Du bon usage de la virtualisation de serveurs

Pascal Mouret
Université de la Méditerranée – DOSI Campus Luminy
163, avenue de Luminy – 13288 Marseille cedex 9

Résumé

La virtualisation de serveurs apporte une souplesse indéniable, qui ouvre de nouvelles perspectives. Mais cette souplesse s'accompagne de nouvelles exigences.

Après une brève introduction rappelant les différentes techniques de virtualisation et leurs atouts respectifs, cet article s'articule autour de deux parties principales.

Dans la première, à partir du constat que la possibilité de monter facilement des serveurs change la façon dont on doit les utiliser, et à la lumière de l'expérience de notre université, nous nous intéresserons à déterminer comment répartir le plus efficacement possible les services / applications dans ces machines, dans le respect des différentes exigences liées à l'exploitation des serveurs.

Dans la seconde, nous aborderons la question de la gestion d'un parc de serveurs. Nous présenterons les outils d'administration centralisée, au travers de l'exemple de Puppet. Nous exposerons ainsi le principe et l'utilisation de ces outils et leur apport, décisif, dans la gestion d'un grand nombre de serveurs. Puis nous montrerons comment maintenir à jour et en cohérence la configuration d'un parc de serveurs avec Puppet.

Enfin, nous compléterons ce retour d'expérience par quelques questions / problématiques induites (supervision, dimensionnement des serveurs physiques, séparation puissance de calcul - stockage, ...).

Mots clefs

Virtualisation, serveurs, administration centralisée, Puppet, OpenVZ, Linux, retour d'expérience

1 Introduction

La virtualisation de serveurs est actuellement de plus en plus utilisée dans nos établissements. Elle apporte une souplesse indéniable, qui ouvre de nouvelles perspectives, mais en même temps, elle induit de nouvelles questions, de nouvelles exigences qu'il est important de bien considérer. En effet, elle change grandement notre manière de fonctionner selon plusieurs axes, notamment :

- Le temps : alors que mettre en place une nouvelle machine physique requiert du temps, on est plus dans l'immédiateté avec les machines virtuelles. La création se fait en quelques minutes. Leur cycle de vie est également beaucoup plus court.
- L'argent : sauf à atteindre les capacités maximum des serveurs physiques sur lesquels ils sont installés, la mise en place d'un serveur virtuel ne nécessite pas de nouvel investissement.
- La disponibilité : on peut maintenir le service très facilement en migrant une machine virtuelle d'un serveur physique à un autre.

Logiquement, il en découle de nouvelles façons d'appréhender une infrastructure de serveurs.

A partir de l'expérience acquise ces dernières années par l'exploitation d'un nombre sans cesse croissant de serveurs virtuels dans notre université, et notamment à la lumière de l'expérience du campus Luminy, cet article tente de répondre à ces questions, ou tout au moins de proposer quelques pistes de réflexions. Après avoir rappelé les différentes techniques de virtualisation et leurs atouts respectifs, nous aborderons les différentes étapes d'une migration de machines physiques vers des machines virtuelles :

- Comment répartir les services/applications dans les machines virtuelles pour tirer le meilleur parti de cette technologie ?

- Comment gérer efficacement un grand nombre de serveurs ?
- Comment optimiser et garantir le fonctionnement de la nouvelle infrastructure ?

2 La virtualisation de serveurs

La virtualisation est multiforme et se décline dans un grand nombre de domaines. Elle correspond, à la base, à la faculté de recréer à l'intérieur d'une unité fonctionnelle donnée d'autres unités fonctionnelles équivalentes. C'est le cas de la virtualisation de serveurs (virtualisation de systèmes d'exploitations), ou de la virtualisation applicative (robots de listes Sympa, sites virtuels Apache, ...). Par extension, elle désigne également la faculté de mettre à disposition d'un système une ressource distante, qui, vue de l'utilisateur, apparaitra comme locale. On retrouve ce principe dans la virtualisation d'applications ou la virtualisation de disques. Combiné au premier point, il donne la virtualisation de postes de travail.

