Rapport de stage

1 avril – 31 septembre 2013

Isolation d'applications en production à l'aide de conteneurs Linux au sein d'une *Plateform as a Service*

Étudiant
Robin Monjo - SI5 - parcours Architecture Logicielle (AL)

Maître de stage
Romain Pechayre

Tuteur
Sébastien Mosser

Entreprise
Applidget
Paris - 75018 - France

Table des matières

Table des figures

Tables des tableaux

Remerciements

Je tiens tout d'abord à remercier Applidget pour m'avoir accueilli afin d'effectuer mon stage. Je remercie plus particulièrement Tristan Verdier, CEO, et Romain Pechayre, CTO, qui était également mon maître de stage et auprès duquel j'ai beaucoup appris.

Je remercie également mon tuteur, Sébastien Mosser, un professeur important pour moi lors de ma dernière année d'études à Polytech' Nice-Sophia.

Enfin je remercie tout le corps enseignant de Polytech' Nice-Sophia pour leur formation.

Conventions

De par le sujet qu'il traite, ce document contient des anglicismes qui ont plus de sens que leur traduction littérale. Ces anglicismes sont écrits en *italique.* Les extraits de codes, commandes ou artefacts informatiques sont encadrés. Les références sont indiquées par des notes de pied de page ([1]) qui dirigent vers la source (une page Internet digne de confiance). Les termes écrits en vert sont définis dans le glossaire.

Introduction

Ce document présente le travail effectué lors de mon stage de fin d'études, stage qui a pour but de mettre en pratique les connaissances acquises au cours de ma formation d'ingénieur en sciences informatiques à l'école Politech' Nice-Sophia (université de Nice). Pour ce stage d'avril à septembre 2013, j'ai rejoint une startup située sur Paris, Applidget.

La société Applidget est une société présente sur deux marchés : elle édite une solution logicielle pour l'émargement et le contrôle d'accès de participants à un événement et propose ses services en tant qu'agence de développement pour des projets web et mobile. J'ai intégré une équipe de développement de huit personnes en tant qu'ingénieur recherche et développement. L'équipe est dirigée par Romain Pechayre, CTO (*Chief Technology Officer*), mon maître de stage. Mon stage s'insère dans le domaine du *cloud computing*

L'émergence du *cloud computing* depuis ces dernières années, est en train de changer la manière dont les ressources informatiques sont utilisées. Ces ressources, qu'elles soient physiques (mémoire, stockage, *CPU*) ou logicielles sont de plus en plus *consommées* comme des services. Entreprises et particuliers peuvent ainsi profiter des infrastructures de grands groupes spécialisés pour héberger leurs données et leurs applications. Le *cloud* permet une grande flexibilité (souvent appelée élasticité dans ce domaine) et une importante maîtrise des coûts :

- Flexibilité : les ressources sont accessibles à la demande. Cela permet de répondre aux problèmes de passage à l'échelle auxquels certains services sont soumis. *Netfix* par exemple, vendeur de vidéos en *streaming* aux *USA* utilise le *cloud* pour répondre aux différents pics d'utilisation de son service afin de respecter la qualité de service vendue à ses clients[1]. Cette flexibilité permet un passage à l'échelle aussi bien pour répondre à la « sous-utilisation » (ne pas payer les ressources non nécessaires) qu'à la « sur-utilisation ».
- Maîtrise des coûts : la flexibilité énoncée précédemment permet une maîtrise des coûts. De plus, les entreprises n'ont plus besoin de mettre en place des *data-centers* et tous les coûts que cela implique (achat du matériel, mise en place, maintenance, électricité, surface immobilière, etc.). Le client ne paye que ce qu'il consomme (*pay-as-you-go*).

[1] http://www.forbes.com/sites/joemckendrick/2012/02/22/6-shining-examples-of-cloud-computing-in-action/

Le *cloud computing* se présente comme la grosse évolution dans le domaine de l'informatique moderne. Beaucoup d'entreprises ont revu leur *business model* en le basant sur le *cloud.* C'est notamment le cas d'Adobe qui a annoncée en mai 2013 que sa très populaire Creative Suite sera désormais *consommée* comme un service[2].

Le *cloud* introduit les notions d'*Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS) et *Software as a Service* (SaaS). De manière simplifiée, chacune de ces notions représente une couche du *cloud computing* : l'IaaS va fournir les ressources physiques ainsi que le système d'exploitation, la PaaS va fournir un ensemble complet d'outils pour déployer des applications dans le *cloud,* et le SaaS fournit le produit fini, généralement une application qui sera accessible via Internet.

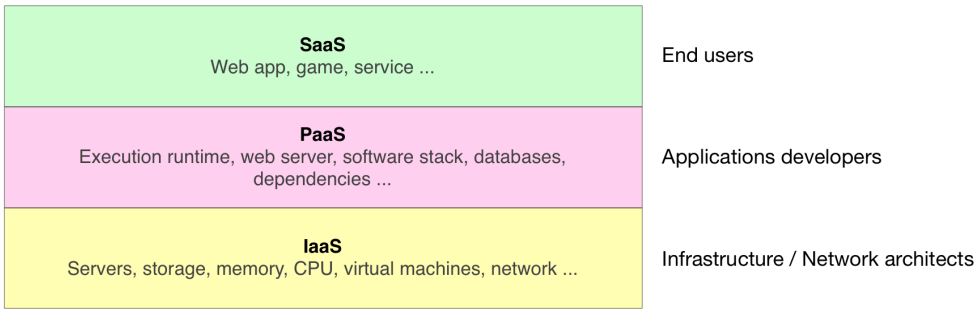

Figure 1: Les différentes couches du cloud

La figure 1 montre ces différentes couches et leur complémentarité. Le *cloud* peut être public, privé ou hybride. Un *cloud* publique est accessible à tout le monde. Les sociétés qui développent une IaaS public ont un *business model* basé sur la location de ressources matérielles. Un *cloud* privé est géré en interne par des sociétés étant dotées d'un ou plusieurs *data centers.* Les ressources sont exclusivement réservées aux besoins de la société. Un *cloud* hybride est un mélange des deux. Une société peut en effet vouloir garder des données sensibles au sein de son propre *data center* tout en profitant d'un *cloud* public pour augmenter la puissance de celui-ci.

Applidget utilise les technologies du *Cloud Computing* pour héberger et rendre accessibles ses services. Ainsi, la société gère une dizaine d'applications au quotidien sur des serveurs dans le *cloud.* Elle y héberge notamment son produit phare, Mobicheckin[3], une solution SaaS d'émargement et de contrôle d'accès à destination des organisateurs d'événements. Les autres applications sont principalement des *backends* pour des applications mobiles et des sites internet développés pour des clients ainsi que certains outils utilisés en interne. Comme toutes les sociétés qui utilisent le *cloud*, Applidget est confronté à des problématiques de sécurité, de haute disponibilité, de passage à l'échelle, de *monitoring*, de sauvegardes, etc. Ma mission au sein de la société était de travailler sur leur PaaS, développé en interne. L'intitulé exacte de cette mission était : « *Isolation d'applications en production à l'aide de conteneurs Linux au sein d'une Plateform as a Service* ». Le stage

[2] http://news.investors.com/technology/050613-654960-adobe-systems-allin-on-cloud-with-creative-cloud.htm
[3] http://www.mobicheckin.com/

proposé s'insère dans l'activité de recherche et développement de la société, j'ai travaillé seul sur ce sujet, en collaboration avec mon maître de stage qui me donnait les directives et avec qui je prenais les décisions.

Ce rapport précise dans un premier temps le sujet du stage en se concentrant sur les enjeux techniques et en détaillant le concept de *Platform as a Service* (PaaS). J'évoquerai aussi l'existant en terme d'architecture et de logiciel au sein de l'entreprise. Je serai alors capable de préciser clairement les problématiques auxquelles j'ai dû faire face lors de ma mission. Dans une deuxième partie, je me focaliserai sur le travail que j'ai accompli, en discutant des choix que j'ai du faire et des difficultés rencontrées. Enfin dans une dernière partie, je parlerai de l'organisation du travail lors de mon stage, en termes de planning et de méthodologie.

I – Description du travail proposé

I.1 – Sujet de stage

Afin de préciser les enjeux de ma mission, la partie suivante détaille le sujet de stage en se concentrant sur les trois mots-clés essentiels du titre : **Isolation d'applications** en production à l'aide de **conteneurs Linux** au sein d'une ***Plateform as a Service***.

I.1.1 – Isolation d'applications

L'isolation d'applications tournant en production est devenue une pratique courante. Cela permet de contrôler les ressources utilisées par chacune d'entre elles (*CPU*, mémoire, bande passante réseau), mais également d'empêcher un processus d'accéder à des informations en concernant un autre (variables d'environnement, dossier de travail, *child processes*).
L'isolation répond donc à une question de sécurité. En effet, isoler un processus permet de limiter son accès à certaines ressources proposées par le système d'exploitation. Le terme de *sandbox* (bac à sable) réfère à cette pratique. Utilisé notamment dans les *app store* des systèmes mobiles iOS et Android et la machine virtuelle Java (*Java Virtual Machine* – JVM), le *sandboxing* fait tourner une application dans un bac à sable, c'est-à-dire que ses appels aux fonctionnalités du système sont contrôlés. Une application tournant dans un tel environnement ne pourra, par exemple, pas accéder à certains périphériques, ou accéder à certaines parties du système de fichier du système d'exploitation.

D'un point de vue plus architectural, outre l'aspect de sécurité, une application isolée est une application plus facilement « maîtrisable » dans le sens où l'on connaît et contrôle son empreinte sur le système. Prenons l'exemple d'une application Java : une application Java est isolée par la JVM. Cette isolation permet à cette application d'être exécutée sur n'importe quel système étant doté de la JVM. En montant encore d'un niveau d'abstraction, on peut imaginer isoler une application au niveau du système d'exploitation via la virtualisation. Une telle isolation rend l'application totalement auto-suffisante, car déjà fournie avec un système configuré (bibliothèques, langages, environnement etc.). À l'heure du *cloud computing,* où les systèmes sont de plus en plus distribués et les contraintes de passage à l'échelle sont

de plus en plus fortes, on voit rapidement les avantages : duplication ou réduction du nombre d'instance de l'application sur des serveurs différents. L'application devient une « brique » que l'on peut dupliquer et déplacer.

En se plaçant dans une optique plus orientée business que technique, le fait de pouvoir contrôler les ressources matérielles utilisées par une application est primordial dans un *cloud* public qui base son *business model* sur la location de ressources.

La virtualisation est donc une bonne réponse à l'isolation d'applications, notamment dans le contexte du *cloud computing.* En revanche, isoler une unique application au sein d'un système d'exploitation virtualisé est une solution coûteuse en termes de ressources matérielles. La technologie LXC (*LinuX Container*) est une réponse à ce problème.

I.1.2 – Conteneurs Linux (LXC)

Les conteneurs linux peuvent être vus comme une forme de virtualisation. La virtualisation permet de faire cohabiter plusieurs systèmes d'exploitation ou applications sur une même machine. Les environnements virtualisés sont isolés les uns des autres et partagent uniquement le matériel (*hardware*). La virtualisation classique repose sur un hyperviseur, un logiciel qui va simuler les *Application Programming Interface* (API) d'un système d'exploitation (accès au *CPU*, à la mémoire etc.) et les rendre disponibles à des systèmes d'exploitation virtualisés. L'hyperviseur agit comme une couche intermédiaire entre le matériel et le système virtualisé. On distingue deux types d'hyperviseurs :

- le type 1 : l'hyperviseur est le seul système à avoir accès au matériel. Il supervise une ou plusieurs machines virtuelles et est le premier système lancé après le chargeur de démarrage (*bootloader*).
- le type 2 : l'hyperviseur tourne au sein d'un système d'exploitation, le système hôte. Il supervise une ou plusieurs machines virtuelles, appelées systèmes invités. Il est lancé par le système hôte.

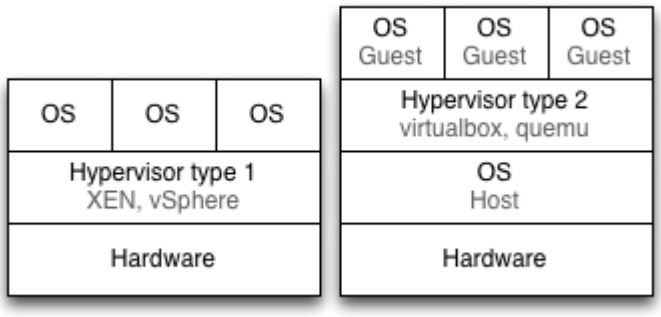

Figure 2 : Les différents types d'hyperviseurs

Les conteneurs *LXC (LinuX Container)* approchent le problème de la virtualisation de manière différente. Aucun hyperviseur n'est nécessaire, les conteneurs partagent le même noyau (Linux) que le système hôte. Cette technologie est techniquement possible depuis la

version 2.6.29 du noyau (mars 2009). LXC ne fournit pas une machine virtuelle, mais fournit plutôt un environnement virtuel qui a son propre espace de processus. LXC s'appuie sur les groupes de contrôle (*control groups* ou *cgroups*) et sur les *namespaces* du noyau Linux[4].

