Puppet – principes

Yannick Perret - CC-IN2P3

Plan de la présentation

Introduction : gérer ce qui se trouve sur les machines

Puppet : ce qu'il fait, ce qu'il ne fait pas

Principe de fonctionnement

Le langage de description
→ exemples

L'architecture des serveurs

Autres outils autour de puppet

Introduction : installer...

Installer des machines :

- Installation « physique » (boot, partitions, système, bootloader…)
- Installation des « packages »
- Configurations spécifiques :
  - Utilisateurs locaux
  - Droits de connexion / sudo / cron / …
  - Services à activer / désactiver
  - Données spécifiques (données privées…)
  - Filtrages…
  - …

Introduction : suivre...

Suivi dans le temps :

- Mises à jour des packages (et du système)
- Mise à jour des configurations
- Gestion de la prise en compte des mises à jour (impact)
- Actions dynamiques / automatiques (ex : selon l'état de la machine)

Introduction : cas de figures

Nombreux cas de figures :

- Machines « clones »
  - Toutes identiques (à part les données d'identification)
  - Exemple : machines de calcul, parc interactif
- Machines type « service »
  - Profils variés
  - Destinataires variés (droits, filtres, particularités…)
  - Contraintes différentes (mises à jour typiquement)
- Machines de test et/ou à usage interne
  - Gestion plus souple du contenu

Introduction : pas si simple

Difficultés :

- Même des profils identiques peuvent avoir des différences
  - Matériel différent, outils différents (ex : surveillance RAID)
  - (Sous-)Réseau différent, droits/filtrages/configurations différents
- Un profil trop générique impose de spécialiser tous les cas
- Un profil trop spécifique n'est pas réutilisable
- Il faut pouvoir « croiser » des profils (1 machine, plusieurs services)
- Il faut pouvoir valider l'état des machines
- Il faut une référence claire (centrale) de l'état théorique des différentes machines

Puppet

Puppet est un gestionnaire de configuration :

- Description de l'état désiré sur la machine cible
- Client local qui applique les changements
- Serveur(s) qui gèrent les actions à effectuer par les clients

De base puppet peut gérer :

- Des packages (présence, absence, à jour...)
- Des fichiers et répertoires (contenu, droits...)
- Des services (actif / inactif)
- Des utilisateurs
- Des commandes arbitraires

Puppet (suite)

Il est possible de construire ses propres ressources, qui combinent ces différents éléments. Certaines sont disponibles par défaut :

- cron
- Interfaces réseau
- Montages
- Routage
- ssh
- Nagios
- ...

