Automatisation de l'administration système avec Puppet à la présidence de l'UHP

Sylvain Zimmermann

Université Henri Poincaré

16 février 2011

Plan

Motivations à utiliser puppet

Constats sur l'administration système par machine (1)

- nombreuses tâches répétitives identiques (*ou presque*)
- installations de machines : laborieuses
- machines jamais tout à fait identiques
- difficultés à maintenir une cohérence
- chaque machine est vue séparément comme un élément à part entière

Constats sur l'administration système par machine (2)

- de plus en plus de machines
- services en augmentation (sauvegardes, surveillance, plus tard : logs, etc.)
- explosion du nombre de relations entre machines et au sein des machines
- documentation dispersée (wiki, helpdesk/redmine, documentations annexes)
- système de suivi et de catalogage des éléments de l'infrastructure globale pas à jour
- en général : augmentation de l'entropie du système

Constats sur l'administration système par machine (3)

- on sait résoudre les problèmes
  - souvent similaires
  - leurs solutions aussi
- ce qui est fait une fois par l'homme peut être reproduit par la machine (résoudre les problèmes une fois pour toutes)

Constats - conclusions (James White Manifesto - 1)

- il n'y a qu'un système et non une collection de systèmes
- il ne sert à rien d'améliorer une procédure manuelle qu'on pourrait automatiser
- le système doit converger vers l'état désiré
- la seule source valable de l'état actuel du système est le système lui-même
- chaque élément du système doit être simple pour être mieux compris

Constats - conclusions (James White Manifesto - 2)

- utiliser de petits outils qui interagissent bien entre eux plutôt qu'un gros outil qui fait tout pas très bien
- tous les outils doivent s'authentifier et être autorisés depuis une source de donnée externe
- Le reste :
  http ://blog.websages.com/2010/12/10/jameswhite-manifesto/

Constats - conclusions (3)

- Pour que l'entropie n'augmente pas trop vite, il faut structurer le système, le contrôler. Un système de gestion de machines centralisé et automatisé rempli ce rôle. Puppet est l'un d'entre eux.
- *Puppet* n'est pas la seule solution au problème, ce n'est qu'une partie de la réponse.
- Il reste
  - les outils d'installation (kickstart et consors) avec les métadonnées des systèmes permettant d'initier l'installation des systèmes
  - le dépôt des applications/des sources de données permettant de reconstruire complètement le système

Autres gestionnaires de configuration centralisés

Cfengine	ancien (langage C)
Bcfg2	ancien (xml)
chef	inspiré par *puppet* - tout est en *ruby* (les recettes entre autres)
Uniconf	connaît pas
Quattor	celui mentionné en son temps par Jean-Michel, utilisé par l'European Data Grid.
D'autres	wikipedia *Comparison of open source configuration management software*

Pourquoi *puppet* ?

- version stable (2.6.4 actuellement)
- langage de description simple
- grosse communauté active
- de nombreux partenaires :
  - RedHat
  - Ubuntu
  - Rackspace
- utilisé par de grands groupes :
  - Google, eBay
  - Oracle, digg.com, the guardian

Généralités sur puppet

Puppet

- C'est
  - Un système de gestion centralisée de configuration de machines *déjà* installées.
  - Créé par Luke Kaniés.
- Ce n'est pas
  - Un outil d'installation de machines (rôle rempli par kickstart, dhcp, pxe, tftp, cobbler, etc.)

Historique

Luke Kanies, créateur de *puppet*

- Spécialiste de *cfengine*.
- Contributeur de *cfengine*.
- À fait du conseil auprès d'entreprises en utilisant *cfengine*.

Cfengine - limitations

- Code trop fermé (même si open source).
- Un seul contributeur majeur acceptant peu de patches (au goût de Luke Kanies).
- Écrit en C.
- Règles d'écriture laborieuses.

Conclusion

Puppet a donc été créé par Luke Kanies pour dépasser les limitations de *cfengine*.

Informations générale sur *puppet*

- Écrit en Ruby ; GPL.
- Projet débuté en août 2005.
- Langage de *description* de l'état des machines (quoi) et non comment.
- C'est à *puppet* de converger vers l'état décrit.
- Volonté de l'auteur de créer une communauté forte et ouverte autour de ce logiciel.