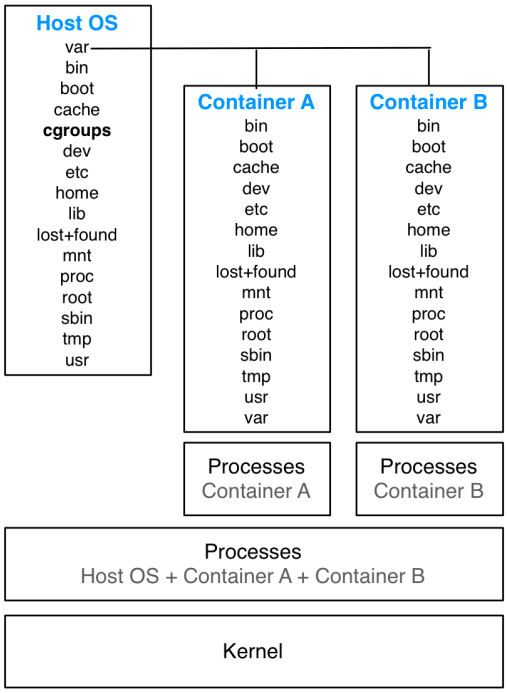

Figure 3 : Schémas conceptuel d'un conteneur

La figure 3 est un schéma conceptuel des conteneurs Linux. Les processus des conteneurs A et B sont complètement isolés, comme s'ils étaient exécutés dans un environnement virtualisé classique. Chaque conteneur possède sa propre hiérarchie de système de fichier.

Les conteneurs Linux fournissent ainsi une virtualisation au niveau du système d'exploitation (*Operating System level virtualisation*). Ils permettent une virtualisation ultra légère bien plus rapide que les autres technologies existantes (pas d'hyperviseurs, accès directe au matériel) tout en fournissant les mêmes avantages qu'un système virtualisé (isolation, plusieurs systèmes sur une même machine, contrôle des ressources, sauvegarde et clonage rapides).

Pour revenir à la problématique d'isolation d'applications évoquée précédemment, on voit que les conteneurs Linux semblent être une réponse idéale. Il est possible d' « emballer » une application à l'intérieur d'un conteneur, dans un environnement auto-suffisant et donc très portable, qui assure que l'application se comportera de la même manière sur toutes les machines (locales, en développement ou serveur en production). Dans un besoin de passage à l'échelle, ou plus simplement de gestion d'applications dans une infrastructure, un conteneur peut être dupliqué/déplacé d'un serveur (Linux) à l'autre en réduisant le temps de mise en place à son minimum. Par analogie au « *write once run everywhere* » du langage Java, on peut parler de « *build once, run everywhere* » et « *configure once run everywhere* ». Les développeurs peuvent se concentrer sur leur application sans se soucier

[4] http://lxc.sourceforge.net/

de l'infrastructure sous-jacente, les administrateurs peuvent déployer des conteneurs sans se soucier des technologies qu'ils embarquent.

I.1.3 – Platform as a Service - PaaS

La figure 1 montre la PaaS comme la couche intermédiaire du *cloud computing*. De manière très vulgarisée, une PaaS est chargée de prendre le code source d'une application, de le rendre exécutable et de déployer l'application sur un serveur. La PaaS est donc l'intermédiaire entre l'état de développement et l'état de production. On distingue trois étapes centrales dans le fonctionnement d'une PaaS :

1. récupération du code source
2. transformer le code source en une application exécutable
3. lancer l'application sur un serveur en production

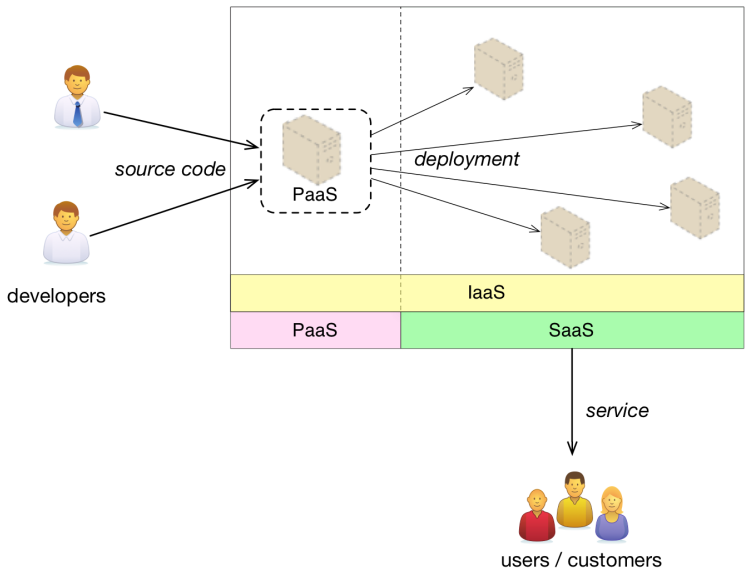

Figure 4 : Rôle d'une Plateform as a Service

La PaaS est donc un intermédiaire entre le développeur et la mise en production d'une application. Les PaaS s'intègrent très bien dans des environnements de développement agiles, car elles permettent un déploiement continu. Ainsi, la mise en production d'une application n'est plus une opération longue (pouvant parfois donner lieux à une interruption de services), ce qui permet aux développeurs d'intégrer régulièrement des corrections de bugs ou de nouvelles fonctionnalités, sans être contraints par un cycle de mise à jour.
Bien que le rôle premier d'une PaaS soit le déploiement, les PaaS modernes permettent de complètement gérer une infrastructure dans un *Cloud.* Elles offrent notamment le monitoring et l'accès aux logs des applications, mais aussi facilite le passage à l'échelle et la haute disponibilité, en permettant de définir le nombre d'instance d'une application à un instant donné.

I.1.4 – Enjeux et motivations

L'isolation d'applications en production à l'aide de **conteneurs Linux** au sein d'une ***Plateform as a Service*** est donc un enjeu majeur pour des raisons de sécurité (d'autant plus que dans le cas d'une PaaS public, l'application d'une société X peut-être exécutée sur le même serveur virtuel que l'application d'une société Y) et de maîtrise des ressources.

Dans un environnement moderne et agile, la virtualisation n'est pas la réponse optimale à cette isolation pour plusieurs raisons. Les environnements de production sont souvent eux-mêmes des serveurs virtualisés (IaaS) donc il peut ne pas être possible de faire tourner des hyperviseurs sur ces machines. Par ailleurs, une machine virtuelle consomme à elle seule une part importante des ressources et prend surtout du temps à démarrer :

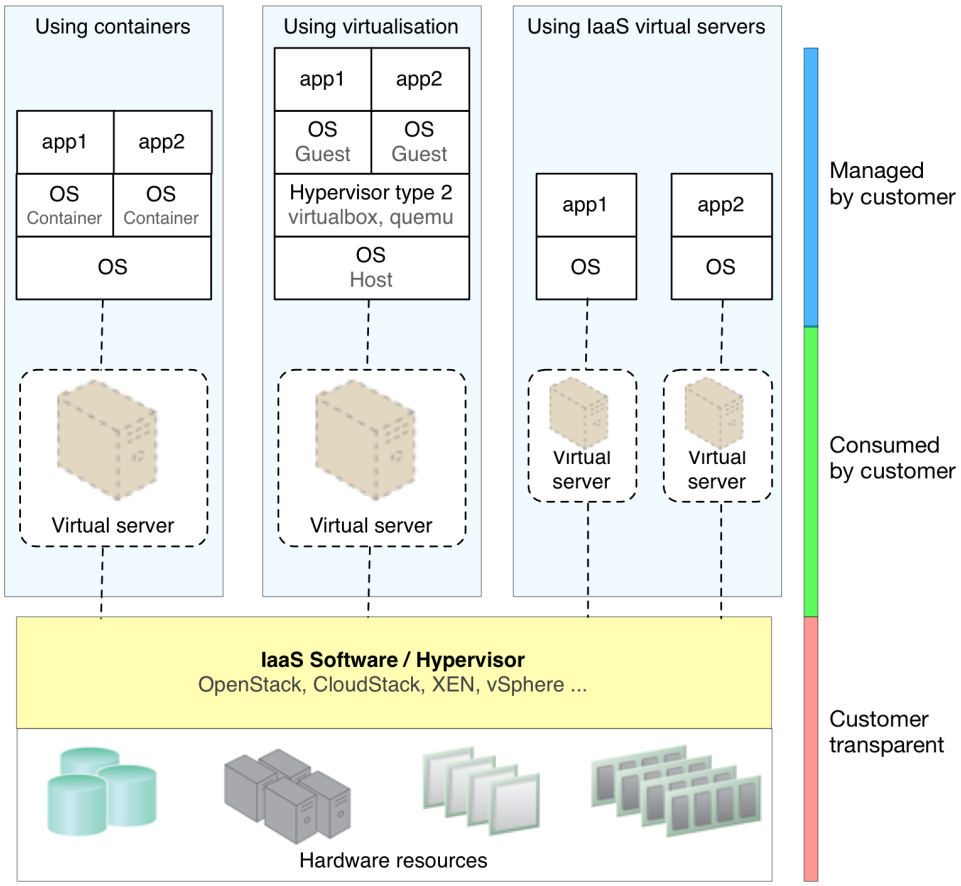

Figure 5 : Solutions de virtualisation dans une IaaS

La figure 5 montre trois choix possibles pour exposer deux applications isolées au niveau du système d'exploitation, en utilisant une IaaS :

- à droite, solution 1, la solution d'utiliser les machines virtuelles fournies par l'*IaaS.* Cela nécessite donc l'utilisation de deux serveurs virtuels distincts. Solution coûteuse (les serveurs virtuels sont facturés) et probable sous-utilisation des ressources

(mauvaise optimisation des coûts). Par ailleurs, cette solution est peu agile dans le sens ou une machine virtuelle prend du temps à démarrer (environ une minute).

- au milieu, solution 2, la solution d'utiliser la virtualisation (avec un hyperviseur de type 2). Répond aux problèmes de la solution 1 (hormis sur le temps de démarrage d'une machine virtuelle) mais les ressources sont gâchées du fait des problèmes de performances induits par ce type de virtualisation.
- à gauche, solution 3, l'utilisation de conteneurs *LXC*. Répond aux points négatifs de la solution 1 et 2. La suite du document montrera qu'il est possible de démarrer un conteneur linux quasi instantanément.

La technologie LXC (conteneurs Linux) fournit une isolation au niveau du système d'exploitation beaucoup plus légère et rapide que les technologies de virtualisation classiques.
L'objectif du stage est donc d'intégrer cette technologie au sein de la *Plateforme as a Service* développée en interne par Applidget.

I.2 – État de l'art des PaaS

Avant de discuter précisément du travail réalisé, il est important de faire un état de l'art des *Plateformes as a Service* du marché, afin de comprendre leur fonctionnement, leur architecture et les choix techniques qu'elles ont entrainés. Deux acteurs importants du marché sont les Américains Heroku (propriété de salesforce.com) et dotCloud (siège social en Californie, créé par des Français). Ces deux acteurs sont relativement portés sur l'open-source et documentent le fonctionnement interne de leur plateforme.

I.2.1 – Heroku

Afin de détailler le fonctionnement de Heroku, je vais reprendre les trois étapes centrales du fonctionnement d'une PaaS cités précédemment : (1) récupération du code source ; (2) transformer le code source en une application exécutable ; (3) lancer l'application sur un serveur en production.

I.2.1.1 – Récupération du code source

Les PaaS sont des outils de déploiement. Le point d'entrée de tout déploiements est le code source de l'application à déployer. Pour récupérer ce code et commencer la mise en production, Heroku (et tous les concurrents à ma connaissance) utilisent les fonctionnalités du gestionnaire de version *Git*[5].

