Rapport projet TOP
Test automatique de la plate-forme Grid'5000

Arthur Garnier
Encadré par Lucas Nussbaum

1er Juin 2015

Table des matières

1 Contexte

INRIA, institut national de recherche en informatique appliquée, est un centre de recherche divisé en plusieurs équipes. Les équipes regroupent parfois plusieurs projets, c'est le cas de Madynes, dans laquelle je travaille pour le projet Grid'5000. Celui-ci a pour objectif de mettre à disposition une plate-forme de test à grande échelle dans tous les domaines de l'informatique et en particulier l'informatique distribuée comme le Cloud et le HPC [1].

Grid'5000 est la première infrastructure dans son genre, elle permet à des équipes de recherche académiques, auxquelles s'ajoutent parfois des partenaires industriels, de réaliser une expérience à échelle réelle, comme tester un réseau P2P, ou un protocole réseau. En effet, Grid'5000 permet de configurer un réseau avec des caractéristiques spécifiques et de déployer sur tout ou une partie des nœuds une pile logicielle spécifique (du système à l'application) pour la durée d'une expérimentation.

L'infrastructure est opérée par un directeur technique qui pilote une équipe support, ainsi qu'une équipe de développement. La plate-forme étant en constante évolution, les décisions d'orientation pour le développement de celle-ci sont prisent par le comité d'architecte, composé de six membres.

1. High Performance Computing

2 Description du problème

La plate-forme est disposée sur 10 sites et regroupe plus de 1000 machines, représentant environ 8000 cœurs. Pour faciliter son utilisation et étendre les possibilités pour les utilisateurs, de nombreux services ont été développés et mis en place. Ces services sont par exemple la réservation de machines, le déploiement de leur propre environnement logiciel sur celles-ci, la possibilité de les isoler au niveau réseau (VLAN) ou encore la supervision de la consommation électrique lors de l'utilisation des machines.

Toutefois, la multiplicité des sites peut provoquer des problèmes d'homogénéité entre ceux-ci, tandis que le nombre important de machines implique des pannes matérielles et les divers services peuvent devenir inaccessible (incompatibilité suite à des mises à jour, bugs divers, incidents sur des services, ...).

Afin de repérer au plus tôt ces problèmes plusieurs solutions ont déjà été mise en place, comme Nagios qui vérifie l'état de chaque service régulièrement et bugzilla afin de reporter d'éventuels bugs repérés par l'équipe technique ou les utilisateurs. Or, il est gênant que les utilisateurs, plus particulièrement les nouveaux, soient confrontés à ces problèmes et qu'ils aient à les remonter. En effet, plusieurs « travaux pratiques » sont mis à disposition pour prendre en main la plate-forme. Si les outils ne fonctionnent pas ou mal, il parait évident que pour un utilisateur désireux d'apprendre comment les utiliser, il peut se sentir freiné s'il pense que le problème vient d'une mauvaise manipulation de sa part.

Il apparaît un besoin de limiter au maximum ces situations en testant régulièrement la plate-forme, et ainsi détecter les problèmes avant les utilisateurs. L'objectif du projet est donc de mettre en place ces tests de façon efficace pour minimiser le délai de découverte du bug. Ces tests devront être facilement modifiables, que ce soit le code en lui même ou en cas de changement de l'architecture. Les résultats devront permettre d'identifier rapidement le problème, ou du moins la commande en cause, tout en limitant les faux positifs.

3 Présentation du travail réalisé

Face à ce problème, il aurait pu paraître intuitif de réaliser des tests unitaires, chacun testant chaque service ou outil, mais ces tests ne reflètent pas l'utilisation qui est faite de la plate-forme par l'utilisateur. Par exemple, si un test vérifie le fonctionnement d'un déploiement OpenStack [2] et un autre le fonctionnement de l'installation de Debian Wheezy sur un noeud, les deux fonctionnent. Mais il est possible que le déploiement d'OpenStack ne fonctionne pas sur Debian Wheezy uniquement, alors qu'il fonctionnait sur notre environnement de test.

Le travail s'est déroulé en quatre étapes majeures, en commençant par la création de scripts pour réaliser les TP automatiquement (3.1), puis par l'intégration à la plate-forme Jenkins (3.2) comprenant une fiabilisation des tests (3.2.1), puis la mise en place d'une génération automatique de la configuration Jenkins afin d'être plus maintenable au sein de Grid'5000 (3.3). Enfin, par une simplification pour les administrateurs de l'utilisation hebdomadaire de Jenkins (3.4).

