Keepalived : haute disponibilité et répartition de charge enfin libérées !

Alexandre SIMON
CIRIL - Centre Interuniversitaire de Ressources Informatiques de Lorraine
Rue du Doyen Roubault
54500 VANDŒUVRE-LÈS-NANCY – FRANCE

Résumé

Le domaine de la haute disponibilité des systèmes informatiques et plus particulièrement les solutions de répartition de charge sont aujourd'hui la chasse gardée de constructeurs d'équipements orientés réseaux. Il existe pourtant depuis de nombreuses années des solutions logicielles alternatives tout aussi pertinentes et permettant d'aborder les mêmes problématiques avec des niveaux de performances très acceptables.

Keepalived est une solution (libre et gratuite) « tout en un » de haute disponibilité, de virtualisation de services applicatifs et de répartition de charge. Elle s'appuie entièrement sur des fonctionnalités du noyau Linux et des protocoles standardisés tout en consolidant ces technologies dans une même solution. Keepalived est un outil extrêmement léger, très facile à appréhender, à configurer et à déployer. Il intègre les fonctionnalités « classiques » de haute disponibilité réseau telles que :

- *la virtualisation et la redondance d'adressage IP (VRRP) ;*
- *la virtualisation de services réseaux et la répartition sur plusieurs nœuds (IPVS) ;*
- *la surveillance et l'utilisation rationnelle des ressources disponibles (HEALTCHECKER).*

Au cours des derniers mois, les « briques » utilisées par Keepalived ont toutes intégré le support d'IPv6. Ces améliorations font de Keepalived un des rares outils consolidant les parties VRRP, IPVS et surveillance de services (health-checking) en IPv4 et IPv6 natifs.

Cet article fait le point sur les généralités d'architectures hautement disponibles et les principes de virtualisation des services applicatifs réseaux. Le framework Netfilter et le module de répartition de charge IPVS seront abordés afin de poser les briques de base disponibles avec Linux. La solution Keepalived sera présentée de manière à comprendre son fonctionnement interne, son originalité et toutes ses possibilités de mises en œuvre. Enfin des exemples concrets de déploiements illustreront Keepalived comme solution de « cœur » pour la haute disponibilité.

Mots clefs

Haute disponibilité, répartition de charge, *Linux*, *Netfilter*, *IPVS*, *VRRP*, *Keepalived*.

1 Introduction

La « haute disponibilité » est un domaine très large qui se retrouve dans les différentes briques composant les systèmes informatiques modernes (matériels réseau, serveurs, *SAN*, procédures, logiciels, …). Elle est souvent appliquée depuis les couches les plus basses (composants électroniques, redondances physiques, code de correction d'erreur, …) jusqu'aux couches les plus hautes (protocoles et procédures fiabilisées, *clusters*, PRA, …). Cet article se focalise sur la haute disponibilité réseau des couches 3 et 4 du modèle *OSI*. En d'autres termes, ce sont les techniques de virtualisation de services réseaux *TCP/UDP IPv4* et *IPv6* et la répartition de charge vers des serveurs « réels » qui seront traitées ici.

Avec la fonctionnalité noyau *IPVS* et l'utilisation d'un démon *VRRP*, *Linux* est tout à fait capable d'assurer des fonctions de haute disponibilité et de répartiteur de charge. Cependant, même si ces technologies sont stables et unitairement simples à mettre en œuvre, peu d'administrateurs les utilisent comme solutions de cœur pour leur haute disponibilité. La cause probable à ce constat est sans doute le manque de « glu » et de consolidation de ces technologies dans une même solution.

Dans un premier temps, cet article « posera le décor » en faisant le point sur l'environnement réseau *Netfilter* et sur le module *IPVS* de *Linux*. Ensuite, la solution *Keepalived* sera présentée afin de comprendre son intérêt notamment dans un contexte d'unification des technologies disponibles sous *Linux*. Enfin des exemples concrets de déploiement, tirés de l'expérience de l'équipe réseau du CIRIL, seront explicités de manière à mettre en application toutes les notions présentées.

2 *Linux*, *Netfilter* et *IPVS* : un jeu de construction pour la haute disponibilité réseau

2.1 Contexte

Le choix de *Linux* comme système d'exploitation pour la mise en œuvre d'applications hautement disponibles n'est plus à démontrer. Néanmoins, il est utile de s'attarder un moment sur les fonctionnalités et l'architecture réseau interne du noyau qui en font une « boîte à outils réseau » et qui ont permis de développer les bases de la haute disponibilité sous *Linux*. L'originalité de *Linux* est la gestion de la pile réseau et plus particulièrement le traitement des paquets dans le noyau qui sont architecturés autour du *Framework Netfilter*. Les deux chapitres suivants détailleront l'environnement *Netfilter* et la fonctionnalité *IPVS* (*IP Virtual Server*) qui s'appuie sur *Netfilter* en constituant la solution de virtualisation et de répartition de charge réseau embarquée dans *Linux*.

2.2 Netfilter

2.2.1 Description

L'environnement *Netfilter* décrit et implémente la gestion des flux des paquets réseaux (niveau 2 et supérieurs) traversant le noyau *Linux*. Il propose des points d'accroche (*hooks*) sur lesquels des fonctions spécifiques vont pouvoir intercepter et manipuler les paquets en provenance ou à destination des interfaces réseaux. Pour illustrer ce principe la Figure 1 détaille l'acheminement des paquets *IP* au travers du noyau.

De manière simplifiée, les paquets provenant des interfaces réseaux (*eth*, *lo*, *bond*, …) s'engagent par le « point d'entrée ». À cette étape, le noyau prend une décision de routage pour déterminer si le paquet est à destination de l'hôte (à destination d'un processus local, *Apache* par exemple) ou si le paquet doit être routé/redirigé (*forward*) vers une autre destination.

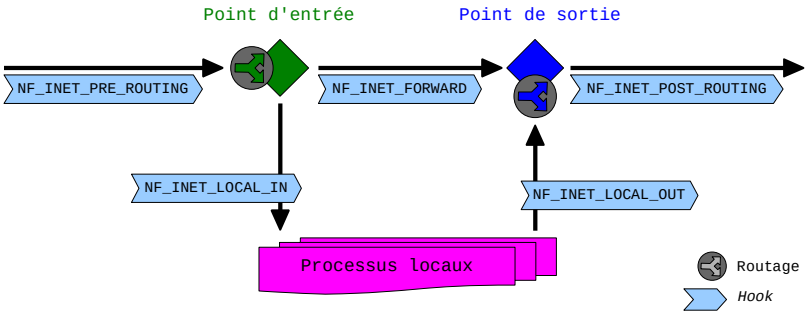

Figure 1: *Netfilter* - Flux des paquets *IP* et *hooks* disponibles

A l'autre extrémité, le « point de sortie » est le passage obligé des paquets directement routés depuis l'entrée et de ceux en provenance des processus locaux (*ping* depuis le serveur ou réponse *d'Apache* par exemple). Pour ces paquets « locaux », une décision de routage est prise par le noyau afin de déterminer la passerelle suivante à utiliser pour acheminer les paquets.

Aux différents étapes de transition (interfaces réseaux → point d'entrée, processus locaux → point de sortie, …) l'environnement *Netfilter* prévoit des points d'accroche (*hooks*) utilisés par des fonctionnalités du noyau afin de réaliser des opérations telles que le filtrage, les translations d'adresses (*NAT*) ou la modification/transformation des paquets (champs *DSCP*, *TTL*, …). Ces points d'accroche sont au nombre de cinq dans *Netfilter* au niveau *IP* :

- `NF_INET_PRE_ROUTING` : fonctions exécutées au plus tôt, juste avant la décision de routage en entrée, avec la possibilité de modifier les paquets mais également d'influencer la décision de routage ;
- `NF_INET_FORWARD` : fonctions appliquées sur les paquets routés par l'hôte (pas à destination de processus locaux) ;
- `NF_INET_LOCAL_IN` : traitement des paquets à destination des processus locaux ;
- `NF_INET_LOCAL_OUT` : traitement des paquets en provenance des processus locaux, avec la possibilité d'influencer la décision de routage de sortie ;
- `NF_INET_POST_ROUTING` : fonctions exécutées au plus tard, juste après la décision de routage en sortie.

