Université Pierre et Marie Curie (UPMC)

Master de Sciences et Technologies - mention Informatique

Spécialité «Systèmes et Applications Répartis» (SAR)
Parcours Systèmes Répartis et Middleware (SRM)

rapport de stage

Projet Résilience:

Surveillance du Cloud par le Cloud

Lahoucine BENLAHMR

Responsable Wallix : Fabien Boucher
Référent UPMC : Gérard Nowak

WALLIX
Année 2011-2012

Table des matières

I. Remerciements

Je tiens à remercier dans un premier temps, toute l'équipe pédagogique de l'UPMC et le corps enseignant du master SAR, pour avoir assuré la partie théorique du master.

Je remercie également monsieur Gérard Nowak pour ses nombreux conseils de direction et de rédaction, qui m'ont permis de fournir un travail que j'espère pertinent et intéressant.

Je tiens à remercier tout particulièrement, et à témoigner toute ma reconnaissance, à monsieur Fabien Boucher, mon tuteur de stage, pour sa confiance et ses conseils, pour m'avoir fait partager son expérience et ses compétences. Je le remercie également pour l'expérience enrichissante et pleine d'intérêt qu'il m'a fait vivre durant ces cinq mois au sein de l'entreprise Wallix.

Je remercie monsieur Matthieu Huin pour le temps qu'il m'a consacré tout au long de cette période de stage, sachant répondre à toutes mes interrogations.

Je remercie monsieur Jean-Noël de Galzain, président de Wallix, qui, avec l'appui de monsieur Fabien Boucher, a facilité mon insertion professionnelle en me proposant une embauche après mon stage.

Finalement, je remercie l'ensemble du personnel de Wallix pour leur accueil sympathique et leur coopération professionnelle qui ont favorisé mon intégration dans l'entreprise.

II. Résumé

Ce document présente le stage de fin d'études effectué par Lahoucine BENLAHMR, étudiant de l'université Pierre et Marie Curie. Ce stage a eu lieu au sein du pôle recherche et développement de l'entreprise Wallix, basée à Paris, du 15 avril au 15 septembre 2012.

En tant que stagiaire informatique, j'ai eu l'occasion de participer à un projet de grande envergure dans le contexte du *Cloud computing*. Ce rapport présente Wallix, le projet Résilience, l'objectif du stage et la réalisation technique effectuée.

Mots Clés : *Cloud computing, SaaS, PaaS, IaaS, SlapOS,* Buildout*, systèmes de fichiers distribués , MongoDB, GridFS, Ceph, GlusterFS, HDFS, indexation, Apache Solr, coordination, Zookeeper.*

III. Généralités

1. Présentation de l'entreprise Wallix

Wallix est éditeur de solutions de sécurité informatique, spécialisé dans la traçabilité et la sécurisation des accès aux systèmes d'information des entreprises.

Wallix propose une gamme logicielle et matérielle, qui permet de contrôler les risques liés aux accès internes et externes aux SI, aux serveurs ainsi qu'aux applications des entreprises, et de tracer les actions des utilisateurs à privilège avec le WAB (Wallix AdminBastion) ainsi qu'avec un boîtier de collecte et d'analyse des logs de connexion : la Wallix LogBox.

L'offre est distribuée à travers un réseau d'intégrateurs et de revendeurs à valeur ajoutée en France, au Benelux, en Suisse, au Royaume-Uni, en Afrique du Nord et au Moyen Orient. Elle s'adresse aux directeurs informatiques et aux responsables de la sécurité informatique qui ont besoin d'améliorer la gouvernance de la sécurité informatique de l'entreprise, en conformité avec les réglementations en matière de gestion des risques informatiques dans les secteurs de la finance, l'industrie, la défense, la santé ou le secteur public.

Implantée en France et en Angleterre, Wallix est une entreprise innovante, lauréate PM'UP de la région Île-de-France, labellisée OSEO Excellence, et membre du Pôle de compétitivité Systematic Paris-Région, dont elle a récemment reçu le label Champion du Pôle. Elle est soutenue par les fonds d'investissement Access2Net, Sopromec, Hedera, Venturi Capital, Auriga Ventures et TDH.

2. Le Cloud computing

Le *Cloud computing* (l'informatique dans les nuages) peut être vu comme un système distribué sur un grand nombre de machines distantes. Ce système d'exploitation distribuée assure l'abstraction de l'infrastructure (matériel, réseau, etc.) et a pour rôle d'héberger et d'exécuter des applications, ou des services, mais également de stocker des données. Les entreprises spécialisées proposent la puissance de calcul et de stockage de l'information aux consommateurs. Ces ressources sont facturées d'après leur utilisation réelle. De ce fait, les consommateurs n'ont plus besoin de serveurs dédiés, mais confient le travail à effectuer à une entreprise qui leur garantit une puissance de calcul et de stockage à la demande.

Le National Institute of Standards and Technology a proposé une définition du *Cloud computing* qui reprend ces principes de base : « L'informatique dans les nuages est un modèle pratique, à la demande, pour établir un accès par le réseau à un réservoir partagé de ressources informatiques configurables (réseau, serveurs, stockage, applications et services) qui peuvent être rapidement mobilisées et mises à disposition, en minimisant les efforts de gestion ou les contacts avec le fournisseur de service. » [1]

Selon les approches des entreprises, on distingue quatre formes de *Cloud computing* :

1. les *Clouds* privés internes, gérés en interne par une entreprise pour ses besoins ;

2. les *Clouds* privés externes, dédiés aux besoins propres d'une seule entreprise, mais dont la gestion est externalisée chez un prestataire ;

3. les *Clouds* publics, gérés par des entreprises spécialisées qui louent leurs services à de nombreuses entreprises ;

4. et les *Clouds* hybrides, composés d'au moins un *Cloud* privé et d'au moins un *Cloud* public.

Le *Cloud* a émergé principalement pour répondre aux exigences de continuité et de qualité du service. Il est la mise en flexibilité (disponibilité) de quatre niveaux correspondant aux termes généralement notés ainsi :

1. SaaS (*Software as a Service*) : l'application est découpée en services ;
2. PaaS (*Platform as a Service*) : la plate-forme est granulaire ;
3. IaaS (*Infrastructure as a Service*) : l'infrastructure est virtualisée ;
4. DaaS (*Data as a Service*) : les données sont fournies à un endroit précis.

3. Les risques du *Cloud computing*

Lorsqu'une entreprise, un particulier ou un État, enregistre ses données sur le *Cloud*, il existe un risque, non seulement de les perdre en cas de panne du fournisseur, mais également de se faire espionner.

Pour le particulier, cela se traduit, par exemple, par le fait de ne plus pouvoir accéder à son compte Facebook pendant deux jours ou plus, comme cela est arrivé en septembre 2010, sans aucune autre alternative que d'attendre et d'espérer un retour du service. Les données privées sont, quant à elles, transmises à des entreprises à des fins de publicité (ex. appel sur votre numéro personnel pour vous vendre une cuisine) ou à des fins politiques (ex. arrestation d'opposants en Chine suite à l'expression d'une opinion).

Pour l'entreprise, cela signifie ne plus pouvoir accéder à sa comptabilité en ligne à un moment critique, comme l'édition du rapport financier aux actionnaires. L'espionnage économique se traduit, quant à lui, par la transmission de données commerciales confidentielles d'entreprises françaises, sur des marchés à l'export, à leurs concurrentes américaines sous couvert du *Patriot Act* ou des lois OCDE anti-corruption.

Pour un État, la centralisation des systèmes d'informations dans quelques fermes de données, ou *data centers,* fait courir le risque d'une attaque ciblée par des moyens conventionnels ou électroniques, y compris lorsqu'il recourt aux entreprises nationales les plus réputées. En 2010, le service scientifique de l'ambassade de France au Japon enregistre sur Google Calendar l'ensemble des rendez-vous de veille technologique effectuée au Japon par les entreprises et laboratoires français ; il s'est alors fait, malgré lui, l'agent de renseignement de l'intelligence économique américaine.

4. Solution

Résilience est une proposition de projet collaboratif de Nexedi, Morpho, Astrium GEO Information (ex. Spot Image), le CEA, Alcatel-Lucent Bell Labs France, l'Institut Télécom, l'Université Paris 13, l'INRIA et quatre PME du logiciel libre (Alixen, Alter Way, Wallix et Xwiki), conçu pour compléter les initiatives françaises dans le domaine du *Cloud computing* (FUI - Fonds Unique Interministériel- COMPATIBLE ONE, ITEA EASI-CLOUDS).

L'objectif du projet Résilience est de proposer un système de *Cloud computing* français, à base de technologie *open source*. Il introduit des technologies innovantes de résilience et de protection des données. L'idée d'un *Cloud computing* sur le territoire français réduit les risques d'intelligence économique, ou de perte de données, liés aux usages croissants du *Cloud* par les entreprises et les particuliers.

5. Sujet et objectif du stage

Dans le cadre d'un projet d'innovation collaboratif de type FUI, la mission est de participer au développement d'une plate-forme *Cloud* de traitement des logs et des données de *monitoring* en provenance de machines d'autres *Clouds*.

L'objectif final de la plate-forme est de pouvoir archiver les données en entrée, et de pouvoir réagir en cas de détection d'anomalie. D'un point de vue technologique, la programmation de la plate-forme sera réalisée en Python, sur un système d'exploitation Linux. Les premières phases du projet seront l'évaluation des solutions de *Cloud* les plus adaptées aux besoins et la mise en place de la surveillance des *Clouds* distants.

L'objectif du stage est la réalisation d'un applicatif de collecte / stockage de logs adapté au *Cloud* et plus particulièrement à SlapOS (*Simple Language for Accounting and Provisioning Operating System*), tout en proposant une nouvelle architecture logicielle de la LogBox. La solution réalisée doit être entièrement distribuée et tolérante aux fautes, que ce soit au niveau du stockage ou au niveau des traitements effectués sur les logs.

Le stage s'est déroulé au sein de l'équipe LogBox de Wallix, avec des participations à des *sprints* avec les partenaires du projet Résilience.

6. La LogBox

Les logs sont les premiers témoins d'un événement sur un équipement d'un système d'information. Qu'ils proviennent d'applications, de systèmes d'exploitation, d'équipements réseau ou de sécurité, ils sont utilisés pour détecter les failles de sécurité, les traces d'activité inhabituelle ainsi que les défaillances matérielles ou logicielles. Les logs sont des sources d'information qui aident ainsi les administrateurs à comprendre les origines d'un événement et à réagir en conséquence.

Toutefois, l'hétérogénéité des formats de logs générés par l'infrastructure informatique rend difficiles la lecture et l'analyse des logs reçus. Même si la plupart des messages comprennent une date et un message dépendant de l'agent ayant généré le log, la collecte, l'analyse et l'exportation sous forme de rapports explicites et lisibles leur interprétation est souvent difficile pour les administrateurs systèmes et réseaux. Le manque de standardisation, la variété des sources possibles, les obligations légales et le volume journalier de logs produits par un système d'information, rendent rébarbatives l'analyse et la supervision des traces d'audit, pourtant indispensables pour la sécurité et le bon fonctionnement des systèmes d'information.

Wallix a developpé Wallix LogBox (WLB) pour répondre à ces besoins. Wallix LogBox permet d'avoir un accès simplifié à tout type de log, quels que soient son format ou son origine. Grâce à la LogBox, les administrateurs système peuvent rapidement identifier et comprendre l'origine d'un incident sur le réseau, et même faire de la prévention en temps réel, en définissant des alertes basées sur la détection d'expressions régulières dans les logs. Wallix LogBox est une solution simple à installer, déployer et utiliser. Elle s'intègre facilement dans l'infrastructure informatique et fonctionne sans agent. Elle s'adapte à la diversité des logs générés par le réseau.

6.1. Collecte

La LogBox est capable de collecter à la volée des lignes de logs en provenance de sources variées : systèmes d'exploitation, logiciels, serveurs, équipements réseaux, etc. Elle intègre, en standard, les protocoles de transport de logs les plus répandus du marché : Syslog (Unix / Linux, équipements réseaux via UDP, TCP ou SSL), SNMP, LEA, SSH, SCP et FTP.

6.2. Normalisation

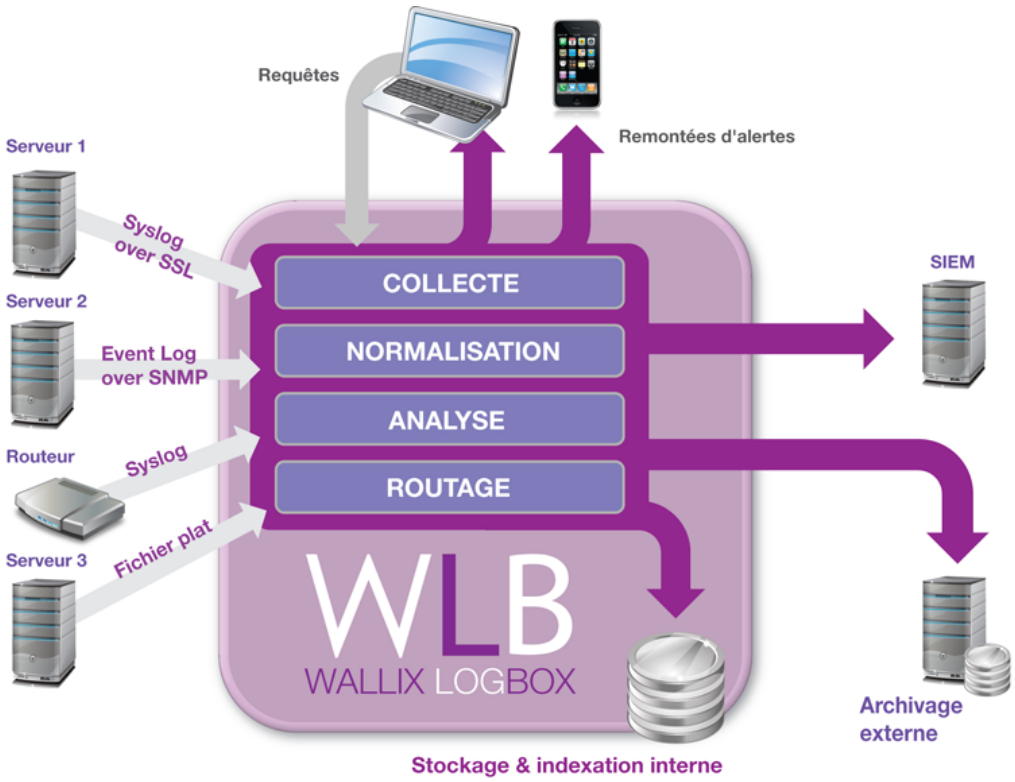
La normalisation permet de rendre un log intelligible, de lui donner un sens. Elle consiste à lui appliquer une série de filtres qui convertissent en *tags* les informations qu'il contient. Ces *tags* peuvent servir de critère de routage ou de recherche de logs.

6.3. Indexation & filtrage

Si l'indexation est activée sur la LogBox, il est possible d'effectuer des recherches en mode « *plain text* » sur les logs indexés, ainsi que sur les *tags* issus de la normalisation. Ces recherches peuvent porter sur une ou plusieurs expressions présentes dans le corps du log. Il est également possible de filtrer les logs en lançant des requêtes plus élaborées, mélangeant plusieurs paramètres de recherche. Ces requêtes peuvent être sauvegardées et exécutées ultérieurement.

6.4. Règles d'accès

L'administrateur de la LogBox peut limiter l'accès aux logs à certains types d'utilisateurs. Ces règles sont utiles pour garder des données concernant certaines machines ou applications confidentielles.



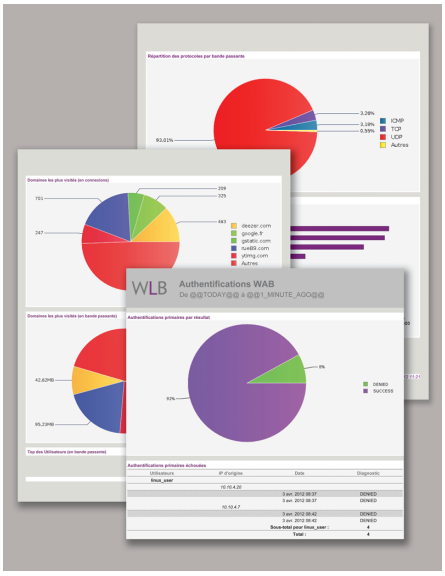
6.5. Analyse & alerte

La LogBox peut être configurée pour envoyer un *e-mail* d'alerte dès lors qu'elle repère un log contenant un *tag* remplissant certains critères. Les conditions d'envoi des alertes en temps réel sont définies par l'administrateur LogBox, et peuvent porter sur tout *tag* défini par le processus de normalisation.

6.6. Routage & stockage

Les logs peuvent être routés vers différentes destinations, via de nombreux protocoles de transport. Le routage peut se faire en fonction de critères fins, donnant à l'administrateur de la LogBox un contrôle total du réacheminement des logs collectés. Cette fonctionnalité est particulièrement intéressante pour l'archivage des logs sur des machines tierces ou pour l'envoi vers une solution d'analyse et de corrélation (SIEM - *Security Information and Event Management*).

6.7. *Reporting* détaillé



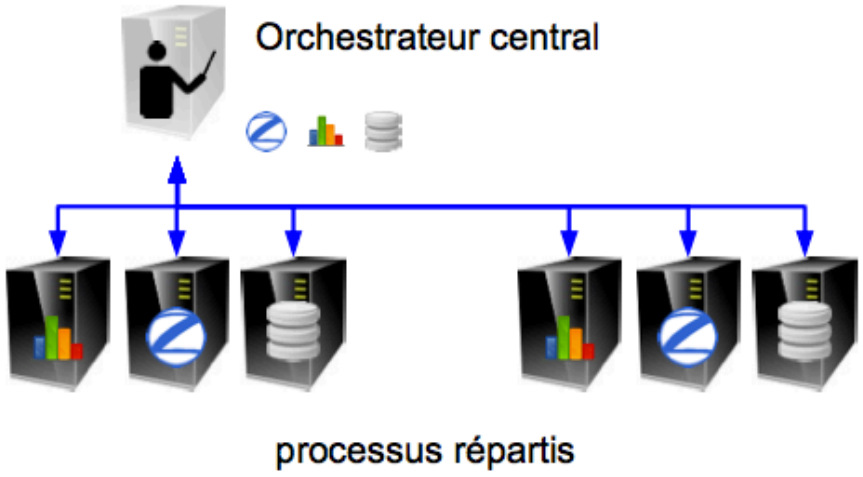
La Wallix LogBox intègre en standard des fonctionnalités de *reporting* clair, précis et détaillé. Elle permet de générer des rapports automatiques retraçant les événements de sécurité sur une multitude d'équipements réseau (*firewall*, serveurs SSH, proxy web, serveurs web, données syslog), sur plusieurs périodes données (quotidienne, hebdomadaire ou sur une période définie au choix). L'administrateur bénéficie donc d'une visibilité accrue, précise et en temps réel, de tout ce qu'enregistrent ces éléments de sécurité, essentiels au bon fonctionnement du réseau informatique. En cas de problème, les actions correctives ne sont que plus faciles à mener. L'administrateur sait d'où vient le problème et peut intervenir précisément et efficacement. Des modèles de *reporting* sont également disponibles et permettent de produire des rapports sur-mesure.

6.8. Une technologie sans agent, simple à déployer

La LogBox fonctionne sans agent spécifique ni sur les équipements du réseau ni sur les postes de travail, ce qui permet un déploiement et une exploitation quotidienne simples et rapides.

7. SlapOS

SlapOS est un système d'exploitation distribué et *open source* qui fournit un environnement permettant d'automatiser le déploiement des applications. Il est créé par ViFiB, puis utilisé par Nexedi pour héberger plusieurs milliers d'ERP (*Entreprise Resource Planning*) pour PME sur le *Cloud*. SlapOS est inspiré des travaux [2] de Christophe Cérin sur les intergiciels de coordination de grilles.



Architecture centralisée de SlapOS

SlapOS est basé sur une vision de « tout est processus ». Il transpose à l'orchestration de processus sur le *Cloud* une partie des principes de BonjourGrid [2], conçus à l'origine pour la coordination de grilles.

