Administration Système Automatisée

Pierre Gambarotto
INPT/Enseeiht
pierre.gambarotto@enseeiht.fr

Résumé

Le nombre de serveurs qu'un même administrateur est amené à gérer a grandement augmenté dans les dernières années, notamment depuis l'arrivée de nouvelles technologies comme la virtualisation et la généralisation des applications web. L'organisation de l'automatisation des tâches d'installation, de configuration et de supervision permet de répondre au défi qui est de gérer correctement ce nombre grandissant de serveurs.
L'objectif de cet article est double : présenter différentes technologies récentes d'administration système, qui permettent en les combinant d'arriver une automatisation complète du provisionning, et investiguer la conséquence sur l'organisation d'un service informatique.

Mots clefs Administration Système, Puppet, Provisionning

1 Automatiser l'administration système

L'administration système a beaucoup évolué en 20 ans. Le schéma de base reste identique :

- installation initiale d'un système d'exploitation ;
- intégration du système dans le parc : configuration réseau, DNS ...
- installation de logiciels et configuration ;
- mise à jour système et logiciels au cours de la vie de la machine ;

Certaines de ces étapes ont été simplifiées par la montée en puissance des distributions GNU/Linux qui a apporté des systèmes de gestions de paquetages logiciels assurant l'installation et la mise à jour d'un logiciel et de ses dépendances (apt/yum).

Mais parallèlement à cette simplification, le nombre de services et donc de serveurs a fortement augmenté. Les architectures systèmes ont généralisé l'utilisation de la supervision et la sauvegarde, qui allongent d'autant l'intégration d'un nouveau système dans le parc existant.

La virtualisation permet d'exploiter rationnellement les ressources matérielles tout en limitant le nombre de services hébergés sur un seul système, mais augmente par là même le nombre de systèmes du parc.

Au niveau gestion, il est fréquent de déléguer l'utilisation d'un OS une fois installé et configuré à une tierce personne, que ce soit un poste de travail ou un serveur. Par exemple, dans le cas d'une application web, l'administrateur du système installe les logiciels nécessaires (apache et mod_php, nginx/passenger, tomcat/java) et permet ensuite l'installation de l'application proprement dite à un tiers. Ceci a un impact sur la sécurité intrinsèque de la machine ainsi gérée.

L'administrateur de parc est donc aujourd'hui en face du challenge suivant :

- plus de systèmes à gérer ;
- plus de personnes avec qui interagir ;

- complexification de l'intégration d'un nouveau système au parc existant.

L'objectif d'une automatisation complète de l'administration système est de relever le gant, en proposant les caractéristiques suivantes :

- installation complète d'un système , de l'OS jusqu'à la configuration en production ;
- Pas de gestion directe d'un système isolé : si un problème survient sur un système, il doit être corrigé au niveau de tout le parc, l'exemple typique étant la faille de sécurité d'un serveur HTTP.
- Pouvoir revenir à un état stable à tout instant : il ne faut pas craindre la mise à jour.
- Il est plus facile de rejouer le processus automatisé d'installation que de réparer.
- Les membres de votre équipe doivent pouvoir participer.

2 Panorama des solutions existantes

Cette partie recense différentes solutions d'automatisation liées à l'administration d'un système d'exploitation. Le but n'est pas de faire une liste exhaustive, mais de fournir des critères permettant de classer et d'évaluer ce genre de produits. Nous distinguons 3 niveaux en fonction du service rendu.

1. Initialisation du système
2. Configuration du système
3. Déploiement et configuration d'application

2.1 Initialisation du système

Le but de ces solutions est de permettre le premier boot de la machine et l'insertion dans le réseau. On peut scinder ces solutions en 2 sous groupes, suivant que la machine sur laquelle le système va être installé est réelle ou virtuelle :

Machine réelle : solutions par déploiement par image disque (Clonezilla ou autre clone du vénérable Ghost) ou par automatisation du processus d'installation (Jumpstart, Kickstart, FAI, Cobler).La configuration réseau dans ce cas là reste classique, manuelle ou par DHCP.

Machine virtuelle : soit on gère soit même le socle (OpenVZ, VmWare, XEN, KVM) soit on fait appel à une infrastructure dans le nuage (AWS d'Amazon, Eucalyptus en open-source). La configuration réseau dans ce cas là doit au moins en partie être assurée au niveau du socle.