Nous nous intéressons ici à la virtualisation de systèmes d'exploitation (plus précisément, à la virtualisation de serveurs), c'est à dire à l'ensemble des techniques matérielles et/ou logicielles qui permettent de faire fonctionner sur un seul serveur physique plusieurs SE (Système d'Exploitation), séparément les uns des autres, comme s'ils fonctionnaient sur des machines physiques distinctes [1]. Ces dernières sont appelées les (systèmes) *hôtes* et les SE qu'elles accueillent sont naturellement les systèmes *invités*. Selon les technologies, on parle indifféremment de VM (Virtual Machine) ou VE (Virtual Environment).

De façon schématique, on peut distinguer deux grandes catégories de techniques de virtualisation :

1. Les isolateurs de contexte. Cette technique est très associée au monde Unix. Dans ce cas-là, les machines virtuelles partagent un même noyau. Les processus tournent sur l'hôte, dans des environnements distincts appelés contextes. Les principaux avantages de cette technique sont sa légèreté, une optimisation des ressources disques et mémoire et sa rapidité (on tourne quasiment à la vitesse d'une machine réelle). Ses inconvénients principaux sont qu'il faut que les machines virtuelles soient homogènes (supportent le même noyau) et que la séparation des machines n'est pas totale. Dans cette catégorie, on trouve, entre autres, Linux-Vserver, OpenVZ, LXC.
2. Les hyperviseurs[1]. Ici, plusieurs SE distincts fonctionnent sur une même machine hôte, de manière étanche entre eux. L'hyperviseur présente une couche d'accès matériel, à partir de laquelle on peut faire fonctionner un SE comme sur une machine physique. On distingue deux grands familles :

    - Les hyperviseurs de type 2. L'hyperviseur s'exécute ici à l'intérieur du SE de la machine hôte. On perd en performances du fait de l'empilement des deux SE (hôte et invité). Par contre, c'est une solution très facile à mettre en œuvre. De fait, elle est surtout utilisée au niveau du poste de travail. On trouve dans cette catégorie notamment Microsoft VirtualPC/VirtualServer, Oracle VirtualBox, Parallels Desktop, VMWare Player, et bien d'autres.

    - Les hyperviseurs de type 1. Dans ce cas-là, l'hyperviseur est lui-même un mini-système d'exploitation optimisé pour gérer les accès des machines virtuelles au matériel du système hôte. On y gagne énormément en performances. Il s'agissait initialement de solutions assez onéreuses, mais des versions gratuites commencent à sortir. Les principaux produits de ce type sont Microsoft HyperV, Citrix XenServer et VMWare ESXi.

    A noter que, dans certains cas, pour optimiser encore l'accès du SE invité aux ressources matérielles de l'hôte, le SE doit être spécialement adapté pour fonctionner sur l'hyperviseur utilisé. On parle alors de « paravirtualisation ». On obtient d'excellentes performances mais on ne peut pas forcément faire tourner tous les SE que l'on souhaiterait. HyperV notamment utilise cette technique.

Pour plus de détails sur les différentes solutions de virtualisation, cf [2] et [3].
Sur le campus de Luminy, étant majoritairement dans un environnement homogène Linux, et pour des raisons de performances, de simplicité, de coût, mais aussi de principe (à fonctionnalités / ergonomie équivalentes, nous privilégions les logiciels libres), nous avons fait le choix d'OpenVZ (au travers de Proxmox Virtual Environment). Dans la suite de cet article, nous nous appuierons donc sur OpenVZ, mais les réflexions exposées sont facilement généralisables.

[1] Nous avons choisi de reprendre ici la terminologie adoptée sur Wikipédia, mais force est de constater qu'elle n'est pas unanimement partagée. Néanmoins, l'important est plus dans les concepts qu'elle recouvre que dans la terminologie elle-même.

3 Mise en place de serveurs virtuels

Les avantages que l'on peut attendre de la virtualisation sont notamment : une meilleure utilisation des ressources (les serveurs physiques sont généralement sous-utilisés en termes de CPU et de mémoire) ; une plus grande souplesse d'utilisation (facilité de création/destruction, facilité de maquettage par clonage simple) ; une meilleure sécurité (isolation des applications), et une plus grande stabilité.

