MSE | MASTER OF SCIENCE IN ENGINEERING

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Master of Science HES-SO in Engineering
Av. de Provence 6
CH-1007 Lausanne

Master of Science HES-SO in Engineering

Orientation : Technologies de l'information et de la communication (TIC)

OpenStack et haute disponibilité

Fait par

Benoît Chalut

Sous la direction de

Prof. Gérald Litzistorf

Dans le laboratoire de Transmission de Données à la **H**aute **E**cole du **P**aysage d'**I**ngénieurie et d'**A**rchitecture

Expert externe : M. Sébastien Pasche, Security and System Engineer à leShop.ch

Genève, HES-SO//Master, 06/02/2014

Accepté par la HES-SO//Master (Suisse, Lausanne) sur proposition de

Prof. Gérald Litzistorf, conseiller de travail de Master
M. Sébastien Pasche, Expert principal

Genève, le 06/02/2014

Prof. Gérald Litzistorf

Prof. Nabil Abdennadher

Table des Matières :

Table des Schémas :

Table des Figures:

Remerciements

Je remercie :

- Monsieur Gérald Litzistorf en sa qualité de professeur responsable pour m'avoir suivi tout au long de ce projet, ainsi que la précieuse aide qu'il m'a apportée dans la rédaction de ce document.
- Monsieur Sébastien Pasche en sa qualité d'expert pour son encadrement et ses précieux conseils qui mon guidés jusqu'à l'aboutissement de ce travail de Master
- Monsieur Khaled Basbous en sa qualité d'assistant et camarade de classe qui m'a épaulé au niveau de la partie sécurité de l'infrastructure OpenStack.
- Madame Stéphanie Favre qui s'est chargée de la relecture consciencieuse de ce rapport.

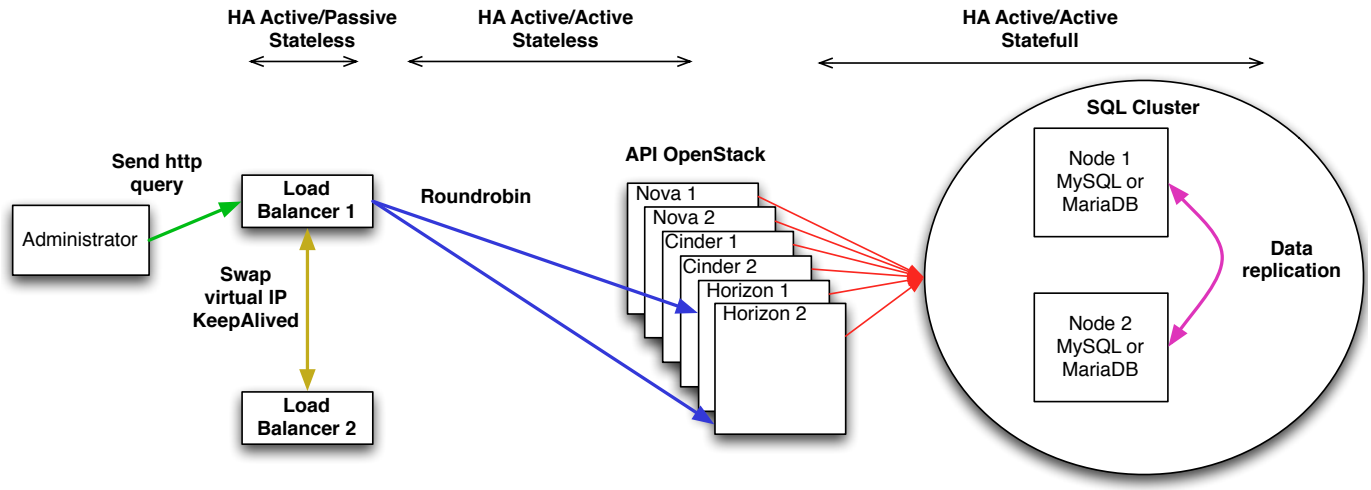
1 Executive Summary

With the emergence of the Information Technologies, we have seen the construction of huge datacenters centralizing many terabytes of data, and significant computing power. These resources provide to users an unlimited access to theirs data and without any geographical limits. This master thesis which took place over 18 weeks, focused on the study and the implementation of an Open Source private Cloud. The main management system of the virtualization infrastructure is OpenStack.
The first part of this work is based on the understanding of the functioning of this system, because it is composed of different services:

- The virtual machine network management, with the transition between the private/public network (Nova-network)
- The virtualization management that communicates with Kernel Virtualization Module in each hypervisors
- The storage management with the creation of persistent storage by blocks based on the iSCSI protocol

This first part of my work also defines the role of each physical server.

The second part is about the study and integration of a high availability system on the OpenStack services with the setup of a second Cloud Controllers. Many types of high availability has been studied:



- Active/Active Stateless: For the API services with the integration of a load balancer Open Source which splits the queries between the two Cloud Controllers and check the availability of both of them
- Active/Passive Stateless: For the availability of the two loads balancers with the swap of virtual IP managed by KeepAlived
- Active/Active Statefull: For the availability of the SQL database through data replication

In the third part, I studied the supervision system, because with this type of infrastructure, which includes many hardware and services, it's most important to control the health of Cloud to detect services errors or server overload. Shinken is a monitoring system, which analyzes the physicals resources and the state services. Moreover, the logs generated by the OpenStack services are stored in a database on virtual machines inside the Cloud, as JSON Objects. We can recover these logs with kibana client, in order to extract logs severities.

1.1 Cahier des charges

L'entreprise LeShop.ch leader dans le domaine du e-commerce alimentaire, doit traiter au sein de son Data Center des millions de requêtes par jour de la part des ses clients et fournisseurs. Dans un esprit de gestion efficiente des ressources, il serait impensable d'avoir un ratio 1:1 entre le matériel physique et les services Web, c'est pour cela qu'un grand nombre de ses services sont hébergés au sein de Machines Virtuelles (VM). L'objectif de l'entreprise est de posséder dans un avenir très proche une infrastructure de Cloud privé Open Source. Ce travail de Master vise à mettre en place une infrastructure basée sur OpenStack. Il va être découpé en plusieurs étapes :

1. Etude du fonctionnement d'OpenStack
2. Mise en place d'une infrastructure pour l'hébergement de VMs hébergeant des serveurs Web
3. Intégration de la Haute disponibilité au sein des services d'API d'OpenStack
4. Etude et mise en place d'une application métier pouvant être hébergée au sein de l'infrastructure
5. Etude d'une solution de monitoring basée sur Shinken, afin de déceler les éventuelles erreurs dans le comportement de l'infrastructure

1.2 Guide du lecteur

Ce travail de Master s'inscrit dans la suite de mon travail de Bachelor, qui s'était intéressé à la gestion d'une infrastructure de moindre ampleur que celle traitée dans ce rapport, basée sur OpenNebula : http://www.tdeig.ch/kvm/Chalut_RTB.pdf
Cette liste a pour but de localiser les grands thèmes de ce rapport :

- Chapitre 2 : Explication des différences entre Cloud public/privé
- Chapitre 3 : Gestion d'un Cloud privé avec OpenStack
- Chapitre 4 : Etude du fonctionnement d'OpenStack, à travers l'étude de deux scénarios mettant en place deux types de stockages : NFS et iSCSI
- Chapitre 5 : Etude de l'intégration de la haute disponibilité au sein du Cloud Controller et de la base SQL
- Chapitre 6 : Mise en place d'une architecture de gestion des Logs sous forme d'objets JSON
- Chapitre 7 : Supervision grâce à Shinken des services et services physiques permettant de détecter des anomalies
- Chapitre 8 : Recommandations à propos de la sécurité, particulièrement au niveau du cloisonnement des rôles des administrateurs

Key-words: Private Cloud, OpenStack, High availability, Storage, Logs management, Shinken, Open Source

2 Qu'est-ce que le Cloud

L'émergence du Cloud computing a permis l'apparition des nombreuses applications en ligne au service des utilisateurs. Ceci a nécessité la création de gigantesques infrastructures physiques, hébergeant des centaines d'hyperviseurs permettant l'exécution et la gestion de milliers de VMs. Ce type de topologie est également appelé infrastructure virtualisée. Le Cloud peut être utilisé dans différents domaines :

- **Le calcul intensif** :

Un **grand nombre de VMs éphémères** vont exécuter le même calcul mais avec des données différentes. Cette technique est utile pour **traiter un grand volume de données**. Comme chaque VM va prendre une petite partie des données globales, nous pouvons facilement **paralléliser le calcul**. Dans ce type d'application, il est important de pouvoir démarrer de nombreuses VMs, mais une fois le calcul terminé, les VMs sont détruites. L'hepiaCloud hébergé dans les locaux de l'hepia est un Cloud mis au service de la science, afin que de pouvoir effectuer du calcul intensif : http://www.lsds-rg.org/hepiacloud/

- **L'hébergement d'applications métiers** :

Chaque VM du Cloud aura un **rôle différent,** comme héberger par exemple des serveurs Web ou des clusters de bases de données. Contrairement au cas précédent chaque VM, ainsi que les données doivent être pérennes dans le temps. Le nombre de VMs sera moins important, mais l'accent sera mis sur la disponibilité de la ressource. **Mon Travail de Master va se baser sur cet aspect du Cloud.**

2.1 Différences entre Cloud privé/public

En fonction des besoins d'une entreprise, deux types de Cloud peuvent être mis en place.

2.1.1 Cloud Public

Les **administrateurs du Cloud** (sur schéma 1 Cloud User) vont pouvoir l'interroger **depuis internet,** par conséquent, le point d'entrée est potentiellement visible de tous. Ceci nécessite de mettre en place une batterie de contrôles afin de garantir la confidentialité, l'intégrité et l'authentification.

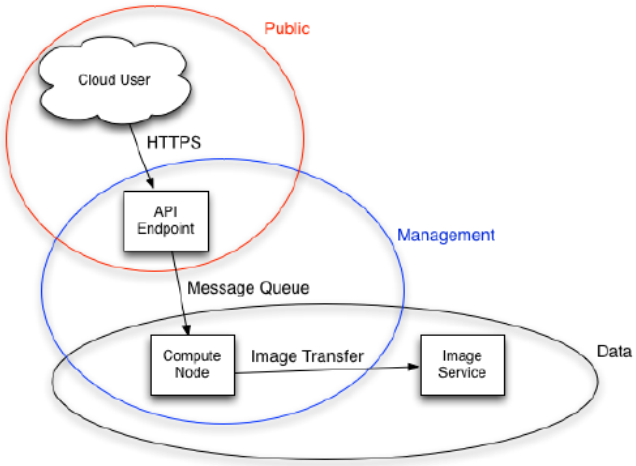


Schéma 1: Public Cloud (Basbous 2013)

Comme illustré sur le schéma 1, le **point critique** est la **transition** entre le réseau **public** (Rouge) et le réseau de **management** (Bleu). Par conséquent, seuls les administrateurs authentifiés pourront effectuer des requêtes de démarrage de VMs.
Cette topologie est utilisée par de gros providers tels que Amazon avec Elastic Compute Cloud (EC2). Ce type de service peut être intéressant pour les entreprises ne voulant plus gérer leur infrastructure physique. Grâce à ce système, elles vont louer une ou plusieurs VMs qui seront hébergées au sein du Cloud EC2, et donc les administrateurs situés dans le réseau public s'occuperont uniquement de leurs VMs. Avec les Cloud Publics, une nouvelle forme de tarification est apparue, le client va être facturé à l'heure d'utilisation de sa VM. Voici un extrait des tarifications proposées pour deux types de VMs :

Tarifs des instances à la demande

Région: UE (Irlande)

	vCPU	ECU	Mémoire (Gio)	Stockage des instances (Go)	Utilisation de Red Hat Enterprise Linux
Usage général - Génération actuelle					
m3.xlarge	4	13	15	2 x 40 SSD	$0,555 par heure
m3.2xlarge	8	26	30	2 x 80 SSD	$1,120 par heure

Schéma 2: Tarification Amazon

Le modèle économique pour les clients d'un Cloud Public est de payer ce qu'il consomme. Mais de nombreux problèmes liés à la dématérialisation des données apparaissent, car en en augmentant le niveau d'abstraction, Amazon va héberger les VMs de milliers de clients. Donc potentiellement deux entreprises concurrentes se verraient héberger leurs données dans une même unité de stockage au sein d'Amazon. Un des gros problèmes du Cloud Computing Public est que ces infrastructures gigantesques sont vues comme des systèmes « black boxes ». C'est pour cela que la solution de Cloud Public sur Amazon par

exemple n'a pas été retenue, du fait qu'il est essentiel pour une entreprise comme LeShop, qu'elle ait la main mise sur toutes les couches de son infrastructure.

2.1.2 Cloud Privé

Comparé au Cloud précédent, il sera de taille moins importante, et sera hébergé dans les locaux de l'entreprise qui l'exploite. **L'administration** du Cloud, ne sera **pas ouverte sur l'extérieur**.

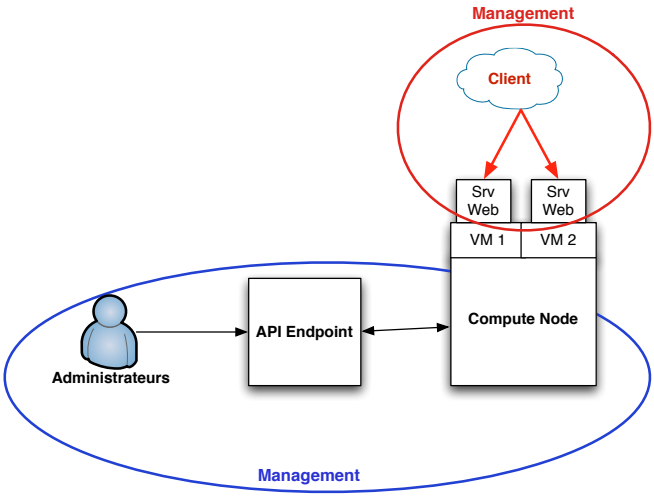


Schéma 3: Cloud privé

Comme le montre le schéma 3, les **administrateurs** du Cloud (à gauche) se situent directement **au sein de l'infrastructure**. L'avantage de ce genre d'architecture est que l'entreprise va optimiser son Cloud pour ses applications métiers. En effet, comme elle est la seule à le gérer et l'exploiter, elle aura une parfaite connaissance de tous les éléments le constituant. Les seules parties publiques sont les applications métiers sur les VMs. Cependant, posséder une infrastructure comme celle-ci, a un coût car au lieu de gérer uniquement les VMs hébergées sur Amazon, il faut également gérer toute l'infrastructure physique.

3 Cloud privé avec Openstack

Pour gérer de bout en bout un Cloud privé, mon projet de Master se base sur l'étude et la mise en place d'OpenStack. La Nasa un des grands acteurs du Cloud grâce à sa plateforme Nebula[1], et Rackspace un des leaders mondial de l'hébergement mutualisé, ont décidé de regrouper leurs compétences afin de développer le projet OpenStack. Il a pour but de créer et de gérer une infrastructure virtualisée sur du matériel standard. Pour cela, Openstack va se baser sur différents services qui seront présentés en §4.1.4. Tous ces services vont servir à créer une abstraction entre le matériel (OS, gestion de la virtualisation) et l'utilisateur final.

3.1 OpenStack en quelques chiffres

OpenStack est solution Open Source implémentée au sein de grands groupes tels que :

- Cisco : Hébergement de leur plateforme WebEx, permettant d'effectuer de la vidéoconférence professionnel : http://www.openstack.org/user-stories/cisco-webex/
- Le CERN : Stocke et analyse les données générées par les capteurs de particules du LHC (Large Hadron Collider) dans son Cloud OpenStack. Il possède environ 700 hyperviseurs hébergeant 2200 VMs : http://indico.cern.ch/getFile.py/access?contribId=217&sessionId=8&resId=0&materialId=slides&confId=214784

3.2 Les réseaux physiques

Le schéma 4, nous montre les différents serveurs physiques qui vont constituer l'infrastructure OpenStack :

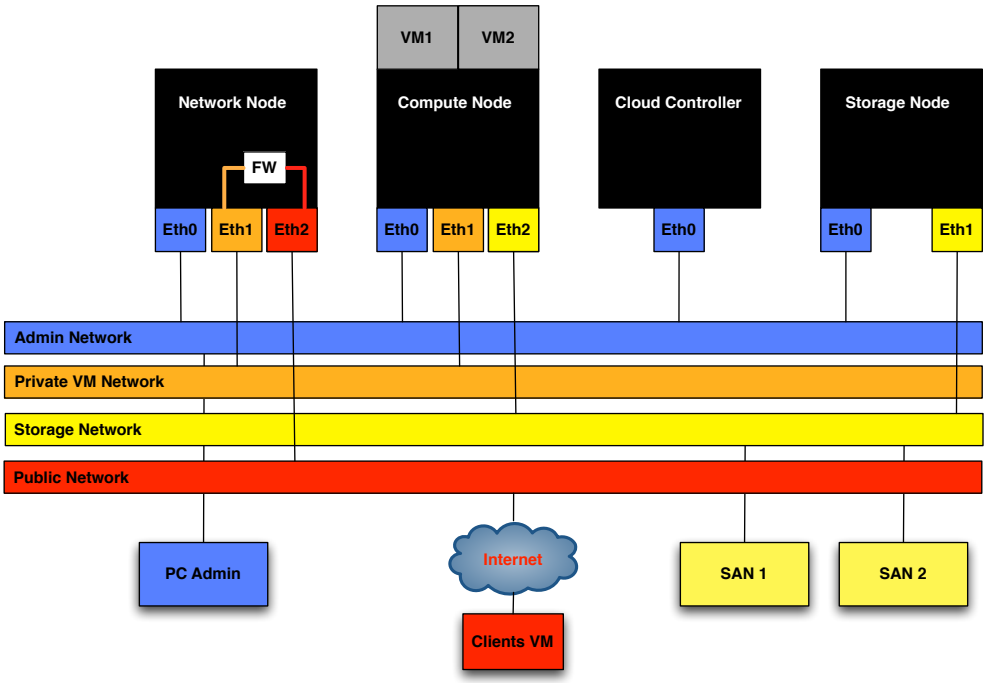


Schéma 4: Topologie globale

[1] Genèse d'OpenStack : http://oig.nasa.gov/audits/reports/FY13/IG-13-021.pdf

J'ai choisi de diviser mon infrastructure en **quatre réseaux physiques distincts**, afin de mieux séparer les types de communications qui vont avoir lieu entre les différents éléments :

- **Le réseau de management** (Bleu): Considéré comme le Backbone du Cloud. Il fait transiter les communications entres les différents services Openstack et permet aux administrateurs de venir le gérer.
- **Le réseau privé des VMs** (Orange): Permet d'interconnecter les VMs entres elles. Il n'est pas directement accessible depuis le réseau public car c'est le Network Node qui sert de firewall depuis l'extérieur.
- **Le réseau de stockage** (Jaune) : A travers ce réseau circule uniquement les données utiles des VMs. La VM utilise ce réseau pour dialoguer avec son disque virtuel grâce au protocole NFS ou iSCSI.
- **Le réseau public** (Rouge): Représente le réseau Internet, nous ne devons avoir aucune confiance en ce réseau, car il comporte des utilisateurs légitimes, mais également des utilisateurs illégitimes tels que des « Black Hat ». La jonction entre le réseau rouge et orange est gérée par le Network Node, qui va directement agir sur le Firewall Iptables (Voir §4.1.7).

4 Scénario 1 : Serveur web virtualisé au sein d'Openstack

Ce scénario a pour but de mettre en place une infrastructure virtualisée fonctionnelle, afin de mieux **comprendre les interactions** entre chaque service intervenant au sein d'Openstack. Au final, nous aurons un **serveur web hébergé sur une VM** que nous pourrons interroger depuis une adresse IP public. Dans un souci de bonnes pratiques, et dans le but de garantir la sécurité, le port 80 de la VM sera le seul port ouvert.

Ce scénario va se matérialiser sous forme de deux variantes afin de mettre en évidence les différents types de stockage que nous offre OpenStack. La première sera la mise en place d'un **stockage partagé** basé sur le protocole **NFS** grâce au NAS QNAP. La deuxième sera l'intégration d'un système de **stockage par bloc** grâce au protocole **iSCSI**.

La gestion d'une infrastructure virtualisée fait intervenir de nombreux éléments tels que la gestion des VMs avec Libvirt, et la gestion du réseau. OpenStack est donc constitué de nombreux services, permettant de piloter ces différents éléments de manière totalement dynamique.
Comme les différents services sont répartis sur plusieurs PC physiques, ils doivent communiquer entres eux grâce au LAN Bleu. Pour cela, ils vont utiliser le protocole Advanced Message Queuing Protocol (Message AMQP) (voir §4.1.3).

4.1 Scénario 1a : Système de stockage NFS

Le schéma 5 représente les différentes entités physiques exécutant les différents services utiles au bon fonctionnement de l'infrastructure Openstack.

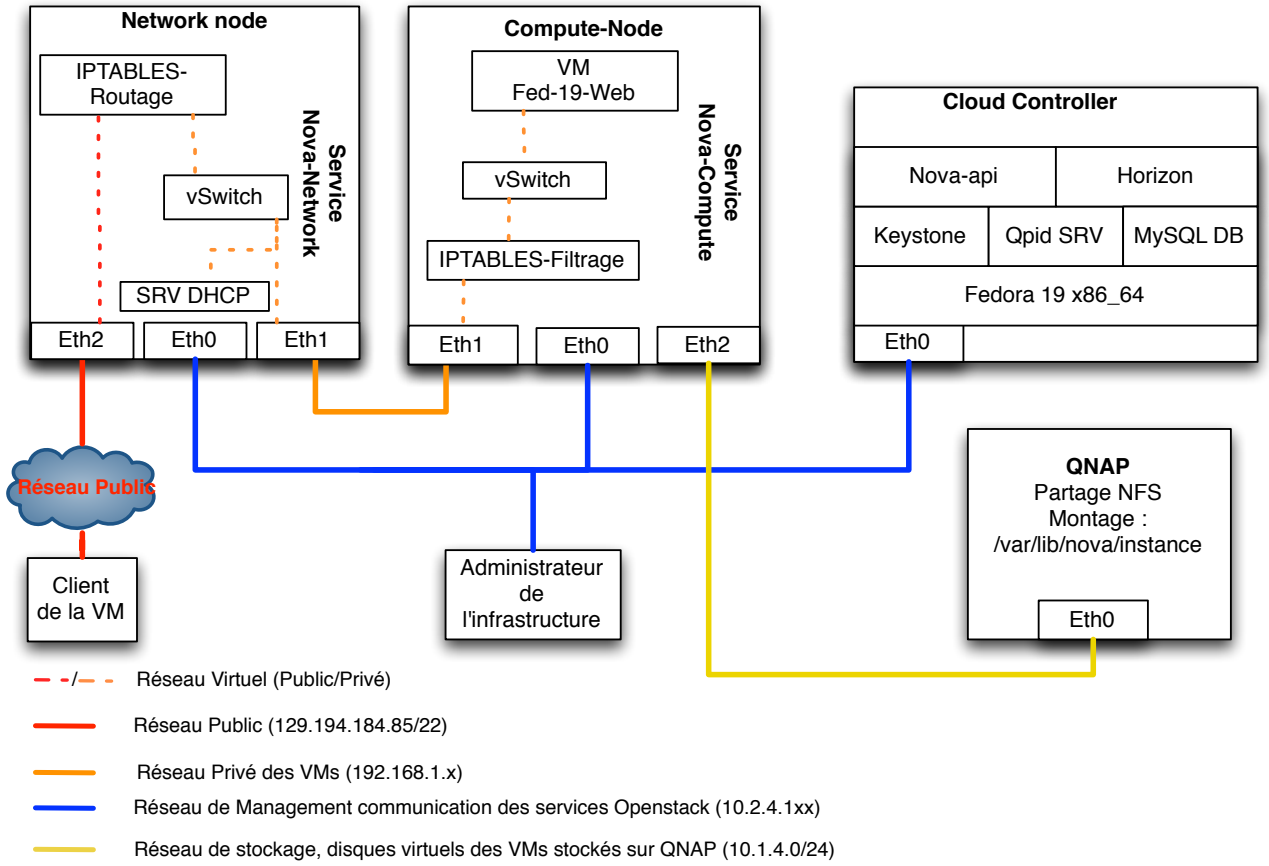


Schéma 5: Scénario 1a

4.1.1 Les entités physiques

Tous les serveurs physiques, utilisent **Fedora 19**. J'ai choisi cette distribution, du fait de ces nombreux points communs avec la distribution professionnelle Red Hat Entreprise Linux (RHEL) proposée par Red Hat. La future version de RHEL (RHEL 7), sera basée sur Fedora 19. Fedora est la version Open Source de RHEL, c'est à dire que la version professionnelle propose en plus uniquement le support, mais les fonctionnalités sont identiques.

Toutes les entités physiques sont basées sur le même hardware :

- Carte mère : Asus P8Q77-M
- CPU : Intel Core i5-3330
- RAM : 8GB
- Disque Dur : 320GB
- Cartes réseaux 1Gb/s Intel

Voici le rôle de chaque serveur physique :

- **L'administrateur du Cloud** : Personne physique, qui depuis un navigateur web, va dialoguer avec le Cloud Controller pour démarrer une VM, gérer les contrôles d'accès ou administrer notre infrastructure.
- **Cloud Controller** : La pièce maitresse de notre infrastructure, il va piloter les différentes parties de notre infrastructure en envoyant des messages aux autres éléments physiques.
- **Compute-Node** : Héberge la couche de virtualisation (Hyperviseur), qui dans notre cas est matérialisée par le module KVM (Kernel-Based Virtual Machine) qui sera rajouté au Noyau Linux. Dans le jargon Openstack un hyperviseur est appelé un Compute Node.
- **Network Node** : Gestion des réseaux dédiés aux VMs, il va aiguiller les paquets venant du réseau public jusqu'à la VM. Cette entité va servir de lien entre le réseau public et le réseau privé en appliquant des règles de sécurité, tout comme un firewall.
- **Le client de la VM** : Dans notre scénario, ce client va interroger via son browser, le serveur Web situé sur une VM possédant une IP public.
- **Le QNAP** : Serveur NFS, afin d'héberger le disque virtuel de la VM.

Les différents services seront présentés en §4.1.4.

4.1.2 L'utilité des services Openstack

Le but premier d'Openstack est d'**ajouter un service au dessus des éléments intégrés au noyau** Linux. Ils vont traduire les messages standardisés transitant à travers le réseau, en commandes interprétables par les modules du noyau tels que KVM ou Iptables.

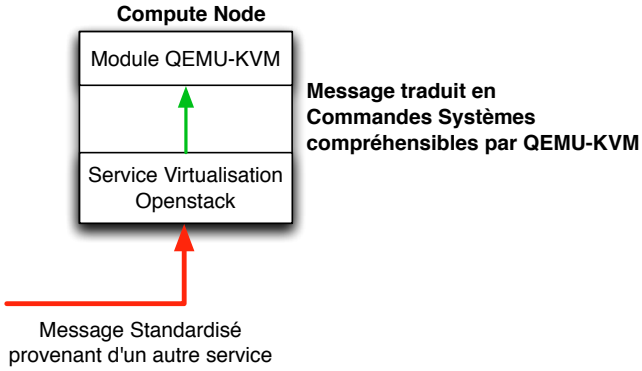


Schéma 6: But des services OpenStack

A travers le schéma 6, nous pouvons comprendre que le service de virtualisation d'Openstack situé au niveau de l'hyperviseur, va recevoir des messages standardisés du Cloud Controller et va les traduire en commandes compréhensibles par KVM. Grâce à ce service, un même message pourrait démarrer une VM soit sur un hyperviseur KVM ou soit sur l'hyperviseur ESX. Ces derniers posséderaient chacun un service de virtualisation OpenStack, qui traduirait le message en commandes interprétables par son système.

4.1.3 Echange de messages

Deux types de communications vont s'effectuer au sein de notre infrastructure :

- **Communication via API**

Les différents **services Openstack fournissent une API**[2], c'est à dire que le service va proposer une interface afin de pouvoir interagir avec ces méthodes ou fonctionnalités. Pour interagir avec les API, le système de requêtes REST (Representational State Transfer) est très employé au sein de l'infrastructure OpenStack, c'est à dire que l'on peut envoyer au service une URL, contenant tous les paramètres que la méthode a besoin pour s'exécuter[3]. Cet échange se fait via le protocole http. L'API va nous retourner un objet JSON, qui pourra être interprété par le client. Un service d'API, comme présent sur le Cloud Controller va écouter sur un port particulier afin de recevoir les requêtes http, et va convertir ces dernières en message AMQP.

- **Communication via file de messages (Advanced Message Queuing Protocol) AMQP**

Afin d'**unifier la communication réseau entre les services**, le système adopté a été de mettre en place une infrastructure de communication inter-serveur basée sur des files de messages, grâce au protocole AMQP. Le protocole est géré par le serveur Qpid qui est le courtier hébergé sur le Cloud Controller. Le système de queues permet aux **services de travailler de manières non synchronisées** pour qu'ils puissent s'envoyer des messages en mêmes temps.
Les appels de procédures distantes entre les services se font par l'envoi de messages RPC (Remote procedure call) encapsulé dans le protocole AMQP.
Chaque service Openstack instancie deux queues :

- Une queue pour recevoir les messages envoyés à tous les nœuds d'un même type, donc par exemple tous les Nova-Compute.
- Une queue spécifique à serveur physique, afin d'envoyer une instruction correspondant à un serveur physique en particulier grâce à son ID. Par exemple instancier une VM sur un hyperviseur en particulier.