L'article ne va pas détailler de solutions techniques appartenant à ce premier niveau, les solutions à base de déploiement d'images et les solutions basées sur des systèmes de virtualisations ayant déjà été abordées dans des éditions passées des JRES.

Ce service n'est normalement utilisé qu'une fois par système. Ces solutions permettent néanmoins de fournir assez simplement la sauvegarde du système une fois installé, par exemple pour gérer des cas de disaster recovery.

Cela peut paraître séduisant de continuer ce processus au cours de la vie du système, mais cela entraîne une dérive des sauvegardes qui réduit la réutilisation : si 4 instances proviennent de la même source mais ont évolué indépendamment, les 4 instances seront à mettre à jour séparément.

2.2 configuration du système

Le postulat de base de ces solutions est de disposer d'un compte administrateur sur le système à configurer. Le but est de spécifier l'utilisation des outils disponibles sur le système pour installer des composants logiciels ou les mettre à jour.

L'intérêt de ces solutions réside dans la possibilité de réutiliser une même configuration sur plusieurs systèmes.

Contrairement au premier niveau (initialisation du système), les solutions de configuration ont vocation à être utilisée tout au long de la vie du système, assurant non seulement la configuration initiale, mais aussi les mises à jour. Le but est d'automatiser au maximum les opérations une fois que la configuration du système a été décrite.

Par exemple, si une mise à jour de sécurité d'un composant logiciel parait, elle doit pouvoir être appliquée automatiquement.

Un des intérêts majeurs des solutions de ce niveau est d'entraîner une centralisation du stockage de la description des différents systèmes d'un parc, et ainsi de faciliter un travail de rationalisation.

Ces produits sont souvent moins connus que les produits du premier niveau, on peut citer parmi les plus utilisés : Puppet, Chef, cfEngine, SmartFrog

2.3 déploiement et configuration d'application

Il est théoriquement possible de tout faire avec une solution du précédent niveau. On arrive cependant souvent à ces 2 limites :

- on veut déléguer l'exploitation de la machine à un tiers, sans lui donner les droits administrateurs.
- la source du composant logiciel à installer/configurer est interne, et l'utilisation des outils fournis par le système n'est pas adéquate.

Cas typique : un administrateur système & réseau d'un laboratoire fournit à une équipe de recherche un serveur pré-configuré, et un compte utilisateur . L'équipe de recherche veut déployer sur ce serveur un composant logiciel développé en interne, et va renouveler plusieurs fois cette opération au fur et à mesure de l'avancement de leur projet.

La préoccupation dans cette optique est d'avoir un environnement cible stable, et de pouvoir basculer des composants logiciels d'un système source (machine d'un développeur, dépôt subversion/git) vers le système cible, de modifier la configuration du logiciel sur le système cible, et optionnellement de pouvoir relancer un des programmes résidents pour prendre en compte la nouvelle version.

Le but des solutions de ce 3^{e} niveau est donc de faciliter ces opérations, voire de les automatiser dans le cadre d'un processus d'intégration continue. Il existe de nombreux produits dans ce domaine, du script empiriquement construit et exécuté par ssh, en passant par des surcouches de ssh comme Capistrano ou Fabric, jusqu'à des produits plus évolués comme Mcollective.

3 Puppet

Puppet est un produit opensource (GPL puis Apache depuis la version 2.7.0) de la société Puppet Labs (ex Reductive Labs).
Puppet fournit une architecture client/serveur pour décrire, gérer et distribuer des descriptions de configuration. La configuration de chaque client est décrite dans un langage spécifique, stockée sur le serveur, et appliquée sur chaque client avec les privilèges administrateur à la demande du client (pull).

3.1 Architecture et vision globale du produit

Puppet repose sur une architecture client/serveur. Sur le serveur central le démon `puppetmasterd` attend sur le port 8140 les connexions des clients sur lesquels tourne le démon `puppetd`.

Le dialogue client/serveur se fait par un web-service (protocole encapsulé dans HTTP) et chiffré par des certificats SSL, chaque client est identifié de manière unique par un certificat associé à son nom DNS. L'outil `puppetca` permet de gérer sur le serveur les certificats clients.

Les clients se connectent périodiquement (toutes les 30 minutes par défaut) au serveur, remontent au serveur une collection de faits renvoyés par la commande `facter` (@IP, hostname, @MAC ...) qui permettent au serveur d'instancier la configuration décrite pour ce client. La configuration est ensuite appliquée au client. Le serveur `puppetmasterd` fait également office de serveur de fichier central pour tous les clients.

