nexthop via le back-to-back.

```
openbgp01# bgpctl show summary
Neighbor     AS    MsgRcvd MsgSent OutQ Up/Down  State/PrefixRcvd
peer eBGP    65002   56068      10    0 00:02:42 203179
lo openbgp02 65001      45   55779    0 00:20:58      1
```

Les routes de l'AS 65002 ont bien été apprises par openbgp01 via la session eBGP. Ici le routeur a récupéré une table de 203179 routes. Il est normal de constater des fluctuations dans ce nombre de routes. C'est le grand Ternet, ça vit.

```
openbgp02# bgpctl show summary

Neighbor     AS    MsgRcvd MsgSent OutQ Up/Down  State/PrefixRcvd
lo openbgp01 65001   55774      47    0 00:21:49 203182
```

Et ces routes ont bien été propagées à openbgp02 via la session iBGP entre les deux routeurs OpenBGPd. Openbgp02 se retrouve avec 3 routes de plus dans sa table. Ce sont les routes qui ont été propagées entre les 2 routeurs en iBGP dans le chapitre précédent.

On va maintenant activer la même configuration sur openbgp02 pour le connecter à son peer eBGP. On force la relecture du fichier de configuration et on observe le résultat.

```
[...openbgp02...]
rereading config
peer_ibgp = "192.168.1.1"
peer_ebgp = "10.0.2.2"
loopback = "192.168.2.1"
public = "10.0.2.1"
as_ibgp = "65001"
as_ebgp = "65002"
nexthop 10.0.1.2 now valid: via 192.168.3.1
nexthop 192.168.1.1 now valid: via 192.168.3.1
SE reconfigured
RDE reconfigured
neighbor 10.0.2.2 (peer eBGP):
  state change Idle -> Connect,
  reason: Start
neighbor 10.0.2.2 (peer eBGP):
  state change Connect -> OpenSent,
  reason: Connection opened
neighbor 10.0.2.2 (peer eBGP):
  state change OpenSent -> OpenConfirm,
  reason: OPEN message received
neighbor 10.0.2.2 (peer eBGP):
  received notification: error in OPEN message,
  unsupported capability
neighbor 10.0.2.2 (peer eBGP):
  received "unsupported capability"
  notification without data part,
  disabling capability announcements altogether
neighbor 10.0.2.2 (peer eBGP):
  state change OpenConfirm -> Idle,
  reason: NOTIFICATION received
neighbor 10.0.2.2 (peer eBGP):
  state change Idle -> Active,
  reason: Start
neighbor 10.0.2.2 (peer eBGP):
  state change Active -> OpenSent,
  reason: Connection opened
neighbor 10.0.2.2 (peer eBGP):
  state change OpenSent -> OpenConfirm,
  reason: OPEN message received
neighbor 10.0.2.2 (peer eBGP):
  state change OpenConfirm -> Established,
  reason: KEEPALIVE message received
```

La session est négociée entre openbgp02 et son point de *peering* BGP. On remarque que le routeur openbgp02 se connecte au même AS qu'openbgp01. Il aurait été possible de mettre en place une configuration de multihoming avec connexion à 2 AS différents.

```
[...openbgp01...]
nexthop 10.0.2.2 now valid: via 192.168.3.2
```

Le routeur openbgp01 voit bien un nouveau nexthop via le back-to-back d'openbgp02.

```
openbgp02# bgpctl show summary
Neighbor     AS    MsgRcvd MsgSent OutQ  Up/Down  State/PrefixRcvd
peer eBGP    65002   59538      33    0  00:11:05 203136
lo openbgp01 65001   58682   59145    0  00:59:14 203138
```

```
openbgp01# bgpctl show summary
Neighbor     AS    MsgRcvd MsgSent OutQ  Up/Down  State/PrefixRcvd
peer eBGP    65002   59133      86    0  00:40:50 203134
lo openbgp02 65001   59143   58680    0  00:59:06 203135
```

Un certain nombre de routes ont été apprises par openbgp02 en eBGP et ces routes ont été propagées à openbgp01 via la session iBGP.

```
openbgp01# bgpctl show rib | head
flags: * = Valid, > = Selected, I = via IBGP, A = Announced
origin: i = IGP, e = EGP, ? = Incomplete

flags destination gateway  lpref med aspath origin
*>    3.0.0.0/8   10.0.1.2   100   0 65002 xASx xASx xASx xASx xASx xASx i
I*    3.0.0.0/8   10.0.2.2   100   0 65002 xASx xASx xASx xASx xASx xASx i
*>    4.0.0.0/8   10.0.1.2   100   0 65002 xASx xASx xASx i
I*    4.0.0.0/8   10.0.2.2   100   0 65002 xASx xASx xASx i
*>    4.0.0.0/9   10.0.1.2   100   0 65002 xASx xASx xASx i
I*    4.0.0.0/9   10.0.2.2   100   0 65002 xASx xASx xASx i
```

Le routeur openbgp01 connaît maintenant un certain nombre de routes vers Internet. Ces routes passent par l'un des deux routeurs BGP de l'AS 65002 avec lesquels nous nous sommes connectés. La table de routage déduite de cette table d'apprentissage fait état des décisions de routage qui ont été prises.

```
openbgp01# netstat -nrf inet | head
Routing tables

Internet:
Destination Gateway  Flags  Refs   Use    Mtu  Interface
default     10.0.1.2   UGS     3  1444      -   dc0
3/8         10.0.1.2   UG1     0     0      -   dc0
4.0/9       10.0.1.2   UG1     0     0      -   dc0
4/8         10.0.1.2   UG1     0     0      -   dc0
4.21.41/24  10.0.1.2   UG1     0     0      -   dc0
4.23.112/24 10.0.1.2   UG1     0     0      -   dc0
```

Toutes les routes que l'on observe passent par le routeur BGP de l'AS avec lequel on est connecté. En fait, ce sont toutes les routes que l'on a apprises de ce routeur BGP.

Vois-tu Gérard, maintenant que tu connais Internet et que tu l'as appris, il te faut changer de comportement. Tu es devenu un homme, maintenant, et tu ne verras plus le monde avec les mêmes yeux. Le jour où ça m'est arrivé, je me suis exclamé : « Mais je suis pas super fort, je suis mieux que ça même, je suis surpuissant. »

3.4 ET ÇA, C'EST POUR MOI

C'est pour ma pomme, ouais, tu me plais bien Gérard et j'avoue que j'ai bien envie de faire un petit quelque chose pour toi. Alors je vais te parler de Juniper Networks **[12] [13]**, une marque de routeurs. Pourquoi Juniper ? Tout simplement parce que le système de base qui fait tourner ces machines est dérivé de FreeBSD, et là tu vois où je veux en venir.

Très rapidement, nous allons considérer que les routeurs en face de nos serveur OpenBGPd sont des routeurs Juniper (type M7i ou M10i), respectivement jun1 comme peer BGP d'openbgp01 et jun2 connecté avec openbgp02. La configuration des sessions BGP se fait de la façon suivante :

```
hr@jun1> configure
Entering configuration mode

[edit]
hr@jun1# edit routing-options

[edit routing-options]
hr@jun1# set autonomous-system 65002

[edit routing-options]
hr@jun1# top

[edit]
hr@jun1# edit protocols bgp group OPENBGP01

[edit protocols bgp group OPENBGP01]
hr@jun1# set local-as 65002

[edit protocols bgp group OPENBGP01]
hr@jun1# set peer-as 65001

[edit protocols bgp group OPENBGP01]
hr@jun1# set local-address 10.0.1.2

[edit protocols bgp group OPENBGP01]
hr@jun1# set neighbor 10.0.1.1

[edit protocols bgp group OPENBGP01]
hr@jun1# set import REJECT

[edit protocols bgp group OPENBGP01]
hr@jun1# set type external

[edit protocols bgp group OPENBGP01]
hr@jun1# set family inet unicast

[edit protocols bgp group OPENBGP01]
hr@jun1# set export [ OPENBGP REJECT ]

[edit protocols bgp group OPENBGP01]
hr@jun1# show
type external;
local-address 10.0.1.2;
import REJECT;
family inet {
```

```
    unicast;
}
export [ OPENBGP REJECT ];
peer-as 65001;
local-as 65002;
neighbor 10.0.1.1;

[edit protocols bgp group OPENBGP01]
hr@jun1# commit check
configuration check succeeds

[edit protocols bgp group OPENBGP01]
hr@jun1# commit
commit complete
```

OPENBGP correspond à un policy-statement (edit policy-options policy-statement OPENBGP) qui ne permet que l'envoi des routes agrégées et dont l'AS-path contient notre AS (65002). On remarque également qu'on peut définir l'AS du routeur de façon globale ou pour chaque groupe BGP.

```
[edit]
hr@jun2# load merge terminal
[Type ^D at a new line to end input]
protocols {
    bgp {
        group OPENBGP02 {
            type external;
            local-address 10.0.2.2;
            import REJECT;
            family inet {
                unicast;
            }
            export [ OPENBGP REJECT ];
            peer-as 65001;
            local-as 65002;
            neighbor 10.0.2.1;
        }
    }
}
load complete

[edit]
hr@jun2#
```

Une fois que les 2 routeurs sont configurés, et que les configurations sont validées, on peut monter les sessions eBGP avec leurs voisins de l'AS 65001 Les sessions une fois montées, les routeurs Juniper vont émettre leurs tables de routes. La connexion peut mettre un certain temps à s'initialiser (on n'apprend pas 200000 routes en un battement de paupière), d'où les deux états de la commande suivante :

```
hr@jun1> show bgp summary
Groups: 3 Peers: 3 Down peers: 1
Table    Tot Paths  Act Paths Suppr History Damp State Pending
inet.0      406022     202986     0       0          0       0
bgp.l3vpn.0      0          0     0       0          0       0

Peer                AS    InPkt    OutPkt OutQ Flaps Last Up/Dwn
State|#Active/Received/Damped...
aaa.bbb.ccc.ddd  xASx    83872      2837    0    13    23:35:24
202986/203000/0      0/0/0
eee.fff.ggg.hhh 65002 11125157  14254540    0     0     50w3d6h Establ
  inet.0: 0/203022/0
  bgp.l3vpn.0: 0/0/0
10.0.1.1        65001        0         0    0     0       14:29 Active

sbe@CLB-JUN3> show bgp summary
Groups: 3 Peers: 3 Down peers: 0
Table    Tot Paths  Act Paths Suppr History Damp State Pending
inet.0      405989     202968     0       0          0       0
bgp.l3vpn.0      0          0     0       0          0       0

Peer                AS    InPkt    OutPkt OutQ Flaps Last Up/Dwn
State|#Active/Received/Damped...
aaa.bbb.ccc.ddd  xASx    84026      2844    0    13    23:39:20
202968/202982/0      0/0/0
eee.fff.ggg.hhh 65002 11125275 14254681    0     0     50w3d6h Establ
  inet.0: 0/203004/0
  bgp.l3vpn.0: 0/0/0
10.0.1.1        65001        2    51126    0     0          51 0/3/0
0/0/0
```

On peut obtenir des informations sur les voisins BGP via la commande bgpctl show neighbor <adresse_voisin>. Ici, on utiliserait 10.0.1.2 sur openbgp01 par exemple.

```
openbgp01# bgpctl show neighbor 10.0.1.2
BGP neighbor is 10.0.1.2, remote AS 65002
 Description: peer eBGP
  BGP version 4, remote router-id 10.0.1.2
  BGP state = Established, up for 01:29:51
  Last read 00:00:01, holdtime 90s, keepalive interval 30s
  Neighbor capabilities:
    Multiprotocol extensions: IPv4 Unicast
    Route Refresh

  Message statistics:
                  Sent       Received
  Opens              2              2
  Notifications      0              1
  Updates            1          61784
  Keepalives       181             13
  Route Refresh      0              0
  Total            184          61800

  Local host:          10.0.1.1, Local port:  43697
  Remote host:         10.0.1.2, Remote port:   179
```

3.5 QUELLE JOUISSANCE DE DÉTRUIRE UN CHÂTEAU DE SABLE !

Maintenant qu'on a monté toute cette superbe mécanique, on va en casser un petit bout, juste pour voir. Gérard, tu m'as l'air intéressé, allez, tiens, regarde, je mets ma clé et tu mets la tienne. On tourne ensemble à **3, 1, 2, 3, click.**

Nous allons maintenant voir comment se comporte notre routage lorsqu'une des sessions eBGP tombe, on va faire tomber la session du routeur openbgp01 et observer comment les tables de routage sont recalculées.

```
openbgp01# bgpctl neighbor 10.0.1.2 down
request processed
```

```
openbgp01# bgpctl show summary
Neighbor     AS    MsgRcvd MsgSent OutQ  Up/Down  State/PrefixRcvd
peer eBGP    65002   62023     202    0  00:00:47 Idle
lo openbgp02 65001   61584   61633    0  01:57:13 203152
```

```
[...openbgp01...]
neighbor 10.0.1.2 (peer eBGP):
  state change Established -> Idle,
  reason: Stop
Connection attempt from neighbor 10.0.1.2
  (peer eBGP) while session is in state Idle
```

La session est bien tombée, les logs nous le disent. La commande `bgp show summary` donne aussi des informations sur l'état de la session qui est notée comme en attente.

```
openbgp01# bgpctl show rib | head
flags: * = Valid, > = Selected, I = via IBGP, A = Announced
origin: i = IGP, e = EGP, ? = Incomplete

flags destination   gateway  lpref med aspath origin
I*>   3.0.0.0/8     10.0.2.2   100   0 65002 xASx xASx xASx xASx xASx
xASx i
I*>   4.0.0.0/8     10.0.2.2   100   0 65002 xASx xASx xASx i
I*>   4.0.0.0/9     10.0.2.2   100   0 65002 xASx xASx xASx i
I*>   4.21.41.0/24  10.0.2.2   100   0 65002 xASx xASx xASx xASx xASx i
I*>   4.23.112.0/24 10.0.2.2   100   0 65002 xASx xASx xASx xASx xASx i
I*>   4.23.113.0/24 10.0.2.2   100   0 65002 xASx xASx xASx xASx xASx i
openbgp01# netstat -nrf inet | head
Routing tables

Internet:
Destination Gateway     Flags  Refs   Use    Mtu  Interface
default     10.0.1.2      UGS     3  6332      -   dc0
3/8         192.168.3.2   UG1     0     0      -   dc1
4.0/9       192.168.3.2   UG1     0     0      -   dc1
4/8         192.168.3.2   UG1     0     0      -   dc1
4.21.41/24  192.168.3.2   UG1     0     0      -   dc1
4.23.112/24 192.168.3.2   UG1     0     0      -   dc1
```

La table d'apprentissage BGP et la table de routage nous signalent que tout le trafic doit maintenant transiter par openbgp02.

```
openbgp02# bgpctl show rib | head
flags: * = Valid, > = Selected, I = via IBGP, A = Announced
origin: i = IGP, e = EGP, ? = Incomplete

flags destination   gateway lpref med aspath origin
*>    3.0.0.0/8     10.0.2.2  100   0 65002 xASx xASx xASx xASx xASx xASx i
*>    4.0.0.0/8     10.0.2.2  100   0 65002 xASx xASx xASx i
*>    4.0.0.0/9     10.0.2.2  100   0 65002 xASx xASx xASx i
*>    4.21.41.0/24  10.0.2.2  100   0 65002 xASx xASx xASx xASx xASx i
*>    4.23.112.0/24 10.0.2.2  100   0 65002 xASx xASx xASx xASx xASx i
*>    4.23.113.0/24 10.0.2.2  100   0 65002 xASx xASx xASx xASx xASx i
```

Le routeur openbgp02 ne connaît plus que les routes apprises par la session eBGP. Les routes apprises en eBGP par openbgp01 et transmises via l'iBGP n'apparaissent plus dans la table d'apprentissage BGP. Nous avons bien obtenu une sécurité en cas de panne d'un des liens vers Internet. Cette sécurité se déclenche même lorsque c'est simplement la session eBGP qui est perdue.

On peut remonter le lien eBGP sur openbgp01 et observer que les routes sont bien, de nouveau, apprises.

```
openbgp01# bgpctl neighbor 10.0.1.2 up

request processed
openbgp01# bgpctl show summary
Neighbor     AS    MsgRcvd MsgSent OutQ  Up/Down  State/PrefixRcvd
peer eBGP    65002  116972     208    0 00:00:44 203194
lo openbgp02 65001   61769  116320    0 02:11:28 203195
```

CONCLUSION

Nous sommes maintenant dans une situation de service en multihoming. Cette plate-forme est dorénavant tolérante à la panne d'un de ses liens vers internet. Ca rassure dans les chaumières. Gérard, je te colle un cigare au bec et tu me dis : « j'adore quand un plan se déroule sans accroc ». Si tu n'as pas compris quelque chose, n'hésite pas à te démerder tout seul et à consulter les documents cités en référence.

RÉFÉRENCES

[1] http://fr.wikipedia.org/wiki/OSPF

[2] http://fr.wikipedia.org/wiki/BGP

[3] http://en.wikipedia.org/wiki/Multihoming

[4] http://fr.wikipedia.org/wiki/Interior_Gateway_Protocol

[5] http://www.ietf.org/rfc/rfc1771.txt

[6] http://www.ietf.org/rfc/rfc2328.txt

[7] http://www.openbsd.org/

[8] http://www.openbsd.org/faq/faq6.html#CARP

[9] http://www.openbsd.org/cgi-bin/man.cgi?query=ifstated

[10] http://www.openbgpd.org/

[11] http://unduli.bsws.de/papers/euroBSDCon2004/

[12] http://fr.wikipedia.org/wiki/Juniper_Networks

[13] http://www.juniper.net

[14] http://www.bonz.org/glmf_hs0207/bgp_ospf_bsd.tar.gz

REMERCIEMENTS

Merci à Laurent « Cariboo » Guinchard pour la précieuse aide franc-comtoise qu'il m'a apportée dans la rédaction de cet article.

Merci aussi à George Abitbol dont l'esprit a toujours guidé mes pas.

Par Stéphane « KingBug » Clodic <kb+glmf_geom(at)gcu.info> et Clément « Twisla » Martin <twisla(gruika)gcu(gkruikdot)info>

UTILISATION DE GEOM AVEC FREEBSD

OU LES AVENTURES DU GEOM VERT ET DE SON AMI SPROUTY !

1. PRÉ-REQUIS : PRÉSENTATION DE GEOM

GEOM **[1] [2]** est un *framework* d'abstraction d'accès à des *raw-devices* via des classes (`CONCAT`, `ELI`, `LABEL`, `MIRROR`, `NOP`, `RAID3`, `SHSEC`, `STRIPE`, `VINUM`) ; chaque classe représente un type d'action sur ces raw-devices. Il est possible de combiner plusieurs classes sur un même device, et ce, dans différents ordres, ce qui éloigne fortement GEOM des autres « *volume managers* » qui ont une typologie beaucoup plus stricte.