L'objectif premier de SlapOS est la performance économique et environnementale. Il se contente de la sécurité, par utilisateur et par groupe, d'un système Unix pour exécuter un grand nombre de processus sur un même serveur. On obtient ainsi un coût d'exploitation bien plus bas qu'avec une approche de machines virtuelles et de multiplication des systèmes de fichiers.

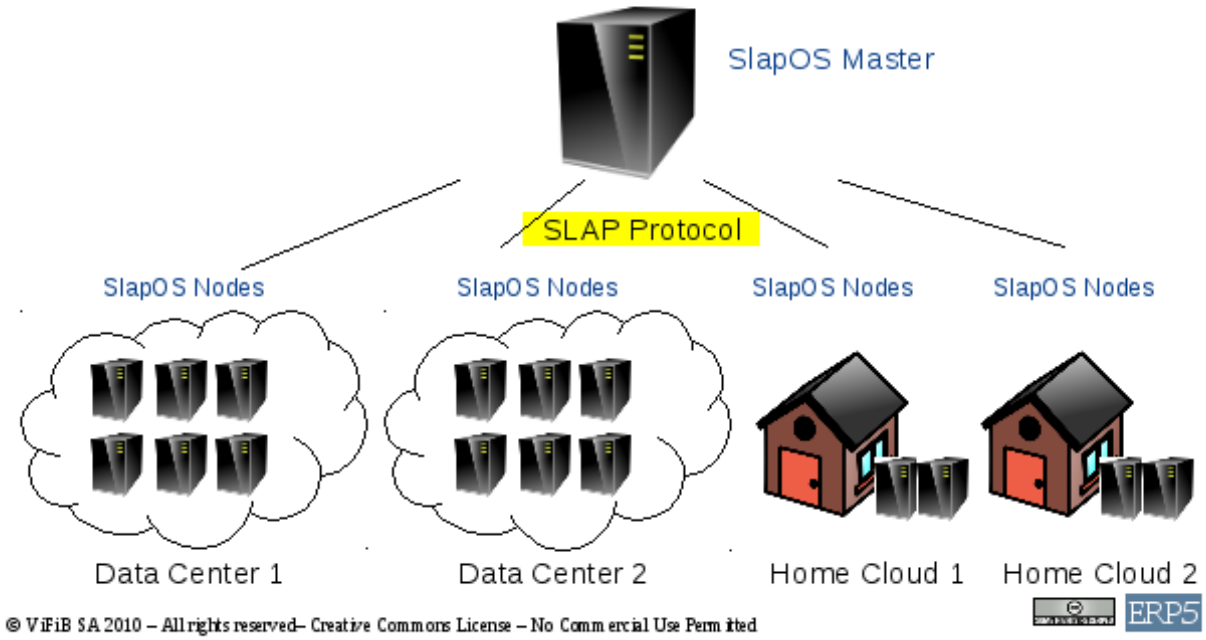
7.1. L'architecture de SlapOS

La première idée de conception de SlapOS est de considérer que « tout est processus ». Ainsi, le système est basé sur la collection d'un ensemble de processus qui communiquent entre eux grâce à des services internet utilisant des protocoles de communication.

L'architecture de SlapOS est constituée de deux types de composants : des SlapOS *masters* et des SlapOS *nodes*. Le SlapOS *master* indique au SlapOS *node* quel logiciel doit être installé et quelle instance d'un logiciel spécifique sera déployée. Il agit comme un annuaire centralisé de tous les SlapOS *nodes* : il sait où ils sont situés et connaît tous les logiciels qui y sont installés.

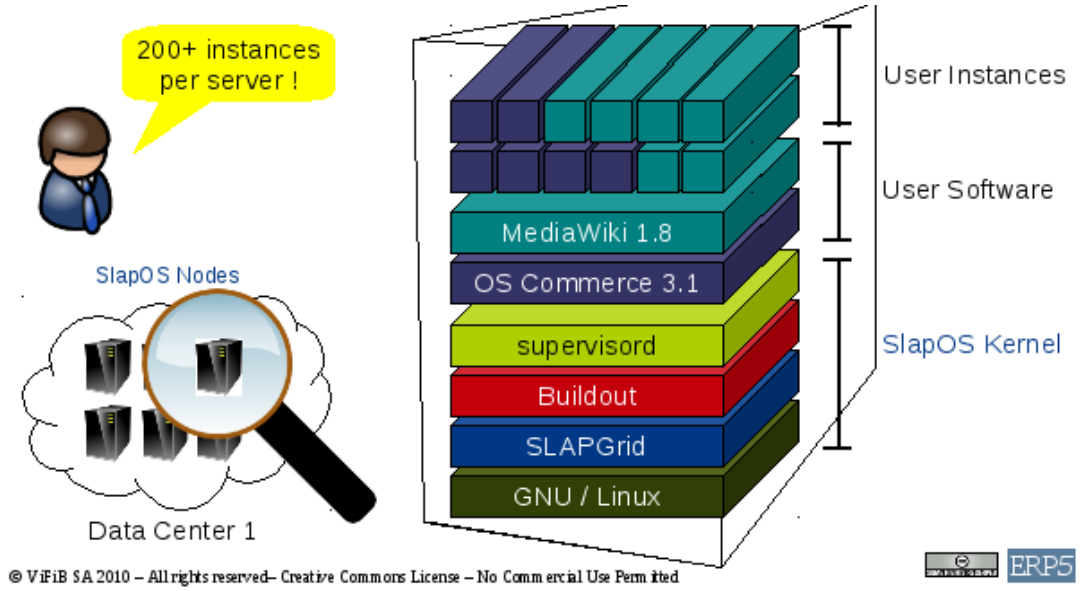
Le SlapOS *node* peut être hébergé sur un ou plusieurs *data centers,* ou encore sur un ordinateur local ; il communique avec le SlapOS *master* à l'aide du SLAP *protocol*, et le renseigne de façon périodique sur l'état des ressources dont il dispose. Essentiellement, le rôle du SlapOS *node* est d'installer et d'exécuter des processus tandis qu'un *master* a pour rôle d'allouer les processus aux SlapOS *nodes*.

SlapOS est constitué de trois éléments : SlapOS *master*, SlapOS *node* et SLAP *protocol*. Un *master* et un *node* peuvent être très distants. Ils peuvent s'échanger les données permettant de définir les capacités du *node*, collecter la liste des logiciels à installer, collecter la liste des partitions à configurer et des informations nécessaires à la comptabilité. La figure suivante montre l'architecture générale de ce système.



7.1.1. Le SlapOS *node*

SlapOS *node* s'exécute sur un noyau appelé SlapOS *kernel*. Il est constitué d'une distribution minimale du système GNU/Linux (GNU *is* Not Unix - acronyme récursif), d'un *daemon* nommé SLAPGrid, d'un environnement d'amorçage des applications appelé Buildout et de Supervisord [3], qui sert à contrôler les processus en cours d'exécution. La figure suivante présente l'architecture d'un SlapOS *node*.



De temps en temps, SLAPGrid reçoit du *master* une requête pour installer un logiciel. Celui-ci télécharge alors le fichier de description du logiciel, que l'on appelle Buildout profile, et lance le processus d'amorçage Buildout qui va installer le logiciel. Buildout est un système de compilation, développé en Python, et utilisé pour créer, assembler et déployer des applications composées de plusieurs pièces, dont certaines peuvent ne pas être basées sur Python. SLAPGrid permet donc de créer une configuration Buildout et de l'utiliser pour reproduire le logiciel plus tard. Il est capable d'exécuter un programme C, C++, Ruby, Java, Perl, etc. Il joue un rôle que l'on peut rapprocher de celui de GNU Make1.

SLAPGrid peut aussi recevoir du *master* une requête demandant de déployer une instance d'un logiciel. Il utilise alors Buildout pour créer tous les fichiers (fichiers de configurations et programmes à exécuter) nécessaires puis, à l'aide de Supervisord, il lance le démarrage de tous les processus utiles. Supervisord, quant à lui, est un système client / serveur qui permet de contrôler plusieurs processus sur des systèmes d'exploitation UNIX.

Un logiciel sur un SlapOS *node* est appelé *software release* et est constitué de l'ensemble des programmes et composants nécessaires à son fonctionnement. À partir d'un *software release*, on peut créer plusieurs instances du logiciel correspondant, qu'on appelle *software instance* et les exécuter dans une ou plusieurs partitions. Le concept de *software instance* renvoie à l'idée selon laquelle un serveur peut exécuter, de façon indépendante, un nombre élevé de processus d'un même logiciel. Puisque ces processus utilisent la même mémoire partagée, l'empreinte mémoire est surchargée et permet d'exécuter une autre instance avec des ressources mémoires minimales, contrairement au principe de virtualisation. Il est donc possible d'exécuter sur un SlapOS *node* plus de 200 instances d'un même logiciel.

Lorsqu'une instance se rend compte qu'elle est sur un *node* cassé (mauvaise connexion internet ou surcharge), elle peut demander au *master* de se faire ré-instancier sur un autre *node*.

7.1.2. Concept de partition : *computer partition*

Une partition SlapOS, ou *computer partition,* peut être vue comme un conteneur léger ou une enceinte close. Elle fournit un niveau d'isolation raisonnable (inférieur à celui d'une machine virtuelle), basé sur la gestion des utilisateurs et des groupes par le système d'exploitation hôte. Chaque partition est constituée d'une adresse IPv6, un nom d'utilisateur de la forme « SlapuserN » et un répertoire dédié (généralement /srv/slapgrid/slappartN). SlapOS est configuré pour fonctionner avec IPv6, l'un des avantages étant la possibilité d'avoir un très grand nombre de partitions. Une partition est destinée à contenir une seule application. Celle-ci peut être « tout *software instance* ». Le *software instance* est alors accessible depuis l'adresse IP de la partition. Seul l'utilisateur SlapuserN a les droits de lecture et d'écriture sur les données de la partition N. De même, les processus de la partition sont exécutés et contrôlés par l'utilisateur SlapuserN. Toutes ces règles permettent d'assurer une certaine sécurité et évitent des accès non autorisés venant, par exemple, des processus d'une autre partition. Puisque SlapOS s'exécute en mode administrateur (« *root* »), il a donc la possibilité d'accéder et de paramétrer la partition N sans avoir besoin d'être SlapuserN.

7.1.3. Le SlapOS *master*

Le SlapOS *master* se compose d'un SlapOS *kernel*. Il garde la trace de tous les SlapOS *nodes* disponibles, liste les *softwares releases* et *softwares instances* disponibles, et définit comment et sur quel SlapOS *node* ils peuvent être installés ou déployés.

Si le *master* se rend compte, après un certain temps, qu'un nœud ne répond plus, il considère que ce nœud n'existe plus. Les instances du nœud défaillant sont alors déployées sur un autre nœud. Par ailleurs, si une instance met trop de temps à passer en « *started* » (ce qui signifie qu'elle a démarré), le *master* considère qu'elle est cassée et peut demander sa réattribution ou, s'il y a trop d'erreurs, considérer que le *software release* qui génère les erreurs est cassé.

SlapOS génère un certificat X509 pour gérer chaque SlapOS *Node* connecté au *master*. Tout utilisateur, logiciel ou SlapOS *Node*, peut communiquer avec le *master* en utilisant son certificat X509 pour s'identifier.

7.1.4. SLAP *protocol*

Le SLAP *protocol* est le format de données échangées entre un SlapOS *Master* et un *node*. Il est basé sur le protocole HTTP et sur le format de données XML.

Après son démarrage, le SlapOS *Node* contacte le *master* pour l'informer que le processus de démarrage est terminé et lui donner la liste des partitions disponibles (en particulier l'identifiant et l'adresse IPv6).

Toutes les cinq minutes, le SlapOS *node* demande la liste des logiciels à installer. Le *master* retourne alors la liste complète de ces logiciels. Le SlapOS *node* demande aussi la liste des partitions à configurer pour le déploiement des logiciels et le master va retourner cette liste. Pour optimiser les traitements, SLAPGrid reconfigure uniquement les partitions qui ont changé depuis la dernière exécution.

8. Buildout

8.1. Présentation de la technologie Buildout

Comme nous avons pu le constater dans le chapitre précédent, SlapOS est essentiellement basé sur Buildout. Il intervient dans plusieurs fonctionnalités. Tout développeur SlapOS est confronté à cette technologie : les profils et les recettes utilisés pour l'automatisation des tâches, la compilation et le déploiement sont basés sur cette technologie.

Buildout ressemble à un bac à sable (*sandbox*) mais avec la différence qu'il s'agit plutôt d'un chantier de construction. Il fournit une isolation et un contrôle de l'environnement d'exécution de l'application.

8.2. Concept et principe de fonctionnement de Buildout

D'un point de vue utilisateur, Buildout prévoit essentiellement trois éléments :

– une collection de modules ;
– des scripts qui font usage de ces modules ;
– une arborescence de fichiers appelés « *parts* ».

Buildout est constitué d'une collection de recettes extensibles, chargées de la conduite du processus d'assemblage. Il exploite, dans la gestion, des paquets Python :

- Setuptools [4], qui est une bibliothèque tierce qui fournit des fonctionnalités au-dessus de Distutils (qui permet de déployer des modules Python), dont un *script* d'installation de paquets Python disponibles sur www.pypi.com, qui est le dépôt central où sont stockés tous les *packages* Python fournis par la communauté Python.

- Les « *eggs* » [5] qui sont des *packages* Python distribuables.

8.2.1. La spécification

La spécification est un fichier texte (fichier d'extension .cfg) qui énumère toutes les pièces appelées « *parts* » (la notion de *parts* est expliquée dans la section qui suit) qui vont être assemblées. Pour chaque *part*, elle donne les noms des différentes recettes qui vont intervenir et les configurations à utiliser par la recette. On peut ajouter dans le fichier des contraintes de version des *eggs*, spécifier des détails sur les endroits où les télécharger automatiquement. Si une *part* est retirée de la spécification, elle sera désinstallée du répertoire de déploiement. Si une recette d'une *part* est modifiée, la *part* sera désinstallée, puis réinstallée.

8.2.2. Qu'est-ce qu'une « *part* » ?

Une *part* est tout simplement un objet manipulé par Buildout. Il peut être n'importe quoi, comme un paquet Python, un programme, un répertoire, ou même un fichier de configuration. La *part* a un nom unique au sein de la spécification. Elle possède son dossier dans le répertoire de déploiement et est installée dans ce dossier. Une *part* peut faire référence ou hériter d'une autre *part* au sein de la même spécification et peut être installée, mise à jour ou désinstallée sur une série de compilations. Chaque *part* est définie par une recette (expliquée dans la section qui suit), qui contient sa logique de gestion, ainsi que des données utilisées par ladite recette spécifique dans cette *part*.

8.2.3. Qu'est-ce qu'une recette ?

Une recette est un objet qui sait comment installer, mettre à jour et désinstaller une *part* précise. Buildout lui-même est construit à partir de recettes. Les recettes elles-mêmes sont des *egg*s. Lorsqu'une recette est référencée dans la spécification, Buildout va automatiquement la rechercher et l'installer dans l'environnement Buildout (cet environnement peut constituer l'isolation évoquée plus haut). Une recette peut contenir plusieurs sous-recettes : elles seront accessibles en tant que points d'entrée distincts de l'*egg*. Un ensemble de recettes de démarrage est associé à Buildout et contenu dans l'*egg* nommé zc.recipe.egg. Un grand nombre d'*eggs* sont publiés sur : http://pypi.python.org

8.3. Rôle de Buildout dans SlapOS

SlapOS utilise Buildout pour décrire comment un logiciel sera compilé, installé et déployé. Il participe à l'isolation des *software releases,* d'une part, et des environnements d'exécution des *software instances,* d'autre part. Un *software release* ne devrait pas utiliser les ressources ou composants d'un autre *software release* ou du système. En utilisant Buildout on est donc en mesure de compiler tous les composants et toutes les dépendances dont le logiciel a besoin, et créer ainsi un environnement dans lequel le logiciel sera installé et va s'exécuter. Ce principe permet, par exemple, d'avoir deux logiciels qui utilisent chacun un même composant (Apache, par exemple) mais dont les versions diffèrent.

IV. Un collecteur de logs pour le *Cloud*

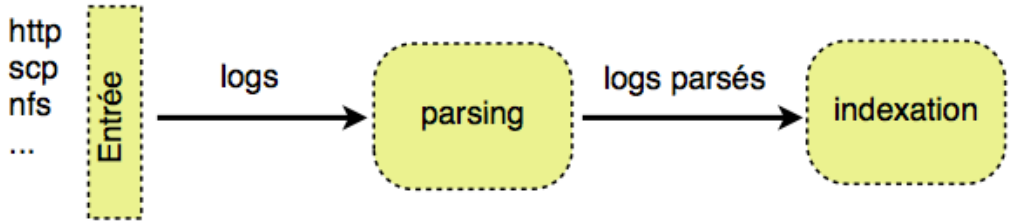
La LogBox de Wallix, comme elle a été décrite dans les sections précédentes, se présente sous forme d'*appliance* (traduit en français par équipement ou serveur dédié, c'est effectivement un serveur dédié sur lequel est pré-chargé un système logiciel complet), qui est basé sur une distribution Linux de type Debian. Cette *appliance* utilise de part et d'autre des *packages* Debian. L'intégration de la LogBox dans son état actuel, dans le système SlapOS, est impossible. Les applications intégrées à ce dernier doivent répondre à certaines exigences, à savoir : ne pas dépendre d'une distribution particulière d'un système et pouvoir s'installer et s'exécuter dans un environnement, sans nécessiter les droits *root*. Comme décrit plus haut, les applications intégrées à SlapOS s'installent et s'exécutent avec les droits et les répertoires du SlapuserN. De plus, l'application à intégrer

doit fortement supporter l'IPv6, ce qui n'est pas le cas de la LogBox actuelle - et d'une grande proportion de solutions logicielles disponibles actuellement sur le marché payant ou *open source*.

Dans cette partie il sera question de présenter le processus d'analyse, les études et choix de solutions utilisées, l'architecture et la réalisation d'une solution adaptée pour la collecte de logs pour le *Cloud*.

9. Analyse des besoins du collecteur de logs pour le *Cloud*

Dans le cadre du projet Résilience, et afin de répondre aux besoins de la LogBox pour le *Cloud*, il a été question de réaliser, pendant mon stage, une version de la LogBox adaptée au *Cloud* (appelé aussi collecteur de logs pour le *Cloud*), supportant l'IPv6, et avec des fonctionnalités allégées.



La figure ci-dessus représente une vision très générale et très abstraite du processus de collecte, de *parsing* et d'indexation de logs dans la LogBox. La LogBox pour le *Cloud* a pour but de profiter de la puissance du *Cloud* pour optimiser et rendre performant ce processus de collecte et de traitement des logs. La solution se doit d'être entièrement distribuée, que ce soit au niveau de la collecte, du *parsing* ou de l'indexation. Elle se veut aussi tolérante aux fautes, respectant des aspects de sécurité et offrant un espace de stockage et une indexation distribuée.

9.1. Les besoins du collecteur de logs pour le *Cloud*

Il s'agit des besoins qui caractérisent le système et ses fonctionnalités. Dans le cadre de la LogBox pour le *Cloud*, ces besoins sont :

- **indexation répartie de logs :** l'indexeur de logs se doit d'être réparti et de permettre une extensibilité de l'espace de stockage des logs indexés, par l'ajout de nouveaux nœuds. Il doit aussi permettre une recherche répartie sur plusieurs nœuds et une stratégie de réplication des différentes parties composant les logs indexés ;

- **stockage réparti de logs :** la solution proposée doit fournir un espace de stockage extensible, distribué, tolérant aux fautes et sans aucun point de défaillance. Ce besoin

est justifié par la grande quantité de logs collectés sur un ensemble de machines sur le *Cloud*. Bien que les logs puissent être stockés par l'indexeur de logs, le format de ces derniers, après l'indexation, n'est pas adapté pour l'horodatage, la signature et le chiffrement des logs, qui peuvent constituer une preuve juridique sur la validité des logs collectés ;

- **authentification :** proposer un mécanisme d'authentification pour déclarer un ensemble de sources de collecte de logs de confiance ;

- **collecte de logs sécurisée :** la collecte de logs doit se faire au travers d'un module d'entrée avec un protocole simple. Il faut surtout sécuriser la communication pendant le processus de réception de logs et n'accepter de communiquer qu'avec les identités déclarées ;

- **processus de traitement distribué :** le processus de *parsing*, collecte et indexation de logs doit se faire de façon répartie sur un ensemble de ressources machines fournies par le *Cloud*. Ce processus se doit d'être bien coordonné et ne tolère aucune perte de lignes de logs ;

- **mécanisme de suppression de logs :** la solution doit proposer un mécanisme de suppression de logs s'appliquant sur les deux *backends* (indexeur et fichiers plats) ;

- **gestion de la configuration :** vu la nature distribuée de la solution, la configuration et la gestion de présence de l'ensemble des composants doit se faire de manière dynamique et ne peut pas être codée en dur à cause de la dynamicité de la solution ;

- **support de l'IPv6 :** pour bien intégrer la solution à SlapOS, les composants choisis et développés doivent principalement supporter et utiliser des adresses IPv6 pour communiquer ;

- **plate-forme d'exécution :** la LogBox doit pouvoir s'installer dans SlapOS.