Selon ce principe, les différentes fonctionnalités du noyau implémentent leurs traitements en « branchant » des fonctions spécifiques sur les points d'accroche adaptés à l'opération à effectuer. Le noyau se charge d'exécuter une après l'autre les fonctions enregistrées sur chaque point d'accroche en transmettant le paquet réseau en argument de la fonction.

Netfilter traite la traversée des paquets de niveau 2 et supérieurs, il existe donc des schémas de flux (du même type que celui de la Figure 1) spécifiques aux trames *Ethernet*, aux paquets *ARP* et aux paquets *DECnet*, avec les points d'accroche associés.

2.2.2 Utilisation de *Netfilter* par *Linux* ?

L'environnement *Netfilter* est « ouvert » et utilisable pour des développements tiers, mais le premier utilisateur de cet environnement est le noyau lui-même. En effet, l'abstraction et la généricité de ce modèle ont permis de concentrer en un même point le principe de traitement des paquets réseaux tout en permettant d'y brancher des fonctionnalités à valeurs ajoutées. Ainsi, *Netfilter* est l'environnement de base pour l'implémentation des fonctionnalités *Linux* suivantes :

- ***ip_tables***, ***ip6_tables*** : *Linux* définit des tables dédiées aux fonctions de filtrage (table *filter*), de translation d'adresses (table *nat*), de modification/transformation (table *mangle*) et d'accès brut (table *raw*) des paquets *IPv4* et *IPv6* . Ces tables peuvent être programmées (via les commandes en ligne `iptables` et `ip6tables`) avec des règles qui seront appliquées aux paquets traversant le noyau.
- ***arp_tables*** : de manière similaire aux tables précédentes, *Linux* permet de définir (via la commande `arptables`) des règles de filtrage (table *filter*) sur les paquets *ARP*.
- ***ebtables*** : les trames *Ethernet* ne sont pas oubliées par *Netfilter* et des tables de filtrage (table *filter*), de translation d'adresses (table *nat*) et routage de niveau 2 (table *broute*) pourront être programmées (via la commande `ebtables`) pour agir sur les paquets *Ethernet* traversant le noyau.
- ***conntrack*** : certains protocoles de niveau 4 (*TCP*) et supérieurs (*FTP*, *H323, ...*) sont dits « connectés ». En effet, les paquets peuvent être mis en relation (notion d'ordre, d'acquittement, de session, …) les uns avec les autres grâce à leur inspection et à l'application du protocole utilisé. C'est la fonctionnalité de *conntrack* (traçabilité des connexions) de *Linux* qui assure ces opérations et qui permet d'accéder aux états de session de protocoles tels que *TCP*, *FTP*, *SIP*, *SCTP*, …
- ***ulogd2*** : avec *ulogd2*, l'utilisateur va pouvoir demander la « remontée » de paquets réseau dans des programmes utilisateurs (*userland*). Une fois le paquet accessible, l'utilisateur pourra le consulter et y appliquer des modifications avant de le renvoyer vers le noyau.
- ***ipvs*** : le module *IPVS* du noyau est une fonction avancée de virtualisation de service réseau *IP* et de routage et répartition de charge vers des serveurs réels. Cette fonctionnalité est détaillée dans le chapitre 2.3.

Dans ce contexte, il est maintenant plus facile de segmenter l'environnement et les fonctionnalités réseau de *Linux*. A titre d'exemple, les administrateurs qui disent utiliser « les *IPtables* » ou « *Netfilter* » comme solution de filtrage des paquets *IP* devraient plutôt dire « les *IPtables* qui s'appuient sur l'environnement *Netfilter* ». La Figure 2[1] propose une synthèse de l'environnement *Netfilter* vis à vis des fonctionnalités réseau disponibles avec *Linux*. Un schéma complet de l'environnement *Netfilter* détaillant les flux, les tables et les points d'accroche pour les paquets *IPv4/6* et les trames *Ethernet* est disponible ici[2].

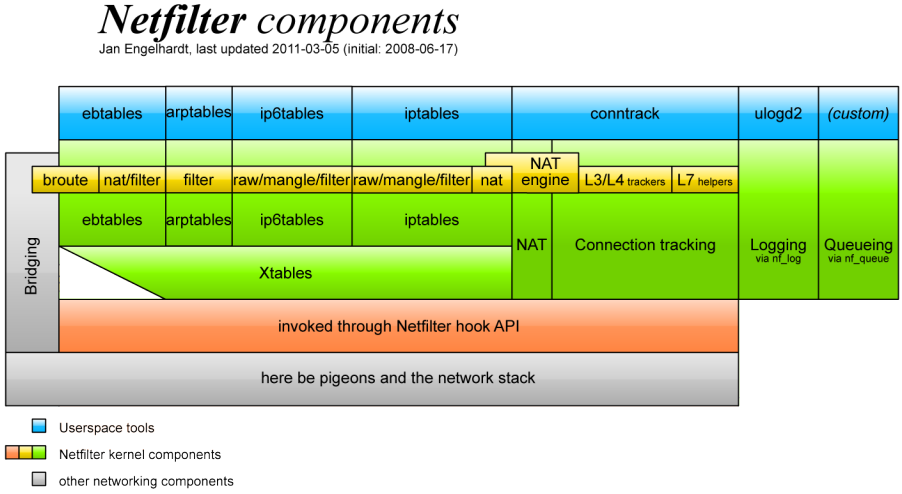

Figure 2: Synthèse *Netfilter* / fonctionnalités réseau *Linux*

[1] Source : *Jan ENGELHARDT - http://jengelh.medozas.de/images/nf-components.svg*

[2] Source : *Jan ENGELHARDT - http://jengelh.medozas.de/images/nf-packet-flow.svg*

2.3 *IPVS - IP Virtual Server*

2.3.1 LVS, IPVS... and co.

Depuis 1998, Wensong ZHANG est à la tête du projet libre *Linux Virtual Server (LVS)*[3]. *LVS* a pour objet de développer une solution avancée, robuste et performante de virtualisation de service réseau et de répartition de charge pour *Linux*. Le projet se concentre plus particulièrement sur les développements :

- *d'IPVS (IP Virtual Server)* : un module *Linux* de répartition de charge *IP* (niveau 3 et 4) ;
- de *KTCPVS (Kernel TCP Virtual Server)* : un module *Linux* de répartition de charge applicatif (niveau 7) ;
- de composants pour la gestion d'ensemble de serveurs (*cluster*).

Le site *web* du projet est également une mine précieuse de documentation, de guides de configuration et de références vers d'autres outils du domaine. *LVS* n'est pas un logiciel ni une solution clef en main, mais plutôt un ensemble de principes et d'outils permettant la mise en œuvre de solutions de haute disponibilité réseau. La suite de ce chapitre traite des concepts décrits par *LVS* en détaillant le module noyau *IPVS* incontournable dans ce domaine.

2.3.2 Virtualisation de services réseau et principes de répartition de charge

Pour rendre un service hautement disponible, par exemple un site *web*, il est naturel de mettre en production plusieurs serveurs assurant l'hébergement de ce site. Ainsi, la panne ou la maintenance d'un des serveurs n'affectera pas le service, celui-ci restant assuré par les autres serveurs disponibles. Afin de présenter un service « unifié » aux utilisateurs (ne pas leur communiquer les adresses de tous les serveurs disponibles) il faut « virtualiser » le service réseau et mettre en place un acheminement des requêtes et une répartition de charge vers l'ensemble des serveurs hébergeant réellement le site.