Le Scheduler (Qpid Server) va servir de table de routage afin d'aiguiller les messages. Il connaît la description de tous les nœuds ainsi que toutes les procédures que l'on peut appeler à distances. Les services recevant des messages AMQP comme Nova-Compute, n'écoutent pas sur port particulier, mais maintiennent une liaison permanente en Unicast avec le serveur QPID.
Au moment d'installer l'infrastructure, l'analyse des messages AMQP se révèle un très bon moyen pour **comprendre les interactions entre les différents services**, et nous renseigne sur les différents problèmes entres les services car les messages échangés sont stockés dans les Logs (Voir §4.1.6 exemple d'échange). En ce qui concerne le Scheduler, je n'ai pas eu besoin d'aller analyser le comportement des queues de

[2] Fonctionnement d'une API :
http://blog.atinternet.com/fr/index.php/2012/03/19/solutions/les-apis-veritables-outils-daide-a-la-decision/1751
[3] Exemple de requêtes REST :
http://docs.openstack.org/content/index.html#Openstack-API-Concepts-a09234

communication, car cette partie est correctement implémentée au sein des différents services.

Le schéma 7 résume les **deux types d'échanges** qui résident au sein d'OpenStack (extrait de http://docs.openstack.org/developer/nova/devref/rpc.html) :

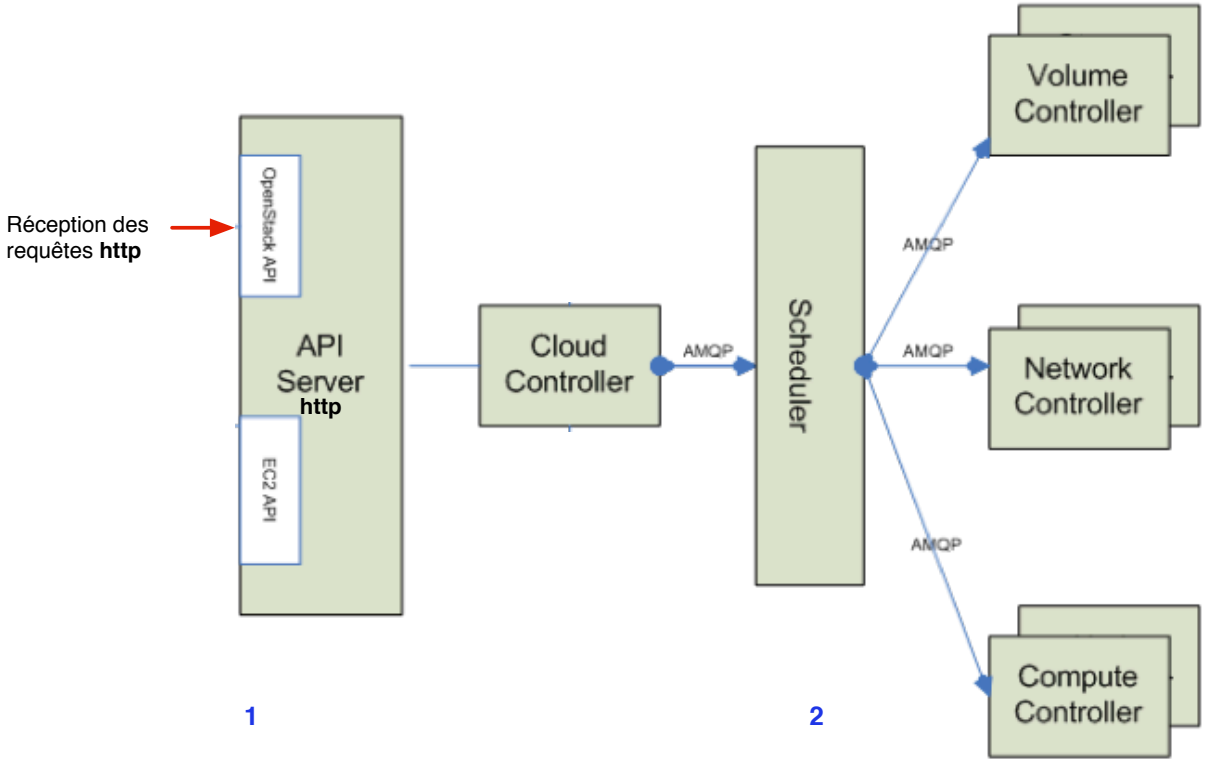


Schéma 7: Différents types de messages

En 1, sont représentés les services réceptionnant les requêtes http provenant d'un client d'API (voir §4.1.4.1). Cette requête va être transformée sous forme de **message AMQP (2)**, afin d'être distribuée au service concerné, par l'intermédiaire du QPID serveur. L'avantage de ce type de message est qu'il utilise un formatage prédéfini. Par conséquent, un même message peut être compris par différents langages de programmation et différents services.

4.1.4 Les services OpenStack

Pour fonctionner correctement, notre infrastructure Openstack a besoin de plusieurs éléments ayant des rôles bien spécifiques. Openstack est constitué de **plusieurs services pouvant communiquer entre eux afin de gérer toutes les facettes de notre infrastructure**.

4.1.4.1 Administration de l'infrastructure Openstack

Pour administrer notre infrastructure, deux solutions s'offrent à nous. En ce qui concerne l'envoi de commandes comme par exemple une création de VM se fait via des requêtes http à une API. Nous pouvons utiliser un **client d'API** (par exemple python-novaclient détaillé en annexe A.2) CLI écrit en python, qui va se charger d'encapsuler dans une requête http les paramètres de la commande :

```
nova boot --flavor 2 --image 397e713c-b95b-4186-ad46-6126863ea0a9
```

Nous avons aussi la possibilité d'administrer notre infrastructure via **une interface Web** appelée Horizon Dashboard. Cette interface Web va se charger de communiquer avec les différentes API. L'utilisateur effectue une requête au près du site Web Horizon, et le serveur Web se charge de dialoguer avec l'API en question.

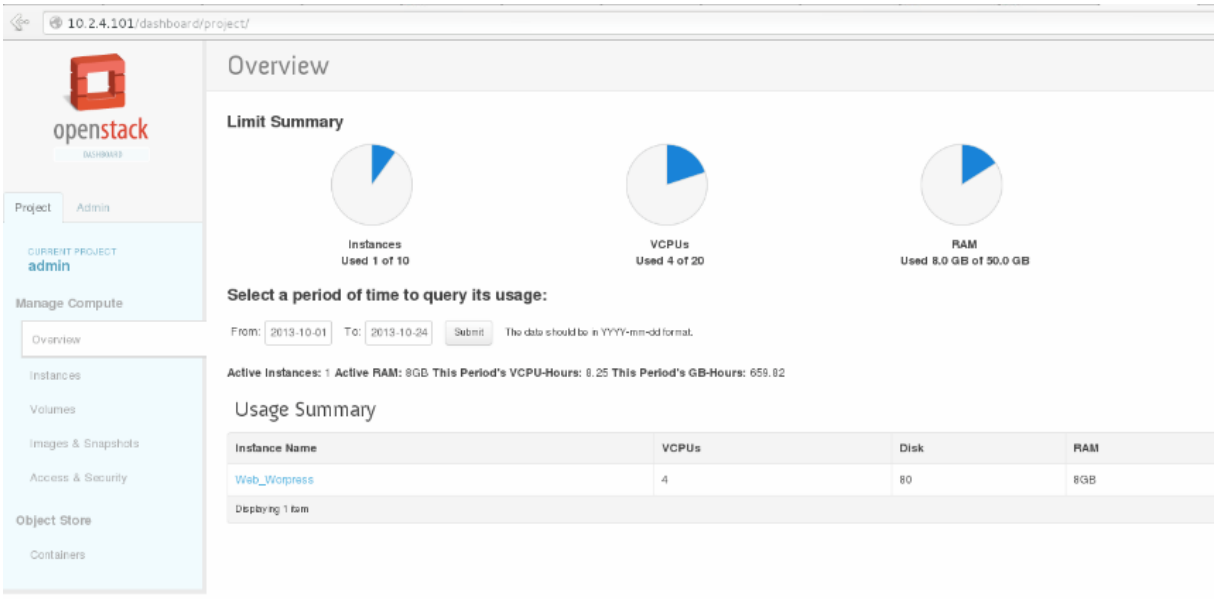


Figure 1: Interface GUI Horizon

4.1.4.2 Gestion de la virtualisation

Quand nous parlons de gestion de la virtualisation, nous devons **piloter le module de virtualisation QEMU-KVM** de l'hyperviseur, ainsi que les règles du firewall afin de contrôler les points d'entrées depuis le réseau public, pour cela nous avons besoin des services Nova. **Trois éléments vont constituer cette gestion**[4] :

Nova-API : Ce service propose une API qui pourra être utilisée via CLI en effectuant des requêtes REST par l'administrateur physique. Les requêtes envoyées par l'administrateur vont être redirigées via des messages AMPQ aux différents services pouvant remplir la tâche voulue par l'administrateur. Il **pourra remonter des informations sur la VM**, comme son adresse IP, ou recevoir des ordres de création de VMs.

Nova-Compute : Ce service est installé sur l'hyperviseur et communique directement avec le **module QEMU-KVM**, grâce à la librairie libvirt. Il va traduire les messages AMQP reçu de Nova-API en commandes compréhensibles par KVM.

Ordonnanceur OpenStack: Au sein d'une infrastructure possédant plusieurs hyperviseurs, **un service d'orchestration doit être mis en place** afin de pouvoir choisir judicieusement sur quel hyperviseur une VM va être démarrée. Nous avons deux types de techniques d'occupation des VMs sur l'hyperviseur :

[4] Architecture Nova : http://my.safaribooksonline.com/book/operating-systems-and-server-administration/virtualization/9781449311223/understanding-nova/nova_architecture

- Répandre : L'ordonnanceur essaie de **répartir les VMs sur l'ensemble des hyperviseurs**, afin d'équilibrer les charges entre chacun. Il va regarder celui qui a le plus de RAM disponible.
- Remplir : L'ordonnanceur essaie de **remplir au maximum les hyperviseurs** afin d'en solliciter un minimum. Cette technique est intéressante afin de réduire son empreinte écologique car un minimum de machines physiques est opérationnel. Par conséquent, celles activent risquent d'être surchargées en cas de pic de charge.

Cependant l'ordonnanceur présente des **limites** au moment où les **hyperviseurs ont des quantités de RAM hétérogènes**. Si un hyperviseur possède beaucoup plus de RAM que tous les autres, malgré le fait que l'on favorise l'étalement, il sera plus sollicité à cause de son importante RAM disponible. Donc il est vivement conseiller d'avoir du matériel uniforme, ou que l'ordonnanceur ajoute dans son choix la quantité de vCPU disponible par hyperviseur.
Pour effectuer les mesures de charges, chaque hyperviseur envoie à l'ordonnanceur sa quantité de RAM disponible. Cette mesure est effectuée à partir des profiles attribués aux VMs s'exécutant sur l'hyperviseur. Par conséquent, si une VM est éteinte la quantité de RAM physique qui lui est allouée est considérée comme consommée. **Les valeurs sont statiques et non dynamiques**.

Nova-Network : Ce service est installé sur notre Network Node, il va physiquement faire la **liaison entre le réseau public (rouge) et le réseau privé** des VMs, car dans notre scénario la VM hébergeant un serveur Web doit pouvoir être interrogée depuis le réseau Internet global. Le service va interagir avec le firewall iptables, cette entité physique doit être vue comme un routeur. Le service Nova-Network va pouvoir proposer une IP public à une VM, pour cela il va utiliser la technique du **floating IP** (Voir §4.1.7).

4.1.4.3 Gestion de l'authentification (Keystone)

Le service **Keystone** situé sur le Cloud Controller permet d'offrir une **gestion des identités et des autorisations d'accès** pour les utilisateurs. Le service Keystone va permettre de contrôler de façon centralisée les accès aux différentes API. Le système de validation est basé sur un **système de token**. Ce système a été implémenté afin d'avoir des périodes d'accès limitées dans le temps. Nous pouvons également cloisonner les utilisateurs, car ils peuvent se voir attribuer un token qui leur permettra uniquement de créer des VMs, ou un autre uniquement pour la gestion des volumes de stockages. En somme nous pouvons créer des contrôles d'accès aux APIs. Comme nous avons une **infrastructure de type PKI**, le token signé est contenu dans un message CMS « Cryptographic Message Syntax[5] », par conséquent Keystone est une autorité de certification (CA)[6]. Le schéma 8 représente les échanges d'authentification de l'administrateur désirant envoyer une requête à une API.

[5] CMS : http://www.vocal.com/secure-communication/cryptographic-message-syntax-cms/
[6] Explication de Keystone : http://www.mirantis.com/blog/understanding-openstack-authentication-keystone-pki/

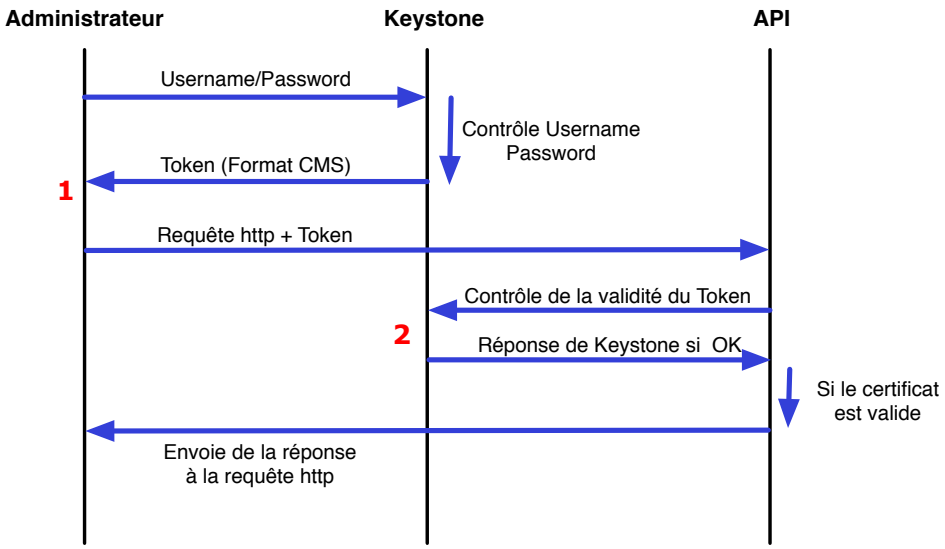


Schéma 8: Authentification Keystone

A l'installation du service Keystone, de par son rôle **d'autorité de certification**, nous devons générer en local sur le Cloud Controller grâce à l'outil OpenSSL:

- La clé privée de la CA :
  ```
  openssl genrsa -out /etc/keystone/ssl/certs/cakey.pem 2048 -config
  /etc/keystone/ssl/certs/openssl.conf
  ```

- Générer le certificat de la CA avec sa clé privé :
  ```
  openssl req -new -x509 -extensions v3_ca -passin pass:None -key
  /etc/keystone/ssl/certs/cakey.pem -out /etc/keystone/ssl/certs/ca.pem -days
  3650 -config /etc/keystone/ssl/certs/openssl.conf -subj
  /C=US/ST=Unset/L=Unset/O=Unset/CN=www.example.com
  ```

- Génération de la clé privée pour signer les Tokens :
  ```
  openssl genrsa -out /etc/keystone/ssl/private/signing_key.pem 2048 -config
  /etc/keystone/ssl/certs/openssl.conf
  ```

- Génération du certificat pour vérifier les Token :
  ```
  openssl req -key /etc/keystone/ssl/private/signing_key.pem -new -nodes -out
  /etc/keystone/ssl/certs/req.pem -config /etc/keystone/ssl/certs/openssl.conf
  -subj /C=US/ST=Unset/L=Unset/O=Unset/CN=www.example.com
  ```

Explications des points 1 et 2 du schéma 8 :

1. Une fois la vérification Username/Password effectuée, Keystone va générer un token signé sous la forme d'un CMS en utilisant sa clé privée (Signing_key.pem) grâce à la commande OpenSSL:
   ```
   openssl cms -sign -signer /etc/keystone/ssl/certs/signing_cert.pem -inkey
   /etc/keystone/ssl/private/signing_key.pem -outform PEM -nosmimecap -nodetach
   -nocerts -noattr
   ```

2. L'API envoie à Keystone le token reçu afin qu'il puisse vérifier sa signature grâce au certificat req.pem.

4.1.4.4 Les Bases de Données

Afin de **stocker les différentes informations relatives à notre infrastructure**, le Cloud Controller possède un serveur de base de données MariaDB[7], qui est un dérivé de MySQL. Pour configurer OpenStack, je n'ai pas eu besoin d'effectuer des requêtes SQL, donc les contenus des bases est présent à titre indicatif. Le serveur va posséder deux bases de données :

- **Base Keystone** : Contient les utilisateurs, les tokens délivrées, ainsi que leurs périodes de validités
  Aperçu des champs de la **table user** de la **base Keystone** :

```
MariaDB []> use keystone;
MariaDB [keystone]> describe user;
+-------------------+--------------+------+-----+---------+-------+
| Field             | Type         | Null | Key | Default | Extra |
+-------------------+--------------+------+-----+---------+-------+
| id                | varchar(64)  | NO   | PRI | NULL    |       |
| name              | varchar(255) | NO   |     | NULL    |       |
| extra             | text         | YES  |     | NULL    |       |
| password          | varchar(128) | YES  |     | NULL    |       |
| enabled           | tinyint(1)   | YES  |     | NULL    |       |
| domain_id         | varchar(64)  | NO   | MUL | NULL    |       |
| default_project_id | varchar(64)  | YES  |     | NULL    |       |
+-------------------+--------------+------+-----+---------+-------+
```

- **Base Nova :** Contient toutes les informations des VMs, comme leurs informations réseaux, leurs caractéristiques, et également la liste des computes nodes à disposition : Comme par exemple la table « Computes_Nodes » :

```
MariaDB []> use nova;
MariaDB [nova]> describe compute_nodes ;
+----------------------+--------------+------+-----+---------+----------------+
| Field                | Type         | Null | Key | Default | Extra          |
+----------------------+--------------+------+-----+---------+----------------+
| created_at           | datetime     | YES  |     | NULL    |                |
| updated_at           | datetime     | YES  |     | NULL    |                |
| deleted_at           | datetime     | YES  |     | NULL    |                |
| id                   | int(11)      | NO   | PRI | NULL    | auto_increment |
| service_id           | int(11)      | NO   | MUL | NULL    |                |
| vcpus                | int(11)      | NO   |     | NULL    |                |
| memory_mb            | int(11)      | NO   |     | NULL    |                |
| local_gb             | int(11)      | NO   |     | NULL    |                |
| vcpus_used           | int(11)      | NO   |     | NULL    |                |
| memory_mb_used       | int(11)      | NO   |     | NULL    |                |
| local_gb_used        | int(11)      | NO   |     | NULL    |                |
| hypervisor_type      | mediumtext   | NO   |     | NULL    |                |
| hypervisor_version   | int(11)      | NO   |     | NULL    |                |
| cpu_info             | mediumtext   | NO   |     | NULL    |                |
| disk_available_least | int(11)      | YES  |     | NULL    |                |
| free_ram_mb          | int(11)      | YES  |     | NULL    |                |
| free_disk_gb         | int(11)      | YES  |     | NULL    |                |
| current_workload     | int(11)      | YES  |     | NULL    |                |
| running_vms          | int(11)      | YES  |     | NULL    |                |
| hypervisor_hostname  | varchar(255) | YES  |     | NULL    |                |
| deleted              | int(11)      | YES  |     | NULL    |                |
| host_ip              | varchar(39)  | YES  |     | NULL    |                |
+----------------------+--------------+------+-----+---------+----------------+
```

[7] MariaDB : http://en.wikipedia.org/wiki/MariaDB

4.1.5 Profil d'une VM

Dans ce chapitre, nous allons étudier sur les différents éléments qui caractérisent une VM. Pour créer une **nouvelle VM** (ou instance chez Openstack), nous devons renseigner le Compute Node sur les caractéristiques techniques Hardware qu'elle possédera comme :

- Le nombre de VCPU
- La taille des disques durs
- La quantité de RAM virtuelle

Ces caractéristiques sont regroupées par profils appelés « Flavors[8] ». Ce système de profil simplifie grandement la tâche pour l'administrateur, du fait que sa VM aura un profil Hardware prédéfini. Voici une liste des Flavors disponible par défaut, mais que l'on peut éditer sans aucun problème :

ID	Name	Memory_MB	Disk	Ephemeral	VCPUs
1	m1.tiny	512	0	0	1
2	m1.small	2048	10	20	1
3	m1.medium	4096	10	40	2
4	m1.large	8192	10	80	4

En ce qui concerne la distinction entre **Disk** et **Ephemeral:**
Une VM peut avoir deux types de disques virtuels. Ces deux types de disques **suivent le cycle de vie**, ils seront donc supprimés au moment où l'on supprime la VM. Les disques virtuels au niveau de l'hyperviseur correspondent à un fichier. Le schéma 9, illustre les deux disques que la VM a à disposition.

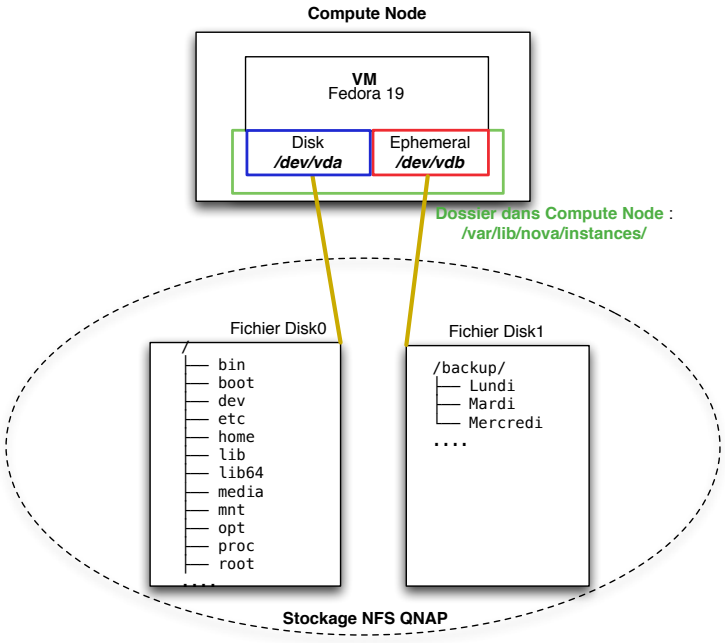


Schéma 9: Disques virtuels d'une VM

Les deux disques virtuels sous formes de fichiers sont stockés dans le dossier ***/var/lib/nova/instances/*** du Compute Node, mais ce dossier pointe sur le serveur NFS du QNAP via le réseau de Storage Jaune :

[8] Flavors : http://docs.openstack.org/folsom/openstack-ops/content/flavors.html

- **Disk (Bleu)**: Héberge la partition Root / de l'OS
- **Ephemeral Disc (Rouge)**: Peut être une deuxième partition pour la VM

Au moment où l'on fait un **`fdisk -l`** dans la VM, voici les disques que nous voyons :
`Disque /dev/vda : 75.2 Go -> Root Disc`
`Disque /dev/vdb : 21.5 Go -> Ephemeral Disc`
Ephemeral est en réalité l'ajout d'une deuxième partition à la VM.
Quand nous regardons le fichier XML généré par Libvirt afin de créer la VM :

```
<disk type="file" device="disk">
      <driver name="qemu" type="qcow2" cache="none"/>
      <source file="/var/lib/nova/instances/84673e27-e948-49c0-a44f-a842aac7684b/disk"/>
      <target bus="virtio" dev="vda"/>
</disk>
<disk type="file" device="disk">
      <driver name="qemu" type="qcow2" cache="none"/>
      <source file="/var/lib/nova/instances/84673e27-e948-49c0-a44f-a842aac7684b/disk.local"/>
      <target bus="virtio" dev="vdb"/>
</disk>
```

Nous voyons bien la définition des deux disques, le premier est le **Disk** et le deuxième est le **Ephemeral** d'après le **Flavor**.
Dans mon cas **l'ajout d'un deuxième disque (Ephemeral), n'est pas d'une grande utilité**, sachant qu'il va être détruit au moment de la suppression de la VM. Voilà pourquoi, j'ai mis en place une unité **Cinder** dans le **§4.2**.

4.1.6 Processus de démarrage d'une VM

Le schéma 10 résume les **communications entre les différents services** d'OpenStack :

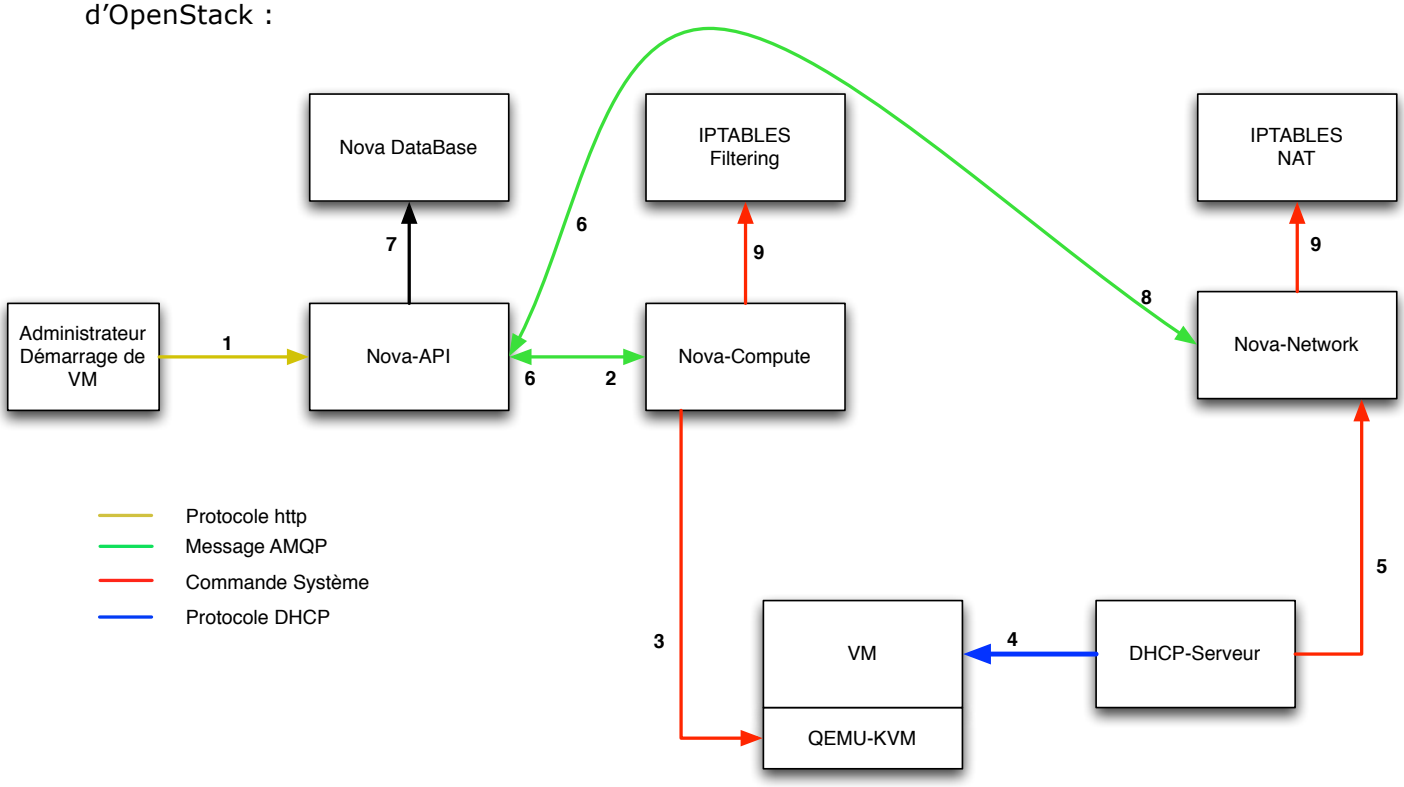


Schéma 10: Démarrage d'une VMs

1. L'administrateur via l'interface Web ou une commande, indique le Flavor de la VM, la liste des Security Group. Tous ces paramètres sont envoyés à l'API Nova via une requête http. Par souci de lisibilité je n'ai pas représenté le contrôle d'autorisation avec Keystone (voir §4.1.4.3).