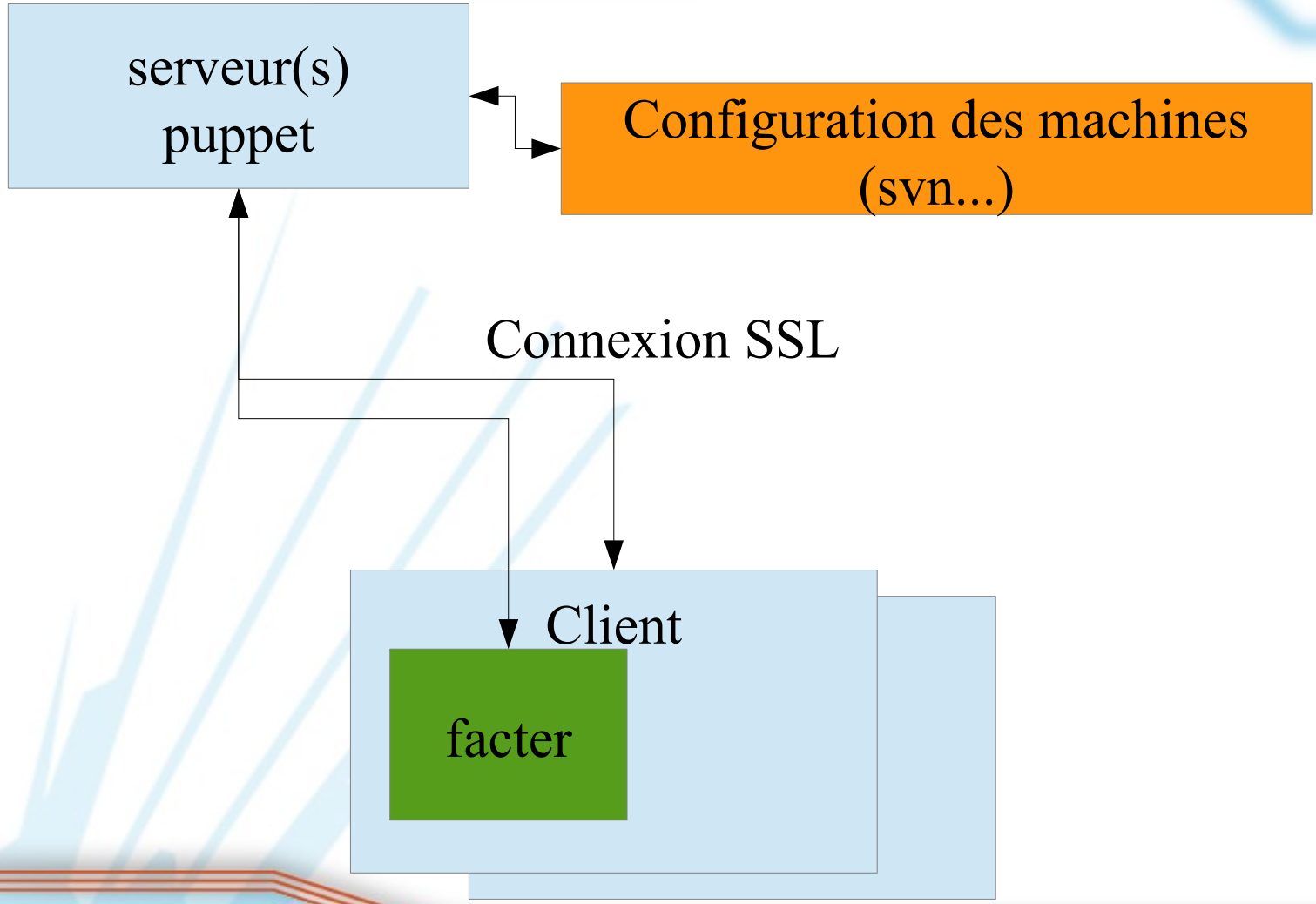
Puppet : ce qu'il ne fait pas

Puppet n'est pas un installeur « physique »
→ il tourne au niveau applicatif
→ il nécessite donc un OS installé et quelques dépendances (ruby...)

La gestion de l'installation physique doit être gérée à part, et conduire à un OS fonctionnel pour puppet :

- Client puppet
- Dépendances associées (ruby, ssl, *facter*)
- Données d'autentification (certificat)

Principe de fonctionnement



Principes

Connexions SSL : certificats signés par le serveur
Facter : extracteur/formateur de configuration locale

Les serveurs proposent :

- Un service d'authentification
- Un canal de communication pour les configurations à appliquer
- Un canal d'accès aux données stockées (contenu)

Le client propose :

- L'authentification auprès du serveur
- La gestion / application des configurations en local
- La sélection et le *scheduling* des actions
- La remontée de logs et d'erreurs

Déroulement d'une transaction

1. le client s'authentifie avec son certificat
2. le client demande tout ou partie de sa configuration
3. le client envoie ses informations locales (facter)
4. le serveur compile la configuration du client en fonction des infos
5. le serveur envoie la configuration au client
6. le client applique chaque élément de configuration

Pour appliquer chaque élément de configuration le client peut aussi accéder aux données stockées sur les serveurs (contenu de fichiers) par le canal authentifié

