Comment mettre son grain de sel partout

Aurélien Minet
École Normale Supérieur de Cachan
Direction des Systèmes d'Information
61 avenue du Président Wilson
94235 Cachan

Résumé

Les besoins d'automatisation sont de plus en plus prégnants : nouvelles pratiques et besoins, charge de travail augmentant contrairement aux effectifs. En parallèle, le paysage des administrateurs système change notamment grâce aux innovations répondant aux besoins des géants du web, les containers et tout ce qui est « software defined » en sont des exemples typiques.

Pour accompagner ces évolutions techniques et culturelles, notamment pour tenir le passage à l'échelle, de nouveaux outils ont donc été créés. On pense rapidement aux solutions de gestion de configuration comme Puppet ou Chef. Mais au final il y a besoin de plus, dans le sens où il faut, par principe, définir et par extension contrôler, vérifier et potentiellement corriger, sans oublier d'orchestrer lorsqu'il y a plusieurs tenants.

Le "new kid on the block" dans ce domaine est SaltStack, on pourrait croire qu'il s'agit d'un gestionnaire de configuration mais ce serait réducteur. D'ailleurs certains l'utilisent en complément d'outils comme Puppet étant donné qu'il couvre d'autres domaines. Il se différencie des solutions existantes parce qu'il s'agit d'abord d'un framework d'exécution distante et qu'il est, pour ce faire, construit autour d'un bus. Cet article propose, après avoir décrit SaltStack et expliqué son fonctionnement, de présenter les possibilités qu'offre cet outil à travers ses différentes facettes et cas d'utilisation pour mettre en relief toute la puissance qu'on peut tirer de son utilisation.

Au final l'article répondra à la question « comment SaltStack peut m'être utile ? » et présentera des exemples pratiques.

Mots-clefs

Automatisation, configuration, software defined infrastructure, infrastructure as data, event driven infrastructure, scale, idempotence ...

1 Introduction

L'automatisation, la répétabilité ou l'idempotence, la disponibilité des services, la résilience… sont des impératifs. Maintenir des scripts shell spécialisés de plusieurs centaines de lignes n'est aujourd'hui pas réaliste avec la diversité et le nombre de composants logiciels installés dans nos systèmes d'information.

De plus les architectures basées sur les micro-services, des machines-virtuelles ou des containers font qu'il y a beaucoup plus de systèmes à maintenir et des configurations plus compliquées à gérer, ce qui au final augmente l'erreur humaine.

SaltStack est l'outil qui a été choisi à la DSI de l'ENS Cachan pour automatiser mais pas seulement :l'approche proposée pour de la gestion d'infrastructures est nouvelle car, au-delà de la gestion de configuration et de l'exécution distante, la flexibilité de SaltStack permet bien plus.

2 Qu'est ce que SaltStack

2.1 Présentation

On pourrait penser que Saltstack est un outil de configuration car il est souvent comparé à CFEngine, référence l'historique, ou aux outils plus récents comme Puppet, Chef ou Ansible.

On pourrait aussi le comparer avec ClusterSSH, Fabric, Capistrano, Func… car c'est initialement un outil d'exécution distante. Mais, depuis début 2011, le projet a évolué pour devenir très actif [1] et pour offrir beaucoup de fonctionnalités tout en étant rapide et en tenant le passage à l'échelle. De grands noms d'internet l'utilisent et ont plusieurs dizaines de milliers de minions.

On peut décrire SaltStack comme un framework d'exécution distante basé sur un bus de communication ZemoMQ utilisant msgpack qui lui permet d'être rapide et simple. Il est sécurisé par la mise en œuvre d'une authentification utilisant des clés publiques AES.

SaltStack a une documentation «épaisse» : toutes les fonctionnalités sont documentées (plus de 1700 pages) et on peut rapidement être submergé par les termes, fonctionnalités, concepts. Cependant la courbe d'apprentissage est rapide, il n'y a pas de grosses difficultés et après la lecture des prochaines lignes il devrait être simple de se lancer.