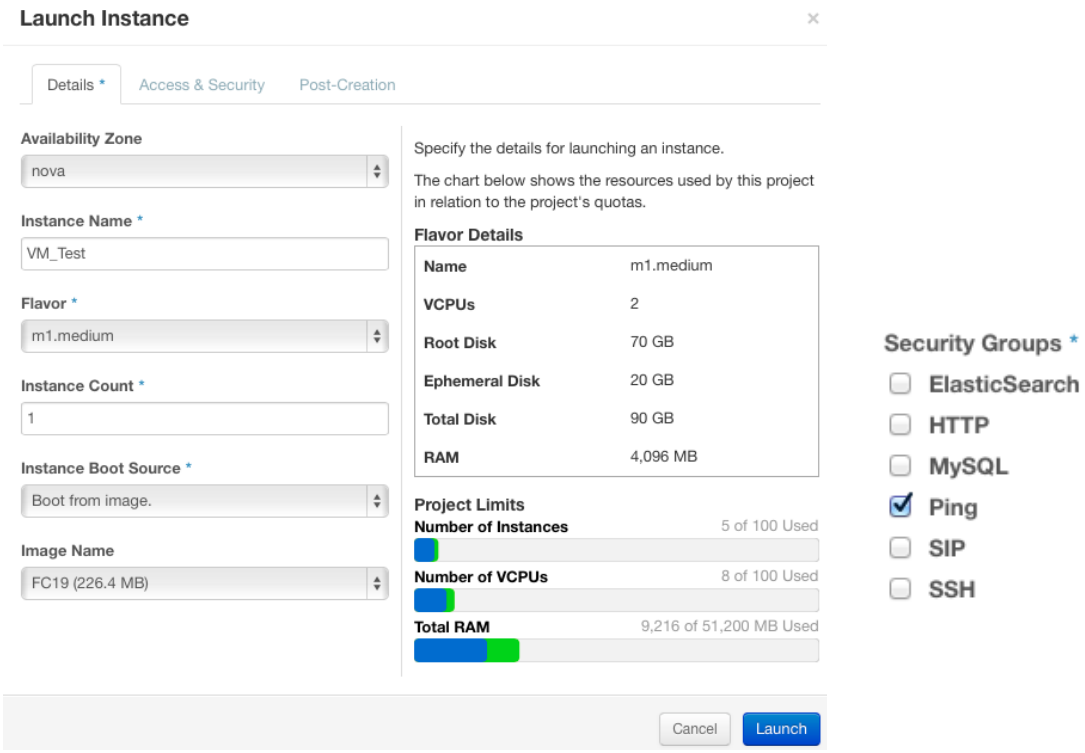


Figure 2: Configuration de la future VM

2. Tous les messages AMQP doivent transiter par un QPID serveur que je n'ai pas représenté, également par souci de lisibilité. Nova-API envoie au service Nova-Compute le profil de la VM qui va être créé
   Extrait des Logs AMQP de Nova-API :

```
qpid.messaging.io.ops [-] SENT[6b9bc68]: "created_at": "2013-10-28T16:43:47.000000",
"disk_format": "qcow2 »,{« instance_type": {"memory_mb": 4096, "root_gb": 70,
"name": "m1.medium", "ephemeral_gb": 20, "vcpus": 2, "extra_specs": {}, "swap": 0,
"flavorid": "3", "id": 10},"security_groups": {"objects": [{"user_id":
"f91a33812d1e42bcb85eb2ac19b7b96b", "description": "Allow Ping"}],"hostname": "vm-
test"
```

3. Nova-Compute envoie à KVM, les caractéristiques reçues par message AMPQ, la VM démarre. Le service Nova envoie le profil complet de la VM à Libvirt (Extrait du fichier de log ne Nova Compute:

```
nova.virt.libvirt.config [req-e9d8349c-bc48-4628-850e-cef552119866
f91a33812d1e42bcb85eb2ac19b7b96b be9797b25e4648fab5d4c763b36e5fa3] Generated XML <domain
type="kvm">
<uuid>cf62e224-cd99-4097-a728-b082dd8d9e19</uuid>
<name>instance-000001e8</name>
<memory>4194304</memory>
<vcpu>2</vcpu>
```

4. Le serveur DHCP envoie une IP à la nouvelle VM, et écrit dans un fichier la MAC-IP ***/var/lib/nova/networks/nova-br100.conf*** :
   `fa:16:3e:20:1b:65,vm-test.novalocal,192.168.1.8`

5. Le service Nova-Network lit ce fichier

6. Nova-Network envoie les informations réseaux à Nova-API

7. Nova-API écrit dans la base de données les informations réseaux ainsi que les caractéristiques de la nouvelle VM. Extrait de la table « **instances** » de la DB Nova :

Field	Type	Null	Key	Default	Extra
created_at	datetime	YES		NULL	
updated_at	datetime	YES		NULL	
deleted_at	datetime	YES		NULL	
id	int(11)	NO	PRI	NULL	auto_increment
vm_state	varchar(255)	YES		NULL	
memory_mb	int(11)	YES		NULL	
vcpus	int(11)	YES		NULL	
hostname	varchar(255)	YES		NULL	
host	varchar(255)	YES	MUL	NULL	
root_gb	int(11)	YES		NULL	
ephemeral_gb	int(11)	YES		NULL	
node	varchar(255)	YES		NULL	

8. Les informations de Floating IP sont envoyées au Nova-Network et les security-group au Nova-Compute

9. Les deux services vont configurer les règles NAT et Filtering des Iptables.

4.1.7 Réseau des VMs

Dans ce chapitre nous allons étudier en détail la **gestion du Floating IP**, ainsi que les Security Group, au niveau IPTABLES et interfaces physiques, en faisant abstraction des services Openstack.

- **Le Floating IP** :

Le Floating IP est utilisé pour attribuer une IP public à une VM. Dans notre exemple, la VM va posséder comme IP public : 129.194.184.85. Le principe est de rajouter une deuxième adresse IP sur une carte Ethernet physique. Dans notre cas, la carte Eth2 possède l'adresse IP : 129.194.184.91 et grâce au **IP Alias nous ajoutons une seconde adresse IP** : 129.194.184.85. Il est possible de rajouter N IPs flottantes sur cette interface physique, correspondant aux IPs Publics de N VMs.

IP Alias est une fonction intégrée dans le kernel linux. Pour afficher la configuration des IP Flottantes sur la carte réseau eth2, nous ne pouvons plus utiliser la commande **ifconfig devenue obsolète**. Il faut utiliser les outils plus élaborés comme le package **iproute2**, avec la commande **ip** :

```
[root@network ~]# ip addr show
4: eth2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen
1000
    link/ether 68:05:ca:11:b5:09 brd ff:ff:ff:ff:ff:ff
    inet 129.194.184.91/22 brd 129.194.187.255 scope global eth2
       valid_lft forever preferred_lft forever
    inet 129.194.184.85/32 scope global eth2
       valid_lft forever preferred_lft forever
    inet 129.194.185.45/32 scope global eth2
```

- **Trajet (Vert) d'un paquet http venant du réseau public**

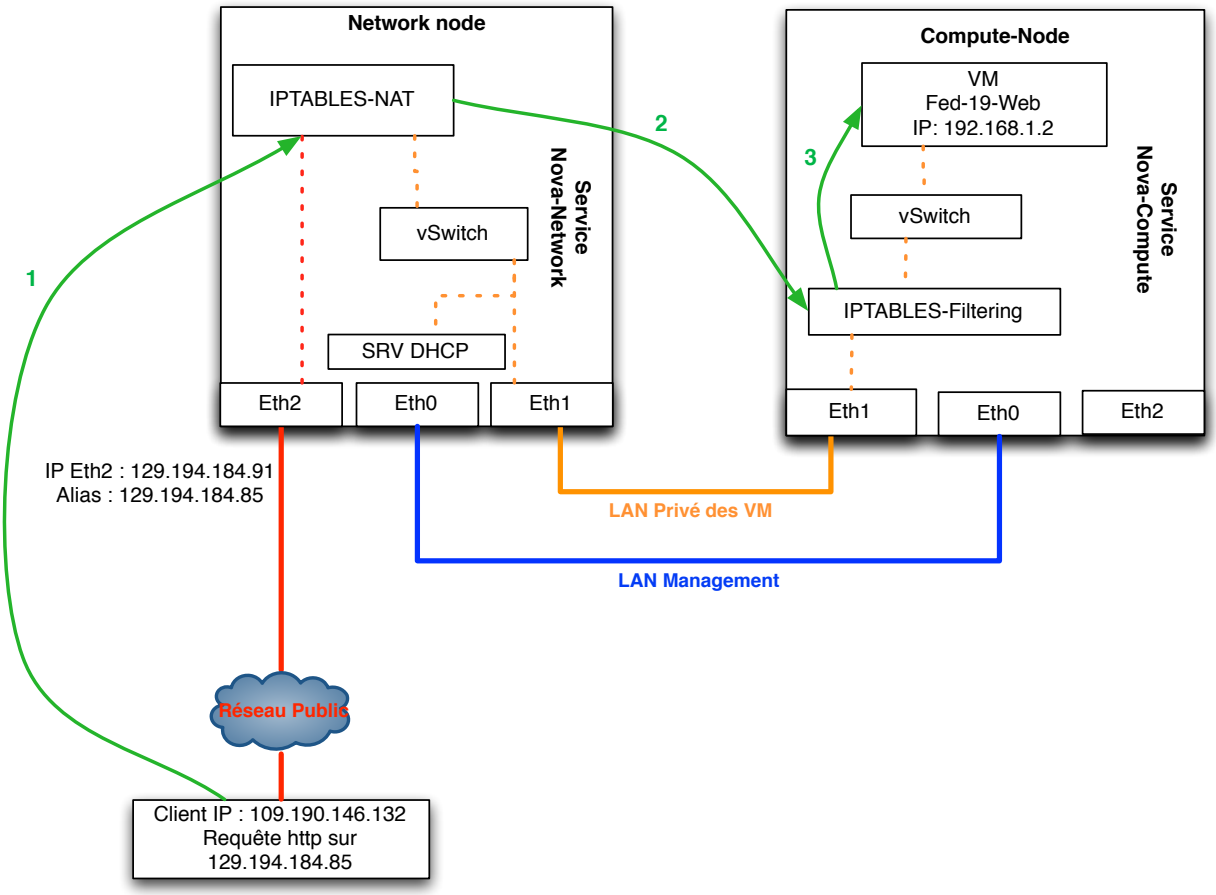


Schéma 11: Gestion de l'IP flottante

Nous supposons que les Security-Group (voir §A.1 Démarrage d'une VM) et le Floating IP ont déjà été configurés par les services Nova-Network et Nova-Compute. Je vais analyser le contenu des IPTABLES-NAT et IPTABLES-Filtering. La règle de Security-group est d'autoriser uniquement le trafic IP à destination du port 80.

1. Le client envoie une requête http au serveur Web possédant l'adresse IP 129.194.184.85. Le paquet traverse l'interface Eth2 ayant l'allias 129.194.184.85. Pour afficher le contenu des tables NAT d'Iptables, nous faisons :
   `iptables -S -t nat`
   Voici un extrait de la table NAT de l'IPTABLES :
   `nova-network-PREROUTING -d 129.194.184.85/32 -j DNAT --to-destination 192.168.1.2`

Les paquets ayant comme IP destination 129.194.184.5 sont redirigés vers l'adresse IP 192.168.1.2. IPTABLES va modifier l'adresse IP de destination par 192.168.1.2. Nous pouvons le voir en faisant un **`TCPDUMP -i eth1`** :

```
109.190.146.132.50875 > 192.168.1.2.http
```

2. Le paquet sort du Network-Node via l'interface Eth1 et rentre dans le Compute Node, mais avant d'arriver à la VM le paquet va être filtré par IPTABLES-Filtering qui a été modifié à partir des Security-Group :

   ```
   nova-compute-inst-9 -p tcp -m tcp --dport 80 -j ACCEPT
   ```

   Les paquets à destination du port 80 de la VM (nova-compute-inst-9) sont autorisés.
   Il faut voir ces règles de Security comme des règles de **Firewalls de types statefulls**. Nous contrôlons les paquets qui viennent du réseau Public en direction de la VM. Ces règles vont être traduites en règles IPtables. Voici une capture d'écran concernant les règles IPtables de l'interface Horizon pour autoriser l'ouverture du port 80:

Security Group Rules

	Direction	Ether Type	IP Protocol	Port Range	Remote
☐	Ingress	-	TCP	80	0.0.0.0/0 (CIDR)

Displaying 1 item

Figure 3: Security-Group sous Horizon

3. En faisant un TCPDUMP dans la VM nous voyons le paquet possédant l'adresse source du client :

   ```
   109.190.146.132.52353 > 192.168.1.2.http
   ```

En conclusion nous voyons que toutes les **règles de filtrage et de NAT doivent être dynamiques** car la VM ou le Floating peuvent changer d'adresse IP, il est primordial que les différents services Nova-Network et Nova-Compute fassent remonter les informations et puissent être contrôlés depuis un système centralisé : Nova-api.

4.1.8 Stockage par fichier des VMs

Plusieurs architectures de stockage s'offrent à nous. Le **stockage des disques virtuels des VMs** doit être **externe à l'hyperviseur**, afin d'avoir une unité de stockage centralisée, pour pouvoir effectuer de la live-migration et faciliter la gestion des systèmes de stockages. Mais l'OS de l'hyperviseur physique est lui stocké sur son propre disque dur. Le fait d'avoir un réseau dédié (jaune) a l'avantage de pouvoir augmenter les débits grâce à l'activation par exemple des Jumbo Frames.

Cette **architecture est moins performante qu'un stockage par bloc**, car le système de fichier est hébergé du côté du serveur de stockage. Cependant, grâce au NFS, nous pouvons avoir un système de fichier partagé simple. Comme représenté sur le schéma global en §4.1, le stockage du disque virtuel de la VM est hébergé sur le QNAP. Il serait

judicieux d'ajouter un second système de stockage, mais cette fois-ci, il devra être par bloc et être persistant. En effet, si la VM est supprimée l'unité de stockage reste. L'étude de l'unité Cinder se fera au **§4.2**

4.1.9 Conclusion du Scénario 1a

A travers ce scénario, j'ai pu **comprendre les interactions entres différents services** qui évoluent au sein de mon infrastructure. J'ai passé beaucoup de temps dans la compréhension des interactions entres ces différents éléments. En effet, la documentation OpenStack au niveau de la configuration des services est relativement bien étoffée. Cependant le rôle que remplit chaque service est beaucoup moins bien documenté, c'est pour cela que j'ai dû analyser de nombreuses lignes de Logs, aller voir dans le code-source des différents services.
L'infrastructure Openstack peut être qualifiée de complexe, de par ces nombreux systèmes qui s'imbriquent les uns dans les autres. Les différents services doivent en permanence communiquer entres eux, car nous sommes dans un milieu totalement dynamique.

4.2 Scénario 1b : Gestion du stockage par bloc

A travers ce scénario, je vais explorer les **stockages par bloc** proposés par le Projet Cinder[9]. **Cinder** est un service OpenStack permettant de créer des **volumes par bloc gérés par le protocole iSCSI**. Cette unité de stockage n'est plus créée par l'ordonnanceur au démarrage de la VM, mais par l'administrateur, donc il ne va plus suivre le cycle de vie de la VM, du fait que lorsqu'elle sera détruite, ce volume ne sera pas supprimé. L'avantage d'un volume comme celui-ci permet d'héberger des données persistantes. Par exemple ce volume pourrait être monté au niveau du dossier */var/www* (emplacement contenant généralement les pages web sur un serveur web), si la VM est hors-service, on pourra en démarrer une autre et lui ajouter ce volume à cette nouvelle instance.

4.2.1 Schéma du scénario 1b

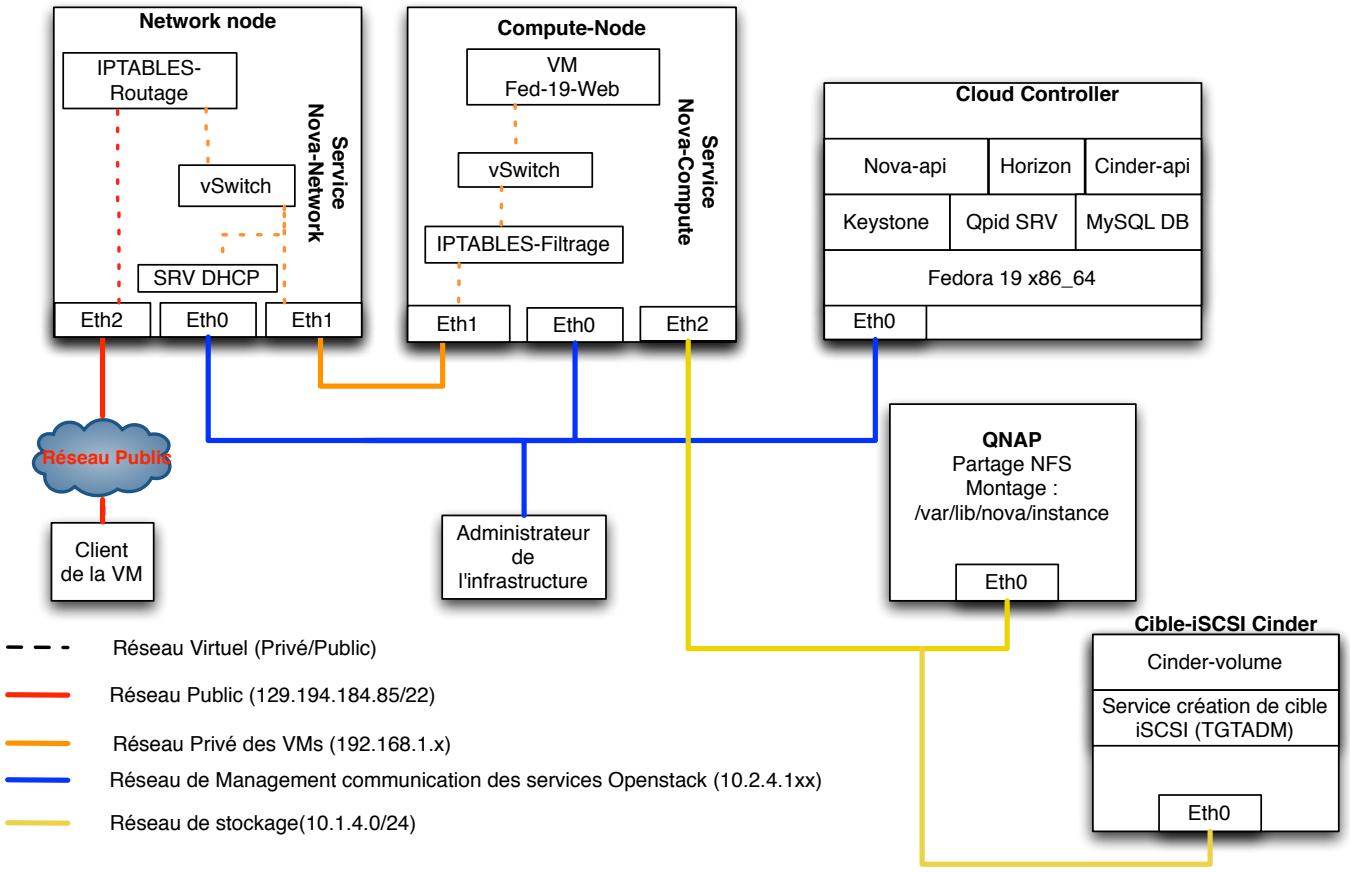


Schéma 12: Scénario 1b

4.2.2 Composition de Cinder

Sur le schéma 12, j'ai ajouté un élément physique **Cible-iSCSI-Cinder**. Cet élément va être piloté par le service placé dans le Cloud Controller : Cinder-api.

Cinder est composé de deux éléments :

- **Cinder API**: Va recevoir les commandes envoyées par l'utilisateur via une requête REST contenant par exemple la taille du volume à créer et sur quelle VM il faut greffer le volume.

[9] Cinder : https://wiki.openstack.org/wiki/Cinder

- **Cinder Volume** : Ce service va communiquer directement avec l'unité de stockage. Plusieurs possibilités s'offres nous, cette communication va se faire via l'utilisation de différents drivers:
    - Piloter la création de cible iSCSI situé dans un SAN propriétaire comme SolidFire[10]
    - Un Cinder-Volume qui contient un serveur iSCSI et un volume LVM. L'API Cinder va piloter la création de la cible iSCSI afin de l'attacher à la VM, en fournissant à Libvirt l'IQN de la cible iSCSI[11]

4.2.3 Fonctionnement de Cinder

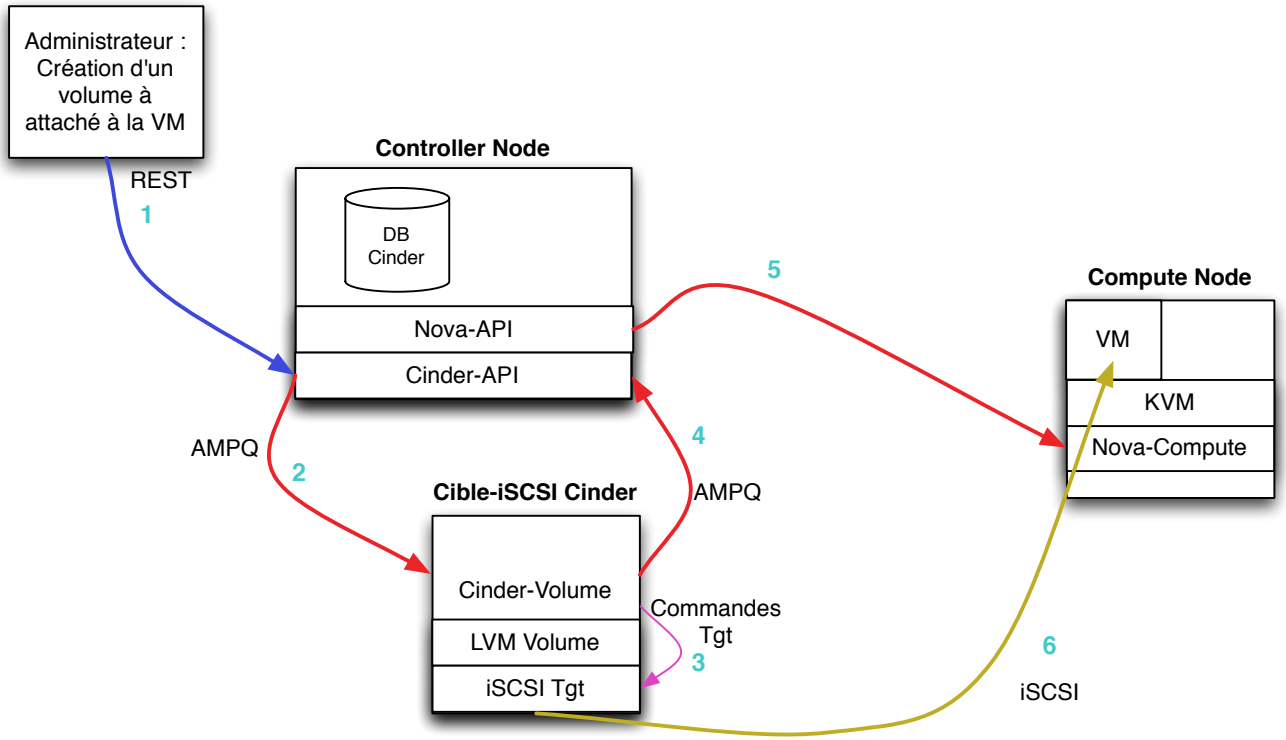


Schéma 13: Attribution d'un volume par bloc

Cinder-Volumes communique avec le serveur iSCSI pour créer une nouvelle cible pour la VM en question.
La chaîne de lancement pour créer une unité de stockage par bloc est décrite par les numéros turquoise dans le schéma 13 :

1. **L'administrateur** envoie une **commande** de **création** d'un **volume** via l'api Cinder, via une requête http, contenant la taille du volume et sur quelle VM le greffer ou par CLI :
   `cinder create --display_name Test_Vol 3GB`

2. **Cinder-API** va envoyer un **message** AMPQ à l'unité de stockage (**Cible-iSCSI-Cinder**

[10] SolidFire : http://solidfire.com/technology/solidfire-storage-system/
[11] ISCSI TOPO : http://www.rackspace.com/knowledge_center/article/building-a-rackspace-private-cloud-with-linux-iscsi-volumes

3. Le **service Cinder-Volume communique** avec le manager de **cible iSCSI** intégré à Fedora « **tgtadm** ». Extrait du code source de Cinder indiquant l'appelle au service « tgadm »[12] :

```
time.sleep(10)
try:
    (out, err) = self._execute('tgtadm', '--lld', 'iscsi',
                               '--op', 'new', '--mode',
                               'logicalunit', '--tid',
                               tid, '--lun', '1', '-b',
                               path, run_as_root=True)
    LOG.debug('StdOut from recreate backing lun: %s' % out)
```

4. Une fois la **cible iSCSI créée**, le service Cinder-volume envoie à Cinder-API les **informations** de la cible iSCSI qui va les relayer à Nova-API, et va créer une nouvelle entrée pour ce volume dans la **base de données** MySQL

5. **Nova-API** envoie l'ordre d'ajout d'un nouveau volume **à Nova-Compute**, qui va transférer ces informations à KVM. Voici un exemple du template généré par Libvirt pour décrire la configuration du nouveau volume iSCSI:

```
<source dev=
"/dev/disk/by-path/ip-10.1.4.4:3260-iscsi-iqn.2010-10.org.openstack:volume-
a0822149-482c-49b0-abd7-fb7de07d4a96-lun-1"/>
```

6. La **VM** possède un **nouveau volume** monté en **/dev/vdc** :

```
Disk /dev/vdc: 3221 MB, 3221225472 bytes, 6291456 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
```

4.2.4 Problèmes rencontrés

J'ai eu quelques problèmes en ce qui concerne l'analyse de Cinder-API et Cinder-Volume, du fait que la plupart du temps, Cinder Volume est couplé avec un SAN propriétaire. La documentation indiquant comment faire fonctionner Cinder-Volume avec le gestionnaire de cible iSCI (tgtadm) intégré à Linux n'est pas très riche. Un des gros problèmes avec la documentation OpenStack, est d'interpréter **sur quelle unité physique tels ou tels services s'installent**. Dans la théorie, chaque service peut s'installer sur n'importe quelle entité physique. Mais dans le cas du Cinder-volume qui pilote la cible iSCSI, il faut qu'il soit installé sur le même serveur physique que le LVM Volume, car le service python fait des appels de commandes en local.

4.2.5 Conclusion du Scénario 1b

A travers ce scénario j'ai pu ajouter un volume persistant à une VM. De plus, ce volume est un stockage par bloc interrogeable via le protocole iSCSI. Comme application, nous pourrions héberger une base de données MySQL, afin d'avoir des performances plus élevées que si nous avions uniquement un stockage par fichier NFS. L'unité Cinder possède un gros avantage de flexibilité, grâce à l'ajout dynamique d'unités de volume.

---

[12] Code source de Cinder :
https://github.com/openstack/cinder/blob/master/cinder/brick/iscsi/iscsi.py

5 Scénario 2 : Haute Disponibilité du Cloud Controller

Au cours des scénarios précédents, nous avons obtenu une infrastructure fonctionnelle, accueillant des VMs pouvant héberger nos applications métiers. Cependant, il réside un gros point faible dans notre infrastructure, concernant un de nos composants de gestion. En effet, si le **Cloud Controller venait à être hors-service**, il nous serait alors **impossible de gérer nos VMs** ou d'en créer de nouvelles. Par conséquent, nous devons repenser l'infrastructure afin de mettre en place des systèmes redondants pour limiter au maximum des situations d'échec. Ceci pourra être mis en place grâce à des mécanismes permettant de **construire un environnement de Haute-Disponibilité** (High Availability -> HA).
De nombreuses solutions de HA existent, mais parfois, elles peuvent vite représenter une lourdeur supplémentaire, et par conséquent compliquer grandement notre système en risquant de le rendre instable. A travers ce chapitre, je vais faire **cohabiter plusieurs** types de **HA**, qui auront toutes un **rôle précis** pour chaque élément que je dois rendre disponible.

5.1 Qu'est ce que la HA

La HA peut être appliquée sur une multitude de services afin de garantir un service opérationnel. Comme le montre le schéma de principe 14, il faudrait par exemple que **l'administrateur** puisse **toujours démarrer** des **VMs**, donc grâce à la HA, nous allons lutter contre les indisponibilités, en installant un second Cloud Controller physique (voir schéma 17 du Scénario 2 pour un contexte réel) afin de dupliquer les services.

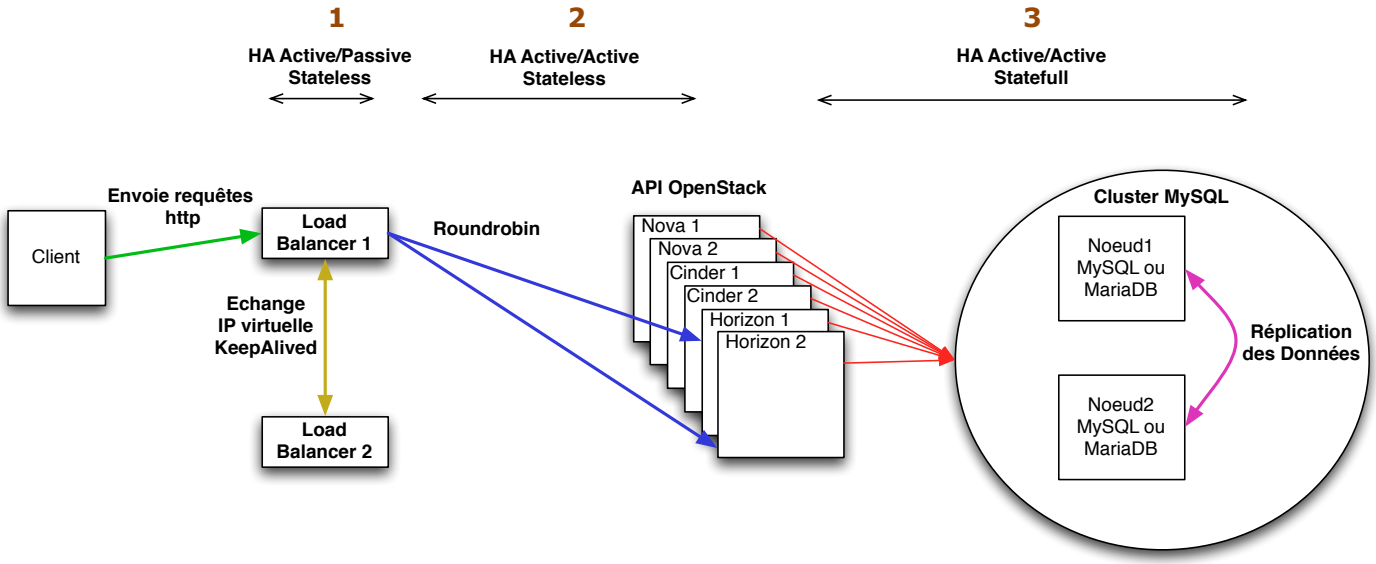


Schéma 14: Principe de HA

Le schéma 14, montre les **différents types de mécanismes de HA** appliqués à chaque élément de mon infrastructure. Dans les points suivants, je vais m'intéresser uniquement aux éléments 1 et 2, la HA SQL (élément 3) sera traitée au §5.3.