10. Conception du collecteur de logs pour le *Cloud*

Le collecteur de logs pour le *Cloud* a été réalisé avec la même philosophie que SlapOS, selon laquelle « tout est processus ». Il est construit à base de composants. Chaque composant joue un rôle spécifique dans le système. Il définit clairement les services qu'il offre et les services requis pour accomplir sa fonction. Le développement de

logiciel basé sur des composants augmente considérablement la fiabilité des systèmes puisque leur construction est basée, en partie, sur des composants testés et certifiés. De même, les coûts du développement sont généralement réduits car une partie des composants est simplement réutilisée. L'étape de maintenance se réduit, quant à elle, à un remplacement de composants, ce qui facilite sa tâche et favorise l'évolution du système. La combinaison de l'approche basée sur des composants et de la séparation des préoccupations, permet un développement rapide des systèmes dont les préoccupations non fonctionnelles sont factorisées en dehors de ces composants, ce qui permet d'avoir des systèmes évolutifs, adaptatifs et faciles à maintenir.

10.1. Architecture de l'approche

Pour répondre aux besoins spécifiés précédemment, en se basant sur une solution à base de composants, les sous-chapitres qui vont suivre traiteront les principaux composants et l'architecture de l'approche.

10.1.1. Un modèle de coordination

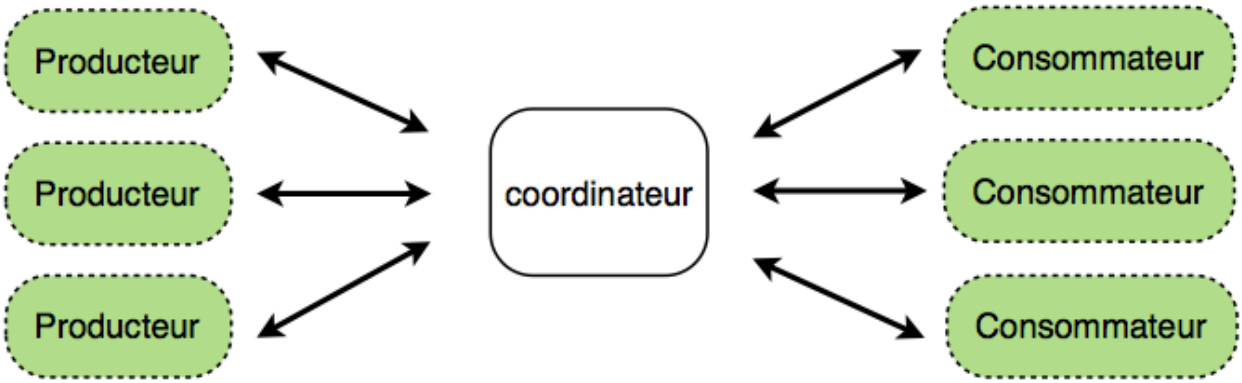
Un modèle de coordination bien adapté est nécessaire pour permettre un traitement distribué des logs récoltés à partir des agents de collecte. Le modèle choisi est le modèle producteur / consommateur, qui se base sur la gestion d'une file de tâches à traiter. Il permet à des producteurs d'ajouter des fichiers de logs collectés dans la file pour qu'ils puissent être traités par les consommateurs.

Ce modèle rend possible un traitement hautement distribué pour la collecte et le traitement des logs, puisque le nombre des producteurs et des consommateurs n'est pas limité, et ils ne sont pas obligés de s'exécuter sur la même machine.

On identifie donc pour l'architecture trois premiers composants :

- composant producteur ;
- composant consommateur ;
- composant coordinateur (file de tâche).

Le schéma ci-dessous montre une vue d'ensemble des trois premiers composants de l'architecture de la solution.



10.1.2. Un système de stockage

Le composant de stockage communique directement avec les producteurs et les consommateurs. Les producteurs l'utilisent pour stocker un fichier collecté et rajouter son chemin d'accès dans la file de tâches pour qu'il puisse ensuite être joint par le composant consommateur.

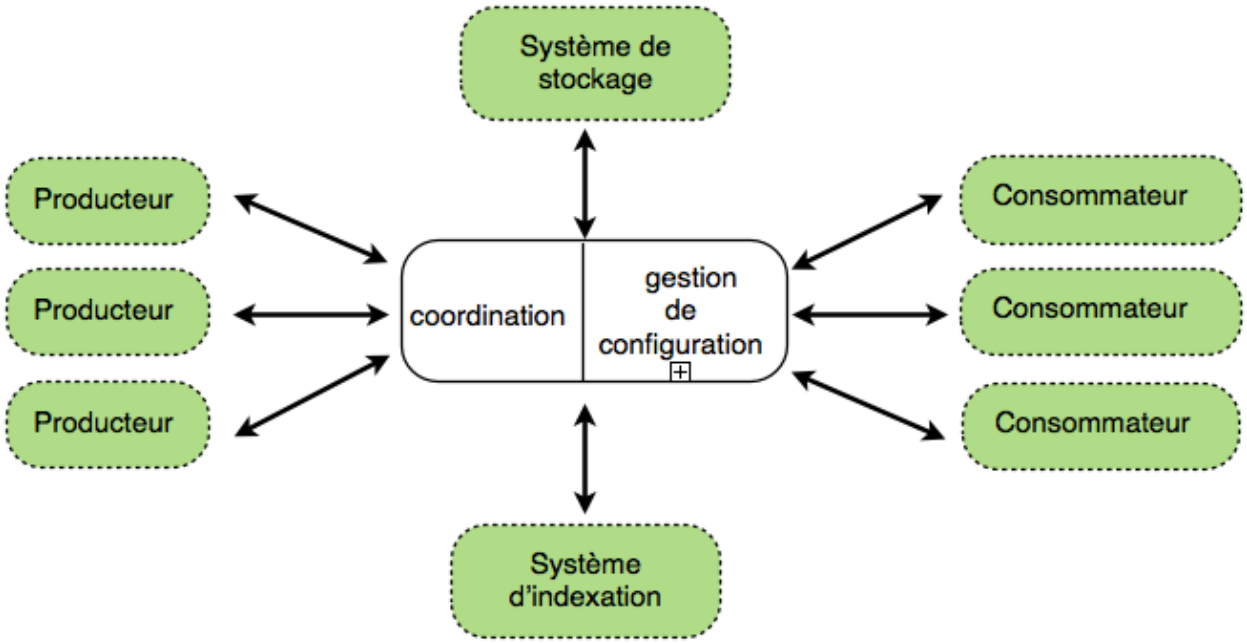
10.1.3. Un système d'indexation

Le composant d'indexation, quant à lui, communique directement avec les consommateurs qui s'occupent de *parser* et d'indexer les logs à traiter.

10.1.4. Une gestion de l'ensemble des composants

Comme on vient de le voir dans la première spécification de l'architecture de la solution, on a besoin de faire communiquer des composant entre eux. Or, la configuration de ces composant peut changer au cours du temps et on doit pouvoir détecter leur présence. Dans ce type de système hautement distribué il n'est donc pas totalement possible de coder en dur les différentes configurations.

Il faut donc un composant pour centraliser les différentes configurations et il doit pouvoir détecter la présence des autres composants. Ce choix d'architecture permet alors une grande flexibilité à la solution.
Le schéma ci-dessous représente une architecture globale de la solution, montrant les différents composants qui ont été recensés.



V. Etude et choix de solutions

11. Stockage réparti

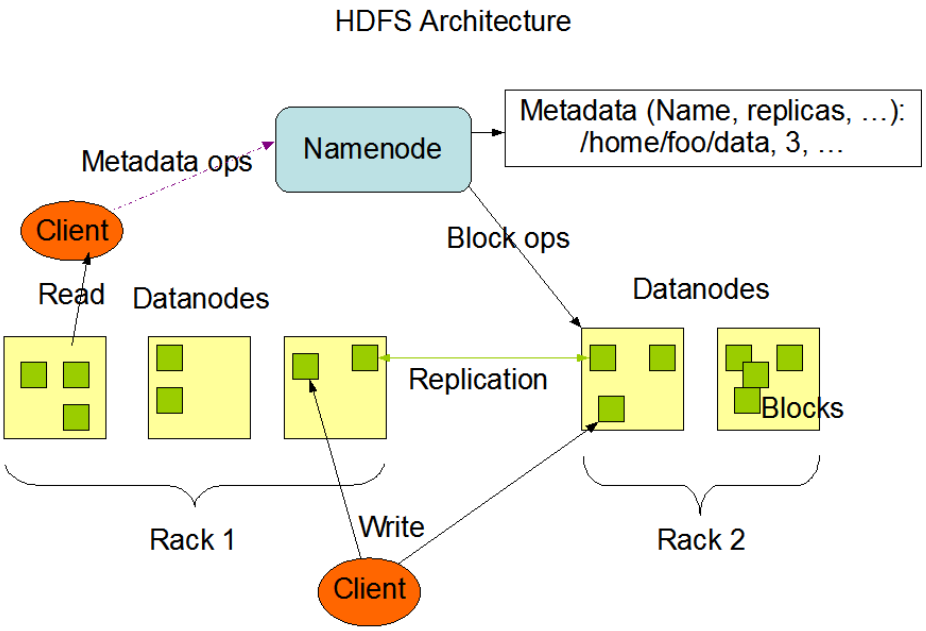
Quand la taille des données dépasse la capacité d'une seule machine physique, il devient nécessaire de les répartir sur un ensemble de machines séparées.

Les systèmes de fichiers qui gèrent le stockage à travers un réseau de machines sont appelés systèmes de fichiers distribués. Du fait de la communication en réseau, ce type de système de fichiers est plus complexe à gérer qu'un système de fichiers normal. Par exemple, l'un des plus grands défis est de rendre ce système de fichiers tolérant aux fautes, sans perdre de données.

Le système de fichiers choisi doit répondre aux besoins de stockage spécifiés précédemment. Il faut aussi prendre en compte la limitation imposée par SlapOS au niveau des droits utilisateurs, puisque ce dernier se base sur les droits pour isoler les instances d'un *software*. Ce qui implique que, dans SlapOS, il n'est pas autorisé d'exécuter des instances avec le droit *root*. Pour répondre à cette contrainte, le champ d'étude se réduit aux systèmes de fichiers distribués qui s'exécutent dans l'espace utilisateur, donc principalement basés sur FUSE (*Filesystem in UserSpac*E/ système de fichiers en espace utilisateur) qui permet à des systèmes de fichiers, implantés en espace utilisateur, d'être intégrés au système de fichiers Unix.

11.1. HDFS (Hadoop Distributed File System)

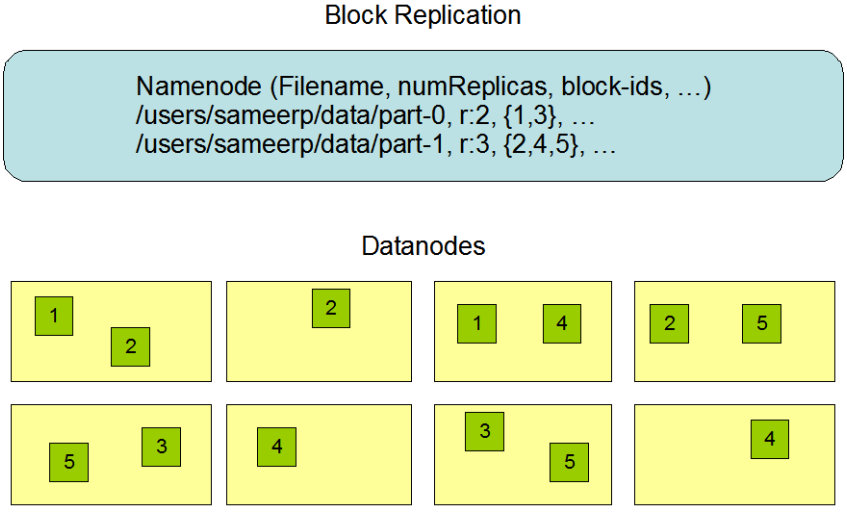
C'est un système de fichiers distribués, créé par la fondation Apache, où les blocs d'informations sont répartis et répliqués sur les différents nœuds du *cluster*.



11.1.1. Architecture du HDFS

Le HDFS a une architecture maître / esclave. Un *cluster* HDFS est composé d'un seul point appelé *namenode*, qui est le serveur maître, dont le rôle est de gérer l'espace de noms du système de fichiers et de régulariser les accès aux fichiers. En plus de ce nœud, le HDFS est composé de plusieurs nœuds *datanodes*, qui gèrent le stockage du nœud auquel ils sont attachés.

Dans le HDFS, un fichier est divisé en un ou plusieurs blocs, et ces blocs sont stockés dans un ensemble de *datanodes*. Le *namenode* exécute les opérations du système de fichiers comme l'ouverture et la fermeture d'un fichier. Il gère aussi le *mapping* des blocs vers les *datanodes*. Le *namenode* a aussi pour rôle de servir les requêtes de lectures et d'écritures du client du système de fichiers. En plus, le *datanode* exécute aussi les instructions demandées par le *namenode* qui sont : la création, la suppression et la réplication de blocs.



Le but de l'architecture du HDFS est de stocker un grand nombre de fichiers de manière fiable à travers les nœuds d'un *cluster*. Chaque fichier est stocké sous forme d'une séquence de blocs. Tous les blocs du fichier, sauf le dernier, ont la même taille. Les blocs d'un fichier sont répliqués pour permettre une tolérance aux fautes. La taille des blocs et le facteur de réplication sont configurables pour chaque fichier. Le facteur de réplication peut être spécifié à la création du fichier et peut être changé plus tard. Les fichiers, dans le HDFS, ne peuvent être écrits qu'une seule fois (*write-once*).

C'est le *namenode* qui prend toutes les décisions de réplications des blocs. Périodiquement, il reçoit de chaque nœud un « battement de cœur » (*heartbeat*) et un rapport sur les blocs. La réception d'un « battement de cœur » implique que le nœud fonctionne proprement. Le rapport de blocs, quant à lui, contient la liste de tous les blocs dans un *datanode*.

11.1.2. Conclusion sur HDFS

Le HDFS est un système de fichiers distribués, tolérant aux fautes et qui tourne en production puisqu'il est largement exploité par le projet Hadoop [7] de Apache, qui en est à l'origine.

Le HDFS fournit le module Fuse-DFS, permettant de monter le système de fichiers distribués en espace utilisateur.

Malgré les qualités du HDFS, le *namenode* représente un SPOF (*Single Point Of Failure* / point unique de défaillance) puisqu'il est le seul nœud présent dans le système qui gère les différents autres nœuds. Donc, dans le cas où ce nœud tombe, le système de fichiers ne sera plus disponible.

Ce système de fichiers ne représente donc pas le meilleur choix pour répondre aux besoins du projet.

11.2. Ceph

Ceph est un système de fichiers dont le but principal est d'être compatible POSIX et d'être complètement distribué sans un seul point de défaillance. Les données sont répliquées de façon transparente, ce qui fait de Ceph un système tolérant aux pannes.

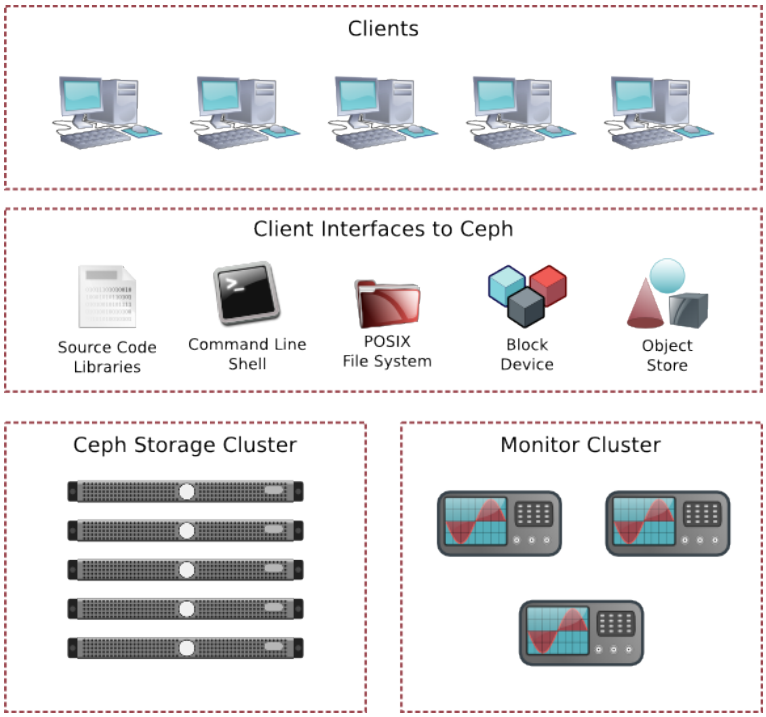
Dans Ceph, les données sont découpées, réparties sur les différents serveurs et répliquées sur x nœuds. Cette répartition lui permet d'avoir une grande tolérance vis-à-vis de la perte de nœuds stockant les données. De plus, il permet de rajouter ou d'enlever des serveurs de données pendant son fonctionnement, sans être obligé d'arrêter le service.

Ceph remplit plusieurs caractéristiques importantes :

- robustesse du stockage distribué *open source*. Ceph fournit une variété de fonctionnalités clés, généralement manquantes dans les systèmes de fichiers *open source*, telles que l'évolutivité transparente (capacité d'ajouter simplement un disque pour étendre le volume de stockage) ou la répartition de charge intelligente ;

- évolutivité en terme de charge de travail ou de capacité de stockage. Ceph a été créé pour supporter des charges de l'ordre de dizaines de milliers de clients accédant au même fichier.

11.2.1. Architecture de Ceph



La partie serveur de Ceph utilise trois types distincts de *daemon* :

- moniteur de *cluster*, qui garde une trace des nœuds actifs et défaillants ;
- serveurs de métadonnées, qui stockent les métadonnées des nœuds et des répertoires ;
- serveurs de données, qui stockent les données.

a. Moniteur de *cluster* :

Le moniteur gère l'administration centrale, la configuration et l'état du *cluster*. C'est à partir du moniteur que l'on modifie le *cluster* : ajout ou suppression de serveurs de données et métadonnées.

Les clients ont accès aux données gérées par Ceph par l'intermédiaire d'un moniteur. Il garde en mémoire une carte du *cluster*, ce qui lui permet de communiquer aux clients sur quels serveurs de métadonnées et de données se connecter pour modifier, ajouter ou supprimer des fichiers.

b. Serveurs de métadonnées :

Le *daemon* gérant les serveurs de métadonnées agit en tant que cache cohérent et distribué du système de fichiers. Il contient les données concernant l'arborescence des fichiers et leurs métadonnées (nom, taille, emplacement mémoire sur les serveurs de données). Avoir plusieurs serveurs de métadonnées présente un certain avantage car cela permet d'équilibrer la charge de travail entre les différents serveurs, et donc de gérer plus de clients au même moment.

c. Serveurs de données :

Les serveurs de données stockent les fichiers du *cluster.* Ces fichiers sont découpés en morceaux et répliqués plusieurs fois sur des serveurs différents. Cela permet une certaine tolérance vis-à-vis de la perte de serveurs. Si jamais un serveur tombe en panne, les données sont toujours accessibles car elles sont présentes sur un autre serveur.