[5] http://git-scm.com/

La fonctionnalité de *Git* intéressante dans ce cas est la manière dont sont gérés les dépôts distants (*repository)* :

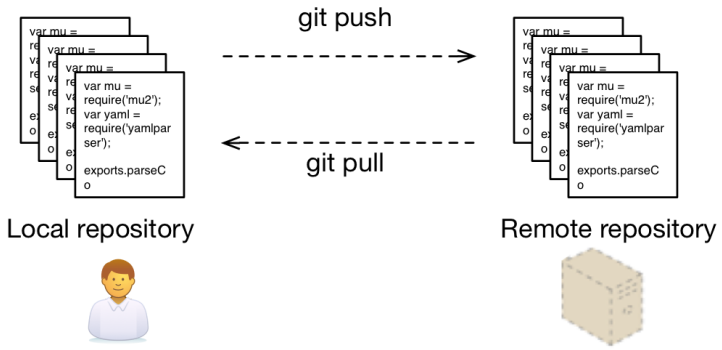

Figure 6 : Git dépôts distants

Un dépôt distant peut être créé sur un serveur personnel, ou bien sur un service spécialisé tel que github.com[6] ou Heroku. Git est très flexible car il permet d'activer des scripts personnels avant ou à l'issue d'une certaine action. Heroku utilise ces scripts, appelés *githooks* pour déterminer quand une action de *push* se termine. À ce moment, la plateforme sait qu'une nouvelle version du code source vient d'arriver (elle y a accès via le dépôt) et qu'une action de déploiement doit être effectuée.
En utilisant Git, Heroku s'intègre parfaitement dans le *workflow* des développeurs et favorise grandement le déploiement continu. En effet, en une commande : `git push heroku master`, l'application est mise en production.

I.2.1.2 – Transformer le code source en une application exécutable

Posséder uniquement le code source est rarement suffisant pour être capable d'exécuter une application. En effet, une application, en plus de son code, est un ensemble de dépendances : environnement d'exécution ; bibliothèques ou modules tierces ; logiciels tierces, etc. Cet ensemble de dépendances est appelé la pile logicielle. La gestion des dépendances est l'une des grosses problématiques à laquelle une PaaS doit faire face. Ainsi, une PaaS ne peut pas déployer tout type d'applications. Heroku supporte officiellement les langages : Ruby, Nodejs, Java, Python, Clojure et Scala. Ces derniers ont la particularité d'avoir un mécanisme de gestion de dépendances qui permet à Heroku de fournir une pile logicielle automatiquement :

Langage	Environnement d'exécution	Description des dépendances	Programme de gestion des dépendances
Ruby	Ruby	Gemfile	Bundler
Nodejs	node	package.json	NPM (Node Package Manager)
Java	JVM	pom.xml	Maven

Table 1 : Langages et leurs outils

[6] https://github.com/

Afin de résoudre cette problématique, Heroku a créé les *buildpacks.* Les *buildpacks* sont des modules qui respectent une simple API permettant de compiler une application. Il existe un *buildpack* pour chaque langage cité précédemment.

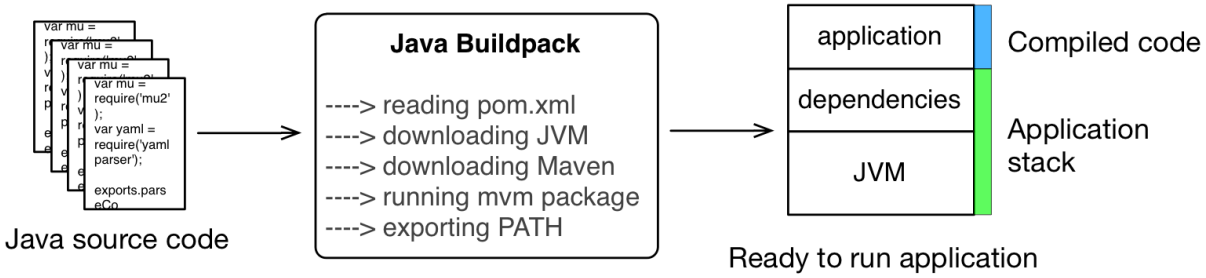

Figure 7 : Fonctionnement du buildpack Java

La figure 7 illustre le fonctionnement du *buildpack* Java. Il commence par lire la description des dépendances. Il va ensuite télécharger la JVM et Maven en respectant la version spécifiée dans le fichier pom.xml. Une fois la JVM et Maven téléchargés, il va pouvoir utiliser la commande Maven `mvn package` afin de compiler et résoudre toutes les dépendances. Le *buildpack* fournit en sortie un dossier contenant « tout » ce qui est nécessaire à l'exécution du code source donné en entrée.
Les *buildpacks* de Heroku sont *open sources* et il est possible d'utiliser des *buildpacks* non officiels tant que ceux-ci respectent l'API définie par Heroku[7]. Ainsi, bien que non supportés officiellement, il est possible de déployer des applications écrites dans d'autres langages que ceux cités précédemment.

I.2.1.3 – Lancer l'application sur un serveur en production

Confronté aux problématiques d'isolation d'application évoquée dans la partie I.1.1, Heroku utilise les conteneurs LXC[8] sous la technologie qu'elle appelle Dyno. Un Dyno Heroku est un conteneur Linux responsable d'exécuter l'un des processus d'une application.
Les processus doivent être spécifiés dans un fichier `Procfile`. Chaque processus est défini par son type et la commande nécessaire pour le lancer. Pour reprendre l'exemple d'une application Java, un `Procfile` (avec un seul processus de type web) pourrait ressembler à :

```
web:    java -cp target/classes:target/dependency/* HelloWorld
```

Heroku va donc lire le `Procfile` défini par le développeur et pour chacun des processus, créer un Dyno sur un serveur de production, y transférer l'application compilée par le *buildpack* et exécuter la commande associée.

Bien qu'une *Plateforme as a Service* se veuille transparente pour le développeur, on a vu que les applications doivent suivre certaines règles pour être correctement déployées

[7] https://devcenter.heroku.com/articles/buildpack-api#buildpack-apii
[8] https://devcenter.heroku.com/articles/dynos

(notamment la présence d'un Procfile et d'un fichier de description des dépendances). Ces règles ont été écrites par Adam Wiggins, acteur majeur de Heroku, et sont recensées dans le document : *The Twelve-Factor App*[9]. L'introduction de ce document présente très bien ce pour quoi il a été rédigé :

> *In the modern era, software is commonly delivered as a service: called web apps, or software-as-a-service. The twelve-factor app is a methodology for building software-as-a-service apps that:*
>
> - *Use declarative formats for setup automation, to minimize time and cost for new developers joining the project;*
> - *Have a clean contract with the underlying operating system, offering maximum portability between execution environments;*
> - *Are suitable for deployment on modern cloud platforms, obviating the need for servers and systems administration;*
> - *Minimize divergence between development and production, enabling continuous deployment for maximum agility;*
> - *And can scale up without significant changes to tooling, architecture, or development practices.*

I.2.2 – dotCloud et docker

La description de Heroku résume bien les enjeux techniques auxquelles une PaaS doit faire face. Lors de mon stage, je me suis aussi intéressé à dotCloud, autre acteur important du marché des PaaS. Comme Heroku, dotCloud utilise aussi les conteneurs LXC pour isoler les applications. Récemment (mars 2013), dotCloud a rendu open-source Docker[10], une surcouche de la technologie LXC adaptée aux besoins des PaaS. Docker facilite et automatise l'utilisation de la technologie LXC. Ce projet est très intéressant dans le cadre de mon stage car il me permet de comprendre comment un professionnel des PaaS utilise la technologie LXC. Pour faire un parallèle avec Heroku, Docker peut être vu comme le module qui gère les Dynos. Je reviendrai plus en détail sur Docker qui a inspiré plusieurs choix techniques lors de mon travail.

Les grands éditeurs de *Platforms as a Service* profitent donc de la puissance des conteneurs. Ces derniers leur apportent à la fois sécurité, gestions fine de leurs ressources (une PaaS public loue ces ressources) et facilitent la gestion de leur infrastructure (duplication d'application aisée à la demande de l'utilisateur, sauvegarde).

I.3 – État de l'art chez Applidget

Afin de préciser totalement le contexte, il est important de connaître l'infrastructure d'Applidget dans le *cloud* ainsi que la plateforme existante sur laquelle ma mission se concentre.

[9] http://www.12factor.net/
[10] http://www.docker.io/

I.3.1 – Infrastructure dans le *cloud*

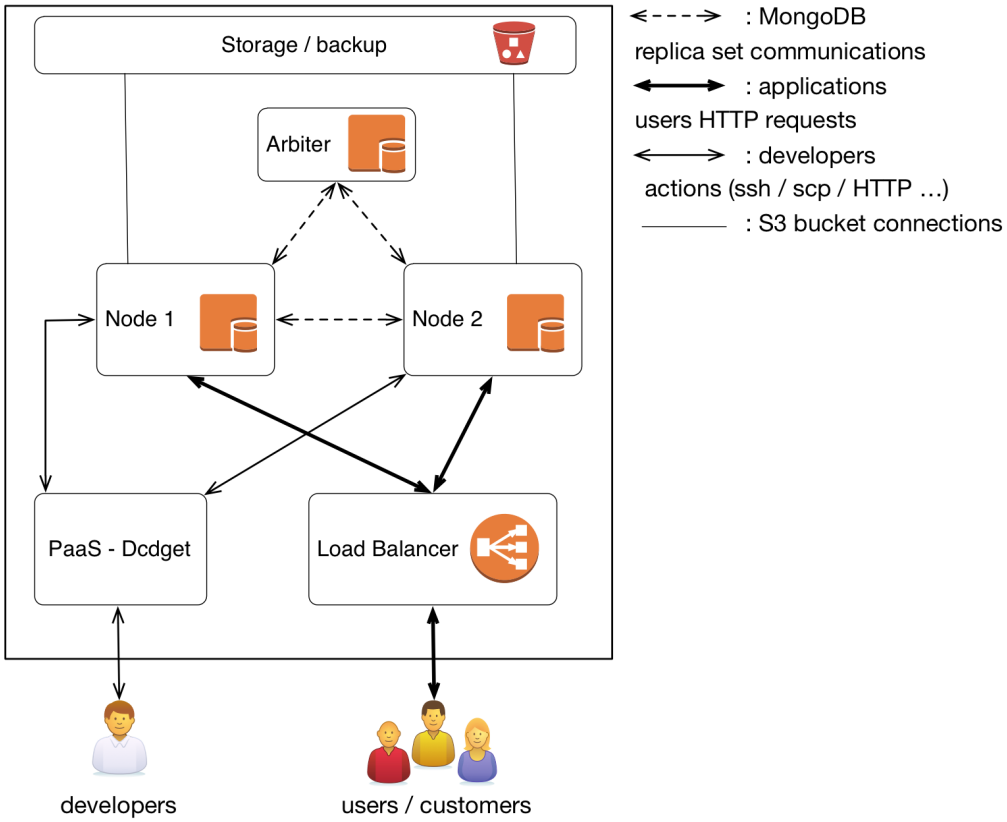

Figure 8 : Infrastructure d'Applidget dans le cloud d'Amazon

La figure 5 représente l'infrastructure d'Applidget dans le *cloud* d'Amazon (*Amazon Web Service* - AWS). Un *load balancer* équilibre la charge sur les deux serveurs de productions (*node 1* et *node 2*). Les serveurs de production hébergent à la fois l'applicatif et les bases de données. Un serveur supplémentaire (*Arbiter*) est présent afin d'assurer le bon fonctionnement de la réplication des bases de données MongoDB[11] (la seule technologie utilisée par la société dans ce domaine). Une sauvegarde des bases de données effectuée toutes les nuits est stockée sur le service de stockage d'AWS, S3. Les applications en production accèdent aussi à ce service de stockage. La PaaS d'Applidget, Dcdget est hébergée sur son propre serveur virtuel. *Remarque* : Cette architecture répond aux besoins de la société bien que certaines bonnes pratiques n'aient pas été respectées, notamment le fait de séparer complètement la couche applicative de la couche persistance des données. Le jour ou le besoin d'un troisième serveur de production apparaîtra, Applidget devra revoir cette architecture.

La mission de mon stage se concentre principalement sur le nœud Dcdget, décrit en détails dans la partie suivante.

I.3.2 – La Plateform as a Service Dcdget

[11] http://docs.mongodb.org/manual/replication/

La *PaaS* développée en interne par Applidget, *Dcdget,* s'inspire de Heroku et suit les bonnes pratiques énoncées dans le *Twelve-Factors App.* Applidget utilisait Heroku à ses débuts et a senti le besoin de développer sa propre plateforme pour plusieurs raisons :

- les serveurs de Heroku sont uniquement localisés en Amérique (bientôt en Europe[12]) et les clients sont majoritairement localisés en Europe. Avoir des serveurs de productions en Europe permet ainsi de réduire la latence.
- Heroku est hébergé sur le cloud d'Amazon. Un utilisateur de Heroku est ainsi soumis à deux *Service Level Agreement* (SLA) différents : directement au SLA de Heroku et indirectement à celui d'Amazon *AWS*.
- l'utilisation des services d'Amazon sans passer par Heroku permet de réduire les coûts d'exploitation.
- Gain de flexibilité étant donné que le code source de la plateforme est totalement maîtrisé.

Dcdget est devenue une pièce centrale de la société, et est utilisée pour déployer une dizaine d'applications.