3.1 Automatisation de l'exécution des TP

Il n'est donc pas pertinent dans ce cas d'utiliser les tests unitaires, mais il n'est pas possible non plus de tester toutes les combinaisons possibles d'utilisation. Afin de couvrir une partie importante des possibilités de la plate-forme et éviter le cas le plus gênant, c'est à dire celui où les problèmes surviennent dans les travaux pratiques, le choix a été de scripter ces derniers en testant l'enchaînement de chaque commande et en vérifiant que le résultat correspond bien à ce qui est attendu.

Les tâches étant souvent répétitives :

— Réserver des ressources
— Déployer l'OS
— Exécuter la commande en ssh, vérifier son code retour
— Libérer les ressources
— Etc.

Il a été nécessaire de développer une bibliothèque pour utiliser rapidement ces fonctions usuelles. Grid'5000 fournit une API REST permettant d'effectuer certaines actions, comme

2. Ensemble de logiciels open source permettant de déployer des infrastructures de cloud computing

la récupération d'informations, ou la soumission d'une réservation de ressources. La bibliothèque, développée en Ruby, a donc été orientée pour interagir avec cette API. Par exemple pour le cas d'un déploiement :

```
1
2  def deploy
3      logger.info "[#{@job.site}] launching '#{environment}' on
4  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣[#{@job.oar_job['assigned_nodes'].join(',')}]"
5      @deployment = @session.session.root.sites[:"#{@job.site}"].deployments.submit(
6          {:nodes => @job.oar_job['assigned_nodes'], :environment => environment,
7                  :key => File.read(File.expand_path(ssh_key)), :vlan => vlan })
8      if deployment.nil?
9        logger.error "[#{@job.site}] can not submit deployment "
10       return false
11     else
12       deployment.reload
13       logger.debug "[#{@job.site}] Got the following deployment: #{deployment.inspect}"
14       logger.debug "[#{@job.site}] Waiting for termination of deployment
15 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣##{deployment['uid']} in #{deployment.parent['uid']}..."
16         ...
```

Sur ce morceau de code, ligne 4, l'API est sollicité pour effectuer un déploiement (via l'objet session qui utilise un Gem d'interaction avec les API REST). En fonction des retours de l'API, il est déterminé si le déploiement peut-être effectué. Si tel est le cas on attend que le déploiement soit terminé (impliquant d'autres vérifications, comme un timeout).

Le nombre important d'opérations et de vérifications pour ces tâches valide la nécessité de cette bibliothèque, qui permet d'avoir les mêmes procédés et vérifications sur chaque script de test. De plus on obtient un code très simple dans les scripts de test, facilitant la mise en place et la maintenabilité.

```
1
2  tp = Tp.new(name)
3  job_id, nodes = tp.sub(site, "-I -t deploy -l {type='kavlan'}/vlan=1+nodes=3,walltime=01:40:00")
4  kavlan_id = tp.exec("frontend.#{site}.grid5000.fr", user, "kavlan -V -j #{job_id}")[:stdout].strip
5  tp.exec("frontend.#{site}.grid5000.fr", user, "kavlan -e -j #{job_id}")
6  tp.deploy("ubuntu-x64-1204", kavlan_id)
7  #...
```

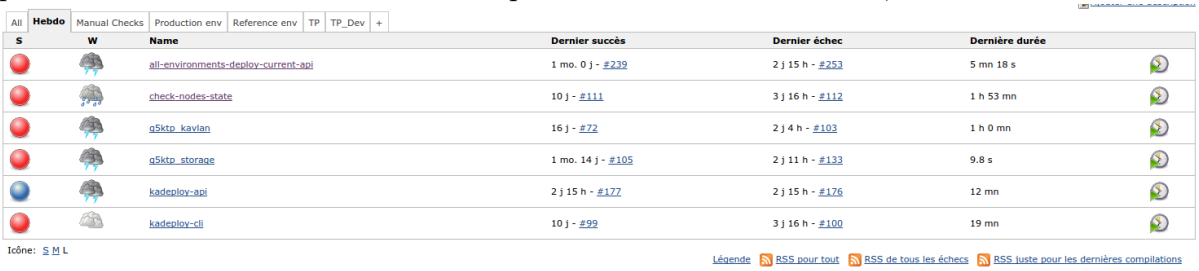
Sur le code ci-dessus, qui teste le début du TP présent ici, on remarque la facilité de mettre en place les commandes comparées à celles proposées sur la page Wiki. À la ligne 1, les ressource sont réservées puis deux commandes sont exécutées sur la machine de « frontend [3] » et enfin une image Ubuntu 12.04 est déployée sur les ressources réservées.