La première solution, très facile à mettre en œuvre, consiste à utiliser le *DNS* (*Domain Name System*) pour faire pointer le nom du service à fiabiliser vers plusieurs enregistrements. Par exemple le site *web reseau.ciril.fr* pourrait être déclaré comme ceci :

```
reseau.ciril.fr.    IN    CNAME   tic.ciril.fr.
                    IN    CNAME   tac.ciril.fr.
```

Ainsi des résolutions successives de *reseau.ciril.fr* fourniront alternativement les noms *tic.ciril.fr* et *tac.ciril.fr*, ce qui aura pour effet d'orienter le trafic utilisateur alternativement sur le serveur *TIC* puis sur le serveur *TAC*. Il n'est pas utile d'approfondir cette technique qui dévoile assez rapidement ses limites. En effet, les différents *caches* de l'architecture *DNS* ne permettent pas une répartition de charge maîtrisée : le *cache* côté client oriente toujours le client vers le même serveur et la répartition selon l'algorithme « tourniquet » est la seul possible. D'autre part, l'indisponibilité d'un des serveurs réels est difficile à prendre en compte car même si le *DNS* est modifié rapidement par l'administrateur, la propagation des changements dans l'ensemble du système *DNS* est soumis à l'expiration du *TTL* (*Time To Live*) des enregistrements modifiés.

La deuxième solution met en œuvre un élément actif assurant le rôle de répartiteur (répartiteur de charge ou *load balancer*) des requêtes utilisateurs sur le service virtualisé vers les serveurs réels. La Figure 3 met en situation cette solution de haute disponibilité. Le répartiteur (*LVS director*) est une machine physique qui est adressée avec l'adresse du service réseau à fiabiliser : la *VIP* (*Virtual IP address*). Il est le « passage obligé » du trafic en provenance des utilisateurs qui accèdent au service, ceux-ci n'ayant connaissance que de l'adresse *VIP* ou d'une déclaration *DNS* faisant référence à cette *VIP*.

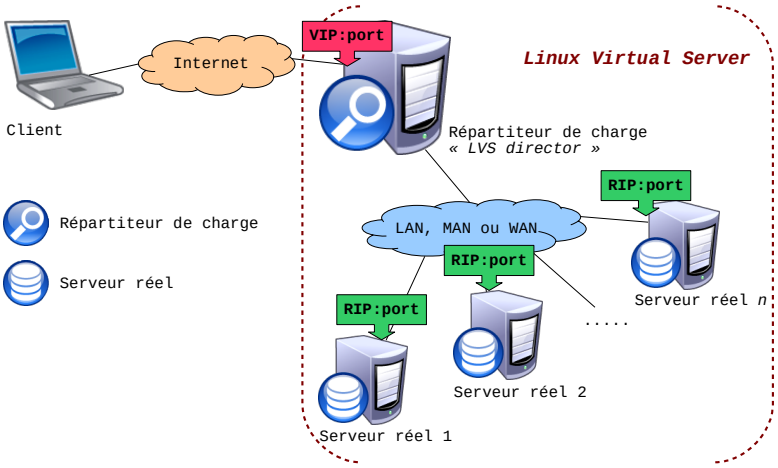

Figure 3: Schéma de principes « répartition de charge » *LVS*

Le répartiteur est configuré pour transmettre, selon plusieurs algorithmes, le trafic utilisateur vers plusieurs serveurs réels. De leur côté, les serveurs réels implémentent concrètement le service à rendre (démons *HTTP*, *SMTP*, …) et sont adressés avec des adresses *IP* souvent inconnues et invisibles pour les utilisateurs : les *RIP* (*Real IP addresses*).

[3] *http://www.linuxvirtualserver.org*

2.3.3 *Modes de redirection des paquets*

Il existe trois modes, schématisés par la Figure 4, de redirection/routage des paquets *IP* entre les éléments client / répartiteur / serveurs réels. Ces trois modes ne sont pas équivalents en terme de prérequis réseau, de complexité de mise en œuvre et de performance quant au passage à l'échelle.

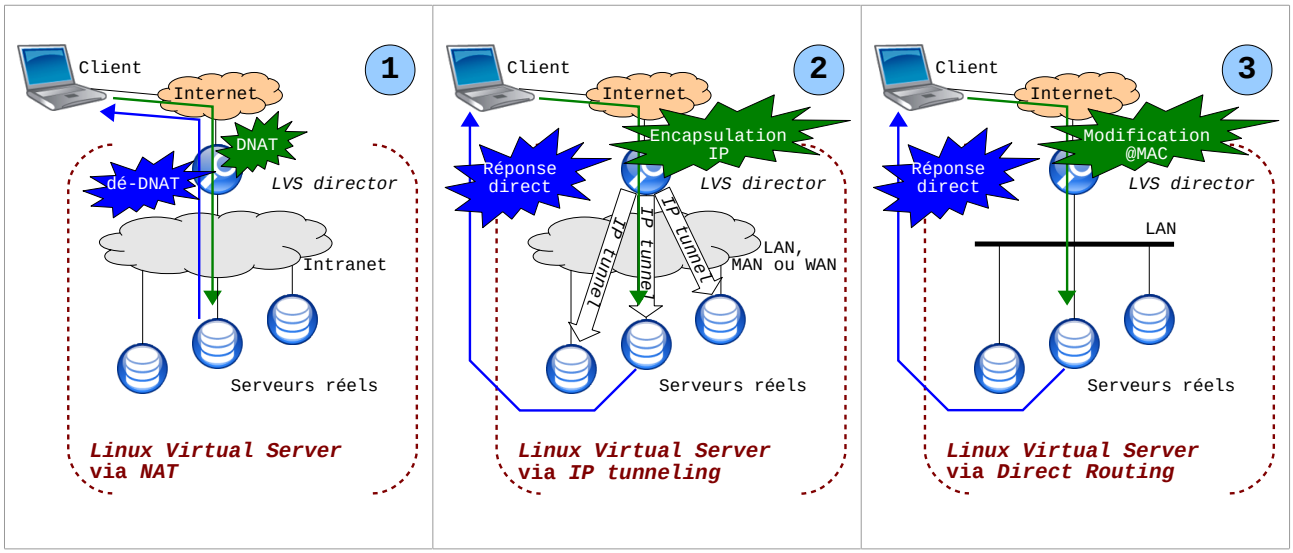

Figure 4: Modes de redirection des paquets (*packet forwarding method*)

Le mode (1) « via *NAT* » implémente une translation d'adresse de type *DNAT* (*Destination* NATing) dans le répartiteur. Cette translation a pour effet de remplacer l'adresse *VIP* du service par les adresses *RIP* des serveurs réels. Les serveurs peuvent être adressés avec des adresses *IP* publiques ou privées mais doivent impérativement utiliser le répartiteur comme passerelle par défaut pour les paquets retour. De ce fait, le réseau entre le répartiteur et les serveurs réels doit être adapté (politique de routage de *l'intranet* maîtrisée) pour permettre le trafic retour. Dans ce mode de redirection, le répartiteur est le point de passage obligé pour le trafic aller ***et*** retour, ce qui peut en faire un goulot d'étranglement potentiel.

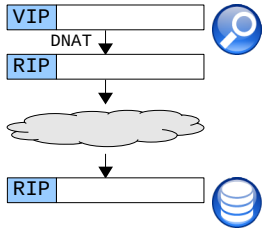

Figure 5: *IPVS « DNAT »*

L'implémentation d'un répartiteur et de serveurs réels sur un réseau « non maîtrisé » ou étendu (type *MAN*/*WAN*) peut être mise en œuvre par le mode (2) via des tunnels *IP (IP tunneling*). Dans ce cas, la non maîtrise du routage entre le répartiteur et les serveurs est compensée par la construction de tunnels *IP* assurant une connectivité « directe » entre les éléments. La configuration des tunnels est à la charge de l'administrateur alors que l'encapsulation dans les tunnels du trafic « aller » est assurée automatiquement par le répartiteur. Les réponses des serveurs réels sont directement adressées aux clients sans re-transiter par le répartiteur. Cette solution offre de meilleurs performances que la solution (1) tout en autorisant un déploiement sur des réseaux non maîtrisés. La configuration des tunnels reste néanmoins une opération complexe et fastidieuse pour l'administrateur.