11.2.2. Conclusion sur Ceph

D'après l'étude effectuée, Ceph paraît répondre à un bon nombre de critères requis par l'architecture du projet :

- la possibilité d'avoir plusieurs serveurs de métadonnées permet à Ceph de ne pas souffrir d'un point unique de défaillance ;

- Ceph peut supporter une grande charge de travail et offre une très grande capacité de stockage.

Cependant, Ceph n'est, à ce jour, pas utilisé dans un environnement de production. À cela s'ajoute le fait que bien que ses anciennes versions fournissent un système de fichiers en espace utilisateur, aujourd'hui, Ceph est passé en espace *kernel*, ce qui le rend plus rapide mais impossible à intégrer dans SlapOS - dû aux contraintes imposées par ce dernier, puisqu'on ne pourra ni l'installer ni l'exécuter avec le droit *root*.

11.3. GlusterFS

GlusterFS est un système de fichiers distribués libre, à la fois très simple à mettre en œuvre et dont les capacités de stockage peuvent monter jusqu'à plusieurs *petabytes* (1 000 milliards d'octets).
GlusterFS est composé de deux éléments logiciels : un serveur et un client. Il supporte plusieurs protocoles de communication tels que TCP/IP, InfiniBand RDMA (*Remote Direct Memory Access*).

Les serveurs de stockage gèrent la réplication des données stockées dans le volume distribué, permettant une reprise rapide du service en cas d'indisponibilité de l'un des nœuds. Les clients reposent sur FUSE pour monter localement l'espace de stockage fusionné, donc facilement administrable.

Le nombre de serveurs et le nombre de clients ne sont pas limités. Les performances linéaires ne sont pas choses faciles à réaliser. Avec les *clusters* de stockage, plus l'espace de stockage et le nombre de fichiers croissent, plus les performances vont se dégrader. En effet, les *clusters* de stockage utilisent un serveur de métadonnées. Le nombre de métadonnées devient tel que le *cluster* de stockage perd beaucoup, en terme de performance, à rechercher l'information concernant un fichier et à la modifier. Le serveur de métadonnées est un goulot d'étranglement au vu du nombre de nœuds clients qui veulent accéder à ce serveur. GlusterFS, lui, ne gère pas les métadonnées. L'emplacement du fichier stocké sur le *cluster* est calculé à l'aide d'un algorithme nommé *Elastic Hash*. Donc, il n'y a pas de serveur de métadonnées. Les nœuds clients vont pouvoir calculer l'emplacement d'un fichier. GlusterFS réalise, avec ce mécanisme, une répartition de charge de calcul et, par la même occasion, supprime le goulot d'étranglement et le remplace par un calcul parallèle, réalisé par les nœuds clients.

Dans GlusterFS il existe différents modes de fonctionnements d'un module :

- volume distribué : un volume est distribué sur plusieurs partitions qui sont distribuées sur plusieurs machines… Les fichiers du volume sont physiquement enregistrés sur l'une des partitions. Deux fichiers d'un même dossier n'ont aucune raison d'être sur la même partition :
- volume répliqué : un volume distribué est automatiquement répliqué n fois sur le *cluster*, n pouvant être 2 (*raid* 1), mais aussi 3 ou plus ! Ce type de volume est intéressant pour se préserver des pannes ;
- volume morcelé (*stripe*) : chaque fichier d'un volume distribué est découpé en morceaux qui sont répartis sur les nœuds. Ce genre de volume est intéressant lorsqu'on cherche de la performance sur des gros fichiers.

Il faudrait aussi noter que quelques opérations de base sont possibles à chaud sur tous les volumes :

- étendre (*expanding*) ;
- réduire (*shrinking*) ;
- migrer (*migration*) ;
- équilibrer (*rebalancing*).

On peut aussi citer quelques fonctionnalités intéressantes dans GlusterFS :

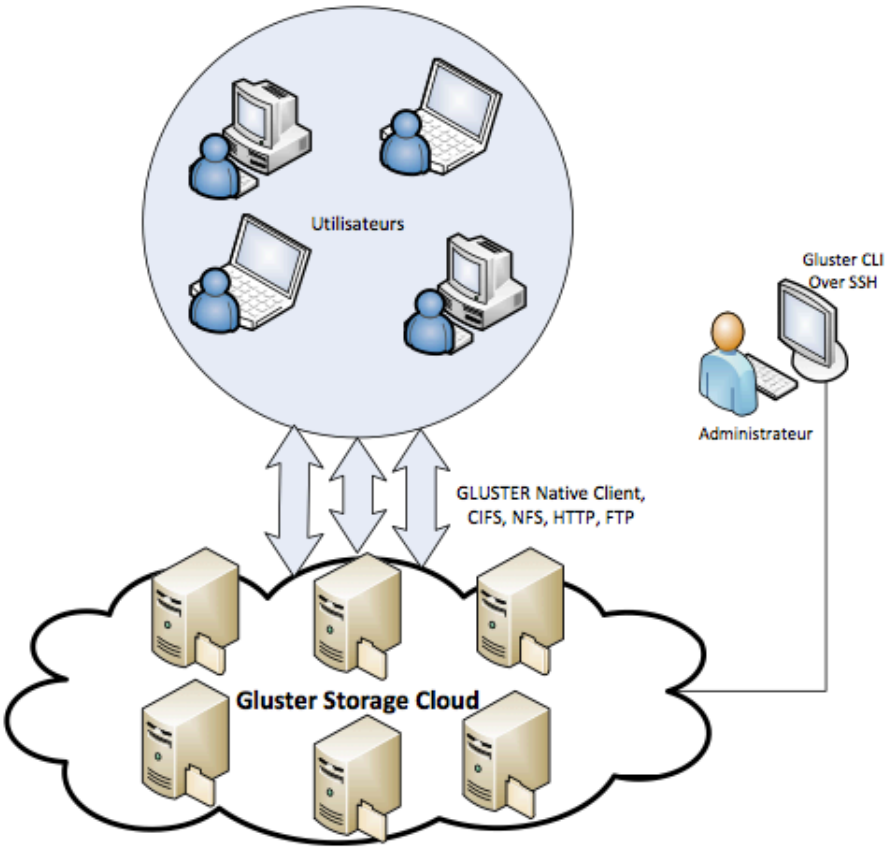
- la réplication : l'idée est d'avoir deux sites distants et de faire de la réplication asynchrone entre les deux sites, afin d'avoir un plan de reprise d'activité (PRA) possible. Il n'est pas nécessaire d'avoir un système de fichiers GlusterFS sur le site secondaire, mais si les volumes de données sont importants, cela semble logique d'avoir le même format des deux côtés. À noter qu'il est possible de cascader les sites : A→B→C, ou de répliquer le site maître vers deux sites distants ;
- les quotas : gestion de quotas par dossier permettant de limiter la taille occupée par un dossier sur un volume. En général, sur les systèmes de fichiers traditionnels, les quotas se font sur les utilisateurs ou sur les groupes. On crée donc souvent un groupe par projet (dossier) afin d'y appliquer un quota. Ce mode de gestion de quotas par

dossier ne devrait donc pas modifier fondamentalement la pratique des administrateurs système ;

- la supervision : elle permet d'obtenir des rapports d'utilisation et de performance. Ce type d'informations peut être utile pour savoir, par exemple, quelles sont les données à déplacer sur des volumes plus performants.

Un *pool* de stockage, ou un *cluster* de stockage, est construit à l'aide d'un ensemble d'ordinateurs liés par un réseau afin de regrouper leurs espaces de stockage. Ceci permet d'exploiter une partie du disque dur et de la puissance de calcul de chacun d'entre eux, afin de fournir un service en parallèle à leurs fonctions de base.

Pour construire un *cluster* de stockage, il faut exécuter Gluster File System Server sur chacun des ordinateurs que nous souhaitons lier au *cluster* pour partager un espace disque. Ensuite, il faut créer un volume. Un volume GlusterFS est constitué de briques de stockage mises à disposition par chacun des serveurs. Une brique de stockage est un répertoire sur le serveur associé avec un volume. Le client du *cluster* va pouvoir monter un volume GlusterFS dans un répertoire pour qu'il puisse stocker ses données, comme si c'était un espace de stockage local, un volume interne à la machine.



Dans la figure ci-dessus, nous pouvons voir une ferme de serveurs GlusterFS appelée *trusted pool,* ou *cluster* de stockage, qui exportent un ou plusieurs volumes. Les clients qui souhaitent accéder au volume de stockage peuvent s'y connecter à travers le réseau, à l'aide d'un logiciel client GlusterFS, Samba ou autre. Le client natif de GlusterFS est l'outil privilégié pour accéder au stockage. L'administrateur peut configurer cette ferme de serveurs en utilisant une connexion sécurisée SSH en ligne de commande, ou bien en s'identifiant en tant que *root* sur un des serveurs GlusterFS.

11.3.1 Conclusion sur GlusterFS

D'après l'étude effectuée, GlusterFS semble, de loin, être la meilleure solution répondant aux besoins du projet :

- aucun point de défaillance ;

- entièrement répartie ;

- support de l'IPv6 ;

- extensible sans interruption de service.

Cependant il s'est avéré difficile d'intégrer GlusterFS dans SlapOS sans modifier son code. En effet, bien que GlusterFS se base sur FUSE, il nécessite des droits *root* pour s'exécuter. Il a donc fallu étudier le code de GlusterFS en profondeur pour repérer les portions de code qui requièrent les droits *root* afin de pouvoir, par la suite, les modifier :

- GlusterFS utilise des ports privilégiés (compris entre 1 et 1023) ;

- GlusterFS écoute sur toutes les adresses locales. Dans SlapOS on a besoin qu'il n'écoute que sur l'adresse assignée à la partition où il s'exécute ;

- GlusterFS utilise deux adresses IPv6 différentes. Dans SlapOS il ne doit utiliser que l'adresse propre à sa partition ;

- GlusterFS stocke les logs dans un répertoire par défaut, donc non spécifique à chaque instance ; ce qui implique des problèmes de droits d'écriture puisque les partitions n'ont pas le droit d'écrire dans le répertoire dans lequel est installé le *software* (GlusterFS) ;

- Le *daemon* GlusterFSd, qui gère les briques de stockage, écoute par défaut sur l'adresse *localhost*, même si on lui a indiqué une adresse spécifique.

En contradiction de ce qui a été annoncé dans sa documentation GlusterFS ne supporte pas parfaitement l'IPv6.

À cause de son manque de documentation et des problèmes rencontrés, GlusterFS était un point de blocage pour l'avancement du projet. Le nombre de modifications du code de GlusterFS devenant de plus en plus important, cela le rend difficile à maintenir. De plus, des aspects de sécurité peuvent avoir été compromis lors de ces modifications. Nous ne pouvions donc pas continuer à modifier le code de GlusterFS sans connaître l'étendue

des dommages éventuellement causés. Après plusieurs réunions avec l'équipe SlapOS, pour trouver des solutions afin de dépasser les limitations du système SlapOS, nous avons conclu que la version actuelle de SlapOS ne supportait pas l'intégration de GlusterFS.

Vu les contraintes de SlapOS pour l'intégration des applications, il m'a été proposé, dans un premier temps, d'abandonner le composant de stockage de fichiers du collecteur de logs pour le *Cloud* et de n'intégrer que le composant d'indexation. J'ai proposé comme alternative d'utiliser le système de fichier GridFS de MongoDB, dont une étude est décrite ci-après.

11.4. GridFS de MongoDB

11.4.1. MongoDB

MongoDB est un SGBD NoSQL de type document, *open source*, proposé par la société 10gen. Sa popularité est grandissante et il est notamment utilisée par SourceForge et GitHub qui, à eux seuls, lui garantissent sa notoriété.

a. Modèle de données

Le modèle de données de MongoDB est orienté documents. Étudions ce modèle via une approche *bottom up*

Documents :

Un document, l'unité basique de MongoDB, est un ensemble ordonné de paires clés / valeur. Il est identifié par son nom. Pour faire une analogie à SQL, un document peut être vu comme une entrée. À la différence d'une entrée SQL, le nombre de champs d'un document n'est pas défini préalablement. Un exemple d'un simple document est le suivant :

```
{'rue' : 'rue de Tocqueville' , 'num' : 115 , 'ville' : 'paris' }
```

Les clés (rue, numéro, ville) sont des *strings* alors que les valeurs sont typées : dans notre exemple, 'rue de tocqueville' et 'paris' sont des *strings* et 115 est un entier.

Collections :

Une collection est un ensemble de documents (et peut donc être vu comme une table SQL). Les documents d'une collection peuvent avoir des formes très différentes, comme l'illustre l'exemple suivant :

```
{'rue' : 'rue de Tocqueville' , 'num' : 115 , 'ville' : 'paris' }

{'type' : 'entreprise' , 'nom' : 'Wallix'}
```

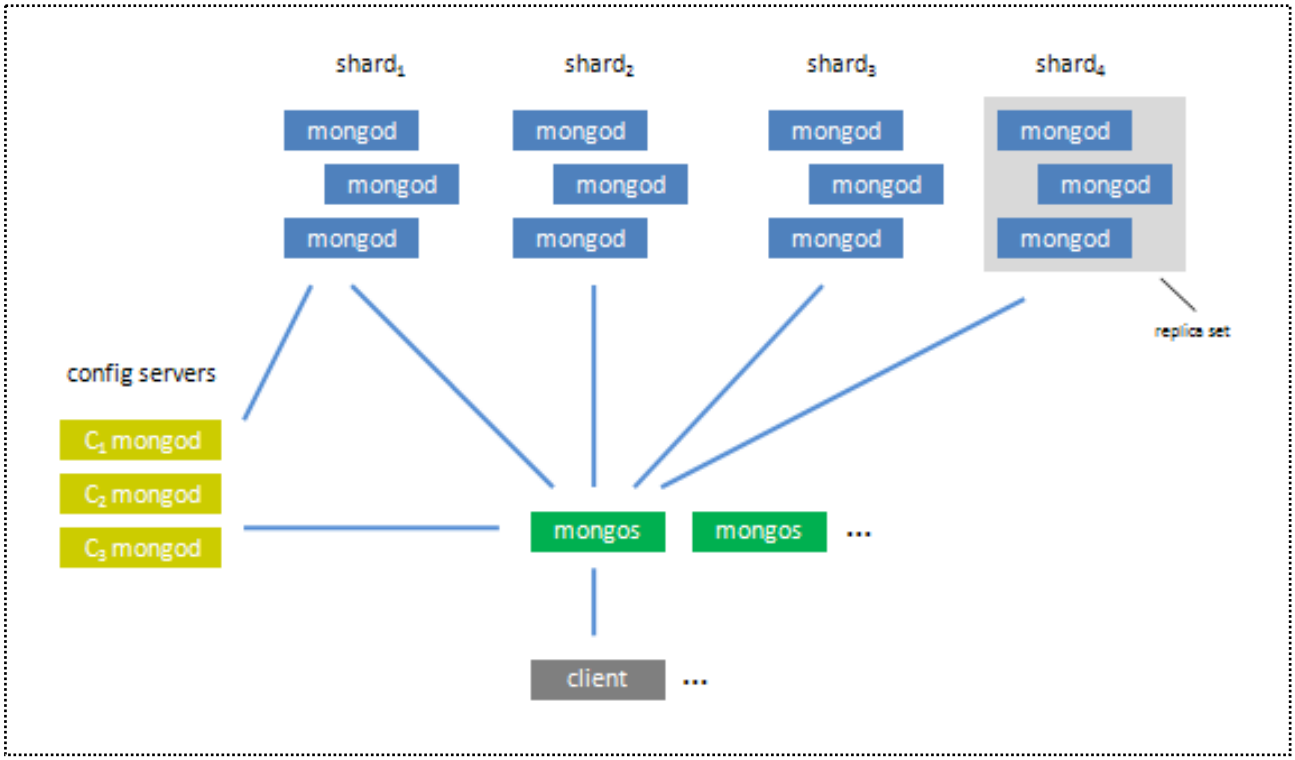
MongoDB ne pose aucune restriction quant aux documents contenus dans une même collection. Pourtant, pour des raisons pratiques, il est intéressant de distinguer des documents différents en les plaçant dans différentes collections (et donc de ne pas procéder comme nous l'avons fait dans l'exemple précédent).

À la différence d'une table SQL, le nombre de champs des documents d'une même collection peut varier d'un document à l'autre.

Bases de données :

Une base de données est un conteneur pour les collections (tout comme une base de données SQL contient des tables) avec ses propres permissions.

b. Vue d'ensemble de l'architecture MongoDB



Comme illustré par le schéma ci-dessus, MongoDB se décompose en trois services différents, mais dépendants les uns des autres.

Un *shard* se compose d'un ou plusieurs serveurs. Via son processus Mongod, il est en charge de stocker les données.

Les serveurs de configuration stockent les métadonnées du système, c'est-à-dire les informations basiques relatives à chaque *shard*, et des indications quant aux données stockées par ceux-ci. Ce service est indispensable. Il est donc nécessaire de l'assurer en lui consacrant deux, voire trois instances.

Le processus Mongos redirige les requêtes des applications clientes vers les *shards* adéquats et regroupe les résultats avant de les transmettre à l'application cliente.

MongoDB distribue les données en fonction de ses collections, c'est-à-dire que chaque collection peut définir sa propre politique de partitionnement de données. Un morceau (*chunk* en anglais) est un ensemble continu de documents d'une collection. Un morceau, défini via un triplet (collection A, minKey, maxKey), contient les données de la collection A, dont la clé est comprise entre les bornes minKey et maxKey.

Les serveurs de configuration ont connaissance de chaque morceau et de leur localisation (comme décrit ci-dessous).

collection	minkey	maxkey	location
users	{ name : 'Miller' }	{ name : 'Nessman' }	$shard_2$
users	{ name : 'Nessman' }	{ name : 'Ogden' }	$shard_4$
...			

La taille d'un morceau ne peut dépasser une taille prédéfinie. Lorsqu'il atteint cette limite, il est divisé en deux morceaux. Lorsque la limite de stockage d'un *shard* est atteinte, des morceaux lui appartenant émigrent vers un autre *shard*.

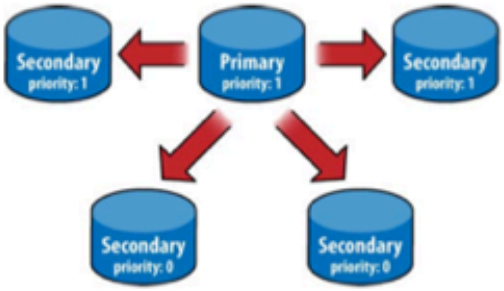
Actuellement, l'algorithme de répartition des données est très simple et est appliqué de manière automatique. Il déplace les morceaux en fonction de la taille des différents *shards*. Le but est de maintenir les données distribuées de manière uniforme, mais aussi de réduire au minimum les transferts de données. Pour qu'un déplacement ait lieu, il faut qu'un *shard* ait au minimum neuf morceaux de plus que le plus impopulaire des *shards*. À partir de ce point, les morceaux vont être répartis de manière uniforme sur les différents *shards*.

c. Réplications des données dans MongoDB

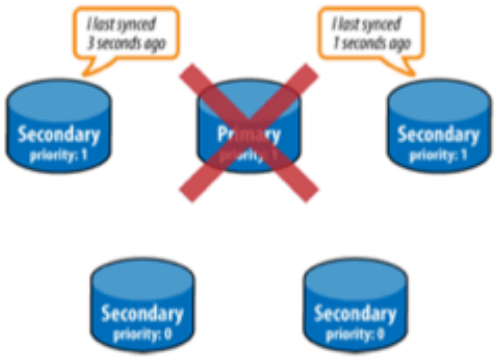
Le premier modèle de réplication proposé par MongoDB est le modèle maître / esclave. Depuis peu, MongoDB propose une nouvelle approche : la réplication par pairs (*replicat set* en anglais), qui voit tous les serveurs d'un *shard* comme des pairs qui éliront un maître temporaire. Dans les deux modèles, la réplication est asynchrone.

La réplication par pairs est une version améliorée du modèle maître / esclave. Elle offre un balancement automatique.