Plateformes supportées

- Linux - plusieurs distributions :
  - style packets debian (debian, ubuntu)
  - style packets redhat (fedora, redhat, centos, suze)
  - gentoo
  - archlinux
- Solaris, HP-UX, AIX
- BSD (free, open, net)
- Mac OS X
- Windows (quelques débuts expérimentaux).

Composants de *Puppet*

puppet	générateur de configuration de machine (à partir des recettes)
puppetmasterd	serveur de configuration
facter	remontée des faits des systèmes clients au *puppetmasterd* (*Les faits, rien que les faits !*)
Dashboard	(2.6+), interface de gestion graphique.
puppetca	le système PKI de *puppet* - gestion des clés
puppetdoc	le système de génération automatique de documentation
puppetrun	pour lancer un *puppet* depuis le serveur sur l'une des machines clientes
ralsh	un shell pour interroger ou modifier l'état d'une machine

Avantages de *puppet*

- Administration à faire sur une seule machine !
- facilite la mise en cohérence globale de l'infrastructure
- opérations standards portables entre systèmes
- serveur de fichiers centralisé (un peu comme rsync)

Avantages (2)

- facilite le travail en équipe, de groupe
  - suivi de règles communes
  - aide à formaliser des manières de faire
  - vision stratégique des machines dans leur ensemble plutôt que tactique au niveau de chaque machine
- Le code *est* la documentation (puppetdoc).
- les recettes permettent de cristalliser les compétences et le savoir faire.

Avantages (3)

- simplifie
  - les migrations impliquant plusieurs serveurs
  - la gestion globale (par exemple lors de changements de politique)
  - la gestion des dépendances entre services ou composants du système

Inconvénients

- nécessite une grande attention aux détails
- des standards de travail plus contraignants
- de nouvelles façons de travailler
- impose des méthodes et un travail en équipe
  - Gestion de version des recettes *puppet*
- travail de conception plus abouti
- nécessite de simplifier et homogénéiser les pratiques

En résumé

Plus de pouvoir suppose plus de responsabilités.

Puppet, une architecture en trois parties

Les trois parties de l'architecture de *puppet*

- Un langage déclaratif.
- Une abstraction des ressources des machines. Les ressources sont abstraites (exemple : *user* ou utilisateur, *package*).
  - les ressources sont donc *indépendantes* du type de machine (que ce soit sur Linux, FreeBSD ou Solaris) ;
  - ressource sur une machine dépend de *providers* qui décrit spécifiquement le code pour une architecture en particulier.

Les ressources

Les ressources

Ressources Ce sont les objets manipulables par *puppet* ayant un *type*, un *titre* et des *attributs*.

Types de ressources

- *user*
- *package*
- *service*
- *file*

Les Ressources (2)

Code

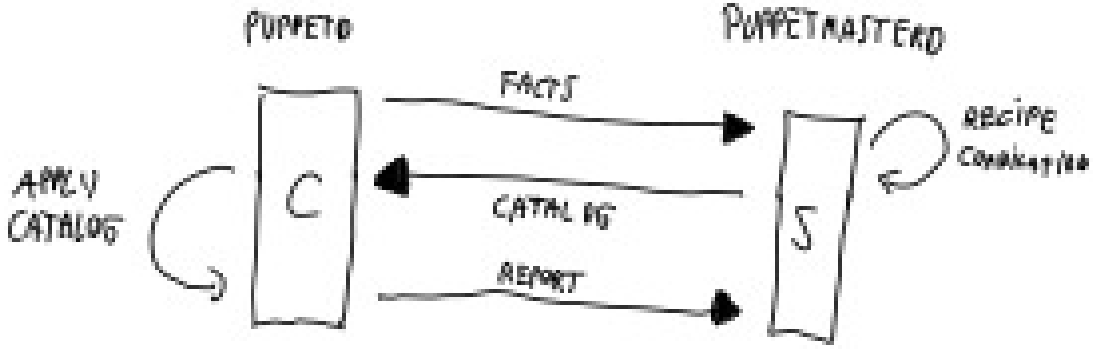
```
file { "/etc/passwd":
    owner => "root",
    group => "root",
}
```

file est le *type* de ressource

/etc/passwd est le *titre* de la ressource

owner,group sont des *attributs* de la ressource

Puppet, comment ça marche ?