2.2 Topologie et fonctionnement

Il existe deux2 types principaux de nœuds :

- le *master* : le serveur qui attend les connexions ;

- les *minions* : les clients qui se connectent au master. Ils ont un identifiant qui doit être unique et correspond donc souvent au FQDN du système.

La topologie basique consiste en un master qui contrôle des minions.

Il peut y avoir plusieurs masters afin d'assurer la haute disponibilité du service, mais ceux-ciles masters doivent avoir les mêmes données.

Pour des raisons de passage à l'échelle, des nœuds de type *syndic*, jouant le rôle de master pour un sous-groupe de minions et le rôle de minon auprès du master principal, peuvent servir de relais entre un groupe de minions et un master.

Avoir un master et un minion sur un seul système est possible.

Enfin un minion peut être seul. Cette topologie est appelée "masterless". Dans ce cas il faut déposer localement tout ce dont le minion a besoin.

Une PKI sur le master est administrée par la commande salt-key afin d'autoriser, supprimer, refuser des clés des minions qui se connectent au master.

2.3 Les grains

Ce sont des données fournies par les minions et considérées comme statiques car elles ne changent pas souvent. Il s'agit par exemple de la taille mémoire, du nombre de CPU, de la famille et de la version du système d'exploitation, du nom FQDN du système. Ces informations peuvent être fort utiles par la suite pour la configuration et l'orchestration.

Si nécessaire, on peut également définir de nouveaux grains. Ils sont déclarés en YAML dans dans /etc/salt/grains (ou éventuellement dans /etc/salt/minion).car c'est le fichier dédié.

Voici un exemple qui montre les métadonnées qu'on peut définir :

```
roles:
  - apache
  - postgresql
```

```
type: metal
datacenter: 1
rack: 2
rack_u: 14-15
```

Dans cle cas, les rôles sont liés à des services. Ils peuvent être exploités pour configurer syslog pour ces services. Les autres données sont utiles en cas de problème physique ou pour la cartographie physique.

Les grains sont utilisables dans les états ou pour cibler des minions. Mais il est possible d'y accéder directement avec module *grains* pour lister ceux qui sont disponibles, voir les valeurs et en ajouter, les supprimer ou les modifier.

2.4 L'exécution distante

La commande *salt* permet d'exécuter une fonction d'un module sur différentes cibles en parallèle.

Un événement est envoyé sur le bus ZeroMQ, les minions connectés le reçoivent et l'exécutent s'ils sont la cible de la commande. La commande est de la forme :

```
salt ciblage module.fonction [arguments] [options salt]
```

Afin de bien comprendre ce qu'on peut réaliser, voici quelques exemples:

```
salt torque.ens-cachan.fr cmd.run "who" --output=json
```

montre le module cmd avec sa fonction run qui remplace l'exécution d'une commande par ssh.

```
salt tomcat.test.ens-cachan.fr pkg.install htop
```

permet d'installer le paquet HTop.

Il faut faire attention à certaines commandes qui peuvent être destructrices :

```
 salt * cmd.run "rm -rf /tmp /*"
```

Mais la même erreur (espace après /tmp) peut être commise depuis n'importe quelle ligne de commande, ici le ciblage qui vise tous les minions (connectés) amplifiera les dommages problème.

Le retour de l'exécution par défaut est en mode texte mais grâce à des *outputrers* la sortie peut être

formatée en JSON ou en YAML, ce qui permet ensuite de faire des traitements.

Le ciblage permet de déterminer sur quel minion la commande va s'exécuter. Le choix le plus immédiat est d'utiliser l'identifiant du minion qui par défaut est son FQDN.