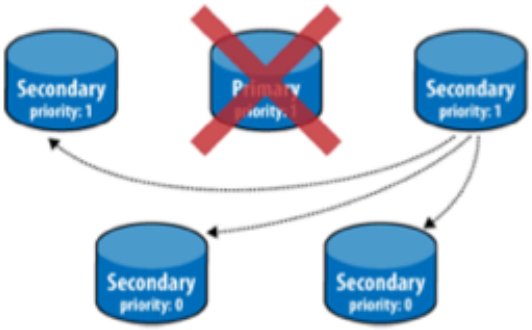
Comme l'indique son nom, tous les serveurs sont identiques et ont un rôle : primaire ou secondaire. Il n'y a qu'un serveur primaire par ensemble de répliques. En cas de défaillance de celui-ci, les membres de l'ensemble procèdent à l'élection d'un nouveau serveur primaire (en fonction de celui qui détient les informations les plus récentes). Les schémas ci-dessous nous montrent un scénario d'élection.



1) Un ensemble de réplicats



2) Défaillance du serveur primaire, procédure ⊞



3) Défaillance du serveur primaire, un nouveau serveur est élu.

Ce modèle de réplication donne à MongoDB la capacité de s'autogérer. Il ne nécessite pas d'intervention humaine pour continuer son bon fonctionnement en cas de défaillance d'une instance de stockage.

d. Modèle de requête

Grâce à son modèle de données en documents, MongoDB propose une interface de programmation assez riche (sans doute parmi les plus riches du mouvement NoSQL). À titre d'exemple, voici deux listes non exhaustives de ses fonctions :

- la première liste illustre les différents opérateurs pour filtrer une sélection :

  - les opérateurs <, <=, >, >= ;

  - l'opération *exists* : utilisée pour filtrer les entrées, qu'il ait ou non un certain attribut ;

  - l'opérateur *in* : qui vérifie qu'une valeur doit être présente dans l'entrée retournée ;

  - des expressions régulières qui peuvent être appliquées aux champs retournés.

- la seconde liste reprend quelques méthodes proposées par MongoDB :

  - *count :* qui retourne le nombre de documents correspondant aux critères de recherche ;

  - *limit :* qui limite le nombre de documents retournés ;

  - *sort* : qui trie les documents retournés ;

  - *skip* : qui permet de ne pas retourner les n premiers documents.

11.4.2. GridFS

GridFS est un système de fichiers distribué qui repose sur MongoDB. Il offre des mécanismes transparents qui divisent un fichier de grande taille en multiples documents. Il permet donc de stocker de manière efficace des objets de très grande taille, spécialement des fichiers - par exemple des vidéos -, et permet un grand nombre d'opérations (exemple : récupérer les N premiers *bytes* d'un fichier).

GridFS est donc une légère spécification pour stocker des fichiers et est construit au dessus de MongoDB. En effet les serveurs MongoDB ne font rien de spécial pour gérer les requêtes de GridFS. Tout le travail se fait au niveau des *drivers* et des outils qui sont du coté client.

L'idée basique, derrière GridFS, est de stocker les fichiers en les découpant en morceaux (*chunks*) et de stocker chaque morceau comme un document différent. En plus de cette action, on stocke également un document unique qui regroupe les morceaux ensemble et contient des métadonnées sur le fichier.

Pour GridFS, les *chunks* sont stockés dans une collection qui leur est réservée. Par défaut, les *chunks* vont utiliser la collection fs.chunks. Cependant, une autre collection peut être spécifiée si besoin. La collection fs.*chunks* rend la structure d'un document individuel simple :

```
{
  // object id of the chunk in the _chunks collection
  "_id" : <unspecified>,
  // _id of the corresponding files collection entry
  "files_id" : <unspecified>,
  // chunks are numbered in order, starting with 0
  "n" : chunk_number,
  // the chunk's payload as a BSON binary type
  "data" : data_binary,
}
```

Comme tout document dans MongoDB, un *chunk* a un '_id' unique. À cela s'ajoutent quelques autres champs :

- 'files_id' : est le '_id' du fichier qui contient les métadonnées du *chunk* ;

- 'n' : est le numéro du *chunk*. Ce numéro représente l'ordre, la position, du *chunk* dans le fichier d'origine ;

- 'data' : contient les données binaires composant ce morceau de fichier.

Les métadonnées de chaque fichier sont stockées dans une collection séparée, appelée par défaut fs.files. Chaque document dans la collection de fichiers représente un fichier unique dans le système de fichier GridFS et d'autres métadonnées peuvent lui être ajoutées si besoin.

```
{
  // unique ID for this file
  "_id" : <unspecified>,
  // size of the file in bytes
  "length" : data_number,
  // size of each of the chunks.  Default is 256k
  "chunkSize" : data_number,
  // date when object first stored
  "uploadDate" : data_date,
  // result of running the "filemd5" command on this file's chunks
  "md5" : data_string
}
```

En plus des clés / valeurs définies par l'utilisateur, GridFS gère un ensemble de métadonnées :

- '_id' : est l'id unique du fichier, représentant la valeur de la clés globale du fichier. Représente aussi la valeur « files_id » référencée pas les chunks dont il se compose ;

- 'length' : est le nombre total de *bytes* qui constituent le contenu du fichier ;

- 'chunkSize' : est la taille de chaque *chunk* composant le fichier. La taille par défaut est 256K, mais elle peut être ajustée si besoin ;

- 'uploadDate' : est la date à laquelle le fichier à été stocké dans GridFS ;

- 'md5' : est la somme de contrôle md5 générée du côté serveur.

Comprendre la spécification de GridFS rend triviale l'implémentation de fonctionnalités non fournies par les *drivers* côté client.

11.4.3. Conclusion sur GridFS

GridFS, de MongoDB, est une très bonne alternative à GlusterFS. Il supporte parfaitement l'IPv6 et offre de bons mécanismes de scalabilité et de tolérance aux fautes.

GridFS s'exécute sans avoir besoin de privilèges *root*. Son intégration dans SlapOS a pu être faite sans problème. Son utilisation nous permet de profiter de l'interface de programmation offerte par MongoDB pour adapter ce système de fichiers à nos besoins spécifiques.

12. Indexation répartie

Comme la LogBox utilise déjà Apache Solr pour l'indexation, les études des solutions d'indexation en répartie, pour le collecteur de logs pour le *Cloud*, se sont faites autour de ce dernier.

12.1. Apache Solr

Les bases de données relationnelles (de type MySQL, PostgreSQL, Microsoft SQL Server ou autre) permettent de conserver des données. Ces données sont enregistrées de manière très structurée, mais elles ne sont accessibles que par le langage SQL.

L'avantage des bases de données relationnelles est que l'on peut aisément retrouver un enregistrement lié à partir d'un autre. En effet, retrouver une donnée élémentaire (un champ) à partir d'une autre donnée élémentaire (un identifiant) est une tâche que SQL accomplit sans problème. Il convient cependant de rappeler l'une des grandes faiblesses de SQL : sa pauvreté en matière de permutations de texte (instructions LIKE et REGEXP). Que se passe-t-il lorsqu'un utilisateur fait une faute de frappe dans un moteur de recherche à base de SQL ? Lorsqu'il recherche un nom propre, a-t-il correctement placé chaque accent, trait d'union ou apostrophe ? Est-ce que toutes les permutations possibles du texte recherché sont bien prisent en compte ? Sans doute n'obtient-il pas de résultats ou, au moins, pas les résultats attendus.

Apache Lucene est un moteur d'indexation de texte, permettant d'effectuer des recherches en langage naturel, à l'aide de diverses manipulations automatiques du texte. Le texte indexé est enregistré sous de multiples représentations - de même pour le texte recherché -, et les résultats de recherche sont déterminés suite à la comparaison de ces variantes.

Apache Solr étend le principe de Lucene en facilitant l'administration (interface RESTful) et en ajoutant des fonctionnalités comme des filtres de recherche ou la manipulation des résultats.

Un noyau (*core* en anglais) Solr est décrit par un schéma (schema.xml) et un fichier de configuration (solrconfig.xml). Le fichier schema.xml définit les types et les champs de données, tandis que le fichier solrconfig.xml définit les manières d'utiliser ce schéma (c'est-à-dire l'API du noyau). Nous détaillerons plus loin ces deux fichiers.

Le schéma définit obligatoirement un champ qui sert d'identifiant unique aux documents : *uniqueKey*.

Chaque schéma est prévu pour enregistrer un ensemble de documents. Chaque document est un ensemble de champs à valeur unique ou à valeurs multiples. À noter qu'un index Solr (c'est-à-dire un couple schema.xml + solrconfig.xml) est orienté vers un unique format de document. Par exemple, si on définit un schéma pour le document « produit », il ne convient pas d'y enregistrer également des documents « fournisseur ». Pour cela, il faudra prévoir un deuxième index.

Le fichier schema.xml contient donc toutes les informations relatives aux champs des documents, et la façon dont ils doivent être traités lorsque les documents sont indexés ou lorsque ces champs sont interrogés. Le schéma est structuré de la façon suivante :

- les types des données :

- la section <types> permet de définir une liste de déclaration de <fieldtype> ;
- possibilité de définir des types de champs « poly », permettant de créer des champs multiples ;

- les champs :

- la section <fields> contient la liste de déclaration des <fields>, composés, entre autre, des attributs *name, type, indexed= true / false* (*true* si le champ doit être classable), *stored = true / false* (*true* si la valeur du champ doit être récupérable dans la recherche) ;

- les champs dynamiques, une des fonctionnalités les plus intéressantes, basés sur des préfixes ou des suffixes :
  <dynamicField name="*_i" type="integer" indexed="true" stored="true"/> ;

- l'indexation des mêmes données dans des champs différents, pour les indexer de manières différentes, par exemple pour la recherche, le tri et le « facettage » ;

- les configurations diverses :

- le champ <uniqueKey> permet d'indiquer à Solr qu'un champ doit être unique pour tous les documents. Si un nouveau document indexé contient la même valeur pour le champ qu'un document existant, l'ancien document est supprimé de l'index ;

- il est possible de déclarer des <copyField> afin de dupliquer les données d'un champ source dans un champ destination.

Modes de Recherche :

Apache Solr met à disposition de ses utilisateurs un certain nombre de modes de recherches et de guidage :

- recherche par mots clés :

- support des opérateurs booléens (ET, OU) ;
- recherche *full-text*, qui consiste à opérer des traitements (l'indexation) sur du texte afin d'y retrouver plus facilement des mots. Ce procédé revient à effectuer une suite de manipulations sur chacun des mots du texte afin d'augmenter la pertinence des recherches ;

- aides à la recherche :

- suggestions d'orthographe ;
- suggestions « *More Like This* » pour un document donné ;
- fonction d'auto-complétion ;

- outils de guidage :

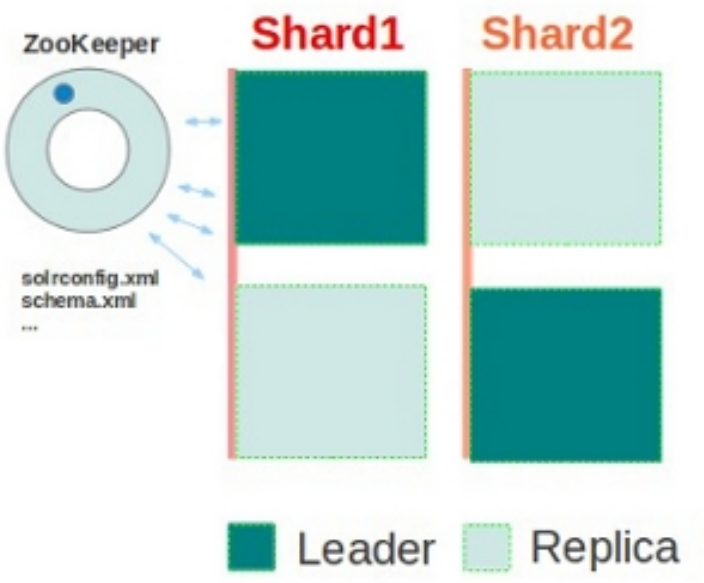
- possibilité de trier la recherche sur autant de champs que souhaités ;
- recherche à facette, qui permet d'affiner les résultats en ajoutant des critères supplémentaires parmi les résultats retournés, en indiquant le nombre de résultats correspondant à chaque critère ;
- possibilité de configurer de meilleurs résultats pour une requête, se substituant au résultat normal selon le score et le tri.

12.2. SolrCloud

Quand les index deviennent trop grands pour être stockés dans un seul système, ou quand une requête devient trop longue à exécuter, la version de base de Solr offre la possibilité qu'un index puisse être divisé en plusieurs *shards,* qui sont des instances Solr, qui gèrent des données de l'index, et permettent donc d'exécuter des requêtes à travers ces *shards* et de fusionner les résultats. Mais le tout exige beaucoup de configuration et la mise en place de règles définies manuellement.

Le projet SolrCloud vise à répondre à ce problème en offrant des fonctions pour le découpage automatique de l'index en plusieurs *shards* qui se chargent d'une partie de

l'index. La gestion de la grappe de serveurs, et de sa configuration, n'est toutefois pas incorporée dans le produit, mais repose plutôt sur le produit Apache ZooKeeper [6].



Comme dans MongoDB, SolrCloud divise l'index en un ensemble de morceaux, et chaque morceau est géré par un *shard* qui se compose d'un ensemble de serveurs Solr. Chaque *shard* dispose d'un *leader* et de plusieurs serveurs de réplication. L'élection d'un *leader* au sein d'un *shard* se fait grâce au coordinateur de systèmes distribués, Apache Zookeeper. Dans le cas où le serveur *leader* n'est plus présent dans le système, un autre serveur parmi les serveurs de réplication est élu à sa place comme *leader*.

Lors de l'exécution des requêtes de recherche, l'ensemble des serveurs de réplication d'un *shard* peuvent participer au traitement de la dite requête pour permettre des traitements distribués et fournir des résultats plus rapidement.

Ainsi SolrCloud offre un ensemble de fonctionnalités adaptées au *Cloud* :

- Tolérance aux fautes
- Haute disponibilité
- Traitement distribué des requêtes
- Stockage distribué des index

Apache SolrCloud est écrit en java, donc comme son exécution ne dépend que de la disponibilité d'une JVM (Java Virtual Machin), son intégration dans SlapOS était possible. De plus SolrCloud supporte bien l'IPV6.

13. Réalisation du collecteur de logs pour le *Cloud*

Après la définition de l'architecture du collecteur de logs pour le *Cloud* et l'étude des différentes solutions disponibles, la réalisation s'est faite en deux étapes :

- la première est celle du développement du collecteur de logs proprement dit ;

- la seconde est celle de l'écriture de la recette et du profil d'installation dans SlapOS.

13.1. Développement du collecteur de logs pour le *Cloud*

Le collecteur de logs a été réalisé suivant les principes que nous avons décrits précédemment.

Tout le processus de développement a été réalisé sur des systèmes d'exploitation de type Unix, l'environnement de développement étant constitué d'outils assez classiques, tels que le terminal Linux, muni de l'extension pour l'éditeur Emacs et l'IDE Eclipse.

Comme la LogBox a été développée en utilisant le langage Python, le collecteur de logs pour le *Cloud* que j'ai développé au cours du stage est, lui aussi, basé sur le langage Python. Le besoin de communication entres les différents composants de la solution incite à choisir un *framework* réseaux adapté dés le départ. L'expérience et les conseils de l'équipe WLB (Wallix LogBox) m'ont guidé vers l'utilisation du *framework* Twisted, qui est au cœur de la LogBox actuelle.

13.1.1. Le *framework* Twisted

Twisted est un *framework* réseaux très complet, écrit en Python, et sous licence MIT. Il permet de construire à la fois des clients et des serveurs utilisant des protocoles très variés : HTTP, SMTP, IMAP, POP, SSH, DNS, FTP, IRC...
Twisted se base sur un paradigme événementiel. Il est donc asynchrone et piloté par les événements, ce qui signifie que les utilisateurs écrivent de courtes fonctions de rappel (*callbacks*) qui sont appelées par le *framework* lorsque des événements surviennent (comme l'arrivée d'un paquet, par exemple).

Le support d'un très grand nombre de protocoles est un atout pour Twisted. Non seulement il supporte plus de protocoles que la bibliothèque standard de Python, mais, en plus, ces protocoles sont le plus souvent disponibles en tant que client et serveur. Enfin, il s'agit le plus souvent d'implémentations très complètes. Par exemple, le serveur SSH supporte SFTP, et le client SSH est capable d'utiliser un agent. C'est un point très important pour les développeurs de Twisted : il s'agit de fournir des implémentations pouvant être utilisées en production.

L'aspect asynchrone de Twisted peut être déroutant. Quand une action est bloquante, on fournit une fonction qui sera exécutée quand cette action sera terminée. Ainsi, quand on veut lire un paquet réseau, on fournit une fonction qui sera appelée avec le paquet reçu en

paramètre. Une fois le paquet reçu, le moteur de Twisted appellera lui-même cette fonction. En attendant, il peut gérer d'autres aspects, comme accepter une nouvelle connexion. Les avantages d'une telle approche sont multiples :

• il n'y a pas à gérer les problèmes de communication interprocessus ou les problèmes de concurrence dans les *threads* ;

• le passage à l'échelle est plus prévisible : il n'y a pas de problème, par exemple, à gérer 200 000 connexions simultanées (en dehors de la consommation mémoire et processeur), alors que ce serait problématique si chaque connexion devait être gérée par un processus ;

• les performances peuvent être meilleures. En effet, une portion de code ne rend la main que lorsqu'elle effectue une action bloquante (souvent liée au réseau). Le reste du temps, elle utilise 100% du processeur.

Utiliser Twisted apporte aussi quelques inconvénients :

• la programmation asynchrone peut être difficile à appréhender. Elle se traduit dans Twisted par la nécessité de définir de nombreuses fonctions pour réagir aux évènements. Le code ne se lit donc pas d'un seul bloc ;

• il n'y a qu'un seul processus, qu'un seul *thread*. On n'exploite donc pas les machines avec plusieurs cœurs. Ceux-ci sont toutefois disponibles pour les autres processus (la base de données par exemple). Twisted offre cependant la possibilité de gérer plusieurs processus ou plusieurs *threads*, mais, dans ce cas, on se prive de certains avantages de la programmation asynchrone ;

• la programmation asynchrone rend difficile la compréhension des *tracebacks*, ce qui peut rendre assez pénible la mise au point des applications : quand une exception survient, si elle apparaît lors du traitement d'un appel différé, le contexte correspondant à cet appel différé n'est plus disponible.

13.2. Mise en place du modèle producteur / consommateur

Pour des raisons de planification du stage, le développement d'un coordinateur spécifique n'a pas été retenu car ce type de développement est très chronophage et risquait de ne pas pouvoir être mis en place dans le temps imparti.

L'écriture et la mise en œuvre d'applications distribuées peuvent s'avérer très délicates car soumises au réseau. En effet, le réseau est très souvent instable et il en résulte des phénomènes indésirables comme, par exemple, un message n'arrivant pas à son destinataire. Ainsi, les applications distribuées doivent prendre en compte tous ces phénomènes, plus ou moins aléatoires, ce qui rend leur programmation complexe et sujette à beaucoup d'erreurs. De plus, dans la majorité des applications réparties, on retrouve souvent les mêmes besoins - comme la synchronisation entre processus ou le consensus -, et ces applications nécessitent certaines techniques usuelles pour leur fonctionnement :

- techniques de localisation (annuaires, découverte, etc.) ;

- horloges globales ;
- identifiants uniques ;
- espace global de stockage (disque, mémoire).

C'est à partir de ce constat qu'ont été conçus les outils de coordination, qui facilitent grandement l'écriture de programmes distribués et implémentent plusieurs primitives de base permettant de construire des protocoles plus complexes, de manière sûr et robuste.

J'ai donc opté pour l'utilisation d'un outil de coordination de systèmes distribués afin de réaliser une file fiable pour un modèle producteur / consommateur.

13.3. ZooKeeper

ZooKeeper [6] est un service de coordination des traitements d'applications distribuées. Il fournit un noyau simple et hautement performant pour construire des primitives de coordination plus complexes au niveau du client. Il intègre plusieurs éléments comme un groupe *messaging*, registre partagé, ou encore un service de *lock* distribué.

