Systèmes Distribués pour le Traitement de Données

BORJA Abir CHADLI Amine DEPAROIS Shannon
GHANOUCH Issam LEOUAFI Mehdi SAMORAH Omar
SANHAJI Omar STROPPA Nicolas

Janvier 2018

Le Git Repo du projet :
`https://github.com/Ghanouch/SDTD-SMACK-BIGDATA-CLOUD`

1 Description des composantes logicielles

1.1 Akka[1]

Akka est une boîte à outils open source qui simplifie la construction d'applications concurrentes et distribuées sur la JVM. Akka supporte plusieurs modèles de programmation concurrente, le plus important étant le modèle Acteur
L'API Akka est disponible en Java et Scala, ce dernier est écrit en Scala. A partir de sa version 2.10, Scala intègre l'implémentation d'Akka parmi les composantes de sa bibliothèque standard.
En ce qui concerne la pile SMACK, il a été jugé utile d'utiliser Akka-HTTP pour capturer les données depuis une source web et les transformer d'un format JSON à un format objet. Utiliser Akka-HTTP Client pour capturer les données en Streaming présente les avantages suivants :

- Gestion de la vitesse du flux de données selon le temps de réponse du traitement.
- Le traitement Streaming permet d'optimiser l'utilisation de la mémoire.
- Gestion du Buffer lors du Stream de données.

Bien que son concurrent SCALA présente plusieurs avantages en terme de richesse de par les bibliothèques qu'il inclut, notre choix s'est porté sur le langage JAVA car mieux dompté par les membres de l'équipe.

1.2 Kafka[2]

Apache Kafka est un système de messagerie distribué publish-subscribe qui permet entre autres de:

- Transférer les messages d'un "writer" vers un "reader" comme un système de messagerie standard.
- Traiter les données en streaming pour créer de la valeur. Ceci peut soit être fait en exploitant une API fournie par Kafka lui-même, soit en s'intégrant avec, par exemple, Spark Streaming qui se chargera du traitement.

Dans Kafka tous les messages émis par les producteurs appartiennent nécessairement à un sujet ou topic. Chaque topic est réparti en plusieurs partitions. Cette répartition permet la scalabilité en mettant chaque partition sur un serveur différent. Les partitions sont répliquées et gérées dans un modèle maître-esclave. En ce qui concerne les consommateurs, ils sont regroupés dans différents "consumer group" qui est l'équivalent d'un "reader" dans

un système publish-subscribe traditionnel.
Ce modèle de flux de messages fournit les garanties suivantes :

- Les messages dans une partition sont ordonnés par temps de publication.
- Une tolérance aux fautes en fonction du nombre de réplications.

1.3 Spark[3]

Spark est un framework libre pour le traitement de données à grande échelle qui implémente le modèle Map Reduce. Spark permet, dans un premier temps, de déployer un traitement parallèle de données suivant le modèle Map Reduce mais aussi de décrire des pipelines de traitement de données plus complexes, à l'aide d'outils tels que Spark Streaming pour traiter des données en flux continu, ou Spark MLib pour l'apprentissage automatique. Spark s'appuie sur des solutions de stockage de données telles que HDFS ou Cassandra. Ladite API est disponible en Scala, Java et Python.

1.4 Mesos

Mesos est un gestionnaire de ressources pour les infrastructures de gestion de données. il permet, en s'exécutant sur toutes les machines du Cluster disponibles, de fournir une API de gestion de ressources et d'ordonnancement[1] aux différents éléments de la pile. Mesos permet la gestion de la haute disponibilité et de la scalabilité horizontale. La haute disponibilité est assurée par la réplication du noeud maître, dès lors que ce dernier est en panne, Il est possible de choisir un réplica qui prend le rôle de leader. La haute disponibilité de Mesos requière qu'une instance d'Apache Zookeeper tourne sur le système. Mesos assure aussi la haute disponibilité des autres services en gérant les pannes lorsqu'ils se produisent.

1.5 Zookeeper[4]

Zookeeper, quant à lui, est responsable d'élir un Leader dans toute distribution basée sur le principe de maitre-esclave. Pour ce faire, Zookeeper va procéder à un consensus en se basant sur l'algorithme "Zookeeper Atomic Broadcast" (lui même inspiré de l'algorithme de Paxos). Dans notre cas d'utilisation, le Zookeeper aura uniquement pour but de régir Kafka et Mesos.

1.6 Cassandra

Avant toute chose, Cassandra étant une base de donnée orientée colonnes, fait abstraction de toutes sortes de schémas imposés par son prédécesseur SQL plus communément utilisé dans les bases de données relationnelles. Dès lors, un nouveau langage d'adressage des

[1] Scheduling en anglais

tables (ou familles de colonnes pour être plus exacte) est instauré, le CQL (C pour Cassandra) que l'on va employer dans notre application JAVA pour toute opération de CRUD.

Ceci dit, notre structure de base de donnée est très simple: plusieurs nœuds (2 pour cette étude) régis par un nœud abritant l'OpsCenter.
Les nœuds Cassandra seront donc configurés de telle sorte que tout KeySpace ou Famille de Colonne soit répliquée au nombre des instances présentes (on parle alors du Replication Factor) afin d'assurer une tolérance aux panne optimale ainsi qu'une même répartition de charges grâce au partitionner . Pour ce faire, Apache nous propose de rassembler l'ensemble des nœuds dans un Rack qui fait office d'une séparation virtuelle de l'espace de stockage.

Dans notre étude, on se propose de définir un seul et même Rack (puisque le nombre de nœud n'est que de 2) dans un même Data Center cote à cote avec les autres machines déployées.

Après avoir définit la structure physique du Cluster, il est temps de s'atteler à l'intelligence qui gère le stockage. Deux cas de figures, Le SimpleStrategy qui "assure un stockage au mieux dans un ring, et le NetworkTopologyStrategy qui garanti le stockage des clés à travers plusieurs Datacenters de façon homogène."Il est vrai que du fait que nous travaillons sur un DC, l'on pourrait aisément choisir Simple Snitch, cela dit, par soucis de scalabilité (au cas où nous voudrions étendre notre cluster à d'autres zone géographiques par exemple), nous choisirons NetworkTopologyStrategy.

Dans ce rapport, nous n'allons pas nous attarder sur la consistance ou encore l'atomicité des données. Une seule chose à savoir, est que "L'atomicité est aussi moins forte que ce à quoi nous sommes habitués dans le monde relationnel. Cassandra garanti l'atomicité pour une ColumnFamily : donc pour toutes les colonnes d'une ligne." On n'a donc pas à se préoccuper de la validité des données.

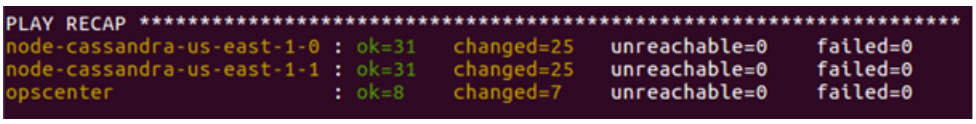

Figure 1: Le cluster une fois déployé

Après le déploiement, nous obtenons un petit cluster Cassandra composé comme cité précédemment de deux nœuds. Lesdits nœuds communiquent entre eux de façon continue grâce au protocole Gossip afin de connaître l'état de chacun d'entre eux. Cependant pour communiquer avec le client (JAVA dans notre cas), Cassandra utilise le protocole Thrift (ouvert sur le port 7160) qui a comme objectif de répartir les requête ,de les router vers les nœuds concernés ou encore de relancer les requêtes si jamais elles échouent.