Un peu de vocabulaire :

- Une *classe* implémente un type particulier d'action sur ces raw-devices (disques durs IDE, SATA, SCSI ; clés USB ; fichiers « *filesystem* » montés sur une *loopback* `mdconfig(3)`) ;
- Un *provider* est le device résultant de l'action de GEOM sur ces raw-devices ;
- Un *consumer* est le device sur lequel s'applique la classe, au départ, le device physique.

Le *handbook* comporte un chapitre très complet sur GEOM **[3]** : « *Chapter 18 GEOM : Modular Disk Transformation* ».

2. INTRODUCTION

Je vais vous conter l'histoire des aventures de Sprouty, l'ami fidèle du GEOM Vert, qui a décidé de ranger son placard à nourriture.

Sprouty est un brin désordonné, il se retrouve assez vite avec des boîtes de maïs doux plein les mains sans en reconnaître les différentes préparations, ni les dates limites de consommation. Heureusement, GEOM Vert va l'aider à s'y retrouver et va même en profiter pour lui montrer des petites astuces.

3. « GEOM VERT ! GEOM VERT ! VIENS VOIR ! » (LES BIENFAITS DE GLABEL)

Sprouty est bien embêté, il a plein de boîtes de maïs étalées partout et plus aucune n'a d'étiquette. Il avait bien constitué des petits tas de boîtes selon les dates limites de consommation et les différentes recettes, mais le chat est passé par-là et les boîtes sont toutes mélangées.

Sprouty est allé chercher son scanner à code-barres portable et commence à scanner les étiquettes de code-barres restant sur les boîtes de maïs, mais c'est quelque chose de bien laborieux, car, à chaque passage, il est obligé de regarder ce qu'affiche le `/var/log/messages` de son scanner à code-barres pour savoir si c'est une boîte de maïs qui est vue en `da0` ou en `da3`, et pouvoir ensuite utiliser le bon ouvre-boîte de la marque `mount`.

GEOM Vert, passant par-là par hasard, voit son ami complètement noyé sous ses boîtes de maïs en train de perdre beaucoup de temps à trier son garde-manger.

GEOM Vert : « Sprouty, mais que fais-tu ? »

Sprouty : « GEOM Vert ! Je suis heureux de te voir ; regarde ce tas de boîtes de maïs, je n'arrive pas à m'y retrouver. »

GEOM Vert : « Ok, je peux t'aider ? »

GEOM Vert prend alors deux boîtes de maïs identiques, les manipule et les insère dans le scanner de Sprouty :

```
% glabel label bsdkey da0

kernel: umass0: <Intelligent Stick Intelligent Stick,
        class 0/0, rev 1.10/1.00, addr 2> on uhub0
kernel: da0 at umass-sim0 bus 0 target 0 lun 0
kernel: da0: <USB Card IntelligentStick 2.01>
        Removable Direct Access SCSI-2 device
kernel: da0: 1.000MB/s transfers
kernel: da0: 255MB (522688 512 byte sectors: 64H 32S/T 255C)
kernel: GEOM_LABEL: Label for provider da0a is label/bsdkey.

kernel: umass1: <Intelligent Stick Intelligent Stick,
        class 0/0, rev 1.10/1.00, addr 3> on uhub0
kernel: da1 at umass-sim1 bus 1 target 0 lun 0
kernel: da1: <USB Card Intelligent Stic 2.02>
        Removable Direct Access SCSI-2 device
kernel: da1: 1.000MB/s transfers
kernel: da1: 255MB (522688 512 byte sectors: 64H 32S/T 255C)
```

GEOM VERT : « Regarde Sprouty, tu vois la première boîte est vue comme `da0` et la seconde comme `da1`. »

Sprouty lui répond alors : « Oui j'ai vu, mais cela ne m'aide pas, car si je change l'ordre d'insertion, les `daX` changent aussi ! »

GEOM Vert fait tranquillement observer à Sprouty la dernière ligne de la première boîte : « Regarde bien la ligne avec `GEOM_LABEL`, elle comporte le `label bsdkey` ».

GEOM Vert retire alors les 2 boîtes de maïs du scanner et les insère dans un ordre différent.

```
kernel: umass0: <Intelligent Stick Intelligent Stick,
        class 0/0, rev 1.10/1.00, addr 2> on uhub0
kernel: da0 at umass-sim0 bus 0 target 0 lun 0
kernel: da0: <USB Card Intelligent Stic 2.02>
        Removable Direct Access SCSI-2 device
```

```
kernel: da0: 1.000MB/s transfers
kernel: da0: 255MB (522688 512 byte sectors: 64H 32S/T 255C)

kernel: umass1: <Intelligent Stick Intelligent Stick,
        class 0/0, rev 1.10/1.00, addr 3> on uhub0
kernel: da1 at umass-sim1 bus 1 target 0 lun 0
kernel: da1: <USB Card IntelligentStick 2.01>
        Removable Direct Access SCSI-2 device
kernel: da1: 1.000MB/s transfers
kernel: da1: 255MB (522688 512 byte sectors: 64H 32S/T 255C)
kernel: GEOM_LABEL: Label for provider da1a is label/bsdkey.
```

Sprouty s'étonne alors : « Oh ! L'ordre des daX a bien changé, mais on retrouve le GEOM_LABEL sur la bonne boîte ! »

Sprouty (se tournant vers GEOM Vert) : « GEOM Vert, est-il possible d'utiliser le **label bsdkey** pour m'y retrouver dans mes boîtes de maïs ? »

GEOM Vert explique à son ami que quel que soit l'ordre d'introduction de la boîte de maïs avec le GEOM_LABEL, la première fois en umass1/da1, la seconde fois en umass0/da0, le label label/bsdkey est reconnu par le scanner à code-barres dans /dev/label/.

GEOM Vert prend alors le scanner portable des mains de Sprouty et commence à taper sur l'interface. « Regarde, le label se retrouve ici, ce qui te permet d'effectuer toutes les opérations courantes (newfs, fsck, mount) sur la boîte /dev/label/bsdkey sans avoir à te préoccuper du nom réel da0 ou daX».

```
% ls -al /dev/label/
crw-r-----  1 root  operator    0, 131 bsdkey
% fsck -t ffs  /dev/label/bsdkey
% mount /dev/label/bsdkey /mnt && mount
** /dev/label/bsdkey
** Last Mounted on /mnt
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
** Phase 5 - Check Cyl groups
4 files, 67 used, 126400 free
        (24 frags, 15797 blocks, 0.0% fragmentation)
/dev/label/bsdkey on /mnt (ufs, local)
```

Sprouty reste sans voix face à cette démonstration pourtant simple et un sourire commence à se dessiner lentement sur son visage. « Tu veux dire que j'ai juste à trouver des noms distincts pour mes boîtes de maïs et je pourrais les utiliser par leur nom indépendamment de l'ordre d'insertion dans mon scanner ? »

« C'est tout à fait ça », lui répond GEOM Vert, « de plus, GEOM_LABEL écrivant ses données dans le dernier secteur du consumer, il est tout à fait possible de glabelliser des boîtes qui ne sont pas en UFS (en VFAT, au hasard, comme les *Memory Card* d'APN par exemple, ou en utilisant les labels ext2/3 des partitions Linux). Le *montage* GEOM ne se fera alors pas dans l'arborescence /dev/label/, mais dans /dev/msdosfs/ ou /dev/ext2fs/ ».

```
kernel: da2: <JetFlash TS1GJF150 8.07>
        Removable Direct Access SCSI-2 device
kernel: da2: 40.000MB/s transfers
kernel: da2: 976MB (1998848 512 byte sectors: 64H 32S/T 976C)
kernel: GEOM_LABEL: Label for provider da2s1 is msdosfs/USBKEY.
```

Sprouty, habituellement si volubile, est éberlué et réfléchit intensément à ce que vient de lui montrer son ami GEOM Vert.

4. LES DIFFÉRENTS MODÈLES DE SCANNER (VNCONFIG/MDCONFIG/GGATEL)

NOTE

Le traducteur a tenu à laisser dans la langue d'origine les expressions de cette magnifique histoire, toutefois pour la bonne compréhension de ce qui va suivre, le lecteur devra avoir à l'esprit que la boîte de cœurs de palmiers dont il est question est en fait un fichier monté sur une loopback et utilisé comme filesystem.

GEOM Vert profite d'avoir l'attention de Sprouty pour lui faire lire les pages de manuel citées au début et insiste sur quelques points liés aux différentes versions de son scanner à code-barres.

« Avant la version 5.X de ton scanner, tu utilisais vnconfig **[4]** pour monter des boîtes de cœurs de palmiers. Depuis la version 5.X, tu utilises mdconfig **[5]** pour le même usage ».

```
% dd if=/dev/zero of=disk0.img bs=1m count=512
% ls -alh disk*
-rw-r--r--  1 root  wheel   512M disk0.img
%  mdconfig -a -t vnode -f disk0.img
md1
% newfs /dev/md1
/dev/md1: 512.0MB (1048576 sectors)
        block size 16384, fragment size 2048
using 4 cylinder groups of 128.02MB, 8193 blks, 16448 inodes.
super-block backups (for fsck -b #) at:
 160, 262336, 524512, 786688
% mount /dev/md1 /mnt
```

Le fichier disk0.img est devenu une véritable boîte de cœurs de palmiers...

GEOM Vert défait le résultat des commandes qu'il vient d'exécuter sur le scanner de Sprouty et lui explique qu'il va utiliser la commande GEOM ggatel **[6]** pour le même résultat.

```
% ggatel create -u 0 disk0.img
% ls -al /dev/ggate*
crw-r-----  1 root  operator    0, 124 /dev/ggate0
% fsck -t ffs /dev/ggate0
** /dev/ggate0
** Last Mounted on /mnt
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
```

```
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
** Phase 5 - Check Cyl groups
3 files, 2 used, 253813 free
        (21 frags, 31724 blocks, 0.0% fragmentation)
```

5. LES BOÎTES DE MAÏS ET LES BOÎTES D'ASPERGES

GEOM Vert a envie d'aller plus loin dans la démonstration des possibilités du scanner à code-barres et il crée plusieurs fichiers sur le scanner à code-barres de son ami pour simuler des boîtes d'asperges et de maïs.

Les différentes boîtes d'asperges auront pour petit nom `diskX.img` avec `X` allant de 0 à 4 et les boîtes de maïs se prénommeront `usbX.img` avec `X` allant de 0 à 2.

```
-rw-r--r--  1 root  wheel   512M disk0.img
-rw-r--r--  1 root  wheel   512M disk1.img
-rw-r--r--  1 root  wheel   512M disk2.img
-rw-r--r--  1 root  wheel   512M disk3.img
-rw-r--r--  1 root  wheel   512M disk4.img
-rw-r--r--  1 root  wheel   256M usb0.img
-rw-r--r--  1 root  wheel   256M usb1.img
-rw-r--r--  1 root  wheel   256M usb2.img
```

Les disques et les clés USB seront *glabelisées* de `diskA` à `diskE` et de `usbA` à `usbC` ce qui donne le tableau suivant :

fichier image	ggateX	label
disk0.img	(/dev/ggate0)	diskA
disk1.img	(/dev/ggate1)	diskB
disk2.img	(/dev/ggate2)	diskC
disk3.img	(/dev/ggate3)	diskD
disk4.img	(/dev/ggate4)	diskE
usb0.img	(/dev/ggate10)	usbA
usb1.img	(/deb/ggate11)	usbB
usb2.img	(/dev/ggate12)	usbC

GEOM Vert explique que la liste des devices est entre parenthèses, car c'est justement une dépendance dont il veut s'affranchir et qu'il n'utilisera que les devices via leur nom de label. Une fois tous ces fichiers images créés, cela donne ceci :

```
% ls -al /dev/label/
crw-r-----  1 root  operator    0, 137 diskA
crw-r-----  1 root  operator    0, 138 diskB
crw-r-----  1 root  operator    0, 139 diskC
crw-r-----  1 root  operator    0, 140 diskD
crw-r-----  1 root  operator    0, 141 diskE
crw-r-----  1 root  operator    0, 133 usbA
crw-r-----  1 root  operator    0, 135 usbB
crw-r-----  1 root  operator    0, 136 usbC
```

5.1 FAIRE DES MIROIRS AVEC DES BOÎTES DE MAÏS (GMIRROR)

...ou comment faire du raid1 soft en 2 minutes. GEOM Vert a envie d'épater son ami Sprouty : il l'interpelle et commence à taper une série de commandes sur l'interface du scanner à code-barres.

```
% gmirror load
% gmirror label -v -b round-robin gm0 /dev/label/diskA
Metadata value stored on /dev/label/diskA.
Done.
% gmirror status
      Name    Status  Components
mirror/gm0  COMPLETE  ggate0

% ls -al /dev/mirror/
crw-r-----  1 root  operator    0, 155 gm0

% gmirror insert gm0 /dev/label/diskB

% gmirror status gm0
      Name    Status  Components
mirror/gm0  DEGRADED  ggate0
                      ggate1 (27%)
```

GEOM Vert explique : « Bien évidemment, pendant ce temps-là, rien ne nous empêche de monter `/dev/mirror/gm0` dans un répertoire de travail sur le scanner et d'effectuer des opérations dessus. De la même manière, rien ne nous empêche de créer un nom de label pour ce volume `gm0` ».

```
% glabel create raid1 /dev/mirror/gm0

% glabel list mirror/gm0
Geom name: mirror/gm0
Providers:
1. Name: label/raid1
   Mediasize: 536870400 (512M)
   Sectorsize: 512
   Mode: r0w0e0
   secoffset: 0
   offset: 0
   seclength: 1048575
   length: 536870400
   index: 0
Consumers:
1. Name: mirror/gm0
   Mediasize: 536870400 (512M)
   Sectorsize: 512
   Mode: r0w0e0
```

GEOM Vert informe Sprouty que les labels `diskA` et `diskB` auront disparu.

5.2 FAIRE DES MIROIRS AVEC DES BOÎTES DE MAÏS À DISTANCE (GGATED ET GGATEC)

Non content d'avoir montré à Sprouty comment faire des miroirs avec ses boîtes de maïs, GEOM Vert va maintenant lui montrer comment faire la même chose à distance (à travers le réseau, simulé ici par 127.0.0.1).

Sprouty reste très attentif, même s'il ne comprend pas encore toutes les commandes que tape son ami. Les deux parties du miroir vont être configurées sur des serveurs « client » et « *remote* ».

Dans un premier temps, GEOM Vert configure le *daemon* qui va recevoir l'autre moitié du miroir sur le serveur « remote ».

```
% cat /home/exports.file
127.0.0.1/32    RW      /home/diskNetwork.img

% ggated -a 127.0.0.1 /home/exports.file
```

Ensuite, sur le serveur « client », on s'attache à la boîte de maïs du serveur « remote » :

```
% ggatec create -o rw 127.0.0.1 /home/diskNetwork.img
ggate0

% ggatel list -v
       NAME: ggate0
       info: 127.0.0.1:3080 /home/diskNetwork.img
     access: read-write
    timeout: 0
queue_count: 0
 queue_size: 1024
 references: 2
  mediasize: 536870912 (512M)
 sectorsize: 512
       mode: r0w0e0
```

« Vu du serveur client », explique GEOM Vert, « c'est comme si le miroir de boîtes de maïs était parfaitement local. On peut donc associer nos boîtes de maïs en miroir comme tout à l'heure. »

```
% gmirror load
% gmirror label -v -b round-robin gm1 /dev/ggate1
Metadata value stored on /dev/ggate1.
Done.

% gmirror insert gm1 /dev/ggate0

% gmirror status
      Name    Status  Components
mirror/gm1  DEGRADED  ggate1
                      ggate0 (19%)

% gmirror status
      Name    Status  Components
mirror/gm1  COMPLETE  ggate1
                      ggate0

% newfs /dev/mirror/gm1
/dev/mirror/gm1: 512.0MB (1048572 sectors)
        block size 16384, fragment size 2048
        using 4 cylinder groups of 128.00MB,
        8192 blks, 16384 inodes.
super-block backups (for fsck -b #) at:
 160, 262304, 524448, 786592
```

Pour parfaire la démonstration, GEOM Vert place un fichier dans le volume raid1 (`gm1`), stoppe ce volume, démonte les images disques locale et réseau (`disk4.img` et 127.0.0.1:3080 `/home/diskNetwork.img`) et tue le daemon `ggated`.

Maintenant, GEOM Vert demande à Sprouty d'imaginer que le serveur « client » a rencontré un problème (des lapins sont venus manger ses boîtes de maïs). Il va utiliser la copie du serveur « remote » remontée dans un volume gmirror dégradé.

Sur le serveur « remote », GEOM Vert attache la boîte de maïs qui était celle distante.

```
% ggatel create diskNetwork.img
ggate0
```

Il regarde sur quel nom de miroir `gmX` elle était attachée :

```
% gmirror dump ggate0
Metadata on ggate0:
     magic: GEOM::MIRROR
   version: 4
      name: gm1
       mid: 3825435594
       did: 1080613717
       all: 2
     genid: 0
    syncid: 1
  priority: 0
     slice: 4096
   balance: round-robin
 mediasize: 536870400
sectorsize: 512
syncoffset: 0
    mflags: NONE
    dflags: NONE
hcprovider:
  provsize: 536870912
  MD5 hash: 8aacc31992e81130db00dbdb410f1bcb
```

Et il recrée un miroir partiel, le vérifie et le monte :

```
% gmirror insert gm1 ggate0
gmirror: Not all disks connected.

% gmirror list
Geom name: gm1
State: DEGRADED
Components: 2
Balance: round-robin
Slice: 4096
Flags: NONE
GenID: 0
SyncID: 2
ID: 3825435594
Providers:
1. Name: mirror/gm1
   Mediasize: 536870400 (512M)
   Sectorsize: 512
   Mode: r0w0e0
Consumers:
1. Name: ggate0
   Mediasize: 536870912 (512M)
   Sectorsize: 512
   Mode: r1w1e1
   State: ACTIVE
   Priority: 0
   Flags: NONE
   GenID: 0
   SyncID: 2
   ID: 1080613717

% fsck -t ffs /dev/mirror/gm1
** /dev/mirror/gm1
** Last Mounted on /mnt
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
** Phase 5 - Check Cyl groups
```

```
3 files, 2 used, 253844 free
        28 frags, 31727 blocks, 0.0% fragmentation)

% mount /dev/mirror/gm1 /mnt

%  ls -al /mnt
total 6
drwxr-xr-x   3 root  wheel     512 .
drwxr-xr-x  20 root  wheel     512 ..
drwxrwxr-x   2 root  operator  512 .snap
-rw-r--r--   1 root  wheel       0 gmirror1.txt
```

Sprouty est épaté, subjugué par la démonstration de GEOM Vert qui paraît tellement simple lorsqu'elle est orchestrée par ses soins.