Exemples : ressources

```
file { "/etc/motd":
  owner => root,
  group => root,
  mode  => 644,
  source => [ "puppet://..." ]
}
```

```
exec { "tar -xf /mnt/nfs02/important.tar":
  cwd    => "/var/tmp",
  path   => ["/usr/bin", "/usr/sbin"]
}
```

```
package { "bind":
   ensure => latest,
   notify => Exec[named_files,named_files2]
 }
```

```
service { "named":
   require => Package[bind],
   enable => false,
   ensure => stopped
 }
```

Exemples : classes

Les ressources sont regroupées en classes :

```
class named {
  package {…}

  # various files
  file {…}
  file {…}

  # service
  service {…}

}
```

Utilisation des classes :

import : rendre accessible un fichier contenant des classes

include : rendre active une classe dans le profil en cours

Exemples : pour aller plus loin

Pour des profils complexes il faut des conditions et des variables
→ puppet possède un langage descriptif : une variable ne peut avoir qu'une seule valeur (ce n'est pas vraiment une variable…)

Certaines variables sont disponibles nativement via *facter* :
$hostname, $fqdn, $kernel, $ipaddress, $operatingsystem, ...

On peut ajouter ses propres variables à volonté dans les différentes classes

On peut surtout fixer des variables par machine, permettant la sélection de profils d'installation

Exemples : les conditions

De nombreuses conditions sont disponibles :

- if/else : permet de définir une ressource – ou pas – dans une classe, ou d'inclure – ou pas – une classe
- case : même chose (permet en plus les *regex*)

selector (?) : proche de l'opérateur ternaire du C. Permet de la sélection à l'affectation

```
if $is_virtual == 'true' {
    service {'ntpd':
      ensure => stopped,
      enable => false }
  } else {
    service { 'ntpd':
      name      => 'ntpd',
      ensure    => running,
      enable    => true,
      hasrestart => true,
      require => Package['ntp'] }
  }
```

```
case $operatingsystem {
    centos: { $apache = "httpd" }
    redhat: { $apache = "httpd" }
    debian: { $apache = "apache2" }
    ubuntu: { $apache = "apache2" }
    default: { fail("Unrecognized
operating system for webserver") }
  }
  package {'apache':
    name   => $apache,
    ensure => latest,
  }
```

```
$apache = $operatingsystem ? {
    centos            => 'httpd',
    redhat            => 'httpd',
    /(?i)(ubuntu|debian)/ => "apache2-$1",
     # (Don't actually use that package name.)
    default           => undef,
  }
```

Exemples : choix de la source

Lorsqu'on indique le contenu d'un fichier ce peut être :

- Un contenu explicite, avec éventuellement remplacement de variables
- Un nom de fichier sur le serveur, le nom pouvant contenir des variables (hostname, OS, …)
- Une liste de noms de fichiers (le premier qui existe est utilisé)
- Un « template » qui est un programme (ruby) construisant le contenu (et ayant accès aux variables)

Un « fichier » peut être aussi un lien symbolique ou un répertoire (avec synchronisation récursive du contenu si besoin)

Exemples : dépendances

Par principe chaque classe (et même chaque ressource) est autonome et peut être évaluée / réalisée dans un ordre quelconque (voire simultanément)

Lorsqu'un ordre est nécessaire (exemple : s'assurer que le package *named* est installé avant d'activer le service) il faut préciser l'ordre. Chaque ressource et classe peut être référencée par son type et son nom, en indiquant la contrainte :

- b*efore / require* : ordonnancement simple
- *notify / subscribe* : pour les ressources « rafraichissables »

Architecture des serveurs

Simple :

- Serveur tournant *puppetmasterd*
- Chaîne de certification créée
- Accès aux profils du « site » (classes, templates, définition des machines)
- Éventuellement configuration du stockage (contenu des fichiers)

Mais : ça ne passe pas à l'échelle

Au CC-IN2P3 :

- 4 serveurs identiques
- apache / passenger pour gérer les *puppetmasterd*
- Alias load-balanced en frontal et répartition statique + failover

Architecture au CC

Installeur : fait-maison → ne s'occupe que des éléments de base (partitions, kernel, packages de base, données privées, puppet)
Au reboot la machine bascule sur puppet.

