Documentation technique du projet SMACK

Ait El Kadi Ouiame, Ansari Othmane, Boudouin Philippe,
Boukrouh Insaf, Giroux Baptiste, Gouttefarde Léo,
Kodjo Edoh, Nahyl Othmane

Mardi 24 Janvier 2017

Table des matières

1 Les différents composants de la stack **2**

1.1 Spark 2

1.2 Mesos 4

1.3 Akka 5

1.4 Cassandra 6

1.5 Kafka 8

2 Structure générale et description du projet **9**

2.1 Installation one-liner 9

2.2 Lancement des scripts depuis le cluster 9

2.3 Mise à jour de l'environnement d'installation 10

2.4 Fonctionnement / organisation des scripts 10

2.5 Scripts d'installation / configuration 10

2.6 Démarrage / arrêt des composants de la stack et automatisation 11

2.7 Tolérance aux fautes 11

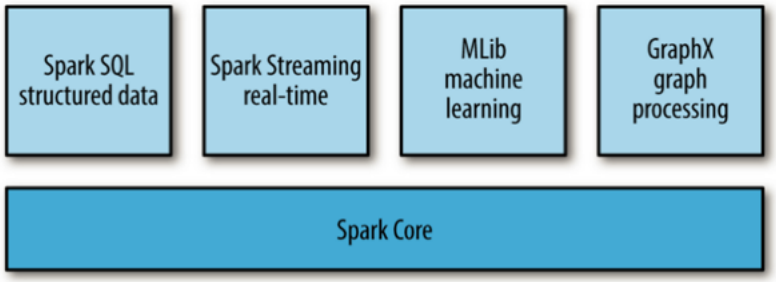
1 Les différents composants de la stack

1.1 Spark

Apache Spark est un framework open source de traitements Big Data, construit pour effectuer des analyses sophistiquées et conçu pour la facilité d'utilisation et la rapidité. Spark permet une parallélisation massive avec une technologie in-memory. Les applications sur clusters Hadoop sont exécutées jusqu'à 100 fois plus vite en mémoire, 10 fois plus vite sur disque.

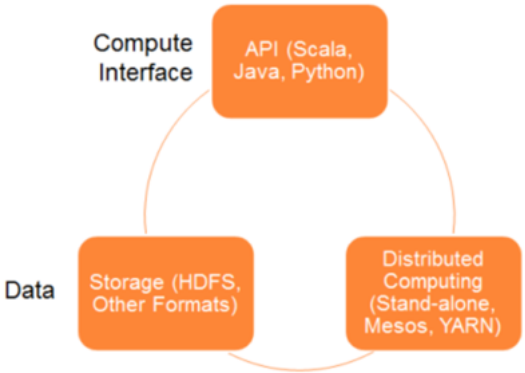
Il est écrit en Scala et s'exécute sur la machine virtuelle Java (JVM). Les langages supportés actuellement pour le développement d'applications sont : Scala, Java, Python, Clojure, R. Spark est complet et unifié pour répondre aux besoins de traitements de données (texte, graphe, etc.) en batch ou en flux temps-réel.

L'écosystème de Spark



1. **Spark Streaming** : Peut être utilisé pour traitement temps-réel des données en flux
2. **Spark SQL** : Permet d'exécuter des requêtes de type SQL
3. **Spark MLlib** : Librarie de machine learning qui contient tous les algorithmes et utilitaires d'apprentissage classiques (la classification, la régression, le clustering...)
4. **Spark GraphX** : La nouvelle API pour les traitements de graphes et de parallélisation de graphes

L'architecture de Spark



1. **L'API**
   L'API permet aux développeurs de créer des applications Spark en utilisant une API standard. L'API existe en Scala, Java et Python.

2. **Le stockage des données**
   Spark utilise le système de fichiers HDFS pour le stockage des données. Il peut fonctionner avec n'importe quelle source de données compatible avec Hadoop, dont HDFS, HBase, **Cassandra**, etc.
3. **Le cluster manager**
   Spark peut être déployé comme un serveur autonome ou sur un framework de traitements distribués comme **Mesos** ou YARN.