ZooKeeper se présente comme un composant logiciel pour de la « coordination de services de haute fiabilité ». Concrètement, ZooKeeper héberge un modèle de données conceptuellement simple, qui facilite grandement l'implémentation de mécanismes divers, comme la présence, l'élection ou la constitution de groupes dynamiques.

Un service ZooKeeper est un programme Java qui fonctionne sur, au moins, trois instances. Les instances communiquent entre elles via le protocole TCP/IP. À un moment donné, le service ZooKeeper a un unique *leader*, chargé de coordonner les réplications. Chaque instance de ZooKeeper conserve en mémoire vive une image des nœuds et répond aux requêtes des clients.

Lorsque l'instance maître tombe en panne, une autre instance prend le relais. Comme toutes les réplications s'effectuent de façon synchrone, il n'y a pas de perte de données.

ZooKeeper utilise en interne une version légèrement modifiée de Paxos. Paxos est une famille d'algorithmes permettant d'obtenir un consensus parmi différents participants dont la fiabilité n'est pas garantie.

13.3.1. ZooKeeper et la partition réseau

La partition de réseau, également appelée *split brain*, est un sérieux problème pour les systèmes à tolérance de fautes. Quand on a un serveur primaire et un serveur secondaire, ils s'envoient en permanence un signal (*heartbeat*) pour se dire qu'ils sont tous les deux en fonctionnement. Lorsque le serveur primaire tombe en panne, le serveur secondaire ne reçoit plus rien et sait alors qu'il doit devenir serveur primaire. Mais, que se passe-t-il si la communication entre les deux serveurs est rompue, alors qu'ils fonctionnent ? On se retrouve avec deux serveurs primaires. Selon la topologie réseau, on peut avoir

des clients effectuant des opérations sur les deux serveurs primaires simultanément. La réconciliation devient problématique.

Pour traiter le cas de la partition de réseau, parmi un total de n participants, ZooKeeper élit un leader au sein d'un *quorum* qui doit compter au moins (n + 1) / 2 participants. Tenant le nombre de participants pour fixé, il ne peut pas y avoir, à un moment donné, plus d'un seul *quorum* satisfaisant cette condition, donc la partition est impossible. Avec trois serveurs, ZooKeeper survit à la mort ou à la déconnexion de l'un d'entre eux. Avec sept serveurs, on peut en perdre trois. Évidemment, les chances de perdre un serveur augmentent avec le nombre de serveurs.

L'avantage : la simplicité de la formule.

L'inconvénient : on peut se retrouver bloqué même lorsqu'il reste presque la moitié des serveurs en état de fonctionner.

ZooKeeper est donc la structure de base sur laquelle nous allons nous appuyer pour gérer une file fiable pour un modèle producteur / consommateur.

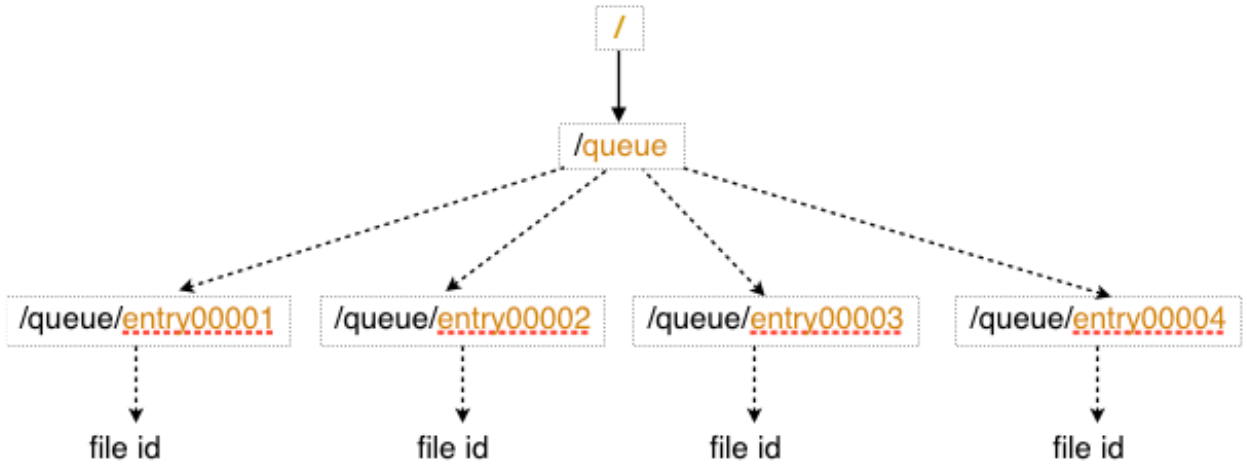
13.4. Une file fiable avec ZooKeeper

ZooKeeper offre une API pour faciliter la réalisation d'applications distribuées. Cette API est utilisée, par exemple, pour gérer des *locks* entre des processus client.

Puisque l'API ZooKeeper n'est pas parfaitement disponible en langage Python, j'ai utilisé le *binding* Python txZookeeper. Un *binding* permet d'utiliser une bibliothèque logicielle dans un autre langage de programmation que celui avec lequel elle a été écrite.

TxZookeeper est une librairie pour ZooKeeper. Elle a l'avantage d'être en Python et d'offrir quelques primitives de programmation distribuée de bases et, surtout, elle met à notre disposition une file fiable. Cette file assure que la consommation de tout élément soit acquittée par le consommateur, ce qui fait que, si un consommateur meurt avant d'avoir acquitté la consommation d'un élément, ce dernier est remis dans la file à disposition des autres consommateurs. L'acquittement se décline par la suppression de l'élément dans la file après consommation. Une file fiable peut être persistante ou transitive. Pour des raisons de fiabilité, il a été choisi d'utiliser une file persistante.

ZooKeeper s'appuie sur une abstraction d'un arbre qui ressemble à un système de fichier où chaque nœud, nommé znode, est un fichier sur lequel on peut effectuer un nombre d'opérations. Chaque znode est associé à une donnée et/ou un autre znode. Il n'y a pas besoin d'utiliser des protocoles complexes pour s'assurer qu'il n y ait qu'un seul processus à la fois qui puisse accéder à un znode. L'arbre représente aussi un espace de nom hiérarchique, de façon à permettre à différentes applications distribuées d'utiliser le même serveur ZooKeeper, sans se soucier du fait qu'il y ait deux fichiers (znode) avec le même nom.



Une file est une structure de données où les producteurs rajoutent des éléments pour que des consommateurs puissent les récupérer. Cette structure repose sur deux opérations basiques : « enqueue », pour rajouter l'élément à la file, et « dequeue », pour récupérer un élément de la file. La file fournie par txZookeeper est structurée d'une façon très simple dans ZooKeeper. Tous ces éléments sont stockés en tant que znodes, qui sont des fils d'un znode représentant l'instance d'une file.

La figure ci-dessus est une représentation d'une instance d'une file 'queue' dans Zookeeper. On peut voir que cette file contient quatre éléments à consommer (entry00001 à entry00004), et que chaque élément contient une donnée qui, dans ce cas, est l'id d'un fichier à traiter.

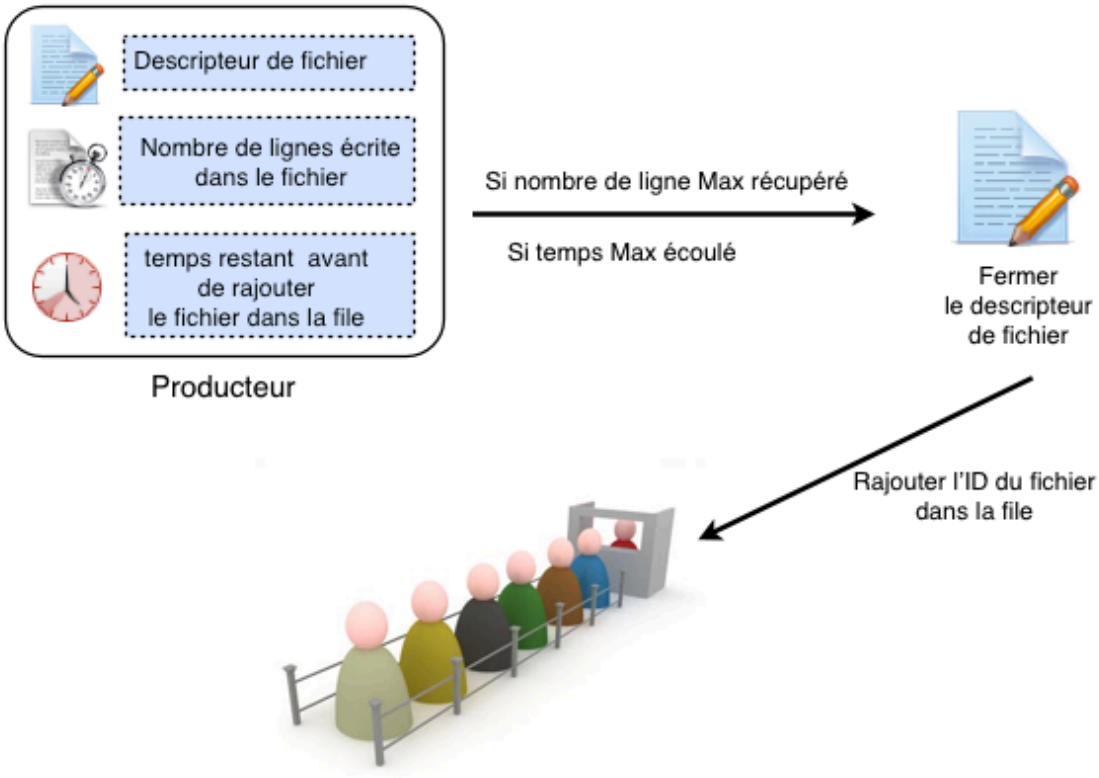
Après avoir décrit la structure de la file, nous allons, dans ce qui suit, approfondir les rôles du producteur et du consommateur, les traitements qu'ils effectuent et leurs implémentations.

13.5. Producteur

13.5.1 Fonctionnement

Le rôle du producteur, dans le contexte du projet, est de pouvoir récolter des lignes de logs qui, par la suite, vont être stockées sous forme de fichiers plats et rajoutées dans la file pour être traitées par des consommateurs.

Chaque instance de producteur embarque un module d'entrée pour recevoir des lignes de logs. La nature et le fonctionnement de ce module seront détaillés par la suite.



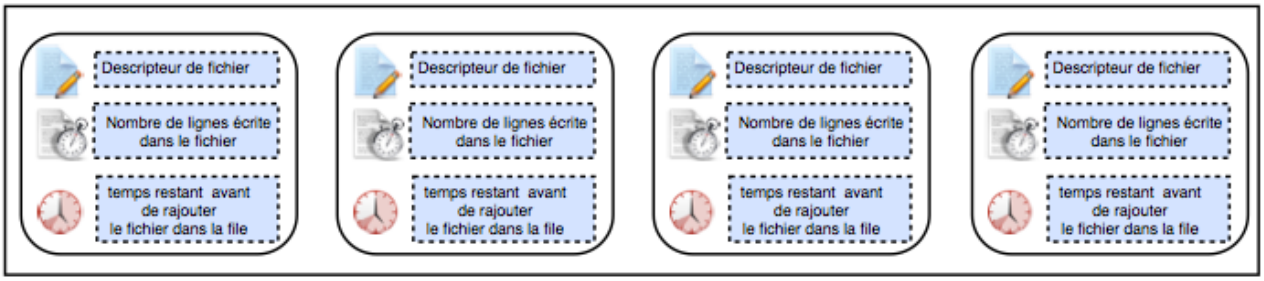
Le producteur reçoit donc un flux de données composé de lignes de logs. À la réception d'une ligne, un descripteur de fichier est créé dans le système de fichier utilisé (GridFS). Ce fichier est rempli avec les lignes reçues jusqu'à atteindre un nombre défini (nombre de lignes MAX). Une fois cette limite atteinte, le descripteur de fichier est fermé et son «_id» est rajoutée dans la file afin qu'il puisse être récupéré et traité par un consommateur. Le descripteur de fichier a une durée de vie définie par un *timer* (Temps MAX). Une fois le *timer* expiré, le fichier est déclaré dans la file d'attente. Ce mécanisme permet d'éviter d'attendre que le nombre de lignes maximum soit atteint, alors que la source de collecte a arrêté d'envoyer des données pendant un moment. Cela rend le fichier disponible à la consommation dans de meilleurs délais.

Avant de fermer un fichier et de le rajouter dans la file, certaines informations lui sont rajoutées en tant que métadonnées :

- *Source :* la source de collecte ;
- *Lines :* le nombre de lignes de logs qui sont stockées dans le fichier à la fin de la collecte ;
- *RemLines :* initialisé avec la même valeur que *Lines*, il correspond au nombre de lignes qui restent dans le fichier ;
- *Position :* initialisé à 0, il indique la dernière position atteinte dans le fichier après une lecture.

Le producteur est capable de traiter simultanément plusieurs sources de logs. Pour ce faire, il gère une structure de données pour chacune d'entre elles. Comme on peut le voir sur le schéma qui suit, chaque structure est composée de :

- un descripteur du fichier courant de la source X ;
- un nombre de lignes de logs écrites dans le fichier de la source X ;
- un *timer* gérant la durée de vie du descripteur de fichier courant de la source X.



Producteur

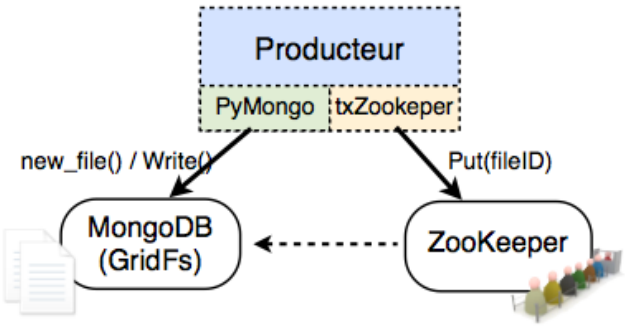
13.5.2 Implémentation

Au lancement du producteur, une file fiable est créée dans ZooKeeper s'il n'existe pas encore. La file est une instance de la classe ReliableQueue fournie par le module txZookeeper. Une fois cette instance créée, on peut utiliser la méthode « *put* » pour rajouter des éléments à la file.

Comme il a été spécifié précédemment, le système de fichiers utilisé est GridFS de MongoDB. Pour communiquer avec ce système de fichiers, MongoDB met à disposition des développeurs Python l'API PyMongo, qui permet d'interagir avec les fonctionnalités de MongoDB et celles du système de fichiers GridFS.
Parmi les primitives qui sont fournies par cette API, on retrouve :

- new_file(**kwargs) : qui permet de créer un nouveau fichier dans GridFS, et renvoie une instance de GridIn, qui fournit un ensemble de fonctions pour interagir avec le fichier ;
- write(data) : cette méthode est fournie par la classe GridIn. Elle permet d'écrire des données dans le fichier, elle peut prendre en paramètre une chaîne de caractères ou un objet, de type fichier, qui implémente une méthode de lecture *read*()

Stocker des fichiers dans GridFS se fait de manière transparente, avec des primitives proches d'un simple système de fichiers normal. Chaque fichier est identifié par un id unique. C'est ce dernier qui permet de récupérer un fichier spécifique, en utilisant la méthode get(id) de l'API. Les éléments qui composent la file contiennent les id de fichiers qui permettront au consommateur de les récupérer par la suite.



Le schéma ci-dessus représente les relations entre les modules décrits plus haut qui composent le producteur.

Pour profiter au maximum de ce système de stockage, il était nécessaire de le coupler avec les niveaux de requête NoSQL de MongoDB. En effet, en passant par des requêtes NoSQL nous avons pu rajouter aux fichiers les métadonnées dont nous avions besoin.

13.5.3. Module d'entrée sécurisé

Pour permettre la collecte de logs, chaque instance de producteur embarque un module d'entrée permettant de recevoir des lignes de logs. La réception de données par le collecteur se fait via la méthode POST du protocole https.

Le choix du protocole https est justifié par deux raisons :

- c'est un protocole standard qui se veut simple et facile à intégrer avec les agents qui vont se charger de l'envoi de données de logs ;
- c'est un protocole qui offre un certain niveau de sécurité permettant de crypter l'envoie de données.

La mise en place du point d'entrée s'est faite en profitant de la puissance de l'API Twisted, qui facilite grandement la création d'un serveur https. Il suffit d'implémenter une classe qui définit une méthode render_POST pour traiter les requêtes POST. Cette API permet de récupérer un certain nombre d'informations sur une requête :

- la source de la requête ;
- l'URL de la ressource demandée ;
- le contenu de la requête.

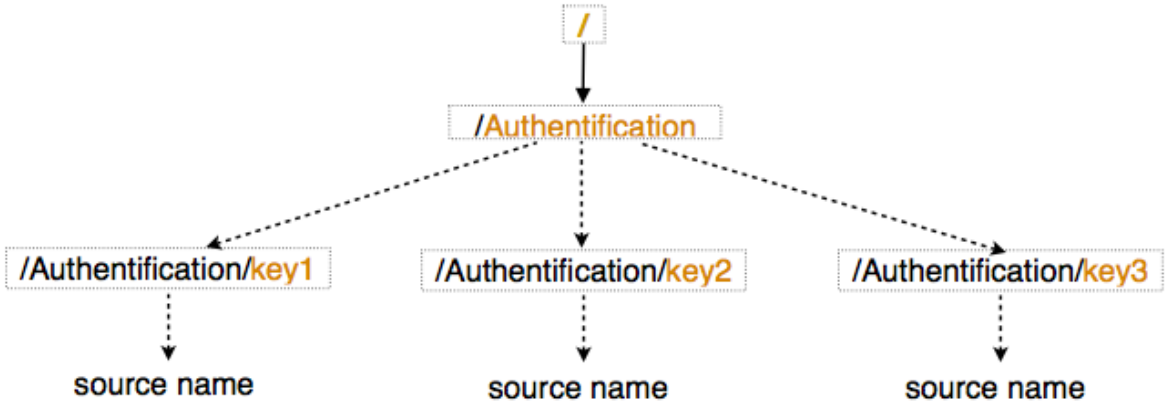
Pour pouvoir envoyer des lignes de logs vers un producteur, la source de collecte doit au préalable être déclarée dans le système de collecte. En effet, pour des raisons de sécurité, on ne peut se permettre d'accepter de recevoir des données de la part de

n'importe quelle entité non connue. Un mécanisme de déclaration de sources de collecte a donc été mis en place.

Une source de collecte demande une clé au système via une ligne de commandes. Cette clé est générée selon les étapes suivantes :

1) l'utilisateur fournit un nom permettant d'identifier la source ;
2) le système génère un certificat aléatoire ;
3) le système calcule une clé de *hash* sur ce certificat ;
4) le système stocke la clé associée au nom ;
5) le système renvoie la clé calculée à l'utilisateur.

Les clés générées doivent être stockées dans un point accessible à toutes les instances de producteurs. Pour ce faire, il a été choisi de profiter des fonctionnalités de ZooKeeper pour maintenir un arbre de stockage de clés.