La plateforme est décomposée en plusieurs briques logicielles :

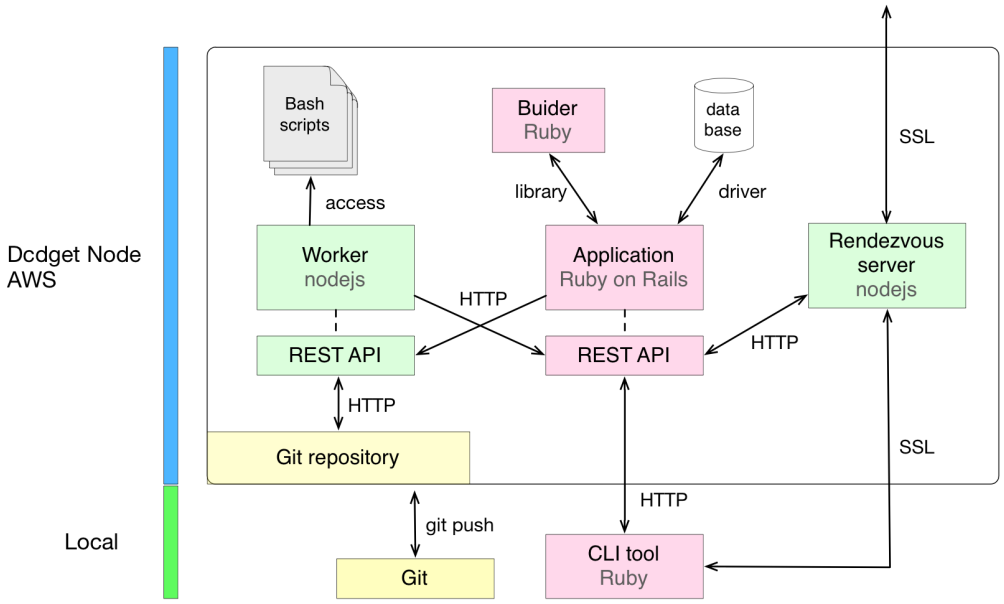

Figure 9 : La PaaS d'Applidget, Dcdget

- *CLI tool, dcdget client*, un programme en ligne de commande, écrit en Ruby qui permet de piloter la plateforme depuis une machine locale. C'est le client Dcdget.
- *Application,* le corps de la plateforme. Elle contient le modèle de données et gère la persistance via une base de données. Elle accède aussi à un programme écrit en Ruby capable de compiler les applications.
- *Worker,* un programme en Nodejs qui est responsable de gérer les opérations longues de manière asynchrone (la force de la technologie Nodejs).
- Un ensemble de scripts bash qui sont exécutés à distance sur les serveurs en production par le *worker.*

[12] https://blog.heroku.com/archives/2013/4/24/europe-region

- Un dépôt Git qui reçoit le code source de l'application (lors d'un *git push*) et informe le *worker* (en utilisant les *githooks*[13]) qu'une action de déploiement doit être effectué (à la manière de Heroku).
- *Rendezvous server*, un module qui permet d'exécuter des commandes sur les serveurs de production directement via *CLI tool.*
- Ps-dock, un outil open-source développé par Applidget, pas représenté sur ce schéma, il est utilisé du côté des serveurs de production. Il englobe le lancement des processus des applications et informe la plateforme de l'état de ces processus.

En l'état, les applications ne sont pas isolées. Elles sont installées dans un dossier dédié, à la racine du système de fichiers des serveurs de production. La plateforme a ses points faibles et ses points forts :

Points faibles	Points fort
Très peu de documentation (que ce soit sur le code source, l'architecture ou le fonctionnement de la plateforme)	Choix de technologies appropriés (API REST, nodejs pour les traitements asynchrones)
Code peu maintenu (beaucoup de codes plus utilisé toujours présents, code parfois très peu lisible, etc.)	Elle remplit son rôle sans bug bloquant
Aucun environnement de développement présent, la plateforme fonctionne uniquement en production	Fonctionnement inspiré de Heroku et respecte au maximum les règles évoquées dans le *12 Factors App*

Table 2 : Forces et faiblesses de Dcdget

Le travail de ma mission se concentre sur la prise en charge de la technologie LXC par cette plateforme. La partie qui suit détaille les points importants du travail que j'ai effectué.

[13] https://www.kernel.org/pub/software/scm/git/docs/githooks.html

II – Travail réalisé

La partie précédente présente le contexte et le travail effectué en termes de recherche sur l'état de l'art et la compréhension du domaine. Dans la partie qui suit, je me concentre sur le travail réalisé spécifiquement sur la plateforme.

II.1 – Environnement et outils

II.1.1 – Environnement de développement

La première étape de ma mission était de faire fonctionner Dcdget dans un environnement de développement. Créer un environnement de développement était une étape essentielle afin :

- de ne pas directement travailler sur la plateforme utilisée en production et d'intégrer des erreurs la rendant inutilisable.
- de permettre aux développeurs de rapidement avoir tous les outils pour travailler.

Il est important qu'un environnement de développement ressemble le plus possible à un environnement de production. Cela permet de retrouver un comportement identique entre les deux environnements. Il est aussi important de pouvoir automatiser la mise en place d'un environnement de développement, car celle-ci peut s'avérer laborieuse (installation des dépendances logicielles, respect des versions, exportation des variables d'environnements etc.). Réunir ces deux points est un gain de temps et de productivité pour les développeurs, qui ont rapidement tous les outils pour travailler, et permet de réduire la durée de la mise en production.

L'environnement de Dcdget requiert la présence d'au moins trois machines (voir figure 8) : une machine locale, ou l'outil en ligne de commande Dcdget (dcdget client) est installé ; une machine qui héberge la plateforme ; une ou plusieurs machines qui simulent les serveurs de production. Il est donc nécessaire de recourir à la virtualisation pour simuler ces différents nœuds. Afin d'automatiser la création de machines virtuelles, j'ai utilisé le logiciel Vagrant[14].

Vagrant est un gestionnaire de machines virtuelles qui s'adresse aux développeurs pour répondre à la problématique d'automatisation de mise en place d'environnements de développements. Cet outil utilise des hyperviseurs tel que VirtualBox[15] ou VMWare[16]. Son fonctionnement est basé autour d'un unique fichier : un Vagrantfile. Ce fichier contient des instructions écrites dans le DSL (*Domain Specific Language*) de Vagrant qui permet notamment de définir le système d'exploitation à utiliser et d'automatiser le provisionnement de ce système à l'aide de scripts. Autre fonctionnalité majeure, le dossier contenant le Vagrantfile est synchronisé sur la machine virtuelle. On peut donc travailler sur le code localement et l'exécuter dans l'environnement virtualisé. Concrètement, un développeur peut

[14] http://www.vagrantup.com/
[15] https://www.virtualbox.org/
[16] http://www.vmware.com/

simplement cloner le dépôt d'un projet contenant un Vagrantfile et avoir un environnement de développement fonctionnel en une seul commande : vagrant up.

J'ai donc créé un Vagrantfile qui permet de démarrer et configurer des machines virtuelles provisionnées de la même manière que les serveurs utilisés en production par Applidget. Cette étape m'a aussi permis de me familiariser avec le fonctionnement, le code et les différents *workflow* de la plateforme.

J'ai également pris conscience de l'importance de l'utilisation des variables d'environnements. Leur rôle est cité dans l'une des règles[17] du *Twelve-Factors app* : la configuration d'une application est tout ce qui varie d'un environnement à un autre (développement, test, production). Cette configuration ne doit pas être stockée dans le code, mais dans des variables d'environnements. Il doit exister une séparation stricte entre le code et la configuration d'une application : la configuration varie, pas le code. Les modifications que j'ai dû apporter à la plateforme pour la rendre compatible avec l'environnement de développement étaient majoritairement liées au non respect de cette règle.

II.1.2 – Utilisation des buildpacks *as a service*

Packman est un nouveau module qui devait être intégré à la plateforme. Il permet d'utiliser les *buildpacks* de Heroku (voir partie I.2.1.2) comme un service, dans le sens *Service Oriented Architecture* (SOA). Il a pour but de remplacer le module « builder » (voire figure 9).

Pour rappel, les *buildpacks* permettent de fournir la pile logicielle nécessaire à une application. Leur API est basée sur trois binaires : *detect, compile* et *release* :

- *detect BUILD_DIR* est chargé de dire si oui ou non le *buildpack* peut compiler l'application contenue dans le dossier donné en argument (en étudiant le type de fichier présent, par exemple, le *buildpack* Nodejs cherchera si un fichier *package.json* existe).
- *compile BUILD_DIR CACHE_DIR* a pour mission de compiler l'application contenue dans le dossier BUILD_DIR en utilisant le dossier de cache CACHE_DIR. Cette étape télécharge l'environnement d'exécution et le gestionnaire de dépendances puis compile l'application (voir table 1 et figure 7).
- *release BUILD_DIR* conclue la compilation en exportant certaines informations telle que la variable PATH à utiliser pour lancer l'application.

Packman est un service sans-état. Avant de compiler une application il faut que le code source de celle-ci ait été envoyé sur un *bucket S3*. Pour déclencher la compilation d'une application, une requête HTTP POST contenant le nom de l'application doit être faite sur le serveur de Packman. Packman va alors télécharger le code source de cette dernière depuis le *bucket S3* et un éventuel dossier de cache (qu'il aurait lui-même créé lors d'une compilation précédente). Il utilise ensuite l'API *detect* afin de trouver le *buildpack* approprié et déclenche les actions *compile* et *release*. À l'issue de la compilation, l'application et le dossier de cache sont envoyés sur le *bucket* S3.

[17] http://12factor.net/config

J'ai du faire une mise à jour de Packman afin qu'il suive l'évolution de l'API des *buildpacks.* En effet, l'étape *release* est en cours de dépréciation. Son rôle est désormais confié à l'étape *compile.* L'étape *compile* crée un dossier `.profile.d` qui contient des scripts pour mettre en place l'environnement d'exécution de l'application.

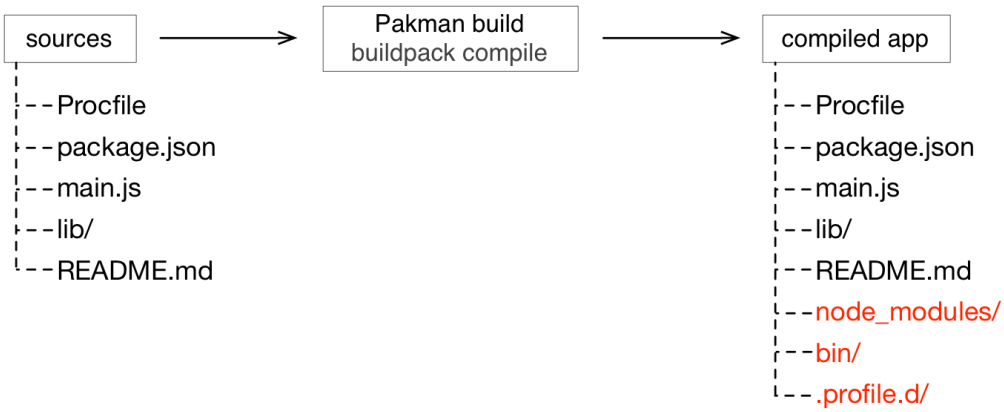

Figure 10 : Résultat d'une compilation avec Packman

La figure 10 montre l'état d'une application Nodejs, avant et après avoir été compilée par le *buildpack* Nodejs. Trois dossiers ont été injectés au code source :

- `node_modules` : contient les modules externes dont à besoin l'application. Ces modules sont décrits dans le fichier package.json, le fichier de description de dépendances de Nodejs (voir table 1)
- `bin` : contient un binaire de Nodejs, indispensable pour exécuter l'application
- `.profile.d` : contient un script qui permet d'ajuster la variable PATH du système afin que l'application puisse accéder à la version de Nodejs et aux modules empaquetés par le *buildpack*.

L'application compilée peut alors être exécutée en utilisant cette commande :

```
source .profile.d/* node main.js
```

J'ai aussi intégré l'utilisation de *trailers* HTTP afin de compléter le contrat d'utilisation de Packman. Les *trailers* HTTP permettent d'envoyer un second *header* à la fin de la communication (en plus du *header* envoyé au début de chaque communication). Ce deuxième *header* est utilisé pour envoyer le résultat de la compilation (succès ou erreur) au service qui l'a invoqué.

Les *buildpacks* ont pour but d'adresser au mieux les problèmes liés à la gestion de la pile logicielle d'une application. Cependant, ils s'occupent uniquement de la couche supérieure de la pile logicielle. En effet, si un module requiert l'accès à une librairie système, celle-ci doit être installée à la fois sur la machine qui exécute Packman afin que celui-ci puisse compiler le module, mais aussi sur la machine où sera exécutée l'application. Les problématiques liées au *provisionnement* de serveurs sont donc allégées mais toujours présentes.