3. Machine frontière entre l'intérieur de Grid'5000 et internet

3.2 Exécution automatique via Jenkins

Dans le cas idéal, les tests devraient être effectués à chaque fois qu'une modifications de la plate-forme a lieu. Mais elle est modifiée plusieurs fois par jours, ce qui impliquerait de nombreux tests et une forte utilisation des ressources, en effet certains tests nécessitent jusqu'à dix noeuds en même temps. Il a donc été décidé de les exécuter régulièrement, les lundis et mardis matin. Ces jours ont été choisis car une réunion audio a lieu tous les mardis à 11h entre les membres de l'équipe technique, pour, entre autre, vérifier l'état de l'infrastructure. La vérification de ces tests à donc été ajoutée au planning de cette réunion chaque semaine.

L'exécution régulière des tests a été confiée à Jenkins, un logiciel d'intégration continue, qui permet d'avoir une vue d'ensemble rapidement de l'état des tests, comme ci-dessous :



L'avantage de Jenkins est qu'il est libre et modulable avec de nombreux plugins. Il permet de mettre en place des configurations de test sous forme de matrice, c'est-à-dire, dans notre cas, le lancement d'un test sur chaque site avec chaque image à déployer, et ce, de façon très simple avec deux listes.

Malheureusement, Jenkins est très laborieux à utiliser, beaucoup de clics sont nécessaires pour créer un nouveau job [4] ou afficher la console de sortie pour voir les erreurs. Plusieurs outils ont été développés pour faciliter cette utilisation.

3.2.1 Gestion des builds instables avec TextFinder

Les résultats de Jenkins par défaut, sont soit rouge en cas d'échec, soit bleu en cas de succès. Mais sur Grid'5000, au moment des tests il n'y a pas toujours les ressources requises le script. Dans ce cas le script retourne une erreur avec le message : « Reservation Timed Out », ce qui pour l'équipe technique, ne correspond pas à une erreur, c'est donc un faux positif. Un plugin a été ajouté, TextFinder, il permet à partir d'une *regex* trouvée dans la

4. C'est-à-dire un même script à lancer sur différentes configurations

sortie console des « runs [5] », de transformer son état en jaune, c'est-à-dire instable, plutôt qu'échec. Ce plugin est configurable avec Jenkins Job Builder, il a donc suffit d'ajouter sa configuration dans chaque job dans le fichier YAML avec comme *regex* : Reservation Time Out, par exemple.

Le passage d'un run à l'état instable signifie qu'il n'est pas exploitable lors de la réunion hebdomadaire, ce qui retarde d'une semaine la détection d'un éventuel bug. Il a donc été convenu que chaque run instable serait relancé une fois quelques heures plus tard. En théorie un plugin permet de le faire : Naginator. Mais deux problèmes sont apparus : Premièrement il ne permet pas de relancer uniquement les runs instables et deuxièmement il n'est pas entièrement compatible avec Jenkins Job Builder.

3.2.2 Relancer les builds instables

Après quelques recherches pour trouver une solution alternative, comme créer un script pour faire le travail de Naginator mais en passant par l'API de Jenkins, il s'est avéré plus efficace de procéder à une modification du code du plugin, qui est sous licence MIT. Le plugin original de Naginator (nous nommerons le nouveau naginator5k), a deux options, l'une permettant de relancer les runs à l'état instable de la même façon que les runs à l'état fail, l'autre permettant de ne relancer que si une *regex* est trouvée dans la sortie console. Sauf que cette dernière option recherche le texte non pas dans la sortie console de chaque run, mais la sortie globale du build. Celle-ci ne ressort que les changements d'états, donc non pertinents. De plus couplée à l'autre option, si la regex est trouvée, tous les runs à l'état d'échec ou instable sont relancés.

Après analyse du code source, seule une classe était en cause. Il a suffit de remplacer le bout de code de la recherche de *regex* pour qu'il analyse la sortie console de chaque run, et ajoute ceux où la chaîne était trouvée dans une liste de runs à relancer. Le repo Github de naginator5k est disponible ici.

3.3 Génération automatique de la configuration Jenkins

L'ensemble des services Grid'5000 sont gérés par Puppet [6], c'est un logiciel libre gérant les fichiers de configurations, basé sur des « recettes ». Par défaut, Jenkins ne se gère

5. **ie.** une configuration en particulier, comme le test d'OpenStack sur Nancy sur le Cluster Graphene
6. `https://puppetlabs.com/`

que via l'interface Web, ou son API. La première n'est pas viable pour une utilisation via Puppet, et la seconde devrait se faire par script faisant appel à l'API pour vérifier si le job existe avant de le créer. De plus l'API n'intègre que très peu de possibilités de configuration des jobs.