5.1.1 La HA Active/Active

Ce type de HA (Elément 2 Schéma 14) oblige la mise en place de deux serveurs physiques **Master + Master** (Cloud Controller 1 et 2) ayant les **mêmes services installés et démarrés**. Par conséquent les deux serveurs sont en permanence actifs, ce qui limite les temps d'indisponibilité au moment de la mise en échec d'un des deux. Comme les **deux serveurs peuvent répondre aux requêtes**, j'ai dû installer un **load balancer** qui se charge de **répartir les charges** entre mes deux serveurs. C'est-à-dire que le client s'adresse au Load Balancer (Flèche Verte) et ce dernier va aiguiller les paquets vers les serveurs disponibles (Flèches Bleues). Ce service de load balancing sera géré par **HAProxy** (voir §5.2.4 pour le fonctionnement de HAProxy). Comme tout le **trafic traverse le load balancer**, il présente un « **point of failure** », par conséquent, nous devons intégrer la HA pour cet élément, mais qui ne pourra pas être traité avec de la HA Active/Active.

5.1.2 La HA Active/Passive

Ce type de HA met en place deux serveurs : **Master + Salve**. Sur ces deux serveurs sont installés les mêmes services, mais sur le MASTER ils sont démarrés, tandis que sur le SLAVE ils sont stoppés. Voici un schéma représentant le système :

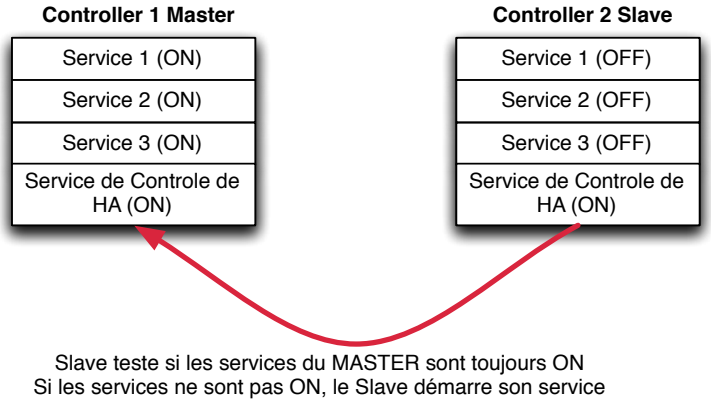


Schéma 15: HA Active/Passive

Si **un des services ne répond plus**, l'autre **service** va **démarrer** sur le **SLAVE**. Le problème de ce type de HA, est que le SLAVE a ces services OFF, donc nous pouvons avoir un **temps d'indisponibilité important** entre le moment où le service commence son démarrage et devient opérationnel. En effet, le script de démarrage du service peut effectuer de nombreuses tâches**,** comme par exemple, se connecter à une base MySQL et attendre la réponse du serveur QPID. Ce type de chaîne de lancement au sein des services Openstack est couramment utilisé.

J'ai implémenté ce type de **HA uniquement au niveau du load balancing** (Elément 1). En effet, l'intégration d'un deuxième Load Balancer considéré comme Slave est primordial mais **seul l'un des deux sera actif**. Dès que le Master ne répondra plus, le Slave prendra le relai. Le client s'adresse à l'un des loads balancers via une adresse IP Virtuelle, qui sera gérée par un service choisissant quel load balancer se verra attribuer cette IP.

5.1.3 L'IP flottante ou virtuelle

Pour avoir un système basé sur de la HA Active/Passive avec deux load balancers, l'**un des deux doit pouvoir répondre aux requêtes**. Pour cela nous allons utiliser le principe d'**IP flottante** comme vu dans §4.1.7. Ce service géré par « Keepalived[13] », va attribuer une adresse IP supplémentaire (virtuelle) sur le Serveur actif. L'utilitaire installé sur les deux serveurs va tester si un service est ON, s'il ne l'est pas, l'IP flottante passe du Master au Slave.

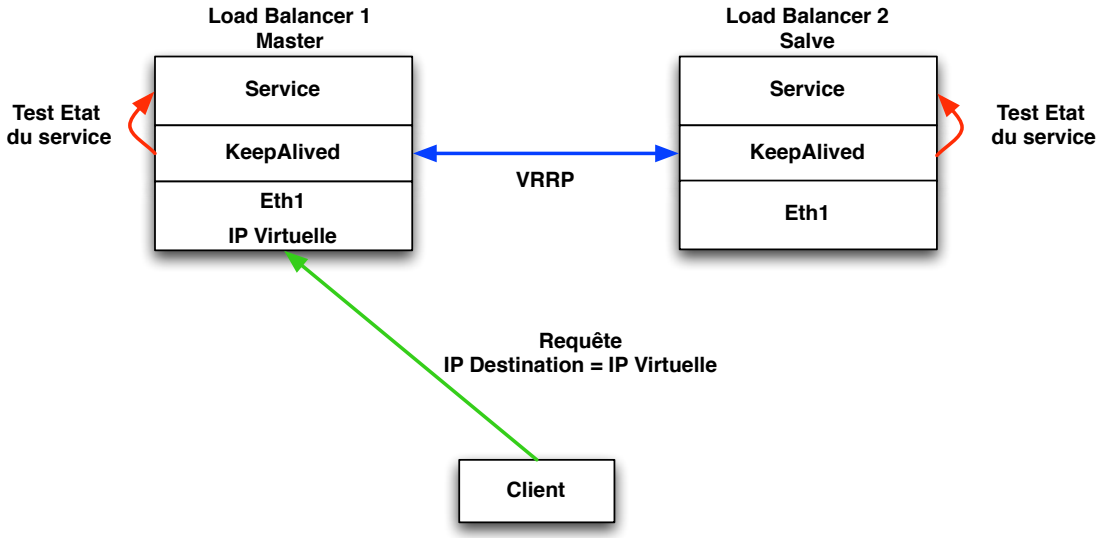


Schéma 16: Gestion IP virtuelle

D'après le schéma 16, grâce à cette technique le **client aura toujours la même IP de destination** mais ne s'adressera pas toujours au même Load Balancer. Les deux services Keepalived communiquent entres eux via le protocole VRRP[14] (Virtual Router Redundancy Protocol), pour que le Master transmette au Slave l'échec de son service. Ce protocole permet l'élection d'une passerelle possédant l'IP flottante, afin qu'il puisse répondre aux requêtes ARP (voir §5.2.5 pour un cas concret).

5.1.4 Services Stateless ou Statefull

Les services ne réagissent pas tous de la même manière au moment où ils sont dupliqués. C'est pour cela que nous devons gérer la haute disponibilité différemment si nous sommes en présence de services Stateless ou Statefull.

- **Services Stateless :**

Un service n'a **pas besoin de connaître son état précédent**, donc les services dupliqués n'ont pas d'obligation à communiquer entre eux afin de se partager leurs états. Par exemple, les Services d'API OpenStack sont Stateless, ils sont indépendants les uns des autres.

- **Services Statefull :**

Un service dit Statefull est obligé de connaître son état précédent pour répondre à une requête. Donc au moment où nous implémentons de la HA Statefull, nous devons

[13] Keealived : http://www.k-tux.com/keepalived-entree-en-matiere
[14] VRRP : http://tools.ietf.org/html/rfc5798

rajouter un système de réplication, afin de propager la donnée au sein de tous les services redondants. Par exemple, une base de données est un service Statefull. La donnée que l'on a écrite de sur un nœud SQL doit être propagée dans tous les autres nœuds SQL.

Les notions Statefull et Stateless sont indépendantes de la HA Active/Active ou Active/Passive. En effet, nous pouvons très bien travailler en Active/Passive avec des services Statefull. Le service Master envoie son état au Slave qui est off, afin que ce dernier possède l'état du Master au moment où il démarrera.

5.2 Scénario 2 : La HA au sein des API OpenStack

Dans ce scénario, je vais implémenter la **HA** au sein de mon Cloud Controller sur **l'ensemble des API** qui composent mon Infrastructure OpenStack :

- Interface Web Horizon
- API Nova
- API Cinder
- API Keystone

J'ai choisi d'installer les **bases de données MySQL** (Keystone DB, Nova DB et Cinder DB) sur une **autre machine physique**, afin de ne m'attarder que sur la HA au niveau des requêtes http avec les API. En effet la HA MySQL doit être traitée avec une approche différente. Pour rappel, le contact d'une API se fait par l'envoi d'une requête http. L'API écoute sur un port particulier.
Le schéma 17 résume la cohésion entres les différents éléments physiques au moment où un administrateur envoie une requête sur l'interface web Horizon écoutant sur le port 80. Pour mettre en place ce scénario, je me suis basé sur la documentation fournie par RedHat [15] et Mirantis [16] (société fournissant des services dans l'implémentation d'OpenStack).

5.2.1 Schéma du scénario 2a

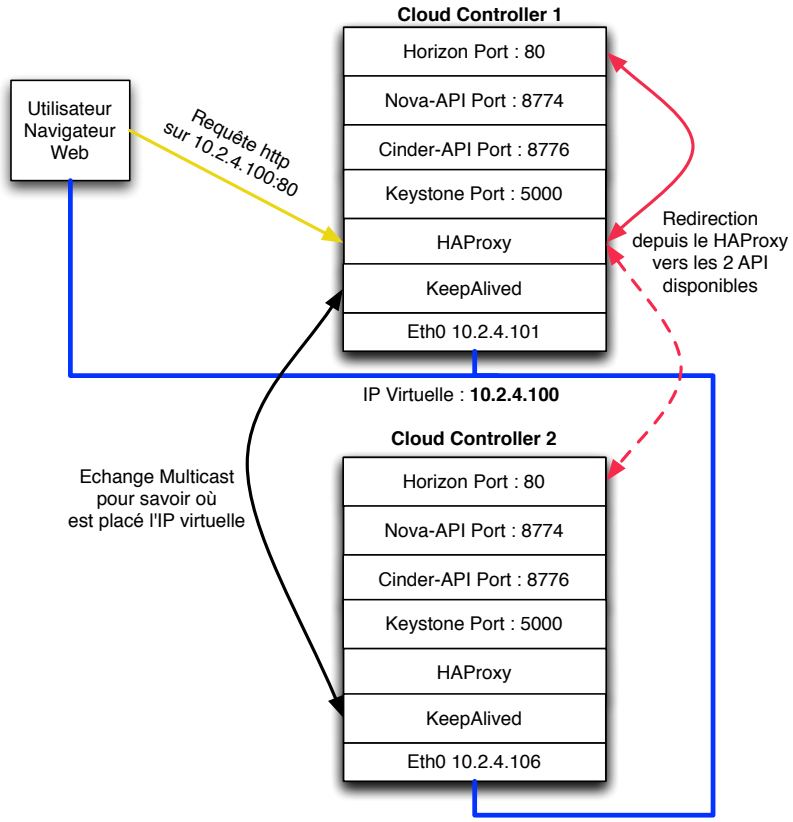
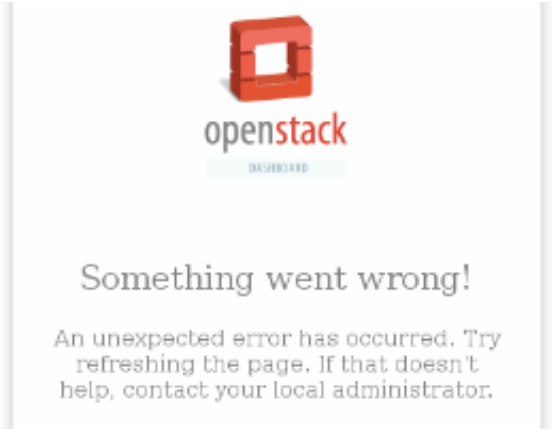


Schéma 17: HA au sein de l'API OpenStack

[15] HA Redhat : http://openstack.redhat.com/Load_Balance_OpenStack_API
[16] Mirantis HA : http://www.mirantis.com/blog/software-high-availability-load-balancing-openstack-cloud-api-service/

5.2.2 But du Scénario

Sans intégration de HA, avec l'infrastructure du Scénario 1, au moment où l'on stoppe un des services d'API OpenStack tel que Nova API nous obtenons ce message d'erreur :



A travers ce scénario, je vais **limiter l'apparition de ce message** en ajoutant un second Cloud Controller permettant de répondre également aux requêtes de l'administrateur et de prendre le relais en cas de défaillance du Controller 1. Lorsqu'un câble réseau du Controller 1 de l'interface Eth0 est débranché pour simuler une panne générale, il sera alors **toujours possible de gérer la création de VMs**. En réalité, le paquet a été aiguillé vers le Controller 2.

5.2.3 Eléments physiques :

Au niveau des machines physiques, j'ai quasiment la même infrastructure que dans les scénarios précédents, j'ai ajouté un deuxième Cloud Controller qui possède la même configuration que le premier. Pour avoir de plus amples détails à propos du paramétrage des différentes machines physiques, voici mon dépôt Github : https://github.com/hepia-telecom-labs/Openstack_Config/tree/master/Object_config/HA qui regroupe les fichiers de configuration utilisés dans mon infrastructure.

5.2.4 HA Active/Active avec HAProxy

Ce type de HA comme vu au §5.1.1 est assuré par un Load Balancer Open Source HAProxy. Je l'ai **installé sur les deux Cloud Controllers** afin de ne pas dédier deux machines physiques à cette tâche et de garantir la HA sur les Load Balancer au cas ou l'un d'eux serait non-opérationnel.

Voici une partie de la configuration de HAProxy afin de comprendre les interactions entre les IP des serveurs physiques et l'IP virtuelle pour une requête émise par l'administrateur au serveur web horizon :

```
listen horizon-web 10.2.4.100:80
  balance  roundrobin
  option  tcplog
  server  controller-1 10.2.4.101:80  check
  server  controller-2 10.2.4.106:80  check
```

La **configuration** du HAProxy doit être **identique sur les deux Controllers**. Le Proxy écoute sur le port 80 au niveau de l'IP virtuelle (10.2.4.100) et va aiguiller les paquets sur les serveurs situés en backend (**controller-1** et **controller-2**). La technique d'aiguillage est basée sur le **RoundRobin**, c'est-à-dire que la charge va être répartie équitablement sur les deux serveurs web des deux Controllers. Il va également tester le serveur cible à qui il choisit d'envoyer le paquet, en faisant par exemple une requête http sur le port d'une API. Si l'API du controller-1 ne répond pas, il va aiguiller le paquet sur controller-2.

5.2.5 HA Load Balancer avec KeepAlived

KeepAlived va assurer la **disponibilité des Load balancers**, en étant installé sur les deux Controllers. Voici un extrait des deux fichiers de configurations des deux services KieepAlived :

Controller 1

```
vrrp_script Test_Service_Alive{
  script "killall -0 HAProxy" #Test the process Alive
  interval 0.1
}
vrrp_instance Controller-1{
  virtual_router_id 42 #Id of communication
                       #between 2 keepAlived
  priority  101        #For electing MASTER
  state     MASTER     #Initial State
  interface eth1
  virtual_ipaddress {
        10.2.4.100
}
  track_script {
       Test_Service_Alive
  }
```

Controller 2

```
vrrp_script Test_Service_Alive{
  script "killall -0 HAProxy"
  interval 0.1
}
vrrp_instance Controller-2{
  virtual_router_id 42

  priority  100
  state     BACKUP
  interface eth1
  virtual_ipaddress {
        10.2.4.100
}
  track_script {
       Test_Service_Alive
  }
```

Les deux KeepAlived communiquent entre eux par VRRP, ils possèdent le même « `virtual_router_id` » que l'on peut apparenter à leur fréquence de communication. **L'attribution de l'adresse IP Virtuelle est basée sur un système d'élection** entre les deux KeepAlived. Comme nous le voyons dans les fichiers de configuration, j'ai arbitrairement choisi que le Controller 1 dans un fonctionnement normal serait MASTER, de par sa priorité qui est de 101 et de son état initial (`state`) MASTER. Au moment où deux KeepAlived démarrent en même temps, ils procéderont à l'élection du Master : celui qui a la plus haute priorité possède l'IP virtuelle.
Le service KeepAlived va tester toutes les 0.1 secondes l'état du service, dans le cas où il est STOP le Master passe en Backup et transmet l'IP Virtuelle au deuxième KeepALived.

Dans mon infrastructure de HA, la relation **Active/Passive**, est **uniquement présente** dans la **gestion de l'IP Virtuelle** administrée par KeepAlived, car il n'y a qu'un seul Load Balancer qui peut répondre à la fois. Autrement, passé les Load Balancers, tous les services OpenStack font de la HA Actif/Actif.

5.2.6 Bilan Scénario 2a :

Ce scénario m'a permis de comprendre comment fonctionne la HA. De plus, un certain nombre d'éléments interviennent dans cette topologie et il est donc important de comprendre comment ils interagissent entres eux.

A la fin de ce scénario, au moment où je débranche le câble réseau de mon Controller1, je peux sans autre continuer à naviguer sur mon interface web, et gérer mes VMs sans pertes de connexions.
Pour visualiser l'échange d'IP Virtuelle nous pouvons analyser les Log, en faisant :
`tail -f /var/log/messages | grep vrrp*`

- Controller1 :

```
controller Keepalived_vrrp[1618]: VRRP_Script(Test_Service_HAProxy) failed
Nov 28 10:25:02 controller Keepalived_vrrp[1618]: VRRP_Instance(42) Received higher prio
advert
Nov 28 10:25:02 controller Keepalived_vrrp[1618]: VRRP_Instance(42) Entering BACKUP STATE
Nov 28 10:25:02 controller Keepalived_vrrp[1618]: VRRP_Instance(42) removing protocol VIPs.
```

- Controller2 :

```
Nov 28 10:25:02 controller2 Keepalived_vrrp[894]: VRRP_Instance(42) forcing a new MASTER
election
Nov 28 10:25:03 controller2 Keepalived_vrrp[894]: VRRP_Instance(42) Transitionto MASTER STATE
Nov 28 10:25:04 controller2 Keepalived_vrrp[894]: VRRP_Instance(42) Entering MASTER STATE
Nov 28 10:25:04 controller2 Keepalived_vrrp[894]: VRRP_Instance(42) setting protocol VIPs.
Nov 28 10:25:04 controller2 Keepalived_vrrp[894]: VRRP_Instance(42) Sending gratuitous ARPs
on eth0 for 10.2.4.100
```

A travers ces extraits de Log nous voyons bien que la **transition** de l'IP Virtuelle se fait très rapidement, de **l'ordre de quelques secondes**, ce qui explique que pour l'administrateur, le fait de perdre un des deux Controller est totalement transparent. Il ne va détecter aucune interruption de service.

Au niveau de cette architecture de HA, j'ai installé deux Cloud Controllers, mais il **serait aisé de rajouter N autre Cloud Controllers**. Il faudrait juste ajouter l'IP des nouveaux serveurs backends dans le fichier de configuration de chaque HAProxy et modifier les priorités des KeepAlived dans leurs fichiers de configurations. Avec la HA Active/Active, l'ajout d'un grand nombre de Controllers permet également une meilleure répartition des charges entre tous les Cloud Controllers grâce au Load Balancers.

Un problème réside à travers ce scénario, **comment détecter la perte d'un Controller** du fait que les échanges de Load Balancing se font automatiquement. Par conséquent pour déceler un problème, il est indispensable de se connecter sur chaque Controller et faire défiler les Logs à la main. Par la suite, il serait judicieux de mettre en place un système de monitoring avec Shinken, qui permettrait d'automatiser les tests de disponibilités pour que nous soyons alertés automatiquement des éventuels problèmes de pertes de services.
De plus, la mise en place d'un **serveur centralisé de Logs** est indispensable, afin de faciliter leurs analyses en cas de panne.

5.3 HA SQL (Active/Active Statefull)

Dans ce chapitre, je vais traiter de la haute disponibilité au niveau des bases de données SQL. Comme nous l'avons vu en §5.1, nous sommes en présence d'un système **Actif/Actif Statefull**. Voici les lignes directrices de mise en œuvre, afin de comprendre au mieux les interactions entre les différents nœuds SQL. Pour effectuer de la HA, au niveau de notre base de données, nous allons **dupliquer** cette **base** au sein de **plusieurs nœuds**, et l'ensemble des nœuds va constituer notre Cluster.

5.3.1 Schéma du Cluster SQL

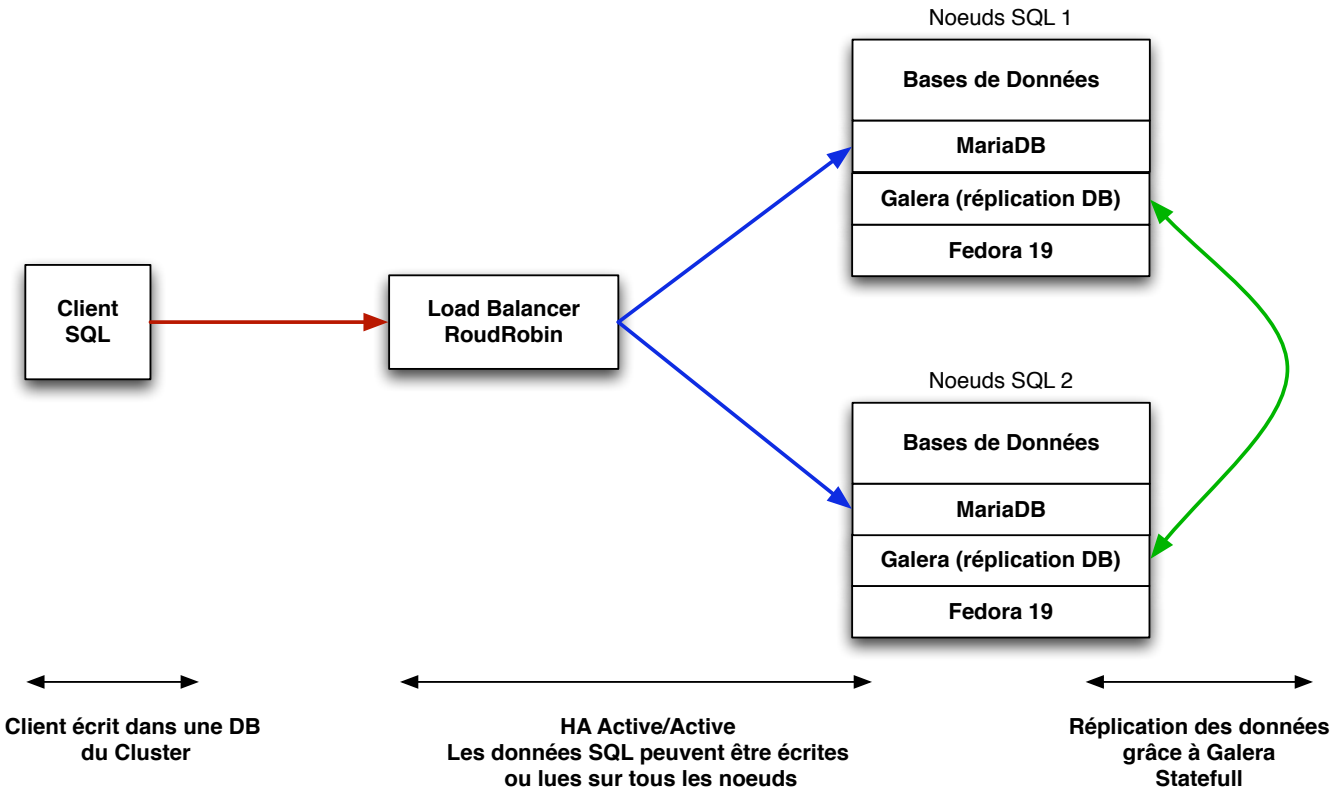


Schéma 18: HA au niveau SQL

Le schéma 18 nous illustre une vue globale des différents types de **HA qui interviennent uniquement au niveau de la partie SQL**. Cette représentation montre un client qui désire écrire dans une base de données. L'élément omniprésent pour une HA Active/Active est le Load Balancer, qui va répartir les requêtes de manière round robin au près des différents serveurs SQL. Depuis l'entrée en fonction de Fedora 19, le serveur MariaDB remplace le serveur MySQL suite au rachat de ce dernier par Sun Microystems.[17]

[17] Remplacement MySQL : http://fedoraproject.org/wiki/Features/ReplaceMySQLwithMariaDB#Detailed_Description

5.3.2 Théorème du CAP

Ce théorème[18] basé sur la conjecture « Brewer's Conjecture[19] » réalisé par le Professeur Eric Brewer de l'université de Berkley, a été prouvé par Nancy Lynch et Seth Gilbert du Massachusetts Institute of Technology.
Il énonce qu'un système distribué est régi par **trois contraintes** :

- **Consistence** (Cohérence) : Le fait d'avoir la même donnée sur tous les nœuds à un instant t
- **Availability** (Disponibilité) : Les clients peuvent en permanence trouver les données
- **Résistance au morcèlement** (Partition Tolerance) : Le système fonctionne malgré des problèmes réseaux

Mais d'après ce théorème, le système ne peut pas se trouver dans **plus de deux états** à la fois.
Prenons un exemple : Si l'on favorise la **cohésion** des données, les utilisateurs devront **attendre que tous les nœuds se synchronisent** entres eux avant de pouvoir effectuer une opération d'écriture ou de lecture.

Si l'on favorise la **disponibilité** des données, comme dans le cas de Google, un utilisateur effectuant une recherche et obtenant un résultat, ne va pas forcément obtenir le même résultat qu'un autre. **L'importance de ce système est de pouvoir retourner un résultat**, bien que la cohérence de ce dernier ne soit pas la règle prioritaire. L'accent est effectivement mis sur la disponibilité.

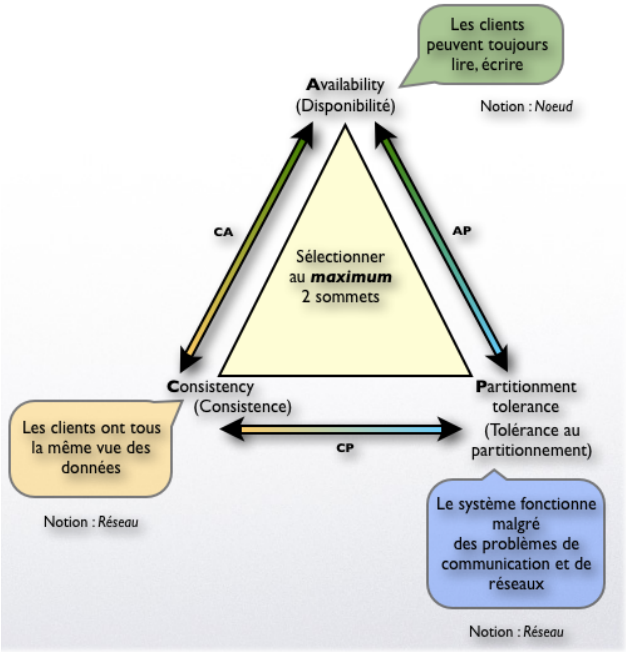


Schéma 19: Théorème du CAP

Le schéma 19 résume **les trois états** où peut se situer le Cluster[20]. Dans notre cas, notre Cluster de base de données se situera dans la partie Consistency et Availability.

[18] Théorème du CAP : http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.1495&rep=rep1&type=pdf
[19] Conjecture de Brewer : http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

L'accent sera donné sur la consistance des données. Par exemple, il est primordial que la base qui héberge les informations des VMs soit en permanence synchronisée, car si elle nous retourne une information fausse, toute notre prise de décision à propos du management des VMs sera erronée. Au final mieux vaut un petit temps de latence dû à la synchronisation, mais permettant d'obtenir des données crédibles.

5.3.3 Gestion du Cluster SQL

Comme nous pouvons le voir sur le schéma 18, les deux nœuds SQL sont basés sur Fedora 19, et les bases sont gérées par MariaDB. Come je l'ai énoncé plus haut, la HA est de type Statefull, donc nous devons mettre en place un système de réplications des données ainsi que la gestion de la disponibilité de notre Cluster SQL.

- **Galera**

Un outil Open Source qui va **gérer la consistance des données** au sein de notre Cluster. Sa mise en place est très bien documentée :
https://wiki.deimos.fr/MariaDB_Galera_Cluster_:_la_réplication_multi_maitres
Je ne l'ai pas mis en place du fait que sa mise en pratique ne rentrait pas dans le périmètre du projet. Comme je l'ai indiqué dans §5.3.2, nous portons une attention toute particulière à la consistance des données auprès de tous les nœuds. Donc avant que le client reçoive un ACK l'informant que sa donnée est bien écrite dans la base, il faut que tous les nœuds soient synchronisés.
L'avantage de mettre en place Galera, est que **tous les nœuds de notre Cluster peuvent recevoir des requêtes**, et donc nous avons l'abolition de la relation Master/Slave présente au sein d'un Cluster SQL standard ne possédant pas cette surcouche. Le point négatif de ce système, est que tant que la réplication n'est pas acquittée par tous, le cluster ne peut pas recevoir de requête supplémentaire[21]. Donc le temps de traitement de la requête dépendra intrinsèquement du nœud le plus lent.

[20] Schéma extrait : http://3.bp.blogspot.com/-XfutdbkRTT0/Uek3LA1lonI/AAAAAAAAAFA/CQeNLOaFk68/s1600/nosql_cap_theorem_triangle.png
[21] Failover Galera : http://fedoraproject.org/wiki/Features/ReplaceMySQLwithMariaDB#Detailed_Description

6 Scénario 3 : Collecte et entreposage des Logs

Comme indiqué ci-dessus, avec la topologie du scénario 2, il m'est **impossible de diagnostiquer l'état des services OpenStack**. Grâce à ce scénario, mon infrastructure Openstack va pouvoir réellement héberger une application qui permettra le stockage des Logs. Ce scénario va porter sur l'étude de différentes variantes d'acheminement des Logs à une base de données dédiée. De par la complexité de l'infrastructure, ainsi que des multiples services qui interviennent au sein d'un Cloud OpenStack, il devient nécessaire d'avoir un point central qui récupère les Logs, en vue d'une analyse du comportement des services.