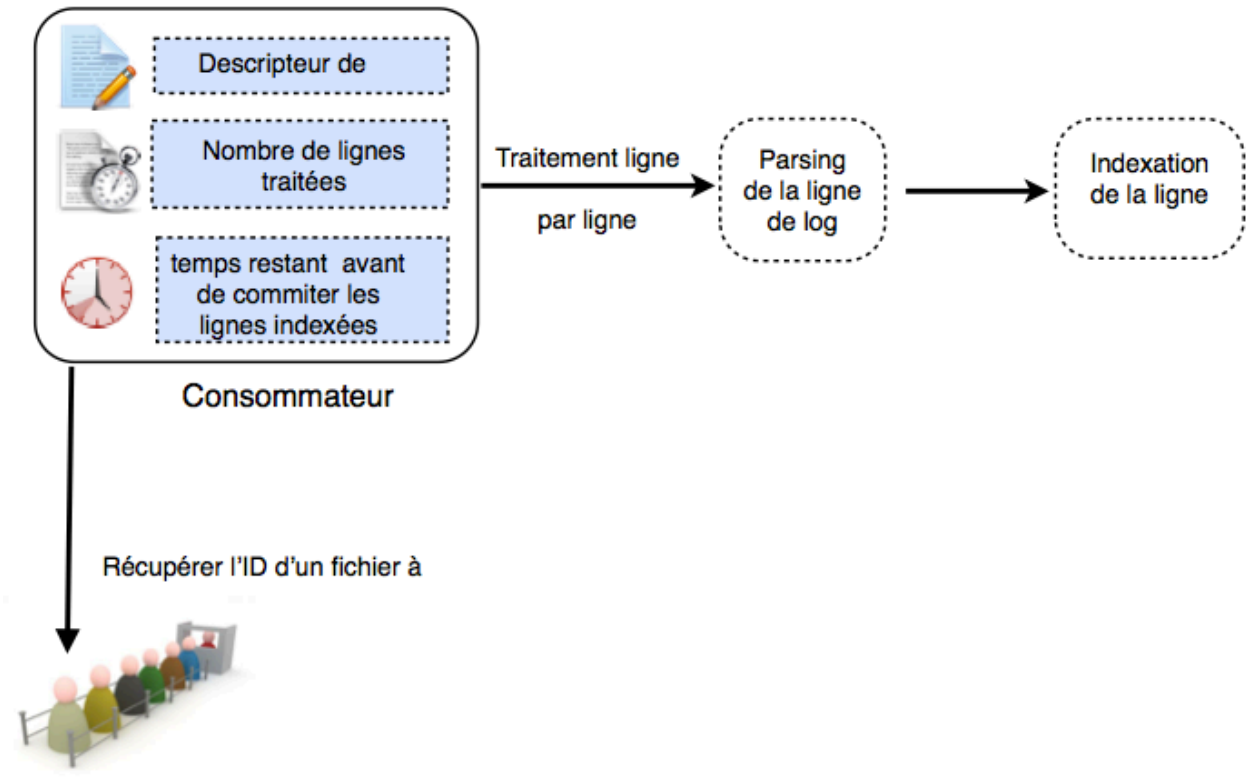
Comme on peut le voir ci-dessus, ZoKeeper maintient un znode *Authentification,* qui lui-même a des fils znodes dont le nom est la clé et le contenu est le nom de la source.

Avec ce mécanisme d'authentification mis en place, le client source doit annoncer sa clé au producteur à chaque envoi de logs. Si, par exemple, la clé de la source «X» est la suivante :
51b74d430b433ac85615cf6747e4a56fdbe2994834843854efc4be3c5468345f
Ce client rajoute cette clé au *path* de l'url du producteur :
https://addr/51b74d430b433ac85615cf6747e4a56fdbe2994834843854efc4be3c5468345f
Le producteur, avant de traiter les données reçues, doit vérifier si la clé est déclarée dans le système en essayant de la récupérer dans ZooKeeper.

En plus de gérer l'authentification, le producteur se doit d'informer la source qui envoie les logs de l'état de traitement des requêtes. En effet, pour chaque ligne de logs reçue, il doit répondre avec les codes suivants :

- code 200 : si le traitement de la ligne de logs s'est passé avec succès ;
- code 500 : si le producteur n'a pas réussi a traiter la ligne de logs ;
- code 503 : si un des composant (GridFS, MongoDB, ZooKeeper) du producteur n'est pas opérationnel, annoncer que le service n'est pas disponible.

13.6. Consommateur



Après avoir récupéré l'id d'un fichier à traiter, le consommateur initialise une structure pour gérer son traitement. La structure se compose des éléments suivants :

- un *timer* : pour gérer un délai entre les *commits* des logs dans SolrCloud ;
- un descripteur de fichiers : pour lire le fichier récupéré ;
- un compteur : pour compter le nombre de lignes traitées.

Une fois la structure initialisée, le consommateur commence par récupérer la métadonnée position, qui correspond à la position à partir de laquelle il faut commencer la lecture du fichier. Une fois la position de lecture changée en utilisant la méthode lseek() fournie par l'API pyMongo, le consommateur commence à lire le fichier ligne par ligne. Chaque ligne lue doit être *parsée* et indexée dans SolrCloud. Après le traitement de chaque ligne, la métadonnée position du fichier est mise à jour.

13.6.1. *Parsing* des logs avec le PyLogsParser

Les logs existent sous des formats variés. Pour *parser* un grand nombre de différents types de logs, un développeur a besoin d'écrire un code basé sur une large liste d'expressions régulières. Ce qui peut rapidement devenir illisible et difficile à maintenir.

PyLogsParser est une librairie *open source* en Python, créée par Wallix. Elle est utilisée au cœur du mécanisme de *tag* et de normalisation de logs dans la LogBox.

En utilisant le LogsParser, un développeur peut se libérer de la contrainte d'écrire un moteur de *parsing* de logs.

Le PylogsParser utilise des fichiers de définitions de normalisation, dans le but de *taguer* les entrées de logs. Les fichiers de définitions sont au format XML. Ces fichiers sont constitués d'un ensemble de *paterns,* définis par des expressions régulières. Chaque *patern* est associé à un mot clé qui définit un nom de *tag*.

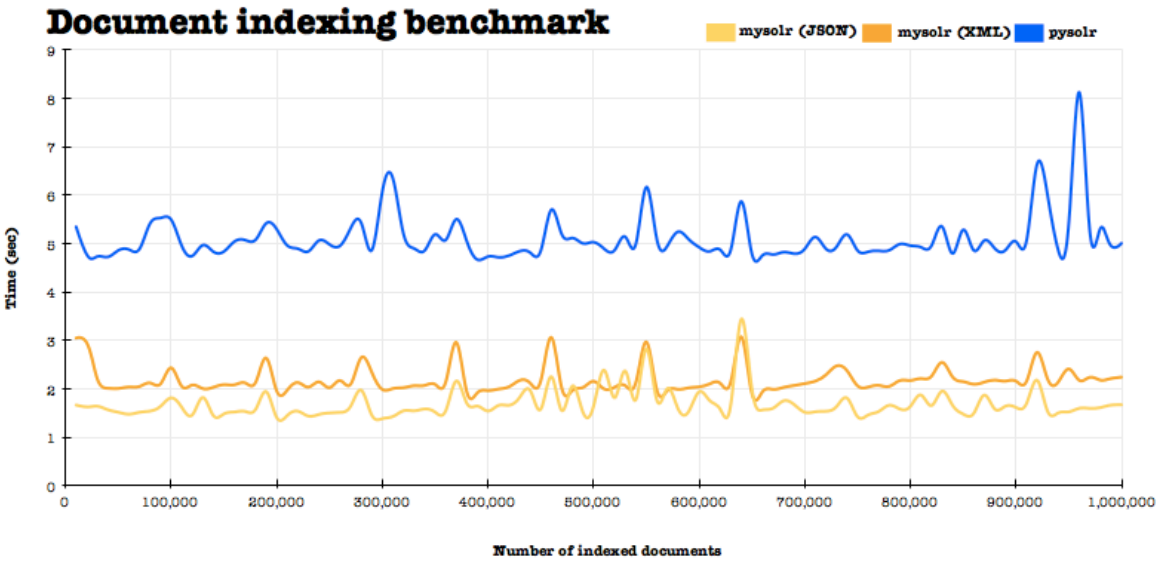
Après le processus de *parsing,* un dictionnaire Python est récupéré, dont les clés correspondent aux *tags* définis par le format de log traité.

13.6.2. Indexation des logs

Une fois le *parsing* de la ligne de log terminé, le dictionnaire récupéré est mis en forme pour qu'il puisse respecter le format schéma défini dans SolrCloud.

Le processus d'indexation est très important dans le cadre de ce projet. Il peut être coûteux si les bons choix ne sont pas faits lors de l'implémentation. En effet, choisir un module Python performant, et un format de données optimal, est primordial.

Le but du module Mysolr est d'offrir le client Python le plus performant pour communiquer avec Solr. En comparaison avec d'autres modules, comme Pysolr, il se veut rapide et est facile à utiliser.



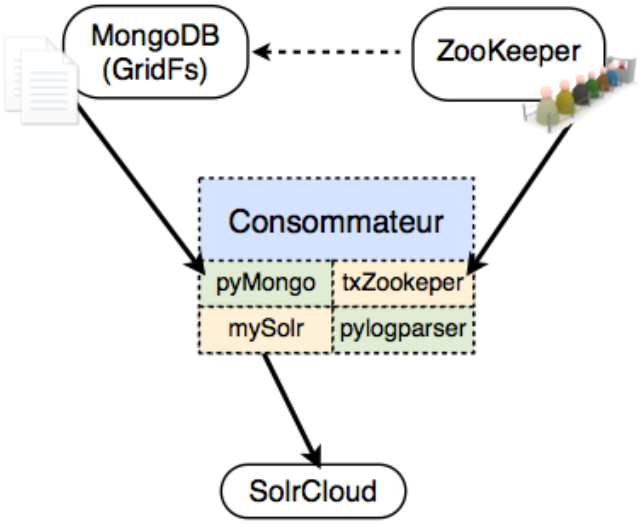
Le graphe ci-dessus montre une comparaison entre les performances de MySolr et celles de PySolr lors de l'indexation de documents. On peut remarquer l'intérêt d'utiliser le format JSON, qui est un format léger d'échange de données et qui est moins verbeux que le format XML.

Le choix du module MySolr découle de ses performances. Mais, ce dernier ne supporte ni l'IPv6 ni la nouvelle version d'Apache Solr (SolrCloud). Pour autant ce n'est pas un obstacle, puisque j'ai pu le modifier et lui appliquer les *patches* nécessaires à son bon fonctionnement.

Lors de l'indexation de documents dans Solr, aucun changement effectué n'est apparent jusqu'à l'exécution d'un *commit*. L'intervalle d'exécution de ce dernier dépend de la vitesse à laquelle on veut voir apparaître les données indexées dans le moteur de recherche de Solr. Cependant, le *commit* est une opération lourde, qui peut prendre un certain temps pour finir son exécution. Pour obtenir de bonnes performances on ne peut pas se permettre d'exécuter un *commit* à chaque indexation d'une ligne de logs.
Le choix qui a été fait est de déclencher les *commits* suivant deux politiques :

- un *timer* fixé se charge de déclencher l'opération de *commit* dans un intervalle de temps défini ;

- un compteur de nombre de lignes indexées est maintenu. Une fois qu'il atteint un seuil déterminé, un *commit* est déclenché.



Le schéma ci-dessus résume les principaux modules qui composent le consommateur.

13.6.3. Fiabilité de traitement de fichiers de logs

Chaque consommateur doit s'assurer qu'un fichier de logs récupéré est entièrement traité.

D'après le fonctionnement de la file, dont a parlé au chapitre 13.4, si le consommateur meurt pendant un traitement, l'élément représentant l'id du fichier dont il avait la charge est remis dans la file. Pour éviter que ce fichier ne soit traité depuis son début par un consommateur, et avoir des entrées dupliquées dans l'index de SolrCloud, chaque fichier maintient une métadonnée qui indique la position de la ligne où le consommateur a arrêté son traitement.

Si, pendant le processus de traitement, le consommateur n'arrive plus à accéder au contenu du fichier à traiter (GridFS inaccessible) ou s'il n'arrive plus à communiquer avec Apache Solr, il arrête son traitement et remet le fichier dans la file pour qu'il soit traité par un autre consommateur.

13.7. Configuration dynamique

Les différents éléments composant le système de collecte de logs réalisé dépendent d'un certain nombre de configurations :

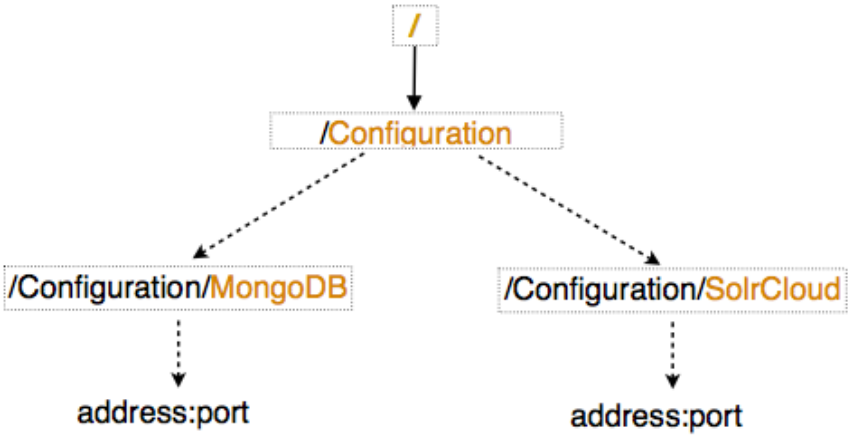
- les producteurs : ils ont besoin de connaître l'adresse de MongoDB ;

- les consommateurs : ils ont besoin de connaître l'adresse de MongoDB et de SolrCloud.

Vu la nature dynamique du *Cloud* et l'instabilité du réseau et des machines, il est préférable d'avoir les différentes configurations indépendantes du code, donc non programmées en dur.

ZooKeeper offre un mécanisme de notification au travers des « *watches* », qui sont des méthodes de *callback* (rappel) et qui sont appelées de manière asynchrone quand un événement déterminé surgit. Un *watch* est typiquement attaché à un znode. Quand ce znode change, tous les *watchers* enregistrés sur ce dernier sont déclenchés sur le client.

Les différents producteurs et consommateurs utilisent les méthodes fournies par l'API configuration qui a été créée pendant le stage. Cette API se base sur les méthodes get_and_watch de la librairie txZookeeper. Elles permettent de récupérer des données de ZooKeeper et d'y attacher une méthode de *callback* qui sera appelée en cas de changement de la donnée récupérée, ce qui peut offrir un mécanisme dynamique de configuration.



Par exemple, si un producteur veut récupérer l'adresse de MongoDB, il passe par l'API configuration qui se charge de récupérer et de surveiller les données du znode /Configuration/MongoDB.

L’adresse de toutes les instances de MongoDB et de SolrCloud doit donc être déclarée dans ZooKeeper pour pouvoir être récupérée par les producteurs et les consommateurs.

Pour être publiées dans ZooKeeper, ces instances sont lancées par l'intermédiaire d’un *script* qui se charge de créer une entrée dans le sous-arbre configuration dans ZooKeeper. Cette entrée est un znode de type éphémère, dont la durée de vie est celle de la durée d'exécution du *script* qui a effectué le lancement de l’instance.
Utiliser ce type de nœud permet de ne pas avoir des entrées invalides dans le sous-arbre de configurations, et qui correspondent à des instances qui ne sont plus en cours d'exécution.

13.8. Politique de suppression de logs

Chaque ligne de logs indexée dans Solr contient un champ fileID qui est L’id du fichier dans GridFS où elle est stockée.

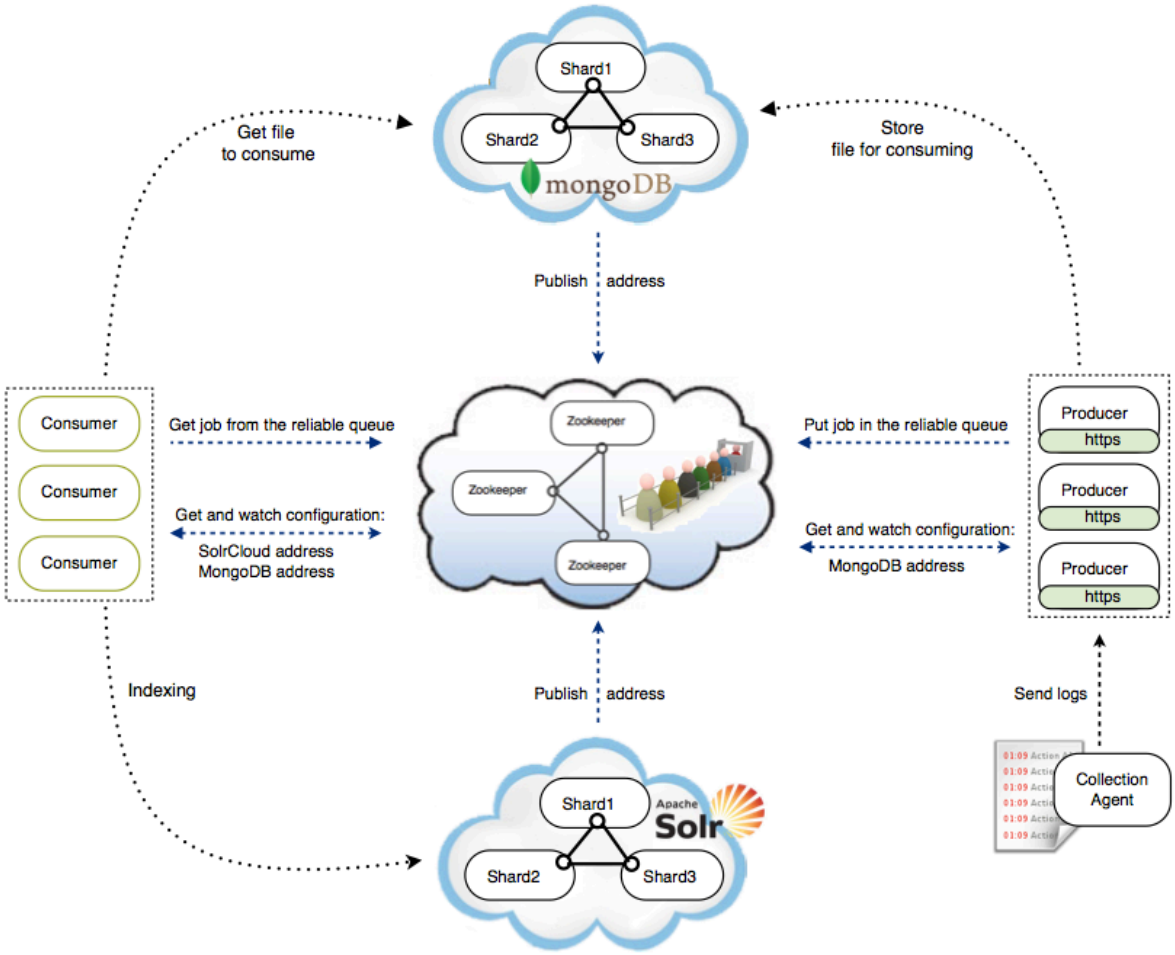
Un *script* a été mis en place pour gérer la suppression des logs dans les deux *backends,* GridFS et Solr. Il prend en paramètre une requête Solr correspondant aux entrées à supprimer. Une fois que les entrées correspondant à cette requête sont récupérées, le *script* les parcourt une par une et effectue le traitement suivant :

- récupération de l’id du fichier où est stockée cette entrée ;
- suppression de l’entrée dans Solr ;
- mise à jour de la métadonnée RemLines (RemLines = RemLines – 1) du fichier correspondent à l’id récupéré au début. RemLines a été initialisé avec le nombre total de lignes contenues dans le fichier.
- si la valeur de RemLines du fichier est égale à 0, cela implique que toutes les lignes qui sont stockées dans ce fichier ne sont plus référencées dans Solr. Dans ce cas le fichier va être automatiquement supprimé dans GridFS.

Ce *script* offre un niveau de requête très fin pour gérer la suppression de logs. Par exemple, on peut facilement supprimer des entrées correspondant à une date précise ou à une source de collecte donnée.

14. Architecture de la solution

L'architecture de la solution finale se résume par le schéma ci-dessous.



L'architecture actuelle de la solution se compose donc de :

- GridFS pour le système de fichier réparti ;
- SolrCloud pour le système d'indexations réparties ;
- ZooKeeper pour le système de coordination.

Comme le montre le schéma, ces composants sont configurés en mode *cluster,* pour tirer profit des aspects distribués et des aspects de fiabilité dont ils disposent. Ce qui offre une fonctionnalité hautement distribuée à toute la solution.

On peut aussi remarquer que les instances SolrCloud et MongoDB publient leur configuration dans ZooKeeper, pour qu'elles puissent être récupérées par les différents producteurs et consommateurs.

15. Intégration à SlapOS

Nous présentons maintenant dans cette partie le fonctionnement du *webrunner*, et son utilisation pour développer les profils et les recettes pour SlapOS.