II.1.3 – Administration de serveurs distants

Une PaaS a un rôle important d'administration de serveurs distants. En effet, c'est elle qui est responsable de configurer et préparer les serveurs de production à recevoir des applications. Aucune intervention humaine sur ces serveurs ne doit être nécessaire, il faut donc être capable d'automatiser certaines tâches. La plateforme doit ainsi pouvoir donner l'ordre aux serveurs de production d'exécuter certains scripts.

En l'état, ces scripts sont exécutés via une connexion SSH, en passant l'intégralité du script à une commande du type :

```
ssh –i key.pem ubuntu@$server_dns « entire script is given here »
```

Cette manière de faire est problématique pour plusieurs raisons :

- les scripts (Bash ou Ruby) sont peu maintenables car ils sont contenus dans une chaine de caractères, au milieu d'un code généralement écrit dans un autre langage.
- Il y a un couplage fort entre les scripts exécutés sur les serveurs de production et le code de la plateforme.
- Les appels à la commande ne sont pas centralisés.

Afin de répondre à ces problèmes et standardiser les interactions sur les serveurs de production, un serveur de scripts (ou serveur de recettes) a été créé. Celui-ci permet d'exécuter un script sur un serveur distant en exécutant une commande SSH qui va télécharger le script puis l'exécuter :

```
ssh –i key.pem ubuntu@$server_dns « curl $SCRIPT_URL | $ENV bash »
```

La commande SSH devient ainsi standardisée, seul l'URL du script à exécuter et les éventuelles variables d'environnements nécessaires au script sont variables. Chaque script a désormais son propre fichier, et est *versionné* avec le serveur de recettes (*low coupling, high cohesion*). L'avantage de cette méthode est aussi qu'elle ne requiert pas de configuration au préalable sur les serveurs distants et ne laisse aucune trace non désirée (les scripts n'étant pas stocké sur les serveurs).

La création de ce module a soulevé une réflexion sur de possibles problèmes de sécurité : les scripts sont téléchargés et, pour certains d'entre eux, exécutés avec les droits administrateurs (*root*). Un script peut subir une attaque *Man In The Middle* (MITM) et être modifié, ce qui pourrait avoir de graves conséquences sur le serveur devant l'exécuter. Cependant, cette menace est rendue impossible car :

- le serveur de recettes est déployé sur un serveur de production de la société. On y accède donc uniquement au sein d'un même *cluster,* situé dans l'un des *data-centers* d'Amazon. Le risque de MITM est donc nul.
- Les requêtes faites au serveur de recettes doivent contenir un jeton d'authentification (*authentication token*) pour être traitées. Le jeton est transmis dans l'URL, encryptée par le protocole HTTPS.

Le serveur de recette permet donc de standardiser l'administration des serveurs de production. La création de ce serveur est d'autant plus importante que toute la gestion des conteneurs est faite via l'utilisation de scripts, comme expliqué dans la partie suivante.

II.2 – LinuX Containers

II.2.1 – Premiers pas

LXC est le programme qui permet d'interagir avec les technologies de containerisation du noyau Linux. Il se présente sous la forme d'un paquet Linux et s'utilise via une interface en ligne de commandes. Un cas d'utilisation basique de LXC est le suivant :

- `lxc-create –n cn –t ubuntu` : permet de créer un conteneur nommé « cn » en utilisant le modèle (*template)* « ubuntu ». LXC est fourni avec plusieurs *templates*, des scripts qui permettent de créer un système complet. On retrouve plusieurs *templates* fournis pour différentes distributions. Cette commande crée un système de fichier complet de Ubuntu dans la hiérarchie du système hôte.
- `lxc-start –n cn –d` : permet de démarrer le conteneur précédemment créé. L'option `–d` signifie de lancer le conteneur comme un *daemon*, c'est à dire en tâche de fond. Le démarrage d'un conteneur est rapide (0 à 2 secondes).
- `lxc-console –n container` : commande qui permet d'attacher un terminal au conteneur. On se retrouve alors dans un environnement complètement isolé du système hôte. On constate que le conteneur a son propre système de fichier, ses propres processus et sa propre pile réseau (une adresse IP, une adresse MAC etc.). On est réellement « comme » dans un système virtualisé.

LXC contient un ensemble de commandes aux noms explicites tels que `lxc-clone, lxc-freeze, lxc-stop, lxc-shutdown, lxc-destroy`. Bien que techniquement très différent de la virtualisation, l'utilisation et le comportement d'un conteneur est très similaire à celui d'une machine virtuelle.
La légèreté et la rapidité des conteneurs sont aussi impressionnantes, toujours en comparaison aux méthodes de virtualisation traditionnelles. En revanche, j'ai rapidement remarqué une limite en ce qui concerne son utilisation dans le domaine des *Plateform as a Service* : la création d'un conteneur est une opération relativement longue. Celle-ci dure plusieurs minutes, lors de la création du premier conteneur, et une trentaine de secondes lors de la création des autres, (rapidité accrue due au fait que certaines données sont mises en cache). De plus, la création crée uniquement un système minimaliste, on s'attend à une durée encore plus importante en ajoutant un script de provisionnement. Ce temps de mise en place n'est pas acceptable dans le domaine des PaaS. La commande `lxc-clone` qui permet de cloner un conteneur existant aurait pu être une solution : création d'un conteneur modèle ; création de tous les autres conteneurs à partir du conteneur modèle. En suivant cette logique, on profite du fait que le script de provisionnement n'est exécuté qu'une seule fois. Cependant, l'opération de clonage reste une opération trop lente, qui s'explique par le fait qu'un système de fichier complet doit être dupliqué sur le disque de la machine hôte (environ 600 Mo de données).

Ce constat m'a poussé à m'intéresser de près à Docker, le projet open-source de dotCloud mentionné dans la partie I.1.2.

II.2.2 – LXC dans le contexte des PaaS : Docker

Au début de mon stage, Docker était encore très jeune et peu documenté. Je devais étudier son fonctionnement afin de comprendre comment il faisait face au constat évoqué précédemment : la création d'un conteneur est une opération « très » lente.

La première utilisation de Docker donne lieu au téléchargement de l'image du système d'exploitation à utiliser dans les conteneurs. Le téléchargement de l'image remplace donc la commande lxc-create. Cette étape est elle aussi relativement longue, étant donné qu'une image pèse environ 600 Mo. En revanche, une fois l'image téléchargée, la création et le lancement d'un ou plusieurs conteneurs se font de manière instantanée. L'étape longue, celle qui consiste à créer un système de fichier complet pour le conteneur, n'est donc effectuée qu'une seule fois par Docker. Il est donc évident que l'image téléchargée est réutilisée par chaque conteneur.
La question est alors : comment Docker partage-t-il un unique système de fichier entre plusieurs conteneurs de manière complètement transparente ? L'étude du code source de Docker (écrit en *Go,* open source et hébergé sur Github[18]) m'a permis de répondre à cette question. Docker utilise la technologie AuFS (*Another union File System*). AuFS est un système de fichier appartenant à la famille des systèmes de fichier unifiés (*union file system).* Il permet de fusionner différentes couches à un même point de montage : une couche en lecture seule, et une couche en écriture. Docker utilise donc l'image système téléchargée comme couche de lecture seule pour tous les conteneurs et donne à chaque conteneur sa propre couche d'écriture. En m'inspirant de cette solution j'ai créé une preuve de concept, un script *Ruby* qui permet d'automatiser l'utilisation de AuFS avec LXC :

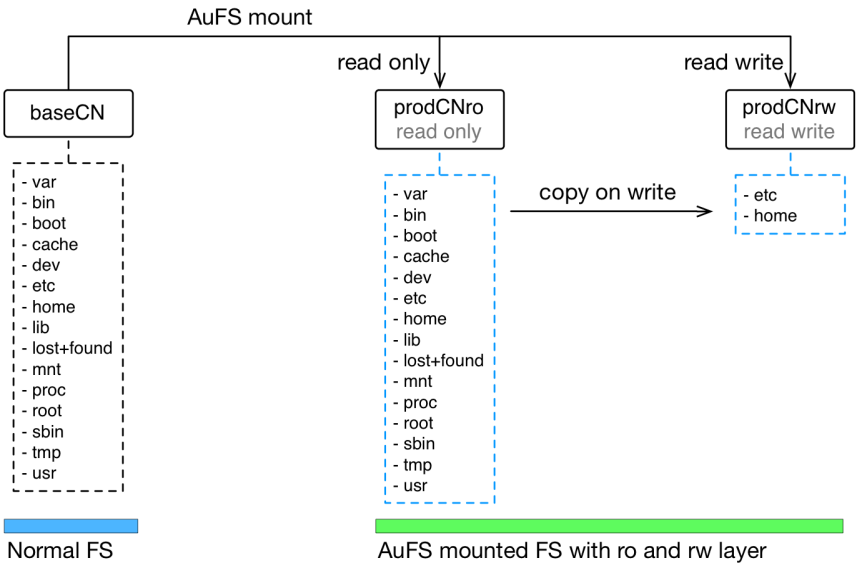

Figure 11 : AuFS

[18] https://github.com/dotcloud/docker

Sur la figure 10, le dossier baseCN contient le système de fichier d'un conteneur qui va servir de base à tous les autres (créé en utilisant la commande lxc-create). Les dossiers prodCNro, la couche lecture seule, et prodCNrw, la couche écriture, sont les dossiers qui accueillent le système de fichier d'un nouveau conteneur (qui est donc composé des deux couches : lecture et écriture). La commande :

```
mount -t aufs -o br=/prodCNrw=rw:/baseCN=ro none /prodCNro
```

permet de monter le contenu de baseCN dans le dossier prodCNro en lui indiquant que toutes les écritures faites dans prodCNro doivent être transmises au dossier prodCNrw (*copy on write*). Le conteneur va interagir avec le contenu du dossier prodCNro. Une écriture dans ce dossier sera automatiquement transférée au dossier prodCNrw. Dans la figure 10, on peut voir que des écritures dans les dossiers home et etc ont été faites et donc, écrites dans le dossier désigné pour recevoir les écritures. Tout cela est totalement transparent pour le conteneur. Le contenu du dossier prodCNro peut être modifié virtuellement, c'est à dire que son contenu n'est jamais réellement affecté.

Pour revenir au problème posé, la durée de création d'un conteneur, AuFS permet :

- d'appeler une seule fois la commande lxc-create (l'opération longue) afin de créer un conteneur de base (baseCN).
- de créer et lancer des conteneurs instantanément (la commande mount citée précédemment étant instantanée) en utilisant le système de fichier du conteneur de base.

En plus de la création instantanée de conteneurs, d'autres effets de bord positifs sont apparus avec l'utilisation de la technologie AuFS :

- une opération de maintenance sur le conteneur de base (installation d'un programme par exemple) est automatiquement perçue par les autres conteneurs étant donné que ceux-ci possèdent le même système de fichier (monté à des endroits différents).
- l'espace disque est grandement économisé du fait qu'un conteneur créé à partir du conteneur de base utilisera de l'espace disque supplémentaire uniquement pour les écritures qu'il aura lui même faites.

J'ai réalisé une preuve de concept, sous forme d'un script Ruby qui orchestre les technologies LXC et AuFS comme décrit ci-dessus. Une fois le script finalisé, une décision devait être prise avec mon maître de stage : devrions nous utiliser Docker pour gérer nos conteneurs ou bien directement travailler avec AuFS et LXC, à la manière de ce qui avait été fait dans la preuve de concept ? La preuve de concept étant convaincante et Docker étant encore trop jeune (l'équipe de Docker déconseille une utilisation en production avant la version 1.0[19]) nous avons décidé de choisir la deuxième solution.

[19] http://blog.docker.io/2013/08/getting-to-docker-1-0/

II.2.3 – La gestion du réseau dans les conteneurs

Les conteneurs vont accueillir des applications web, il est donc important de comprendre comment la couche réseau est gérée et notamment comment accéder à un service web qui tourne à l'intérieur d'un conteneur depuis l'extérieur.

Tel que mentionné précédemment, les conteneurs ont leur propre adresse IP et adresse MAC. L'installation de LXC entraîne la création d'une interface réseau sur la machine hôte, cette interface va être un pont entre les conteneurs et l'extérieur (internet) et agir comme la passerelle des conteneurs. Les conteneurs ont donc accès à internet, ils sont accessibles depuis la machine hôte, mais ne le sont pas depuis l'extérieur. Pour compléter la preuve de concept réalisée mettant en œuvre LXC et AuFS, la notion de redirection de paquets a été intégrée afin que les applications tournant dans les conteneurs soient accessibles depuis l'extérieur. Pour cela, j'ai utilisé l'outil iptables, inclus de base dans le noyau Linux, qui m'a permis de rediriger le trafic entrant sur la machine hôte à un port donné vers un conteneur à un port donné (exemple, `host:3000 => containerA:3000`, redirection du trafic entrant port 3000 sur `containerA` port 3000). Ainsi, les services web contenus dans les conteneurs sont accessibles depuis l'extérieur.