Il est possible d'utiliser * pour cibler une série de noms ou un sous-domaine, d'utiliser une expression régulière ou une liste :

```
salt '*.ens-cachan.fr' test.ping
```

On peut se baser sur les grains :

```
salt -G 'cpuarch:x86_64' system.reboot
```

Il est également possible de cibler un subnet et même de composer avec plusieurs types de critères :

```
salt -C 'S@172.16.3.0/24 and G@os:MacOS' desktop.say "Bonjour"
```

Il est également possible d'exécuter un module localement depuis le système où est le minion avec la commande *salt-call*. Celle-ci ne permet pas de cibler mais elle c'est pratique pour tester et est obligatoire en mode masterless.

2.5 La gestion de configuration

SaltStack utilise des fichiers « states » (avec l'extension .sls pour SaLt State) afin de définir les états dans lequel doit être le système.

Un nombre important de modules configuration (plus de 180), permettent de paramétrer les interfaces réseau, de gérer les packages, d'ajouter des utilisateurs, de manipuler les fichiers, de gérer le démarrage des services...

Les states sont « rendus » (interprétés) sur le serveur afin de gérer les dépendances, l'ordre... Le système d'exécution distante est utilisé pour lancer les fonctions utilisées. Certains modules comme Augeas nécessitent d'avoir le paquet correspondant sur le client avec son paquet/module python correspondant.

Par défaut, les états sont déclarés en YAML qui est un format de données assez simple à appréhender.

Un système de renderers permet d'utiliser d'autres formats, langages et même de les composer entre eux. D'ailleurs, il est possible d'utiliser directement Python et on peut souligner l'existence de renderers pour pyobjects, Jinja, JSON et même PGP.

Les états définis dans les fichiers sls doivent avoir un identifiant unique.

La structure d'un fichier sls type en YAML est de la forme :

```
apache.install:
  pkg:
```

```
    - installed
    - name: httpd
  file.manage:
    - name: /etc/apache2/httpd.conf
    - source: salt://files/apache/httpd.conf.jinja
```

Cet état présente plusieurs éléments :

- « apache.install » est l'identifiant de l'état et doit être unique ;
- « pkg » est un début de déclaration comme « file.manage », file et pkg correspond à des modules ;
- « file.manage » est une fonction préfixée parde son module ;
- « installed » est une fonction du module pkg ;
- « name » et « source » sont des arguments de la fonction située précédemment.

Ainsi l'approche que propose SaltStack à ce niveau est plus « data defined infrastructure » ou « infrastructure as data » que « software defined infrastructure » étant donné qu'on ne programme pas à proprement parlé mais qu'on décrit l'infrastructure via des structures de données.

Les modules SaltStack fournissent une couche d'abstraction gommant les différences entre systèmes d'exploitation. Ainsi, pas besoin de se préoccuper du système de lancement des services, du gestionnaire de packages, de la gestion des utilisateurs et des groupes.

Par exemple l'état suivant :

```
mtr :
  pkg.installed
```

lancera le gestionnaire de paquet du système (yum apt-get brew ebuild pkg_add…) pour installer le paquet mtr.

Suivant les distributions Linux certains paquets peuvent avoir des noms différents et donc abstraire le gestionnaire de paquets n'est pas suffisant, il faut aussi abstraire le nom du paquet.

Pour cela il est possible d'utiliser Jinja et d'exploiter les informations présentes dans les grains :

```
vim:
  pkg.installed:
    {% if grains['os_family'] == 'RedHat' %}
    - name: vim-enhanced
    {% elif grains['os_family'] == 'Debian' %}
    - name: vim
    {% endif %}
```

Les états sont appliqués via la fonction *state.apply* du module state. Cependant la fonction *state.highstate*, qui permet d'appliquer en une fois tout les états qui ont été définis pour un ou des minions, est sûrement plus utilisée. Celle-ci lit un fichier top.sls présent à la racine des states dans /srv/salt pour identifier tous

les états et sur quels minions ils doivent être appliqués.

L'argument très pratique pour cette fonction est test=true qui permet de voir ce qui va être fait par une exécution à blanc (dry-run).

Il y a également la fonction *state.single* pour appliquer une fonction d'un module (ex : *pkg.removed*) et la fonction *state.sls* qui applique une liste d'états définit par des fichiers sls.