Tous ces éléments nous ont amenés à penser qu'il était plus intéressant de migrer toute notre infrastructure depuis un parc de serveurs physiques vers un ensemble de serveurs virtuels. Ayant maintenant la possibilité d'utiliser un nombre assez « élastique » de serveurs (virtuels), la question s'est donc posée de déterminer comment répartir au mieux ces services.

La sécurité apporte clairement des éléments de réponse. A priori, on a tout intérêt à séparer des services qui n'ont rien à voir les uns avec les autres. Cela évite que des problèmes sur l'un impactent l'autre. Par exemple, on peut séparer le web et le DNS, l'annuaire ldap, etc ... De même, il parait pertinent de séparer des applications de niveaux différents (en terme de Disponibilité Intégrité Confidentialité, DIC), même basées sur un même service. Par exemple, le serveur web de pages personnelles, l'interface web d'administration du serveur ldap ou le serveur web institutionnel ne devraient pas avoir d'interactions. De fait, on répartira un même service sur plusieurs machines distinctes pour prendre en compte ces niveaux de sécurité différents. Dans notre cas, nous avons distingué 4 niveaux pour les applications web : les outils d'administration à destination du service, ceux à destination des utilisateurs, les sites web personnels et sites de labo, et les sites institutionnels.

De façon symétrique, plusieurs serveurs destinés à un même public peuvent être regroupés en un seul un peu plus gros. Si ce gros serveur venait à nécessiter plus de ressources que disponible sur la machine hôte, il serait facile de le migrer ailleurs. Ainsi, alors que, dans un premier temps, nous avions plutôt opéré une division par application, avec par exemple au niveau du web encore, un serveur pour GLPI (gestion de parc), un serveur pour un CMS (site institutionnel), un pour Netdisco (découverte réseau), etc..., il apparaît maintenant clairement que toutes les applications d'administration pour le service par exemple (Netdisco et GLPI ici) ont tout intérêt à être regroupées.

Nous nous sommes ensuite plus particulièrement intéressés au cas du service web et du service de bases de données. Les deux sont intimement liés dans bon nombre d'applications et il nous semblait logique qu'ils soient regroupés sur un même serveur. Cependant, toutes les applications web ne nécessitent pas forcément une base de données. Celles qui en utilisent une ne la sollicitent pas tout le temps ou disposent parfois de caches. Les séparer permettrait qu'une compromission ou un dysfonctionnement de l'un n'ait pas d'impact sur l'autre. Et surtout, en les séparant, nous pouvions mettre en place de la haute disponibilité entre les deux. C'est donc finalement ce qui a été retenu. Par contre, pour quelques services peu consommateurs de ressources (dhcp ou ntp), nous avons choisi de ne pas dédier une machine virtuelle spécifique, et de simplement les héberger sur les hôtes, d'autant qu'ils ne présentent pas (ou peu) de problèmes de sécurité.

Enfin, nous avons dupliqué tous les serveurs critiques et mis en place une solution de haute disponibilité (HAProxy). C'était une volonté antérieure, mais c'est la migration vers les serveurs virtuels qui l'a vraiment rendue possible.

Quoi qu'il en soit, au final, nous arrivons à une certaine répartition, qui semble stable aujourd'hui. Il est évident que, selon le contexte, la conclusion ne sera pas forcément la même, mais nous pouvons énoncer un certain nombre de bonnes pratiques qui devraient avoir du sens pour tous :

- Des applications de niveaux de sécurité différents ou qui n'ont aucun service en commun seront sur des serveurs séparés ;
- Deux services indépendants ou nécessitant une liaison en haute disponibilité entre eux seront également sur des serveurs séparés ;
- Un même service utilisé par plusieurs applications différentes et s'adressant à une même catégorie d'utilisateurs sera mis sur un serveur dédié.

Au niveau du campus de Luminy, nous avons donc engagé cette démarche il y a un an. Des 71 serveurs initiaux répartis sur tout le campus (gérés par différentes équipes), nous sommes arrivés à 25 serveurs, dont 8 dédiés à l'infrastructure virtuelle (2 répartiteurs

de charges en *failover* l'un de l'autre[2] et 6 hôtes OpenVZ). Sur ces hôtes, nous faisons tourner 33 serveurs virtuels (certains services étant donc dupliqués, comme vu plus haut). La migration n'est pas encore terminée, mais nous approchons du but.