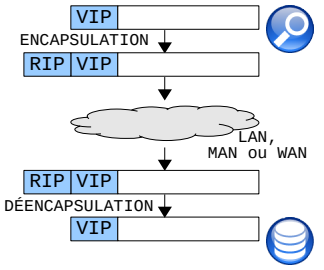

Figure 6: *IPVS « IP Tunneling »*

La technique (3) de routage direct (*Direct Routing*) manipule les adresses *MAC* des trames *Ethernet* entre le répartiteur et les serveurs réels. Avec cette technique, le répartiteur va substituer l'adresse *MAC* de la trame d'origine par l'adresse *MAC* du serveur réel à utiliser, après quoi il retransmet la trame modifiée sur le lien physique. La trame modifiée est nativement acheminée par le réseau de niveau 2 au serveur visé. Cette modification d'adresse *MAC* est très simple à mettre en œuvre par le répartiteur et peu

coûteuse, ce qui apporte à cette solution les performances les plus élevées par rapport aux deux autres modes présentés. La seule contrainte du routage direct est que le répartiteur et tous les serveurs réels doivent avoir un lien commun sur le même segment de réseau de niveau 2 (répartiteur et serveurs avec une interface réseau sur le même *VLAN*).

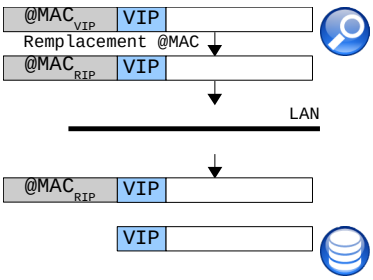

Figure 7: *IPVS* « *Direct Routing* »

Le choix du mode de redirection des paquets est conditionné par l'environnement réseau disponible entre le répartiteur et les serveurs réels, la stratégie de localisation des serveurs – proche du répartiteur (*LAN*) ou distribués géographiquement (*MAN/WAN*) – , le niveau de performance requis et l'adressage *IP* des serveurs (adresses *IP* publiques ou privées).

Il n'est pas inutile de répéter *qu'IPVS* s'appuie sur l'environnement *Netfilter* présenté dans le chapitre 2.2, en attachant sur les points d'accroche *Netfilter* des fonctions spécialisées au traitement de répartition de charge pour les manipulations *DNAT*, *IP Tunneling* ou *Direct Routing*. Enfin, il faut préciser que le protocole *IPv6* et son traitement dans *IPVS* n'ont pas été écartés par *Linux*. Ils ont été intégrés dans la version 2.6.28-rc3 (novembre 2008) du noyau et la version 1.25 (novembre 2008) de l'utilitaire `ipvsadm`.

2.3.4 Algorithmes de répartition

Lors de la configuration du répartiteur, l'administrateur va affecter à un service virtualisé un ensemble de serveurs réels qui assureront le traitement effectif des requêtes utilisateur. Le choix de la redirection d'une requête vers un serveur réel varie en fonction de l'algorithme de répartition de charge choisi. *IPVS* implémente 10 algorithmes différents : *Round-Robin* (**rr**), *Weighted Round-Robin* (**wrr**), *Least-Connection* (**lc**), *Weighted Least-Connection* (**wlc**), *Locality-Based Least-Connection* (**lblc**), *Locality-Based Least-Connection with Replication* (**lblcr**), *Destination Hashing* (**dh**), *Source Hashing* (**sh**), *Shortest Expected Delay* (**sed**) et *Never Queue* (**nq**).

Le détail de chaque algorithme d'ordonnancement peut être consulté ici[4].

2.3.5 L'article est-il fini ?

La description précédente des principes *LVS* et du module *IPVS* laisse à penser qu'ils sont suffisants pour déployer une solution de haute disponibilité. Il est vrai, qu'avec la commande en ligne `ipvsadm`, il est possible de piloter le module *IPVS* et de configurer la virtualisation d'un service réseau et sa répartition de traitement sur plusieurs serveurs réels selon les modes et algorithmes décrits précédemment. Mais tous les aspects de fiabilisation et de mise en œuvre ne sont pas traités par *IPVS* :

- Comment rendre hautement disponible la fonction du répartiteur lui-même ?
- Comment prendre en compte les indisponibilités des serveurs réels en modifiant dynamiquement le répartiteur avec les serveurs véritablement opérationnels ?
- Comment industrialiser le déploiement et la gestion de services fiabilisés ?

Ces points sont en dehors du périmètre de compétence *d'IPVS* et sont plutôt du ressort de solutions tierces qui prennent en compte ces problématiques tout en s'appuyant sur *IPVS* pour les parties « répartition de charge et algorithmes d'ordonnancement ». Des solutions telles que *Piranha*[5], *surealived*[6], *Linux-HA (heartbeat)*[7] ou *Keepalived* se positionnent justement sur ce créneau. La chapitre suivant va traiter de *Keepalived* qui apparaît comme la solution la plus complète et la plus simple à mettre en œuvre en proposant une gestion rationalisée d'un ensemble de serveurs coopérant à une répartition de charge.

[4] Algorithmes d'ordonnancement *IPVS* : *http://www.linuxvirtualserver.org/docs/scheduling.html*

[5] *Piranha : http://www.redhat.com/software/rha/cluster/piranha*

[6] s*urealived :http://surealived.sourceforge.net*

[7] *Linux-HA (heartbeat) :http://www.linux-ha.org*

3 Keepalived

3.1 Quand les dinosaures étaient petits...[8]

Keepalived[9] est un logiciel français développé par Alexandre CASSEN depuis décembre 2000. Après avoir fait un état des lieux des outils disponibles pour piloter et compléter les fonctionnalités *IPVS,* Alexandre s'est rapidement rendu compte du vide en la matière et il a naturellement décidé de se lancer dans le développement de *Keepalived*.

Afin de répondre aux problématiques non traitées par *IPVS* (comme exprimé dans le chapitre 2.3.5), *Keepalived* a été architecturé autour de deux fondamentaux :

- un module de test et de vérification de disponibilité de services réseau des serveurs réels : *HEALTCHECKER ;*
- un module de « résistance aux pannes » (*failover*) permettant d'assurer l'existence et la migration de l'adresse *IP* virtualisée (*VIP*) d'un service fiabilisé : implémentation du protocole *VRRP* (*Virtual Router Redundancy Protocol*).

Evidemment, *Keepalived* s'appuie sur le module *IPVS* pour la programmation du noyau et la prise en charge de la répartition charge.

Keepalived est écrit en langage *C* et le code a été pensé selon des modèles empruntés aux architectures logicielles orientées « démon logiciel et communication réseau » (*watchdog* logiciel, multiplieur centralisé d'entrées/sorties, gestion stricte de la mémoire, supervision inter-processus, …). Le code a été modularisé autour d'une librairie de cœur et les fonctionnalités contingentées à leur périmètre d'action. Ces efforts et ces techniques de développement assurent la fiabilité des fonctionnalités implémentées et permettent un *débogage* et une maintenabilité plus aisés.

Certaines versions ou fonctionnalités du logiciel ont été financées par des constructeurs, des centres de données (*data center*) ou des fournisseurs d'accès *Internet* (*ISP*), ce qui démontre l'attrait de cette solution de haute disponibilité pour ces entreprises.

Keepalived est considéré par la communauté (et par son auteur !) comme une solution mature et stable, capable de prendre en compte des problématiques de haute disponibilité au cœur d'importants systèmes informatiques.