Les RDDs

Le concept clé de Spark est les RDDS. Un RDD (Resilient Distributed Dataset) est une abstraction de collection partitionnée d'enregistrements pouvant être distribuée entre machines de manière tolérante aux pannes.

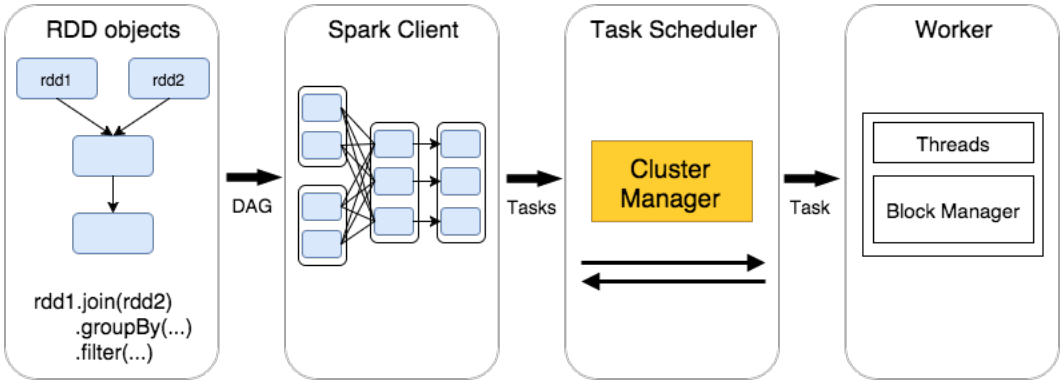
Les RDD supportent deux types d'opérations :

— Les transformations : les transformations ne retournent pas de valeur seule, rien n'est évalué, elles retournent un nouveau RDD.
— Les actions : les actions évaluent et retournent une nouvelle valeur.

Le workflow

Le workflow de Spark comporte quatre phases principales :

— Les RDDs (transformations et actions) forment le DAG (Direct Acyclic Graph)
— Le DAG est divisé en ensembles de tâches qui sont ensuite soumises au gestionnaire de cluster
— Un ensemble de tâche combine les tâches qui ne nécessitent pas de shuffling/répartition
— Les tâches sont exécutées sur les workers qui ensuite retournent les résultats au client



1.2 Mesos

Mesos est un gestionnaire de cluster basé sur l'architecture maître / esclave, qui permet d'abstraire les ressources des machines du cluster afin d'éliminer les problèmes de scalabilité.

Il permet aux datacenters de fonctionner comme s'ils étaient un unique serveur hébergeant plusieurs applications

Traditionnellement, les datacentres comportaient des clusters réservés pour un framework/application bien précise. Ceci impliquait des problèmes de scalabilité dès que des noeuds d'un cluster deviennent indisponibles. Mesos permet à plusieurs frameworks de cohabiter dans le même cluster.

Mesos se base sur un principe appellé Two-Tier Scheduling (Planification à deux niveaux) :

— De manière continue, les esclaves proposent leurs ressources disponibles au master
— Par une politique d'allocation précise, le master décide à quel framework il va attribuer les ressources
— Le scheduler du framework a le choix d'accepter ou de refuser l'offre des ressources
— Le master affecte par la suite les tâches aux différents esclaves capables de faire le travail.

Ainsi on peut déduire que le premier niveau de planification des tâches se fait au niveau du master selon la politique d'allocation, et le deuxième niveau se fait au niveau du scheduler du framework qui décide quelles tâches exécuter.

Mesos propose des mécanismes de tolérance aux pannes à plusieurs niveaux :

— Au niveau du master : Mesos propose un design hot-standby. Plus précisément, un cluster mesos peut contenir plusieurs master dont un est fonctionnel et les autres sont en mode stand-by. Ceci peut être mis en place par le service ZooKeeper qui permet d'élire le master du cluster. Les esclaves devront contacter ZooKeeper pour connaître le master élu.
— Au niveau des esclaves : Le master est responsable de monitorer le statut des esclaves. Quand un esclave ne répond plus aux messages du master, ceci est retiré de la liste des esclaves et le scheduler du framework qui a réservé le noeud est notifié pour replanifier la tâche échouée.
— Au niveau des tâches : De façon similaire à l'échec d'un noeud, le master notifie le scheduler du framework pour lui demander de replanifier la tâche dans un autre esclave à la condition que les ressources requises soient disponibles dans ce dernier.