Ainsi, dans un fichier top.sls on peut cibler de plusieurs manières les minions auxquels on assigne des états. En tête il y a l'id du minion suivi par les grains, une expression régulière, la valeur d'une clé d'un pillar…

```
'*':
  - dns
  - ntp
'db*':
  - postgresql
'os:(RedHat|CentOS)':
  - match: grain_pcre
  - repos.epel
'web* or G@role:webserver':
  - match: compound
  - apache.install
```

Le système de priorisation des états est de type impératif, cela implique que l'ordre d'exécution sera celui de l'ordre d'apparition dans les fichiers sls.

Pour des raisons pratiques on peut basculer dans un mode déclaratif avec des pré-requis. Pour cela les directives les plus utilisées sont :

- *require* qui permet d'indiquer si un état a besoin d'un autre, utile pour une dépendance directe
- *watch* qui permet d'observer si un état est appliqué, pratique pour relancer un service après un changement de configuration
- *onfail* qui permet de « réagir » si un état échoue

Il y a des directives similaires qui se terminent par "_in" et qui définissent le ré-requis en sens inverse.

Comme avec ce système, il peut être fait référence à des états potentiellement déclarés dans d'autres fichiers il y a la directive *include* (et *exclude*).

Dans la même logique, il peut y avoir besoin que l'état ne soit appliqué que dans certaines conditions. Pour cela des directives sont disponibles notamment *unless*, *onlyif*.

Un planificateur de tâches, qui peut remplacer cron, permet de définir l'application d'un état de manière simple et souple directement dans un état via la fonction *schedule.present*. L'exemple type est :

```
schedule:
  highstate:
    function: state.highstate
    minutes: 60
```

2.5.1 Les pillars :

Ce sont des données stockées sur le master. Contrairement aux states ces données ne seront présentes que

sur les minons pour lesquels elles sont définies. C'est pourquoi il est recommandé de définir les données sensibles dans les pillars.

Ainsi, un état qui sera appliqué à différents minions dédiés à MySQL pourrait être :

```
mysql-set-root-password:
  mysql_user:
    - present
    - name : root
    - password : {{ pillar['root-pwd-mysql'] }}
```

sachant que chaque minion aura son propre pillar avec une valeur spécifique pour la clé *root-pwd-mysql*. Les pillars permettent donc d'avoir des états génériques où les clés lors du rendu sont remplacées par leur valeur définie dans un pillar.

Comme pour les états il y a un top.sls pour les pillars afin de pouvoir les affecter à différents minions.

Par défaut , les pillars sont en YAML et on peut également utiliser des renderers.

Par exemple, le renderer PGP peut être utilisé pour des données très sensibles. Concrètement cela se traduit par une ligne en début de fichier avec un shebang spécifique, puis par la clé et la valeur au format PGP (sans oublier l'indentation avec 2 espaces)

```
#!yaml|gpg
root-pwd-mysql:
  -----BEGIN PGP MESSAGE-----
  Version: GnuPG v1

  aeT7+PfGtDJ/oz3ucNh6Kd5hQEMAweRH…
  …
  …
  …
  -----END PGP MESSAGE-----
```

Au final cela permet de partager directement des states sur une forge publique sans dévoiler des données sensibles.

2.6 La mine

C'est un élément du master qui collecte des données générées par des minions pour les rendre disponibles auprès des autres minions (avec le module *mine*). Contrairement aux grains, les données issues de la mine ont vocation à être rafraîchies (60 minutes par défaut).

Dans l'exemple suivant, correspondant au fichier /srv/pillar/ssh_host_keys.sls, les clés ssh des systèmes sont extraites :

```
mine_functions:
  public_ssh_host_keys:
```

```
    mine_function: cmd.run
    cmd: cat /etc/ssh/ssh_host_*_key.pub
  public_ssh_hostname:
    mine_function: grains.get
    key: id
```

La commande ci-après permettra de visualiser les données extraites :

```
salt '*' mine.get '*'  public_ssh_host_keys
```

2.7 Les Formulas

Ce sont des états prédéfinis, généralement assez complexes dans le sens où elles utilisent les fonctions présentes dans les modules et des patrons Jinja avancés pour fournir des configurations de haut au niveau prêtes à l'emploi.