6. PROTÉGEONS NOTRE MIROIR DE BOÎTES DE MAÏS

Continuant sur sa lancée, GEOM Vert a envie de montrer à Sprouty comment protéger ses boîtes de maïs des museaux un peu trop curieux. GEOM Vert lui demande alors un mot de passe secret : « *raide one* », lui répond-il.

```
% geli init -e aes -l 256 -s 4096 /dev/label/raid1
Enter new passphrase:<raid1>
Reenter new passphrase:<raid1>

% geli attach /dev/label/raid1
Enter passphrase:<raid1>

 ls -al /dev/label/
crw-r-----  1 root  operator    0, 138 raid1
crw-r-----  1 root  operator    0, 158 raid1.eli

% newfs /dev/label/raid1.eli
/dev/label/raid1.eli: 512.0MB (1048568 sectors)
        lock size 16384, fragment size 4096
        using 4 cylinder groups of 128.00MB,
        8192 blks, 8192 inodes.
super-block backups (for fsck -b #) at:
 160, 262304, 524448, 786592
```

Pour prouver à Sprouty qu'il retrouvera bien ses boîtes de maïs originales dans le container protégé, GEOM Vert crée une étiquette témoin.

```
 mount /dev/label/raid1.eli /mnt
touch /mnt/SuperSecretFile
```

7. METTRE LE TOUT À LA BANQUE (GSHSEC)

Comme Sprouty ne comprend pas très bien ce que veut lui expliquer GEOM Vert, ce dernier va tenter d'utiliser une métaphore issue d'un monde irréel qu'ils ont vu tous les deux lors de la projection d'un film de science-fiction.

« Rappelle-toi ce film : il y avait un coffre à la banque. L'acteur principal et son conseiller avaient chacun une clé les deux, une fois réunies, leur permettaient d'ouvrir le coffre. Sans la clé de son conseiller, il ne pouvait pas avoir accès à son coffre tout seul ; de même, son conseiller ne pouvait avoir accès au contenu du coffre de l'acteur sans sa clé. »

GEOM Vert : « Dans ce que je veux te montrer, c'est un peu la même chose, nous allons remplacer ces deux clés mécaniques par des ouvre-boîtes qui contiendront chacun une partie d'un fichier servant à accéder à un container *encrypté*. Je te demande un peu d'attention, car cela n'est pas forcément simple. »

```
% gshsec label -v secret /dev/label/usbA  /dev/label/usbB
Metadata value stored on /dev/label/usbA.
Metadata value stored on /dev/label/usbB.
Done.

kernel: GEOM_SHSEC: Device secret created (id=1623862870).
kernel: GEOM_SHSEC: Disk ggate10 attached to secret.
kernel: GEOM_SHSEC: Cannot add disk label/usbA to secret
                    (error=17).
kernel: GEOM_SHSEC: Disk ggate11 attached to secret.
kernel: GEOM_SHSEC: Device secret activated.
kernel: GEOM_SHSEC: Cannot add disk label/usbB to secret
                    (error=17).

% ls -al /dev/shsec/
crw-r-----  1 root  operator    0, 158 secret
```

GEOM Vert explique : « Chaque ouvre-boîte (usbA et usbB) comporte la moitié des informations de la clé accessible via /dev/shsec/secret. Par contre, les précédents labels usbA et usbB auront disparu après l'exécution de cette commande. Seul le label secret subsistera. Celui-ci sera automatiquement monté dès l'insertion des deux clés. Si on retire une des deux clés USB, le device secret est supprimé. »

```
% ggatel destroy -u 11
kernel: GEOM_SHSEC: Device secret removed.
kernel: GEOM_LABEL: Label label/usbB removed.
kernel: GEOM_GATE: Device ggate11 destroyed.
```

« On réinsère la seconde clé USB (peu importe sous quel device). Le device secret est de nouveau accessible. »

```
% ggatel create usb1.img
kernel: GEOM_SHSEC: Disk ggate5 attached to secret.
kernel: GEOM_SHSEC: Device secret activated.
```

GEOM Vert : « Regarde bien Sprouty. On change l'accès au volume encrypté créé précédemment en passant de l'utilisation d'une *passphrase* à l'utilisation d'une clé. »

```
% cat /dev/shsec/secret | \
  geli setkey -P -K -  /dev/label/raid1
```

On vérifie que l'accès avec la clé fonctionne (sans passphrase) :

```
% geli detach /dev/label/raid1
% cat /dev/shsec/secret | \
  geli attach -p -k - /dev/label/raid1

% ls -al /dev/label/
crw-r-----  1 root  operator    0, 138 raid1
crw-r-----  1 root  operator    0, 135 raid1.eli

% fsck -t ffs /dev/label/raid1.eli
% mount /dev/label/raid1.eli /mnt
** /dev/label/raid1.eli
** Last Mounted on /mnt
```

```
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
** Phase 5 - Check Cyl groups
3 files, 2 used, 128968 free (12 frags, 32239 blocks,
       0.0% fragmentation)

% ls -al /mnt/
drwxrwxr-x   2 root  operator  512 .snap
-rw-r--r--   1 root  wheel       0 SuperSecretFile
```

Sprouty vient d'en prendre plein la vue. Il ne s'attendait pas à posséder un scanner à code-barres si puissant.

Il va lui falloir procéder à plusieurs expérimentations avec des boîtes de maïs, des ouvre-boîtes et des boîtes d'asperges pour réellement comprendre ce qu'a fait son ami GEOM Vert sous ses yeux.

GEOM Vert, absorbé par l'écran du scanner avec lequel il s'amuse maintenant depuis plusieurs minutes, continue de taper sur l'interface, et propose à Sprouty de vérifier par lui-même ce qu'il vient de faire.

```
% gshsec list
Geom name: secret
State: UP
Status: Total=2, Online=2
ID: 1623862870
Providers:
1. Name: shsec/secret
   Mediasize: 268434944 (256M)
   Sectorsize: 512
   Mode: r0w0e0
Consumers:
1. Name: ggate5
   Mediasize: 268435456 (256M)
   Sectorsize: 512
   Mode: r0w0e0
   Number: 0
2. Name: ggate6
   Mediasize: 268435456 (256M)
   Sectorsize: 512
   Mode: r0w0e0
   Number: 1

% ggatel list -v
       NAME: ggate5
       info: usb0.img
     access: read-write
    timeout: 0
queue_count: 0
 queue_size: 1024
 references: 2
  mediasize: 268435456 (256M)
 sectorsize: 512
       mode: r0w0e0

       NAME: ggate6
       info: usb1.img
     access: read-write
    timeout: 0
queue_count: 0
 queue_size: 1024
 references: 2
  mediasize: 268435456 (256M)
 sectorsize: 512
       mode: r0w0e0
```

```
% gmirror list
Geom name: gm0
State: COMPLETE
Components: 2
Balance: round-robin
Slice: 4096
Flags: NONE
GenID: 0
SyncID: 1
ID: 983470882
Providers:
1. Name: mirror/gm0
   Mediasize: 536870400 (512M)
   Sectorsize: 512
   Mode: r0w0e0
Consumers:
1. Name: ggate0
   Mediasize: 536870912 (512M)
   Sectorsize: 512
   Mode: r1w1e1
   State: ACTIVE
   Priority: 0
   Flags: NONE
   GenID: 0
   SyncID: 1
   ID: 976669846
2. Name: ggate1
   Mediasize: 536870912 (512M)
   Sectorsize: 512
   Mode: r1w1e1
   State: ACTIVE
   Priority: 0
   Flags: NONE
   GenID: 0
   SyncID: 1
   ID: 3391284983
```

Pour aider Sprouty à assimiler ces dernières manipulations, GEOM Vert lui conseille de lire la page de manuel sur `gshsec` **[7]**.

8. C'EST L'HEURE DE PLEX-SCHOOL (GVINUM)

Pour sa démonstration finale, GEOM Vert décide de réutiliser les boîtes d'asperges pour démontrer la facilité d'utilisation du jeu *plex-school*. Ce jeu consiste à disposer plusieurs boîtes de maïs dans une « grappe » logique, de capacité plus grande que chaque boîte de maïs, avec une ou plusieurs boîtes de maïs en *spare*. Si une boîte de maïs vient à être défectueuse, c'est une des boîtes en spare qui prend le relais, assurant la continuité de la grappe logique.

```
% for i in 0 1 2 3 4
do
  ggatel create -u ${i} disk${i}.img
done
% ggatel list -v | grep -E 'NAME|info'
       NAME: ggate0
       info: disk0.img
       NAME: ggate1
       info: disk1.img
       NAME: ggate2
       info: disk2.img
       NAME: ggate3
       info: disk3.img
```

```
      NAME: ggate4
      info: disk4.img

% cat gvinum.txt
drive vol1_disk1 device /dev/ggate0
drive vol1_disk2 device /dev/ggate1
drive vol1_disk3 device /dev/ggate2
drive vol1_disk4 device /dev/ggate3
drive vol1_disk5 device /dev/ggate4 hotspare
volume raid5_vol1
plex org raid5 261k
sd drive vol1_disk1
sd drive vol1_disk2
sd drive vol1_disk3
sd drive vol1_disk4

% gvinum create gvinum.txt
5 drives:
D hotspare     State: up /dev/ggate4  A: 511/511 MB (100%)
D vol1_disk4   State: up /dev/ggate3  A: 0/511 MB (0%)
D vol1_disk3   State: up /dev/ggate2  A: 0/511 MB (0%)
D vol1_disk2   State: up /dev/ggate1  A: 0/511 MB (0%)
D vol1_disk1   State: up /dev/ggate0  A: 0/511 MB (0%)

1 volume:
V raid5_vol1   State: up   Plexes: 1   Size: 1535 MB

1 plex:
P raid5_vol1.p0   R5 State: ok   Subdisks: 4   Size: 1535 MB

4 subdisks:
S raid5_vol1.p0.s3  State: up  D: vol1_disk4   Size: 511 MB
S raid5_vol1.p0.s2  State: up  D: vol1_disk3   Size: 511 MB
S raid5_vol1.p0.s1  State: up  D: vol1_disk2   Size: 511 MB
S raid5_vol1.p0.s0  State: up  D: vol1_disk1   Size: 511 MB

% ls -l /dev/gvinum/
total 1
dr-xr-xr-x  2 root  wheel         512 plex
crw-r-----  1 root  operator    0, 128 raid5_vol1
dr-xr-xr-x  2 root  wheel         512 sd
```

GEOM Vert prend le temps d'expliquer à Sprouty que pour l'instant toutes les commandes de vinum n'ont pas encore été implémentées dans gvinum (voir la section BUGS **[8]**) et qu'il ne serait pas judicieux de l'utiliser pour garder des informations importantes dans ses boîtes de maïs.

9. IL EST L'HEURE DE LIRE LE JOURNAL (GJOURNAL)

Afin de terminer tranquillement tout ce qu'il a montré à Sprouty, GEOM Vert termine sur une fonctionnalité disponible dans la toute fraîche version 6.2-STABLE de son scanner à code-barres. Il s'agit de la possibilité de tenir un journal des déplacements des boîtes de maïs sur les différentes étagères du garde-manger de Sprouty.

Sprouty demande alors à GEOM Vert s'il ne pourrait pas utiliser ce journal pour remplacer les étiquettes perdues de ses boîtes de maïs. GEOM Vert lui répond : « Tu te rendras compte quand tu auras pratiqué un peu ton scanner de code-barres que cela ne sert à rien ». Sprouty reste alors des heures à essayer de refaire calmement tout ce que

```
% ggatel create disk3.img
ggate0

% gjournal label -f -s 1048576 ggate0

kernel: GEOM_JOURNAL: Journal 952452390: ggate0 contains data.
kernel: GEOM_JOURNAL: Journal 952452390: ggate0 contains journal.
kernel: GEOM_JOURNAL: Journal ggate0 clean.
kernel: GEOM_JOURNAL: BIO_FLUSH not supported by ggate0.

% ls -al /dev/ | grep gga
crw-r-----   1 root  operator    0, 118 ggate0
crw-r-----   1 root  operator    0, 119 ggate0.journal

% gjournal list
Geom name: gjournal 3994593347
ID: 3994593347
Providers:
1. Name: ggate0.journal
   Mediasize: 535821824 (511M)
   Sectorsize: 512
   Mode: r0w0e0
Consumers:
1. Name: ggate0
   Mediasize: 536870912 (512M)
   Sectorsize: 512
   Mode: r1w1e1
   Jend: 536870400
   Jstart: 535821824
   Role: Data,Journal
```

lui a montré GEOM Vert sur son scanner à code-barres dont il ignorait la plupart des possibilités.

GEOM Vert s'éloigne sans bruit. Il a envie de jouer avec d'autres fonctionnalités de son propre scanner à code-barres.

CONCLUSION (GCONCLU)

À travers cette simple « mise en jambes », nous avons voulu vous présenter les différentes possibilités de GEOM. Mais il ne s'agit bien que d'une présentation ! Il est possible d'en faire un peu plus et pour aller plus loin avec GEOM en particulier (et avec FreeBSD en général), il existe une quantité d'excellentes documentations : les pages de manuel et le handbook **[9]**.

N'hésitez surtout pas à en user et en abuser !

FreeBSD®

RÉFÉRENCES

[1, 2, 4, 5, 6, 7, 8] pages de manuel **http://www.freebsd.org/cgi/man.cgi**

[3] http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/geom.htm

[9] http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/

__nico pour GCU – nico {pouet} rottenbytes {pouet} info

UNE NOUVELLE FLEUR DANS VOTRE JARDIN (MAGIQUE)

En bon jardinier, vous aimez votre jardin, et ajouter de nouvelles boutures vous procure une joie immense. Sauf que parfois, un simple rempotage n'est pas possible. Certaines de vos machines n'ont pas nécessairement de port pour un clavier, un écran (comme les convis boîtes Soekris) ou bien vous avez la loutre attitude et vous ne voulez pas fouiller derrière votre machine pour utiliser celui actuellement en place.

Qu'à cela ne tienne, la manipulation suivante vous propose d'installer, depuis une machine Linux ou BSD, une serviette de bain ^W^W^Wtulipe orange de type netBSD.

On va donc utiliser ce petit appendice merveilleux de votre machine qu'est le port série, et un câble réseau (oui, c'est tout !).

1. A PROPOS DU PXE

Un petit mot pour expliquer ce qu'est PXE avant de commencer. PXE (*Preboot eXecution Environment*) est un système créé par Intel qui permet aux machines qui respectent cette norme, devenue un « standard », de booter sur le réseau. Dans notre cas : la séquence de boot va se faire selon cet ordre :

- Le client lance une requête DHCP.
- Le serveur lui donne une adresse et un nom de fichier accessible via TFTP.
- Le client le charge et *boote* dessus.

Pour des infos plus avancées sur PXE (*proxy* DHCP notamment), voir en **[1]**.

2. CONFIGURER LE DHCP

On installe le serveur DHCP (via votre système de ports/packages favoris) et on le configure comme suit :

```
subnet 192.168.1.0 netmask 255.255.255.0 {
 range 192.168.1.101 192.168.1.105;

 host fleurorange {
  # L'adresse MAC de la machine à installer. On peut
  # la spécifier si on veut être sûr qu'aucune
  # autre machine ne démarre sur l'installeur.
  hardware ethernet 00:00:AA:AA:BB:BB;

  # L'adresse IP à attribuer à fleurorange
  fixed-address 192.168.1.101;
  # Le fichier de boot PXE, qui envoie les
  # infos sur le port série.
  filename "pxeboot_ia32_com0.bin";

  # Le NFS
  next-server 192.168.1.195;
  option root-path "/home/install/";
 }
}
```

Avec toutes ces options, notre nouvelle fleur va pouvoir commencer à sentir bon, et s'ouvrir au monde rayonnant. On a donc précisé :

- l'adresse IP ;
- le fichier de boot, qui nous déposera délicatement les infos sur le port série ;
- les infos concernant le NFS (pour le / du système).

Le fichier `pxeboot_ia32_com0.bin` se trouve dans le répertoire `/usr/mdec` de votre crémerie^WnetBSD favori. Si vous n'en avez pas, vous pouvez l'obtenir en décompressant le fichier `i386/binary/sets/base.tgz` se trouvant sur un CD de netBSD.

3. DE L'ENGRAIS ET DU TERREAU

Récupérez l'image d'installation nommée `netbsd-INSTALL.gz` (sur une ISO netBSD montée en *loopback* par exemple) et décompressez-la dans `/home/install` :

```
cp netbsd-INSTALL.gz /home/install
cd /home/install
gunzip netbsd-INSTALL.gz
mv netbsd-INSTALL netbsd
```

Installez le serveur TFTP avec votre système ports/*package* favori, et faites pointer sa racine vers `/home/install`.

Il faut exporter `/home/install` en NFS pour la suite de l'install :

```
# /etc/exports: the access control list
#               for filesystems which may be exported
#               to NFS clients.  See exports(5).
/home/install 192.168.1.0/255.255.255.0(ro)
```

4. ARROSER DÉLICATEMENT AU PIED DE LA NOUVELLE POUSSE

A ce stade, la machine peut booter sur le réseau, va charger le loader PXE et proposer de faire l'install. Bien sûr, vous aurez lancé `(minicom|cu|screen)` pour récupérer la sortie sur votre port série :)

5. NOTE SUR L'UTILISATION DU PORT SÉRIE

Tout le monde n'est pas forcément un adepte du port série, même si celui-ci peut vous sauver la vie. Voici quelques infos pour les profanes. Le port série est généralement accessible via `/dev/ttyS0`, ou `/dev/ttyUSB0` si vous utilisez un adaptateur USB vers série. Pour brancher 2 machines ensemble, un câble série-série est nécessaire, disponible à pas cher dans tous les magasins d'informatique. Sous Linux, `minicom` est disponible pour y accéder, et `cu` sous FreeBSD. On utilise le plus souvent le mode « 9600 8N1 ». Commun aux 2 OS, le magique `screen` peut lui aussi y accéder, via `screen /dev/ttyUSB0` par exemple.

COMALINKS

[1] http://en.wikipedia.org/wiki/Preboot_Execution_Environment

[2] http://netbsd.org/Documentation/network/netboot/intro.i386.html

[3] http://netbsd.org/Documentation/network/netboot/local.install.html

Jean-Sébastien Pédron, aka Dumbbell <dumbbell@FreeBSD.org>

DÉVELOPPEMENT SUR LE NOYAU DE FREEBSD

DIS QUE « LE CODE EST BEAU » OU TA MACHINE REBOOTE.