1.7 DataStax OpsCenter

OpsCenter est un des services proposés par DataStax Entreprise afin de Monitorer l'ensemble des nœuds Cassandra et ainsi palier à une éventuelle panne. Ceci en proposant, par exemple, des réparations par les agents Datastax ou encore une possibilité de créer d'autres nœuds et de faire migrer les données du nœud en panne à celui fraîchement créé.
La configuration se fait en amont des nœuds Cassandra et ce n'est qu'après la création desdits nœuds, qu'ils viennent se connecter à l'Ops Center.
Par soucis de simplicité, aucune authentification JMX ne sera utilisée pour les nœuds. Ceci dit, l'OpsCenter lui, sera protégé par une authentification car accessible depuis l'extérieur sur le port http (user : admin, mdp : admin).
Plusieurs graphiques sont mis à notre disposition pour voir l'état des nœuds:

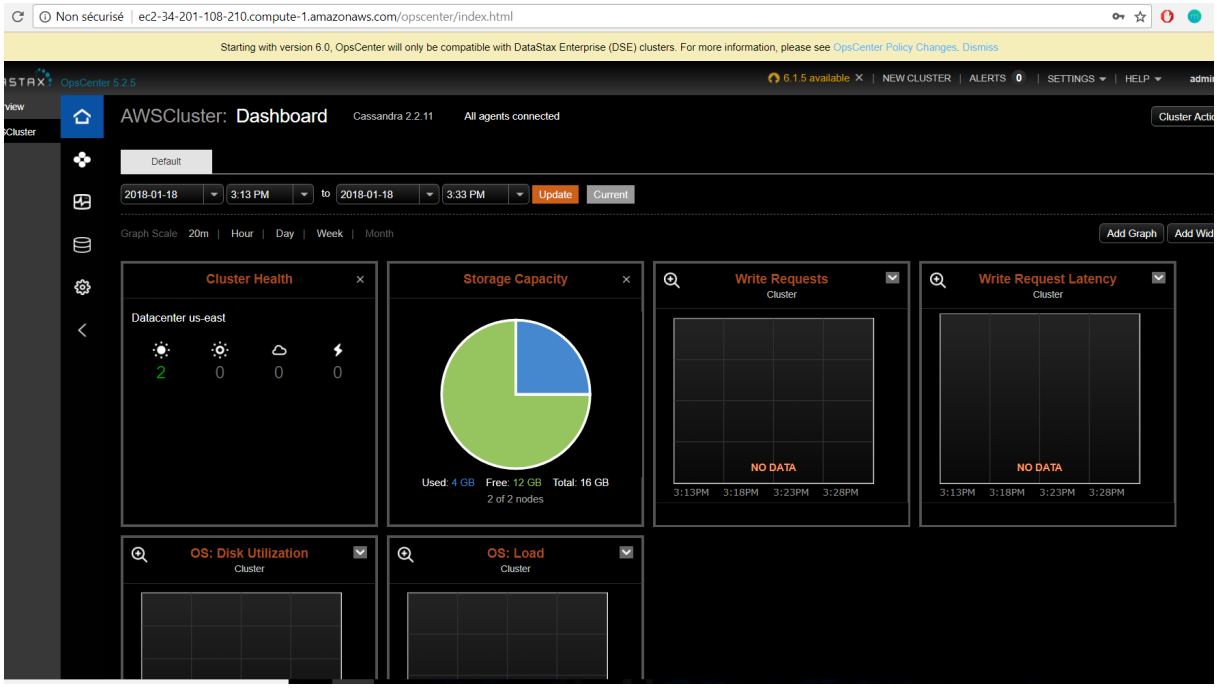

Figure 2: L'OpsCenter à son premier déploiement en amont de l'approvisionnement de la base de donnée

On aura également la possibilité d'ajouter d'autres nœuds à un Cluster existant directement depuis l'interface.

1.8 Terraform[5]

Terraform est un outil qui sert à la construction, le Provisioning, et le Versioning d'une infrastructure.
Terraform génère à partir des scripts un plan d'exécution décrivant les actions nécessaires pour atteindre l'état souhaité, puis l'exécute pour construire l'architecture décrite.
Parmi les fonctionnalités clé de Terraform, on peut citer :

- "Infrastructure as a code": L'infrastructure est décrite à travers une syntaxe de configuration haut niveau. Ceci permet de traiter et Versionner l'architecture.
- Graphe de ressources : Terraform construit un graphe des ressources disponibles, et permet d'effectuer de façon parallèle la création et la modification des ressources indépendantes. Grâce à cela, Terraform construit l'infrastructure de manière optimale, et permet d'acquérir une vision sur toutes les dépendances au sein de l'architecture.

Terraform permet de décomposer un projet en modules, chaque module contient les fichiers suivants:

- variables.tf : fichier qui contient les variables d'entrée du programme, ces variables sont réutilisées pour définir les spécifications des instances. Par exemple: la taille ou le nombre d'instances.
- main.tf : programme exécuté lors de l'appel d'un module.
- output.tf : fichier qui contient l'ensemble des variables retenues par Terraform, celles-ci sont accessibles en dehors du module ou par invite de commande.

Notre choix d'utiliser Terraform est évident:

- La structure de code organisée est plus simple à maintenir.
- Les différentes variables de configuration sont déjà implémentées
- Il représente une couche d'abstraction de l'AWS API

1.9 Ansible[6]

Ansible est un moteur d'automatisation qui permet d'exécuter des tâches sur plusieurs serveurs.
Ansible offre deux types de fonctionnements différents :

- Fonctionnement Ad-hoc : cela permet d'exécuter des commandes sur des serveurs spécifiques. Ce mode de fonctionnement est utilisé pour des tâches particulières à un serveur. Par exemple : tester l'état d'un serveur, tester le réseau.
- Fichier Playbook: Playbook est le language d'orchestration, de déploiement et de configuration d'Ansible, le language est similaire au YAML.

Ansible, comme Terraform, permet de décomposer un projet en rôles, chaque rôle contient les dossiers suivants:

- tasks: contient l'ensemble des tâches à réaliser, le fichier main.yml est celui appelé dès lors qu'on référence le rôle dans le programme principal.
- templates: contient l'ensemble des templates utilisés dans les rôles par les fichiers yaml.