Pour ajouter des formulas il faut modifier la configuration du master (et le relancer).

Cela se paramètre au niveau de file_root en indiquant où est enregistré l'arborescence de la formula. Ppour avoir une version à jour la directive gitfs_remotes peut être utilisée :

```
gitfs_remotes:
  - https://github.com/saltstack-formulas/openssh-formula
```

Cette formula offre plusieurs fonctions, l'une d'entre elles permet d'exploiter la mine en l'occurrence les clés ssh publiques des serveurs (voir l'exemple précédent). Pour créer le fichier known_hosts, on peut avoir un fichier /srv/salt/known_hosts.sls qui contient :

```
/etc/ssh/known_hosts :
  openssh.known_hosts
```

(les données collectées par la mine dans l'exemple sur cette dernière sont exploitées)

Les formulas sont normalement prêtes à l'emploi avec des paramètres par défaut. S'il y a besoin de les changer, un modèle de pillar, facilement adaptable pour coller rapidement aux besoins, est généralement disponible.

Depuis la version 2015.8 il y a SPM : Salt Package Manager, qui permet une redistribution des formulas tel des paquets. Dans la configuration du master, il faudra seulement définir les entrepôts qui sont utilisés. La commande spm permet ensuite d'installer ou d'enlever les formulas installées.

Les formulas sont une capitalisation de la communauté des utilisateurs de SaltStack qui continuent de les enrichir au fil du temps.

2.8 Les returners

Par défaut les minions retournent des données au master, mais ces dernières ne sont pas forcément exploitables. Dans certains cas, il serait plus opportun de les enregistrer dans des bases pour une analyse ou un archivage. Ou plus orignal, les données pourraient, en cas de problème, être envoyées par SMS ou XMPP, ou au contraire, en cas de disponibilité d'un service, ce dernier pourrait être enregistré auprès d'un service comme ETCD.

2.9 Les beacons

Ce mécanisme permet aux minons de générer des événements lorsque qu'une activité système à lieu au niveau fichier, charge, service, shell (authentification), usage du disque et du réseau.

Par défaut, la surveillance de nouvelle activité a lieu toutes les secondes, cette période peut être changée via l'argument interval.

Voici un exemple de configuration placé dans /etc/salt/minion.d/beacon.conf :

```
inotify:
  /opt/transfert/fichier.csv:
    mask:
      - modify
```

Lorsque le fichier /opt/transfert/fichier.csv est modifié, l'événement :

salt/beacon/minion_id/inotify//opt/transfert/fichier.csv
est envoyé avec un tag et une structure de données contenant des informations sur l'événement contenant l'id du minion et, dans le cas présent, une clé *change* avec comme valeur IN_MODIFY et, une clé *path* avec le chemin vers le fichier.

Pour surveiller les événements, il faut lancer sur le master :

```
salt-run state.event pretty=True
```

Depuis la version 2015.8 il est possible d'envoyer un évènement lorsqu'un état est appliqué avec l'option fire_event. Cela peut être utile pour déclencher une suite d'actions en fonction du résultat.

2.10 Reactor

Ce système peut être assimilé à un trigger sur le master, il permet de réagir à la réception d'événements .

Dans le configuration du master il est possible de définir, en fonction des événements reçus, des fichiers sls au format YAML et Jinja avec, contrairement aux autres fichiers, 2 variables accessibles : data et tag.

Dans /etc/salt/master.d/reactor.conf est défini :

```
reactor:
  - 'salt/beacon/*/inotify//opt/transfert/fichier.csv':
    - /srv/reactor/import_csv
```

Par suite, il faut un fichier /srv/reactor/import_csv.sls

```
force-reload-csv-data:
  local.state.sls:
    - tgt: {% data['data']['id'] %}
    - arg:
      - db.reloadcsv
```

L'action du reactor aura pour effet d'appliquer le state qui, comme son nom l'indique, lancera le rechargement des données csv dans la base sur le système du minion qui a émit l'événement.