1.3 Akka

Akka est un toolkit pour les applications Java et Scala distribuées.

Il introduit entre autres diverses abstractions de système distribué / parallélisme de haut-niveau comme les modèles d'acteur, de stream et de futurs.

1.3.1 Modèle Acteur

Le modèle d'acteur est une abstraction qui permet d'écrire des systèmes concurrents / distribués (évite d'avoir à utiliser des locks / threads etc).

1.3.2 Modèle Stream

Le modèle de stream correspond à une interface de stream permettant la communication de manière distribuée.

1.3.3 Modèle Future

Un futur est une structure de données permettant de récupérer le résultat d'opérations concurrentes. Ce résultat peut être accédé de manière synchrone (bloquant) ou asynchrone (non-bloquant).

Tolérance aux fautes

Akka permet la tolérance aux fautes et le paramétrage de la stratégie de tolérance à adopter. Autrement une stratégie de tolérance par défaut est utilisée.

1.4 Cassandra

Apache Cassandra est un système permettant de gérer une grande quantité de données de manière distribuée. Elle a été conçue pour être hautement scalable sur un grand nombre de serveurs tout en ne présentant pas de Single Point Of Failure. Cassandra fourni un schéma de données dynamique afin d'offrir un maximum de flexibilité et de performance.

Architecture

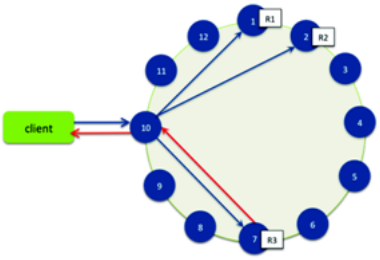
Cassandra aborde le problème de défaillance (failure) en employant un système distribué peer-to-peer où tous les nœuds sont identiques et les données sont réparties entre tous les nœuds du cluster. Chaque nœud échange des informations à travers le cluster toutes les secondes.

Les composants de Cassandra sont les suivants :

- **Gossip** : Permet de découvrir et partager des informations d'emplacement et d'état sur les autres nœuds d'un cluster Cassandra.
- **Partitioner** : Détermine comment distribuer les données entre les nœuds du cluster
- **Stratégie de placement de réplicas** : Détermine les nœuds sur lesquels placer les répliques (La première réplique de données est simplement la première copie)
- **Snitch** : Définit les informations de topologie que la stratégie de réplication utilise pour placer les répliques et les demandes de route de manière efficace.
- **Le fichier cassandra.yaml** : C'est le fichier de configuration principal de Cassandra

Du point de vue de l'application Client

Tous les nœuds de Cassandra sont égaux. Ainsi, une demande de lecture ou d'écriture peut interroger indifféremment n'importe quel nœud du cluster. Quand un client se connecte à un nœud et demande une opération d'écriture ou de lecture, le nœud courant sert de coordinateur du point de vue du client. Le travail du coordinateur est de se comporter comme un proxy entre le client de l'application et les nœuds qui possèdent la données. C'est lui qui a en charge de déterminer quels nœuds de l'anneau devront recevoir la requête.