Le choix d'utiliser Ansible se base sur les raisons suivants:

- La structure des tasks et leurs descriptions permet une compréhension rapide du comportement désiré
- Permet une meilleur gestion des machines puisqu'il n'effectue que les nouvelles tâches
- Il propose un bon nombre de modules qui facilitent le déploiement et configuration des applications

2 Architecture logicielle globale

2.1 Description de l'architecture

Notre architecture est la suivante:

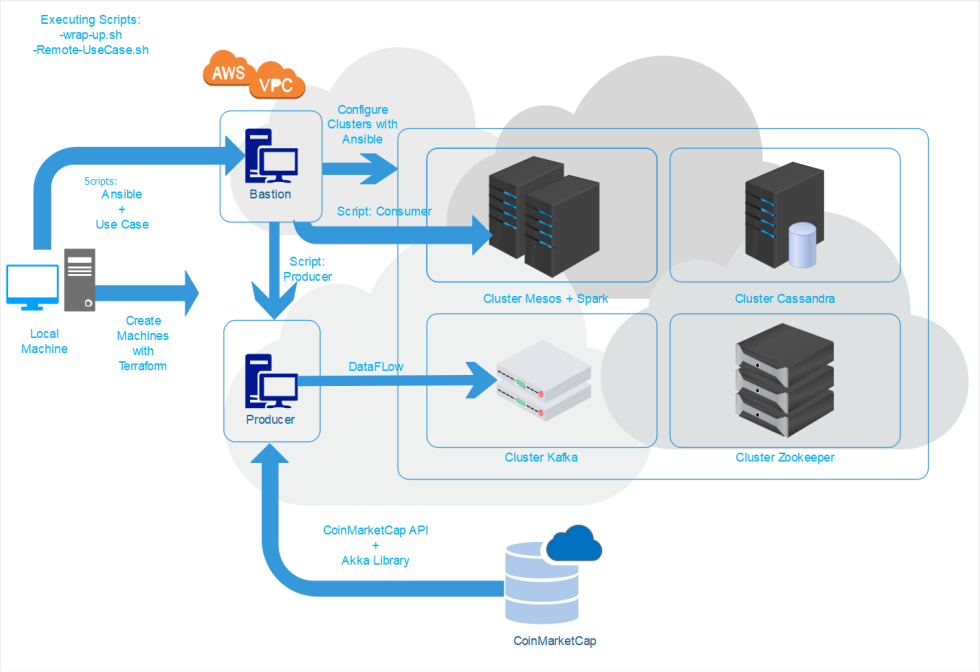

Figure 3: Architecture de la pile SMACK

Comme précisé précédemment, on utilise les deux outils Terraform et Ansible pour dresser notre architecture. En premier lieu, Terraform crée et approvisionne tout les machines présentes dans l'illustration ci-dessus. On remarquera l'existence de deux nouvelles machine hors de la stack régulière, la machine Bastion et la machine Producer. La première , après avoir reçu les différents scripts Ansible rassemblés dans un dossier, installera le nécessaire sur ladite stack. Le rôle de la machine Producer sera explicité dans la partie use case

2.2 Deploiement

Pour savoir comment effectuer les différentes manipulation pour le déploiement, veuillez vous référer au lien git

3 Application de démonstration

3.1 L'application de démonstration envisagée

L'application de démonstration utilise des données concernant les crypto-monnaies les mieux classés dans le monde. Le but étant de calculer certains indices financiers sur ces crypto-monnaies à fin d'en déduire des décisions. Les fonctions décrites ci-dessous permettront d'aider les éventuels investisseurs à choisir sur quelle monnaie investir afin d'assurer plus ou moins un gain.

3.1.1 Le Retour sur Investissement

Le RSI aussi appelé capital investi est un ratio utilisé en calculs financiers afin de mesurer le pourcentage d'argent gagné ou perdu suite à un investissement (l'intérêt).
Une de nos fonctions consistera donc à calculer le dit rendement par rapport à un certain capital investi pour une crypto-monnaie donnée.
Pour ce faire, on aura recours au rendement arithmétique calculé comme suit :

Rentabilité arithmétique (0,T)(%) = (gain de l'investissement(T) –
coût de l'investissement(0)) / coût de l'investissement(0).

Pour des fins de simplification et de précision, on calculera plutôt la rentabilité par une formule logarithmique (choix qui s'avère plus particulièrement utile lors du calcul de la rentabilité annuelle). De plus de son asymétrie, le rendement composé permet de mieux voir l'évolution du retour sur investissement pour plusieurs investissements continus.

$$RSI = ln(\frac{V_f}{V_i})$$

avec Vf : la valeur actuelle (prix) de la dite crypto-mmonnaie er Vi est la valeur à une date précédente : t = date_actuelle - T. où T est la période de temps sur la quelle on calcule l'indice de retour sur investissement. Par exemple : rsi_24h : est le retour d'investissement d'une crypto-mmonnaie calculés sur une période d'un jour.

3.1.2 L'Annualisation

Contrairement au rendement annuel où il s'agit dans ce cas-ci de convertir la rentabilité en rentabilité annuelle, le rendement annualisé est le retour sur investissement calculé sur une période donnée multipliée ou divisée pour avoir un aperçu du rendement d'une année.

3.1.3 La RSI Moyenne

Il s'agit de la valeur du retour sur investissement moyenne sur une période donnée : et donc en gros cette valeur reflète plus le sens de variation de la crypto-monnaie : car même s'il y' a une forte et rapide variation des valeurs de ces crypto-monnaie, cependant en calculant la moyenne sur une période on peut avoir une idée plus claire sur le sens globale de la variation (croissance ou décroissance).

3.1.4 La Volatilité

Il s'agit là du degré de variation du rendement de la crypto-monnaie. Elle a pour objectif de mesurer le risque de détenir ledit actif, la volatilité est calculée à partir des logs-rendements cités plus haut. Sa précision dépend du laps de temps choisi mais peut cependant être adaptée selon les besoins à différentes fréquences. La volatilité, quand elle est animalisée, représente l'écart-type (à condition que les rendements calculés soient deux à deux indépendants).

$$\sigma = \sqrt{\frac{252}{N} \sum_{t=1}^{N} (ln(\frac{V(t)}{V(t-1)}) - moyenne\_rsi)^2}$$

avec moyenne_rsi est la moyenne des rsi de N valeurs de rsi successives dans le temps.

3.1.5 Le Taux du marché

cet indice reflète la part du marché d'une crypto-monnaie parmi toutes les autres crypto-monnaies.Il est calculé en divisant le capital de la crypo-omonnaie sur la somme des capitaux de toutes les crypto-monnaies analysées.

3.1.6 Extension

Avec l'ensemble des données à notre disposition sur les crypto-monnaies, nous essaierons dans un premier temps de voir celles qui ne sont pas encore arrivés à un stade de maturité, puis par la suite, identifier parmi celles-là les monnaies qui présentent une croissance intéressante.
En effet, à travers l'historique des données des crypto-monnaies ainsi que d'autres données sociales (tweets, recherches. . .), il est possible de sélectionner les monnaies apparues récemment. Ensuite, à travers l'indice d'évolution sur une durée déterminée, on pourrait identifier les crypto-monnaie cibles, avant d'y effectuer l'ensemble des études statistiques citées dans la partie précédente.

3.2 Scénario de fonctionnement

la figure 12 montre un résumé de la pile logicielle utilisée dans le scénario, elle sert à illustrer l'exemple d'utilisation suivant:

- Récupération des JSON par "Scraping" grâce à Akka-http, cela est réalisé par un programme qu'on nomme "Producer".
- L'utilisation de l'API Kafka Procuder pour publier les données sur les différents Topics du Cluster KAFKA.
- L'utilisation de l'API Kafka Consumer, par les différents exécuteurs de Spark, qui sont abonnés à des topics spécifiques.
- Après l'agrégation et le traitement de données par Spark Streaming,on stocke les données sur un Cluster Cassandra grâce au Spark Cassandra Connector.