Toutefois, il s'agit maintenant de correctement les administrer.

4 Administration centralisée

A la création d'une machine virtuelle, on a toujours un certain nombre de tâches de "préparation" à faire ainsi que des tâches régulières de maintenance, ajout de fonctionnalités... Le nombre de machines ayant tendance à augmenter avec la virtualisation, il est impératif de pouvoir automatiser et harmoniser la gestion de ces machines pour gagner du temps, homogénéiser les configurations et être sûr que toutes les machines sont à jour. Après diverses confrontations d'idées, nous nous sommes intéressés aux outils d'administration centralisée [4]. Les quatre outils les plus cités dans ce domaine sont : CFEngine, Puppet, Chef et Bcfg2. CFEngine est le précurseur. Sa première version date de 1993. Longtemps le plus connu du genre, il en est aujourd'hui à sa troisème version majeure. Puppet, créé en 2005, puis, Chef, en 2009, découlent tous les deux de CFEngine. Bcfg2, quant à lui, a été initié vers 2004.

Nous allons ici décrire Puppet, dont le modèle nous est apparu très intéressant.

4.1 Principe général

Puppet est donc un outil d'administration centralisée de parc de machines. Il permet de gérer les configurations d'un ensemble de machines à partir de spécifications communes. Il se compose d'un serveur, appelé *master*, sur lequel sont définis les états cibles des différentes machines, et d'un client (*agent*) installé sur chaque machine à contrôler. Les échanges entre l'*agent* et le *master* se font en XML-RPC (pour les anciennes versions de Puppet), ou avec une API REST spécifique développée pour Puppet (pour les versions plus récentes). Le tout transite via une connexion HTTPS. Un échange de clé SSL se produit lors de la première connexion de l'*agent* au *master*. Il faut que le *master* signe explicitement la clé de l'*agent* pour que la communication soit autorisée et que l'*agent* puisse récupérer sa configuration. Cela permet ainsi d'éviter la connexion au *master* de machines non autorisées et la « fuite de données » que cela pourrait entraîner. Puppet est multiplateforme. Il est écrit en Ruby, et sa syntaxe s'en inspire. Sa dernière version est la 2.7.

Le point important est que Puppet définit non pas une suite d'instructions à appliquer, mais un « état » à atteindre pour chacune des *ressources* que doit avoir (ou ne pas avoir) la machine cible. Par ressource, on entend tout objet du système (un fichier, un paquetage, un service, etc...) sur lesquels on peut agir. L'état à atteindre peut être, pour un fichier, d'être présent/absent ou d'avoir tel ou tel contenu, pour un paquetage, d'être installé ou pas, éventuellement dans une version spécifique, etc...

A intervalles réguliers, l'*agent* se connecte au *master*, récupère le *catalogue* de ressources qui s'applique à lui et vérifie qu'il est bien dans l'état décrit. Si ce n'est pas le cas, il exécute toutes les actions nécessaires pour atteindre cet état. De fait, un *agent* peut appliquer un même catalogue autant de fois qu'il le souhaite : son état sera toujours le même (c'est à dire celui décrit dans le catalogue). Cette stabilité dans l'exécution est un des principes majeurs dans Puppet, et est appelée l'*idempotence*.

Enfin, l'état d'une ressource est exprimé de manière indépendante par rapport au système. De façon générale, on n'a pas à décrire le moyen d'arriver à l'état voulu. Les ressources incluent la notion de *provider*. Le *provider* (ou *prestataire*) est dépendant de la plateforme sur laquelle il tourne et représente l'ensemble des outils permettant d'atteindre l'état voulu de la ressource à laquelle il est associé.

4.2 Définition d'un catalogue de ressources

Toutes les configurations de Puppet et des machines à gérer sont disposées dans l'arborescence de base de Puppet (*/etc/puppet* sous Unix/Linux). Les ressources sont déclarées dans un *manifest*. Le manifest de base, qui s'applique à tous est *site.pp* et se trouve dans le répertoire *manifests* (*/etc/puppet/manifests*).