6.1 Cahier des Charges

Les Logs sont des éléments **essentiels dans la gestion d'un Système d'Information**. Ils servent à analyser son comportement ou déceler diverses erreurs. Mais le gros problème réside dans le fait qu'ils sont souvent **répartis sur différentes machines physiques**, ce qui rend impossible la vision globale de l'état du système. LeShop.ch au sein de son infrastructure récolte des centaines de milliers de lignes de Logs/min provenant de ses différentes applications métiers.
LeShop.ch demande :

- La mise en place d'une infrastructure de stockage centralisée des Logs générés par les services d'OpenStack
- L'entreposage de manière durable au sein d'une base de données (DB) dédiée aux Logs
- La mise en place d'un moyen de visualisation des Logs, afin de déceler divers problèmes ou effectuer des recherches rapides dans la base de données

6.2 Contraintes

La liste des contraintes :

- La DB stockant les Logs doit se situer sur des VMs
- L'accent doit être mis sur la **disponibilité des données**. Par conséquent, il faut que la DB soit toujours disponible malgré la perte de deux VMs
- Il faut **scinder au maximum les rôles de chaque VM** intervenant dans la chaîne de traitement des Logs. Dans le cas contraire, en regrouper une majorité au sein d'une VM, va charger à 100% sa RAM et son CPU et risque de faire crasher ses services
- La suite d'outils développée en JAVA donc veiller à optimiser la Java Virtual Machine
- Travailler en temps réel, du fait que les Logs affluent continuellement
- Intégrer au maximum des éléments de HA en essayant de dupliquer au maximum les services. Comme ils sont installés sur des VMs, le fait de les dupliquer ne demande pas de ressources physiques supplémentaires

6.3 Variantes de stockage de Logs

Pour mettre ces variantes sur pieds j'ai suivi ce très bon lien proposé par le CERN : http://indico.cern.ch/getFile.py/access?contribId=57&sessionId=1&resId=0&materialId=slides&confId=220443

Quatre variantes vont être étudiées, qui vont se baser sur ces éléments communs :

- **DB ElasticSearch**: DB Clés-Valeurs qui stockera les Logs
- **Serveur physique stockant les Logs** envoyés par les différents services OpenStack grâce au protocole Syslog, de manière temporaire, avant de les envoyer à la DB. Le fonctionnement du serveur sera développé dans la variante 1
- **Outil de visualisation des Logs** : Kibana: Se matérialise par un serveur Web s'exécutant du côté de l'administrateur désirant avoir une vue d'ensemble des Logs dans le but de rechercher quels services génèrent des Logs d'erreurs

6.3.1 Variante 1 : Stockage des Logs sous forme de fichiers

Cette variante vise à mettre en place un serveur physique de Logs sous forme de fichier. Les différents services d'une distribution vont écrire dans un fichier tous leurs événements. Les services ont la possibilité d'être plus ou moins prolixe, nous avons le mode Debug, où un maximum d'informations sera stocké dans les Logs. Mais il est **très utile pour analyser le comportement d'un service**, et analyser quels messages il reçoit ou envoie. De base, chaque **fichier de Logs des services est hébergé en local** sur la machine physique qui l'exécute. Cette architecture ne permet pas une analyse aisée car il faut se connecter sur chaque machine physique pour interroger les Logs en local. Donc il faut absolument mettre en place une architecture qui centralise les Logs. L'infrastructure est la même qu'en §5.2.1, mais un PC qui va héberger les Logs de tous les services a été ajouté.

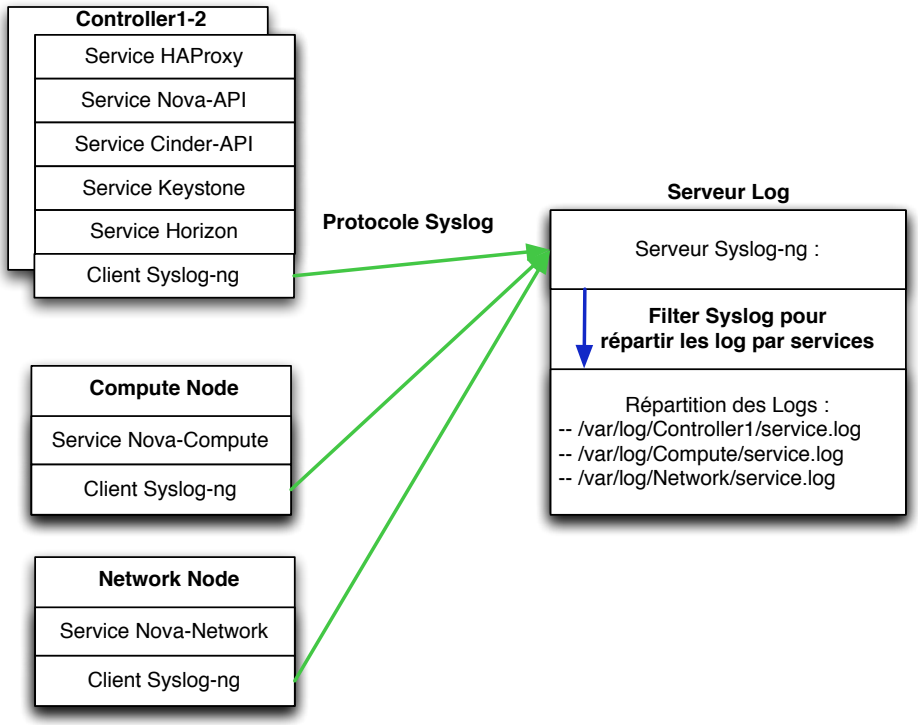


Schéma 20: Récolte des Logs en un point centrale

A travers le schéma 20, nous avons tous les **services OpenStack qui envoient leurs Logs** via le **client Syslog-ng** au serveur de Log. Syslog-ng est un package Open Source qui regroupe les fonctions client-serveur en ce qui concerne l'envoie des Logs via le protocole Syslog. Chaque client va établir une connexion avec le serveur Syslog afin de lui envoyer les Logs des services qu'il héberge. Et ensuite le serveur va stocker en local, les différents Logs en les séparant dans différents fichiers en fonction des règles de filtrages que j'ai construites.

- **Les Logs des services OpenStack**

Chaque service Openstack stocke ces Logs en local sur la machine physique qui l'héberge. Voici un exemple de Log généré par le service Nova-Network stocké dans ***/var/log/nova/nova.conf*** du Network Node:

```
2013-12-03 08:45:53.944 10131 DEBUG qpid.messaging.io.raw [-] SENT[472d248]:
'\x0f\x00\x00\x10\x00\x00\x00\x00\x00\x00\x00\x01\n\x00\x00' writeable
/usr/lib/python2.7/site-packages/qpid/messaging/driver.py:466
```

La majorité des Logs sont des messages AMQP à cause de l'activation du mode Debug.

- **Problématique de la provenance des Logs**

Comme nous avons pu le voir dans l'exemple du message précédent, en voyant uniquement cette ligne nous **ne savons pas par quel service a généré cette ligne**, excepté en faisant la commande : `nano /var/log/nova/network.log` sur le Network Node. Ceci est un problème car sur le Serveur de Log, il serait intéressant de répartir les Logs au sein de différents fichiers propres à chaque service et répartis dans différents dossiers correspondants à chaque entité physique.
Voici un exemple de configuration que nous aimerions avoir sur notre Serveur de Logs.

Serveur Log

```
/var/log/controller1/
├── cinder.log
├── glance.log
├── haproxy_access.log
├── haproxy_status.log
├── horizon.log
├── keepalived.log
├── keystone.log
└── nova.log

/var/log/controller2/
├── cinder.log
├── glance.log
├── haproxy_access.log
├── haproxy_status.log
├── horizon.log
├── keepalived.log
├── keystone.log
└── nova.log

/var/log/compute_node/
└── compute.log

/var/log/network/
└── network.log
```

Pour cela les clients Syslog ne vont pas parcourir les différents fichiers de chaque service, ils vont interroger et envoyer les lignes s'ajoutant au fichier ***/var/log/messages***. J'ai choisi d'envoyer les lignes qui s'ajoutent dans ce fichier car les éléments qui s'y inscrivent possèdent un formatage plus simple à extraire. Voici un extrait du ***/var/log/messages/*** du Network-Node :

```
Dec  1 03:27:37 network nova-network[10131]: 2013-12-01 03:27:37.883 10131 DEBUG
qpid.messaging.io.ops [-] RCVD[604d3f8]: ConnectionHeartbeat() write
/usr/lib/python2.7/site-packages/qpid/messaging/driver.py:642
```

Comme nous pouvons le voir, les lignes générées dans ce fichier centralisent les Logs de tous les services indexant leur journal d'événement. De plus le **formatage** des lignes est généré en suivant le **standard** proposé par la **RFC3164**. Le message est constitué d'un **HEADER**[22] :

- `Dec  1 03:27:37` : Timestamp, à quelle date et heure a été généré le message
- `network` : La machine physique qui a généré le message
- `nova-network[10131]` : Le nom du processus avec son numéro de PID qui a généré cette ligne

Au final, les **Logs** sont **centralisés** sur notre serveur de Logs (voir A.3 pour la configuration du Syslog-NG client-serveur), mais sous la forme de fichier. A ce stade, pour analyser le comportement des services OpenStack, nous **sommes toujours limité par la lisibilité des Logs**. Pour ce faire, l'idéal serait de pouvoir les interroger depuis une interface Web. De plus le stockage des Logs sous forme de fichiers est un grand consommateur d'espace disque.

6.3.2 Variante 2 : Logstash -> ElasticSearch

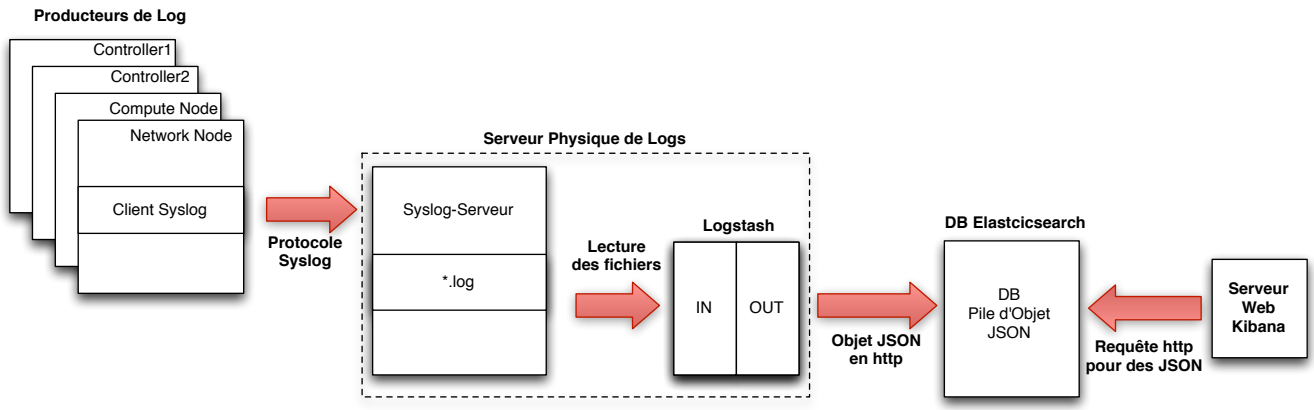


Schéma 21: Producteurs de logs -> Consommateurs

Le schéma 21 représente la chaîne de traitement qui va acheminer les Logs contenus sous forme de fichier jusqu'à une DB dédiée à ce stockage.

Cette **variante permet** :

- **La récolte des Logs sous forme de fichiers** grâce au protocole Syslog des différents PC gérant l'infrastructure OpenStack au sein du serveur de Logs. (Voir variante 1 §6.1.3)

- La **conversion des Logs** grâce à **Logstash** (Exécuté sur serveur de Logs):

Pour centraliser les Logs dans la base de données, nous ne pouvons pas les garder sous forme de fichiers, car ceci consommerait trop de ressources CPU de parcourir tous ces fichiers de Logs au moment où nous faisons des requêtes d'extractions. Pour cela nous devons utiliser un outil écrit en JAVA qui va transformer les Logs bruts (**ligne d'un**

[22] HEADER Syslog : http://tools.ietf.org/html/rfc3164#section-4.1.2

fichier) en **Objet JSON** afin de les envoyer à notre DB en http (voir Annexe A.4 pour une explication des objets JSON).
Exemple d'un objet JSON transformer à partir d'une ligne de Log:

```
"message"  =>  "Dec    3  00:24:38  controller  haproxy[1474]:    10.2.4.107:39510
[03/Dec/2013:00:24:38.457] nova-compute-api nova-compute-api/controller-2 0/0/4 280
-- 0/0/0/0/0 0/0",
              "@timestamp" => "2013-12-03T00:24:38",
                    "type" => "syslog_haproxy",
                    "path" => "/var/log/controller1/haproxy_access.log",
                 "program" => "haproxy",
              "received_at" => "Dec  3 00:24:38",
            "syslog_severity" => "INFO",
            "@source_host" => "controller",
               "@message"  =>  "  10.2.4.107:39510  [03/Dec/2013:00:24:38.457]  nova-
compute-api nova-compute-api/controller-2 0/0/4 280 -- 0/0/0/0/0 0/0"
}
```

Remarque : Lien détaillant la structure des Logs de HAProxy :
http://www.exceliance.fr/sites/default/files/biblio/aloha_load_balancer_memo_log.pdf

Cet outil, Logstash a besoin d'un fichier de configuration qui regroupe les filtres que j'ai a dû créer manuellement dans le but d'obtenir un template pour découper notre ligne de Log en un objet JSON. Voici un exemple d'un fichier de configuration de Logstash : https://github.com/hepia-telecom-labs/Openstack_Config/blob/master/Object_config/HA/Rsyslog_serv/Logstash/config_els.conf

Chaque Log est représenté par un Objet comportant des champs facilitant le classement au sein de la DB et permettra des recherches plus fines avec notre outil de visualisation des Logs.

- La DB ElasticSearch :

ElasticSearch est une DB permettant le **stockage** de données de types **JSON**, la particularité de ce gestionnaire de documents est le fait de traiter l'entrée de données en temps réel. La DB va recevoir les Logs de Logstash, pour les entreposer. Pour la consultation, elle va recevoir des requêtes via son API, et va retourner au client les différents objets JSON concernés par sa requête.

La DB est hébergée sur des VMs ce qui permet d'augmenter facilement la taille de notre Cluster sans rajouter d'éléments physiques.

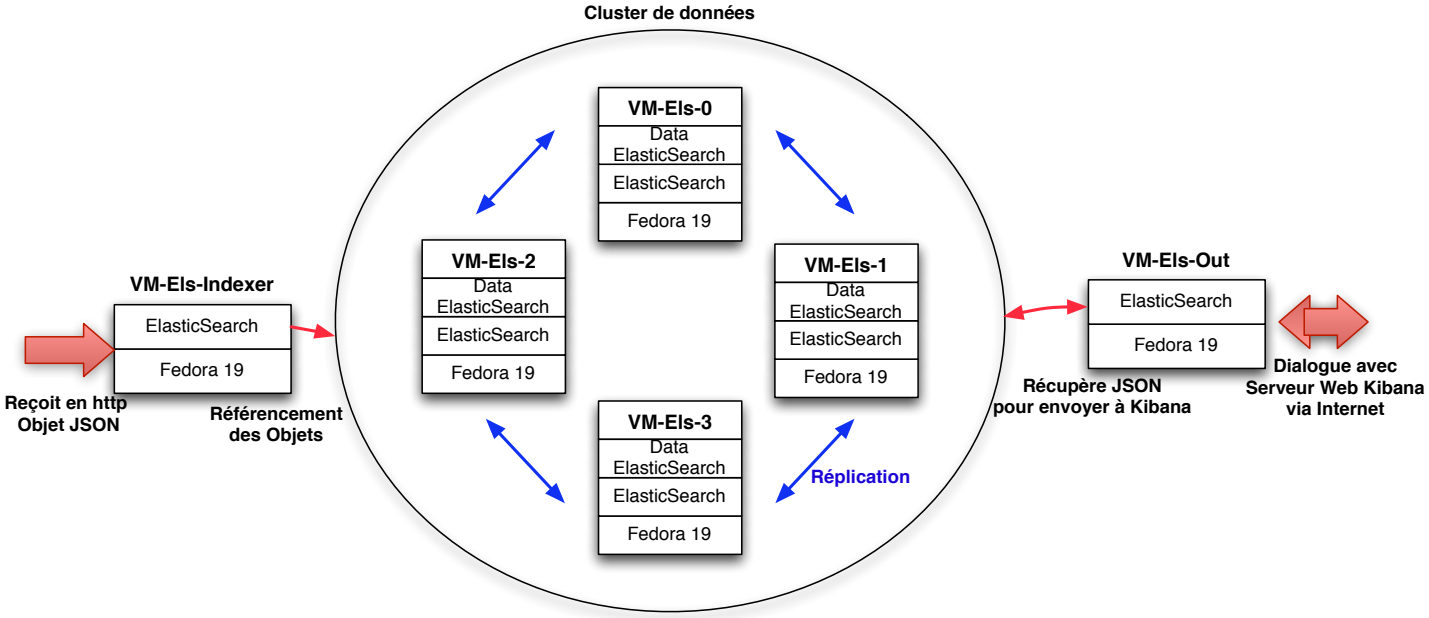


Schéma 22: Cluster ElasticSearch

Voici le cœur du stockage, nous retrouvons trois types de VMs :

- **VM-Els-Indexer** : Ne possède **aucune unité de stockage** au niveau de la vue du Cluster, mais doit estampiller avec un ID chaque Objet JSON pour qu'ils puissent être classés dans les VMs de stockages.
- **VM-Els-0..3** : **Hébergent les Objets JSON** en effectuant en permanence des réplications pour maintenir une consistance des données au sein du Cluster.
- **VM-Els-Out** : Reçoit les requêtes depuis Internet pour transmettre les Logs à l'outil de visualisation.

Ce cloisonnement de rôle a été mis en place pour étaler au maximum les charges CPU et RAM, et également faciliter l'expansion du Cluster.

- **Exploitation des Logs** :

Le serveur Web **Kibana** s'exécutant sur le PC local de l'administrateur peut **interroger la DB afin de récupérer les Logs**. A l'aide des outils fournis par Kibana, il est possible d'afficher ces derniers sous la forme de graphiques (histogrammes et camembert) afin de détecter d'éventuelles erreurs. Cette interface GUI est entièrement personnalisable du fait que l'on choisit de générer ces graphiques en fonction des clés des Objets JSON. Voici les différents graphiques que l'on peut générer avec Kibana :

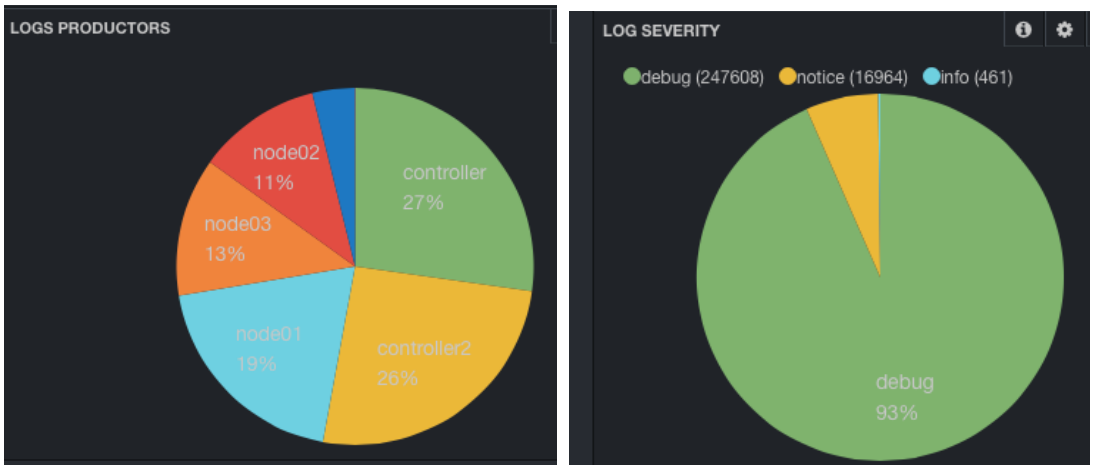


Figure 4: Graphiques générés par Kibana

La **particularité** de ce serveur Web ne va **pas s'exécuter sur le Cluster** mais sur le poste de l'administrateur, déchargeant ainsi le Cluster de la gestion des templates html.

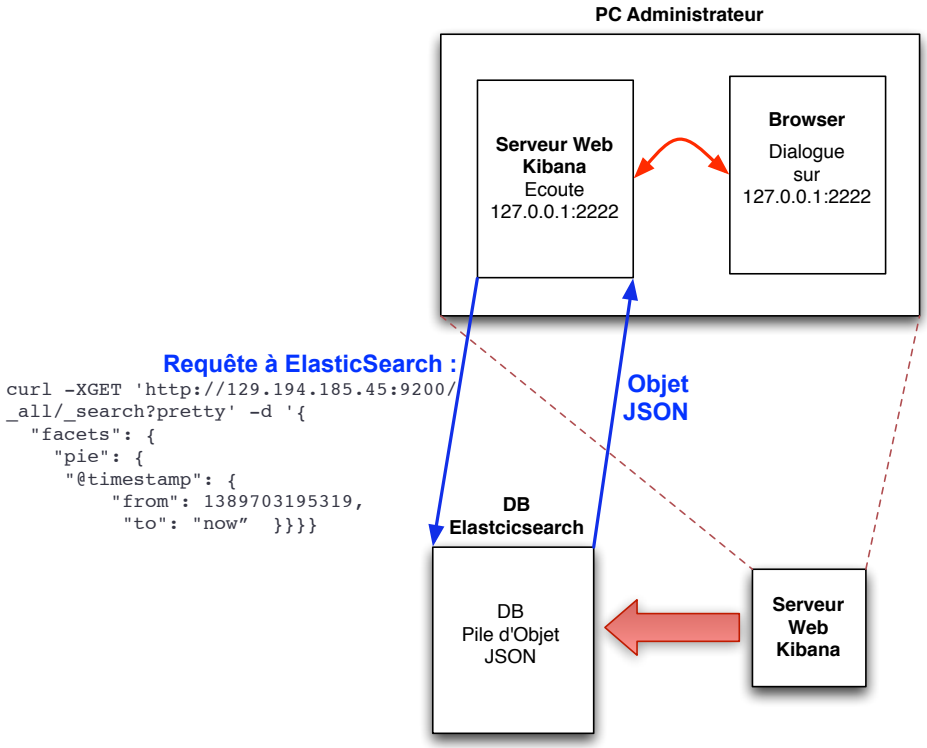


Schéma 23: Récupération des Logs avec Kibana

Comme nous le voyons sur le schéma 23, toute la gestion du template du serveur Web est gérée local, s'exécutant sur le poste de l'administrateur. Le serveur dialogue avec la

DB ElasticSearch uniquement pour récupérer les objets JSON. En utilisant l'inspecteur réseau intégrer au Browser Firefox, nous voyons bien le **chargement du template en local** et la **récupération des Objets JSON distant** :

GET	module.html?r=f4cc564	127.0.0.1:2222	html
GET	module.html?r=f4cc564	127.0.0.1:2222	html
GET	module.html?r=f4cc564	127.0.0.1:2222	html
POST	_search	129.194.185.45:9200	json
POST	_search	129.194.185.45:9200	json

Figure 5: Chargement des JSON distants

Au final, cette variante permet de **stocker nos Logs de manière centralisée** dans une DB pouvant être **toujours opérationnelle malgré la perte de deux VMs** de stockage. De plus, l'utilisation de Kibana permet de ressortir les Logs en vue d'analyse de comportement de notre infrastructure.

6.3.3 Variante 3 : Logstash-> Redis -> Logstash -> ElasticSearch

Sachant que les Logs inscrits dans les fichiers .log du serveur Physique de **Logs arrivent plus vite qu'ils ne sortent vers la DB ElasticSearch**, Logstash est obligé de les bufférisier en interne, mais ses performances de ce domaine sont plus que médiocre. Par conséquent, cette deuxième variante va intégrer un **élément supplémentaire servant de Buffer de Logs** (VM-Buffer), qui sera géré par l'outil Redis. La grande force de cet outil est de stocker les Objets JSON dans la RAM, ce qui le rend très performant dans la mise en mémoire des Objets envoyés par Logstash.

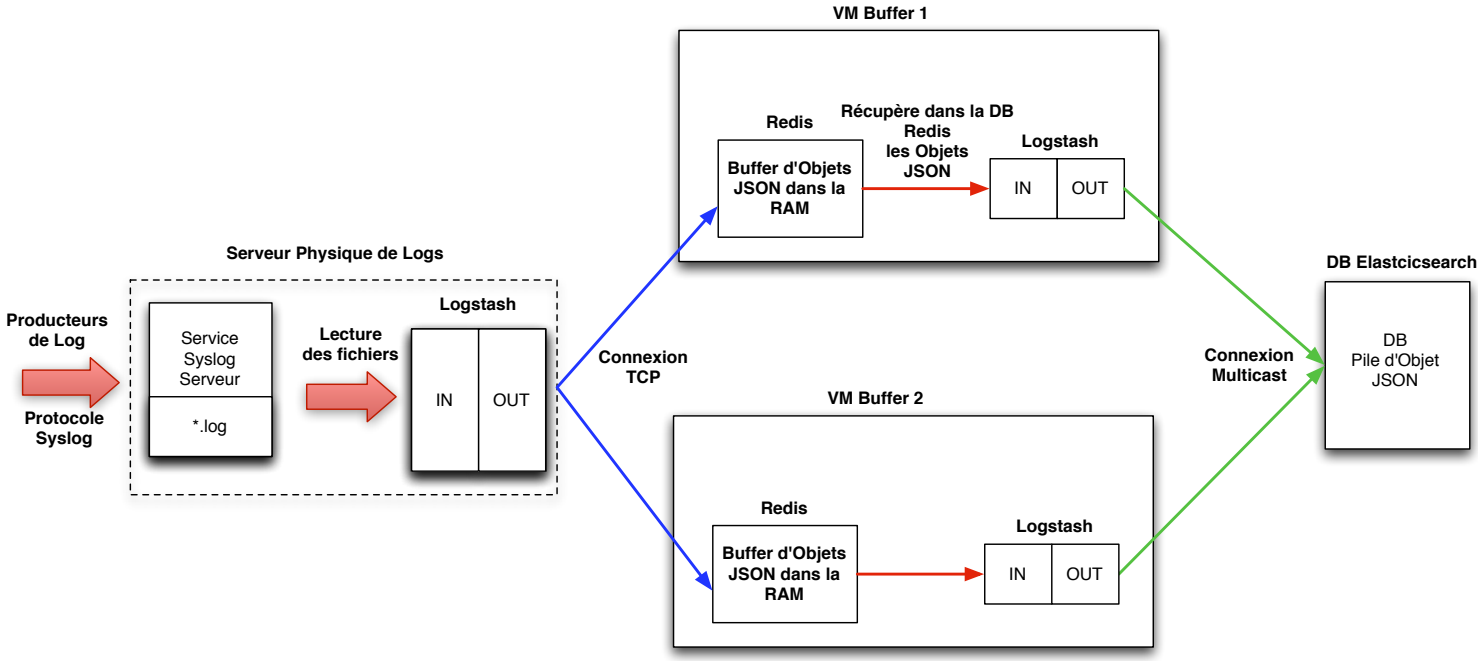


Schéma 24: Mémoire tampon de Logs

Le fonctionnement de Logstash dans le Serveur Physique de Logs a toujours la même fonctionnalité que précédemment, mais il maintient une connexion TCP avec le Buffer Redis, afin de lui envoyer les Logs transformés à la volée.

Un deuxième Logstash est installé sur cette VM, il va être utilisé uniquement pour envoyer les objets JSON à ElasticSearch.
Nous avons la présence de **deux VMs Buffer pour créer de la redondance** afin que le flow de Logs ne soit pas interrompu. Si la connexion TCP tombe avec le premier il bascule sur la 2ème VM-Buffer.

La communication entre le Logstash des VM-Buffer et ElasticSearch se fait par Multicast donc contrairement à la variante 2, il n'a **plus besoin de dialoguer avec la VM-Indexer**. Cette topologie permet d'intégrer les Logstash des VM-Buffer comme un nœud du Cluster, les Logs peuvent être alors envoyés à n'importe quelle VM-Data, et par conséquent nous n'avons plus de « point of failure »:

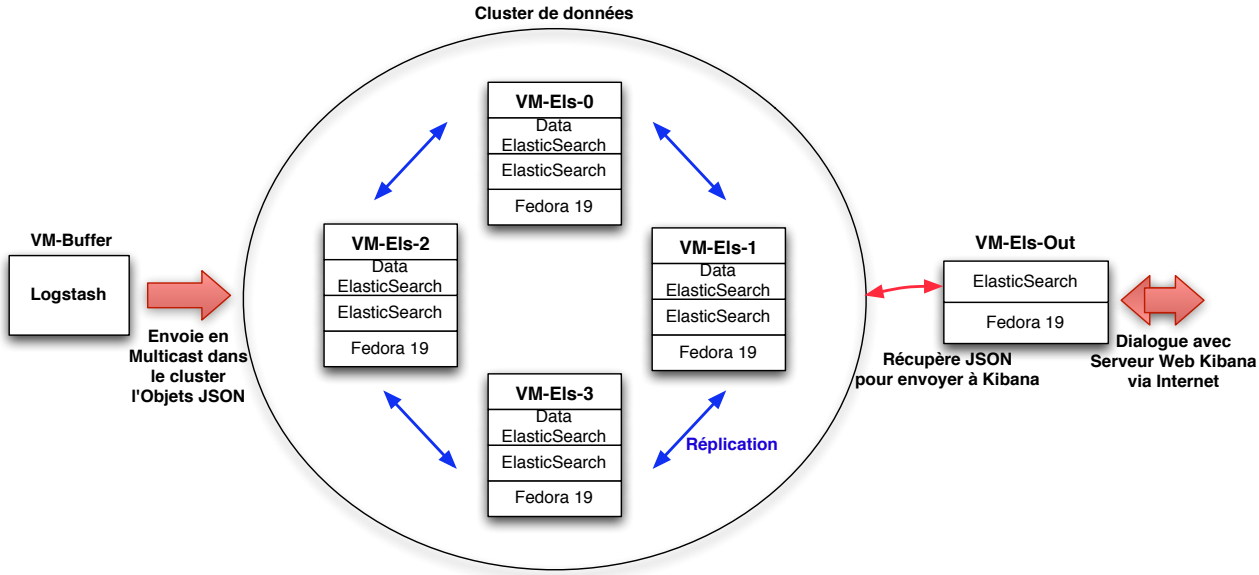


Schéma 25: Cluster ELS variante 2

6.3.4 Variante 4 : Abolition du serveur de Logs physique

Cette variante a pour but de **supprimer le serveur physique de Logs**, et d'installer Logstash sur chaque serveur physique. Chaque serveur physique gérant l'infrastructure communiquerait avec les deux VMs Buffer pour leurs envoyer les Logs sous format JSON.