Coté puppet :

- SVN des différentes classes puppet + fichiers exportés
  - Accessible en R/W pour sysadmin, en R/O pour les serveurs
- Base de données des machines (usage, type, OS, services...)
- Serveur d'installation : s'occupe de l'installeur + génère les variables puppet qui définissent la machines (à partir de la base de données)
- Classes regroupées en 4 catégories : système, packages, configuration, state (exécution sélective)

Quelques remarques

Évolution rapide ces dernières années :

- Client plus stable (fuites de mémoire)
- Outils de gestion / centralisation
- Documentation plus fournie
- Support et packages pour de nombreuses distributions
- De nombreux exemples et tutoriaux

Mais : au delà d'une certaine complexité des configurations il y a un effort de codage / adaptation aux besoins à faire
Exemples : sélection par profil ou par « service » ; traitement extensif ou intensif ; système d'exceptions ou multiplication des services/profils ?

Conclusion

Puppet n'est pas un système *all-in-one* (pas d'installation physique)

Puppet est un système à description d'état, permettant d'indiquer ce qu'on veut avoir, et non comment l'obtenir

Puppet permet une grande souplesse de construction des profils

Puppet permet de définir des sous-ensembles de configurations, qui peuvent tourner à des moments différents (installation, récurent)

Puppet permet de définir des « environnements » de profils, permettant de basculer une machine sur une configuration de test

Au CC-IN2P3 : utilisé pour installer et mettre à jour +2000 serveurs, plusieurs 10aines de profils, SL5/RHEL5/SL6...

Et voilà...

```
class sudo {
 file { "/etc/sudoers":  # le sudoers générique
   owner => "root",
   group => "root",
   mode  => 440,
   links => 'follow',
   source => [ "puppet://$ccpuppet/.../sudoers.$hostname",
               "puppet://$ccpuppet/.../sudoers.$operatingsystem.$cfg_usage",
               "puppet://$ccpuppet/.../sudoers.$cfg_os_version.$cfg_usage",
               "puppet://$ccpuppet/.../sudoers.$cfg_usage",
               "puppet://$ccpuppet/.../sudoers.$operatingsystem",
               "puppet://$ccpuppet/.../sudoers.$cfg_os_version",
               "puppet://$ccpuppet/.../sudoers" ]
 }
 file { "/etc/sudoers.d":   # répertoire d'include pour sudoers
    ensure => directory,
    mode => 755
 }
 file { sudo_host:   # les entrées constantes (sysunix, outils système)
    owner => "root",
    group => "root",
    mode  => 440,
    name => "/etc/sudoers.d/host",
    source => [ "puppet://$ccpuppet/.../sudoers.d/host.$hostname",
                "puppet://$ccpuppet/.../sudoers.d/host" ]
 }
 file { sudo_smurf:  # droits pour le monitoring
    owner => "root",
    group => "root",
    mode  => 440,
    name => "/etc/sudoers.d/smurf",
    source => "puppet://$ccpuppet/.../sudoers.d/smurf"
 }

 file { sudo_usage:  # droits spécifiques par usage (ou vide)
    owner => "root",
    group => "root",
    mode  => 440,
    name => "/etc/sudoers.d/$cfg_usage",
    source => [ "puppet://$ccpuppet/.../sudoers.d/$cfg_usage",
                "puppet://$ccpuppet/.../sudoers.d/empty" ]
 }
} # fin de la classe 'sudo'
```

Cette fois c'est vraiment fini

Juré.