2.11 Les runners

Il s'agit d'un ensemble de commandes spécifiques fonctionnant sur le master. Elles peuvent être lancées en ligne de commande via *salt-run* et n'interagissent pas forcement avec les minions.

On peut citer le runner *manage* pour obtenir des informations sur le status des minions, le runner *job* qui permet de suivre tout ce qui est lancé via la commande *salt*.

Et pour finir le runner *state.orchestrate* mérite qu'on y porte une attention particulière. En effet, les fonctions du module *state* comme *state.sls* ou s*tate.highstate* s'exécutent de manière concurrentielle et indépendante sur chaque minion alors que *state.orchestrate* s'exécute sur le master. Cela lui permet d'appliquer des états sur des minions sans que cela soit simultané mais orchestré dans le sens où les exécutions doivent être coordonnées entre les différents minions.

Par exemple, il peut être pratique dans une architecture applicative N-tiers de créer une base de données puis le compte pour se connecter à cette dernière, sur un autre minion d'installer l'application et de la démarrer ensuite, pour au final modifier la configuration d'un serveur HAProxy sur un troisième minion.

2.12 Virtualisation et Cloud

SaltStack offre de solides fonctionnalités pour gérer les machines virtuelles ou les containers.

Pour la virtualisation, il y a le module *virt* qui s'appuie sur libvirt pour gérer des machines virtuelles, sur le même modèle il y a un module *lxc* et un module *dockerio*.

Pour le cloud public ou privé, il y a Salt Cloud, qui à partir de profils, permet de gérer des VM. Pour cela il faut définir un provider dans le répertoire /etc/salt/cloud.providers.d sous la forme d'une fichier qui va contenir les informations nécessaires pour utiliser la plateforme (un driver, une adresse, un identifiant et un mot de passe) et une ou plusieurs machines virtuelles dans /etc/salt/cloud.profiles.d .

Ces fonctionnalités sont essentielles pour gérer une infrastructure actuelle, à noter que pour les serveurs physiques SaltStack peuvent être couplé à Foreman.

2.13 Interfaces REST :

Les services web sont incontournables dans les systèmes d'information modernes ne serait-ce que pour l'interopérabilité entre services.

C'est pourquoi Saltstack peut consommer des services REST. Par exemple, sur le master, il est possible d'exécuter :

```
salt-run http.query http://...fr/app params='{"param": "value"}'
```

On peut ainsi imaginer qu'un reactor devienne un « webhooks ». Par exemple, un événement correspondant à une erreur déclenche la création un ticket dans une application de suivi des incidents via un runner.

En sens inverse, SaltStack offre sa propre API, la « salt-api », la version Entreprise offre une interface Web l'utilisant. Un bon exemple d'utilisation de cette API est Saltpad [2] qui offre une interface web pour effectuer des tâaches et appliquer des états tout en permettant d'avoir un suivi sur le résultat d'une exécution. C'est une alternative pratique à la ligne de commande, notamment pour ceux qui ne sont pas des utilisateurs avertis de SaltStack.

3 Retour d'expérience et prospective

3.1 Quelques installations et configurations particulières

Pour installer des logiciels comme Oracle Database on ne peut pas juste utiliser le gestionnaire de paquet de la distribution même après y avoir ajouté un dépôt. Nous avons donc réalisé un état qui se décompose en étapes successives :

- installation des pré-requis via le module *pkg* ;
- création des groupes et de l'utilisateur par les modules ad hoc ;
- modification des fichiers dans /etc avec *file.append* ;
- utilisation de *cmd.wait* pour prendre en compte les modifications avec sysctl ;
- copie des fichiers d'installation avec *file.manage* ;
- Décompression, installation avec *cmd.run* et validation de l'état avec *unless* et *onlyif*.