L'orientation a donc été faite sur un outil externe : Jenkins Job Builder [7]. Ce script Python permet de configurer Jenkins à partir d'un simple fichier YAML. Ce dernier a l'avantage d'être facilement lisible, maintenable et analysable, de plus c'est un simple fichier texte à gérer côté Puppet. Enfin il permet la gestion de certain plugins utilisés pour Grid'5000, comme les plugins Git (copie du script utilisé à partir du repo Git à chaque exécution) et Matrix Configuration (pour créer une matrice d'exécution à partir de deux listes).

En revanche Jenkins Job Builder n'intègre pas tous les plugins existants, il a donc été nécessaire de créer un outil permettant d'étendre ses capacités (cf 3.3.1). De plus, un fichier YAML est statique et ne s'adapte pas aux changements de Grid'5000, un outil pour mettre à jour ce fichier a dû être mis en place (cf 3.3.2)

3.3.1 Extension des possibilités de Jenkins Job Builder avec PostScriptJJB

Le second problème de Naginator était sa compatibilité seulement partielle avec Jenkins Job Builder (JJB). En effet, l'option permettant de ne relancer que la partie de la matrice ayant échoué n'est pas configurable via le fichier YAML. Le problème pouvant se représenter avec d'autre plugin, il a été décidé de créer un script Ruby générique pour ce genre de problème.

La configuration de chaque build est contenue dans un fichier XML, le principe a donc été de créer un parser XML permettant d'ajouter du texte à l'endroit souhaité et de « prévenir » Jenkins du changement. Ce qui donne ce principe de fonctionnement :

$\boxed{\text{YAML}} \Rightarrow \text{JJB} \Rightarrow \boxed{\text{XML}} \Rightarrow \text{PostScriptJJB} \Rightarrow \boxed{\text{Final XML}}$

Pour rester utilisable facilement, les seuls paramètres demandés pour l'utilisation du script sont : le nom du job pour lequel la configuration doit être modifiée, le fichier XML à inclure dans l'autre, le chemin XPath où l'inclure et s'il faut remplacer un éventuel fils du même nom déjà existant à cet endroit.

Pour prévenir Jenkins, l'API REST permet d'initier le rechargement de la configuration

7. `https://github.com/openstack-infra/jenkins-job-builder`

à partir d'un fichier sur le disque, un POST HTTP sur cette page avec le Gem RestClient suffit donc à prendre en compte les modifications.

3.3.2 Mise à jour automatique du fichier YAML avec AutoUpdateJJB

Le dernier point demandé était la mise à jour automatique des jobs en fonction de l'architecture de Jenkins. Chaque point de la configuration de l'architecture de Grid'5000 est présente dans son API, donc si l'on ajoute un cluster il sera ajouté dans l'API. Nous savons que Jenkins se base (indirectement) sur le fichier YAML pour créer la configuration de ses jobs. Le but est donc de relier l'API avec les listes présentes dans le fichier YAML.

Ce travail a été décomposé en deux parties. La première étape a été de créer un script (Ruby) capable de récupérer des informations de l'API relativement facilement, le but étant de créer des listes que l'on inclura par la suite dans le fichier YAML. Les gems Ruby permettent de récupérer les informations facilement, en revanche l'inconvénient de ce script est que si un utilisateur souhaite ajouter une nouvelle liste basée sur des informations différentes des autres, il faut des **bases** en Ruby et en algorithmique pour créer une nouvelle méthode capable de récupérer les bonnes informations.