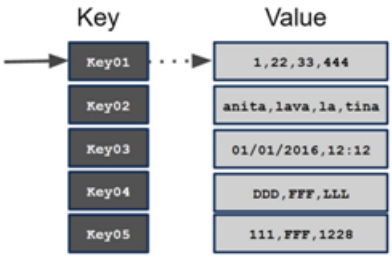


Caractéristiques

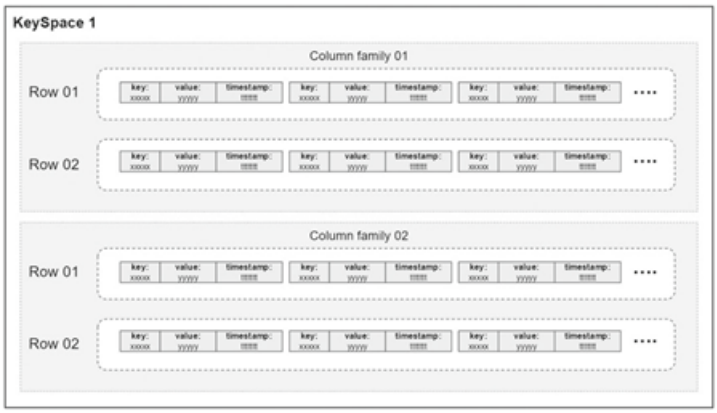
- **Tolérance aux pannes** : Les données d'un nœud sont automatiquement répliquées vers d'autres nœuds. Ainsi, si un nœud est hors service les données présentes sont disponibles à travers d'autres nœuds.
- **Décentralisé** : dans un cluster tous les nœuds sont égaux. Il n'y pas de notion de maître, ni d'esclave, ni de processus qui aurait à sa charge la gestion.
- **Modèle de données riche** : le modèle de données proposé par Cassandra basé sur la notion de clé/valeur permet de développer de nombreux cas d'utilisation dans le monde du web.
- **Élastique** : la scalabilité est linéaire. Le débit d'écriture et de lecture augmente de façon linéaire lorsqu'un nouveau serveur est ajouté dans le cluster.
- **Haute disponibilité** : possibilité de spécifier le niveau de cohérence concernant la lecture et l'écriture. On parle alors de Tuneable Consistency.

Modèle de données

Apache Cassandra stocke les données dans une suite triée de couples **clé / valeur**. On classifie ainsi Cassandra comme une base de données NoSQL orientée colonne.



- Une **ligne** est composée d'un ensemble de colonnes et identifiée par une clé
- Une **famille de colonnes** est un regroupement logique de lignes
- Un **Keyspace** est un regroupement de famille de colonnes

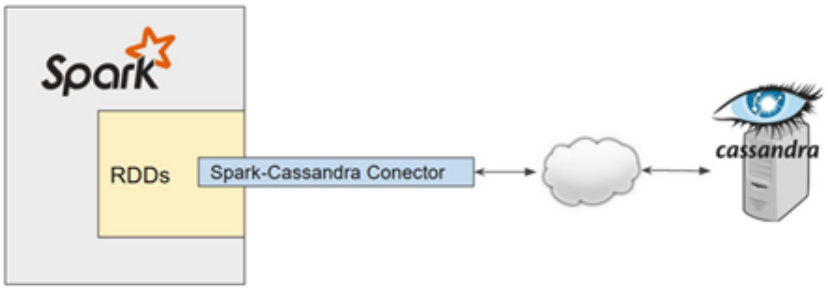


Connexion avec Spark

Apache Cassandra utilise l'architecture client/serveur. La connexion entre le client et le serveur peut se faire via :

- **Le CQLs** : c'est un shell pour Cassandra Query Language.
- **Les Drivers** : Il y a des drivers pour Cassandra dans presque tous les langages de programmation modernes (Python, Java, Scala, ...)

Pour faire la connexion entre Cassandra et la stack SMACK, et plus précisément avec Spark, nous avons besoin du « **Spark-Cassandra Connector** ». Ce client est spécial car il est destiné spécifiquement pour Spark, et pas pour un langage spécifié.

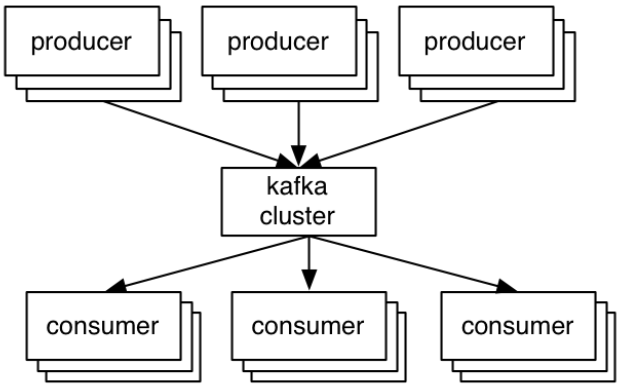


1.5 Kafka

Kafka est une plateforme de streaming distribuée pouvant être utilisée comme un agent de message (message broker). Elle permet à des applications qui échangent des messages en temps réel de les enregistrer et de les distribuer au fur et à mesure avec une latence faible et un débit élevé. De plus, Kafka permet de passer facilement à l'échelle grâce à sa notion de groupe de consommateur détaillé plus bas.