4 Réalisation du cas d'utilisation :

4.1 Producer :

4.1.1 Architecture du Producer :

La figure 13 montre les différentes composantes mises en oeuvre pour l'exécution du programme Producer.

4.1.2 Production : (Voir le code sur Github)

La Production permet de recevoir et filtrer (en streaming) les données des Cryptomonnaies puis transmettre les résultats à un Kafka broker. Cela se fait en plusieurs étapes :

1. Un acteur de AKKA HTTP récupère un JSON à partir de l'URI suivant : `https://api.coinmarketcap.com/v1/ticker/`
2. Filtrage Des données récupérées : Grâcee à Akka HTTP, nous pouvons récupérer Le flux de données (JSON) en Streaming, cela, nous permettra d'effectuer un filtrage basé sur le classement actuel des cryptomonnaies afin de se concentrer que sur une partie (par exemple : les 20 mieux classées)
3. Publication des résultats sur un TOPIC : Après avoir réalisé le filtrage, nous pouvons publier les résultats sur un ou plusieurs topics auxquels nous sommes abonnées.
4. Lancement Du JOB de Producer (Lien Job Producer) : Ce job prend 4 paramètres qui sont :
   - Broker KAFKA : l'URL du Kafka Broker
   - TOPIC CHOISI : L'ensemble des topics choisis

- BORNE MINIMALE : Le classement minimal des cryptomonnaies
- BORNE MAXIMALE : Le classement maximal des cryptomonnaies
- NOMBRE REQUETE PAR MINUTE: C'est le nombre de JSON à récupérer par minute (cela permet de contrôler la vitesse de flux)

Example pour lancer le JOB Producer :

```
#java -jar NOM_JOB_PRODUCER
--broker:@IP_BROKER:PORT --topic:NOMTOPIC
--min:BORNE_MINIMAL --max:BORNE_MAXIMAL
--speed:NOMBRE_REQUETE_PAR_MINUTE
```

4.2 Consumer :

4.2.1 L'Architecture du Consumer :

La figure 14 illustre l'architecture des composantes nécessaires à l'éxecution du programme consumer.

4.2.2 Consummation :

Voir le code source du programme consumer sur Github

4.2.3 Lancement Du Job Consumer :

Ce job prend 3 paramètres qui sont :

1. Broker KAFKA : L'URL du kafka Broker ou de la listes des kafka Broker.
2. TOPIC CHOISI : L'ensemble des topics choisis.
3. L'URL du Cassandra: L'URL de Cassandra.

Example pour lancer le Job :

```
#java -jar NOM_JOB_CONSUMER :@IP_BROKER:PORT NOMTOPIC @IP_CASSANDRA
```

4.3 La mise en place de la base de donnée Cassandra :

4.3.1 La modélisation de la base de donnée

Comme l'on a pu voir, Cassandra s'autogère en ce qui concerne la tolérance aux pannes et la haute disponibilité, sous réserve d'avoir correctement définit certains paramètres, et plus particulièrement ceux cités précédemment dans la section Cassandra.

Pour le CRUD (create, read, update, delete) de notre keyspace et des ses colonnes, c'est l'application qui se charge de les mettre en place. Avant de commencer tout job, est impératif de construire notre keyspace qui va contenir bon nombre de familles de colonnes. Une propriété des bases de données orientées colonnes est leur aptitude à s'adapter aux requêtes souhaitées sans pour autant se soucier de la redondance des enregistrements. Notre famille de colonnes se présente donc comme suit:

1. Une FdC "Stats" dédiée au stockage des résultats de la fonction de retour sur investissement. On dira de cette FdC qu'elle est éphémère puique elle se réécrit à chaque nouvelle "fournée" de la part de Spark, la table est donc écrasée par les nouveaux éléments.

2. Pour chaque crypto monnaie, une table dédiée qui stocke toute information relative à cette denière (MarketCap ,TimeStamp...). Cependant, selon le classement des crypto monnaies, une nouvelle table contenant les informations relatives à cette dernière est ajoutée aux reste.

3. Une dernière table "GlobalStats" se crée est qui regroupe plusieurs variables calculées (Moyenne, Ecart-type, Volatilité et également L'évolution temporelle de la crypto monnaie).

4.4 L'automatisation du cas d'utilisation :

4.4.1 Introduction :

L'idée principale est de pouvoir réaliser l'ensemble des tâches requises pour le lancement du cas d'utilisation en un seul click, tout en respectant l'architecture préalablement déployée par Terraform et Ansible.

4.5 Réalisation et Scénario de l'automatisation (Lien code source):

Le scénario du cas d'utilisation se fait à travers deux étapes :

1. Récupération des fichiers d'entrées :

   (a) La récupération d'un fichier HOSTS (par Terraform) qui contient les adresses IP des différentes machines attribuées à leurs rôles. (Annexe 1)

   (b) La récupération des clés ssh utilisées pour accéder à la machine Bastion "deploy_cle", ainsi que la clé utilisée par la machine Bastion pour accéder aux différentes autres machines "cluster_interconnection_cle".

2. Le lancement un seul script ("sh Remote-UseCase.sh") qui permettra de :

(a) Envoyer à la machine Bastion le script "UseCase.sh" qui contient les différentes tâches à exécuter, et le paramétrage personnalisé (Nombre de réplication, nom du topic, La vitesse de requêtage ...)

(b) Envoyer à la machine Bostian:

- Le fichier HOSTS
- La clé "cluster_interconnection_cle".
- Le lien pour accéder au JOB PRODUCER
- Le lien pour accéder au JOB CONSUMER

(c) Exécuter le script "UseCase.sh" sur la machine Bastion.

L'exécution du script "UseCase.sh" permet :

(a) L'extraction de l'ensemble des adresses IP, chacune représentant une machine.

(b) La création du Topic sur un ou plusieurs broker Kafka, tout en précisant un ensemble de paramètres (Zookeper IP, le facteur de réplication, Le nom du Topic)

(c) Lancement du JOB PRODUCER sur la machine Producer, cela se fait après que cette machine récupérera l'ensemble des paramètres envoyés par la machine Bastion :

- URI JOB PRODUCER : Lien permettant de télécharger le JOB PRODUCER EN LIGNE
- L'ENSEMBLE DES PARAMETRE PERMETTANT DE LANCER CE JOB (Voir la partie "LANCEMENT DU JOB de Producer")

(d) Lancement du JOB Consumer sur une machine sur laquelle tourne le service Spark Driver (sur notre cas, une machine avec le rôle d'un Mesos MASTER, puisqu'elle contient les deux services : Mesos Master, Spark Driver) aussi cela se fait après avoir récupérées l"ensemble des paramètres par la machine Bastian:

- URI permettant de télécharger le consumer (ou bien, comme sur notre cas, la réception du job préalablement reçu par la machine Bastian)
- -L'Ensemble Des Paramètres Permettant De Lancer Ce Job (Voir la partie "LANCEMENT DU JOB de Consumer")

5 L'Application de démonstration :

L'application de démonstration consiste en un programme en Java (Consumer) qui réalise les traitements suivants :

1. Lecture en streaming depuis un ou plusieurs topics (dans des brokers) aux quels le consumer (notre programme) est abonné : les adresses IP et ports pour spécifier ces topics sont passés en paramètre à notre programme. La Lecture en streaming est

faite en utilisant Kafka Spark streaming (Spark Streaming integration for Kafka), les données sont donc récupérées en format Json.