Ce travail préliminaire sur les conteneurs devait aussi déboucher sur un indice de faisabilité de la mission. En effet, LXC n'est pas encore massivement utilisé et il y a peu de ressources disponibles. Il était donc difficile de rapidement se faire une idée sur son utilisation dans le contexte de la mission proposée. La preuve de concept complète, intégrant LXC, AuFS et iptables pour la redirection du réseau a permis de démontrer qu'une utilisation des conteneurs performante, à l'image de ce que proposent les *PaaS* Heroku et dotCloud, était possible. La transparence de dotCloud, créateur du projet open-source Docker, a comblé le manque de ressources.

II.3 – Intégration de LXC dans Dcdget

À ce stade, le module Packman et le serveur de recettes étaient disponibles et des scripts permettant d'automatiser la création de conteneurs Linux avaient été réalisés. Tous les outils étaient donc réunis pour intégrer la technologie LXC dans la plateforme Dcdget.

II.3.1 – Modèle de données

Le modèle de données de la plateforme est géré par l'application centrale (voir figure 9). La figure ci-dessous présente le diagramme de classe légèrement simplifié du modèle :

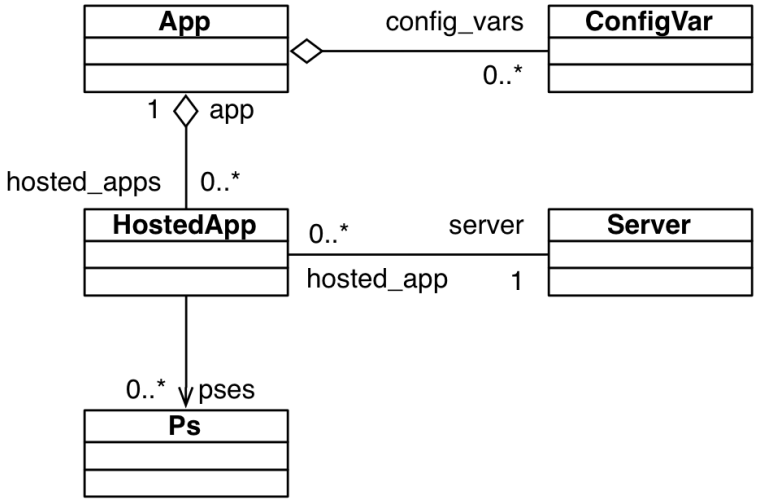

Figure 12: Diagramme de classe simplifié de Dcdget

Une application gérée par la plateforme est représentée par la classe App. Chaque App possède une ou plusieurs HostedApp, une application hébergée sur un serveur. Logiquement, une HostedApp est associée à un serveur et un serveur peut héberger plusieurs HotedApp. Une HostedApp a aussi un ensemble de processus représentés par la classe Ps. Enfin, une application peut posséder des ConfigVars, des variables d'environnements (concept important du *Twelve-Factors App*).

II.3.2 – Cadre de travail

Dcdget est un module central de la société et un problème sur la plateforme peut rapidement devenir problématique. Afin d'éviter cela, il était nécessaire d'isoler au maximum mon travail pour que celui-ci ait un impact très faible sur l'existant. La majorité de ma mission se concentre sur le *Worker* de Dcdget (voire figure 9). J'ai créé un second *Worker*, branché à la même API. Concrètement, un attribut *stack* a été ajouté à la classe App dans le modèle de données (le mot *stack* a été emprunté de la terminologie utilisée par Heroku, celui-ci désigne un environnement complet de déploiement). Cet attribut peut avoir deux valeurs : Antik ou Batman, Batman représentant la *stack* utilisant les conteneurs et Antik représentant l'ancienne *stack*. Quand l'application centrale demande au *Worker* d'effectuer une opération, elle lui demande via son API REST. L'API va simplement vérifier la *stack* de l'application et rediriger la requête en conséquence, soit à l'ancien *Worker* soit au nouveau.

Le *Worker* est écrit en Nodejs, un langage fonctionnel issu du JavaScript, qui a la particularité d'être asynchrone, basé sur les événements (un seul processus, pas de *thread)* et ainsi très léger et performant[20]. J'ai utilisé le *coffeescript* pour écrire le nouveau *Worker.* Le *coffeescript*[21] est un langage qui se compile en JavaScript, il est donc possible d'écrire du code Nodejs avec. L'avantage de celui-ci est qu'il est peu verbeux et a une syntaxe pensée pour l'utilisation des paradigmes de la programmation orientés objet en JavaScript.

[20] http://nodejs.org/
[21] http://coffeescript.org/

II.3.4 – Implémentation

Trois fonctionnalités importantes de la plateforme sont impactées par l'intégration des conteneurs :

- le déploiement d'une application
- l'ajout de variables d'environnements à l'application
- la destruction d'une application

Chacune des ces fonctionnalités sont un enchaînement d'actions. Le déploiement d'une application par exemple comporte les étapes principales suivantes :

#	Nom de l'étape	Description
1	Compilation de l'application	Utilisation du module Packman pour compiler et préparer l'application au déploiement
2	Téléchargement de l'application compilée	L'application est téléchargée sur la plateforme après avoir été compilée par Packman
3	Envoi de l'application sur les serveurs	L'application est envoyée sur le ou les serveurs de productions sur lesquels elle doit être déployée
4	Création de conteneurs	Pour chacun des processus de l'application, un conteneur est créé
5	Démarrage des processus	Les processus sont ensuite démarrés
6	Mise à jour de la configuration du serveur	La configuration du serveur Nginx est mise à jour pour que la nouvelle application soit accessible

Table 3 : Principales étapes d'un déploiement

Cet aperçu des actions nécessaires pour déployer une application montre bien l'aspect de *workflow* qui ressort. Pour adresser au mieux cette situation, j'ai décidé de m'inspirer des concepts de la SOA dans le design de mon code : chacune des fonctionnalités évoquées précédemment (déploiement d'une application ; ajout de variable d'environnements ; destruction d'une application) est une orchestration de fonctions des « managers ». Chaque « manager » est un ensemble de fonctions sans état, ayant une interface bien définie. Les « managers » sont l'équivalent des services en SOA.
Ce design permet une lecture aisée du code source, car les orchestrations ne contiennent que très peu de logique, on distingue ainsi clairement les différentes étapes. Les « managers » ont des interfaces très simples qui permettent notamment une bonne gestion des erreurs : si une étape échoue, on est capable de précisément identifier le service qui a échoué et arrêter l'action en cours.

Le choix d'un bon design était important. En effet, le code implémenté est complexe car il communique avec des services différents (l'application centrale et Packman) et il doit exécuter des scripts Bash ou Ruby sur des serveurs distants. De plus, le problème a été présenté de manière simplifiée, un déploiement compte en réalité 15 étapes, certaines étant des boucles. Il y a donc de nombreux points susceptibles d'être source d'erreurs d'autant

plus que beaucoup d'opérations sont exécutées en passant par le réseau. Le couplage très faible impliqué par le design choisi permet d'avoir un code stable rapidement, les éventuels problèmes étant aisément distinguables et corrigeables.

II.3.5 – Panorama de l'évolution de Dcdget

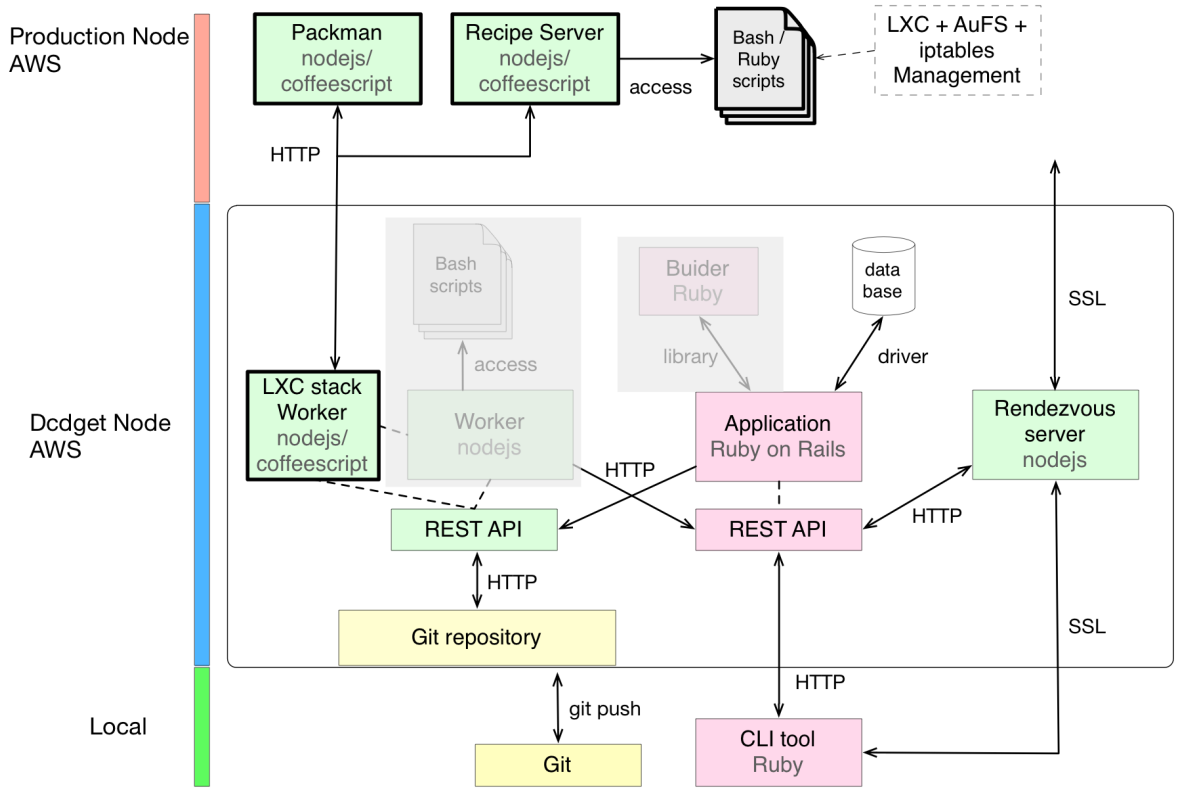

Figure 13 : Dcdget après intégration

La figure 11, basée sur la figure 9, décrit la plateforme à l'issue de mon stage. Les modules grisés (*stack* Antik*)* sont dans une phase de dépréciation, c'est à dire qu'ils ne sont plus maintenus, et seront supprimés quand les nouveaux modules (ceux mis en évidence par des bordures plus épaisses) seront considérés comme étant stables (*stack* Batman). La plateforme progresse en découplage. Les scripts sont désormais gérés par le serveur de recette. Le module « builder » est remplacé par Packman, beaucoup plus flexible et performant dans ce qu'il fait. Le nouveau *worker* gère désormais les applications dans des conteneurs LXC.

II.4 – Applications en production

Après s'être intéressé au code de la plateforme et aux différents modules qui ont été développés autour, il est intéressant de détailler ce qu'il se passe sur les serveurs de production et comment la plateforme organise et supervise les applications sur ces serveurs.