Cela fait au final un fichier sls d'environ 200 lignes.

L'installation d'HAProxy est simple puisqu'en général il y a un paquet fourni par la distribution utilisée. Par contre, la configuration n'est pas immédiate car il faut définir les différents frontend et backend. Par chance, il existe une formula qui simplifie beaucoup de travail puisqu'il n'y a plus qu'un pillard à configurer. On peut l'optimiser en utilisant les grains pour obtenir des informations sur les serveurs de backend.

3.2 Gestion de bornes WiFi

Sur le campus, depuis plus de deux2 ans, une centaine de bornes WiFi ont été déployées, elles fonctionnent sous OpenWRT et leurs quelques Mo de stockage ne permettent pas d'installer SaltStack. Par défaut, leur gestion se fait par ssh mais ajouter un SSID ou changer un paramétrage sur toutes les bornes manuellement peut être vite laborieux.

Les bornes étant déclarées dans un sous-domaine spécifique le fichier /etc/salt/roster est généré facilement à partir du DNS :

```
{% for ip in
'be0-ap01.wifi.ens-cachan.fr',
  …
'test-ap01.wifi.ens-cachan.fr',
%}
{{ ip }}:
  host: {{ ip }}
  user: root
  priv: rsakey_wifi
{% endfor %}
```

Il permet au master de se connecter sur les bornes et comme notre nommage des bornes est lié à leur localisation physique il nous est possible. en cas de problème sur une zone, de relancer rapidement les bornes la desservant avec une commande du type :

```
salt-ssh 'mc4-ap*.wifi.ens-cachan.fr' -r  'wifi reload'
```

A noter qu'il est possible depuis les dernières versions d'utiliser des roster plus dynamiques, notamment avec scan qui a comme argument l'adresse d'un réseau, cloud qui ré-utilise les informations relatives aux créations de machines virtuelles via *salt-cloud* et, plus étonnant, ansible, ce dernier étant parfois utilisé en même temps que SaltStack, il est par son mode de fonctionnement à même de fournir les éléments de connexion ssh.

Dans le cas de systèmes où on ne peut pas installer un minion ni même avoir ssh, il y a la possibilité de mettre en place des Salt-proxy-minion afin de pouvoir communiquer avec ce genre d'équipement. On pense rapidement aux objets connectés, aux systèmes industriels utilisant un bus (USB, série…).

3.3 Gestion de parc

Salt-minion peut s'installer sous Windows, MacOS X et pour ces systèmes d'exploitation des modules spécifiques sont disponibles.

Nos images Windows pour les postes clients sont maintenant vierges : le sysprep télécharge un script powershell qui installe un minion lui indiquant son id et l'adresse du master.

La clé du minion est acceptée manuellement et un hightstate est lancé. L'ordinateur est alors configuré automatiquement : création d'utilisateur, connexion au domaine, modification de la base de registre, ajout de logiciels, d'une GPO ou d'imprimantes …

Régulièrement nous mettons à jour winrepo qui est l'entrepôt pour le système de paquets que propose Saltstack. Nous l'alimentons avec des paquets comme Kaspersky et nous réalisons des campagnes de mises à jour.

Nous envisageons d'automatiser l'exécution d'un hightstate lorsqu'un minion démarre et de consolider les informations à propos l'ordinateur client et de son installation dans une base de données afin de constituer un inventaire comme peut le faire OCS.

Notre parc MacOS est moindre, nous utilisons un script shell qui est lancé manuellement après la première ouverture de session pour installer Salt et ses dépendances. Pour l'instant nous n'utilisons quasiment que les modules *brew* et *cmd* pour Homebrew/ cask afin d'installer des logiciels.