2. Transformation des données en Json à des objets de type notre model (classe Monnaie) à l'aide de la librairie Jackson Mapper. Ainsi on récupère une liste d'objets de type Monnaie après chaque lecture (en streaming) depuis le(s) topic.

3. Après avoir récupéré cette liste on la converti en JavaRDD (à l'aide de la fonction parallelize) pour pouvoir effectuer sur elle un ensemble de calcul de manière distribuée.

4. L'ensemble des calculs qu'on effectue (en streaming) sont les suivants :

   (a) Calcul du retour sur investissement (rsi) pour chaque cryptomonnaie récupéré, sur une période d'une heure, 24 heures et 7 jours.

   (b) Calcul de la rsi annualisée : cette valeur permet de voir la variation logarithmique annuelle de la valeur d'une cryptomonnaie : en terme d'interprétation, elle n'ajoute pas une grande valeur vu la variation rapide de ces cryptomonnaie. Cependant elle peut être vue comme un indice global pour la comparaison entre les déférentes cryptomonnaies : une cryptomonnaie qui a un fort croissement aura une RSI très grand, cependant une cryptomonnaie de faible croissance aura un RSI petit.

   (c) Calcul de l'indice de de Herfindahl : cet indice reflète la part du marché d'une cryptomonnaie parmi toutes les autres cryptomonnaies.

5. Persistance des indices calculés dans une base de données Cassandra : pour cela on effectue les actions suivantes :

   (a) Au début de l'application on appelle une seule fois :

      i. La création d'une connexion avec la base des données Cassandra à l'aide de Spark Cassandra Connector.

      ii. La création d'un keyspace (crypto) s'il n'existe pas déjà. Puis, on utilise ce Keyspace et on crée une table Stats qui va contenir l'ensemble des indices calculées pour toutes les cryptomonnaies après un traitement (streaming), c'est une table statique dont les valeurs subissent des updates et non des insertions.

   (b) Après chaque traitement streaming :

      i. Édition de la table statique Stats avec les nouvelles valeurs calculés.

      ii. Pour chaque cryptomonnaie dans la liste du résultat, on crée une table (s'elle n'existe pas déjà) avec son nom (ex : Table : Bitcoin_Cach) et on insèrent les nouvelles valeurs calculés avec le champs TimeInsert qui spécifie le temps d'insertion des nouvelles valeurs.

6. Le deuxième ensemble des traitements est le suivant :

   Après avoir rempli les tables des différentes cryptomonnaies avec les valeurs calculées, le programme lit à partir de l'ensemble de ces tables toutes les valeurs à partir d'une date T (par exemple dernière 24 heures) et sur ces listes le programme effectue les calculs suivants :

   (a) Calcul de la valeur moyenne des retours sur investissement (rsi) pour chaque cryptomonnaie.

   (b) Calcul de la volatilité : c'est en fait l'écart type de la distribution dans le temps des valeurs de rsi pour chaque cryptomonnaie.

   Pour cela le programme parallelise l'ensemble des liste récupérées avant d'appeler la fonction compute qui effectuent ces calculs (sur chaque RDD). Ainsi on récupère à la fin une liste d'objets (nom_crypto, moyenne, ecart_type) pour chaque cryptomonnaie. En fin on persiste ces objets dans une table nommée GlobalStats, cette table est utilisée juste pour visualiser les valeurs calculées.

6 Déployer L'infrastructure

Comme Indiqué section **??**, l'outil de déploiement est un script shell qui utilise Terraform et Ansible pour déployer l'infrastructure.

6.1 Spécification

L'outil est conforme à la spécification suivante:

- Une commande: déploiement des composants de la pile SMACK en une commande.
- Sécurisé : le déploiement restreint les ports TCP ouverts. Les ports ouverts sont ceux nécessaires aux communications entre les différents composants. De plus, les ports ne sont ouvert qu'aux machines du sous réseau.
- Modulaire : les modules Terraform et Ansible sont définit en sous modules indépendants les uns des autres. Terraform utilise la terminologie module, et Ansible role 9
- Hautement disponible: l'infrastructure déployée est tolérante à 1 faute
- Configurable: Il est possible de changer le nombre de machines pour le déploiement.

6.2 Limitations

Néanmoins, l'outil présente les limitations suivantes:

- Infrastructure fixe: Il n'est pas possible de déployer deux services sur une même machine, la répartition physique des services telle que Cassandra, Mesos ou Spark est implicite à la définition des modules.
- Sous réseau publique: La plage d'adresse est fournie par AWS, les services sont donc ouvert à toutes les machines de ce sous réseau.

7 Visualisation des données traîtées

Une fois nos données traitées sont bien stockées dans notre base de données Cassandra, Il est pertinent de visualiser ces dernières afin de mieux pouvoir les interpréter les résultats obtenus. Pour cela, on a décidé de s'appuyer sur un outil extérieur à notre pile : Apache Zeppelin.

7.1 Zeppelin

Apache Zeppelin est un Notebook multi-usage qui fourinit une interface web permettant d'analyser et mettre en forme simplement de manière visuelle et interactive de gros volumes de données traités. Dans notre projet Zeppelin est utilisé comme un Dashboard pour la visualisation et démonstration de données persistent dans la base de données Cassandra.

7.2 Résultats visualisés

Une fois la connexion Cassandra établie, on peut récupérer et visualiser l'ensemble des données traitées par de simples requêtes CQL sur un notebook utilisant l'interpréteur Cassandra. Avec un choix judicieux de visualisation pour chacune de ces requêtes, le Dashboard final s'apparente à ceci :

Figure 4: Tableau de Bord final

7.3 Interprétation des résultats

On s'attèlera maintenant sur chacun des six graphes qui constituent notre tableau de Bord.

Les deux premiers diagrammes constituent une visualisation des RSI annualisés, hebdomadaire, journalier, et horaire pour les cryptomonnaies auxquelles on s'intéresse, à savoir pour le cas de ces diagrammes le Bitcoin, Ethereum, Ripple, Bitcoin Cash et le Cardano.

Nous rappelons que le RSI (Retour sur Investissement) est un ratio utilisé en calculs financiers pour mesurer le pourcentage d'argent gagné ou perdu suite à un investissement.