Cet article a pour objectif de vous présenter le développement sur le noyau FreeBSD. De la mise en place de l'environnement de développement à l'examen d'un core, vous saurez tout tout tout sur le FriBi !

Nous allons commencer par la récupération des sources. Ça va nous permettre de nous familiariser ensuite avec leur organisation et de compiler un premier noyau qui fonctionne encore. Après ça, codaz ! Nous allons réaliser un module kernel inutile donc indispensable. Enfin, nous terminerons avec un paragraphe sur le debugging et tout ce qui concerne la finalisation d'un projet pour permettre son inclusion dans le CVS officiel.

Pour comprendre la suite, vous devez :

- savoir ce qu'est un *kernel* (ou « noyau » en bon français de France) ;
- savoir utiliser la commande `cvsup`(1) ;
- savoir utiliser la commande `cvs`(1) pour un accès en lecture à un `repository` ;
- savoir lire du code C ;
- savoir compiler un noyau FreeBSD ;
- éventuellement, savoir faire un `make world` ;
- savoir accéder au prompt du *loader* au *boot* ;
- savoir utiliser un minimum les outils de développement comme `make`(1), `gdb`(1) et `objdump`(1).

Les manipulations qui suivent doivent être faites sur une machine de test qui ne fournit aucun service indispensable, qui ne contient aucune donnée critique et dont vous avez l'accès au compte root. De plus, l'article se base sur la version 7-CURRENT encore en développement à l'heure de la rédaction de l'article. Des ISO *snapshots* sont disponibles sur le site de FreeBSD.

Au niveau espace disque, prévoyez 2,5 Go pour la mise en place de l'environnement et la compilation d'un noyau.

1. ENVIRONNEMENT DE DÉVELOPPEMENT

Avant de pouvoir coder tranquillement, il faut mettre en place un environnement de développement qui facilite notamment la mise à jour des sources de FreeBSD et la préparation de patchs.

1.1 CONFIGURER LES DUMPS KERNEL

En cas de *panic*, un *core dump* du kernel est très utile. Sous FreeBSD, ça fonctionne de la manière suivante :

1. Au panic, le core est écrit dans une partition swap préalablement indiquée au noyau ;
2. Au reboot suivant, le programme `savecore` lit le swap et écrit le core dans le répertoire `/var/crash`.

La ligne magique à ajouter à votre fichier `/etc/rc.conf` :

```
dumpdev="/dev/ad4s1b"
```

Bien entendu, remplacez `/dev/ad4s1b` par votre partition swap. Il faut aussi vous assurer que `/var/crash` a suffisamment d'espace libre (sachez qu'un core peut atteindre l'équivalent de votre RAM en espace disque).

1.2 MIROIR DU CVS

Pour éviter de désynchroniser `/usr/src` avec le système installé, nous allons travailler dans notre répertoire *home*, d'autant plus que les sources dans `/usr/src` ne sont pas accessibles en écriture par les utilisateurs normaux.

FreeBSD utilise un `repository` CVS pour la gestion des sources. Il existe donc un serveur accessible publiquement à partir duquel nous pouvons récupérer directement la tête du CVS. Par contre, il est interdit de faire des `cvs update` ou des `cvs diff` à partir de ce serveur pour ne pas le surcharger. Pour tout de même pouvoir mettre à jour son arbre local et faire des patchs facilement, il existe une solution qui apporte en bonus d'autres avantages.

La première étape consiste à faire un miroir local du *repository* CVS maître. Pour cela, la commande `cvsup`(1) qui s'utilise aussi pour mettre à jour les ports ou `/usr/src` est notre amie. Ceux, parmi vous, qui utilisent déjà la nouvelle commande `csup`(1) vont être déçus : elle ne supporte pas encore le mode qui nous intéresse.

CVSup utilise un fichier de configuration pour savoir ce qu'il doit récupérer et où le mettre. Pour notre utilisation, nous allons partir du *template* `cvs-supfile` présent dans `/usr/share/example/cvsup`. Mais avant de le remplir, voici la hiérarchie de répertoires utilisée dans l'article (bien entendu, modifiez-la pour qu'elle corresponde à votre habitude) :

```
/home/user
`-- monfreebsd
    |-- cvs
    |-- cvsup
    |   `-- db
    `-- 7-CURRENT
```

Par convention, pour désigner un fichier dans le répertoire `cvsup` ci-dessus, nous utiliserons le chemin `monfreebsd/cvsup/fichier`. Donc sachez que ce chemin sera relatif au répertoire *home* de l'utilisateur.

Revenons à CVSup et son fichier de configuration. Commençons par copier l'exemple dans `monfreebsd/cvsup`. Nous allons l'éditer pour indiquer nos préférences :

```
# Miroir CVS français
*default host=cvsup.fr.FreeBSD.org
# Fichiers d'état de CVSup
*default base=/home/user/monfreebsd/cvsup/db
# Racine du miroir CVS
*default prefix=/home/user/monfreebsd/cvs
```

Pour terminer cette étape, il suffit d'exécuter la commande suivante qui va mettre les fichiers CVS dans `monfreebsd/cvs` ; pour mettre à jour votre miroir, il suffira simplement de la relancer.

```
~/monfreebsd/cvsup$ cvsup cvs-supfile
```

Ce miroir CVS occupera environ 1,5 Go.

Profitez-en pour faire la vaisselle.

1.3 CHECKOUT DU CVS

Maintenant que vous avez une copie du CVS en local, il faut faire un *checkout* de la tête du CVS et peut-être de certaines autres branches. Mais avant de le faire, parlons de la politique de modification des différentes versions de FreeBSD.

La tête du CVS est la version de développement de FreeBSD : celle où toutes les nouveautés arrivent. On y fait référence avec le suffixe « -CURRENT ». Par exemple, en ce moment, c'est la 7-CURRENT.

En parallèle de celle-ci, les versions stables qui sont en production chez les utilisateurs doivent être maintenues pour apporter en priorité des corrections de bug et, quand ça ne peut pas attendre la prochaine *release* majeure, certaines fonctionnalités. Ces versions sont notées par le suffixe « -STABLE ». En ce moment, les branches 6-STABLE, 5-STABLE et 4-STABLE sont supportées, mais la branche 4-STABLE arrive en fin de vie.

Pour ne pas perdre de modifications en route, voici comment nous fonctionnons :

- Pour l'ajout d'une fonctionnalité, nous la *committons* uniquement dans -CURRENT, sauf cas exceptionnel.
- Pour la correction d'un bug, nous la committons d'abord dans -CURRENT (si le bug y existe encore) pour une période de probation. Une fois cette période écoulée et si la modification n'a pas créé de souci, celle-ci est committée dans 6-STABLE & *friends* (toujours si ça a un sens). Elle sera donc intégrée à la prochaine release faite sur chaque branche.

Maintenant, vous comprenez pourquoi nous allons travailler sur une -CURRENT. Donc, c'est parti, *checkout* de la tête :

```
~/monfreebsd$ cvs -d/home/user/monfreebsd/cvs \
co src -d 7-CURRENT
```

Le checkout s'étale sur environ 500 Mo.

Sur ce, café.

1.4 ORGANISATION DE L'ARBRE DES SOURCES

L'action se déroule maintenant dans `monfreebsd/7-CURRENT`.

Le checkout fraîchement réalisé contient les sources du système complet : le noyau, les bibliothèques et programmes de base. C'est exactement la même chose que nous trouvons dans `/usr/src`.

À la racine de ce checkout, se trouve donc le fichier `UPDATING` qui liste les changements qui peuvent impacter de manière significative un système en production. À côté, le `Makefile` qui sert pour les classiques `make kernel` et `make world`. Parmi les répertoires, nous avons les programmes et les bibliothèques de bases, les *headers*, mais surtout le répertoire `sys` qui contient le kernel. Y'a de la lumière, entrons !

Dans les noms de répertoires ci-dessous, je redonnerai à chaque fois le préfixe `sys` pour montrer que nous sommes dans le kernel ; cette habitude est aussi utilisée sur les *mailing-lists* par exemple quand on veut indiquer un fichier du noyau. De la même manière, le préfixe `src` désigne les sources du système, autres que le noyau.

Commençons par un des répertoires que nous consulterons très souvent : `sys/sys`. Ce dernier contient les headers exportés : ceux-là même qui sont installés dans `/usr/include/sys`. Une de leurs fonctions principales est la déclaration des appels système.

Le code de ces derniers est placé dans `sys/kern`. Par exemple, vous trouverez les fichiers `sys/kern/vfs_*` qui contiennent les appels système liés aux opérations sur les systèmes de fichiers (`mount`(2) pour n'en citer qu'un) et sur les fichiers eux-mêmes (`open`(2) & friends).

À côté, vous trouverez un répertoire nommé `sys/libkern`. Cela n'est pas évident au premier abord, mais, dans le

noyau, nous n'avons pas la bibliothèque standard du C (elle est dans l'*userland*). L'objectif des sources dans ce répertoire est de combler ce manque. On y retrouve donc des appels communs comme les fonctions de traitement de chaînes de caractères (`strlen`(3), `strcmp`(3), etc.).

Pour terminer avec la partie générique du kernel, notons le répertoire `sys/conf`. En gros, il contient des fichiers qui listent les sources à intégrer au noyau (c'est-à-dire, pas les modules) et la déclaration des options qu'on retrouve dans le fichier de configuration du noyau. Mais nous y reviendrons en détail lors de la compilation de nos futurs modules dans le noyau.

Au même niveau se situent différents composants du noyau comme :

- `sys/vm` : la VM s'occupe de gérer tout ce qui concerne la mémoire ;
- `sys/net*` : comme leurs noms l'indiquent, ces répertoires sont liés à la partie réseau du noyau ;
- `sys/dev` : tous les drivers matériels sont là (clavier, cartes réseau, son, USB, etc.) ;
- `sys/ufs`, `sys/fs`, `sys/isofs`, `sys/nfs`, `sys/coda`, `sys/gnu/fs` : tous les systèmes de fichiers supportés ; ceux dans `sys/gnu/fs` ont une license non-BSD (typiquement du code GPL).

À côté de tout ce petit monde, nous avons le code dépendant de l'architecture. Chaque plate-forme supportée par FreeBSD a son propre répertoire. Par exemple :

- `sys/i386` : regardez autour de vous, y'en a partout ;
- `sys/amd64` : Opteron & Athlon 64 ;
- `sys/sparc64` : machines Sun avec de l'UltraSPARC par exemple ;
- `sys/arm` : processeur basse consommation utilisé dans certains PDA.

Concluons cette rapide visite par le répertoire `sys/modules`. Il contient un sous-répertoire par module noyau disponible. Chacun de ces sous-répertoires contient à son tour un `Makefile` où l'on trouve surtout la liste des fichiers sources nécessaires à la compilation du module. Les fichiers sources sont, bien entendu, dans le répertoire du composant. Par exemple, le `Makefile` présent dans `sys/modules/fxp` référence les sources du driver de la carte réseau fxp(4) présents dans `sys/dev/fxp`. Nous en reparlerons quand il sera l'heure de compiler un premier module noyau.

1.5 COMPILATION DU NOYAU

En compilant et utilisant un premier noyau avant toute modification, nous pouvons, d'une part, nous assurer que la machine fonctionne avec la tête du CVS de FreeBSD. D'autre part, ce noyau nous servira de référence pour vérifier l'impact que nos modifications auront, en voyant les différences (ou l'absence de différence) de comportements entre ce noyau et le futur *customisé*.

La nouvelle méthode `make buildkernel`/`installkernel` est très pratique pour la compilation d'un noyau de production, mais comme nous allons certainement recompiler souvent l'engin, l'ancienne méthode sera plus souple. Notez que nous allons travailler avec un noyau GENERIC qui est un bien meilleur référentiel qu'un kernel custom, surtout si nous avons besoin de poser des questions à la communauté. *Fight* !

```
sys/i386/conf$ config GENERIC
sys/i386/conf$ cd ../compile/GENERIC
sys/i386/compile/GENERIC$ make cleandepend
sys/i386/compile/GENERIC$ make depend && make
(...)
sys/i386/compile/GENERIC$ make install KERNEL=kernel.ref
```

Si vous ne testez pas sur une machine i386, modifiez l'architecture dans les répertoires de l'exemple ci-dessus et ailleurs dans l'article si nécessaire.

Sur i386, un noyau utilise un peu moins de 400 Mo de disque pour compiler.

Sur la dernière ligne de commande, l'option `KERNEL=kernel.ref` nous permet d'installer le noyau sous un autre nom ; nom qui est `kernel` par défaut. Donc, les fichiers seront copiés dans `/boot/kernel.ref`. Pour booter ce noyau, il faut appuyer sur une touche au moment du loader (ou appuyer sur une touche s'il n'y a pas de menu), puis taper :

```
OK boot kernel.ref
```

Bien entendu, le clavier qwerty est familier pour vous.

À présent, nous avons :

1. Les sources en local pour travailler en paix.
2. Une connaissance superficielle de leur organisation.
3. Un noyau de référence installé.

Tout est donc en place pour s'offrir quelques panics.

2. CODAGE D'UN DEVICE DRIVER

C'est l'heure de coder ! Pour ne pas rompre avec la tradition et pour faire un parallèle avec l'article qui est paru dans *GNU/Linux Magazine* en novembre 2006 sur le développement dans Linux, commençons avec un driver *Hello World* ; d'abord dans le noyau, ensuite en module.

2.1 SOURCE DU MODULE

Comme nous l'avons vu dans la présentation de la hiérarchie des sources, les *device drivers* sont dans `sys/dev`. C'est là que nous nous installons en créant un sous-répertoire `hello`. À l'intérieur, éditons un fichier `hello.c`. La première chose à faire est de déclarer notre intention :

```
#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/conf.h>

DEV_MODULE(hello, hello_modevent, NULL);
```

DEV_MODULE(9) est une macro qui permet de déclarer un device driver :

- En 1er argument, nous indiquons le nom du module.
- En 2ème argument se trouve le nom de la fonction main (le point d'entrée) qui traite les chargements/déchargements du module.
- En dernier argument, nous pouvons mettre une donnée qui sera passée à cette fonction.

Notez que la ligne DEV_MODULE est très souvent placée en fin de fichier, même si ça n'apparaît pas dans l'exemple ci-dessus. Dans la logique, il faut au moins qu'elle apparaisse après la déclaration du prototype ou de la fonction elle-même.

Le plus important est donc ce fameux point d'entrée. Dans le cas d'une compilation dans le noyau, il est appelé au boot et au *shutdown*. Dans le cas d'un module externe, il est d'abord appelé quand le module est chargé. Il est à nouveau rappelé quand le module doit être déchargé ou quand le système est arrêté (ou rebooté).

Le prototype de ce *callback* est :

```
static int  hello_modevent(
        module_t mod, int type, void *data);
```

Les arguments sont :

1. Une structure module_t qui décrit l'instance du module.
2. Un entier qui nous indique si le module est chargé, déchargé ou si le système est en cours de reboot.
3. La donnée précisée en 3ème argument de DEV_MODULE(9).

Le module que nous réalisons doit afficher un message quand il est chargé, puis un autre quand il est déchargé. Le code est donc relativement court :

```
static int
hello_modevent(module_t mod, int type, void *data)
{
  int error;

  error = 0;

  switch(type) {
    case MOD_LOAD:
      /* Le module est fraichement chargé. */
      printf("Module chargé\n");
      break;
    case MOD_QUIESCE:
      /* L'utilisateur aimerait décharger le module. */
      printf("Module prêt pour déchargement\n");
      break;
    case MOD_UNLOAD:
      /* Le module va être déchargé. */
      printf("Module déchargé\n");
      break;
    case MOD_SHUTDOWN:
      /* Reboot du système. */
      printf("Reboot\n");
      break;
    default:
      error = EOPNOTSUPP;
      break;
  }

  return (error);
}
```

La fonction utilise les 4 valeurs possibles de type :

- MOD_LOAD : c'est simple, le module vient d'être chargé, il peut procéder à l'initialisation de ce dont il a besoin.
- MOD_QUIESCE : c'est la première étape, quand un utilisateur demande le déchargement d'un module. Le rôle du module est d'indiquer soit qu'il est prêt à être déchargé, soit qu'il est encore utilisé (et retourne un code d'erreur).
- MOD_UNLOAD : seconde étape du déchargement : le module tente cette fois de s'arrêter réellement. Il peut bien sûr retourner une erreur s'il ne peut pas.
- MOD_SHUTDOWN : le module est prévenu que le système va rebooter.

Notez que les phases MOD_QUIESCE/MOD_UNLOAD ne sont appelées qu'en cas de déchargement explicite du module. Donc lors d'un reboot, seule la phase MOD_SHUTDOWN est appelée.

La phase MOD_LOAD est appelée que le module soit compilé dans le noyau ou sous forme de module externe.

2.2 COMPILATION DANS LE NOYAU

Quand un module est compilé directement dans le noyau, nous pouvons en profiter dès le boot, même avec un boot réseau. Ça s'avère très pratique si, par exemple, le système est installé sur un disque branché sur un contrôleur exotique. Le second avantage est de rendre le *debugging* plus facile. Le noyau par défaut livré avec FreeBSD, GENERIC, est compilé pour avoir un maximum de drivers embarqués, plutôt que sous forme de modules externes. L'objectif est de supporter la plupart des plateformes sans avoir à préparer soi-même son CD d'installation.

Pour compiler notre beau module dans le noyau, il y a deux étapes :

1. Ajouter la liste de nos fichiers sources dans les fichiers présents dans sys/conf – répertoire dont nous n'avons pas encore parlé.
2. Faire une copie du fichier de configuration GENERIC pour y ajouter notre module.

Dans le répertoire sys/conf, vous trouverez deux types de fichiers que vous serez probablement amenés à éditer. Le premier type correspond aux fichiers sys/conf/files*. Il y a d'abord un fichier sans extension commun à toutes les plateformes, puis un fichier par plate-forme (dont le nom sert d'extension). Ces fichiers contiennent la liste des sources à inclure dans le noyau en fonction de ses options. Sur i386, la compilation prendra donc en compte les fichiers nécessaires dans sys/conf/files et sys/conf/files.i386. Cette séparation permet de déclarer les modules multiplateformes dans le premier et les modules spécifiques aux PC dans le second.

Notre module étant multiplateforme, nous allons ajouter la ligne suivante dans sys/conf/files :

```
dev/hello/hello.c          optional hello
```

Quelques explications sur le format de ce fichier :

- Une entrée de sys/conf/files* correspond à un fichier source à inclure et ses options d'inclusion. Une entrée se limite généralement à une ligne du fichier, mais peut s'étaler sur plusieurs lignes en échappant avec \, le retour à la ligne comme on le ferait avec un #define multi-ligne.
- Une entrée commence par le nom du fichier. Ce nom est relatif au répertoire sys du noyau.
- En 2ème position, le mot clé indique le type d'inclusion. Soit standard pour dire que le fichier source est tout le temps dans le noyau, soit optional pour indiquer qu'il ne doit être inclus que si une option est précisée.
- Dans le cas de optional, nous donnons ensuite l'option qui entraînera l'inclusion dudit source.
- Il existe plusieurs arguments possibles, mais les décrire tous sort du cadre de cet article.