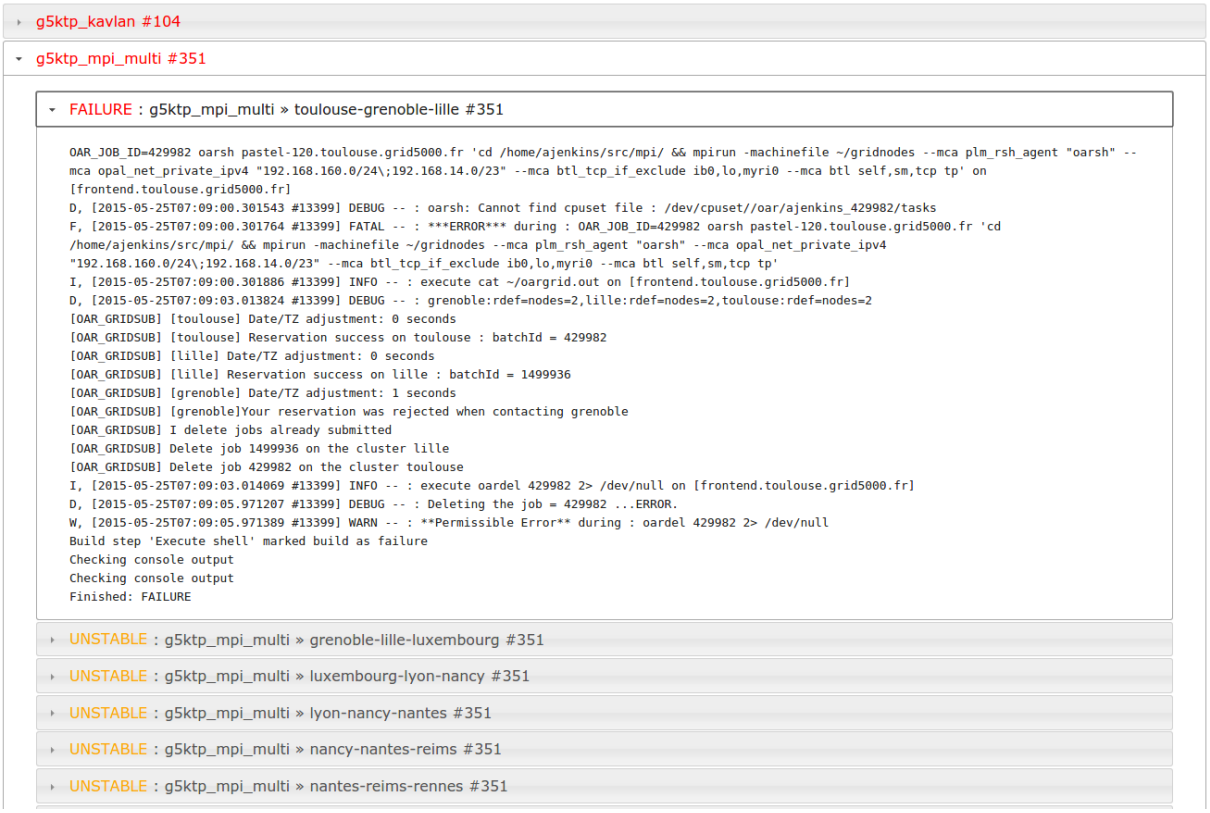
La deuxième partie est un *parser* YAML fonctionnant de la même manière que PostScriptJJB. C'est-à-dire qu'à partir d'une entrée donnée, il injecte du contenu à l'endroit souhaité dans le fichier YAML. L'entrée ici, est une liste retournée par le script de la première partie ainsi que le nom de la liste à mettre à jour. Avec une exécution hebdomadaire, les jobs sont de cette façon toujours à jour avant leur exécution au sein de Jenkins.

3.4 Interface web simplifiée pour consulter les résultats des tests

Il a été précisé plus haut que l'utilisation de Jenkins est laborieuse, même pour des réunions « seulement » hebdomadaires. En effet, il ne faut pas moins de cinq clics depuis la page d'accueil pour atteindre la page de sortie console, et trois à quatre clics pour atteindre la sortie console d'un autre run du même build. Le besoin de synthétiser ces pages s'est donc rapidement ressenti.

Le principe est de récupérer les informations de l'API Jenkins pour obtenir l'état des derniers builds de chaque job, et de les rendre présentables en un à deux clics. On récupère donc uniquement les informations utiles, à savoir le nom des jobs qui ne sont pas des succès

et leur état (les succès ne sont pas pertinents lors des réunions hebdomadaires), le nom et l'état des builds de ce job qui ne sont pas des succès et leur sortie console. Une étape supplémentaire a été ajoutée, permettant d'analyser la sortie console et de ne ressortir que le contenu pertinent, c'est à dire de l'erreur jusqu'à la fin du script. Cette page est générée chaque mardi à 10h, soit une heure avant la réunion. Son rendu peut être apprécié ci-dessous :



4 Conclusion

Pour conclure, l'ensemble des tests mis en place permettent de tester un large ensemble des possibilités de Grid'5000, ils permettent d'être facilement créés ou édités grâce à la bibliothèque mise en place, et facilement automatisés grâce aux outils rattachés à Jenkins et Jenkins Job Builder. Après plusieurs semaines de test, il semble que les tests ne relèvent pas de faux-positifs et ont permis de signaler plusieurs bugs et failles dans la plate-forme jusqu'alors non découverts. L'automatisation permet de faire gagner beaucoup de temps aux administrateurs et augmente également la fiabilité des tests en restant à jour.

L'intégration continue permet de compléter le manque des autres outils de supervision qui ne surveillent que l'état des serveurs et services sans approfondir sur leur bon fonctionnement.