[2] Il s'agit de deux serveurs HAProxy en mode actif/passif (ou maître/esclave). Pour chaque service, ils partagent une même adresse IP. Seul le maître répond à cette adresse et relaie les requêtes vers les machines du pool de serveurs correspondant selon un algorithme de répartition donné. Les deux serveurs ont également un processus *heartbeat* qui permet à chacun de surveiller l'état de l'autre par différents moyens (réseau dédié, lien série) et assure le *failover*. Lorsque le maître ne répond plus aux messages hearbeat de l'esclave, ce dernier devient maître et assure la répartition du trafic sur les adresse IP communes.

Une déclaration de ressource s'exprime de la manière suivante :

```
file { 'cron': path => '/etc/cron.d/puppet', ensure => present, source => 'puppet:///puppet.cron' }

package {ntp : ensure => present }
```

On a ici deux ressources, correspondant respectivement à un fichier et à un paquetage. L'ordre de déclaration n'a aucune signification, *path* correspond au fichier cible et *source* au fichier de référence. L'url *puppet:///puppet.cron* désigne le fichier *puppet.cron* situé sur le *master* dans le répertoire *files*. Si le fichier cible existe, et afin de vérifier si les deux fichiers sont identiques ou non, Puppet compare simplement leurs signatures md5. Si il n'existe pas, il est copié depuis le serveur vers l'*agent*. On peut également générer un fichier au travers de templates et de variables. Dans ce cas, on utiliserait l'attribut *content* (*content => template('nom du template')*) au lieu de source. Le *template* est quant à lui un fichier ERB (Embedded Ruby, le mécanisme de templates natif de Ruby) qui contient un certain nombre de balises provoquant l'affichage d'une variable ou l'exécution d'un code Ruby arbitraire. Il est stocké par défaut dans le répertoire *templates*.
L'attribut *ensure* indique ce que l'on exige de ces ressources, à savoir ici que la ressource *ntp* soit présente/installée. Dans la déclaration de la ressource package, *ntp* correspond au nom du paquetage. Si celui-ci n'est pas déjà installé, le *provider* entre en jeu : Par exemple, sur une Linux Debian/Ubuntu, il exécutera un *apt-get install ntp*, sur une Linux Red Hat/Fedora/..., il utilisera *yum install ntp*, etc... Depuis la version 2.6, en plus d'Unix et Mac OS X, Puppet fonctionne également sous Windows. La plupart des types de ressources possèdent également un *provider* pour Windows. Ainsi, s'il existait un fichier *ntp.msi* (pour peu qu'on rajoute l'attribut indiquant où le trouver), Puppet l'installerait aussi.

Souvent, une même ressource a des noms différents selon les plateformes. Ainsi, par exemple pour un client Samba, le paquetage à installer est *smbclient* sous Debian mais *samba-client* sous Red Hat. Pour permettre de gérer cela, Puppet dispose sur chaque machine d'un petit utilitaire nommé *facter*, dont le rôle est de collecter des « faits », c'est à dire des informations sur la machine sur laquelle il tourne. Ces faits sont communiqués au *master* par l'*agent*, et sont disponibles dans les *manifests* au travers de variables. On peut connaitre ainsi l'adresse IP, le nom d'hôte, le type de noyau et sa version, la version de Puppet, le fait que la machine soit virtuelle ou non, la distribution utilisée, et bien d'autres choses ... et faire des déclarations conditionnelles. Ainsi, nous pouvons modifier notre exemple en conséquence :

```
if $operatingsystem == 'debian' { $sambaclient='samba-client' }
elsif $operatingsystem == 'redhat' { $sambaclient='smbclient' }
else { fail("Je ne connais pas cet os") }

package {$sambaclient : ensure => present }
```

Il existe un grand nombre de types de ressources [5]. Parmi les plus utiles, on peut citer, outre les paquetages, les fichiers (*file*), les services (*service*), les utilisateurs (*user*), les tâches planifiées (*cron*), ..., sans oublier le « couteau-suisse » *exec* permettant de lancer n'importe quelle commande, sous réserve de la bonne exécution d'une autre commande. On peut également créer ses propres ressources, en Ruby.