Les nœuds (*node*)

Nœud une machine identifiée par un nom dns complet

- Des nœuds peuvent hériter d'autres nœuds - par exemple pour spécialiser un comportement sur une machine spécifique.
- Chaque nœud (machine) doit être autorisé à interroger le *puppetmaster* grâce à une gestion de clés publiques/privées (`puppetca`).

Les nœuds - exemples

Code

```
import "classes/*.pp"

node default {
    # administrateurs
    include unixadmins
}
```

Les nœuds - exemples (suite)

Code

```
...
node "test.cri.uhp-nancy.fr" inherits default {
    include autres\_utilisateurs
    include logrotate, crontab, fstab, snmp, git, bashr
}
```

- Nom du nœud entre guillemets et c'est le fqdn de la machine.
- Inclusion de différentes classes avec le mot-clé include.
- Les classes doivent être *importées* dans le fichier (import) pour pouvoir être inclues dans un nœud.
- L'héritage incluera donc *unixadmins* dans le nœud `test.cri.....`

Les classes (*class*)

Classe Permet de rassembler plusieurs définitions décrivant un ensemble plus complexe que les types de base. C'est une sorte de type composé.

- Comme les *nœuds*, les *classes* peuvent être héritées.
- C'est un peu la brique de base de la description d'un système.
- Héritage utilisé comme spécialisation pour des nœuds particuliers.

Les classes - exemples

Code

```
class ntp {
    package { "ntp": ensure => latest }

    file { "/etc/ntp.conf":
        owner   => "root",
        group   => "root",
        mode    => 644,
        content => template("ntp.conf.erb"),
        require => Package["ntp"],
    }
```

latest s'assure que c'est la dernière version du paquet qui est installée - il existe aussi le terme installed.

Les classes - exemples (suite)

Code

```
file { "/etc/ntp.conf":
    owner   => "root",
    group   => "root",
    mode    => 644,
    content => template("ntp.conf.erb"),
    require => Package["ntp"],
    notify  => Service["ntpd"],
}
```

template pour définir un contenu paramétré.

require pour indiquer que le paquet doit être installé avant que la ressource *file* soit traitée.

notify pour envoyer un message au service *ntpd* en cas de modif.

Les classes - exemples (suite 2)

Code

```
        service { "ntpd":
            ensure    => running,
            subscribe => [ Package["ntp"], File["/etc/ntp.c
        }
    }
```

running pour que *puppet* s'assure que le service tourne et le fasse tourner si ce n'est pas le cas.

subscribe s'inscrit aux modifications éventuelles du fichier de conf ou du paquet. Le service se redémarrera alors. Je ne sais pas s'il faut mettre à la fois *subscribe* et *notify*.

Les classes - dernières remarques

- On peut souscrire à une classe.
- Les références à des objets existants ailleurs se font avec une Majuscule initiale (*Package*, *Service*, etc.) contrairement à leur déclaration (en minuscule cette fois).

Les modules

- Plutôt une structuration pratique qu'une syntaxe du langage de Puppet.
- Rassemble un ensemble de classes, de définitions, de fichiers et de modèles (*templates*).
- Généralement conseillé : rassemblement de classes qui contribuent à la description d'un service.
- Considéré comme une bonne pratique de faire surtout des modules plutôt que de simples classes non organisées.

File : fichiers, répertoires et liens

- Le type file regroupe les notions de fichier, de répertoire ou de lien (symbolique ou non).
- Permet de gérer les droits (groupe et propriétaire) mais aussi les autorisations d'accès (lecture, écriture, exécution, *sticky bit*, etc. (rwx).
- Le contenu peut provenir d'un fichier existant dans l'infrastructure de *puppet* (*filebucket*) ou être défini *in situ* dans sa définition ou, enfin, venir d'un modèle (*template*).

File - exemple

Code

```
        file { "/etc/snmpd/snmpd.conf":
            ensure   => file,
            source   => "puppet///snmpd/files/snmpd.conf",
            require  => Package["snmpd"],
            notify   => Service["snmpd"],
            backup   => main,
            path     => "/etc/snmpd/snmpd.conf",
        }
    }
```

- Plusieurs sources sont possibles. La première qui existe sera utilisée.
- path : là où le fichier doit être copié sur le client.

À virer

- Linux
  - archlinux
- Windows