Donc, dans l'exemple, on indique au processus de compilation que si le fichier de configuration contient device hello, il faut utiliser notre fichier source.

Le 2ème type correspond aux fichiers sys/conf/options*. Comme pour le 1er type, nous retrouvons le même genre de séparation générique/par plate-forme. Dans ces fichiers, nous pouvons indiquer les éventuelles options (ligne options OPTION dans la configuration du noyau). Pour le fichier source, chaque option est un #define. Notre module n'a pas d'option, donc nous n'y touchons pas.

Maintenant, la 2ème étape. Dans le répertoire sys/i386/conf (adaptez-le en fonction de votre plate-forme). Copions le fichier GENERIC pour y ajouter notre module et compilons l'ensemble :

```
sys/i386/conf$ cp GENERIC GENERIC_HELLO
sys/i386/conf$ $EDITOR GENERIC_HELLO # pour ajouter " device hello "
sys/i386/conf$ config GENERIC_HELLO
sys/i386/conf$ cd ../compile/GENERIC_HELLO
sys/i386/compile/GENERIC_HELLO$ make cleandepend && make depend && make
sys/i386/compile/GENERIC_HELLO$ make install KERNEL=kernel.hello
```

Pour tester ce noyau, rebootez la machine et, au loader, tapez :

```
OK boot kernel.hello
```

Très tôt dans le processus de boot, vous apercevrez le message « Module chargé ». Si vous le ratez, remontez dans l'historique de la console ou consultez le fichier /var/run/dmesg.boot.

En revanche, nous n'avons rien à décharger, donc nous ne pouvons pas tester les phases MOD_QUIESCE/MOD_UNLOAD. Par contre, MOD_SHUTDOWN va être appelée dès que nous allons rebooter la machine. Faites-le et guettez le message « Reboot » : il apparaît entre l'affichage de l'*uptime* et le message « Rebooting... ».

2.3 COMPILATION EN MODULE EXTERNE

C'est souvent plus élégant de pouvoir compiler du code sous forme de module, bien que ça ne soit pas toujours possible. L'inconvénient du module est la difficulté à le debugger, mais nous y reviendrons plus tard.

Cette fois, nul besoin de toucher aux fichiers dans sys/conf. Tout se passe dans le répertoire sys/modules. La première chose à faire est de créer un sous-répertoire hello, puis un Makefile à l'intérieur :

```
sys/modules$ mkdir hello
sys/modules$ cd hello
sys/modules/hello$ $EDITOR Makefile
```

Le contenu du Makefile est très court :

```
1  .PATH: ${.CURDIR}/../../dev/hello
2
3  KMOD= hello
4  SRCS= hello.c
5
6  CFLAGS+= -I${.CURDIR}/../..
7
8  .include <bsd.kmod.mk>
```

La ligne 1 indique où se trouvent les sources du module. Ceux-ci sont listés dans la variable SRCS. Ici nous avons un seul fichier compilé systématiquement, mais il est possible d'utiliser tous les mécanismes des Makefile pour construire cette variable de manière plus élaborée. La ligne 3 précise le nom du module (sans l'extension .ko). Enfin, la ligne 8 inclut le Makefile qui contient toutes les règles pour la compilation d'un module.

Avant de continuer, vous pouvez tester la compilation du module seul, sans avoir à refaire tout le noyau. C'est un autre avantage de cette méthode. Pour cela :

```
sys/modules/hello$ make
```

En période de développement, avoir les informations de debugging peut nous intéresser, donc à la place d'un simple `make` :

```
sys/modules/hello$ make DEBUG_FLAGS=-g
```

Si tout fonctionne bien, un `make install` copiera le module au bon endroit. Sachez que vous pouvez tester votre module dès maintenant, si vous utilisez un noyau compilé à partir des mêmes sources que ceux du présent module.

Maintenant que nous sommes sûrs que la compilation du module est correcte, nous allons lier sa compilation à celle du noyau. Pour cela, il faut simplement modifier le `Makefile` dans `sys/modules` pour ajouter notre nouveau répertoire à la liste : c'est la variable `SUBDIRS` qui vous intéresse dans ce `Makefile`, sinon, rien de très dépaysant. La compilation du noyau ne demande pas de modification du fichier de configuration.

Que ce soit avec un `make install` du module seul ou un `make installkernel`, vous retrouverez votre module `hello.ko` dans `/boot/kernel` par défaut.

Pour le tester, nous le chargeons avec la commande classique :

```
$ kldload hello
Module chargé
```

Parfait, notre message apparaît et la machine est encore debout. Testons le déchargement maintenant :

```
$ kldunload hello
Module prêt pour déchargement
Module déchargé
```

Excellent travail.

Si vous désirez tester l'appel à `MOD_SHUTDOWN` comme à l'étape précédente, laissez le module chargé (ou rechargez-le) et rebootez la machine. Le message « Reboot » apparaîtra au même endroit.

Avec ça, nous avons une base de code pour faire des choses intéressantes.

3. CODAGE DU DEVICE DRIVER « PERDU »

Le module `hello` est bien joli, mais nous n'avons pas touché à grand-chose dans ce noyau. Ne parlons même pas de l'interaction avec les utilisateurs... Il est temps de faire quelque chose d'inutile donc indispensable !

L'objectif est de programmer un module à qui on doit envoyer la phrase magique « Le code est beau ! » régulièrement, sans quoi il paniquera. Cette idée a été récemment reprise dans une célèbre série télévisée. Pour détailler un peu plus, il devra :

- créer une entrée dans `/dev` pour que les utilisateurs puissent lui envoyer la phrase magique ;
- s'attendre à recevoir des données en provenance de cette entrée ;
- vérifier régulièrement s'il est l'heure de contrôler la phrase magique ou de paniquer.

Reprenons l'architecture de `hello` en créant les répertoires `sys/dev/perdu` et `sys/modules/perdu` avec un code source et un `Makefile` équivalent (en renommant les `hello` en `perdu`).

3.1 GESTION DE L'ENTRÉE DANS /DEV

La première étape consiste à créer une entrée dans `/dev` pour pouvoir envoyer la phrase magique au module. Pour permettre cette communication, il faut aussi implémenter un callback qui sera appelé à chaque fois que l'utilisateur écrira des données sur cette entrée.

Pour décrire l'entrée, nous utilisons une structure de type `struct cdevsw` que nous appellerons `perdu_cdevsw` :

```
static d_write_t        perdu_write;

static struct cdevsw perdu_cdevsw = {
        .d_name = "perdu",
        .d_version = D_VERSION,
        .d_write = perdu_write
};
```

Elle est définie dans l'*include* `<sys/conf.h>`.

Le plus important dans cette structure est la déclaration des callbacks. Quelques exemples :

- `.d_open`, appelé quand quelqu'un *open*(2) l'entrée ;
- `.d_read`, appelé quand quelqu'un lit à partir de l'entrée ;
- `.d_write`, appelé quand quelqu'un écrit sur l'entrée ;
- `.d_ioctl`, appelé quand quelqu'un utilise *ioctl*(2) sur l'entrée.

Il en existe encore bien d'autres, mais celui qui nous intéresse est `.d_write` comme le montre l'extrait du source ci-dessus.

Avant d'expliquer ce callback, revenons à la création de l'entrée. Nous voulons le faire au moment où le module est chargé, donc dans la partie `MOD_LOAD` de la fonction `perdu_modevent` :

```
#define PERDU_MINOR     0
#define PERDU_FILENAME  "perdu"

/* For use with make_dev(9)/destroy_dev(9). */
static struct cdev *perdu_dev;

  static int
perdu_modevent(module_t mod, int type, void *data)
{
  ...
    switch (type) {
      case MOD_LOAD:
```

```
...
    perdu_dev = make_dev(&perdu_cdevsw, PERDU_MINOR,
        UID_ROOT, GID_WHEEL, 0666, PERDU_FILENAME);
```

Notre ami est la fonction make_dev(9). En argument, elle prend :

- la fameuse structure que nous avons définie juste avant ;
- le numéro « minor » (le major étant automatiquement attribué) ;
- l'utilisateur et le groupe (ici, « root:wheel ») ;
- les permissions (ici, accès en lecture/écriture à tout le monde) ;
- le nom de fichier (ici, « perdu »).

En retour, nous avons une structure de type struct dev qui représente l'instance de notre entrée. Nous nous en servons pour la destruction au moment du déchargement du module :

```
        case MOD_UNLOAD:
        case MOD_SHUTDOWN:
                ...
                destroy_dev(perdu_dev);
```

Maintenant que l'entrée est proprement créée et détruite, voyons ce qui se passe quand l'utilisateur écrit sur /dev/perdu. Tout se déroule donc dans le callback évoqué plus tôt. Celui-ci a le prototype suivant :

```
static int
perdu_write(struct cdev *dev, struct uio *uio, int ioflag) {
```

En argument, nous trouvons :

1. La même structure que celle retournée par make_dev(9).
2. Une struct uio, utilisée pour les transferts de données entre le *kernel space* et l'*user space*.
3. Des flags que l'utilisateur peut par exemple donner avec fcntl(2) comme O_NONBLOCK.

Dans notre fonction, nous allons nous concentrer exclusivement sur cette structure uio(9) ; les autres paramètres ne nous intéresseront pas.

L'algorithme est simple : la fonction récupère les données écrites par l'utilisateur puis, si le *timer* est arrivé en phase de vérification, elle compare les données à la phrase magique attendue. En fonction du résultat, le timer est remis à zéro ou continue.

Commençons donc par allouer un *buffer* et copier les données dedans :

```
 1  int error;
 2  size_t len, pos, data_available;
 3  char *buf;
 4
 5  len = 0;
 6  error = 0;
 7
 8  buf = malloc(PAGE_SIZE, M_TEMP, M_WAITOK | M_ZERO);
 9  pos = 0;
10
11  while (uio->uio_resid > 0) {
12          data_available = PAGE_SIZE - pos;
13          if (data_available > 0) {
14              len = MIN(uio->uio_resid, data_available);
15              error = uiomove(buf + pos, len, uio);
16              if (error)
17                  goto OUT;
18
19              pos += len;
20          } else
21              break;
22  }
```

Pour l'allocation mémoire, pas de suspens inutile : malloc(9). Les arguments varient par rapport au malloc(3) de la bibliothèque standard :

1. La taille à allouer, classique.
2. Le « type » (ici, M_TEMP) qui sert pour diverses vérifications et statistiques (affichéés par vmstat -m)
3. Des flags.

Nous n'allons pas nous étendre sur le type, ça sort du cadre de cet article. Pour en apprendre plus, consultez le manuel de malloc(9). Concernant les flags, ici, nous déclarons que :

- Le module est d'accord pour attendre que les ressources demandées se libèrent si la mémoire vient à manquer (M_WAITOK).
- La zone mémoire doit être mise à zéro (M_ZERO).

Notre buffer est alloué, la boucle while qui commence en ligne 11 va récupérer les données de l'userland jusqu'à ce qu'il n'y en ait plus ou que notre buffer soit plein.

La structure uio(9) et la fonction uiomove(9) permettent les transferts de données, même si celles-ci doivent traverser la frontière entre le kernel space et l'user space. Dans le cas présent, nous n'avons justement pas à nous soucier de la provenance des données.

uio->uio_resid indique le nombre d'octets restant à récupérer ; c'est pourquoi nous bouclons sur sa valeur. Au cœur de cette boucle, nous appelons la fonction uiomove(9) en ligne 15 pour effectuer le transfert de len octets de la source décrite par uio vers notre buffer buf ; le décalage de pos est là pour gérer la concaténation des données si les données sont copiées en plusieurs fois.

Le reste de cette fonction s'occupe de la comparaison de la phrase entrée avec celle attendue pour le *reset* du timer (seulement si le module est en phase d'alerte).

3.2 BOUCLE PRINCIPALE AVEC CALLOUT(9)

Maintenant que nous avons mis en place la récupération de la phrase magique, il faut faire tourner la boucle principale qui vérifie le temps restant. Pour s'approcher du comportement du terminal utilisé dans la série télévisée, voici comment se déroule le compte à rebours :

1. Le module démarre un décompte de `CYCLE_DURATION` secondes en se rappelant lui-même toutes les `CYCLE_UPDATE_PERIOD` secondes.
2. `CYCLE_ALARM1` secondes avant la fin, le module notifie `devd`(8) toutes les `CYCLE_ALARM_UPDATE_PERIOD` secondes du niveau d'alarme, en commençant par le niveau 1.
3. `CYCLE_ALARM2` secondes avant la fin, le module passe en niveau d'alerte 2.
4. `CYCLE_ALARM3` secondes avant la fin, le module passe en niveau d'alerte 3.
5. À la fin du compte à rebours, il génère un `panic`(9).

Pour implémenter ça, le kernel propose un système de *callout*(9) grâce auquel on demande à ce qu'une fonction soit appelée au bout de `N`*(*ticks*/`HZ`) secondes. En gros, `HZ` est la fréquence d'exécution du *scheduler*. Le terme « tick » est employé pour désigner un cycle. Par exemple, si `HZ` vaut 1000, il y aura un tick toutes les millisecondes.

Une instance de callout est représentée par la structure `struct callout`. Nous en définissons une qui est initialisée au `MOD_LOAD` :

```
/* For use with callout(9). */
static struct callout perdu_callout;

static int
perdu_modevent(module_t mod, int type, void *data)
{
        ...
        switch (type) {
        case MOD_LOAD:
                callout_init(&perdu_callout, 0);
```

Rien de spécial avec cette initialisation. Il suffit maintenant de démarrer le décompte. Pour ça, nous allons ajouter une fonction `cycle_start` appelée à deux endroits :

1. À la fin du `MOD_LOAD` (une fois que la structure `callout`(9) est initialisée et que l'entrée dans `/dev` est créée).
2. Dans `perdu_write`, si la phrase magique a été correctement entrée par l'utilisateur.

Et voici, en avant-première, le contenu de la fonction `cycle_start` pour vous !

```
/* Remaining time. */
 1 static int remaining;
 2
 3 int
 4 cycle_start()
 5 {
 6
 7         remaining = CYCLE_DURATION;
 8
 9         return (callout_reset(&perdu_callout,
10             CYCLE_UPDATE_PERIOD * hz,
11             cycle_update, NULL));
12 }
```

À la ligne 7, la variable `remaining` prend comme valeur la durée de départ du cycle. Cette variable sera décrémentée à chaque rappel des fonctions données à `callout`(9). Elle servira également à `perdu_write` pour savoir si le module est en phase d'alerte pour décider s'il faut ou non vérifier la phrase magique entrée.

À la ligne 9, la fonction `callout_reset`(9) nous permet de lancer un timer qui appelera `cycle_update` sans paramètre (`NULL`) au bout de `CYCLE_UPDATE_PERIOD` secondes. Le fait d'utiliser cette fonction annule automatiquement un éventuel timer encore actif qui appellerait `cycle_update` sans argument. Nous donnons le temps à `callout_reset`(9) en ticks, c'est pourquoi nous multiplions la valeur en seconde par `hz`.

La fonction `cycle_update` commence par décrémenter notre variable `remaining`, puis, en fonction de sa valeur, utilise `callout_reset`(9) pour se rappeler elle-même ou pour appeler `cycle_update_alarm` en phase d'alerte :

```
static void
cycle_update(void *args)
{

        remaining -= CYCLE_UPDATE_PERIOD;

        if (remaining <= 0) {
                cycle_system_failure();
                return;
        } else if (remaining <= CYCLE_ALARM1) {
                cycle_beep1();
                callout_reset(&perdu_callout,
                    CYCLE_ALARM_UPDATE_PERIOD * hz,
                    cycle_update_alarm, NULL);
        } else {
                callout_reset(&perdu_callout,
                    CYCLE_UPDATE_PERIOD * hz,
                    cycle_update, NULL);
        }
}
```

Vous remarquerez que la fonction appelle `cycle_beep1` juste avant de donner la main à `cycle_update_alarm`. Cette fonction émet un évènement correspondant au premier niveau d'alerte.

Elle est aussi susceptible d'appeler `cycle_system_failure` si le temps total est écoulé. Cette dernière s'occupe de signifier à l'utilisateur qu'il a échoué.

La fonction `cycle_update_alarm` fait exactement la même chose que `cycle_update`, mais avec un cycle de rappel

différent, typiquement plus court (CYCLE_ALARM_UPDATE_PERIOD secondes). En plus d'utiliser cycle_beep1, elle fait appel à cycle_beep2 et cycle_beep3 pour les deux autres niveaux d'alerte.

Les évènements envoyés par ces trois fonctions sont des notifications devd(9). L'objectif est que l'utilisateur puisse configurer le *daemon* avec devd.conf(5) pour exécuter la commande de son choix pour signaler l'alarme. Ça peut être jouer un son, envoyer un mail ou entrer automatiquement la phrase magique (mais là, c'est pas drôle).

Prenons cycle_beep3 comme cobaye :

```
#define DEVD_SYS        "PERDU"
#define DEVD_SUBSYS     "ALARM"

void
cycle_beep3()
{

        devctl_notify(DEVD_SYS, DEVD_SUBSYS, "3", NULL);
}
```

En argument, elle prend des chaînes de caractères désignant le système et le sous-système qui envoient la notification, le type de cette notification et d'éventuelles données supplémentaires.

Dans devd.conf(5), une simple entrée comme celle-ci sera exécutée à chaque passage dans cycle_beep3 :

```
notify 50 {
        match "system"          "PERDU";
        match "subsystem"       "ALARM";
        match "type"            "3";
        action "play alarm3.wav";
};
```

Avec cette série de fonctions, nous avons implémenté la boucle principale du module : celle qui gère le compte à rebours et déclenche les alarmes.

3.3 PANIC

Comme nous avons l'habitude de coder sans bug, le kernel ne va pas paniquer, ce qui nous empêche de jouer avec un core dump. Pour remédier à ça, si la phrase magique n'a pas été entrée dans les temps, le module va forcer un panic.

C'est la fonction cycle_system_failure qui s'en occupe. Encore une fois, du code extrêmement obscur :

```
void
cycle_system_failure()
{

        panic("SYSTEM FAILURE");
}
```

panic(9) prend les mêmes arguments que printf(3). Ici, nous n'avons pas besoin d'une chaîne de format compliquée, un simple message suffira. La fonction panic(9) ne retourne pas, donc c'est inutile de vouloir procéder à toute sorte de *cleanup* après.