A l'instar de l'utilisation de la ressource *exec*, il est impératif de respecter le principe d'idempotence.

Il y a des cas où une ressource n'a de sens que lorsqu'une autre est disponible. Par exemple, il est inutile d'essayer de monter une arborescence via NFS si le paquetage *nfs* n'a pas été chargé. Pour exprimer cela, Puppet propose deux attributs disponibles pour toutes les ressources : *before* et *require*.

```
require => Package['nfs']
```

Dans ce cas, la ressource, dont la définition contient cet attribut, doit être évaluée après la ressource *nfs* de type package. Si la ressource requise ne peut pas être passée dans l'état voulu, celle qui la requiert ne pourra pas être (dés)installée non plus. A noter un élément important de la syntaxe Puppet : quand on déclare une ressource (*package {'nfs' : ... }* par exemple), le type de ressource doit être écrit en minuscules. Quand on y fait référence, le type de ressource doit toujours avoir l'initiale en majuscules (*require => Package['nfs']* par exemple) ! Le mot-clé *before* exprime la relation symétrique à *require*. Au choix, on indique soit avec *require* qu'une ressource en nécessite une autre, soit avec *before* qu'une ressource passe avant une autre. Ce sont deux façons équivalentes de dire la même chose. S'il y a plusieurs dépendances, on peut utiliser un tableau (une liste de valeurs entre crochets droits) :

```
require => [ Package['nfs'], Service['portmapper'] ]
```

Egalement, *notify* et *subscribe* ont la même signification que respectivement *before* et *require*, avec en plus une notion de « rafraichissement » (redémarrage d'un service, ré-exécution d'un code, ...). Ainsi :

```
notify => Service['ssh']
```

pourra être mis dans la ressource correspondant au fichier de configuration de *ssh*. Une fois que celui-ci aura été mis à jour, c'est le service ssh qui sera mis à jour si besoin est (comme *before => Service['ssh']*), et qui sera de surcroit redémarré (en l'occurence pour prendre en compte le nouveau fichier de configuration). On peut exprimer ainsi un arbre de dépendance complet des ressources d'une machine.

Pour simplifier la lecture (et l'écriture) d'un *manifest*, Puppet propose une organisation sous forme de *classes*. On définit une classe en y déclarant l'ensemble des ressources qu'elle contient. Par exemple :

```
class sambaclient { # on met tout le code vu plus haut }
```

Ensuite, si, n'importe où dans le code, on écrit *class {sambaclient : }*, cela revient à déclarer l'ensemble des ressources contenues dans la classe. Pour les utiliser plus simplement, Puppet propose un mécanisme d'« auto-chargement » des classes, qui les rend automatiquement disponibles dans tous les *manifests*. Il suffit de les organiser en *modules*. Un module est une simple arborescence du nom de la classe de base qu'il contient et placée dans le répertoire */etc/puppet/modules*. La classe de base est définie dans un *manifest* de nom *init.pp*, naturellement placé dans le répertoire *manifests* du module. Si celui-ci contient d'autres classes, elles seront chacune dans un *manifest* à part, de même nom qu'elles et avec l'extension *.pp*. Les templates et les fichiers du module seront respectivement dans les sous-répertoires *templates* et *files* du module. Ainsi, pour charger automatiquement la classe *sambaclient* vue plus haut, il suffit de la déclarer dans */etc/puppet/modules/sambaclient/manifests/init.pp*.

Enfin, Puppet propose un mécanisme très simple pour attribuer à une machine donnée sa configuration spécifique. Il existe une structure de langage, *node*, qui permet d'affecter des définitions à une machine ou à un groupe de machines, en se basant sur leur nom. Le nom peut être spécifié en entier ou au moyen d'une expression rationnelle. Le nom *default* correspond à l'ensemble des machines pour lesquelles il n'existe pas de déclaration *node* explicite. Les nodes supportent l'héritage. Par exemple :

```
node www { # definition serveur www }
node www1, www2  {# la definition s'applique aux deux machines www1 et www2 }
node www.jres.org {# on peut aussi bien indiquer le nom d'hôte que le nom complet }
node /^www\d$/ {# la définition s'applique à tous les serveurs dont le nom commence par www et finit par un chiffre }
node default {# tous les hôtes qui n'ont pas de déclaration node spécifique appliquent cette définition }
node wperso inherits www {# la définition qui s'applique à wperso est la même que celle de www plus les ressources
spécifiques listées ici }
```

Cf [5] et [6] pour plus d'informations.