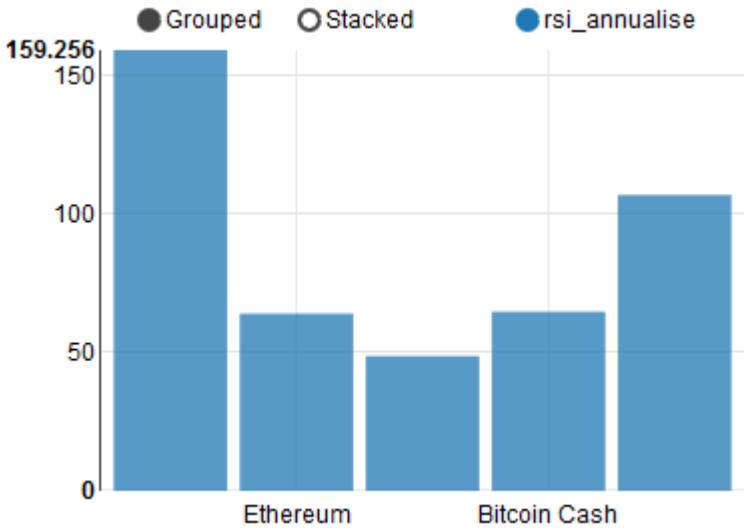

Figure 5: RSI annualisé

Ce premier diagramme s'intéresse uniquement à l'RSI annualisé des cinq Cryptomonnaies précitées. On remarque non seulement qu'il est positif pour tous les cinq, démontrant déjà que l'investissement sur les cryptomonnaies fut assez rentable cette dernière année, chose assez conforme à ce que nous entendons parler. Plus que cela, on remarque que le Bitcoin excède un RSI annualisé de 150 pourcent; signifiant qu'un investisseur lambda qui s'est mis depuis l'an dernier à trader en Bitcoin, devrait avoir maintenant jusqu'à deux fois et demi son investissement initial.

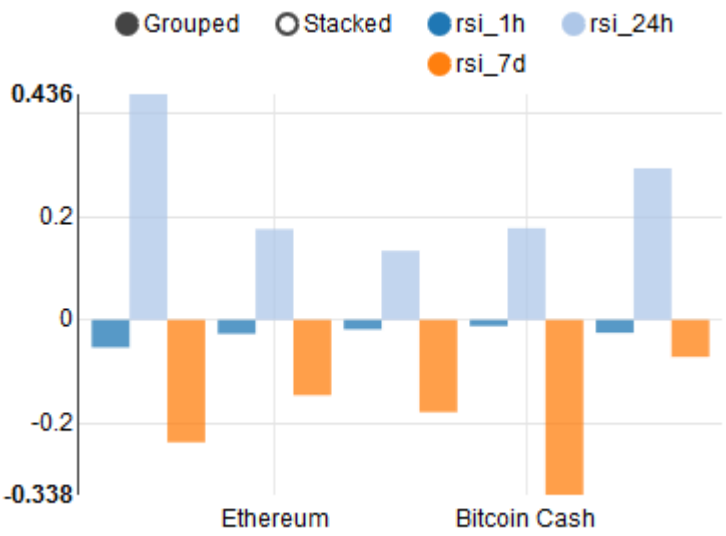

Figure 6: RSI horaires, journaliers, hebdomadaires

Ce deuxième diagramme qui s'intéresse à la fois au RSI horaires, journaliers, et hebdomadaires, est un peu moins optimiste que le premier. On y remarque que les RSI journaliers sont globalement positifs, tandis que l'horaire est légèrement négatif, et l'hebdomadaire très négatif. Quoique ceci puisse sembler un peu contradictoire, ce résultat obtenu est tout à fait logique, puisque dans la semaine de création de ce diagramme ; les cryptomonnaies ont pris une assez lourde chute dans l'ensemble, et s'en reprennent doucement dans les derniers jours. Ce diagramme démontre parfaitement la volatilité tant constatée de la valeur des cryptomonnaies.

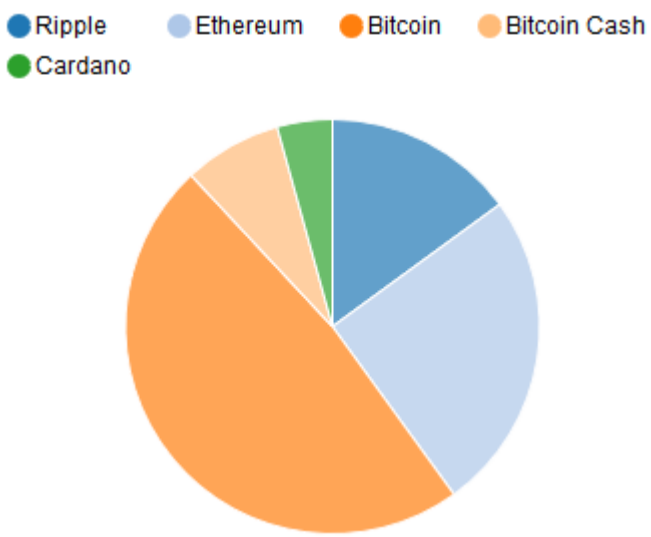

Figure 7: Taux de marché

Ce troisième diagramme est bien plus expressif et simple que les deux autres, et s'intéresse à une caractéristique différente : le taux de marché des cryptomonnaies. Il s'agit simplement d'un ratio du volume monétaire de transactions pour une cryptomonnaie, sur celui de l'ensemble des cryptomonnaies. On y remarque du coup que le Bitcoin occupe la quasi moitié du marché, suivi par l'Ethereum qui en occupe le quart environ, suivi en suite du Ripple. Ce résultat est tout à fait logique et connu par toute personne qui poursuit les actualités des Cryptomonnaies, qui s'explique aussi par le fait qu'il s'agit respectivement des premières cryptomonnaies qui ont vu le jour à l'époque.

Les trois derniers diagrammes s'intéressent à la fluctuation du RSA hebdomadaire des trois cryptomonnaies dominantes : Bitcoin, Ethereum, Ripple durant une très courte période, de façon à en déduire leur volatilité.