3.4 TESTS EN PROD

Pour le compiler, procédez comme pour le module hello. N'oubliez pas de mettre à jour votre configuration devd(8) pour les alarmes. Prenons comme exemple cette entrée qui ne fait que loguer le niveau de l'alarme :

```
notify 50 {
        match "system"          "PERDU";
        match "subsystem"       "ALARM";
        action "logger PERDU: alarm (level: $type)";
};
```

Une fois votre nouveau noyau booté ou votre module externe chargé, le décompte commence. Guettez donc vos logs *syslog* pour repérer le déclenchement de l'alerte. Dès que ça arrive, choisissez votre camp :

- Vous pensez que cette cause est la vôtre :

```
$ echo "Le code est beau !" > /dev/perdu
```

- Au contraire, vous êtes persuadé que c'est du bluff :

```
panic: SYSTEM FAILURE
db>
```

4. DEBUGGING

Si pour vous le code est beau, passez directement à la prochaine section.

Vous êtes resté ? Vous faites moins le malin maintenant. Le debugging est un sujet qui mérite un article à lui seul. Je vous renvoie donc à la documentation FreeBSD [1] : elle informe sur l'utilisation de DDB, GDB (*remote* ou *post-mortem*) et le cas des modules externes.

Cependant, nous allons nous attarder un peu sur le debugging avec GDB d'un module externe, puisqu'il manque un exemple dans la documentation.

Vous avez donc atterri dans le debugger après le panic(9). Pour obtenir votre dump, vous appuyez sur [c] comme « continue ». Une fois que le dump est fait, votre machine reboote. Lors du redémarrage, le core est écrit dans /var/crash sous le nom vmcore.XXX (où XXX est un entier). Il suffit ensuite de démarrer kgdb(1) (et non GDB) de la manière suivante :

```
$ kgdb /boot/kernel/kernel /var/crash/vmcore.27
```

Bien entendu, si vous avez utilisé un autre kernel que celui par défaut, adaptez le chemin ci-dessus à votre cas.

Une fois lancé, GDB aura pris connaissance des symboles du noyau, mais pas des modules. C'est à vous de parcourir la liste des modules internes au noyau en commençant par linker_files.tqh_first[0] :

```
(kgdb) p linker_files.tqh_first[0]
(...)
```

```
$1 = {ops = 0xc33b7800, refs = 26, userrefs = 0, flags = 0,
link = {
  tqe_next = 0xc3482e00, tqe_prev = 0xc06b0b1c},
  filename = 0xc33b2770 "kernel", id = 1,
  address = 0xc0400000 "\177ELF\001\001\001\t", size =
3873960, ndeps = 0,
  deps = 0x0, common = {stqh_first = 0x0, stqh_last =
0xc3483030}, modules = {
    tqh_first = 0xc33b6e40, tqh_last = 0xc3486d48}, loaded =
{tqe_next = 0x0,
    tqe_prev = 0x0}}
```

Là, c'est l'image du noyau elle-même, puisqu'on reconnaît le nom de fichier `kernel`. La liste des modules commence à `link = { tqe_next = 0xc3482e00 }`.

```
(kgdb) p *(struct linker_file *)0xc3482e00
$2 = {ops = 0xc33b7800, refs = 1, userrefs = 1, flags = 1,
link = {
  tqe_next = 0xc3482d00, tqe_prev = 0xc3483010},
  filename = 0xc33b2750 "snd_ich.ko", id = 2,
  address = 0xc07b2000 "\177ELF\001\001\001\t", size =
24648, ndeps = 2,
  deps = 0xc33b2670, common = {stqh_first = 0x0, stqh_last =
0xc3482e30},
  modules = {tqh_first = 0xc33b5b80, tqh_last = 0xc33b5b88},
loaded = {
    tqe_next = 0xc3482b00, tqe_prev = 0xc3482d40}}
```

Ici, nous trouvons le module `filename = 0xc33b2750 "snd_ich.ko"`. Vous aurez compris, il faut parcourir les `link.tqe_next` jusqu'à ce que nous trouvions le module à debugger. Toujours avec l'exemple ci-dessus, il faut continuer avec `link = { tqe_next = 0xc3482d00 }`.

Quand vous avez mis la main sur le bon module, notez l'adresse de chargement notée dans le membre `address` de la `struct linker_file`. Pour `snd_ich.ko`, l'adresse est donc `address = 0xc07b2000`.

En plus de cette information, il nous faut l'adresse du début des instructions dans le binaire du module. Nous l'obtenons avec `objdump(1)` :

```
$ objdump --section-headers /boot/kernel/snd_ich.ko | grep text
  4 .text         00001c5b  00001cf8  00001cf8  00001cf8  2**2
```

C'est le quatrième nombre hexadécimal qui nous intéresse, 0x00001cf8.

Il ne reste plus qu'à additionner ces deux adresses (0xc07b2000 + 0x00001cf8 = 0xc07b3cf8) et la fournir à GDB en argument de la commande pour charger le module :

```
(kgdb) add-symbol-file /boot/kernel/snd_ich.ko 0xc07b3cf8
add symbol table from file "/boot/kernel/snd_ich.ko" at
        .text_addr = 0xc07b3cf8
(y or n) y
Reading symbols from /boot/kernel/snd_ich.ko...done.
```

Maintenant, vous pouvez debugger confortablement votre module.

5. FINALISATION

C'est fait, nous avons ajouté une fonctionnalité importante au kernel FreeBSD. Maintenant, il convient d'en faire profiter les autres. Le meilleur moyen est d'envoyer un mail avec le patch sur la mailing-list **freebsd-current@FreeBSD.org**.

Mais pour que le patch ait plus de chance d'être inclus dans le CVS, il faut régler quelques points avant : le style du code et la documentation.

5.1 STYLE(9)

Le style utilisé dans le code de FreeBSD est décrit dans la page *man* style(9). Je ne vais pas m'amuser à paraphraser ce manuel ici : il faut le lire, même si c'est un peu pénible.

Les exemples montrés ici, ainsi que le code fourni, respectent ce style.

5.2 DOCUMENTATION

Le minimum est une page man dans la section 4 pour notre module. Cette page est fournie avec le code source. Elle doit être ajoutée dans le répertoire `src/share/man/man4`, puis déclarée dans le `Makefile` présent dans ce même répertoire.

6. TROUVER L'INSPIRATION

Pour conclure, vous trouverez le code des deux modules, les `Makefile` et les pages man sur **people.FreeBSD.org** [2].

À présent, le développement noyau vous intéresse ? Si vous ne trouvez pas d'idée pour débuter, inspirez-vous de celles proposées sur la page Projects [3] du site officiel ou tapez dans les rapports de bug [4] (appelés « PR » pour *Problem Report* dans le jargon FreeBSD). En parlant d'idée, merci à **Julien Barbot, aka Klyr (<klyr@quicheaters.org>)** pour celle du module `perdu` !

RÉFÉRENCES

[1] The FreeBSD Projects, « *Kernel Debugging* », in *FreeBSD Developers' Handbook*, **http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/kerneldebug.html**

[2] Jean-Sébastien PÉDRON, fichiers sources des exemples, **http://people.freebsd.org/~dumbbell/linuxmag_hs/**

[3] *The FreeBSD Projects, FreeBSD list of projects and ideas for volunteers*, **http://www.freebsd.org/projects/ideas/**

[4] *The FreeBSD Projects, Problem Report Database*, **http://www.freebsd.org/support/bugreports.html**

Eric Auge <eau@gcu-squad.org>

INTRODUCTION À LA PROGRAMMATION WIFI EN C SOUS NETBSD

OU COMMENT GUÉRIR D'UNE CONJONCTIVITE AVEC UNE GOUSSE D'AIL !

Tu aimes Dave, tu aimes aussi Sheila, tu es fan de Dalida, tu as toujours rêvé de savoir danser sur un damier illuminé et tu répètes « Parole » depuis des années devant le miroir de ton armoire.

Si c'est bien ce que tu cherches, cet article n'est pas pour toi, mais je suis enchanté de faire ta connaissance. Cependant, si tu as un peu de temps (tu répèteras plus tard devant ton miroir), je vais te faire rencontrer des lutins, des lutins magiques, des miettes de réseau qui volent dans l'air que tu respires chaque jour.

Je vais te parler de code et de *wireless* sous NetBSD.

Il s'agit d'une introduction, je répète une introduction, à la programmation 802.11 en utilisant l'API unifiée (`/usr/include/net80211/*`) ieee80211 de NetBSD. C'est probablement ce dont j'aurais eu besoin lors de mes premières idées de programmes *wifi* ou de portages d'applications sans fil Linux vers NetBSD (comme toi quand tu as répété « L'été indien » devant ton miroir, imagine si tu avais pu avoir Joe Dassin à tes côtés).

Nous allons tenter d'être pragmatique dans notre approche (oui, pour apprendre à danser, il faut danser, mais ça tu le sais déjà) : montrer concrètement comment effectuer les opérations de base et indiquer où se renseigner sur d'autres opérations que tu aimerais réaliser.

Cet article n'expliquera pas le fonctionnement du protocole 802.11 et ses variantes ; le lecteur possède déjà les concepts de programmation de base en C, une compréhension générale des réseaux wireless, ainsi qu'une bonne connaissance de la variété française (Pouuurr le plAaAAAaiiiSliiiiilrrrr). De plus, le code ne se veut pas exhaustif, ni « *secure* », ni audité, ni même utilisable dans le cadre d'une application destinée à tourner en production. C'est une simple illustration de la manipulation de cette API.

1. WIRELESS : LE STANDARD

Je ne vais pas rentrer dans les détails, Wikipédia en parlera mieux que moi :

- **http://fr.wikipedia.org/wiki/Wi-Fi**
- **http://fr.wikipedia.org/wiki/Wireless_LAN**
- **http://en.wikipedia.org/wiki/Wireless_LAN**
- **http://en.wikipedia.org/wiki/IEEE_802.11**

En gros, c'est génial, plus de fils, plus de trucs qui traînent (pas comme chez moi), etc.

2. MODES D'OPÉRATION COURANTS

Voici la liste des modes dans lesquels une carte wifi peut se trouver :

- ibss (*independant bss or infrastructure bss*), connu aussi sous d'autres termes : *managed*, client, *adhoc*, etc. ;
- hostap : la carte se comporte comme un point d'accès (*access point*) au réseau wireless ;
- monitor : la carte récupère passivement tous les paquets wireless qu'elle observe ou qui lui passent sous le nez.

C'est vite résumé et simplifié, on est d'accord.

3. NETBSD ET LE 802.11

Le système d'exploitation NetBSD possède une couche d'abstraction permettant le contrôle de différentes cartes wireless, indépendamment du fabricant. La majorité de cette abstraction est présente dans le noyau et chaque *driver* wireless propose une partie ou le support complet des différents appels de cette API.

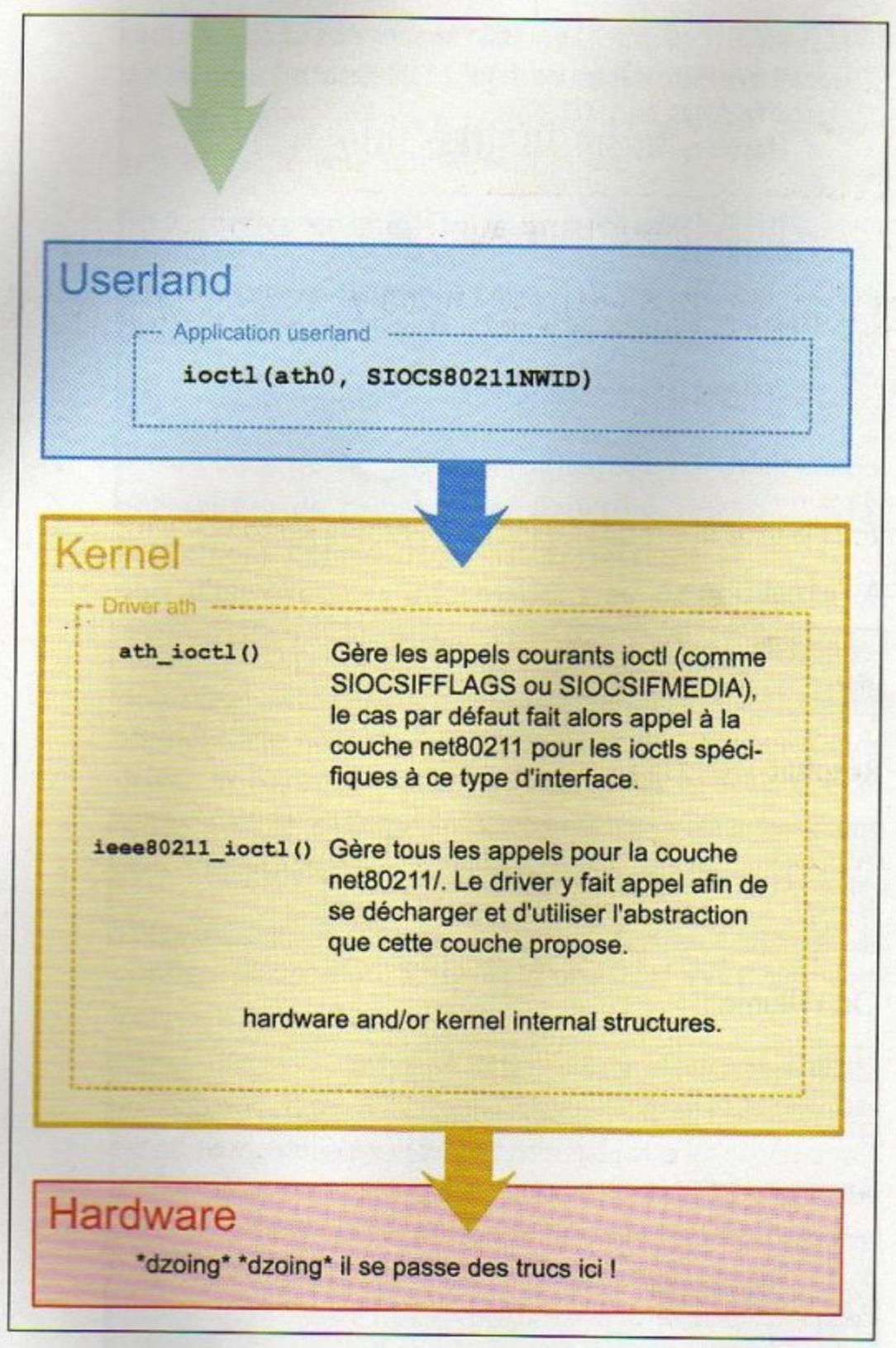
Le schéma 1 très simplifié montre son intégration au sein du système.

La majorité des drivers récents en font un usage quasi complet, mais certains drivers ne l'utilisent pas complètement. Ils fournissent leurs propres méthodes pour certaines opérations comme configurer la clef WEP (par exemple dans le driver `an`(4)).

Cette API a été écrite par Atsushi Onoe, puis reprise et maintenue par Sam Leffler. FreeBSD et NetBSD travaillent de concert (avec un *lead* très FreeBSDien) pour la maintenir et la faire évoluer. La personne responsable pour NetBSD est David Young.

Résumons donc ! En gros, ça facilite le code, ça rend les choses plus lisibles et plus simples à maintenir et évidemment « *small is beautiful* », tout ça, tout ça, tout ça... De plus, c'est plus simple à gérer que 36 appels pour chaque driver qui devrait alors implémenter ses propres *ioctls*, etc.

Pour en faire usage ? Simple, il suffit d'utiliser les *includes* appropriés qui se situent dans le répertoire `/usr/include/net80211/`. Ils définissent aussi bien les différentes options pour demander et configurer les paramètres du driver, que les structures et des macros permettant de créer



un *parseur* de paquets 802.11 assez facilement. Étant utilisés par le noyau lui-même pour gérer les paquets arrivant sur la carte, ils sont parfaitement adaptés si on désire gérer le *parsing* de données à partir de paquets bruts arrivant directement du noyau vers l'espace utilisateur (en *monitor mode* avec `pcap` par exemple).

Dans les exemples qui suivent, nous aurons uniquement besoin des includes suivants :

- `<net80211/ieee80211.h>` contient les structures des différentes trames wireless, des valeurs et quelques macros ;
- `<net80211/ieee80211_ioctl.h>` contient les ioctls courants pour l'API ;
- `<net80211/ieee80211_radiotap.h>` contient la structure du *header* RadioTap et les différents champs, mais nous en reparlons dans l'exemple n°5.

Afin d'accéder aux informations des interfaces ou de les configurer, il nous faudra également utiliser les includes relatifs à la gestion des interfaces réseau. Dans les exemples qui suivent, nous utiliserons uniquement les includes suivants :

- `<net/if.h>`
- `<net/if_media.h>`

Si d'autres informations sont nécessaires, il est très facile d'aller jeter un coup d'œil au code source de l'outil `ifconfig` et de regarder comment celui-ci s'y prend pour les obtenir ou les modifier.

Passons à la pratique.

Nous allons passer en revue quelques opérations simples et introduire la manière d'effectuer ces opérations sous NetBSD. Cet article a été écrit avec la version 4.0_BETA2, une carte Intel 3915ABG et une carte Atheros, ainsi qu'une carte Aironet.