15.1. Présentation du SlapOS Web Runner

Bien que l'utilisation de cet outil ne soit pas la seule alternative de développement pour SlapOS, elle demeure tout de même la meilleure solution. L'avantage de cet outil est surtout sa disponibilité en tant que service. Son déploiement se fait automatiquement et l'utilisateur peut se connecter depuis n'importe quel navigateur web pour commencer ou continuer son travail. Il permet à toute personne de développer des profils Buildout et ensuite d'exécuter l'application résultante, tout en ayant la possibilité de détecter, comprendre et pouvoir corriger des erreurs qui surviennent pendant la compilation ou le déploiement. Deux interfaces web sont mises à la disposition du développeur :

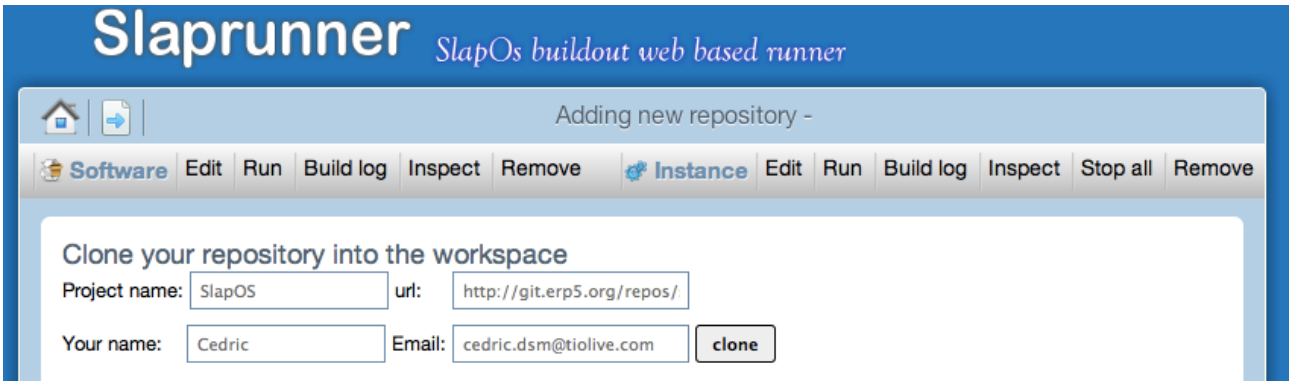
- une interface web permettant de créer et éditer les profils des logiciels : cette partie permet de créer un nouveau logiciel et de l'exécuter à base de SLAPGrid et Buildout. ;

- une interface web permettant de développer des recettes et autres applications pour SlapOS. Cette partie est constituée de l'éditeur libre et *open source* Cloud9. Il est configuré sur l'espace de travail du *webrunner* et permet d'éditer plusieurs fichiers avec beaucoup plus d'ergonomie et d'efficacité.

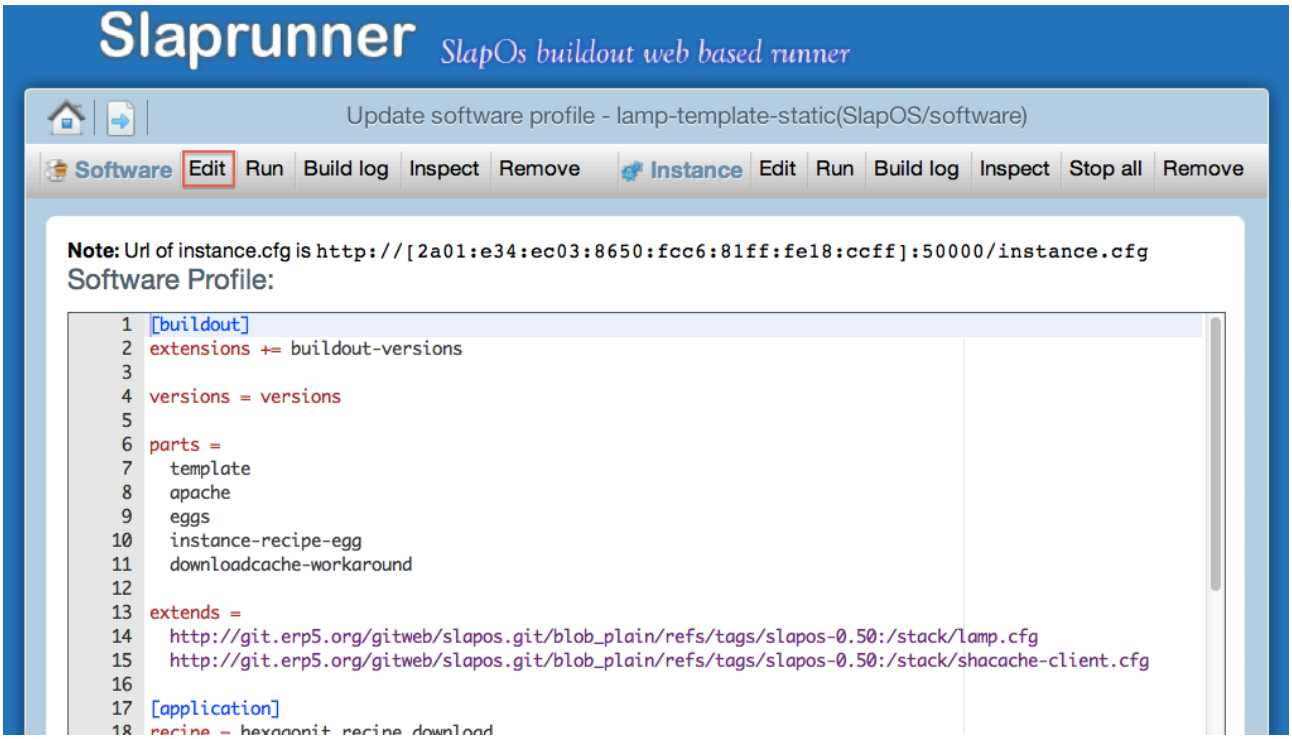
La figure ci-dessous présente la page d'accueil de cet environnement de développement. Nous remarquons surtout le bloc *software* qui permet de compiler le *software release* et l'instance qui concerne l'instanciation.

15.2. Edition des profils avec le *webrunner*

Lors du premier lancement de l'application, l'utilisateur est directement dirigé vers la page permettant de cloner un dépôt GIT. Cette opération permet de créer un premier projet, qui sera manipulé par défaut par l'application. Il faudra alors fournir l'URL d'un projet hébergé sur le dépôt GIT, ainsi que le nom du projet. L'utilisateur peut cloner son projet par SSH ou par HTTPS, ces deux modes lui donnant la possibilité de publier ses changements sur le dépôt originel. Il a aussi la possibilité de cloner un projet en lecture seule (si le dépôt le permet), c'est-à dire qu'il n'aura pas la possibilité par la suite de publier ses changements s'il en fait, car le projet est cloné avec une URL non sécurisée. La figure ci-dessous présente la page de création d'un nouveau projet.

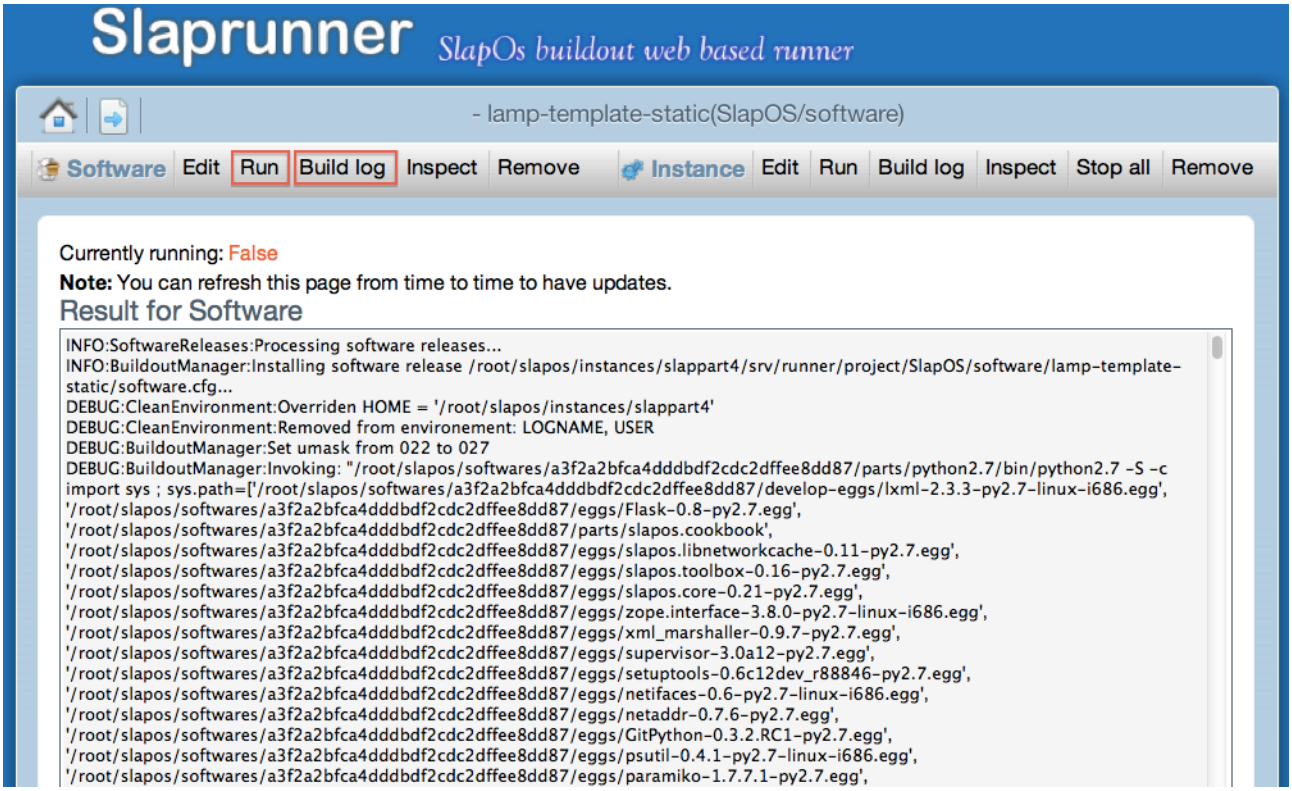


Lorsque cette opération est terminée, l'utilisateur peut soit ouvrir un *software release* contenu dans le projet qu'il vient de cloner, soit en créer un nouveau.

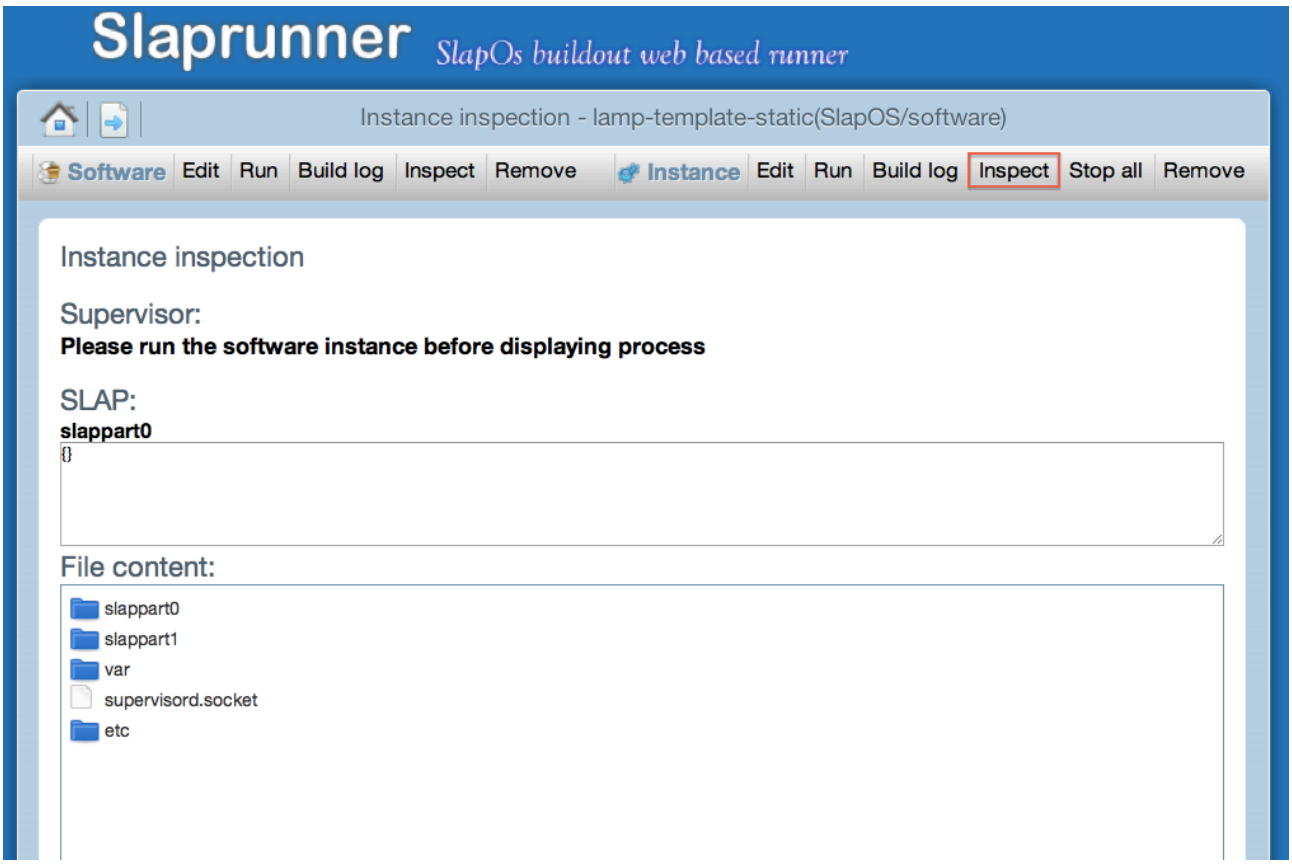


Lorsqu'on lance la compilation, le *webrunner* compile le *software release* en cours d'édition, c'est-à dire celui qui a été ouvert précédemment. Ce lancement redirige vers une

page affichant en temps réel les sorties du processus SLAPGrid. Généralement le processus s'arrête dès qu'il y a une erreur, et l'utilisateur peut donc lire ses messages pour savoir quelle est la source de l'erreur.



Le déploiement d'un logiciel se fait lorsque sa compilation s'est entièrement terminée. Comme dans le cas de la compilation, c'est le *software release* ouvert qui sera déployé dans les partitions du *webrunner*. Lorsqu'elle est lancée, on peut aussi voir en temps réel les sorties de SLAPGrid ; cela permet également de voir s'il y a eu des erreurs pendant cette opération. Après le déploiement, l'utilisateur peut alors, dans la page d'inspection des instances, voir les partitions qui ont été utilisées ainsi que les paramètres nécessaires pour exécuter son application. La figure ci-dessous présente la page d'inspection des instances d'un logiciel installé.



15.3. Conclusion intégration à SlapOS

L'avantage de SlapOS est sa simplicité, mais on se retrouve facilement entouré de contraintes lorsqu'on veut y intégrer des applications. On l'a notamment démontré lors des tentatives d'intégration du système de fichiers GlusterFS.

Porter les applications dans SlapOS nécessite une bonne connaissance de Buildout et du fonctionnement de SlapOS. À cet effet, des formations on été suivies, tout au début du stage, dans les locaux de la société Nexidi, qui est l'entreprise derrière SlapOS.

Le système SlapOS est en constante évolution. Il était donc nécessaire, tout au long du stage, de rester en contact direct avec l'équipe SlapOS, et aussi, lors des réunions, de proposer des idées pour l'améliorer. À cet effet on peut reprocher l'utilisation des droits utilisateur pour faire de l'isolement d'instances, au lieu d'utiliser des technologies comme LXC[12] (Linux Containers) qui est plus flexible et qui se base sur des mécanismes de virtualisation qui sont très légers.

16. Conclusion

Ainsi, j'ai effectué mon stage de fin d'études de Master 2 Systèmes et applications réparties, au sein de l'entreprise Wallix. Lors de ce stage de cinq mois, j'ai pu mettre en pratique mes connaissances théoriques acquises durant ma formation pour participer à un projet ambitieux, qui a représenté un défi technique en terme de passage à l'échelle et de fiabilité de service. Ce stage était pour moi l'occasion de montrer mes compétences dans des technologies d'avenir qui composent le *Cloud computing* et de participer à un projet *open source* d'envergure qu'est Résilience.

Le travail effectué a permis de mettre en place une architecture pour une solution de collecte de logs pour le *Cloud*, hautement distribuée et tolérante aux fautes, et qui se veut indépendante du système du *Cloud* sur lequel elle s'exécute. Puisque aucune partie du code ne dépend du système SlapOS, il peut donc bien s'exécuter sur d'autres systèmes pour le *Cloud*, comme OpenSack.

Ce travail a un double impact. D'un côté il a permis de définir une nouvelle architecture sur laquelle peuvent se baser les prochaines versions de l'*appliance* LogBox de Wallix. Et, d'un autre côté, durant le processus d'intégration de la solution de collecte de logs pour le *Cloud*, j'ai pu enrichir le catalogue d'applications proposées par le système SlapOS, en intégrant un certain nombre de composants dont avait besoin notre solution. Je cite notamment Zookeeper, SolrCloud, txZookeepr, MongoDB, Twisted, MySolr, PyMongo ....

Les études effectuées ont permis de construire les briques de base pour la solution de collecte de logs pour le *Cloud*. Les fonctionnalités du système de base ont, bien sûr, besoin d'être étendues. Je propose, à la suite de cette étude, quelques perspectives en vue d'améliorer le résultat actuel. Elles consistent à :

- intégrer à l'application des mécanismes permettant de détecter s'il n'y a plus d'espace de stockage disponible et d'y remédier par l'instanciation automatique de nouveaux nœuds, permettant ainsi d'étendre cet espace ;
- créer une interface générique pour communiquer avec la solution, et qui permet l'ajout de nouvelles instances (producteur, consommateur, nœud de stockage et nœud d'indexation) au système ;
- développer une interface graphique qui permet de :
    - visualiser l'état de la solution : les différents composants déployés et leur états ;
    - contrôler les instances de composants : permettre de les arrêter, les démarrer, les redémarrer, les supprimer ou les créer ;
- utiliser des sessions lors des communications https pour éviter la vérification de l'identité de la source en accédant à ZooKeeper à chaque envoi de ligne de logs ;
- intégrer d'autres fonctionnalités de la LogBox à la solution (*reporting* ....) ;
- proposer des mécanismes d'analyse de logs pour la détection d'anomalies et pouvoir réagir en conséquence.

VI. Bibliographie

[1] NIST.gov - Computer Security Division - Computer Security Resource Center. Csrc.nist.gov.

[2] Heithem Abbes, Christophe Cérin, and Mohamed Jemni. Bonjourgrid as a decentralised job scheduler. In APSCC 08. Proceedings of the 2008 IEEE Asia-Pacific Services Computing Conference, pages 89-94, Washington, DC, USA, 2008. IEEE Computer Society.

[3] Supervisor : A Process Control System. www.supervisord.org

[4] Jeff Rush, SetupTools : Python Eggs, Dependencies and Plugins - March 25, 2009.

[5] Jeff Rush, Eggs and Buildout Deployment in Python - March 13, 2008.

[6] Patrick Hunt and Mahadev Konar Yahoo! Grid, Flavio P. Junqueira and Benjamin Reed Yahoo! Research. ZooKeeper : Wait-free coordination for Internet-scale system.

[7] Thom White, hadoop the definitive guide. publié par O'REILLY

[8] K. Chodorow. Scaling MongoDB. O'Reilly Media, 2011.

[9] MongoDB, page d'accueil. http://www.mongodb.org

[10] http://www.slapos.org/

[11] http://www.openstack.org/

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

VII. Glossaire de sigles et acronymes

DaaS : *Data as a Service*
ERP : *Entreprise Resource Planning*
FUI : Fonds Unique Interministériel
FUSE : *Filesystem in Userspace*
GNU : GNU *is* Not Unix - acronyme récursif
HDFS : *Hadoop Distribued Dile System*
IaaS : *Infrastructure as a Service*
IPv6 : *Internet Protocol version 6*
JVM : *Java Virtual Machine*
LXC : Linux Containers
PaaS : *Platform as a Service*
RDMA : *Remote Direct Memory Access*
SaaS : *Software as a Service*
SIEM : *Security Information and Event Management*
SlapOS : *Simple Language for Accounting and Provisioning Operating System*
SPOF : Single Point Of Failure
WLB : Wallix LogBox