6.3.5 Comparaison des quatre variantes et choix

Bilan Variante 1

- Cette variante a permis de centraliser les Logs de tous les éléments de l'infrastructure OpenStack au sein d'un même serveur. Mais elle ne remplit pas pleinement le cahier des charges car les logs sous forme de fichiers sont impossibles à exploiter.

Bilan Variante 2

- Cette variante a pour but de mettre en place une gestion des Logs avec les éléments essentiels qui sont :
  - La récupération des Logs
  - La conversion
  - Le stockage

- La visualisation des Logs de manière à effectuer des recherches d'aides à la décision

- Points Positifs :
  - Remplit le cahier des charges dans la majorité des points
  - Bonne **séparation des tâches** pour l'acheminement des Logs
  - Kibana permet de faire du **regroupement de données**, pour que l'administrateur qui consulte ces Logs puissent faire ressortir les informations qui lui semblent pertinentes
- Points Négatifs :
  - **Le goulet d'étranglement est le buffer** géré par Logstash
  - La disponibilité n'est pas vraiment au rendez-vous entre Logstash et ELS. Du fait que Logstash ne communique qu'avec un seul point de la DB ElasticSearch en http. Notre serveur physique de Logs est également un élément sensible

Bilan Variante 3

- Points Positifs :
  - On **décharge le buffer de Logstash** du serveur physique et donc un gain de performance
  - De par la présence de deux VMs Buffer, nous augmentons la HA.
  - Les Logstasch des VM-Buffer sont considérés comme un nœud du Cluster (inutilité de la VM-Indexer)
- Points Négatifs :
  - Présence de notre serveur physique récupérant les Logs

Bilan Variante 4

- Point Positif :
  - **Abolition du serveur physique qui centralise les Logs**, ce sont les producteurs de Logs qui hébergent chacun un Logstash
- Points négatifs :
  - Comme Logstash est un service JAVA, il a besoin de la Java Virtual Machine pour s'exécuter. Cette **JVM peut vite devenir très gourmande en ressource**. Il faut absolument dimensionner correctement la JVM, qui n'est pas une tâche facile
  - Comme Logstash a besoin d'un fichier de configuration qui contient les règles de transformations des lignes de Logs en objets JSON, il faudra veiller à ce que tous les Logstash aient le bon fichier de configuration, ce qui rajoute une couche supplémentaire de contrôle

6.3.6 Choix de la variante

Suite aux différents points évoqués, je préconise comme système de gestion des Logs **la variante n°3** (§6.3.3). Même si le « Point of Failure » est le serveur physique de Logs, nous n'encombrons pas les Producteurs de Logs avec le service Logstash qui peut vite s'avérer gourmand en ressource. Grâce à cette variante les Logs sont également dans une DB gérant les réplications des données au sein de son Cluster, afin de pouvoir survivre à la perte de deux VM-Els.
Comme nous sommes dans un **univers virtualisé,** il est aisé de **rajouter des VMs afin d'augmenter nos capacités de stockages**.

6.4 Vélocité de mon infrastructure OpenStack

Le but de ce stockage de Logs, a été de générer de la charge au sein de mes différentes VMs grâce à l'important volume que produisent les services OpenStack en mode Debug.

J'ai mis en place la **variante n°2**, ma DB ElasticSearch est composé de

- **4 VM-ELS-Node** (VM-Data), **1 VM-Els-Indexer** et **1 VM-Els-Out**:
  - 1 vCPU
  - 1024 MB de RAM
  - 20 GB de disque virtuel

Au sein des VM-Data, j'ai appliqué une **réplication de deux**, c'est-à-dire pour une donnée inscrite sur un nœud, elle va être répliqué sur deux autres nœuds. Donc au niveau de la disponibilité de la DB, on peut **perdre jusqu'à deux nœuds** pour toujours garder la consistance des données.
Grâce à un plugin GUI intégré à la DB ElasticSearch, j'ai pu voir le nombre d'objets JSON enregistrait pendant 24h. Il faut savoir que Logstash va affilier ces objets JSON à un index, et cet index change toutes les 24 heures. Voici un aperçu de l'interface GUI :

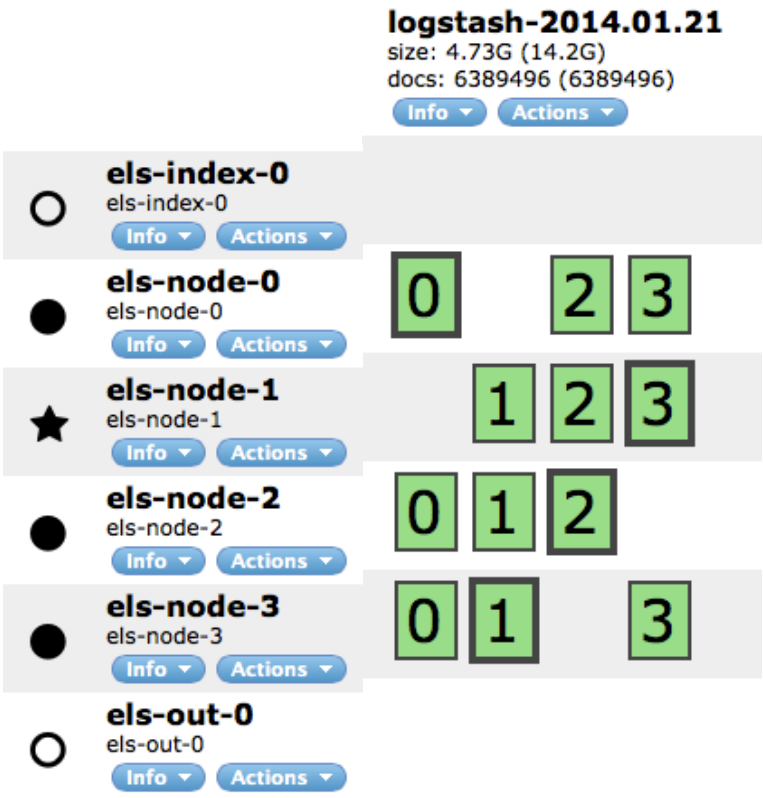


Figure 6: Aperçu du Cluster

Nous voyons à **gauche** la liste de tous les **nœuds du Cluster ElasticSearch**, et les 4 VM-Data ont à droite les parties de données qu'elle possède.
Attardons-nous maintenant sur les chiffres : rien que le **21 janvier** entre 00h et 23 :59, les services OpenStack ont généré environ **6'400'000 lignes de Logs** ou Objet JSON, ce qui représente **4.73GB**, mais avec le répliqua de deux, ces Logs occupent au sein du Cluster 14.2GB. Sachant qu'il n'y a que 4 VMs de Data, je peux stocker un peu moins de 80GB, c'est-à-dire environ 5 jours et demi de Logs. A travers ces chiffres, le **stockage** peut vite **devenir problématique**, car ma « petite » infrastructure, génère déjà

14.2GB/Jour de Logs avec la réplication, pour conserver une année de Logs il faudrait 5TB, ce qui est infime face au volume généré par les services Web au sein de LeShop.ch. Au niveau de la vitesse de génération de Logs, ceci représente environ **4'500 Objets JSON/Minute**.
Malgré cette charge importante au sein des VMs, il est toujours possible de démarrer de nouvelles VMs, et les services OpenStack sont toujours disponibles.

6.5 Problèmes rencontrés

J'ai passé beaucoup de temps à travailler sur les **formatages des Logs** afin que Logstash puisse créer des objets JSON standards quel que soit le service OpenStack générant les Logs.
Un problème topologique au niveau du réseau s'est posé. Comme illustré dans mes différents schémas d'infrastructure, les VMs possèdent leur propre réseau (LAN Orange), qui est physiquement séparé du réseau administratif (LAN Bleu). Pour faire remonter les objets JSON de Logstash vers ElasticSearch situé sur une VM, j'ai **installé une seconde carte réseau à mon serveur de Logs**, pour qu'il puisse se **connecter au réseau privé** des VMs. Ceci n'est pas recommandé dans la pratique, je le fais uniquement pour générer du trafic au sein des VMs.

6.6 Bilan du scénario 3

A travers ce scénario, j'ai pu étudier et mettre en place une réelle stratégie de gestion de Logs, avec l'intégration d'un **système de recherche performant**. Ce scénario s'adapte parfaitement à un environnement virtualisé, où le maître mot est la « **scalability** ». Si nous voyons notre Cluster devenir trop faible au niveau stockage, nous pouvons ajouter une nouvelle VM facilement. Ce sont les **ressources** telles que les VMs qui **s'adaptent à la demande**, grâce à ce type de gestion, nous ne sommes jamais dans des cas extrêmes de sous-dimensionnement ou surdimensionnement.

7 Scénario 4 : Supervision des ressources avec Shinken

A travers ce scénario, j'ai ajouté à mon serveur de Log une fonction de supervision grâce à l'outil Shinken. Ce scénario ne va pas vous faire découvrir le fonctionnement de l'outil Shinken, mais comment j'ai **monitoré la HA des API OpenStack**.
Le Projet d'Approfondissement de Master de M. Lionel Schaub propose une vue détaillée des possibilités qu'offre Shinken : http://www.tdeig.ch/shinken/Schaub_RPA.pdf

7.1 But du scénario

Le but de ce scénario est de **monitorer la HA** que j'ai mise en place au cours des scénarios précédents. Je vais également monitorer les **ressources physiques** tels que la quantité de RAM utilisée des mes deux Controllers ainsi que de mon Compute-Node.
La page de mon dépôt GitHub à propos des différents fichiers de configuration que j'ai utilisés pour configurer les services et les hôtes se trouvent: https://github.com/hepia-telecom-labs/Openstack_Config/tree/master/Object_config/HA/Rsyslog_serv/Shinken

7.2 Schéma de principe

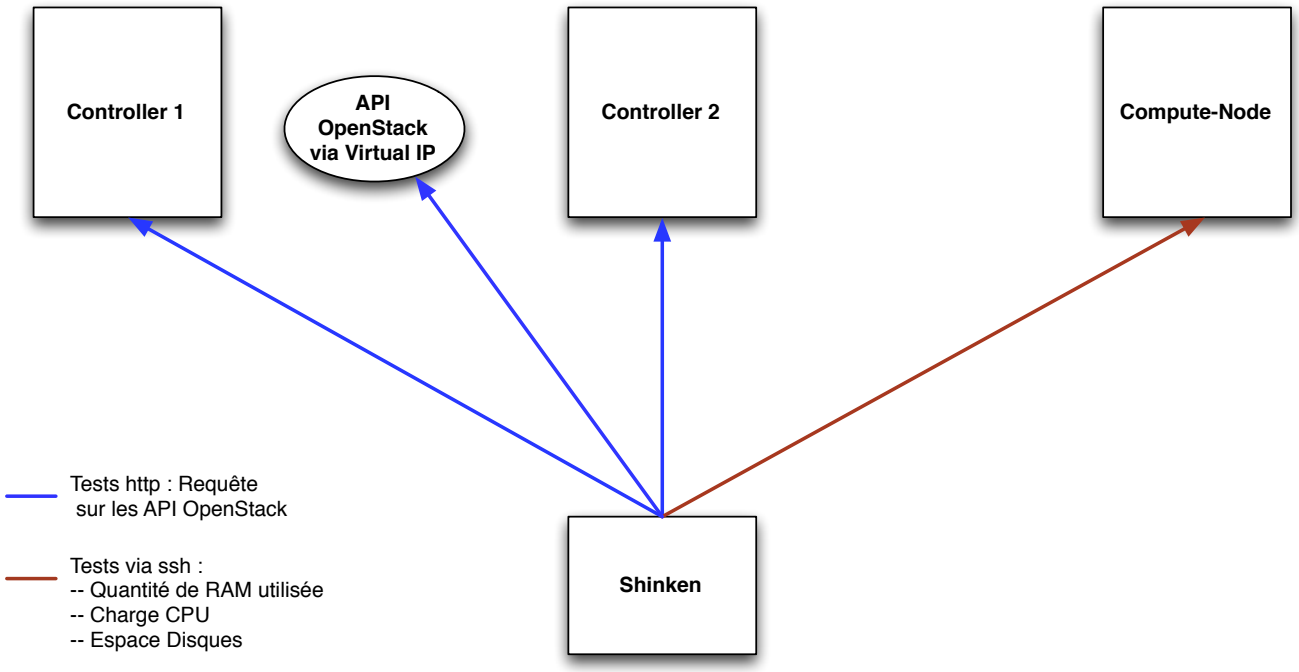


Schéma 26: Monitoring des API et serveurs physiques

7.3 Supervision des services d'API OpenStack

Dans Shinken, j'ai crée deux types de tests http :

1. **http sur l'IP physique** : Teste le service en l'interrogeant directement sur l'adresse IP d'un des deux Controller (10.2.4.101 et 10.2.4.106).
2. **http sur Virtual IP** : Teste le service mais en **passant par HAProxy** en envoyant une requête sur l'adresse IP Virtuelle, donc le paquet sera aiguillé sur l'un ou l'autre Controller

En testant les API via leur adresse IP virtuelle, même si par exemple le service de l'interface Web est inactif sur le Controller1, le service Web dans sa globalité est toujours en fonction car il sera redirigé vers l'autre Controller.

Etat du Système où tous les services OpenStack sont actifs:

clusterAPI.leshop		UP	1w 16h	TCP OK - 0.000 second response time on port 22	0.000335s
	Glance API	OK	1d 16h	HTTP OK: Status line output matched "300" - 692 bytes in 0.004 second response time	0.004181s
	Horizon Dashbo...	OK	1d 16h	HTTP OK: Status line output matched "200" - 2104 bytes in 0.019 second response time	0.018779s
	Keystone Admin	OK	1d 16h	HTTP OK: Status line output matched "300" - 1110 bytes in 0.003 second response time	0.003347s
	Nova API	OK	1d 16h	HTTP OK: Status line output matched "200" - 280 bytes in 0.005 second response time	0.005187s
controller1.leshop		UP	5d 22h	TCP OK - 0.000 second response time on port 22	0.000295s
	Glance API	OK	1d 4h	HTTP OK: Status line output matched "300" - 692 bytes in 0.002 second response time	0.002124s
	Horizon Dashbo...	OK	1d 4h	HTTP OK: Status line output matched "200" - 2122 bytes in 0.016 second response time	0.015827s
	Keystone Admin	OK	1d 4h	HTTP OK: Status line output matched "300" - 1110 bytes in 0.003 second response time	0.002919s
	Nova API	OK	1d 4h	HTTP OK: Status line output matched "200" - 280 bytes in 0.003 second response time	0.003106s
controller2.leshop		UP	3d 5h	TCP OK - 0.000 second response time on port 22	0.000238s
	Glance API	OK	8h 46m	HTTP OK: Status line output matched "300" - 692 bytes in 0.003 second response time	0.002973s
	Horizon Dashbo...	OK	8h 44m	HTTP OK: Status line output matched "200" - 2148 bytes in 0.016 second response time	0.016165s
	Keystone Admin	OK	8h 46m	HTTP OK: Status line output matched "300" - 1110 bytes in 0.003 second response time	0.00281s
	Nova API	OK	8h 46m	HTTP OK: Status line output matched "200" - 280 bytes in 0.003 second response time	0.003394s

Figure 7: Infrastructure opérationnelle

Voici la situation où le service Horizon est inactif sur le Controller1 :

controller1.leshop	Horizon Dashbo...	CRITICAL	16s	Connection refused	
clusterAPI.leshop		UP	1w 17h	TCP OK - 0.000 second response time on port 22	0.000431s
	Glance API	OK	1d 17h	HTTP OK: Status line output matched "300" - 692 bytes in 0.003 second response time	0.003286s
	Horizon Dashbo...	OK	1d 17h	HTTP OK: Status line output matched "200" - 2148 bytes in 0.029 second response time	0.029342s
	Keystone Admin	OK	1d 17h	HTTP OK: Status line output matched "300" - 1110 bytes in 0.003 second response time	0.003075s
	Nova API	OK	1d 17h	HTTP OK: Status line output matched "200" - 280 bytes in 0.005	0.004682s
controller1.leshop		UP	5d 22h	TCP OK - 0.000 second response time on port 22	0.00026s
	Glance API	OK	1d 4h	HTTP OK: Status line output matched "300" - 692 bytes in 0.003 second response time	0.003063s
	Keystone Admin	OK	1d 4h	HTTP OK: Status line output matched "300" - 1110 bytes in 0.002 second response time	0.002s
	Nova API	OK	1d 4h	HTTP OK: Status line output matched "200" - 280 bytes in 0.003 second response time	0.003273s
controller2.leshop		UP	3d 5h	TCP OK - 0.000 second response time on port 22	0.000185s
	Glance API	OK	8h 56m	HTTP OK: Status line output matched "300" - 692 bytes in 0.002 second response time	0.00239s
	Horizon Dashbo...	OK	8h 54m	HTTP OK: Status line output matched "200" - 2148 bytes in 0.025 second response time	0.025384s
	Keystone Admin	OK	8h 56m	HTTP OK: Status line output matched "300" - 1110 bytes in 0.003 second response time	0.002676s
	Nova API	OK	8h 56m	HTTP OK: Status line output matched "200" - 280 bytes in 0.003 second response time	0.00336s

Figure 8: Fonctionnement de la HA malgré un des deux Horizon Down

Nous voyons que l'interface **Horizon** est **toujours disponible pour l'administrateur** qui l'interroge via l'IP virtuelle. Grâce à ce procédé, je peux contrôler le fonctionnement de ma HA.

7.3.1 Problème au niveau des tests via l'IP Virtuelle

J'ai rencontré un **problème lorsque mes deux services horizons étaient inactifs** sur les deux Controllers. Le test au niveau du ClusterAPI indiquait que le service horizon était actif, car j'utilisais uniquement la commande check_tcp. Comme HAProxy en cas d'indisponibilité de service renvoie une réponse http avec un code 503 (Service indisponible), le check_tcp de Shinken restait à OK, car il arrivait à établir une liaison TCP avec HAProxy. Par conséquent, il a fallu que j'utilise le test check_http, pour qu'il puisse détecter les différentes réponses http.
Voici la réponse en faisant un « Curl » sur HAProxy en cas d'indisponibilité des deux services sur les deux Controllers :

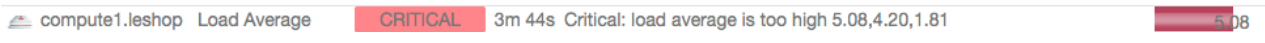
```
curl 10.2.4.100:80 -v

* About to connect() to 10.2.4.100 port 80 (#0)
*   Trying 10.2.4.100...
* Connected to 10.2.4.100 (10.2.4.100) port 80 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.29.0
> Host: 10.2.4.100
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 503 Service Unavailable
< Cache-Control: no-cache
< Connection: close
< Content-Type: text/html
<
<html><body><h1>503 Service Unavailable</h1>
No server is available to handle this request.
</body></html>
```

7.4 Supervision des ressources physiques

J'utilise Shinken non plus pour tester la disponibilité d'une API, mais pour **remonter les informations de charge CPU**, **la quantité d'espace disque utilisée et la quantité de RAM utilisée**. Pour cela, Shinken va se connecter en ssh au différentes machines physiques afin d'exécuter des commandes qui nous retourne ces différentes informations. Comme par exemple, la charge CPU est obtenue en faisant :

`cat /proc/loadavg`
`0.21 0.15 0.10` -> Retourne la moyenne de la charge sur 1 minute, 5 minutes et 15 minutes.
Grâce à ces différents tests, nous pouvons fixer des alertes au moment où le système devient trop critique.
Par exemple, au moment où je crée 10 VMs qui vont démarrer en même temps sur le Compute Node, Shinken passe la charge CPU en critique :



Ces plugins de récupération de charges développées par M. Sébastien Pasche et M. Jean Gabes (Fondateur de Shinken), permettent de **s'affranchir de différents agents** (**nrpe**) installés sur les machines testées. Cet agent va recevoir les commandes

envoyées par Shinken et les exécuter en local. Toutefois ceci nécessite de rajouter un programme sur les différentes machines physiques, tandis qu'avec ssh on utilise un protocole de communication déjà employé à travers mon infrastructure.

7.5 Mise en place de tests interdépendants

Précédemment, j'ai mis en place deux types tests : le premier teste la disponibilité des API et le second teste les ressources physiques. A l'heure actuelle, les tests que fait Shinken sont indépendants les uns des autres. Il serait intéressant d'**effectuer des successions de tests pour desseller à quel niveau se situerait le problème**. Par manque de temps, j'ai uniquement fait une analyse théorique des tests imbriqués.
Par exemple, si le Serveur d'API du Controller 2 ne répond pas au check_http et le check_RAM indique une erreur critique. On pourrait immédiatement savoir que le problème se trouve au niveau du serveur et que le problème réseau est à écarter, car un processus utilise toute la RAM. Grâce à ce type de tests interdépendants, en fonction du type d'erreur que Shinken remonte, il pourrait avertir soit la personne gérant le réseau, soit la personne en charge de la gestion des serveurs.

7.5.1 Finalité du test

Comme nous l'avons vu tout au long de ce projet, OpenStack est constitué d'une multitude de services dédiés à la gestion de l'infrastructure. Pour avoir une vue d'ensemble des services opérationnels, Shinken va monitorer grâce à une suite de tests dépendants, les éléments intervenant dans le démarrage d'une VM. Au final, nous devons être capable de savoir si notre infrastructure pourra :

- **Recevoir les commandes envoyées par l'administrateur** :
  - Tester si les API répondent aux requêtes http via l'IP virtuelle. Ce test est critique, car s'il y a une erreur, ceci veut dire que les deux Controllers sont en échec.
  - Tester sans passer par l'IP Virtuelle, si l'un des deux Controllers ne répond pas, ceci est moins critique grâce à la HA mise en place.
- **Tester les ressources physiques de chaque serveur**. Le degré de gravité change pour les types de serveurs. Un Compute Node sera plus chargé, que les deux Controllers. Par conséquent, si les Controller sont surchargés, il faudra y porter plus d'importance, car normalement ils n'hébergent que des services d'API peu gourmands en ressources.
- Tester si l'on peut écrire dans la DB MySQL.
- Tester si les différents services OpenStack sont démarrés.

Au final, nous devons être capable de tester toute la chaîne qui s'exécute au moment où l'administrateur décide de démarrer une VM.

Le schéma 27 représente la chaîne de tests que Shinken va effectuer afin de détecter la moindre erreur ou panne intervenant dans l'infrastructure.

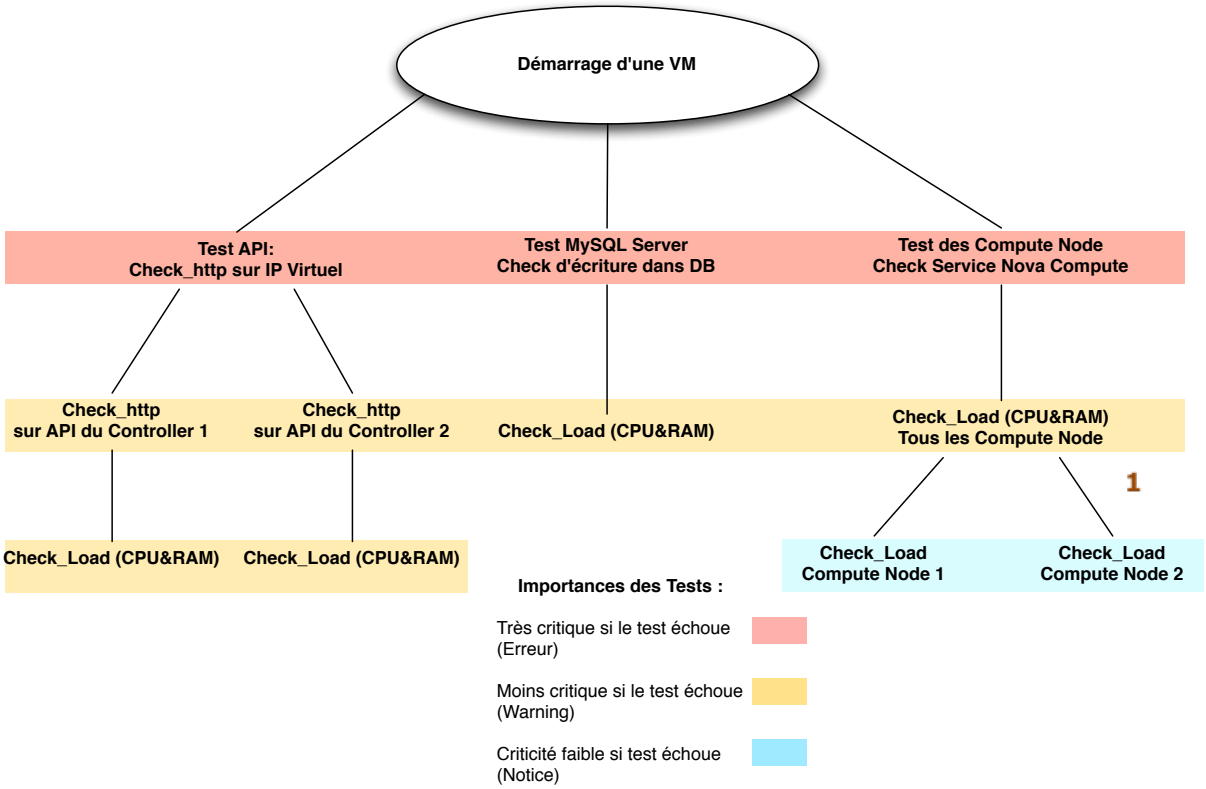


Schéma 27: Diagrammes de tests Shinken

Les tests s'effectuant dans la zone rouge sont critiques, si l'un d'entre eux échoue, aucune VM ne pourra être démarrée. Les tests dans la zone jaune sont moins critiques, mais, peuvent quand même présenter une anomalie. Prenons le test n°1, le « Check_Load Tous les comute Node » indique une erreur si tous les Compute Node sont chargés à 100%, ceci est important car plus aucune VM supplémentaire ne pourrait démarrer. Tandis que si un Compute Node est chargé à 100%, les VMs pourront toujours être démarrées sur le deuxième.

7.5.2 Plugin de démarrage d'une VM intégré à Shinken

Pour rendre notre système de supervision le plus complet possible, il serait intéressant de développer un plugin, afin que **Shinken puisse simuler le comportement de l'administrateur en démarrant une VM** et vérifier qu'elle est bien active. En intégrant ce test avec les tests du schéma 27, il sera alors possible de superviser tous les services de cette infrastructure en vue de déceler les moindres changements comportementaux du système. La chaîne de lancement du plugin va se dérouler comme suit:

1. Démarrer une VM avec le client python d'OpenStack
2. Attendre 5 secondes que la VM reçoive une IP
3. Récupérer l'IP de cette VM en interrogeant le fichier du serveur DHCP sur le Network Node contenant la corrélation : MAC, Nom_VM, IP
4. Au bout d'une dizaine de secondes, se connecter en ssh à la VM et effectuer un Ping sur un serveur distant. A travers ces deux tests, nous contrôlons si nous la VM est accessible par ssh et si le Network Node aiguille correctement les paquets du réseau privé vers le réseau public.

5. Détruire la VM grâce au client python, et regarder sur le QNAP que le disque virtuel sous forme de fichier est bien supprimé
6. Si l'ensemble des tests est OK, l'infrastructure OpenStack est fonctionnelle

7.6 Bilan du scénario 4

Avec une telle infrastructure, constituée de nombreux services répartis sur plusieurs serveurs physiques, il est essentiel de **mettre sur pieds un élément de supervision**, afin de remonter les différentes erreurs. Ce système va permettre **d'anticiper d'éventuelles pannes** critiques qui rendraient l'infrastructure inutilisable. Pour effectuer ces différents tests, il est nécessaire de connaître le comportement des différents services. Le fait d'intégrer de la HA, dédouble le nombre de serveurs physiques, ce qui va augmenter le nombre de tests. Par conséquent, il va être essentiel de prioriser les tests grâce aux dépendances, afin de ressortir des erreurs pertinentes.

8 Recommandations

Ce chapitre va mettre en place les bonnes pratiques afin d'avoir une infrastructure fonctionnelle, et garantissant certaines règles de sécurité.