4.3 Perspectives

On voit ici toute la puissance d'un outil d'administration centralisée de configuration, et plus particulièrement de Puppet. La gestion d'une machine virtuelle est ainsi grandement facilitée et se limite donc aux seules étapes suivantes : création de la VM ; installation du SE (sous OpenVZ, cela se fait en même temps que la première étape) ; installation de Puppet ; et exécution de Puppet. Il n'y a ensuite plus qu'à faire évoluer la configuration centralisée, lorsque c'est nécessaire, sur le *master* Puppet.

Il est clair que le *master* Puppet est stratégique. Il peut ne pas être toujours disponible, mais il faut absolument que ses données soient protégées. Il va fatalement contenir des informations confidentielles (mots de passe, clefs SSH par exemple), mais également toute modification non contrôlée d'un ou plusieurs *manifests* peut avoir des conséquences désastreuses sur tout ou partie des serveurs gérés.

Au delà de l'aspect malveillance, il faut bien mesurer toutes les modifications que l'on apporte à une configuration, car elle peut être appliquée très vite sur un nombre conséquent de serveurs. Le mieux est de toujours tester au préalable. Il suffit de « cloner » la machine cible via Puppet et d'ajouter dans sa configuration les nouvelles ressources à valider. Cela s'écrit simplement de cette manière :

```
node machinetest inherits machinecible {# définition des ressources à tester}
```

L'utilisation d'un tel outil nécessite donc incontestablement une certaine discipline, et il ne faut pas non plus configurer directement les machines (sauf à des fins de test) mais toujours le faire dans le *master* Puppet.

Au delà de cet aspect d'administration centralisée, on perçoit plusieurs autres domaines où Puppet pourrait être utile :

- La sécurité (en complément d'un « vrai » outil de sécurité) : du fait qu'il cherche à maintenir le système dans l'état décrit dans sa configuration en central, et pour peu que l'on surveille les bonnes ressources, toutes les modifications faites en local à une machine, intentionnellement ou non (par exemple, modification du */etc/passwd*, modification des binaires pour cacher les traces d'une intrusion, ...) seront annulées dès l'exécution suivante de l'*agent* Puppet. Couplé à un reporting adéquat, cela peut permettre de détecter (et corriger) rapidement des anomalies.

- La gestion de salles d'enseignement ou de postes de travail : au prix peut-être d'une installation initiale un peu longue (s'il y a beaucoup de paquetages ou logiciels à rajouter), les modifications deviennent quasiment instantanées, et peuvent se faire pendant l'utilisation des postes, sans nécessiter aucun redémarrage, sauf si cela est requis par la ou les application(s) installée(s).

- La gestion des équipements réseau. Depuis sa version 2.7, Puppet permet de lancer des commandes sur des switches/routeurs. Les ressources sont définies de la même façon que pour les ordinateurs, à la différence près que l'agent est sur une machine intermédiaire qui appliquera les actions aux équipements via ssh (voire telnet si encore nécessaire).

5 Retours sur l'exploitation

Voici quelques réflexions, alimentées par l'expérience de ces derniers mois.

La séparation entre la puissance de calcul et l'espace de stockage est une tendance de fond depuis plusieurs années. Directement ou non, la virtualisation renforce cette tendance. En effet, on va créer, sauvegarder, dupliquer des VM. Si on doit à chaque fois transporter des données utilisateur notamment, on arrive à une perte de place, de temps, et à des risques de désynchronisation des données. A contrario, on a uniquement à se préoccuper d'avoir des VM dans un état système stable.