La dépendance principale de Puppet est le langage ruby, aucune compilation n'est nécessaire pour installer puppet lui-même. Le plus simple reste néanmoins d'utiliser les paquetages disponibles de la distribution GNU/Linux utilisée.

3.2 Langage de description de configuration

La configuration d'un système est décrite dans un langage spécialisé. La brique de base est la ressource : chaque ressource dispose d'un type, et de certains paramètres dépendant du type qui vont permettre de réaliser la ressource.

Voici quelques types de ressource parmi les plus utiles : `Exec`, qui permet d'exécuter un script, `File` qui permet de définir le contenu d'un fichier, `User` qui permet de définir un utilisateur, `Service` qui assure le démarrage et le fonctionnement d'un service, et `Package` qui permet d'installer un paquetage logiciel.

Pour instancier une ressource, i.e spécifier que cette ressource va devoir être réalisée sur le système cible, il faut donner une valeur aux différents paramètres fixés par le type de la ressource, ainsi qu'un nom unique. Par exemple pour réaliser le fichier `/etc/passwd` :

```
file { "/etc/passwd":  # nom unique donné à la ressource
  owner => root, # paramètre owner, valeur donnée : root
  group => root,
  mode => 644
}
```

Par défaut, c'est puppet qui décide de l'ordre de réalisation des ressources sur un client, mais on peut préciser manuellement des dépendances entre les ressources. En plus des ressources, Puppet fournit les outils classiques d'un langage : variable, conditionnelle, gestion des chaînes de caractères et des tableaux. Les faits collectés par `facter` sur le système client sont disponibles sous forme de variable, et permettent ainsi d'adapter une recette. Voici un exemple récapitulatif commenté assez complet de ces fonctionnalités :

```
$ssh = $operationgsystem ? { # $operatingsystem : remonté par facter
 (redhat|fedora) => 'openssh-server',
 (debian|ubuntu) => 'ssh',
 default => 'openssh'
}
package { $ssh: # $ssh est la variable définie ci-dessus
 ensure => installed,
}
```

```
file { '/etc/ssh/sshd_config':
 source => 'puppet:///modules/sshd/ sshd_config', # on récupère le contenu du fichier sur le serveur
 owner => 'root',
 group => 'root',
 mode => '640',
 notify => Service['sshd'], # redémarage de sshd après changement de ce fichier
 require => Package[$ssh], # on force l'ordre de résolution, le paquetage avant le fichier de configuration
}
service { 'sshd':
 ensure => running,
 enable => true, # actif au boot de la machine
 hasstatus => true,
 hasrestart => true,
}
```

Pour organiser le code, le langage fournit des structures supplémentaires :

```
node 'machine.domaine.tld' { # restriction de la réalisation de ressources à un système spécifique }
class NomClasse { # Définit un espace de nommage }
include NomClasse # réalisation des ressources définies dans la classe
define mon_type($var1, $var2) { # définition d'une collection de ressources utilisant var1 et var2 }
mon_type { nom : # réalisation du type personnalisé mon_type
 var1 => valeur,
 var2 => valeur
}
```

La description de la configuration d'un produit complet se compose d'un ensemble de classes et de types personnalisés définissant une collection de ressources, et d'un ensemble de fichiers référencés par les ressources de type `File` (attribut `source`) ou des templates qui vont permettre de paramétrer la génération de contenus de fichiers.

Puppet permet de rassembler ces éléments en module, ce qui permet de distribuer ou de récupérer les principes d'installation et de configuration d'un produit complet, par exemple un serveur Apache et un site virtuel, un serveur de base de données et une base, un serveur ldap maître et un serveur ldap esclave.
Un module puppet peut donc être considéré comme une procédure d'installation et de configuration d'un produit.

Les implications sont particulièrement intéressantes :

* la documentation de la procédure d'installation est la procédure elle même
* la mise à jour de la procédure permet la mise à jour de tous les serveurs sur lesquels elle est employée.
* caractère open-source : la communauté redistribue des définitions complètes de modules.

La dernière caractéristique de Puppet à souligner provient de son côté centralisé. Le serveur reçoit la connexion de tous les clients du parc géré, et a donc connaissance de l'ensemble des systèmes. Pour profiter de cette centralisation, il est possible dans une recette puppet de spécifier qu'une ressource quelconque est exportée, i.e sauvegardée sur le serveur. Comme les recettes sont calculées sur le serveur avant d'être appliquées sur un client, cela permet de définir la configuration d'un client puppet en fonction de caractéristiques d'un autre client.