3.2 Architecture globale

La Figure 8[10] décrit l'architecture globale de *Keepalived*. Au niveau espace utilisateur (*User space*), on retrouve la fonctionnalité *VRRP* qui assure la partie « résistance aux pannes » et la gestion de l'adresse *VIP.* Ce module pilote la configuration réseau de l'hôte (ajout/suppression d'adresses *IP*, de routes, …) via des commandes *NETLINK* passées directement au noyau *Linux.*

Il y a également le module *Checkers* qui implémente différents tests pour la vérification de disponibilité des serveurs réels. En fonction des résultats des tests (service / serveur disponible ou non), ce module (re)programme dynamiquement la configuration *IPVS* en assurant que seuls les serveurs opérationnels participent au traitement des requêtes utilisateur et à la répartition de charge.

La mise en œuvre de *Keepalived* sur un serveur *Linux* est des plus simples, pour ne pas dire « un jeu d'enfants ». D'une part, le logiciel est disponible en paquets auto-installables dans les distributions *Linux* classiques (*Debian*, *Redhat*, …). D'autre part, *Keepalived* a très peu de dépendances avec des bibliothèques ou outils tiers, et les administrateurs système habitués aux installations « à la main » n'auront aucune difficulté à exécuter le célèbre triptyque `./configure ; make ; make install`.

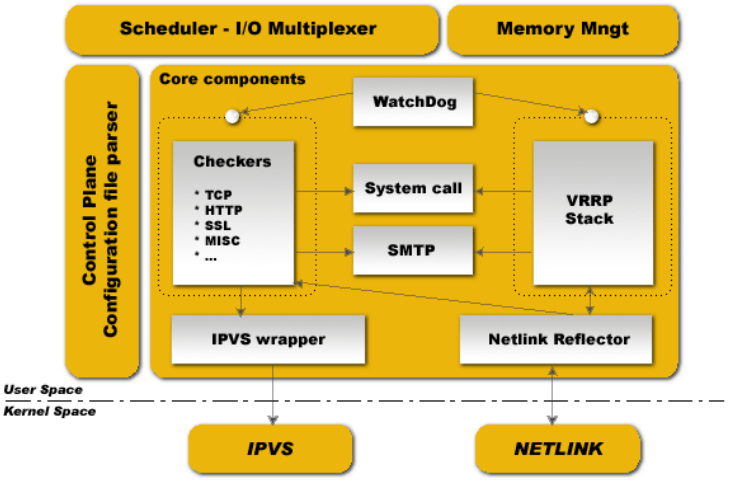

Figure 8: Schéma de principe *Keepalived*

[8] Expression empruntée à *Stéphane Borztmeyer* lors des JRES'2009 à Nantes

[9] Site *web* : *http://www.keepalived.org*

[10] Source : *Alexandre CASSEN - http://www.keepalived.org/software_design.html*

La configuration de l'outil est centralisée dans un seul fichier (`/etc/keepalived/keepalived.conf`) dont la compréhension et la syntaxe sont des plus élémentaires. Dans le même ordre d'idée, une fois démarré, *Keepalived* se présente sous la forme d'un seul processus avec deux activités filles (*child threads*) pour la gestion des fonctions *VRRP* et *HEALTCHECKER*.

Keepalived se démarque de la concurrence par sa simplicité et par son périmètre d'intervention bien cadré. En effet, plutôt que d'être une « boîte à tout faire » avec des outils dont les administrateurs n'auraient pas (ou peu) besoin, *Keepalived* se concentre sur les fonctions de base pour la construction et le déploiement d'un service réseau hautement disponible.

3.3 VRRP : failover IP

3.3.1 Principes

La première étape pour la construction d'un service réseau hautement disponible est de mettre en place l'adresse *IP* virtuelle qui sera présentée et utilisée par les clients. Cette adresse *IP* doit se trouver sur le répartiteur de charge qui aura la tâche de re-router les paquets qui lui seront adressés vers les serveurs réels. Afin de garantir la disponibilité du répartiteur de charge, donc de l'adresse *VIP*, il faut prévoir la redondance de cette fonction en déployant deux (ou plus) répartiteurs de charge. La difficulté dans ce contexte est de garantir que la *VIP* se trouvera toujours déclarée et associée au répartiteur opérationnel. *VRRP* (*Virtual Router Redundancy Protocol*) est un protocole standardisé par la *RFC3768*[11] qui a pour objectif de traiter cette problématique.

La *RFC* résume la fonction du protocole comme ceci :

> « *VRRP* implémente un protocole d'élection qui assigne dynamiquement la fonctionnalité de routeur virtuel à un des routeur *VRRP* parmi ceux présents sur le *LAN*. Le routeur *VRRP* qui contrôle la(les) adresse(s) *IP* associée(s) au routeur virtuel s'appelle le Maître (*Master*), et il re-route (redirige) les paquets envoyés à ces adresses *IP*. Le processus d'élection implémente une procédure de recouvrement de la fonctionnalité de routage en cas d'indisponibilité du Maître. Ceci permet à toutes les adresses *IP* du routeur virtuel sur le *LAN* d'être utilisées par les hôtes finaux comme passerelle par défaut. L'avantage apporté par *VRRP* est d'avoir une plus grande disponibilité de la route par défaut sans pour autant avoir à configurer des routes dynamiques ou des protocoles de découverte sur chaque hôte. »[12]

Une lecture rapide de l'énoncé du protocole laisse entendre que *VRRP* n'est utilisé que pour la haute disponibilité du service de routage d'un ensemble de routeurs. Cette utilisation était l'orientation principale du protocole lors de sa standardisation mais *VRRP* est totalement transposable à des serveurs (donc à des répartiteurs de charge), afin de rendre hautement disponible un adressage *IP* de serveurs.

Le principe de basculement de la *VIP* en cas de panne de l'instance « maître *VRRP* » est illustré par la Figure 9.

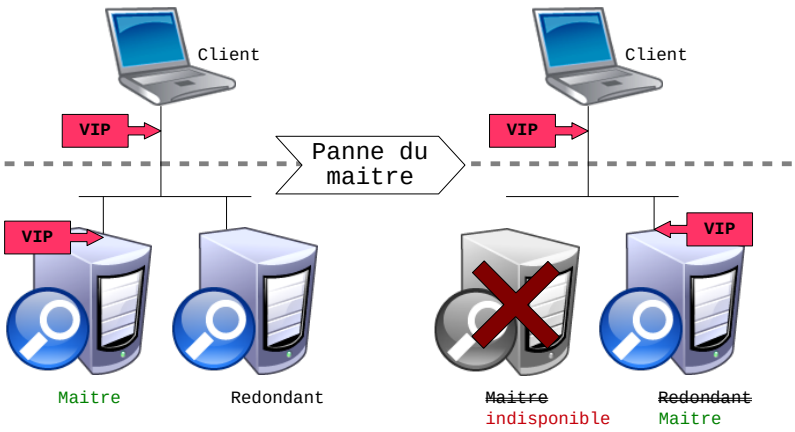

Figure 9: Panne du maître *VRRP*

En fonctionnement normal, c'est le répartiteur maître *VRRP* qui possède l'adresse *VIP* du service à fiabiliser. Le client ne connait que cette *VIP* pour accéder au service et le trafic utilisateur passe par ce répartiteur. En cas de panne du maître, un autre nœud « redondant » (*backup*) participant à *VRRP* va être élu maître par le protocole. Lors de cette élection, le répartiteur nouvellement maître s'assigne lui-même l'adresse *VIP* sur son interface permettant ainsi la continuité de l'adressage *IP*. Du côté du client, la bascule est transparente dans le sens où il ne doit rien changer à sa configuration réseau pour s'adapter à la panne du premier répartiteur.

[11] *http://tools.ietf.org/html/rfc3768*

[12] Traduction réalisée par mes soins

VRRP utilise des émissions de paquets *IP* (champ « protocole » de l'entête *IP = 112*) sur le groupe *multicast 224.0.0.18* afin que chaque nœud s'annonce (*VRID,* priorité) aux autres. La réception et l'analyse de ces paquets déterminent le processus d'élection d'un nouveau maître. Un nouveau maître « force » l'acheminement des paquets vers lui-même en annonçant sur le réseau *Ethernet* l'association [@*VIP*, @*MAC* nouveau maître] selon la technique de *Gratuitous ARP*[13].