Le point faible du système, c'est l'hôte. Au niveau sécurité, une compromission de l'hôte implique que toutes ses machines virtuelles risque d'être compromises aussi. Il faut donc extrêmement bien le sécuriser. C'est également le goulot d'étranglement au niveau des ressources, et plus particulièrement au niveau des disques (toutes les VM écrivent et lisent via le même contrôleur disque) – encore que ce point devienne moins gênant en cas d'utilisation de stockage centralisé -, et surtout du réseau (le trafic de toutes les VM passe par la/les mêmes cartes). Dans ce dernier cas, si on a beaucoup de VM, il va très probablement falloir passer par de l'agrégation de lien, ce qui permet au passage d'augmenter la disponibilité. Attention toutefois, le type d'agrégation choisi et la configuration (présence d'un proxy ou non) vont avoir une influence non négligeable sur la robustesse et/ou les performances de la solution.

Quel est l'hôte idéal pour de la virtualisation ? Sans compter les données utilisateur, une machine la plus simple possible, avec des disques rapides, mais pas forcément gros, et surtout de la CPU et de la mémoire, suffit amplement. Tout dépend des applications à faire tourner, mais pour donner un ordre d'idée, la machine la plus petite que nous souhaitons maintenir est un Quad core standard du marché avec 16 Go de RAM, qui nous donne déjà une bonne latitude de fonctionnement. Les plus grosses que nous avons sont des bi-Quad core avec 48 Go de RAM, que nous utilisons pour les services d'infrastructure.

Plus il y a d'hôtes et de machines virtuelles, et moins on sait où est quoi. En cas de panne d'un hôte, c'est toutes les VM qui sont arrêtées. Il faut donc pouvoir les remonter au plus vite. De même, il est nécessaire de connaitre l'état de consommation des ressources des différentes VM et des différents hôtes, pour, le cas échéant, augmenter dynamiquement les ressources, ou pour migrer des VM sur d'autres hôtes. Pour toutes ces raisons, un outil de supervision est indispensable, ou mieux un outil d'« orchestration ». En plus d'offrir de la supervision, l'outil d'orchestration permet d'agir sur l'infrastructure. Les solutions de virtualisation commerciales en proposent généralement. C'est moins vrai pour les produits libres, quoique Proxmox VE en est déjà un dans une certaine mesure. Un nouveau projet, Archipel, est en cours de développement (dernière version : béta 4) et propose de l'orchestration multi-plateforme. C'est certainement un produit à suivre de très près !

6 Conclusion

La virtualisation de serveurs apporte sans conteste beaucoup de souplesse, et permet d'optimiser l'utilisation des serveurs physiques. En contrepartie, elle introduit un certain nombre de nouvelles problématiques, notamment liées à la multiplication des serveurs, qui nécessitent d'adapter ses pratiques et de mettre en œuvre de nouveaux outils. Au travers de notre expérience à l'Université de la Méditerranée et notamment sur le campus de Luminy, nous espérons avoir brossé un portrait à peu près complet de la situation et des solutions, notamment de gestion centralisée, telles que Puppet.

7 Bibliographie

[1] Françoise Berthoud et Maurice Libes, La virtualisation, pour quoi faire ? Dans *Journée Système (JoSy) « La virtualisation »* , 28 septembre 2006, http://www.resinfo.org/spip.php?article3

[2] Fabrice Lorrain, Virtualisation : Un état de l'art. Dans *CUME - Journée nationale sur la Virtualisation*, 13 mars 2008, http://cume.univ-angers.fr/Lorrain-Virtualisation-cume-pdf.pdf

[3] Bernard Perrot, État de l'art des techniques de virtualisation. Dans *TutoJRES n°6 : Virtualisation*, 12 et 13 mars 2008, http://www.jres.org/tuto/tuto6/index

[4] Comparison of open source configuration management software, Wikipedia, http://en.wikipedia.org/wiki/Comparison_of_open_source_configuration_management_software

[5] Puppet Labs Documentation, http://docs.puppetlabs.com/

[6] Benjamin Sonntag, Puppet, administration système centralisée, 29 janvier 2008, http://www.octopuce.fr/Puppet-Administration-systeme-centralisee.