4. EXEMPLE N°1 : ACTIVER/DÉSACTIVER UNE INTERFACE WIRELESS

Cette opération n'a rien de spécifique au wifi. Elle est commune à toute interface. Il suffit de remplir les membres `ifr_name` et `ifr_flags` de la `struct ifreq` (`<net/if.h>`). Le *flag* d'activation d'une interface est `IFF_UP`.

```
/*
 * Interface request structure used for socket
 * ioctl's.  All interface ioctl's must have parameter
 * definitions which begin with ifr_name.  The
 * remainder may be interface specific.
 */
struct  ifreq {
  char    ifr_name[IFNAMSIZ]; /* if name, e.g. "en0" */
  union {
    struct  sockaddr ifru_addr;
    struct  sockaddr ifru_dstaddr;
    struct  sockaddr ifru_broadaddr;
    short   ifru_flags;
    int     ifru_metric;
    int     ifru_mtu;
    int     ifru_dlt;
    u_int   ifru_value;
    caddr_t ifru_data;
    struct {
      uint32_t         b_buflen;
      void            *b_buf;
    } ifru_b;
  } ifr_ifru;
/* address */
#define ifr_addr        ifr_ifru.ifru_addr
/* other end of p-to-p link */
#define ifr_dstaddr     ifr_ifru.ifru_dstaddr
/* broadcast address */
#define ifr_broadaddr   ifr_ifru.ifru_broadaddr
/* flags */
#define ifr_flags       ifr_ifru.ifru_flags
/* metric */
#define ifr_metric      ifr_ifru.ifru_metric
/* mtu */
#define ifr_mtu         ifr_ifru.ifru_mtu
/* data link type (DLT_*) */
#define ifr_dlt         ifr_ifru.ifru_dlt
/* generic value */
#define ifr_value       ifr_ifru.ifru_value
```

```
/* media options (overload) */
#define ifr_media       ifr_ifru.ifru_metric
/* for use by interface XXX deprecated */
#define ifr_data        ifr_ifru.ifru_data
/* new interface ioctls */
#define ifr_buf         ifr_ifru.ifru_b.b_buf
#define ifr_buflen      ifr_ifru.ifru_b.b_buflen
};
```

Et voici le code d'exemple `wifi_switch.c` :

```
/*
 *  code d'exemple sous license BSD
 *  Eric Auge <eau@gcu-squad.org>
 *
 *  pour compiler :
 *  gcc wifi_switch.c -o wifi_switch
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <sys/ioctl.h>

#include <net/if.h>
#include <net/if_media.h>

#include <net80211/ieee80211.h>
#include <net80211/ieee80211_ioctl.h>

#define IFACE_NAME "wpi0"

int main(int argc, char ** argv)
{
  struct ifreq ifr;
  char * interface = IFACE_NAME;
  int s;

  if (!argv[1])
  {
    printf("no argument\n");
    printf("\n%s <on|off>\n", argv[0]);
    exit(1);
  }

s = socket(AF_INET, SOCK_DGRAM, 0);
if  (s <0)
{
  perror("socket()");
  return -1;
}

strlcpy(ifr.ifr_name, interface, sizeof(ifr.ifr_name));

if (strcmp(argv[1], "on") == 0)
{
  printf("activation de %s\n", interface);
  ifr.ifr_flags |= IFF_UP;
  if (ioctl(s, SIOCSIFFLAGS, (caddr_t)&ifr) == -1)
  {
    perror("ioctl()");
    return -1;
  }
} else if (strcmp(argv[1], "off") == 0)
{
  printf("desactivation de %s\n", interface);
  ifr.ifr_flags &= (~IFF_UP);
  if (ioctl(s, SIOCSIFFLAGS, (caddr_t)&ifr) == -1)
  {
    perror("ioctl()");
    return -1;
  }
} else
{
  printf("option inconnue\n");
}
return 0;
}
```

A l'exécution :

```
eau@kamehouse:~/lm $ sudo ./wifi_switch off
desactivation de wpi0
```

Résultat `ifconfig` :

```
[...]
wpi0: flags=8802<BROADCAST,SIMPLEX,MULTICAST> mtu 1500
[...]
```

On rallume :

```
eau@kamehouse:~/lm $ sudo ./wifi_switch on
activation de wpi0
```

Résultat `ifconfig` :

```
[...]
wpi0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu
1500
[...]
```

Il n'est pas nécessaire de préciser le média, sachant que, par défaut, le comportement est l'auto-sélection du média et du mode d'opération (OFDM24, OFDM36, 11b, 11g, etc.). Le mode d'opération par défaut est en infrastructure (IBSS).

L'ioctl(2) pour obtenir l'état de l'interface est `SIOCGIFFLAGS`. Il suffit alors de vérifier si le flag `IFF_UP` est actif dans le membre `ifr_flags` de la structure `ifreq`.

Requêtes ioctls :

- `SIOCGIFFLAGS` : récupérer les flags pour l'interface ;
- `SIOCSIFFLAGS` : configurer les flags pour l'interface

Maintenant, on est capable d'allumer et d'éteindre une interface : tu dois déjà te sentir un peu plus proche de tes chanteurs préférés.

5. EXEMPLE N°2 : L'ASSOCIATION

C'est très simple, le driver se charge de toute la négociation et des échanges. Côté *userland*, il suffit de proposer à la

carte un SSID (*Service Set ID*) via l'ioctl(2) `SIOCS80211NWID` de la couche net80211 et une clef WEP via l'ioctl(2) `SIOCS80211NWKEY`. Nous n'aborderons pas WPA dans cet article, qui se veut une introduction.

Pour configurer le SSID, il vous suffit de :

1. Remplir la structure `ieee80211_nwid` définie dans le fichier `<net80211/ieee80211_ioctl.h>` ;
2. L'envoyer au driver via l'ioctl `SIOCS80211NWID`.

Il faut faire de même pour la(les) clef(s) WEP, ce qui nécessitera de remplir une structure `ieee80211_nwkey` (également définie dans `<net80211/ieee80211_ioctl.h>`), puis d'envoyer ces informations au driver via l'ioctl `SIOCS80211NWKEY`.

Bon nombre de ces structures attendent qu'un champ contienne le nom de l'interface pour déterminer à quel driver envoyer les informations et les ordres. C'était le cas avec le champ `ifr_name` de la structure `ifreq` vue précédemment.

Voici un petit exemple `wifi_assoc.c`, afin d'illustrer la configuration d'un SSID et d'une clef WEP sur une interface nommée `wpi0` :

```
/*
 *  code d'exemple sous license BSD
 *  Eric Auge <eau@gcu-squad.org>
 *
 *  pour compiler :
 *  gcc wifi_assoc.c -o wifi_assoc
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#include <sys/ioctl.h>

#include <net/if.h>
#include <net/if_media.h>

#include <net80211/ieee80211.h>
#include <net80211/ieee80211_ioctl.h>

#define IFACE_NAME "wpi0"

int main(int argc, char ** argv)
{
  struct ifreq ifr;
  /* structure definissant le SSID */
  struct ieee80211_nwid nwid;
  /* structure definissant les clefs WEP */
  struct ieee80211_nwkey nwkey;
  /* interface recevant la configuration */
  char * interface = IFACE_NAME;
  /* structure contenant une clef WEP */
  unsigned char keybuf[16];
  int s;

  if (!argv[1])
  {
    printf("no argument\n");
    printf("\n%s <ssid> [<wepkey>]\n", argv[0]);
    exit(1);
  }

  s = socket(AF_INET, SOCK_DGRAM, 0);
  if  (s <0)
  {
    perror("socket()");
    return -1;
  }

  /* Etre propre sur soi ! */
  memset(&nwid, 0, sizeof(struct ieee80211_nwid));
  memset(&nwkey, 0, sizeof(struct ieee80211_nwkey));

  strlcpy(ifr.ifr_name, interface, sizeof(ifr.ifr_name));
  /* Un ptit sanity check a la noix. */
  if (strlen(argv[1]) > IEEE80211_NWID_LEN) {
    printf("ssid too long!\n");
    exit(1);
  }

  /* Copier le ssid dans la structure associee. */
  strlcpy(nwid.i_nwid, argv[1], sizeof(nwid.i_nwid));
  nwid.i_len = strlen(argv[1]);

  /* ifrequest */
  ifr.ifr_data = (void *)&nwid;

  if (ioctl(s, SIOCS80211NWID, (caddr_t) &ifr) == -1)
  {
    perror("ioctl(SIOCS80211NWID)");
    return -1;
  }

  printf("ssid set to '%s'\n", argv[1]);

  if (argv[2])
  {
    if (strlen(argv[2]) > sizeof(keybuf))
    {
      printf("key too long\n");
      exit(1);
    }
    /* Copier le nom de l'interface a configurer. */
    strlcpy(nwkey.i_name, interface, sizeof(nwkey.i_name));
    memcpy(keybuf, argv[2], sizeof(keybuf));

    nwkey.i_wepon = IEEE80211_NWKEY_WEP;
    nwkey.i_defkid = 1;
    nwkey.i_key[0].i_keylen = sizeof(keybuf);
    nwkey.i_key[0].i_keydat = keybuf;

    /* Pas d'autres clefs, juste une seule */
    nwkey.i_key[1].i_keylen = 0;
    nwkey.i_key[2].i_keylen = 0;
    nwkey.i_key[3].i_keylen = 0;
    if (ioctl(s, SIOCS80211NWKEY, &nwkey) == -1)
    {
      perror("ioctl(SIOCS80211NWKEY)");
      return -1;
    }
    printf("wep key #0 is set to '%s'\n", argv[2]);
  }

  return 0;
}
```

A l'exécution, nous obtenons les résultats suivants :

Sans clef WEP :

```
eau@kamehouse:~/lm $ sudo ./wifi_assoc testor
ssid set to 'testor'
```

Vérifions avec `ifconfig` :

```
eau@kamehouse:~/lm $ ifconfig wpi0
wpi0: flags=e843<UP,BROADCAST,RUNNING,SIMPLEX,LINK1,LINK2,MULTICAST>
        mtu 1500
        ssid testor
        powersave off
        address: 00:01:02:03:04:05
        media: IEEE802.11 autoselect (DS1 mode 11g)[...]
```

Avec clef WEP :

```
eau@kamehouse:~/lm $ sudo ./wifi_assoc testor proutproutproutp
ssid set to 'testor'
wep key #0 is set to 'proutproutproutp'
```

Vérifions avec `ifconfig` :

```
eau@kamehouse:~/lm $ sudo ifconfig wpi0
wpi0: flags=8802<BROADCAST,SIMPLEX,MULTICAST>
        mtu 1500
        ssid testor nwkey proutproutproutp
        powersave off
        address: 00:01:02:03:04:07
        media: IEEE802.11 autoselect (autoselect mode 11g)
[...]
```

Dans notre cas, le driver wpi(4) va maintenant se charger de faire la sale besogne, c'est-à-dire l'association avec le réseau sans fil. La partie userland se résume donc à ça. Simple, non ? :)

Vous pouvez bien entendu récupérer ces informations via les requêtes `G` (Get) ; les requêtes `S` (Set) étant présentes pour configurer (« *setter* ») un des paramètres (cf. ioctl(2)).

Requêtes ioctl :

- `SIOCG80211NWID` : récupérer la valeur du SSID courant
- `SIOCS80211NWID` : configurer la valeur du SSID courant
- `SIOCG80211NWKEY` : récupérer la valeur de(s) la clef(s) WEP
- `SIOCS80211NWKEY` : configurer la valeur de(s) clef(s) WEP

Ces requêtes `ioctl`(2) et d'autres sont décrites dans le fichier `<net80211/ieee80211_ioctl.h>`. On y trouve également les structures associées aux requêtes.

6. EXEMPLE N°3 : SCANNER ACTIVEMENT

Vous pouvez demander à votre driver, par conséquent à votre carte, de « balayer » l'environnement afin de trouver d'autres points d'accès dans son périmètre.

La requête se fait au travers de deux ioctl(2) 802.11 génériques :

- `SIOCS80211` : configurer des paramètres 802.11 ;
- `SIOCG80211` : récupérer la valeur des paramètres 802.11.

La structure associée à ces requêtes est `ieee80211req` définie dans le fichier `<net80211/ieee80211_ioctl.h>` (eh oui, encore lui !) :

```
struct ieee80211req {
  char       i_name[IFNAMSIZ];  /* if_name, e.g. "wi0" */
  u_int16_t  i_type;            /* req type */
  int16_t    i_val;             /* Index or simple value */
  int16_t    i_len;             /* Index or simple value */
  void      *i_data;            /* Extra data */
};
```

Elle définit un champ `i_type` qui permet de choisir une sous-requête (spécifique aux opérations wireless) à appeler.

Voici un exemple de sous-requêtes définies dans le fichier `<net80211/ieee80211_ioctl.h>` :

```
/* driver capabilities */
#define IEEE80211_IOC_DRIVER_CAPS       36
/* key management algorithms */
#define IEEE80211_IOC_KEYMGTALGS        37
/* RSN capabilities */
#define IEEE80211_IOC_RSNCAPS           38
/* WPA information element */
#define IEEE80211_IOC_WPAIE             39
/* per-station statistics */
#define IEEE80211_IOC_STA_STATS         40
/* MAC ACL operation */
#define IEEE80211_IOC_MACCMD            41
/* channel info list */
#define IEEE80211_IOC_CHANINFO          42
/* max tx power for channel */
#define IEEE80211_IOC_TXPOWMAX          43
/* per-station tx power limit */
#define IEEE80211_IOC_STA_TXPOW         44
/* station/neighbor info */
#define IEEE80211_IOC_STA_INFO          45
[...]
/* beacon interval (ms) */
#define IEEE80211_IOC_BEACON_INTERVAL   53
/* add sta to MAC ACL table */
#define IEEE80211_IOC_ADDMAC            54
/* del sta from MAC ACL table */
#define IEEE80211_IOC_DELMAC            55
```

Dans le cas du scan, les sous-requêtes nécessaires pour effectuer cette opération sont les suivantes :

- `IEEE80211_IOC_SCAN_REQ`
- `IEEE80211_IOC_SCAN_RESULTS`

L'application n'a absolument rien à faire, si ce n'est envoyer la demande au driver via ces appels. Le driver se chargera encore et toujours de la sale besogne : il enverra alors des *probe requests*, afin d'obtenir une réponse des points d'accès environnants. Lors d'une telle requête, le driver va simplement empiler les résultats en utilisant la structure suivante :

```
/*
 * Scan result data returned for IEEE80211_IOC_SCAN_RESULTS.
 */
struct ieee80211req_scan_result {
  u_int16_t  isr_len;           /* length (mult of 4) */
  u_int16_t  isr_freq;          /* MHz */
  u_int16_t  isr_flags;         /* channel flags */
  u_int8_t   isr_noise;
  u_int8_t   isr_rssi;
  u_int8_t   isr_intval;        /* beacon interval */
  u_int8_t   isr_capinfo;       /* capabilities */
  u_int8_t   isr_erp;           /* ERP element */
  u_int8_t   isr_bssid[IEEE80211_ADDR_LEN];
  u_int8_t   isr_nrates;
  u_int8_t   isr_rates[IEEE80211_RATE_MAXSIZE];
  u_int8_t   isr_ssid_len;      /* SSID length */
  u_int8_t   isr_ie_len;        /* IE length */
  u_int8_t   isr_pad[5];
  /* variable length SSID followed by IE data */
};
```

Les résultats sont écrits les uns derrière les autres dans le *buffer* que vous aurez fourni (dans le champ `i_data`) lors de la demande des résultats.

Le buffer, une fois rempli, devrait ressembler à ça :

`scan_result1`	structure `ieee80211req_scan_result` avec les informations concernant le premier point d'accès à avoir répondu
`result1_ssid`	valeur du ssid si il existe
`result1_ei`	valeur de l'élément d'information si il existe
`scan_result2`	structure avec les informations concernant le second point d'accès à avoir répondu
`result1_ssid`	
`result1_ei`	
...	

et ainsi de suite.

Lors de la demande des résultats, la taille effective des données écrasera la taille initiale du buffer dans le champ `i_len`.

Rien de mieux qu'un petit exemple avec `wifi_scan.c` :

```
/*
 *  code d'exemple sous license BSD
 *  Eric Auge <eau@gcu-squad.org>
 *
 *  pour compiler :
 *  gcc wifi_scan.c -o wifi_scan
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#include <sys/ioctl.h>
#include <net/if.h>
#include <net/if_media.h>

#include <net80211/ieee80211.h>
#include <net80211/ieee80211_ioctl.h>

#define IFACE_NAME "wpi0"

int main(int argc, char ** argv)
{
  struct ifreq ifr;
  char * interface = IFACE_NAME;
  int s, i;
  struct ieee80211req req;
  char buf[24*1024];
  char ssid[64];
  struct ieee80211req_scan_result * results;

  s = socket(AF_INET, SOCK_DGRAM, 0);
  if  (s <0)
  {
    perror("socket()");
    return -1;
  }

  memset(&req, 0, sizeof(struct ieee80211req));
  memset(buf, 0, sizeof(buf));
  memset(ssid, 0, sizeof(ssid));

  /* on demande à la carte de scanner les
     access points environnants */
  strlcpy(req.i_name, interface, sizeof(req.i_name));
  req.i_type = IEEE80211_IOC_SCAN_REQ;
  req.i_val = 0;

  if (ioctl(s, SIOCS80211, &req) < 0)
  {
    perror("ioctl(SIOCS80211)");
    return -1;
  }

  /* on attend un ptit peu, c'est pas joli je
     sais mais on le fait au plus simple pour
     le moment :)  */
  sleep(10);

  /* on demande au driver le résultat du scan */
  memset(&req, 0, sizeof(struct ieee80211req));
  strlcpy(req.i_name, interface, sizeof(req.i_name));
  req.i_type = IEEE80211_IOC_SCAN_RESULTS;
  req.i_len = sizeof(buf);
  req.i_data = &buf;
  if (ioctl(s, SIOCG80211, &req) < 0)
  {
    perror("ioctl(SIOCG80211)");
    return -1;
  }

  printf("taille des resultats: %d octets\n", req.i_len );
  printf("taille d'un resultat: %d\n",
    sizeof(struct ieee80211req_scan_result));

  /* observons le premier resultat */
  if (req.i_len > 0)
  {
    results = (struct ieee80211req_scan_result *) buf;
```

```
    printf("taille du resultat: %d taille du SSID: %d"
      " taille de l'element d'information: %d\n",
        results->isr_len,
        results->isr_ssid_len,
        results->isr_ie_len);
    printf("BSSID: ");
    for ( i = 0 ; i < IEEE80211_ADDR_LEN ; i++)
      printf("%02x ",  results->isr_bssid[i]);
    printf("\n");
    if (results->isr_ssid_len > 0)
    {
      memcpy(ssid, &buf[
        sizeof(struct ieee80211req_scan_result)
        ], results->isr_ssid_len);
      printf("SSID: %s\n", ssid);
    }
  }
  return 0;
}
```

A l'exécution, nous obtenons les résultats suivants :

```
root@kamehouse:~/lm # ./wifi_scan
taille des resultats: 48 octets
taille d'un resultat: 40
taille du résultat: 48 taille du SSID:
  8 taille de l'element d'information: 0
BSSID: 00 03 52 ef 72 60
SSID: swisscom
```

Le point d'accès près de moi au moment du scan est celui auquel j'étais attaché précédemment :

```
eau@kamehouse:~/lm # ifconfig wpi0
wpi0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST>
        mtu 1500
        ssid swisscom
        powersave off
        bssid 00:03:52:ef:72:60 chan 1
```

Le BSSID du scan et celui du point d'accès sont identiques.

Notez que l'API net80211 est en mouvement ! Des changements sont encore à venir concernant cette interface qui permet aux applications utilisateurs de demander un scan. Il se peut donc que les appels actuels soient déjà obsolètes ou en chantier. De plus, ils ne semblent pas complètement fonctionnels selon les cartes utilisées.

Auparavant, des outils comme `wimon` ou `wiconfig` utilisaient l'ioctl `SIOCGWAVELAN`, ainsi que les sous-requêtes associées `WI_RID_SCAN_APS` et `WI_RID_READ_APS`.