3.3.2 Configuration et paramètres VRRP

Dans la partie *VRRP* de *Keepalived*, Alexandre CASSEN ne s'est pas limité à la seule implémentation de la *RFC* mais il a développé des « facilités » permettant d'interagir en fonction d'événements du protocole : ajouter/supprimer des routes *IP* en plus de la *VIP*, exécuter des « *scripts* maisons » sur changement d'état *VRRP*, … Sans entrer dans le détail de chaque paramètre de configuration *VRRP*, il est essentiel d'aborder les fondamentaux qui permettent la configuration minimale de nœuds coopérant selon ce protocole. La Table 1 résume et commente la configuration minimum d'un nœud *VRRP* avec le logiciel. L'extrait proposé respecte la syntaxe du fichier de configuration de *Keepalived* :`/etc/keepalived/keepalived.conf`.

```
vrrp_instance <STRING> {                  # Déclaration de l'instance VRRP (possibilité de plusieurs instances VRRP par serveur).
  virtual_router_id <INTEGER-1..255>      # Identificateur du routeur virtuel ( VRID). Doit être unique sur l'ensemble des nœuds.
                                          # VRRP gère au maximum 254 nœuds.
  priority          <INTEGER-1..254>      # Priorité utilisée lors de l'élection d'un nouveau maître. C'est la priorité la plus élevée qui gagne
                                          # l'élection. A priorités égales, c'est le serveur avec l'adresse IP la plus grande qui est élu maître.
  interface         <STRING>              # Interface d'écoute et d'échange des paquets multicast VRRP. La perte de cette interface ou un
                                          # défaut de réception de paquets VRRP émis par le maître démarre le processus d'élection.
  mcast_src_ip      <IP ADDRESS>          # [optionnel] Adresse IP utilisée pour l'échanges des paquets multicast VRRP.
  track_interface {                       # [optionnel] Interfaces supplémentaires de surveillance. La perte de l'une de ces interfaces
    <STRING>                              # démarre le processus d'élection.
    ... }                                 #
  virtual_ipaddress {                     # Liste des adresses IP virtuelles à gérer en tant que maître VRRP.
    <IP ADDRESS>/<MASK> dev <STRING>      #
    ... }                                 #
}                                         #
```

Table 1: Résumé de configuration et paramètres *VRRP*

La Figure 10 propose un exemple concret de configuration *VRRP* à deux nœuds. Seules 8 lignes de configuration dans le fichier `/etc/keepalived/keepalived.conf` suffisent à virtualiser et à rendre une adresse *IP* hautement disponible.

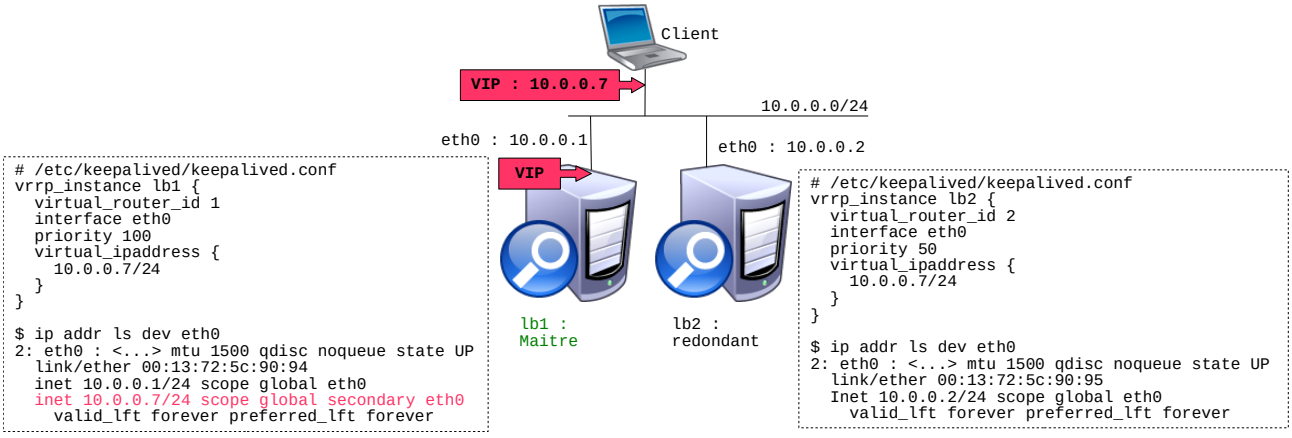

Figure 10: Exemple de configuration *VRRP* à deux nœuds

[13] http://en.wikipedia.org/wiki/Address_Resolution_Protocol#ARP_announcements, *http://wiki.wireshark.org/Gratuitous_ARP*

3.3.3 *Les autres choses « sympas » de VRRP dans Keepalived*

Dans sa version 1.2.0 (mai 2010) *Keepalived* implémente l'extension à *IPv6* du protocole *VRRP* (RFC5798[14]), faisant de cette solution une des rares à proposer nativement un mécanisme de tolérance aux pannes pour ces adresses.

Il faut également souligner les fonctionnalités additionnelles implémentées autour de *VRRP* qui peuvent s'avérer utiles :

- les notifications automatiques par *email* de changements d'état *VRRP ;*
- les `vrrp_script {...}` agissant sur les états *VRRP* en fonction du résultat d'exécution de *scripts* maison ;
- les `vrrp_sync_group {...}` synchronisant plusieurs instances *VRRP* du serveur en une seule ;
- les `notify_*` permettant l'exécution de *scripts* maison sur changement d'état *VRRP.*

3.4 *IPVS + HEALTCHECKERS* = Service réseau fiabilisé

3.4.1 *Principes*

Comme expliqué dans le chapitre précédent, *VRRP* permet de rendre insensible aux pannes l'adresse *IP* d'un service. La première moitié du problème étant résolue, il ne reste plus qu'à s'occuper de la fiabilisation du service lui-même.

Keepalived permet de déclarer un service réseau à fiabiliser en lui associant un ensemble de serveurs réels implémentant concrètement ce service. Cette fonction accomplit ni plus ni moins ce qu'un administrateur peut faire avec la commande `ipvsadm`. *Keepalived* a néanmoins l'avantage d'être plus clair et surtout persistant (au redémarrage du serveur) à l'aide du fichier de configuration.

La véritable avancée est de pouvoir attacher à la définition d'un serveur réel des procédures de tests qui évalueront la disponibilité et la capacité de traitement de celui-ci. *Keepalived* se charge d'exécuter à intervalles réguliers ces tests et il modifie, en fonction des résultats, la configuration *IPVS* pour la mettre en cohérence avec la disponibilité des serveurs réels. Ce fonctionnement est illustré par la Figure 11.

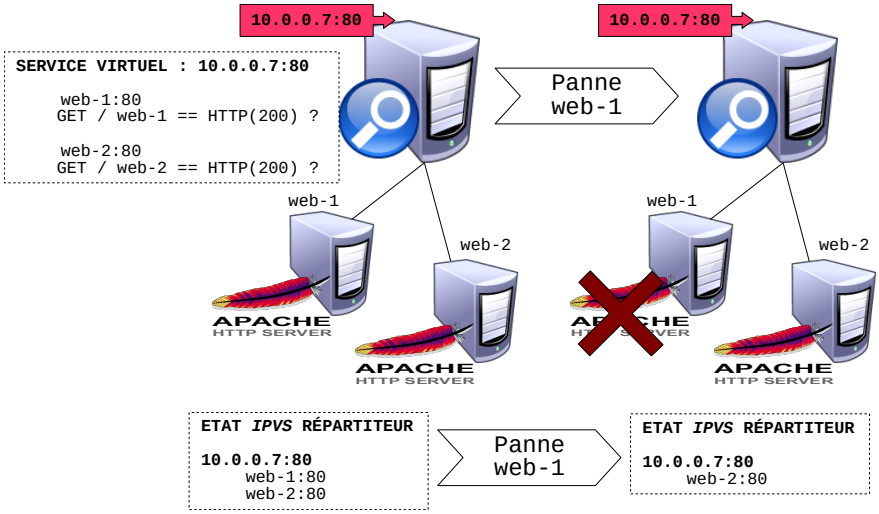

Figure 11: Virtualisation service et tests serveurs réels

Dans cet exemple, le service web virtualisé `10.0.0.7:80` est réparti sur deux serveurs réels : *web-1* et *web-2*. La disponibilité du service réel est testée par une requête *HTTP* sur les démons *Apache* de ces serveurs. Dans ce scénario, le simple arrêt du démon *Apache* de *web*-1 (alors que le serveur reste opérationnel) fera sortir ce serveur de la liste de ceux disponibles pour le traitement du trafic utilisateur.