II.4.1 – Représentation

Sur les serveurs de production, une certaine hiérarchie au sein du système de fichier est mise en place. Rappelons que pour une application déployée, un nouveau conteneur est créé pour tous les processus de cette application. Chacun des conteneurs doit donc avoir accès à l'application et cela nécessite donc que l'application soit disponible dans le système de fichier de tous les conteneurs (ceux-ci étant totalement isolé les uns des autres). Afin de ne pas avoir à copier l'application dans chaque conteneur, j'ai tiré parti de la possibilité de monter des dossiers du système hôte dans des dossiers de conteneurs. Cela permet de créer des dossiers partagés :

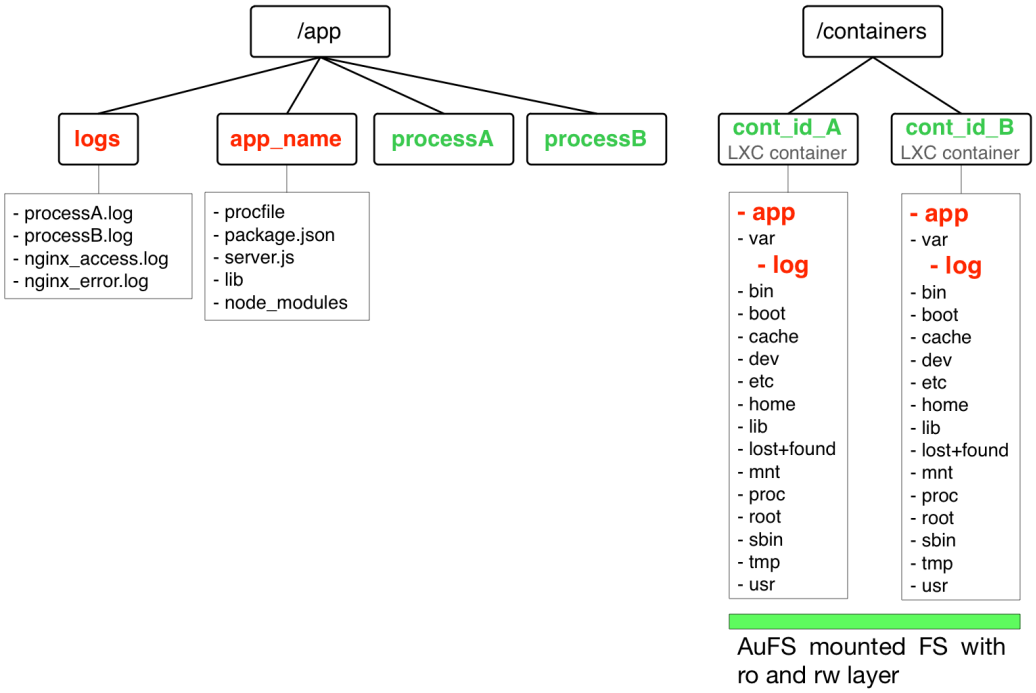

Figure 14 : Hiérarchie d'une application en production

Les dossiers app et log sont ainsi montés dans chaque conteneur, ce sont donc des dossiers partagés entre les conteneurs d'une même application. Cela permet de centraliser l'application ainsi que l'accès à ces logs. Les conteneurs ont chacun un identifiant unique et un lien symbolique est fait dans le dossier de l'application. Cette hiérarchie permet à la fois d'avoir une vue d'ensemble de l'application et de garder une séparation entre applications et conteneurs. Cela apporte de la flexibilité sur l'utilisation des conteneurs, ils n'appartiennent pas nécessairement à une application et peuvent être utilisés pour contenir d'autres services (évolution évoquée en partie II.5). La mise en place de cette hiérarchie est déclenchée par la plateforme Dcdget et effectuée par des scripts gérés par le serveur de recette, lors d'une procédure de déploiement.

La figure 15 représente la pile logicielle d'un processus d'une application Nodejs en production :

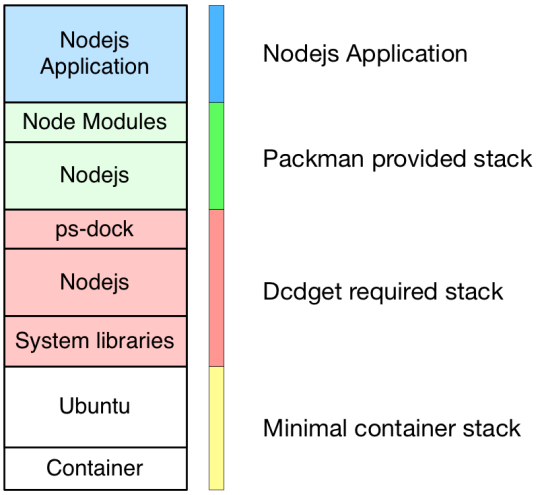

Figure 15 : Pile logicielle d'un processus de Dcdget

La création d'un conteneur fournit la pile logicielle de base : un système Ubuntu. Un script de provisionnement fournit la seconde couche : un ensemble de programmes et de bibliothèques dont ont besoin les applications de la société. Cela inclut notamment ps-dock, un programme JavaScript utilisé pour superviser les processus (plus de détailles dans la partie suivante). Packman fournit ensuite les dépendances applicatives. On voit que Nodejs est présent deux fois. En revanche les versions peuvent-être différentes, Packman fournissant une version spécifique à l'application (spécifiée dans le fichier de dépendances de l'application).
Cette figure présente bien le but initial recherché : l'application est totalement isolée du système hôte. La pile logicielle et l'environnement sont conteneurisés.

II.4.2 – Supervision et haute disponibilité

Une fois les applications lancées sur les serveurs de production, il faut être capable de superviser leur processus, dans un but de haute disponibilité. Pour cela, deux outils sont utilisés : Upstart, et ps-dock.

Upstart est un processus du système d'exploitation Ubuntu (le système utilisé par la société sur ses serveurs). Il est l'équivalent de init dans la majorité des autres distributions Linux. C'est le premier processus exécuté au démarrage du système et il est responsable de lancer d'autres processus. Son intérêt est qu'il supervise ses processus fils et est ainsi capable d'automatiquement relancer un processus qui *crash*. Afin de maximiser la haute disponibilité, toutes les applications déployées par Dcdget sont démarrées et supervisées par Upstart. En ce qui concerne la *stack* Batman, chaque conteneur est supervisé par le processus Upstart du système hôte et les processus des applications déployées sont supervisés par le processus Upstart des conteneurs et sont configurés pour être exécutés au démarrage. De cette manière, si une application rencontre un problème, celle-ci est automatiquement relancée et son temps d'arrêt est minimal.

Upstart est donc un outil puissant de supervision de processus, cependant, il fonctionne de manière totalement transparente, et ne permet pas de savoir quand une application à crashé et pourquoi. Pour remédier à cela Applidget a développé ps-dock, un outil écrit en Nodejs, open-source[22] qui permet :

- de faire de la log rotation (puissante gestion des logs qui permet notamment l'archivage)
- de surveiller les signaux Unix[23] émis par une application et d'informer un *webhook*[24]

Ps-dock est donc utilisé en complément de Upstart pour englober tous les processus déployés par Dcdget :

```
Upstart ---
        Ps-dock ---
                Application-process
```

Quand l'un de ces processus rencontre un problème, la plateforme est avertie (via un *webhook*), elle est donc capable de garder un historique de l'état de santé de tous les processus.

II.4.3 – Interventions humaines

La haute disponibilité implique aussi de mettre en place des processus afin d'éviter qu'une intervention humaine cause un temps d'arrêt. Le serveur de rendezvous (voir figure 9) à pour vocation de supprimer toutes connexions humaines directes aux serveurs de production. Il permet par exemple de faire des opérations de maintenance sur la base de données d'une application, sans se connecter directement sur les serveurs. Il doit donc permettre d'exécuter des commandes dans le même environnement qu'une application. Il s'utilise côté client, de cette manière :

```
dcdget run $CMD –app my_app
```

La commande passée en argument est exécutée sur le serveur de production en héritant de l'environnement de l'application `my_app`. Le résultat de cette commande est directement retranscrit. Cela est rendu possible par ps-dock. Comme dit précédemment, ps-dock permet de superviser un processus fils qu'il lance lui même. Il est aussi capable de rediriger la sortie (*stdout*) de ce processus fils sur un socket TCP/SSL. Le rendezvous serveur fonctionne donc à l'aide à la fois de ps-dock et de dcdget client. Quand le premier parti d'une connexion arrive (ps-dock), le serveur va attendre que la connexion lui envoie un secret. Si le secret est vérifié par l'application centrale, la connexion est authentifiée et les données qu'elle envoie sont bufférisées. Quand le deuxième parti arrive (Dcdget client), sa connexion va elle aussi être authentifiée via un secret. Si le secret est identique au secret du premier parti, les deux connexions vont être « branchées » entre elles et leur buffer vidé mutuellement.

[22] https://github.com/applidget/ps-dock
[23] http://en.wikipedia.org/wiki/Unix_signal
[24] http://en.wikipedia.org/wiki/Webhook

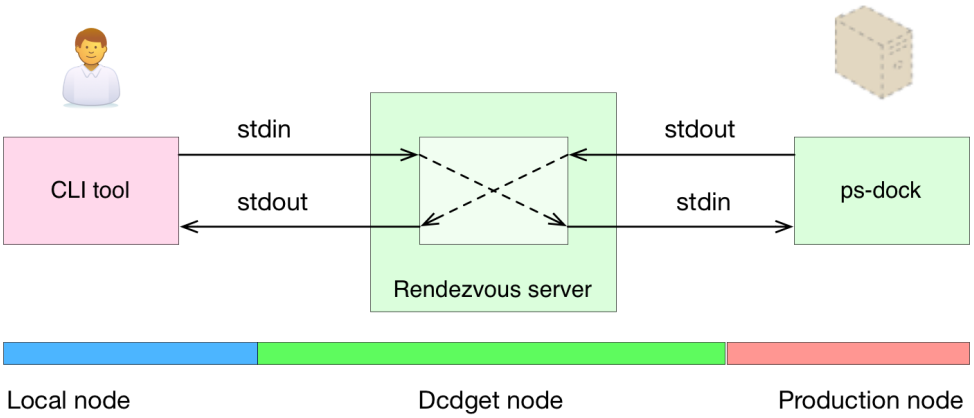

Figure 16 : Serveur de rendezvous

Les deux partis restent branchés tant qu'aucun ne ferme sa connexion.

J'ai dû faire fonctionner le serveur de Rendezvous avec la nouvelle *stack*. Pour cela, la commande distante (gérée par ps-dock) est exécutée dans un conteneur ayant le même environnement que les autres conteneurs de l'application. Les changements dans le cycle de vie d'un conteneur étant rapides (création, démarrage et destruction), notamment grâce à l'utilisation de AuFS, l'utilisation de `dcdget run` est aussi performante sur la *stack* Antik que sur la *stack* Batman. La différence est que le processus lancé sur la *stack* Batman bénéficie de l'isolation fournie par les conteneurs, et ne peut pas affecter le serveur et les applications qu'il héberge.

II.5 – Bilan sur le travail réalisé

À l'heure où j'écris ces lignes, il me reste un peu plus d'un mois de stage. Le travail que j'ai effectué et décrit dans ce rapport est en phase de pré-production. La plateforme a été déployée. La *stack* Antik fonctionne comme avant et n'a pas été impactée par le développement de la nouvelle *stack.* Un troisième serveur de production est venu compléter l'infrastructure d'Applidget. Ce serveur est le premier à accueillir des applications déployées avec la *stack Batman.* L'objectif est de migrer toutes les applications vers cette nouvelle *stack*. La phase de pré-production permet aussi de tester le comportement des applications dans les conteneurs sur du long terme.

Le travail effectué permet d'envisager l'ajout de fonctionnalités importantes à moyen terme dans la plateforme. Il est notamment prévu de travailler sur une méthode de déploiement ne donnant pas lieu à un temps d'arrêt. En effet, quand une application est redéployée, celle-ci subit un temps d'arrêt égal à la durée de son démarrage. L'utilisation des conteneurs va permettre de faire de la rotation : la nouvelle version de l'application sera déployée dans de nouveaux conteneurs, et ce n'est qu'une fois totalement redémarrée qu'elle prendra la place des anciens conteneurs.
Il est aussi envisagé de créer une nouvelle commande pour l'outil local de Dcdget, une commande qui permet de consulter les logs de toutes les instances d'une application en temps réel (note : cette fonctionnalité, très liée au serveur de rendezvous et à ps-dock a été implémentée peu avant de rendre ce rapport).

Une bonne chose serait aussi de réécrire ps-dock dans un langage qui se compile en binaire afin de l'utiliser comme le premier programme exécuté quand un conteneur démarre. En effet, la dernière étape de démarrage d'un conteneur, comme tout autre système Linux, est d'appeler le programme init. Init est chargé de lancer les processus de base qui tournent dans un environnement utilisateur. En remplaçant le binaire init des conteneurs par un binaire ps-dock, on lance uniquement le processus de l'application qui nous intéresse. Docker utilise cette technique et remplace totalement le programme init des conteneurs qu'il gère par son propre binaire.
Enfin, il est prévu d'utiliser les conteneurs afin d'isoler les applications tierces utilisées tels que les outils de stockage Redis[25] et Memcached[26]. La manière d'intégrer les conteneurs à la plateforme a été pensée pour ce cas d'utilisation.

La difficulté principale du stage était la variété des technologies. Il m'a été demandé d'être à l'aise avec de nombreuses d'entre elles, allant de la programmation, aux réseaux et à l'administration système. Il était aussi important de faire les bons choix dès le début. En effet, travailler sur des systèmes distribués requiert de définir des interfaces afin que les différents modules puissent communiquer. Choisir la bonne interface dès le début est crucial car, changer une interface entraîne une incompatibilité qui nécessite une intervention sur tous les modules utilisant cette interface. J'ai aussi rencontré de nombreux problèmes liés aux différentes versions de Nodejs. Nodejs est encore jeune et évolue rapidement. Applidget n'ayant pas maintenu leurs outils pour qu'ils fonctionnent avec les versions récentes de Nodejs, j'ai donc dû passer un certain temps sur cette tâche. En revanche, j'ai participé à la mise à jour de MongoDB sur les serveurs de production et celle-ci c'est faite sans aucun problème ni temps d'arrêt. Cet aspect de difficulté de changement de version est a prendre en compte lors d'un choix technologique.