7. EXEMPLE N°4 : ACTIVER ET UTILISER LE MONITORING AKA RF_MON

Une grande majorité des cartes réseau sans fil proposent la capacité d'écouter passivement tout le trafic. Ce mode d'opération est souvent appelé « monitor mode » ou RFMON. Il est équivalent au mode « *promiscuous* » des réseaux filaires. Ce mode est celui utilisé pour la découverte de réseaux ou *wardriving* par des logiciels comme Kismet, Airsnort ou Wireshark.

Regardons comment nous pouvons utiliser ce mode d'opération et l'activer sur notre carte. L'outil `ifconfig` nous propose de voir et/ou configurer le mode d'opération de la carte.

```
wpi0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST>
        mtu 1500
        ssid default
        powersave off
        bssid 00:90:9c:00:0d:2e chan 1
        address: 00:01:02:03:04:11
        media: IEEE802.11 autoselect (OFDM18 mode 11g)
        status: active
```

La ligne qui nous intéresse est la ligne `media`, la carte fonctionne en 802.11 en auto-sélection (fréquence, mode, etc.), et aucune option ne semble être présente concernant ce mode de fonctionnement.

Pour altérer ces informations concernant l'interface wireless, il suffit de s'y prendre comme à l'exemple n°1. On va passer par notre bonne vieille structure `ifreq` et on ne va pas altérer les flags, mais le média maintenant.

C'est logiquement que je vais mettre les doigts sur le champ `ifr_media` (hmm, je touche le dernier 33 tours de Claude François, hmmm Cloclo, hmmmMmmMMmmmmm...) et voici les deux flags qui m'intéressent :

- `IFM_IEEE80211`
- `IFM_IEEE80211_MONITOR`

Allez hop, c'est parti, c'est bien plus parlant avec un exemple. Nous allons éteindre l'interface, activer le mode monitoring, rallumer l'interface et récupérer des données dessus. Pour simplifier, j'ai décidé d'utiliser la `libpcap`. Elle est simple d'utilisation, nécessite peu de lignes et elle m'est assez familière.

Le petit exemple avec `wifi_mon.c` :

```
/*
 *  code d'exemple sous license BSD
 *  Eric Auge <eau@gcu-squad.org>
 *
 *  pour compiler :
 *  gcc wifi_mon.c -o wifi_mon -lpcap
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

/* pcap to the rescue, je suis trop feignant pour
   utiliser autre chose...genre bpf etc.. */
#include <pcap.h>

#include <sys/ioctl.h>
```

```
#include <net/if.h>
#include <net/if_media.h>

#include <net80211/ieee80211.h>
#include <net80211/ieee80211_ioctl.h>

#define IFACE_NAME "wpi0"

/*
 * un exemple de _début_ de décodage de paquets
 * c'est un exemple hein !
 */
void ieee80211_decode(u_char * packet, size_t packet_size)
{
  struct ieee80211_frame*i=(struct ieee80211_frame *)packet;
  unsigned char fcs[4];
  int rc;

  printf("Paquet recu (%d bytes) >> ", packet_size);
  if (!i)
    return;

  if (i->i_fc[1] & IEEE80211_FC1_MORE_FRAG)
    printf("IEEE80211_FC1_MORE_FRAG ");

  switch (i->i_fc[0] & IEEE80211_FC0_TYPE_MASK)
  {
    case IEEE80211_FC0_TYPE_MGT:
      printf("IEEE80211 MANAGEMENT FRAME ");
      /*
       rc=ieee80211_decode_mgmt(&ctx,packet,packet_size);
       */
      break;
    case IEEE80211_FC0_TYPE_CTL:
      printf("IEEE80211 CONTROL FRAME ");
      /*
       rc=ieee80211_decode_ctl(&ctx,packet,packet_size);
       */
      break;
    case IEEE80211_FC0_TYPE_DATA:
      printf("IEEE80211 DATA FRAME ");
      /*
       rc=ieee80211_decode_data(&ctx,packet,packet_size);
       */
      break;
    default:
      printf("IEEE80211 UNKNOWN FRAME");
      /*
       rc=ieee80211_decode_unknown(&ctx,packet,packet_size);
       */
      break;
  }
  printf("\n");
}

/*
 * la jolie routine (callback) qui s'occupe de
 * traiter chaque paquet reçu,
 * merci pcap pour ce pain d'epice tout prêt !
 */
void pkt_handler(u_char * dump,
  const struct pcap_pkthdr * phdr, const u_char * pkt)
{
  ieee80211_decode((u_char *)pkt, phdr->caplen);
  return;
}
```

```
int main(int argc, char ** argv)
{
  struct ifreq ifr;
  int s;
  /* c'est parti! */
  pcap_t * p;
  char error[2048];
  int rc, i;
  int *dlt;
  int ndlt;

  if (!argv[1])
  {
    printf("no argument\n");
    printf("\n%s <iface>\n", argv[0]);
    exit(1);
  }

  /* il nous faut un file descriptor ouvert oui ! */
  s = socket(AF_INET, SOCK_DGRAM, 0);
  if  (s <0)
  {
    perror("socket()");
    return -1;
  }

  /* nom de l'interface */
  strlcpy(ifr.ifr_name, argv[1], sizeof(ifr.ifr_name));

  /* on switch l'interface down */
  ifr.ifr_flags &= (~IFF_UP);
  if (ioctl(s, SIOCSIFFLAGS, (caddr_t) &ifr) == -1)
  {
    perror("ioctl(SIOCSIFFLAGS:down)");
    return -1;
  }

  /*
   *  on change le options concernant le media,
   *  hola hup barbatruc, carte
   *  en monitor s'il vous plait monsieur le kernel !
   */
  ifr.ifr_media = IFM_IEEE80211|IFM_IEEE80211_MONITOR;
  if (ioctl(s, SIOCSIFMEDIA, (caddr_t)&ifr) == -1)
  {
    perror("ioctl(SIOCSIFMEDIA)");
    return -1;
  }

  /* on re switch l'interface up ! */
  ifr.ifr_flags |= IFF_UP;
  if (ioctl(s, SIOCSIFFLAGS, (caddr_t) &ifr) == -1)
  {
    perror("ioctl(SIOCSIFFLAGS:up)");
    return -1;
  }

  /* ouvrir l'interface pcap ! */
  p = pcap_open_live(argv[1], 1500, 1, 1000, error);
  if (!p)
  {
    perror("pcap_open_live()");
    printf("error: %s\n", error);
  }

  /* qui sommes nous ? */
  ndlt = pcap_list_datalinks(p, &dlt);
```

```
/* que sommes nous capables de supporter ! */
printf("# dlt: %d\n", ndlt);
for ( i = 0 ; i < ndlt ; i++) {
  switch (*(dlt+i)) {
    case DLT_IEEE802_11:
      printf("STD IEEE802_11 est supporte!\n");
      break;
    case DLT_PRISM_HEADER:
      printf("PRISM est supporte!\n");
      break;
    case DLT_AIRONET_HEADER:
      printf("HEADER AIRONET supporte!\n");
      break;
  }
}

/* on configure et on boucle */
pcap_set_datalink(p, DLT_IEEE802_11);
pcap_loop(p, -1, pkt_handler, NULL);

return 0;
}
```

A l'exécution :

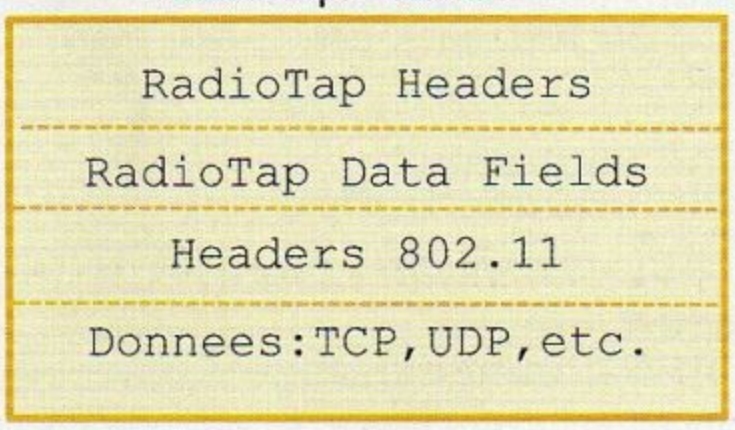
```
root@kamehouse:~/lm # ./wifi_mon wpi0
# dlt: 3
STD IEEE802_11 est supporte!
Paquet recu (81 bytes) >> IEEE80211 MANAGEMENT FRAME
Paquet recu (81 bytes) >> IEEE80211 MANAGEMENT FRAME
Paquet recu (81 bytes) >> IEEE80211 MANAGEMENT FRAME
Paquet recu (42 bytes) >> IEEE80211 MANAGEMENT FRAME
Paquet recu (81 bytes) >> IEEE80211 MANAGEMENT FRAME
Paquet recu (81 bytes) >> IEEE80211 MANAGEMENT FRAME
Paquet recu (42 bytes) >> IEEE80211 MANAGEMENT FRAME
Paquet recu (81 bytes) >> IEEE80211 MANAGEMENT FRAME
Paquet recu (81 bytes) >> IEEE80211 MANAGEMENT FRAME
Paquet recu (42 bytes) >> IEEE80211 MANAGEMENT FRAME
Paquet recu (81 bytes) >> IEEE80211 MANAGEMENT FRAME
```

Youpi, nous voilà donc avec quelque chose qui reçoit les paquets directement depuis le *kernel* sans modification, à la manière dont le font Kismet et Cie. J'ai inclus un micro-exemple de routine de décodage, histoire de donner des idées :)

8. EXEMPLE N°5 : RADIOTAP

RadioTap, initialement développé sur NetBSD, fournit une couche de description du signal commune à tous les drivers wireless.

Voilà la structure du paquet avec RadioTap activé :



Ce header transporte une variété d'informations liées à chaque trame reçue comme :

- la qualité du signal ;
- la qualité de la réception ;
- la présence ou non du FCS (*Frame Check Sequence*, une somme de vérification de l'intégrité du paquet).

Le header RadioTap se trouve dans le fichier <net80211/ieee80211_radiotap.h> et se présente de la manière suivante :

```
/* XXX tcpdump/libpcap do not tolerate
 * variable-length headers, yet, so we pad every
 * radiotap header to 64 bytes. Ugh.
 */
#define IEEE80211_RADIOTAP_HDRLEN       64

/*
 * The radio capture header precedes the 802.11 header.
 *
 * Note well: all radiotap fields are little-endian.
 */
struct ieee80211_radiotap_header {
/* Version 0. Only increases for drastic changes,
   introduction of compatible new fields does not count. */
u_int8_t        it_version;
u_int8_t        it_pad;

/* length of the whole header in bytes, including
   it_version, it_pad, it_len, and data fields.  */
u_int16_t       it_len;

/* A bitmap telling which
* fields are present. Set bit 31
* (0x80000000) to extend the
* bitmap by another 32 bits.
* Additional extensions are made
* by setting bit 31. */
u_int32_t       it_present;
} __attribute__((__packed__));
```

Les différents champs possibles sont décrits quelques lignes en dessous dans le même fichier, ainsi que dans la page de manuel « ieee80211_radiotap(9) ».

Le champ it_present vous indique quelles sont les informations que le driver vous remonte dans le paquet reçu. Il suffit donc de parser ce champ pour savoir exactement quelles informations nous pouvons obtenir du driver en temps réel et pour chaque trame. En exemple, nous allons prendre le programme précédent et rajouter le support RadioTap, en vous laissant, pour le fun, la joie d'écrire un parser approprié pour récupérer les informations dont vous avez besoin dans ce header :)

Le header RadioTap à *parser* se trouve dans le code d'exemple wifi_radiotap.c. Il est nommé r dans la fonction ieee80211_decode :

```
/*
 *  code d'exemple sous license BSD
 *  Eric Auge <eau@gcu-squad.org>
 *
 *  pour compiler :
 *  gcc wifi_radiotap.c -o wifi_radiotap -lpcap
 */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <pcap.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <net/if_media.h>
#include <net80211/ieee80211.h>
#include <net80211/ieee80211_ioctl.h>
#include <net80211/ieee80211_radiotap.h>

#define IFACE_NAME "wpi0"

/* AVEC RADIOTAP CETTE FOIS ! */
void ieee80211_decode(u_char * packet, size_t packet_size)
{
  struct ieee80211_radiotap_header * r =
    (struct ieee80211_radiotap_header *) packet;
  struct ieee80211_frame * i =
    (struct ieee80211_frame *)
    ((unsigned char*)packet+IEEE80211_RADIOTAP_HDRLEN);
unsigned char fcs[4];
int rc;

printf("Paquet recu (%d bytes) >> ", packet_size);

if ((!i) || (packet_size < (sizeof(struct ieee80211_frame)
      +IEEE80211_RADIOTAP_HDRLEN)))
  return;

if (i->i_fc[1] & IEEE80211_FC1_MORE_FRAG)
  printf("IEEE80211_FC1_MORE_FRAG ");

switch (i->i_fc[0] & IEEE80211_FC0_TYPE_MASK)
{
  case IEEE80211_FC0_TYPE_MGT:
    printf("IEEE80211 MANAGEMENT FRAME");
    break;
  case IEEE80211_FC0_TYPE_CTL:
    printf("IEEE80211 CONTROL FRAME");
    break;
  case IEEE80211_FC0_TYPE_DATA:
    printf("IEEE80211 DATA FRAME");
    break;
  default:
    printf("IEEE80211 UNKNOWN FRAME");
    break;
  }

  printf("\n");
}

void pkt_handler(u_char * dump,
  const struct pcap_pkthdr * phdr,
  const u_char * pkt)
{
  ieee80211_decode((u_char *)pkt, phdr->caplen);
  return;
}

int main(int argc, char ** argv)
{
  struct ifreq ifr;
  char * interface = IFACE_NAME;
  int s;
  /* c'est parti! */
```

```
pcap_t * p;
char error[2048];
int rc, i;
int *dlt;
int ndlt;

if (!argv[1])
{
  printf("no argument\n");
  printf("\n%s <iface>\n", argv[0]);
  exit(1);
}

/* il nous faut un file descriptor ouvert oui ! */
s = socket(AF_INET, SOCK_DGRAM, 0);
if  (s <0)
{
  perror("socket()");
  return -1;
}

/* nom de l'interface */
strlcpy(ifr.ifr_name, interface, sizeof(ifr.ifr_name));

/* on switch l'interface down */
ifr.ifr_flags &= (~IFF_UP);
if (ioctl(s, SIOCSIFFLAGS, (caddr_t) &ifr) == -1)
{
  perror("ioctl(SIOCSIFFLAGS:down)");
  return -1;
}

ifr.ifr_media = IFM_IEEE80211|IFM_IEEE80211_MONITOR;
if (ioctl(s, SIOCSIFMEDIA, (caddr_t)&ifr) == -1)
{
  perror("ioctl(SIOCSIFMEDIA)");
  return -1;
}

/* on re switch l'interface up ! */
ifr.ifr_flags |= IFF_UP;
if (ioctl(s, SIOCSIFFLAGS, (caddr_t) &ifr) == -1)
{
  perror("ioctl(SIOCSIFFLAGS:up)");
  return -1;
}

/* ouvrir l'interface pcap ! */
p = pcap_open_live(argv[1], 1500, 1, 1000, error);
if (!p)
{
  perror("pcap_open_live()");
  printf("error: %s\n", error);
}

/* qui sommes nous ? */
ndlt = pcap_list_datalinks(p, &dlt);

/* que sommes nous capables de supporter ! */
printf("# dlt: %d\n", ndlt);
for ( i = 0 ; i < ndlt ; i++) {
  switch (*(dlt+i)) {
    case DLT_IEEE802_11:
      printf("STD IEEE802_11 est supporte!\n");
      break;
    case DLT_PRISM_HEADER:
```

```
        printf("PRISM est supporte!\n");
        break;
      case DLT_AIRONET_HEADER:
        printf("HEADER AIRONET supporte!\n");
        break;
      case DLT_IEEE802_11_RADIO:
        printf("IEEE802_11_RADIO (RADIOTAP) est supporte\
n");
        break;
    }
  }

  /* on configure et on boucle */
  pcap_set_datalink(p, DLT_IEEE802_11_RADIO);
  pcap_loop(p, -1, pkt_handler, NULL);
  return 0;
}
```

A l'exécution, peu de changements :)

```
root@kamehouse:~/lm # ./wifi_radiotap wpi0
# dlt: 3
IEEE802_11_RADIO (RADIOTAP) est supporte
STD IEEE802_11 est supporte!
Paquet recu (145 bytes) >> IEEE80211 MANAGEMENT FRAME
Paquet recu (145 bytes) >> IEEE80211 MANAGEMENT FRAME
Paquet recu (145 bytes) >> IEEE80211 MANAGEMENT FRAME
Paquet recu (145 bytes) >> IEEE80211 MANAGEMENT FRAME
Paquet recu (145 bytes) >> IEEE80211 MANAGEMENT FRAME
Paquet recu (145 bytes) >> IEEE80211 MANAGEMENT FRAME
```

Et voilà !

CONCLUSION

Certaines choses n'ont pas été abordées dans cet article, comme l'envoi de paquets 802.11 depuis les applications utilisateurs ou simplement la gestion de WPA, la gestion de l'énergie, les statistiques, la mesure du signal (même si RadioTap donne déjà ces dernières informations). Cet article se voulait une introduction pragmatique à l'utilisation de cette couche d'abstraction et du wireless en C sous NetBSD pour des applications utilisateurs.

J'espère que l'article pourra en aider quelques-uns à démarrer ou à comprendre comment utiliser cette API, l'abstraction étant étroitement partagée entre NetBSD et FreeBSD. Les exemples demanderont un effort minime pour fonctionner également sous FreeBSD.

J'espère également que vos connaissances en matière de variété française ont été améliorées et que les nombreux artistes qui ont fait notre culture musicale (merci Gilbert M. !) continueront à vous bercer après la lecture de cet article.

Voilà, c'est terminé, amusez-vous bien, vous pouvez maintenant repartir vous entraîner à chanter « L'été indien » devant votre miroir.

On irrrrrraaaaa, où tu voudraaaaaAaas quand tuuuuuu vouUUdrrAaaaaaAAAaaaaaas et on s'aimera encoOOOOOOooooooreee, lorsque l'amMMMMooouuuUUUurr sera moOOooort... (Vive les tomates !).

RÉFÉRENCES

- NetBSD : http://netbsd.org
- FreeBSD : http://freebsd.org
- Page de manuel « ieee80211(4) »
- Page de manuel « ieee80211(9) »
- Page de manuel « ieee80211_radiotap(9) »
- /usr/src/sbin/ifconfig/ieee80211.c
- /usr/include/net80211/*.h
- http://www.freebsd.org/~sam/BSDCan2005.pdf
- http://www.guill.net/