8.1 Best Practises : Matériels physiques

Pour mettre en place le scénario 4, je recommande de répartir les différents services OpenStack sur 7 machines physiques au minimum :

- 2 Cloud Controllers : Hébergent les API Openstack et les Load Balancers, afin de garantir la HA
- 1 Network Node : Aiguille les paquets venant des VMs et vice versa, ainsi que la gestion des règles Iptables
- 1 Serveur MySQL : Héberge les bases de données pour les services Openstack et le serveur de message QPID
- 1 Serveur de Logs : Centraliser les logs des différents services Openstack ainsi que des Load Balancers
- 1 Compute Node : Ce type d'éléments peut être dupliqué en masse, en effet le nombre de VMs hébergées dépend du nombre d'hyperviseurs installés
- 1 Serveur NFS : Pour le stockage des Disques virtuels des VMs
- 1 Cible iSCSI : Cible iSCSI géré par Fedora avec l'ajout du service Cinder Volume

8.2 Bests Practises : Sécurité

8.2.1 Gestion des administrateurs sous OpenStack

Dans cette section, nous allons nous attarder sur la **gestion des administrateurs** sous OpenStack. Nous devons différencier deux types de comptes Administrateur :

- **Administrateur de l'infrastructure** : Il va gérer les ressources mises à disposition des VMs, comme la gestion des hyperviseurs, veiller au bon fonctionnement des services Openstack, ainsi que paramétrer les « Flavors » des VMs. Au final, cet administrateur doit veiller à ce que l'administrateur des VMs puisse en tout temps démarrer ces VMs. Dans OpenStack ce rôle est appelé « admin ». Pour simplifier, dans la suite du chapitre nous l'appellerons Admin.

- **Administrateur des VMs** : Il va démarrer les VMs, créer des SnapShots, ajouter et supprimer les disques de stockages par bloc (Cinder). Il va s'occuper de tout ce qui est en relation avec les VMs. Dans OpenStack ce rôle est appelé « member ».

Sur Openstack, l'administration des VMs via l'interface Web Horizon est découpée en projets. C'est-à-dire que chaque utilisateur ayant le rôle « member », est affilié à un projet. **Le compte Admin pour chaque projet peut définir des quotas**, comme par exemple, le nombre maximum de VMs que peuvent démarrer les Administrateurs de chaque projet. L'avantage de cette découpe en projet, est que nous pouvons avoir **un administrateur de VM par projet**, qui peut appliquer des configurations individuelles propre à son projet. Par exemple, les Security-Groups créer au sein d'un projet, ne seront pas communes à tous les projets. Dans notre schéma global, ces administrateurs se situent dans le réseau bleu d'administration. Tous ces utilisateurs sont gérés par le service Keystone hébergé sur le Cloud Controller.

Par défaut au moment de l'installation d'OpenStack, un compte administrateur de l'infrastructure est créé, et un projet rattaché à cet administrateur est aussi créé. Si notre administrateur n'est pas affilié à un projet on ne pourra pas se connecter sur l'interface Web[23]. Ceci ne pose pas vraiment de problème, car pour tester son infrastructure, Admin a besoin de voir si l'intégrité de la chaîne de lancement des VMs est garantie.

8.2.1.1 But du scénario

Le but est de créer deux projets :

- **Projet Production (Prj_Prod)** : Géré par l'administrateur « Admin_prod »
- **Projet Développement (Prj_Dev)** : Géré par l'administrateur « Admin_dev »

L'administrateur de l'infrastructure globale Admin, pourra voir les VMs exécutées sur l'infrastructure, mais il n'aura pas accès à toute la partie d'administration au sein du projet, qui est réservée à l'administrateur du projet.

[23] Affiliation à un Projet : https://bugs.launchpad.net/horizon/+bug/1099883

8.2.1.2 Rayon d'action des Administrateurs membres d'un projet

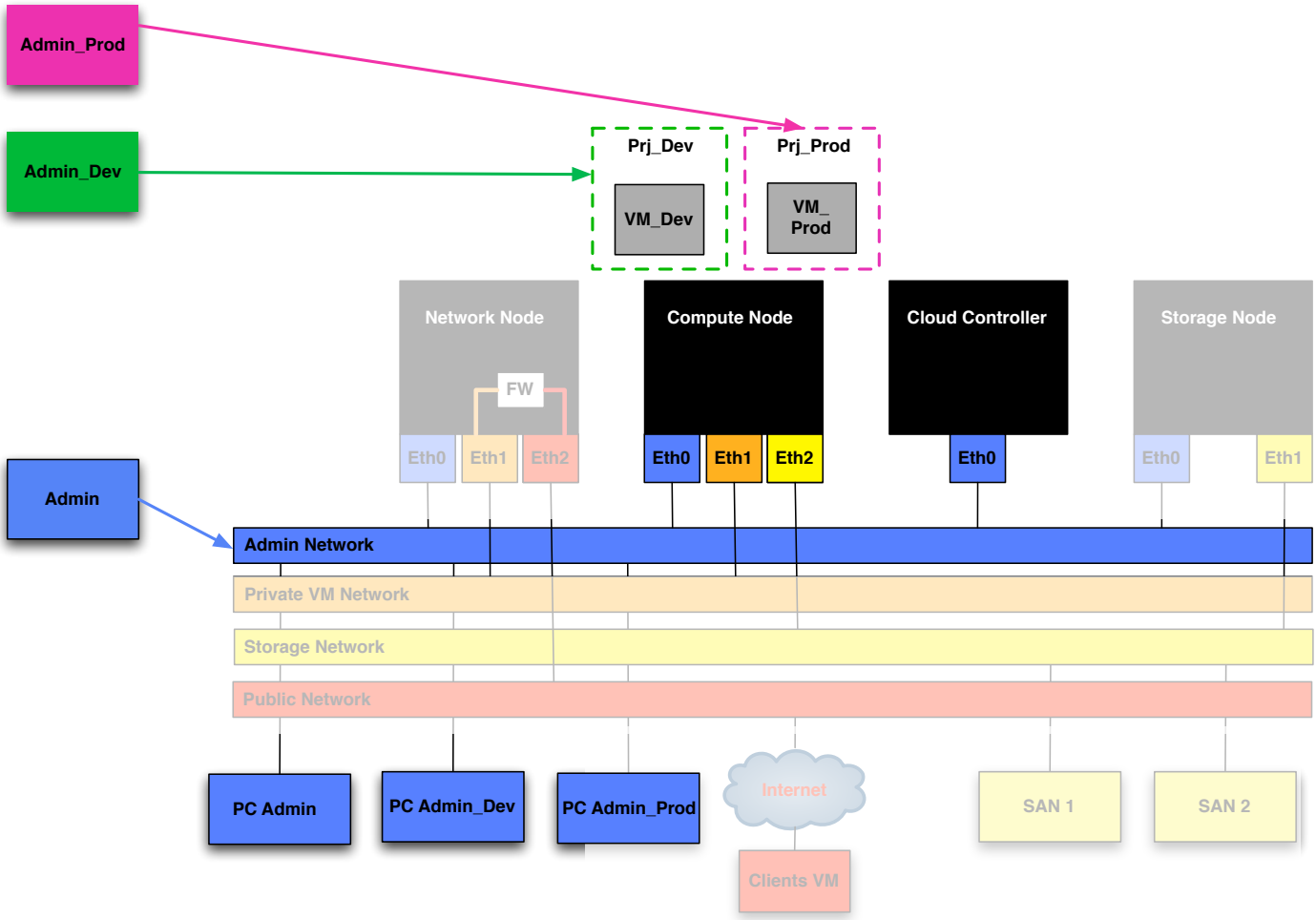


Schéma 28: Différents administrateurs

Le schéma 28 reprend le schéma 4 mais nous nous occupons uniquement du réseau d'administration. Voici ci-dessous quelques vues de l'interface horizon en fonction des différents administrateurs.

Vue de :

- Admin (Root)

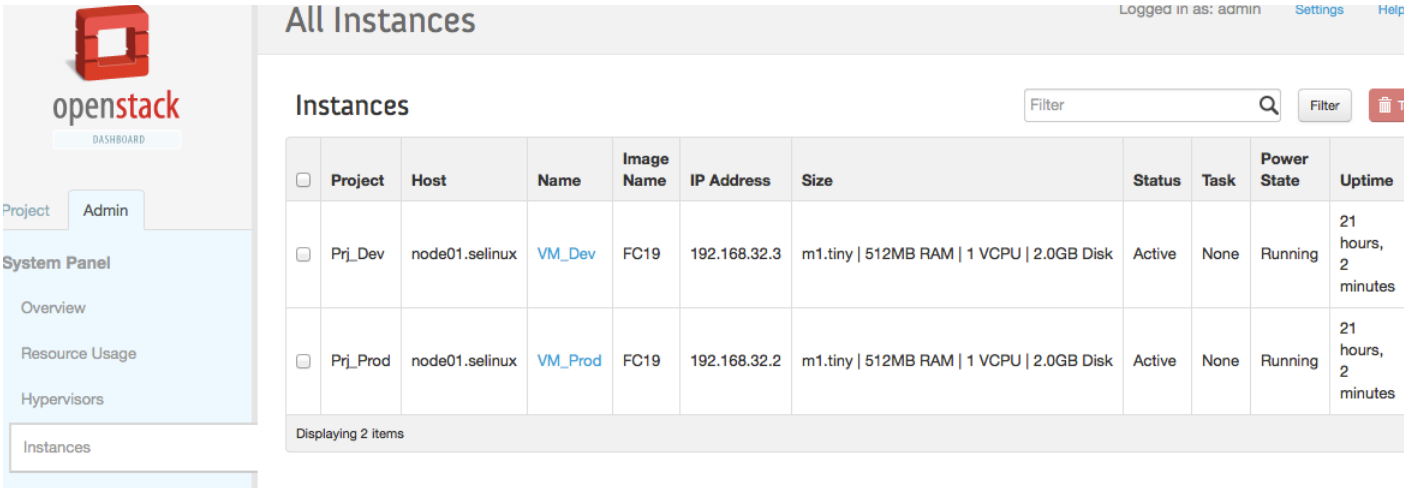


Figure 9: Vue de Admin

L'Admin peut voir l'ensemble des VMs, qui sont exécutées sur son infrastructure, et sur quel Compute Node elles sont exécutées. Toutefois il n'a **pas accès à l'administration des projets**, comme la possibilité d'effectuer des SnapShot. Il peut supprimer les VMs, ce qui est intéressant si l'une d'entre elles venait à présenter un risque pour l'infrastructure.

Il peut voir également les ressources physiques des Compute Node :

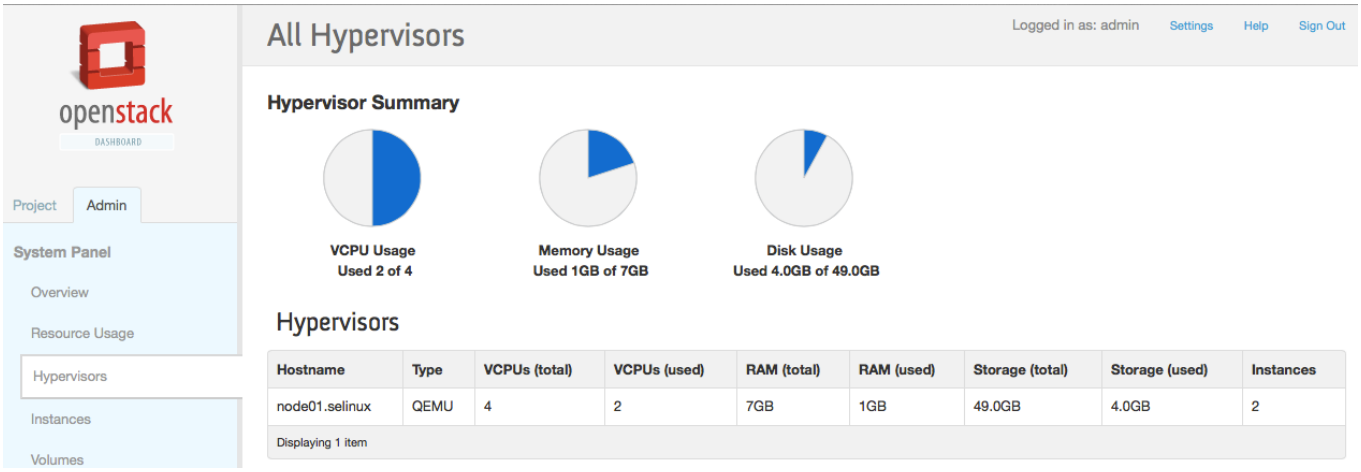


Figure 10: Vue des ressources physiques

- Admin_Dev :

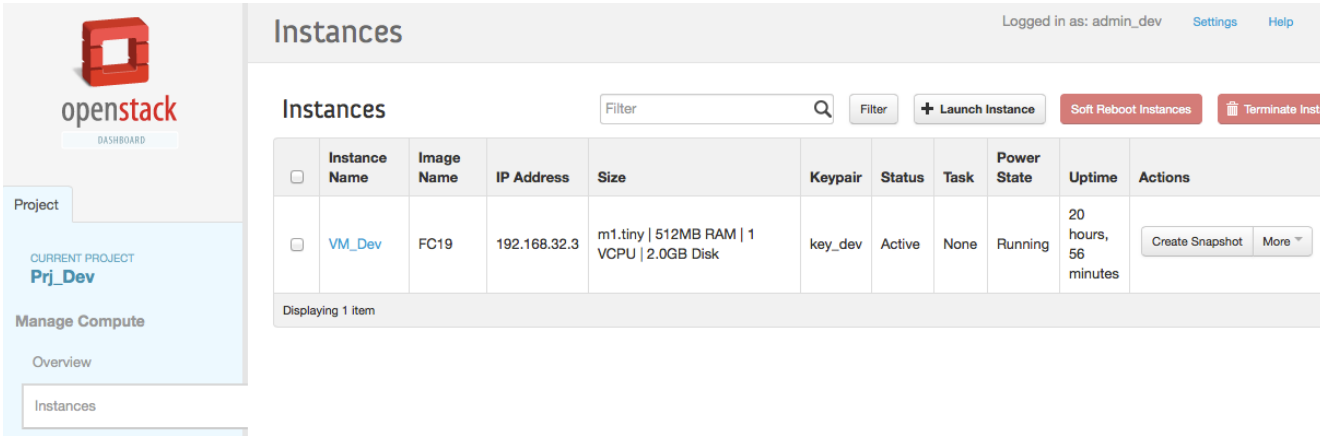


Figure 11: Vue de l'Admin_Dev

Il possède uniquement un accès au projet Prj_Dev. Comme montré ci-dessus, l'administrateur de ce projet ne peut pas voir sur quel Compute Node est exécuté sa VM. Nous avons un niveau d'abstraction supplémentaire.

- Admin_Prod :

Sa vue sera similaire à Admin_Prod, mais ne concerne que les VMs de son projet :

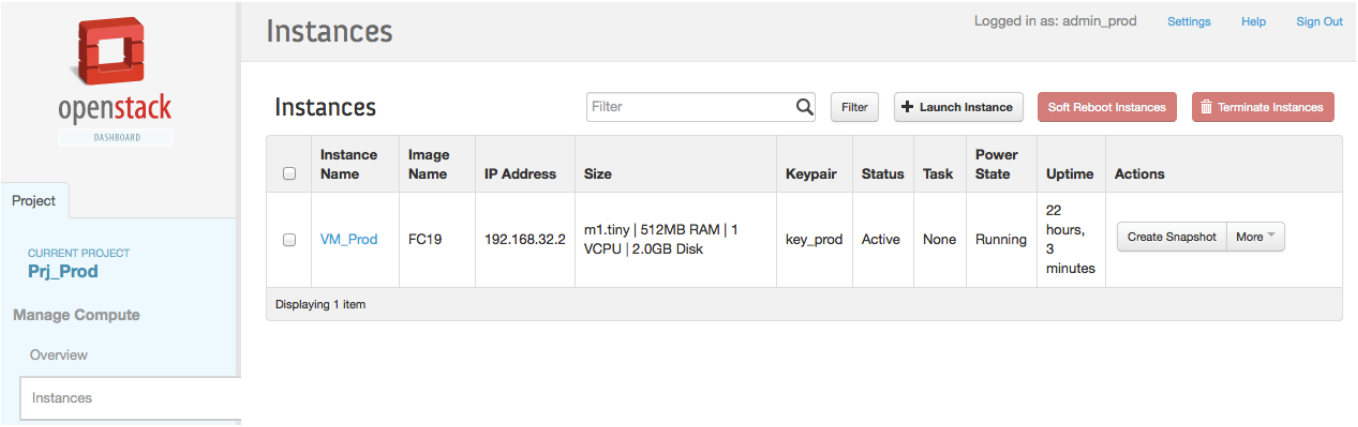


Figure 12: Vue Admin_Prod

8.2.2 Bilan du scénario

Au final, nous avons pu **cloisonner les tâches de chaque administrateur**, grâce à deux types d'administrateurs.

Avec la collaboration de M. Khaled Basbous, nous avons essayé d'obtenir une granularité plus fine dans la gestion des droits d'accès, au sein d'un même projet. Comme par exemple, nous aurions aimé avoir un admin_dev_images, qui s'occuperait uniquement des Snaphsots, ou un autre admin_dev_vm, qui s'occuperait de démarrer ou arrêter les VMs. Pour cela, l'interface horizon ne pouvait pas remplir se type de configuration. Mais en regardant dans les fichiers de configurations des différents services OpenStack, nous avons vu la présence d'un fichier policy.json, répertoriant toutes les actions que l'on peut faire avec le service. Les deux services sur lesquelles nous nous sommes attardés, ont été le service Nova et Glance (Gestion des Images et des Snapshots).

Voici un extrait du service Nova :

```
"compute_extension:admin_actions:pause": "rule:admin_or_owner",
"compute_extension:admin_actions:unpause": "rule:admin_or_owner",
"compute_extension:admin_actions:suspend": "rule:admin_or_owner",
"compute_extension:admin_actions:resume": "rule:admin_or_owner",
```

Pour avoir un exemple complet du fichier : https://github.com/hepia-telecom-labs/Openstack_Config/blob/master/Object_config/HA/Controller1/Nova/policy.json

La documentation fournie par OpenStack à propos des policy.json est très sommaire. En effectuant divers tests, nous sommes parvenus à créer un administrateur qui pouvait uniquement mettre en pause la VM, et un autre qui pouvait la remettre en activité. Mais nous n'avons pas réussi à cloisonner la création de Snapshot, car contrairement au service Nova où toutes les actions sont répertoriées, le fichier policy.json de Glance ne répertorie aucune action :
https://github.com/hepia-telecom-labs/Openstack_Config/blob/master/Object_config/HA/Controller1/Glance/policy.json
Ceci limite grandement le cloisonnement des rôles.

Par conséquent, le problème se situe au démarrage d'une VM : nous avons besoin de ces deux services, au final le cloisonnent a pu se faire uniquement au niveau du service

Nova. Le fait de créer deux administrateurs, un qui met en pause les VMs et l'autre qui les réveille, n'a pas un grand intérêt au niveau du cloisonnement des rôles.

Comme OpenStack est en constante évolution, je ne doute pas que dans les prochaines versions, de nouvelles fonctionnalités au niveau de la gestion des rôles et des utilisateurs seront intégrées à Horizon.

8.2.3 Configuration des Logs

Dans une configuration par défaut d'OpenStack, tous les services sont en mode Debug. Ce mode va enregistrer tous les messages échangés entre les différents services au sein des différents fichiers de Log. Dans un premier temps, c'est un réel avantage de posséder cette mine d'information, afin de connaître les interactions entre les différents services. Cependant, **en production, il faudrait au maximum limiter la verbosité des services**, afin d'éviter qu'un attaquant récupère le maximum d'information sur notre infrastructure, pour cela il faudra éditer les fichiers de configuration des services afin de désactiver ce mode Debug.

9 Conclusion

9.1 Conclusion sur OpenStack

Avec l'avènement des moyens de communication qui n'ont cessé d'accroitre leurs débits, depuis moins d'une décennie, nous avons vu fleurir à la surface du globe de gigantesques datacenters centralisant des térabits de données ainsi que d'importantes puissances de calcul. Ces ressources permettent ainsi, aux utilisateurs d'avoir accès à leurs données en permanences et sans limite géographiques. Pour gérer ces fermes constituées de dizaines, voir de centaines de serveurs, hébergeant des centaines de machines virtuelles, il est essentiel de mettre en place différents mécanismes de gestion de la virtualisation et supervision. Ce travail de Master s'est concentré sur différentes facettes qui permettent de construire une infrastructure virtualisée, à la fois, fiable, performante et évolutive. Il s'est articulé autour de **3 grands thèmes** :

- La gestion d'un Cloud privé avec OpenStack
- La haute disponibilité
- La supervision du système

Ces trois domaines représentent les trois piliers fondamentaux sur lesquels mon infrastructure s'est basée. Le fait de les mettre en relation, a permis de minimiser les risques tels que l'indisponibilité des services de gestion à cause qu'un des deux Cloud Controllers subissent un arrêt complet suite à une coupure de courant, tout en ayant des performances accrues.

9.1.1 Evolution d'OpenStack

Le système OpenStack a pour but de gérer l'administration des VMs. Etant un **produit Open Source**, il est poussé par une vaste communauté le faisant évoluer continuellement. Mais cette constante évolution à son revers de médaille, du fait que **chaque nouvelle version change radicalement**, ce qui nécessite une mise à niveau de l'ensemble de l'architecture. Par conséquent, les services doivent posséder la même version. J'ai expérimenté cette hétérogénéité de versions, en installant un Compute Node avec le service nova-compute d'une version plus évolué que le service Nova-API du Controller, ce qui provoqua une erreur des compréhensions entre les messages AMQP envoyés par le Controller au Compute Node. En effet, le problème d'une mise à niveau globale nécessite de mettre en place une infrastructure test afin de valider les nouvelles fonctionnalités et effectuer les examens de fiabilité.

9.1.2 Composition de l'infrastructure

Comme traité dans ce travail, l'infrastructure mise en place est constituée de nombreux services assurant une fonction bien précise. De par la **lourdeur en terme d'interaction entre ces différents services**, pour des entreprises désirant héberger quelques VMs tout en restant dans le monde Open Source, je préconiserai **OpenNebula**. Cette solution a pour avantage d'être plus légère tout en proposant évidemment moins de fonctionnalités. Avec l'émergence des Cloud publiques, une entreprise peut, avec un investissement minimum, démarrer une VM exécutant ces services Web, sans avoir le moindre serveur physique dans ses locaux. Voici une liste non exhaustive de leader du Cloud :

- Green.ch : Société suisse proposant une solution d'hébergement de serveurs virtualisés basée sur OpenStack
- OVH : Société française proposant un Cloud public basé sur VMware, mais depuis quelques mois sa plateforme de stockage en ligne Hubic (même fonctionnement que DropBox) est hébergée au sein de leur Cloud OpenStack
- Amazon Elastic Compute Cloud: Leader mondial du Cloud public

Pour une entreprise telle que leShop.ch basant son business model sur le e-commerce, il est nécessaire qu'elle ait sa propre infrastructure afin de pouvoir gérer finement l'ensemble des rouages, dans le but d'augmenter les performances et la fiabilité. Si l'on quitte le domaine OpenSource, Microsoft grâce à sa couche de virtualisation Hyper-v est une bonne alternative à la mise d'un Cloud privé. Le point fort de cette solution est qu'elle s'intègre dans le système d'exploitation Windows Server, et donc il n'est pas utile de rajouter un autre type d'OS, contrairement à VMware où les hyperviseurs fonctionnent sous un OS qui leur est propre qui est ESX.

9.1.3 Faiblesse de la haute disponibilité

Dans une architecture sensible comme celle mise en place et devant être opérationnelle à tout moment, il est essentiel de mettre sur pied une solution de haute disponibilité, permettant de minimiser les risques, tout en sachant que le risque 0 n'existe pas. Malgré que les services OpenStack soient Stateless, il aurait été intéressant qu'un service propre à OpenStack gère l'état de la HA des services en soulevant des alertes.
Ceci amène à un autre point essentiel : la supervision des ressources. Dans des infrastructures constituées de nombreux éléments, le maître-mot est l'anticipation. Le fait d'anticiper par exemple, la surexploitation d'un hyperviseur, ou la saturation des volumes de stockage, vont permettre de ne pas conduire notre système dans une situation de blocage. Dans cette gestion, **OpenStack ne propose pas de réelle solution de monitoring**, malgré que les services des Compute Node remontent au Controller leurs informations à propos des ressources, l'administrateur ne peut pas l'exploiter de manière intuitive sauf en allant interroger la DB. Ceilometer qui est apparu avec la nouvelle version d'OpenStack propose une API de récupération des métriques, mais elle est encore à l'état embryonnaire. Pour palier à ce manque, j'ai installé Shinken qui va permettre d'effectuer des tests avec une granularité plus fine.
La solution de gestion des Logs, est également très utile pour déceler les irrégularités de fonctionnement des différents services. Grâce au formatage en objets JSON, nous pouvons extraire les données utiles sans avoir à parcourir des volumes impressionnants à la recherche d'éventuelles erreurs.

9.2 Gestion du travail de Master

Ce travail de master s'est déroulé sur 18 semaines. Tout au long du travail, j'ai appliqué une méthode agile matérialisée par des cycles d'itération de deux-trois semaines. A la fin de chaque itération une fonctionnalité supplémentaire était étudiée, mise en place et testée. Au fur et à mesure que mon projet avançait, mon infrastructure devenait plus stable et plus performante afin qu'elle puisse s'intégrer dans un univers professionnel. Cette technique de gestion de projet appuyée par des cycles d'itération est basée sur la

méthodologie SCRUM[24]. Cette gestion de projet nécessite une forte implication de la part du Product Owner (Initiateur du projet). Ce rôle était représenté par M. Sébastien Pasche, qui fixait en début de chaque sprint (itération) les objectifs que devait remplir le livrable.

Activités	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Etude théorique d'OpenStack																		
Etude et réalisation du scénario 1a																		
Etude et réalisation du scénario 1b																		
Etude et mise en place de la HA																		
Etude de la supervision avec Shinken																		
Etude et mise en place de la gestion des Logs																		
Rédaction du mémoire																		

Figure 13: Déroulement du travail de Master

A un niveau plus technique, en ce qui concerne la gestion de mes fichiers de configurations et mes scripts, j'ai utilisé un gestionnaire de version géré par Git. Grâce au stockage des historiques de versions je pouvais revenir en arrière dans mes configurations ou voir mes dernières modifications en travaillant toujours sur le même fichier. Les modifications apportées au sein de mes fichiers sont appelées des « comit ». Exemple d'un commit effectués sur le script python permettant l'automatisation d'installation d'une VM :
https://github.com/hepia-telecom-labs/Openstack_Config/commit/6e346b5c99258a8bc734237b254093384dbd3542

[24] Méthodologie Scrum : http://ineumann.developpez.com/tutoriels/alm/agile_scrum/

9.3 Conclusion Personnelle

A travers ce projet, j'ai pleinement pu approfondir les différentes facettes qui constituent une infrastructure virtualisée pouvant gérer un grand nombre de serveurs. Voici un diagramme représentant les notions que j'ai abordées en fonction du niveau d'approfondissement et du degré d'ouverture par rapport au fil rouge du projet :

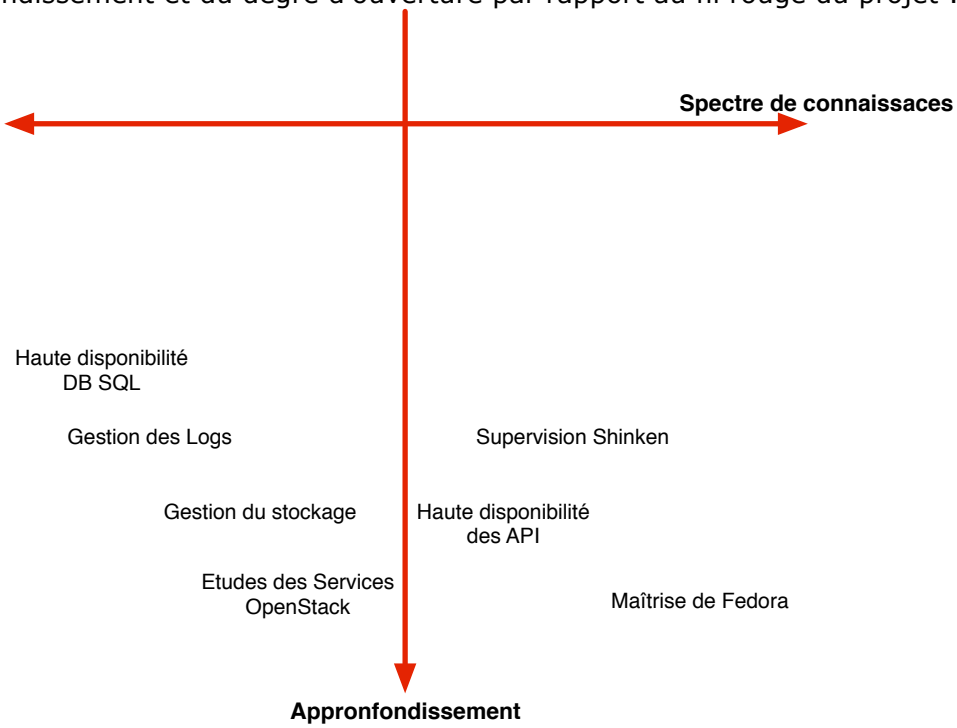


Schéma 29: Diagramme des connaissances

La latitude représente le degré d'approfondissement de la notion, comme on peut le voir les services OpenStack on été beaucoup approfondies, c'est-à-dire que j'ai été parcourir la documentation, analyser les Logs générés par les services et étudié le code source. La longitude représente l'éloignement de la notion par rapport au thème principale (Ligne verticale rouge), par exemple l'étude de la haute disponibilité SQL a été analysée en théorie mais sa mise en pratique était hors du cadre du projet. Sachant que mon environnement était basé sur Fedora, mes connaissances dans les systèmes Linux ont fait un réel bond en avant.
Ce qui m'a vraiment plu en faisant ce travail à été de pouvoir fusionner toutes mes connaissances, à la fois dans le domaine des réseaux, de la sécurité et du développement avec la mise sur pied de divers scripts.
La fonction d'ingénieur système au sein de Datacenters opérant sous OpenStack est loin d'être une tâche aisée, je comprends que les départements IT d'entreprises voulant mettre en place un Cloud privé, concluent des partenariats avec de grands acteurs de l'hébergement tel que eNovance[25] pour du supports ou du consulting.