[25] http://redis.io/
[26] http://memcached.org/

III – Planning prévisionnel et planning réalisé

III.1 – Liste des tâches prévues

Le tableau ci-dessous présente la liste des tâches prévues initialement :

#	Nom	Description
1	Étude de l'existant	Analyse du code des différentes briques logicielles existantes (Dcdget, dcdget-server, packman, ps-dock)
2	Mise en place de l'environnement de développement	La plateforme est déployée dans un environnement de production, il faut la migrer vers un environnement de développement (tout simuler via des machines virtuelles locales)
3	Apprentissages des technologies	Maîtriser les technologies : Ruby, Ruby on rails, Nodejs, coffee script et LXC
4	État de l'art	Étudier ce qui existe dans le domaine des plateformes de déploiement, notamment Heroku, et de ce qui a été réalisé avec les conteneurs LXC (Docker). Cette tâche consiste aussi en la réalisation de preuves de concepts
5	Intégration de Packman	Packman a été développé mais pas intégré, il faut donc l'intégrer à la plateforme
6	Design de l'intégration des conteneurs	Décider la manière dont les conteneurs seront intégrés à la plateforme et la manière de cohabiter avec les autres composants
7	Implémentation de l'intégration des conteneurs	Implémentation technique des conteneurs à la plateforme, basée sur les design réalisé précédemment
8	Test et validation	Test et validation des ajouts à la plateforme
9	Mise en place d'un système de *monitoring* des conteneurs	Étudier ce qui existe dans le domaine de la surveillance des conteneurs et intégrer une solution à la plateforme
10	Intégration de ps-dock	Intégrer ps-dock à la plateforme
11	Documentation	Documentation d'une part de l'utilisation de la plateforme pour un administrateur et d'autre part pour un développeur
12	Mise en production conteneurs	Remplacement de la plateforme actuellement en production par la version produite
13	Projet extérieur	Cette tâche englobe les différentes interventions sur des projets extérieurs au sujet du stage (développement mobile, web back-office etc.)
14	Gestion des risques	Tâches vides, qui permet de réserver du temps à la gestion des problèmes rencontrés

Table 4 : Liste des tâches

Dans l'ensemble, cette liste a été respectée. Les tâches numéro 9, 4 et 11 n'ont pas été faite. Concernant la tâche 9, la société utilise déjà l'outil Newrelic[27] permettant de surveiller très finement l'activité de ses serveurs. La tâche 4 est venue se greffer à la tâche 3 (apprentissage des technologies). La tâche numéro 11 a été très légère en terme de durée et n'a pas eu de poids dans le planning.

III.2 – Planning

Le planning prévisionnel ainsi que le planning réalisé sont disponibles à cette adresse : http://goo.gl/yaZEM (en ligne pour des raisons de lisibilité).

La tâche de gestion des risques était importante pour les raisons suivantes :

- bien qu'utilisée en production, la plateforme est jeune et est propice à d'éventuels bugs qui devront être corrigés
- l'envergure de l'existant
- les technologies ne sont pas maîtrisées
- le code existant des différentes briques logicielles est peu documenté
- le côté très dynamique de l'aspect recherche et développement

La durée effective de la gestion du risque était aussi liée au temps passé sur les projets extérieurs et a finalement était de deux semaines. Il n'y a pas eu d'écart significatif entre la durée prévue et la durée réelle. Seule la tâche de mise en production est beaucoup plus importante que prévue. En effet, celle-ci implique de migrer toutes les applications actuelles sur de nouveaux serveurs. De plus cette migration n'est pas faite en une fois, un certain temps est utilisé pour vérifier le comportement des applications conteneurisées. Cette tâche est critique car elle ne doit pas causer de temps d'arrêts aux services de la société et implique donc une méthode rigoureuse. La gestion du risque et le temps économisés sur les tâches n'ayant pas eu lieux (9, 4 et 11) ont permis d'absorber cette erreur d'appréciation. Dans le planning réel, une tâche 15 est apparue, elle correspond au travail qui sera effectué lors des dernières semaines de mon stage (travail sur les améliorations possibles évoquées en partie II.5).

[27] http://newrelic.com/

Conclusion

Le *cloud computing* est une des évolutions majeures dans le domaine de l'informatique de ces dernières années. Il est source d'innovation car il permet aux entreprises n'ayant pas les ressources nécessaires pour avoir un *datacenter* de bénéficier d'infrastructures flexibles et performantes. Applidget profite de cet aubaine et oriente son activité autour des *Software as a Service.* Les problématiques liées au déploiement d'applications dans le *cloud* peuvent être prises en charge par des *Platform as a Service.* Ces plateformes ont un rôle d'intermédiaire entre le développeur et les serveurs de production et permettent de cacher la complexité de l'administration d'infrastructure aux développeurs. Elles facilitent ainsi l'accès au *cloud.*
Applidget a développé une *PaaS* en interne, adaptée à ses besoins et à son infrastructure. J'ai rejoint leur équipe de développement pour travailler sur cette plateforme afin que celle-ci utilise la technologie LinuX Containers (LXC). LXC permet d'isoler des applications au niveau du système d'exploitation. Très proche en pratique des technologies de virtualisation, les conteneurs offrent une isolation beaucoup plus légère et performante. Les conteneurs Linux permettent ainsi de résoudre l'un des défis techniques majeurs des PaaS : l'isolation d'application.

Le stage s'insère dans l'activité recherche et développement de la société, son objectif n'était donc pas strictement fixé. Mon travail a dans un premier temps apporté à la société une connaissance sur la technologie LXC, et sur la manière d'utiliser cette technologie efficacement dans le domaine des *Platform as a Service.* J'ai ensuite mis en pratique ces connaissances pour intégrer la containerisation à Dcdget. Le travail quotidien sur la plateforme a aussi permis de corriger certains problèmes et a apporté de la maturité à la plateforme. Les attentes autour de ce stage ont donc été satisfaites, malgré le fait que le travail effectué ne soit pas encore utilisé en production. Cependant, les premiers retours de la phase de pré-production sont positifs et une utilisation exclusive de la nouvelle *stack* est prévue avant la fin du stage.

Mon travail m'a permis de progresser dans de nombreux domaines et m'a apporté de l'expérience et de la maturité sur la façon d'aborder un projet de systèmes distribués. J'ai dans un premier temps découvert de nombreuses technologies, que ce soit des langages de programmation, des technologies relatives aux systèmes d'exploitation ou des outils pour supporter un processus de développement. Cela m'a permis d'élargir mon éventail de connaissances et ainsi de me donner plus de recul sur les choix technologiques que je serai amené à faire dans le futur. J'ai aussi pris conscience de la difficulté de produire des logiciels robustes, « sans erreur », « faciles » à maintenir et à faire évoluer. Dans ce sens, je pense que le développement dirigé par les tests, ou du moins le fait de produire et maintenir une suite de tests, aurait été réellement bénéfique durant mon stage. Travailler sur ce projet m'a également appris certaines bonnes pratiques, notamment celles regroupées dans le document *Twelve-Factors app,* qui est un guide pour la création d'applications portables d'un environnement à un autre et qui sont pensées pour le passage à l'échelle. La réalisation de cette mission m'a ainsi permis de mettre en pratique et de compléter les connaissances acquises en architecture logicielle lors de ma formation universitaire.

La technologie LXC a un très fort potentiel dans le monde de l'administration système et des systèmes distribués quels qu'ils soient. Le besoin d'une telle technologie n'est plus à démontrer au vu du fort succès rencontré par le projet Docker (150 contributeurs et 5431 personnes suivant le projet sur github.com en un peu plus de cinq mois d'existence), premier à simplifier l'utilisation des conteneurs tout en tirant parti du système de fichier AuFS. Plusieurs projets utilisant Docker vont voir le jour, notamment une *Platform as a Service* open-source, flynn.io[28] qui a pour but de servir aussi bien les *clouds* privés que publics. Les *Platforms as a service* ont donc encore beaucoup à offrir au *cloud computing*. Marché encore dominé par les sociétés Américaines, il est probable que l'apparition d'acteurs Européens se fasse rapidement, notamment pour répondre aux problèmes de confidentialités récemment mis en avant par l'affaire PRISM[29] aux Etats-Unis.

[28] http://flynn.io

[29] http://en.wikipedia.org/wiki/PRISM_(surveillance_program)

Glossaire

Amazon Web Services – AWS : L'*Infrastructure as a Service* fournie par la société Amazon

Application Programming Interface – API : Une API est une interface qui permet de définir comment des composants logiciels doivent interagir

AWS S3, bucket S3 : S3 est le service de stockage de données d'AWS. Un même espace de stockage est appelé *bucket*

Backend : désigne la partie d'une application cachée à l'utilisateur final, partie qui fonctionne en général sur un serveur

Domain Specific Language – DSL : Un DSL est un langage de programmation créer pour servir un domaine spécifique. Le SQL est un DSL créé pour le domaine des bases de données relationnelles par exemple

Lien symbolique : équivalent de l' « alias » ou du « raccourcis », les liens symboliques permettent de référencer un dossier à un autre endroit sur le disque que son emplacement « réel »

MongoDB : MongoDB est une base de donnée dite « no-sql », orienté document.

Nginx : Nginx est un puissant serveur HTTP. Il est souvent utilisé comme un serveur *frontend* pour gérer plusieurs applications sur un même serveur ayant des noms de domaine différents.

Pile logicielle : La pile logicielle d'une application représente toutes les dépendances dont une application à besoin pour fonctionner

POST (verbes HTTP) : Le protocole HTTP est fait de différents verbes qui définissent tous une action précise. Les plus courant sont POST (création), PUT (mise-à-jour), GET (accès) et DELETE (suppression).

Provisionnement : Du terme anglais *provisionning*, l'action de provisionner représente le fait d'installer un ensemble de programmes sur une machine.

REST : Pour REpresentational State Transfer, une architecture REST ou API REST est basée sur le protocole HTTP et les différents verbes qui le compose (voir POST)

Variable PATH : La variable PATH est une variable d'environnement présente dans les systèmes basé sur Unix. Elle définit le chemin d'accès aux exécutables du système

Versionner : Du terme anglais *versionning, versionner* un fichier est l'action de gérer un fichier en utilisant un outil de gestion de version tel que Git ou SVN

Bibliographie

12 Factors App, Adam Wiggins
http://12factor.net/

Heroku : Dynos and the Dyno Manager
https://devcenter.heroku.com/articles/dynos

Heroku : Buildpacks API
https://devcenter.heroku.com/articles/buildpack-api

Heroku : platform API reference
https://devcenter.heroku.com/articles/platform-api-reference

Lightweight Virtualization LXC containers & AUFS, Jérôme Petazzoni, dotCloud : (Février 2013) :
http://www.socallinuxexpo.org/sites/default/files/presentations/Jerome-Scale11x%20LXC%20Talk.pdf

Docker
https://www.docker.io/

Containers and Docker : How secure are they ? Jérôme Petazzoni, dotCloud (aout 2013)
http://blog.docker.io/2013/08/containers-docker-how-secure-are-they/

LXC Linux Container
http://lxc.sourceforge.net/

Upstart Cookbook
http://upstart.ubuntu.com/cookbook/

Résumé

L'isolation d'applications tournant en production au sein d'une *Plateform as a Service* (PaaS) est essentielle pour des raisons de sécurité et de maîtrise de l'empreinte d'un processus sur un système. Isoler une application au niveau du système d'exploitation permet un contrôle total des ressources logicielles et matérielles. Cependant, dans le contexte du *cloud computing*, la virtualisation coûte trop chère à la fois en terme de performance que de budget. Les conteneurs Linux (LXC) permettent une alternative à la virtualisation classique, beaucoup plus légère et performante.
Ce document explique ce qu'est une PaaS et les enjeux techniques auxquelles ce type de plateforme est confronté. Il présente ensuite comment les conteneurs Linux peuvent être utilisés afin d'isoler des applications en respectant les contraintes imposées par le contexte des PaaS, incluant l'automatisation et la rapidité de mise en œuvre.

Abstract

The isolation of running applications used in production within a Platform as a Service (PaaS), is essential for two reasons: security and footprint control of processes over a system. Operating system level isolation allows a complete control of resources whether they are software or hardware. However, cloud-computing context doesn't encourage the use of virtualisation, due to its cost in term of performance and budget. Linux containers (LXC) provide a lightweight and fast alternative to classic virtualisation.
This document explains what a PaaS is and the technical and organisational issues such platform must face. It then shows how Linux containers may be used in order to isolate cloud applications, matching constraints of the context of PaaS such as automation and setup speed. It also presents how powerful containers may be for a cloud infrastructure administration.

Keywords

chroot, LXC, PaaS, IaaS, SaaS, cloud, Heroku, git, AWS, vagrant-up, docker, Nodejs, Ruby, Rails, Go, virtualisation, container, isolation, buildpacks, Linux, Bash, Mongodb, Redis, AuFS, HTTP, REST, SSH, Upstart, Docker, High availability, portability