Donnons un exemple concret : pour établir la réplication dans un domaine kerberos, il faut que le serveur maître connaisse le nom des serveurs esclaves. Pour cela, on va exporter un fichier contenant ces informations sur chaque client puppet correspondant à un serveur esclave kerberos :

```
@@file{"/etc/krb5kdc/$realm_real/servers/$fqdn":
 tag => "kerberos_$realm_real", # tag pour repérer ces ressources plus tard
}
```

Sur le client puppet correspondant au serveur maître kerberos, on va instancier les ressources exportées par les client :

```
File <<| tag == "kerberos_$realm_real" |>>    # réalisation des ressources correspondant au tag
```

Maintenant que l'information se trouve sur le serveur kerberos (le nom des serveurs esclaves), il suffit de les utiliser dans un script pour mettre en place la réplication kerberos. Dès qu'un esclave kerberos va être rajouté, la configuration du maître va s'adapter en prenant en compte cette nouvelle information.

De manière plus générale, ce mécanisme de stockage d'information par export de ressources va permettre de définir des configurations non plus uniquement au niveau d'un système isolé, mais à l'échelle d'un parc tout entier. Puppet et son serveur centralisé permettent ainsi d'automatiser l'installation et la configuration de logiciels d'infrastructures tels que des solutions de supervision, où un serveur central de collecte a besoin des informations des systèmes surveillés, et chaque système surveillé a besoin d'information concernant le serveur de collecte.

4 Capistrano

Capistrano est une librairie ruby qui permet de décrire et d'exécuter des scripts sur plusieurs machines, et est typiquement employé pour configurer les différentes composantes d'une application web (serveur de base de données, frontend http, backend). Une recette Capistrano est constituée de différentes tâches interdépendantes (comme dans un makefile) écrites en ruby. L'exécution d'une même recette s'effectue sur plusieurs serveurs.

Conceptuellement, c'est un équivalent à make (gestion de tâches dépendantes) couplé à une surcouche de ssh, qui permet ainsi de manipuler plusieurs systèmes cibles.

Première conséquence, il faut disposer des droits suffisants de connexion sur les systèmes cibles, typiquement par une distribution préalable de clefs ssh.

La configuration de déploiement se décrit dans un fichier `Capfile,` dont voici un extrait commenté :

```
role :db, 'postgres.domain.tld' # définition de serveurs avec leur rôle associé
role :app, 'backend.domain.tld'
role :web, 'frontend.domain.tld'
set :user, "pierre" # définition de variable
desc "Description de la tâche" # définition d'une tâche
task :my_task, :roles =>:app,:web do # restriction de la tâche aux serveurs remplissant les rôles spécifiés
 run "echo #{user} fait des choses > /tmp/example"
end
after("another_task, "my_task") # ordonnancement des tâches
```

Pour exécuter la tâche ainsi spécifiée :

```
cap my_task
```

Capistrano va alors exécuter la commande spécifiée par `run` sur les serveurs dont les rôles sont spécifiés dans la définition de la tâche `my_task.`

Capistrano est fourni par défaut avec un ensemble de tâches ciblant le déploiement d'une application dont le code est dans un gestionnaire de configuration (SCM) type subversion ou git.

```
cap deploy:setup # initialisation : création de répertoire
cap deploy:update # déploiement du code à partir de la dernière version du SCM
cap depoly:start, stop, restart # contrôle de l'exécution de l'application
cap deploy:revert # retour à la version précédente
```

Il suffit pour profiter de cette configuration de base de spécifier le nom des serveurs de déploiement, l'identité à utiliser sur les serveurs, le nom des répertoires pour la destination du code, et de renseigner les coordonnées du SCM utilisé.

5 Place dans l'organisation d'un service informatique

Nous allons maintenant regarder les répercussions de l'utilisation de ces outils sur l'organisation d'un service informatique.
Tout d'abord, l'utilisation de produits du premier niveau (installation initiale du système) abaisse le coût d'installation d'une machine, et permet une utilisation optimisée des ressources matérielles.
Au lieu d'installer N services sur un même système, il est aussi facile d'installer N systèmes avec un seul service par système. On obtient ainsi moins de couplages entre les différents services installés. Il devient facile de faire évoluer un service indépendamment des autres.

Le coût du provisionning d'un nouveau système est grandement abaissé.

Chaque système est plus simple, et il est donc plus facile de le décrire avec un produit comme Puppet.