Malgré que OpenStack soit Open Source, au cours de mes différentes recherches, j'ai trouvé de nombreuses sociétés, vendant des outils de déploiement tel que la société

[25] eNovance (Société française spécialisé dans le consulting de déploiement d'OpenStack) : http://www.enovance.com/fr/produits-solutions/opencloud-opensource/opencloud/openstack

Mirantis avec Fuel[26]. Avec de bonnes connaissances dans le domaine Open Source, il est possible de faire du commerce avec notre savoir-faire.

9.4 Perspectives futures

A travers ce travail de Master, je n'ai évidemment pas couvert tous les domaines qu'offre ce genre d'architecture.
Avec l'arrivée des réseaux SDN (Software Defined Network), un nouveau service remplaçant Nova-Network, appelé Neutron permet de gérer le réseau virtuel des VMs à l'aide de switch virtuel gérer par OpenvSwitch grâce au protocole OpenFlow.

Au niveau des couches du Cloud, mon travail est resté au niveau de la couche IaaS (Infrastructure as a Service), mais par la suite, il serait intéressant d'**étudier le fait d'ajouter une couche PaaS** (Plateform as a Service) dans le but de simplifier la tâche de développeurs.
Jusqu'à présent, ma plateforme offre des VMs visibles depuis Internet, mais l'administrateur ou le développer Web doivent les démarrer eux-mêmes, installer les différents packages, les configurer et à partir de ce moment-là, ils pourront envoyer leur code html, Java ou Python. Malgré que toute la configuration puisse être « scriptée », cette tâche est longue. Pour palier à ce problème, il est possible de créer un niveau d'abstraction supplémentaire, en installant la couche PaaS pouvant être gérée par un service fourni par RedHat qui est OpenShift[27]. Par la suite, le développeur choisira via une interface Web ou CLI, ses environnements de développement (compilateur Java, Python…) et enverra son code via un dépôt Git directement au compilateur ou au serveur Web. La gestion des VMs sera entièrement gérée par la couche PaaS qui dialoguera avec OpenStack afin de démarrer de nouvelles VMs contenant un serveur Web ou un load balancer en vue d'augmenter les capacités de ses service Web. Le développeur n'aura plus d'interaction avec la VM en elle même.

[26] Fuel Mirantis : http://software.mirantis.com/key-related-openstack-projects/project-fuel/
[27] OpenShift : https://www.openshift.com/walkthrough/developer-workflow

10 Bibliographie

- https://github.com/hepia-telecom-labs/Openstack_Config/tree/master/Object_config/HA : Mon dépôt GitHub regroupant tous mes fichiers de configuration et scripts que j'ai utilisés durant ce travail
- Schéma n°1 : Schéma extrait d'un document écris par M. Khaled Basbous à propos de la sécurité dans un environnement virtualisé
- http://docs.openstack.org/user-guide/user-guide.pdf : Guide de l'utilisateur afin d'utiliser au mieux OpenStack
- http://openstack.redhat.com/Quickstart : Guide d'installation d'OpenStack sur une seule machine physique grâce à PackStack
- http://www.mirantis.com/blog/openstack-networking-single-host-flatdhcpmanager/ : Explication du fonctionnement du Network Node
- http://ken.pepple.info/openstack/2011/04/22/openstack-nova-architecture/ : Découpage de l'architecture OpenStack
- https://wiki.openstack.org/w/images/3/3b/Cinder-grizzly-deep-dive-pub.pdf : Présentation de Cinder
- http://www.mirantis.com/blog/understanding-openstack-authentication-keystone-pki/ : Fonctionnement de Keystone
- http://www.mirantis.com/blog/software-high-availability-load-balancing-openstack-cloud-api-servic/ : Fonctionnement de la HA des API OpenStack
- http://indico.cern.ch/getFile.py/access?contribId=57&sessionId=1&resId=0&materialId=slides&confId=220443 : Présentation des différentes manières de stocker les Logs

A Annexes

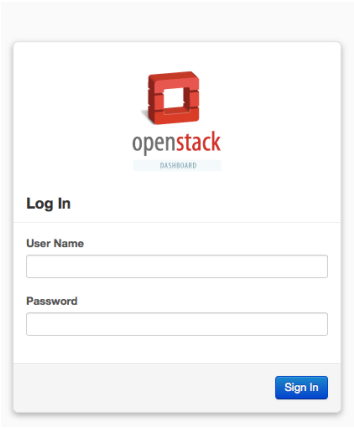
A.1 Démarrage d'une VM avec Horizon

Démarrer une VM via GUI qui hébergera un serveur Web interrogeable depuis Internet.

Login sur machine physique :
User : labotd
Password : labolabo

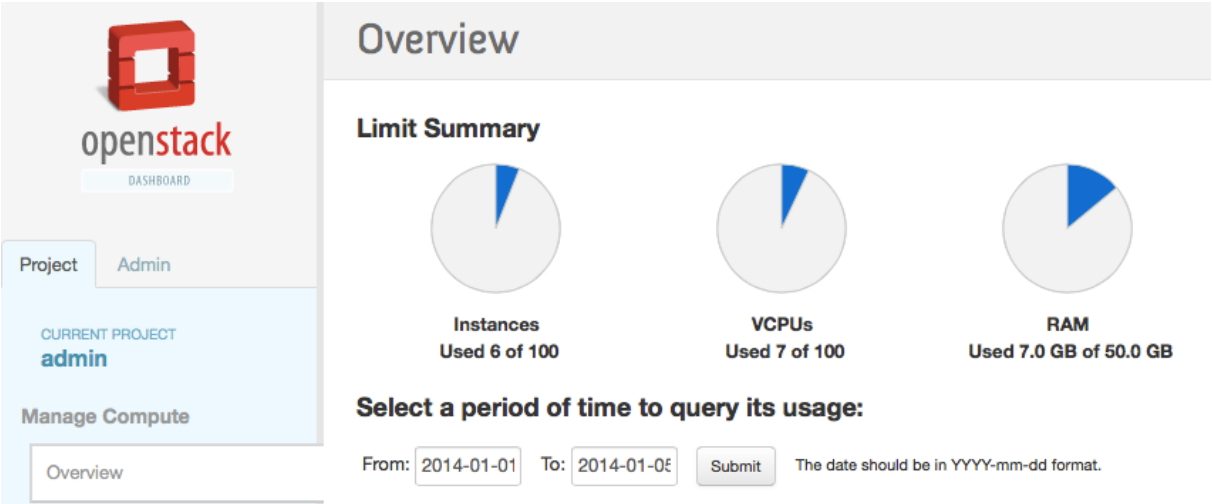
A.1.1 Page d'administration Horizon

1. Ouvrir le Browser Firefox
2. Ouvrir: **http://10.2.4.100**



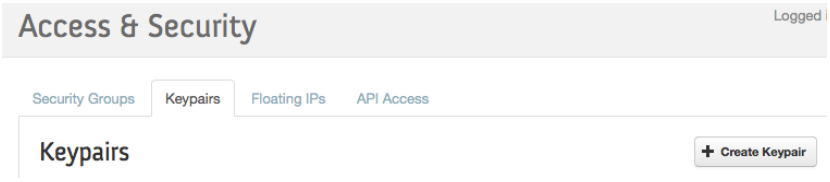
3. Se loguer avec vos identifiants de Team :
   Exp pour team 1:
   Username -> team1
   Password -> team1

4. Redirection vers la page **Project**

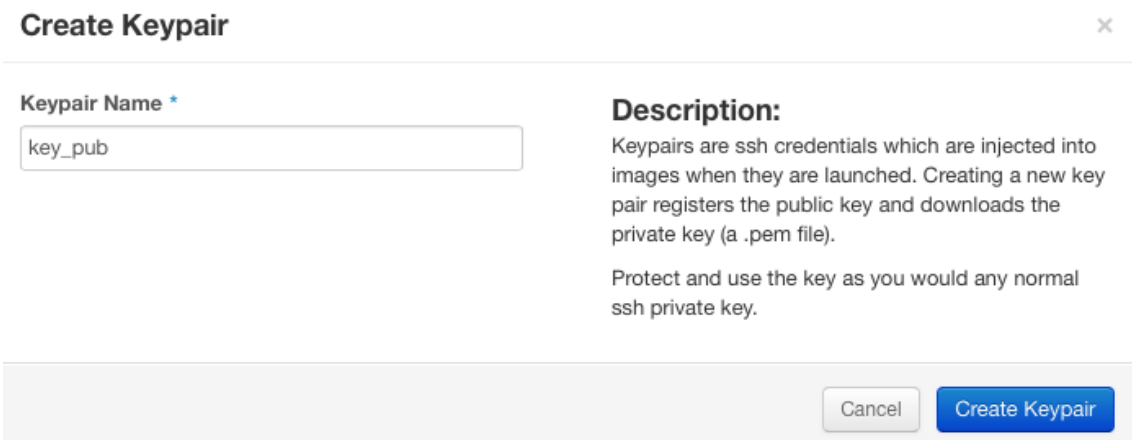


A.1.2 SSH

5. Cliquer sur **Access and Security** et **Keypairs**, génération des paires de clés ssh, pour connecter à distance à la VM une fois créée



6. Cliquer sur **Create Keypairs** et rentrer comme **nom de clés** : **key_pub**

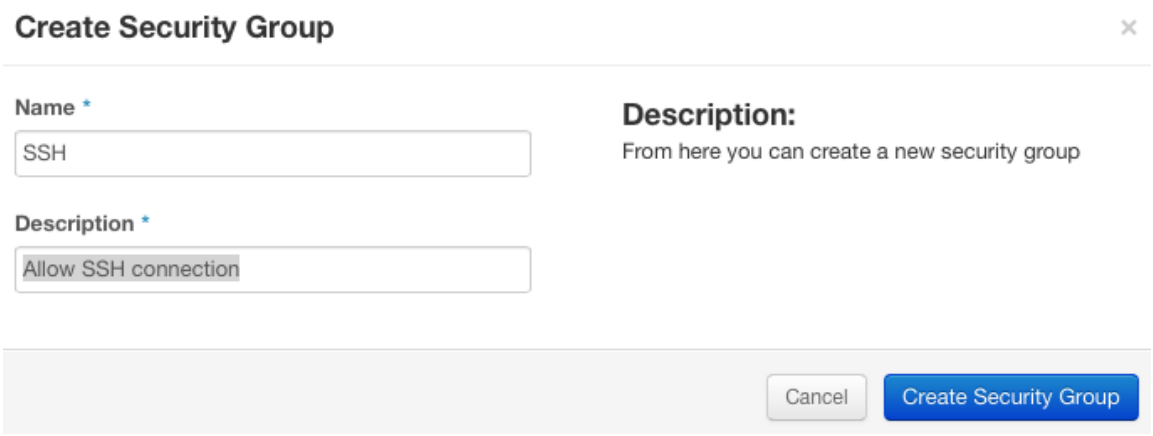


Cliquer sur **Create Keypair**
Le PoPup de téléchargement s'ouvre cliquer sur **Save File** et **OK**,
La clé publique sera enregistrée : ***/home/labotd/Downloads/key_pub.pem***

A.1.3 Firewall

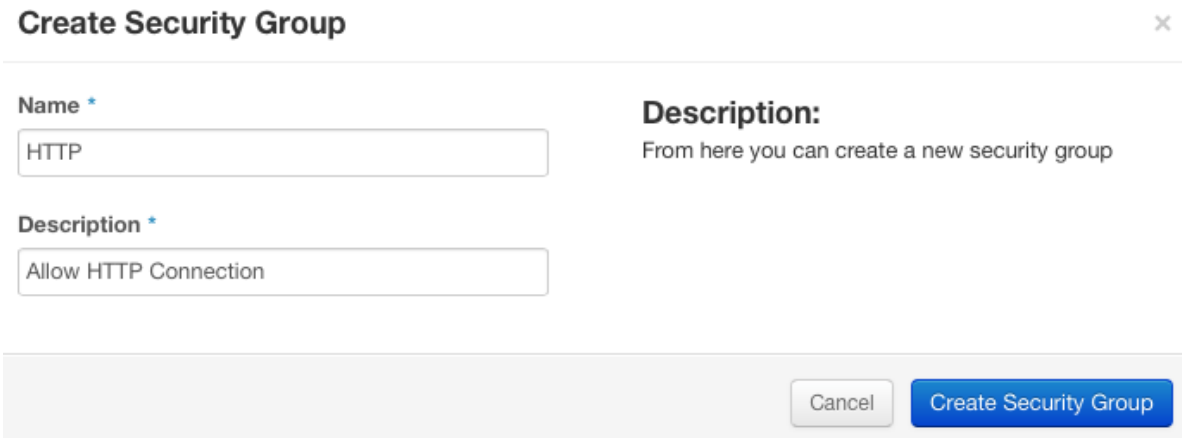
7. Cliquer sur **Access & Security** et **Security Groups**

8. Cliquer sur **Create Security Group**
   Création de deux groupes :
   - SSH : Se connecter en SSH à la VM (Port 22)
     Name : SSH
     Description : Allow SSH connection



Cliquer sur **Create Security Group**

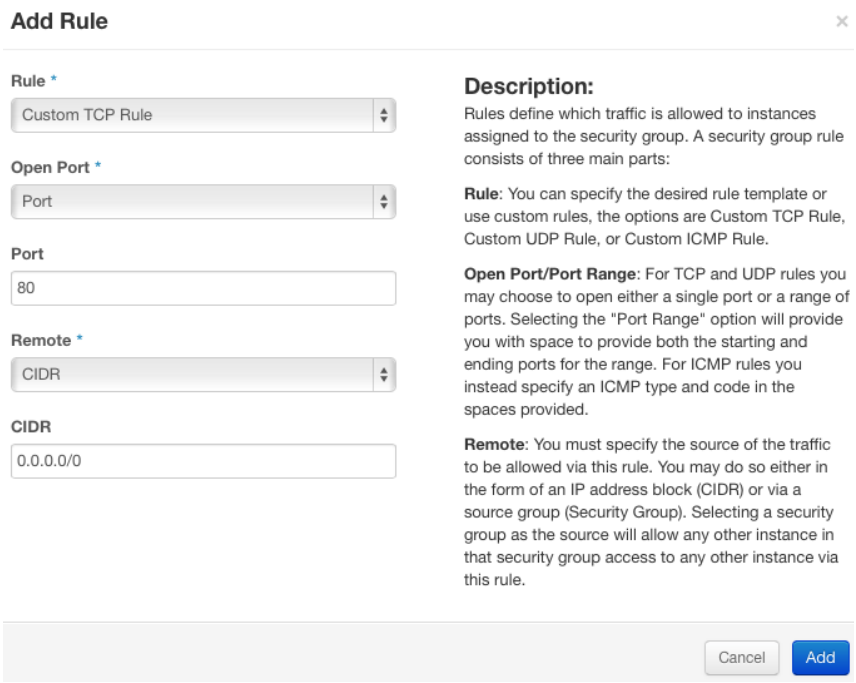
- HTTP: Se connecter au Port 80 de la VM
  Name : HTTP
  Description : Allow HTTP connection



Cliquer sur **Create Security Group**

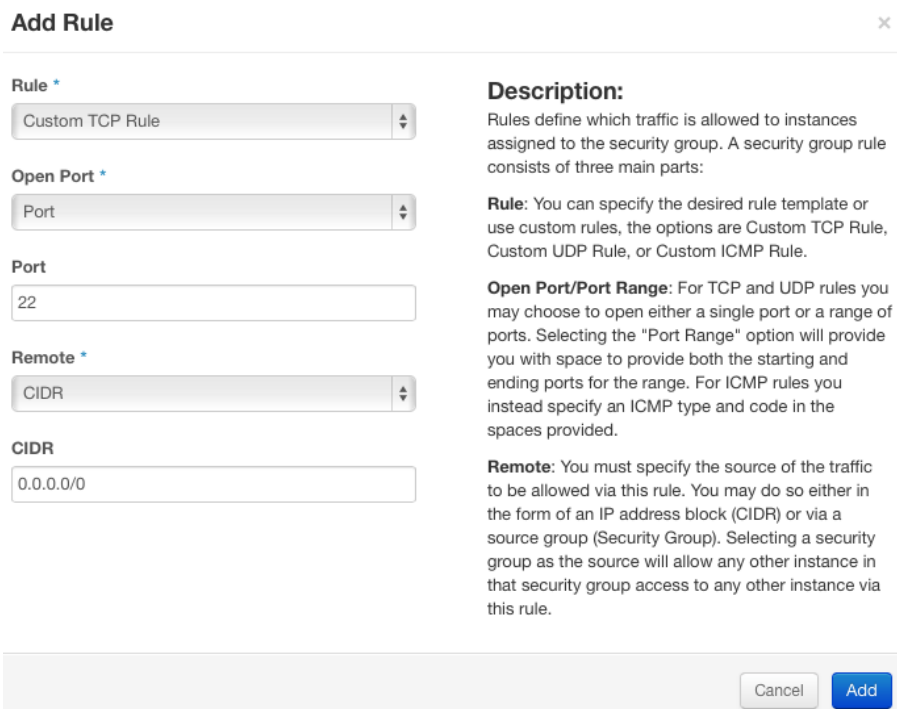
9. Pour la **security-group HTTP**, éditer les règles de firewall en cliquant sur **Edit Rules** (Colonne Action)

10. Cliquer sur **Add Rule** et remplir les champs comme ci-dessous :



Cliquer sur **Add**

11. Cliquer sur **Access & Security**, cliquer sur **Edit Rule** pour le group **SSH**
12. Cliquer sur **Add Rule** et remplir les champs comme ci-dessous :



Cliquer sur **Add**

A.1.4 DB Nova

Les **Security-Group** et les **rules** sont stockés sur le serveur **SQL physique** (Voir Slide 4).
Un extrait des tables de la **DB Nova** :

security_group_rules

created_at	updated_at	deleted_at	id	name	description	user_id
2014-01-14 15:31:27	NULL	NULL	22	HTTP	Allow HTTP Connection	5b61dca810674f8a9c1d9a25e4c6...
2014-01-14 15:31:07	NULL	NULL	21	SSH	Allow SSH Connection	5b61dca810674f8a9c1d9a25e4c6...
2014-01-14 15:28:40	NULL	NULL	20	default	default	5b61dca810674f8a9c1d9a25e4c6...

security_groups

created_at	updated_at	deleted_at	id	parent_group_id	protocol	from_port	to_port	cidr	group_id	deleted
2014-01-14 15:33:38	NULL	NULL	25	21	tcp	22	22	0.0.0.0/0	NULL	0
2014-01-14 15:32:09	NULL	NULL	24	22	tcp	80	80	0.0.0.0/0	NULL	0

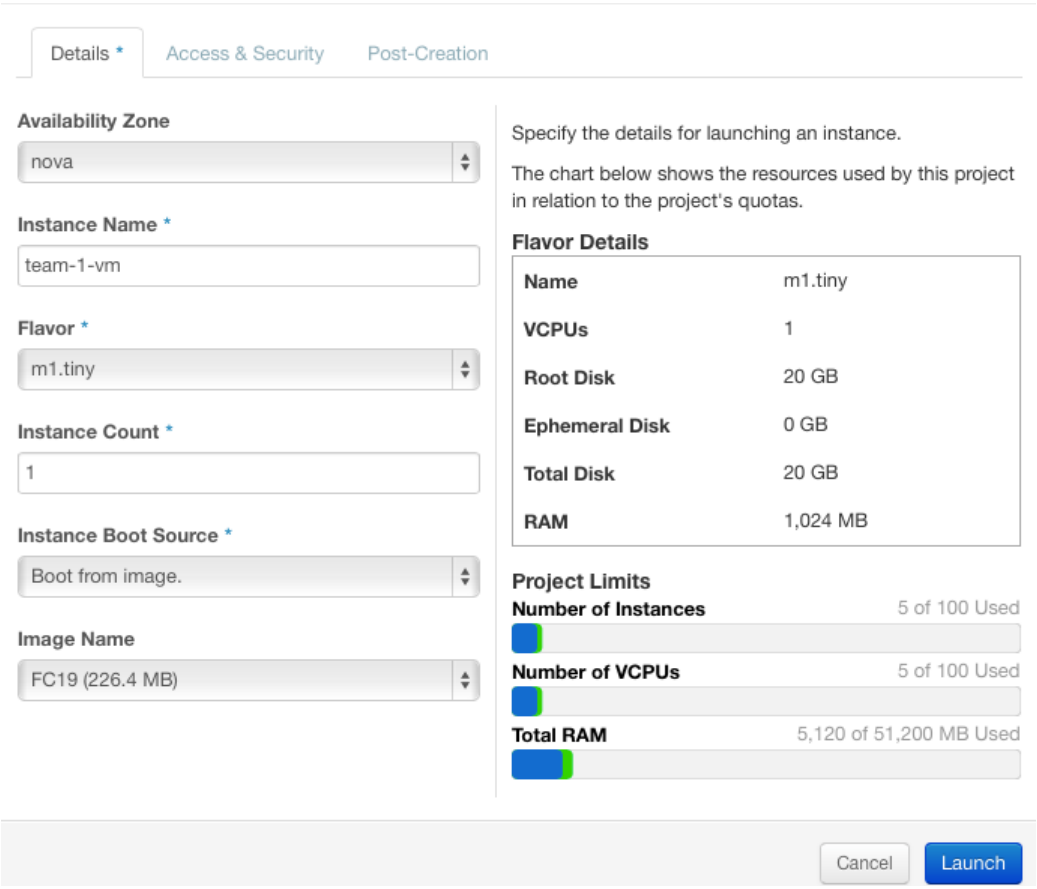
A.1.5 Démarrage de la VM

13. Cliquer sur **Images & Snapshots**

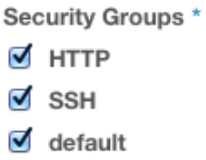
Nous avons une Image au format QCOW2 : FC19

14. Cliquer sur **Launch**

15. Une fenêtre s'ouvre, indiquer :
    - (Instances Name) Le Nom de la VM : **team-#_de_team-vm** (Exp : Pour Team 1 -> team-1-vm)



16. Cliquer sur **Access & Security**
17. Cocher **HTTP** et **SSH** pour appliquer les security-groups précédemment créés



Cliquer sur **Launch**
Redirection automatique sur la page **Instances**

A.2 Client Python pour l'administration d'OpenStack

Client Python pour démarrer une VM en CLI, voici la liste des paramètres du client Nova :

```
nova :
Positional arguments:
  <subcommand>
    absolute-limits     Print a list of absolute limits for a user
    actions             Retrieve server actions.
    add-fixed-ip        Add new IP address to network.
    add-floating-ip     Add a floating IP address to a server.
    aggregate-add-host  Add the host to the specified aggregate.
    aggregate-create    Create a new aggregate with the specified details.
    aggregate-delete    Delete the aggregate by its id.
    aggregate-details   Show details of the specified aggregate.
    aggregate-list      Print a list of all aggregates.
    aggregate-remove-host
                        Remove the specified host from the specified aggregate.
    aggregate-set-metadata
                        Update the metadata associated with the aggregate.
    aggregate-update    Update the aggregate's name and optionally
                        availability zone.
    boot                Boot a new server.
    console-log         Get console log output of a server.
    credentials         Show user credentials returned from auth
    delete              Immediately shut down and delete a server.
    describe-resource   Show details about a resource
    diagnostics         Retrieve server diagnostics.
    dns-create          Create a DNS entry for domain, name and ip.
    dns-create-private-domain
                        Create the specified DNS domain.
    dns-create-public-domain
                        Create the specified DNS domain.
    dns-delete          Delete the specified DNS entry.
    dns-delete-domain   Delete the specified DNS domain.
    dns-domains         Print a list of available dns domains.
    dns-list            List current DNS entries for domain and ip or domain
                        and name.
    endpoints           Discover endpoints that get returned from the
                        authenticate services
    evacuate            Evacuate a server from failed host
    flavor-create       Create a new flavor.
    flavor-delete       Delete a specific flavor.
    flavor-list         Print a list of available 'flavors' (sizes of
```

```
                    servers).
flavor-show         Show details about the given flavor.
flavor-key          Set or unset extra_spec for a flavor.
flavor-access-list  Print access information about the given flavor.
flavor-access-add   Add flavor access for the given tenant.
flavor-access-remove
                    Remove flavor access for the given tenant.
floating-ip-create  Allocate a floating IP for the current tenant.
floating-ip-delete  De-allocate a floating IP.
floating-ip-list    List floating ips for this tenant.
floating-ip-pool-list
                    List all floating ip pools.
get-vnc-console     Get a vnc console to a server.
get-spice-console   Get a spice console to a server.
host-action         Perform a power action on a host.
host-update         Update host settings.
image-create        Create a new image by taking a snapshot of a running
                    server.
image-delete        Delete an image.
image-list          Print a list of available images to boot from.
image-meta          Set or Delete metadata on an image.
image-show          Show details about the given image.
keypair-add         Create a new key pair for use with instances
keypair-delete      Delete keypair by its name
keypair-list        Print a list of keypairs for a user
list                List active servers.
live-migration      Migrates a running instance to a new machine.
lock                Lock a server.
meta                Set or Delete metadata on a server.
migrate             Migrate a server.
pause               Pause a server.
rate-limits         Print a list of rate limits for a user
reboot              Reboot a server.
rebuild             Shutdown, re-image, and re-boot a server.
remove-fixed-ip     Remove an IP address from a server.
remove-floating-ip  Remove a floating IP address from a server.
rename              Rename a server.
rescue              Rescue a server.
resize              Resize a server.
resize-confirm      Confirm a previous resize.
resize-revert       Revert a previous resize (and return to the previous
                    VM).
resume              Resume a server.
root-password       Change the root password for a server.
secgroup-add-group-rule
                    Add a source group rule to a security group.
secgroup-add-rule   Add a rule to a security group.
secgroup-create     Create a security group.
secgroup-update     Update a security group.
secgroup-delete     Delete a security group.
secgroup-delete-group-rule
                    Delete a source group rule from a security group.
secgroup-delete-rule
                    Delete a rule from a security group.
secgroup-list       List security groups for the curent tenant.
secgroup-list-rules
                    List rules for a security group.
show                Show details about the given server.
ssh                 SSH into a server.
start               Start a server.
stop                Stop a server.
```

```
    suspend             Suspend a server.
    unlock              Unlock a server.
    unpause             Unpause a server.
    unrescue            Unrescue a server.
    usage-list          List usage data for all tenants
    volume-attach       Attach a volume to a server.
    volume-create       Add a new volume.
    volume-delete       Remove a volume.
    volume-detach       Detach a volume from a server.
    volume-list         List all the volumes.
    volume-show         Show details about a volume.
    volume-snapshot-create
                        Add a new snapshot.
    volume-snapshot-delete
                        Remove a snapshot.
    volume-snapshot-list
                        List all the snapshots.
    volume-snapshot-show
                        Show details about a snapshot.
    volume-type-create  Create a new volume type.
    volume-type-delete  Delete a specific flavor
    volume-type-list    Print a list of available 'volume types'.
    x509-create-cert    Create x509 cert for a user in tenant
    x509-get-root-cert  Fetches the x509 root cert.
    bash-completion     Prints all of the commands and options to stdout so
                        that the nova.bash_completion script doesn't have to
                        hard code them.
    help                Display help about this program or one of its
                        subcommands.
```

A.3 Configuration du Syslog Client et Serveur

La communication entre le client et le serveur se fait via UDP. J'ai utilisé ce protocole pour éviter de surcharger le réseau avec des Acknowledge et des demandes de renvoi de paquets. Il sera déjà suffisamment sollicité avec l'envoi des messages de Logs.

- Configuration du Client Syslog-ng

La configuration indique que le client doit parcourir le fichier */var/messages* et l'envoyer en UDP au serveur de Log.

```
source s_log {file("/var/log/messages"); };
destination d_network { udp( "10.2.4.107" port(514) ); };
```

- Configuration du Serveur Syslog-ng

Le serveur va centraliser les Logs et les repartir en suivant l'arborescence décrite au §6.1.3. Pour cela nous devons appliquer des filtres au moment de la réception d'un message de Log. Par exemple nous avons :

```
destination d_network-nova {
  file(
    "/var/log/network/network.log",
};
filter f_network-nova {
  program("nova*");
  and host("network");
};
log { source(s_net); filter(f_network-nova); destination(d_network-nova); };
```

Comme nous le voyons ci-dessus, un traitement est fait pour chaque Log. De plus ce qui rend plus intéressant Syslog-NG par rapport à un autre outil de gestion de Log comme Rsyslog, est que nous pouvons avoir une granularité plus fine au niveau des filtres. Il reconnaît automatiquement les champs Syslog prédéfinis dans la RFC comme le champs « Programm » ou « Host », afin d'appliquer un aiguillage au niveau du stockage des Logs.

A.4 Objet JSON

Comme en JAVA, le but ici est de créer un objet qui sera basé sur une structure « Clé : Valeur ». Voici un exemple d'Objet JSON :

```
"fruits": [
    { "kiwis": 3,
      "mangues": 4,
      "pommes": null
    },
]
```

L'objet Fruits est composé de 3 kiwis, 4 mangues et d'aucune pomme. Ce formatage permet une lecture aisée par l'œil humain, et facilement interprétable par du Code Python ou autre. La finalité de cette étape est de convertir nos lignes de messages situées dans les fichiers de Log en Objet JSON. Ces types d'objets peuvent aisément transiter à travers une requête http.

Descriptif des Objets JSON : http://fr.wikipedia.org/wiki/JavaScript_Object_Notation