3.4.2 *Configuration et paramètres IPVS + HEALTCHECKERS*

Pour cette partie, le fichier de configuration de *Keepalived* va intégrer dans une même déclaration la définition du service virtuel, son association à des serveurs réels et les procédures de tests. Les possibilités d'expression étant assez importantes la Table 2 résume le socle commun et les fondamentaux de configuration.

[14] *http://tools.ietf.org/html/rfc5798*

```
virtual_server  <IPADDR> <PORT> {      # Déclaration du service virtualisé : adresse IP et port du service à fiabiliser.
  protocol TCP|UDP                     # Protocole du service réseau à fiabiliser.
  lb_kind  NAT|DR|TUN                  # Choix du mode de redirection des paquets entre le répartiteur et les serveurs ( cf. chapitre 2.3.3).
  lb_algo  rr|wrr|lc|wlc|lblc|sh|dh    # Choix de l'algorithme de répartition entre les serveurs réels ( cf. chapitre 2.3.4).
  sorry_server  <IPADDR> <PORT>        # Définition d'un serveur de « dernier recours » en cas d'indisponibilité de tous les serveurs réels.
  real_server   <IPADDR> <PORT> {      # Définition du premier serveur réel, dont le service réel est joignable sur <IPADDR>:<PORT>.
    weight  <INT>                      # Affectation statique d'un poids pour ce serveur dans le cas d'utilisation d'un algorithme pondéré.
    HTTP_GET|SSL_GET|SMTP_CHECK|       # Procédure de test du service hébergé sur le serveur réel. Keepalived définit 4 procédures prêtes
    TCP_CHECK|MISC_CHECK  {            # à l'emploi et une libre. HTTP_GET et SSL_GET permettent d'effectuer des requêtes (sécurisées
                                       # ou non) HTTP. SMTP_CHECK teste la connexion à un serveur de messagerie. TCP_CHECK
                                       # vérifie la disponibilité d'un port TCP. Enfin, MISC_CHECK est une procédure de test libre définie
                                       # par l'exécution d'un programme ou d'un script maison.
      ...                              # Paramètres spécifiques à la procédure de test (cf. documentation de Keepalived)
    }                                  #
  }                                    #
  real_server  <IPADDR> <PORT> {       # Définition du second serveur réel.
    ...   }                            # Paramètres de configuration du second serveur.
}                                      #
```

Table 2: Résumé de configuration et paramètres *IPVS + HEALTCHECKERS*

3.4.3 Les autres choses « cool » avec IPVS et HEALTCHECKERS dans Keepalived

La configuration de la partie *IPVS + HEALTCHECKERS* nécessite d'être approfondie afin d'exploiter toute la puissance de *Keepalived*. On peut néanmoins citer quelques fonctionnalités intéressantes à mettre en œuvre :

- `fwmark` : plutôt que d'être défini par le couple *IP:PORT*, le service virtuel peut être spécifié par le marquage interne à *Linux* de certains paquets *IP* (action *MARK* dans la table *mangle* via les *IPTables*). Ce concept permet de sélectionner avec d'autres critères (que simplement *IP:PORT*) le service à fiabiliser ;
- `notify_up` et `notify_down` : permettent l'exécution de *scripts* maison sur le résultat des tests de serveurs ;
- `weigth` : poids associé aux serveurs qui peut être utilisé par certain algorithmes de répartition afin d'orienter plus ou moins de trafic vers ces serveurs ;
- `MISC_CHECK` et le code de retour du *script* permettent de re-programmer dynamiquement le poids associé au serveur pour lequel *Keepalived* évaluera le test.

La version 1.2.2 (janvier 2011) supporte nativement *IPv6* dans la partie *HEALTCHECKERS* et dans le pilotage *d'IPVS.*

3.4.4 Les Bee Gees chanteraient aujourd'hui « Stayin' Keepalived »

Keepalived est une solution totalement aboutie qui complète parfaitement la fonctionnalité *IPVS* de *Linux*. Avec le développement des modules *VRRP* et *HEALTCHECKERS*, elle apporte la « glu » manquante pour le déploiement de services réseau hautement disponibles et fiabilisés à tous les points de pannes potentielles. Elle a su, au fil du temps, s'adapter aux technologies émergentes notamment en suivant de près les fonctionnalités du noyau *Linux* et en implémentant le support *d'IPv6* dans tous ses modules.

Keepalived est surprenant de simplicité et d'efficacité, et beaucoup d'architectures devraient trouver dans cet outil des solutions d'implémentation de leur haute disponibilité.

Une dernière chose toute simple : merci *Keepalived*, merci Alexandre CASSEN.

4 Exemples

4.1 La quadrature du répartiteur : { 1 + 1 + 1 + 1 = 2 }

L'énoncé du problème est simple : pour fiabiliser un service réseau (site *web* par exemple) opéré par un serveur (1 = 1) il est nécessaire de redonder ce serveur par un second (1+1 = 2). Afin de rendre accessible le service aux utilisateurs, il faut mettre en frontal un répartiteur de charge (1+1+1=3). Ce répartiteur devenant le point unique de panne (*SPOF*), il est impératif de le redonder lui-aussi par un doublon (1+1+1+1=4).

Ce scénario où il faut quatre éléments physiques pour redonder un seul service est assez « classique ». Certes, la mutualisation du répartiteur redondé et sa réutilisation pour fiabiliser plusieurs services permet de factoriser l'équation. Mais une autre approche peut être exploitée en fusionnant sur un même serveur physique les parties « services applicatif » et « répartiteur de charge », ce qui fait passer l'architecture de quatre éléments actifs à seulement deux. La Figure 12 illustre ces différentes approches.

Le déploiement du logiciel *Keepalived* sur les nœuds opérant le service permet la construction de cette architecture fiabilisée à seulement deux équipement actifs. Ce concept de fusion des fonctionnalités est très rarement explicité par les éditeurs de solutions et encore moins utilisé par les administrateurs. Il faut dire que le résultat obtenu n'est pas très « didactique » et qu'un certain mélange des genres peut perturber les équipes qui se lancent dans la haute disponibilité réseau.

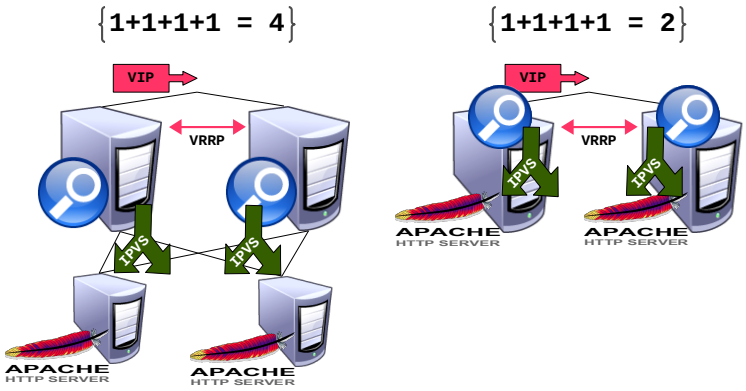

Figure 12: Architecture à quatre et deux éléments actifs

Au delà de la simple économie de deux serveurs au lieu de quatre, ce modèle apporte d'autres avantages non négligeables :

- la paire de serveurs peut fonctionner en mode « actif-actif », là où dans l'utilisation de répartiteurs séparés le seul mode possible est « actif-passif » : les ressources matérielles sont donc mieux utilisées ;
- le répartiteur est dédié aux serveurs qui opèrent le service réseau. N'étant pas mutualisé avec d'autres projets sa configuration et son fonctionnement sont simplifiés ;
- les performances du répartiteur sont nativement liées et dédiées au service à fiabiliser et elles ne sont pas affectées par d'autres trafics extérieurs ;
- le modèle est extensible à plus de deux nœuds par simple duplication.