3.4 Création d'un module

Suite au changement d'antivirus, l'idée est venue de remplacer la console centralisée d'administration par un module SaltStack. J. Fromont, gestionnaire de parc au laboratoire PPSM avec qui la DSI collabore étroitement, a donc, sans être un développeur Python, pris l'initiative d'écrire un module permettant de lancer un scan, vérifier la licence, ajouter une licence, lancer une mise à jour…

Techniquement l'écriture d'un module passe par l'implémentation de quelques fonctions en Python pour implémenter les fonctionnalités souhaitées et éventuellement l'exploitation de variables et fonctions propres à SaltStack, par exemple, les grains pour vérifier des prérequis.

Comme étendre SaltStack est relativement simple nous envisageons de réaliser divers modules notamment un pour la gestion des paquets sous MacOS X. Actuellement, il y en a un qui s'appuie sur Homebrew via un provider spécifique mais ce dernier ne gère pas directement les installations d'applications sous forme de fichier dmg. Comme le fonctionnent du provider *pkg_win* avec le module *winrepo* nous donne satisfaction, un système similaire est envisagé.

3.5 Comme pour la sauvegarde

« Automatiser c'est bien, vérifier de l'automatisation c'est mieux »

Certains déploiements comptent plusieurs dizaines de milliers de minions, mais sans atteindre ces chiffres effectuer un highstate sur un nombre important de minions peut générer beaucoup de retours qu'il faut analyser et manuellement ce n'est pas réaliste.

Aussi un système, tout au long de son cycle de vie, va évoluer via une intervention humaine, via les mises à jour... il faut donc vérifier que ce qui à été défini et configuré l'est toujours de manière correcte.

Enfin, certains états ou configurations peuvent être dynamiques et évoluer en même temps que l'infrastructure. Certains effets de bords doivent donc être détectés.

Pour ces problématiques on peut envisager des solutions comme ServerSpec [3] ou des tests unitaire mais sans les remplacer. Il existe un programme original : salt-highstate-stats [4]. Il réalise un highstate avec test=True et collecte les retours des minions pour les analyser afin de calculer la distance entre ce qui est défini (la théorie) et ce qui est au niveau de l'infrastructure (la réalité).

3.6 Salt-monitoring

Via les modules *psutil*, *status*, *service*, *cmd* il est relativement simple de faire du monitoring. En effet, ces modules proposent des fonctions du type *ps.cpu_percent* ou *status.loadavg* qui permettent d'obtenir des informations sous forme de mesure.

Si on va plus loin, il ne reste qu'à les enregistrer avec l'aide d'un des nombreux returner : de type NoSQL à une base SQL en passant par Carbon il y a le choix. Il faut ensuite mettre en œuvre une solution comme Graphite pour visualiser ce qui est stocké.

Un exemple de cette logique est Saltmon [5], il illustre bien ce qu'il est possible de faire avec Saltstack.

4 Conclusion

SaltStack peut permettre d'aller très loin dans l'automatisation, dans le provisonning cloud, la configuration, la reconfiguration ou même l'ajout et la suppression dynamique de nœud pour faire évoluer l'infrastructure en fonction d'événements comme la charge, une panne physique...

Gérer une infrastructure avec SaltStack prend du temps mais il faut l'envisager comme un investissement surtout qu'il peut rapidement être rentabilisé. En premier lieu, l'approche Data Defined Infrastructure permet aux states de faire office de pseudo documentation, surtout s'ils sont commentés.

Ensuite, parce qu'on peut adopter une approche Test Driven Infrastructure qui permet d'améliorer la fiabilité et la qualité de ce qui est déployé.

Il faut souligner que SaltStack peut faire peur car avec on peut avoir le contrôle total de toute l'infrastructure et il ne faut pas que cela tombe entre de mauvaises mains.

Webographie

[1] Statistiques mensuelles: https://github.com/saltstack/salt/pulse/monthly

[2] Boris Feld, https://github.com/tinyclues/saltpad/

[3] Gosuke Miyashita, http://serverspec.org

[4] Arthur Lutz, https://bitbucket.org/arthurlogilab/salt-highstate-stats

[5] Peter Baumgartner, https://github.com/lincolnloop/salmon