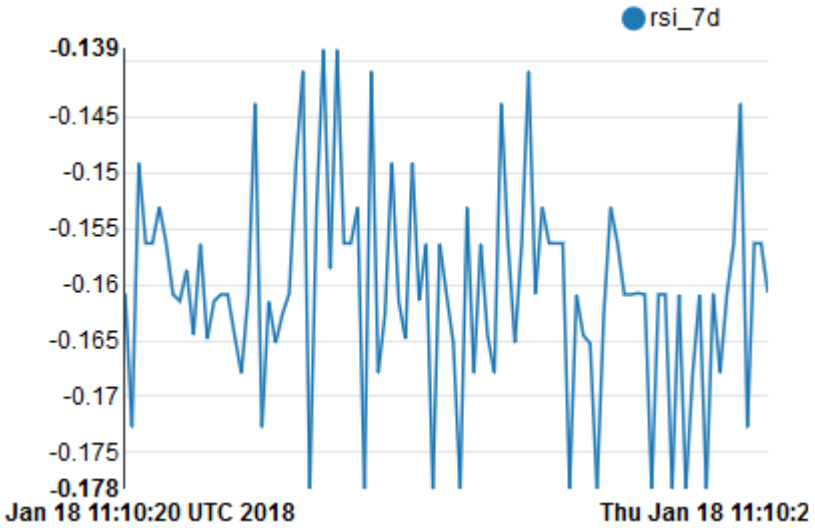

Figure 8: Volatilité Bitcoin

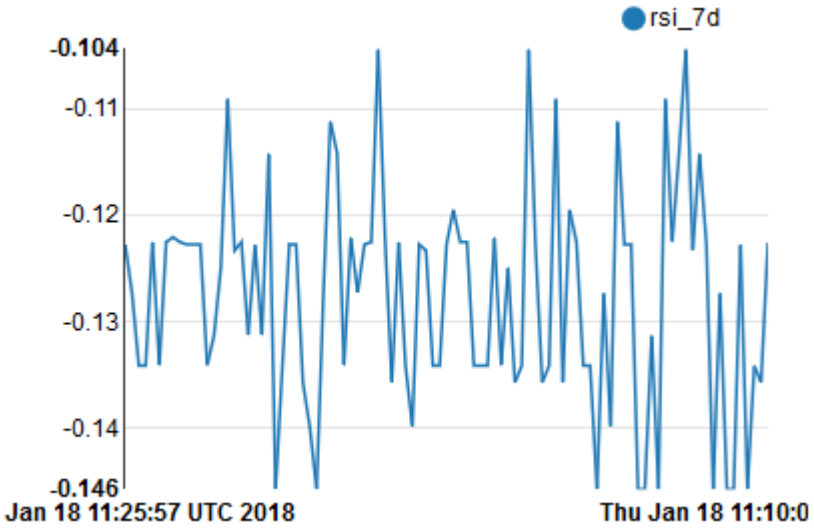

Figure 9: Volatilité Ethereum

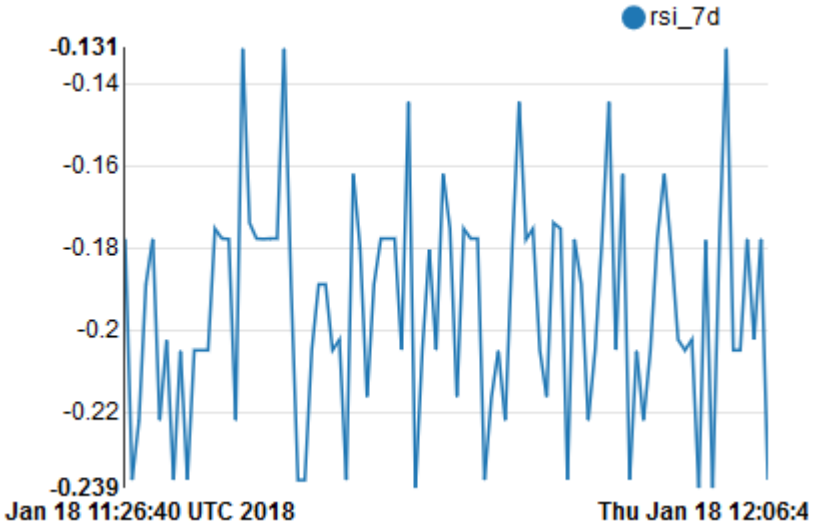

Figure 10: Volatilité Ripple

La première chose qui nous saute aux yeux par ses trois diagrammes est une fois encore la grande fluctuation des Cryptomonnaies, malgré la très courte durée étudiée. Il est clair et net qu'elles sont très volatiles, signifiant qu'il existe un risque assez conséquent dans leur investissement, pouvant chuter ou monter brusquement en quelque minutes. On remarque toutefois une cohérence avec les résultats précédents, dans le sens ou le RSA hebdomadaire reste dans le négatif ; et qu'il est légèrement en meilleur état chez l'Ethereum dans l'ensemble ; chose que l'on pourrait interpréter comme un encouragement à investir sur ce dernier.

8 Difficulté rencontrées

8.1 Problèmes de conflit de dépendances :

Le développement des JOBS (Producer et Consumer) a commencé par une phase de sélection de briques logicielles fournissant une ou plusieurs des fonctionnalités requises.
Par exemple, pour le consumer, il fallait avoir les dépendances suivantes : KAFKA CONSUMER, SPARK CORE, SPARK STREAMING, CASSANDRA CONNECTOR, Scala, JACKSON. Au moment de l'intégration apparaît les conflits entre dépendances . C'est pourquoi on a procédé à deux solutions :
- Utiliser le gestionnaire des dépendances Maven pour les deux JOBS, surtout nous avons profité du concept de transitivité, à l'aide de Maven qui réalise cette opération en construisant un graphe des dépendances et en gérant les conflits et les recouvrements qui pourraient arriver
- Utiliser des dépendances qui sont déja intégrable entre eux :
* spark-streaming-kafka
* spark-cassandra-connector

8.2 le compte verrouillé par AWS

le verrouillage de tous les comptes de nôtres équipes sur AWS, quand nous lançons un script qui mettra en place l'ensemble de la pile. QuickLab considère étrangement la chose comme étant une fraude, et on se retrouve à essayer de les convaincre en vain du contraire (voir image annexe)

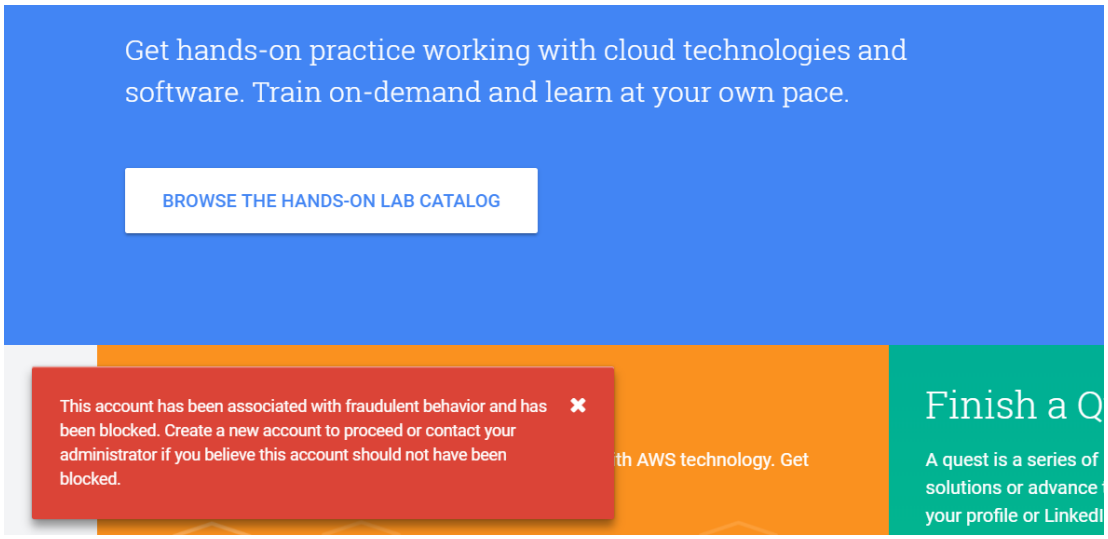

Figure 11: Exemple d'un compte bloqué

8.3 Problème de versions

Ce problème a surtout été constaté lors du déploiement de Cassandra et OpsCenter. D'une part, la librairie de python ne prend plus en charge le cassandra-driver qui est essentiel à CQLSH (client CQL de Cassandra) et l'on se retrouve, une fois le noeud déployé à avoir un client non fonctionnel. Aucunement ce bug n'a été répertoriée par DataStax ou Apache. D'un autre coté Ansible dispose d'une multitude de versions, qui au gré des développeurs, va supprimer des primitives pour en rajouter d'autres. On se retrouve à devoir changer régulièrement de versions pour trouver les primitives dont on a besoin.