Cette architecture est systématiquement déployée par l'équipe réseau du CIRIL pour le développement de services hautement disponibles. Les chapitres suivants présentent deux mises en œuvre récentes qui implémentent ce concept.

4.2 Portail des services réseau Lothaire: *http://reseau.ciril.fr*

Le portail des services Lothaire est la centralisation de tous les outils *web* développés par le CIRIL pour la gestion des réseaux et de toute la documentation des services proposés. Les serveurs hébergeant ce portail implémentent également des services réseau de l'entité CIRIL qui doivent être visibles de l'extérieur, comme par exemple le service *radius* utilisé dans le cadre *d'eduroam*. Naturellement la disponibilité de ce portail est sensible car son arrêt empêche les administrateurs des établissements lorrains d'accéder aux outils de gestion des réseaux. De ce fait, l'équipe du CIRIL a entrepris courant 2011 la fiabilisation de l'ensemble des services proposés par ce portail, à savoir :

- le service *web* en *HTTP* et *HTTPS*, accessible en *IPv4* et *IPv6 ;*
- le service *radius* utilisé par *eduroam*.

La mise en œuvre suit le concept expliqué précédemment et seulement deux serveurs (*TIC* et *TAC*) sont utilisés dans l'architecture de ce portail. La Figure 13 résume la

```
# /etc/keepalived/keepalived.conf
vrrp_instance tic(tac) {
 virtual_router_id 1(2)
 interface bond0
 priority 200(100)
 virtual_ipaddress {
  193.50.27.132/28
  2001:660:4503:203::132/64
 }
}

virtual_server VIPv4 [1812,1813,1814] {
 lb_algo rr ; lb_kind DR ; protocol UDP
 real_server tic 1812 {
  MISC_CHECK /opt/test-radius.sh
 }
 real_server tac 1812 {
  MISC_CHECK /opt/test-radius.sh
 }
}

virtual_server VIPv4+VIPv6 [80,443] {
 lb_algo rr ; lb_kind DR ; protocol TCP
 real_server tic 80 {
   HTTP_GET { « GET / » == 200 }
 }
 real_server tac 80 {
   HTTP_GET { « GET / » == 200 }
 }
}
```

Figure 13: Configurations TIC/TAC

configuration des serveurs du portail. La syntaxe du fichier y est volontairement fausse, tronquée et allégée pour ne faire apparaître que les points importants du déploiement. On notera plus particulièrement :

- la déclaration des *VIP* en *IPV4* et *IPv6 ;*
- la déclaration du service *radius* en *IPv4* uniquement sur les ports *UDP* 1812, 1813 et 1814 ;
- la déclaration du service web en *IPv4* et *IPv6* sur les ports *TCP* 80 et 443 ;
- la procédure standard *HTTP_GET* pour le test de serveurs *web* réel ;
- la procédure de test « maison » *MISC_CHECK* pour la validation des serveurs *radius*.

Les derniers points de finalisation de ce projet sont toujours en cours. Il faut néanmoins préciser que les difficultés rencontrées pour ce déploiement n'étaient pas situées du côté du réseau ou de *Keepalived*, mais plutôt dans des sujets tels que « comment fiabiliser la base données MySQL ? », « comment monter un système de fichier réparti ? », …

4.3 Service de filtrage web mutualisé : *iWash*

L'équipe réseau du CIRIL a développé courant 2010/2011 un service de filtrage *web*, *iWash*[15], destiné à la protection des réseaux du portail captif mutualisé *YaCaP*[16]. Sans entrer dans les détails techniques d'implémentation, il est intéressant d'énumérer les caractéristiques et les particularités de ce service quant à sa haute disponibilité.

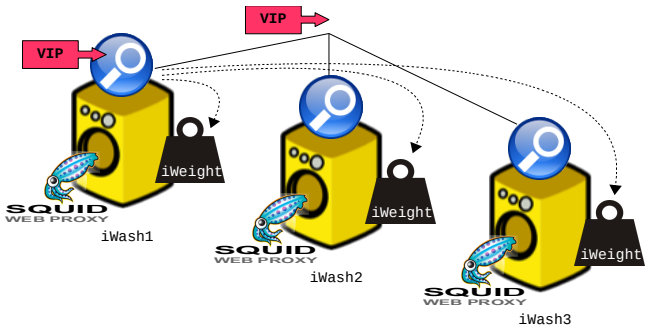

Figure 14: *iWeight* - calcul de poids dynamique *IPVS*

Des analyses et une évaluation de l'utilisation ont montré qu'il fallait deux serveurs pour traiter efficacement la totalité du trafic utilisateur. La mise en œuvre compte donc 3 serveurs physiques assurant un traitement non-dégradé en cas de panne. La fusion des fonctions de répartiteur et de *proxy SQUID* permet d'avoir un ensemble coopérant selon le mode « actif-actif-actif ». L'utilisation des ressources est optimale, tant pour le traitement de la charge que pour la haute disponibilité.

L'autre particularité de ce déploiement est l'utilisation de l'algorithme de répartition de charge *Weighted Round Robin* (***wrr***). Un développement « maison » (*iWeight*) permet à chaque serveur de calculer son propre poids en fonction de métriques locales (charge CPU, charge SQUID, …). Ce poids est ensuite « remonté » à *IPVS* via la procédure de test *MISC_CHECK* de *Keepalived*, ce qui permet au serveur maître *VRRP* (celui qui détient la *VIP*) d'appliquer une répartition de charge avec pondération dynamique « temps réel » calculée sur la base des capacités de traitement de chaque serveur. On notera également la synchronisation des tables de connexions réseaux (*conntrack*) des serveurs entre eux avec l'utilisation de la suite *conntrack-tools*[17]. Cette synchronisation garantit la reprise des connexions actives, sans « casser » les sessions protocolaires, lors d'un basculement de maître *VRRP* en cas de panne.

5 Conclusion et notes personnelles

Cet article vient de faire le point sur les techniques et les outils de haute disponibilité réseau sous *Linux*, notamment avec l'utilisation de *Keepalived*. Cette solution centralise l'ensemble des fonctionnalités à mettre œuvre pour fiabiliser les couches réseau d'un service. *Keepalived* est utilisé depuis plus de 7 ans dans les infrastructures réseaux et systèmes déployées par l'équipe du CIRIL et nous pouvons assurer qu'il a permis des avancées majeures en terme de fiabilisation et de nouveaux services. Le développement de nouveaux services fiabilisés passent automatiquement par l'utilisation de *Keepalived* dont nous ne pouvons plus nous passer : « l'essayer, c'est l'adopter ! ».

Seules les bases ont été traitées dans cet article, aussi, les concepts présentés nécessitent, sans doute, d'être approfondis par le lecteur. La bonne approche est d'essayer dans un premier temps chaque outil indépendamment les uns des autres (*VRRP* tout seul, configuration *IPVS* « à la main » via `ipvsadm`, …). Cette étude unitaire permettra une bonne compréhension de chaque outil, après quoi l'assemblage centralisé par *Keepalived* apparaitra comme une évidence, aussi facile qu'un jeu d'enfants.

Très bonne haute disponibilité à tous ! *and* ***Keep*** *it* ***alived***.

[15] *http://reseau.ciril.fr/doc/Services/FiltrageWeb*

[16] *http://reseau.ciril.fr/doc/Services/PortailCaptif*

[17] *http://conntrack-tools.netfilter.org*