La centralisation des descriptions permet de travailler en un même point pour modulariser les configurations et permettre une meilleure réutilisation : quand je décris un serveur web, il y a une partie fixe (configuration du système, paquetage apache, configuration basique du serveur) et une partie variable (modules apache à utiliser, description des hôtes virtuels). Il devient intéressant de produire une description reflétant ce découpage pour faciliter la réutilisation, voire de récupérer un module déjà écrit par la communauté. On se construit ainsi petit à petit une bibliothèque des composants logiciels que l'on utilise dans son parc, avec sa description associée, ainsi que des exemples concret d'utilisation. L'ensemble constitue également une documentation de qualité.

Le coût initial de réalisation de la description est donc amorti au fil du temps, car réutilisée.

Les produits du niveau 1 permettent de gagner du temps immédiatement, les produits du niveau 2 consomment le temps ainsi gagné en l'investissant dans la réalisation des descriptions. Le coût en temps effectif passé sur un système est donc sensiblement inchangé au premier abord, et baisse fortement au fur et à mesure de la construction de sa bibliothèque de descriptions. Le coût réel est donc constitué par l'investissement en formation à l'utilisation coordonnée de ces produits.

Considérons maintenant par un exemple le dialogue nécessaire entre un administrateur système et un développeur pour la mise en place coordonnée d'un solution pour déployer facilement l'application du développeur :

Le développeur doit d'abord fournir les spécifications du système cible, par exemple : une applications Php à déployer sur un environnement LAMP (Linux Apache Mysql Php). Il va également fournir sa clef publique ssh.

L'administrateur système va réaliser la description de cette configuration avec Puppet en réutilisant des modules déjà écrit, éventuellement en en créant un rassemblant tous les éléments pour le cas à gérer s'il

pense que l'investissement est rentable et que le cas de figure va se représenter. Ici, la configuration consiste en : installation par paquetage d'apache, de mod_php, de mysql, création d'un utilisateur, d'une base de données et des identifiants pour s'y connecter, création d'un répertoire devant abriter le code du produit, et la configuration de l'hôte virtuel apache pour utiliser le répertoire.

Il fournit alors les coordonnées de la machine, le nom de l'utilisateur à utiliser pour la connexion ssh, les identifiants pour se connecter à mysql, le chemin du répertoire accessible en écriture.

Le développeur va alors configurer Capistrano avec ces informations, et l'utiliser pour déployer son application.

L'administrateur système est en charge d'assurer la disponibilité du serveur, et d'assurer sa sécurité. Le développeur ne s'occupe que du côté applicatif.

Cette exemple montre que les outils présentés dans l'article permettent donc une coopération entre administrateur et développeur tout au long de la vie d'une application, du développement à la mise en production.

6 Conclusion et perspectives

Cette article vous a présenté une classification des solutions techniques permettant d'aboutir à une automatisation complète de la gestion d'un système, et un focus sur Puppet et Capistrano.

L'emploi de ces outils techniques est fortement structurant, et s'il est déjà intéressant de les utiliser seul, ils prennent toute leur plénitude dans une organisation coordonnée.

L'insertion de ces produits dans votre pratique peut se réaliser de 2 manières différentes : dans une approche partant de la base, comme l'exemple de collaboration entre un administrateur système et un développeur (voir section 5), on cherche à utiliser en commun des produits qui permettent la collaboration entre des métiers différents. Cette approche est portée par le mouvement DEVOPS, qui œuvre à rapprocher les DEVeloppeurs des OpérationnnelS.

En fonction de votre organisation et de sa politique, vous recherchez peut être une approche descendante plus structurée. ITIL et CMMI-dev décrivent des processus à mettre en place pour une «bonne» gestion, et sont des approches requérant une forte volonté politique. ITIL, dont le focus initial est l'organisation d'un service informatique (avant la version 3), met l'accent sur un système de gestion de connaissance parmi les composantes nécessaires. Un produit tel que Puppet fournit une base de connaissance sur les services installés, qui peut servir de base pour développer une démarche ITIL. ITILv3 ou CMMI-dev encadrent également la création de nouveaux services, et dans ce cadre le déploiement des services au fur et à mesure de leur vie : développement, test, production, mise à jour. L'utilisation conjointe d'un produit de niveau 2 tel que Puppet et d'un produit de niveau 3 tel que Capistrano permet de fixer une procédure pérenne de déploiement du service, qui s'adapte aux différents environnements (développement, test et production), documentée et fiable.