8.4 Terraform

Un autre problème rencontré et que Terraform refuse sans raison apparente d'exécuter des commandes shell (on teste manuellement, et pourtant tout à l'air de fonctionner) . Il a fallut donc mettre en place des stratégies pour contourner ce problème. Un Exemple à cela, est que pour Cassandra le service refusait de démarrer automatiquement. Une solution à cela fut de transformer Cassandra en un service bootable directement au lancement de la machine.

8.5 Port et groupes de sécurité

Il a fallut consulter tous les services un à un pour déterminer les ports à utiliser afin de les reproduire sur les groupes de sécurité. Pas une mince affaire donc tout en sachant que certains de ces services changent de ports assez souvent.

9 Annexe

9.1 Images

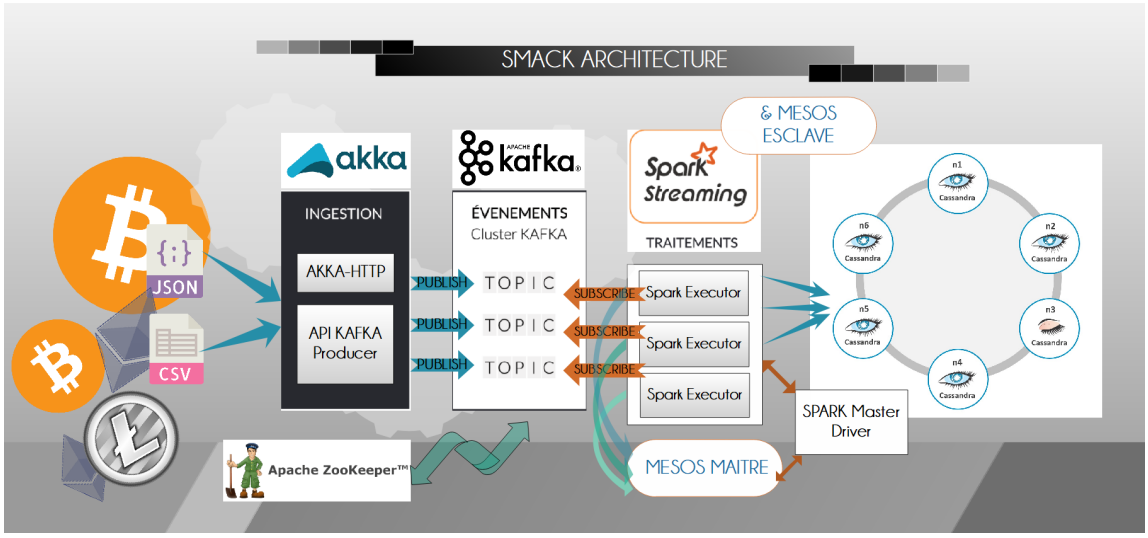

Figure 12: Smack architecture summary

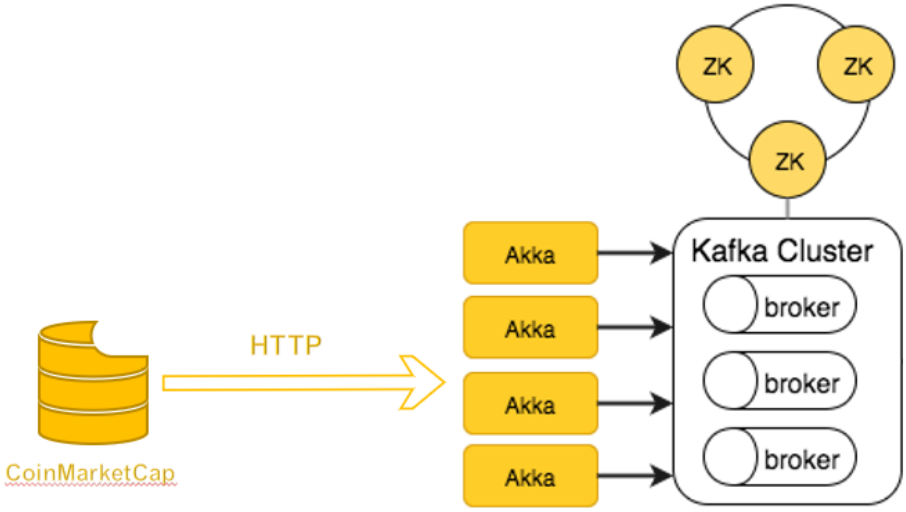

Figure 13: Architecture du Producer

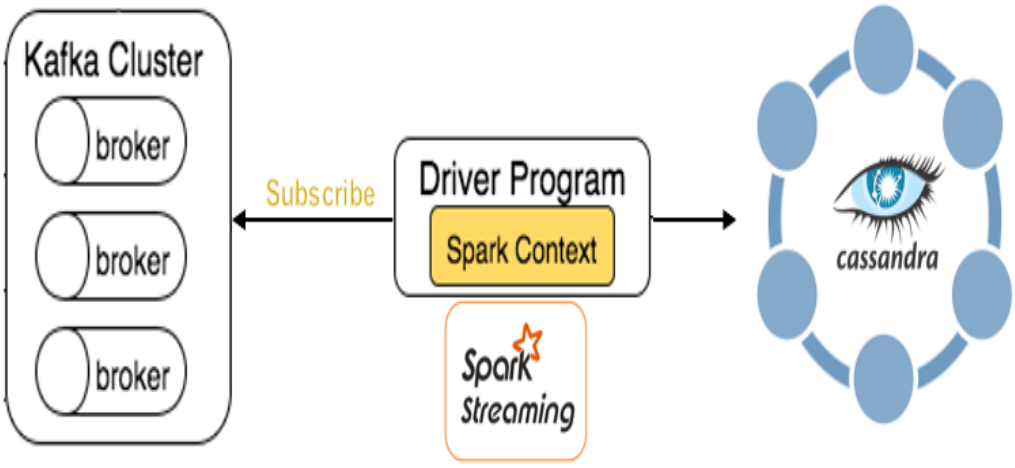

Figure 14: Architecture du Consumer

References

[1] https://doc.akka.io/docs/akka-http/current/scala/http/introduction.html

[2] https://kafka.apache.org/quickstart

[3] https://spark.apache.org/docs/latest/

[4] https://zookeeper.apache.org/doc/r3.3.3/zookeeperStarted.html

[5] https://www.terraform.io/intro/index.html

[6] http://docs.ansible.com/ https://cassandra.apache.org/doc/latest/getting_started/querying.html

https://www.digitalocean.com/community/tutorials/how-to-run-a-multi-node-cluster-database-with-cassandra-on-ubuntu-14-04

https://www.datastax.com/dev/blog/how-to-create-a-datastax-enterprise-cluster-on-a

http://docs.datastax.com/en/landing_page/doc/landing_page/current.html