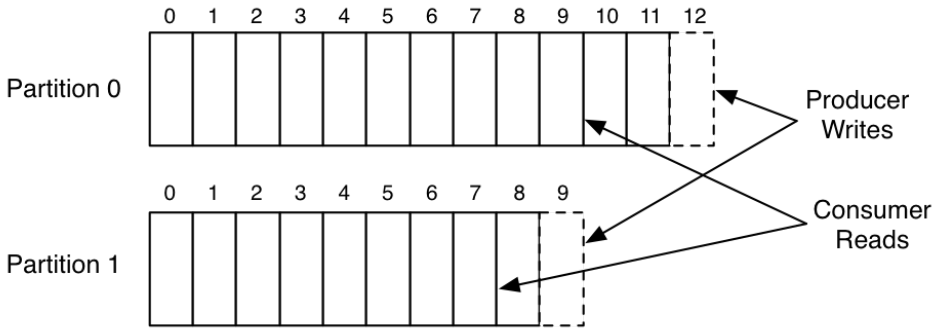
Principe de fonctionnement

Le modèle de fonctionnement de Kafka à haut niveau est simple, les producteurs envoient des messages (de n'importe quel type) à l'un des topic du cluster de Kafka et les consommateurs souscrivent à un ou plusieurs topics :



Un topic est en fait un log de tous les messages qui ont étés envoyés par les producteurs. Un topic se décompose en plusieurs partitions afin d'offrir un passage à l'échelle, il n'est donc pas nécessaire qu'une machine soit capable de contenir un topic en entier sur son disque.

La figure suivante montre un topic composé de 2 partitions :



Le producteur écrit chaque message à la suite de l'un des deux topics tandis que le consommateur lit les messages à l'offset qu'il désire (généralement le dernier offset lu + 1). Les messages sont stockés sur disque et disposent d'une durée de vie configurable au bout de laquelle ils seront supprimés, il est donc possible de les lire dans n'importe quel ordre mais aussi de les relire. Pour assurer la tolérance aux pannes, chaque partition est écrite sur plusieurs machines.

Groupe de consommateur

Afin de permettre le passage à l'échelle, Kafka propose une notion de groupe de consommateur, cela signifie que chaque message envoyé à un topic sera délivré à un consommateur de chaque groupe souscrivant à ce topic. L'application client utilisant Kafka peut regrouper les consommateurs par groupe à sa guise afin de répartir la charge et d'obtenir la redondance qu'il souhaite. Pour obtenir ce comportement, Kafka donne à chaque consommateur une liste de partitions de taille similaire à traiter qui est modifiée dynamiquement.

Garanties et performances

— Un topic répliqué N fois peut supporter N-1 pannes sans perdre de messages.
— Kafka nécessite une instance de serveur zookeeper pour fonctionner, celui ci utilise un algorithme proche de paxos et il est donc nécessaire d'avoir une majorité des machines sur lequel zookeeper s'exécute pour assurer le fonctionnement.
— Au sein d'une partition, Kafka maintient l'ordre des messages envoyés par un même producteur.
— Un consommateur voit les messages dans l'ordre dans lequel ils sont enregistrés sur le topic.

2 Structure générale et description du projet

Les différents scripts du projets sont réalisés en bash.

On en distaingue plusieurs types :

— les scripts communs contenants des fonctions, variables et autres données partagées
— les scripts d'installation et de configuration
— les scripts de désinstallation
— les scripts de lancements des composants de la stack
— les scripts d'arrêt des composants de la stack

2.1 Installation one-liner

L'installation initiale de l'environnement est réalisée à l'aide d'une unique commande à exécuter sur l'une des machines du cluster :

```
ZIP=~/setup.zip; cd ~; wget -O $ZIP \
  https://raw.githubusercontent.com/leogouttefarde/smack/master/setup.zip; \
  sudo apt -y install unzip; unzip -o $ZIP
```

Ceci permet de facilement pouvoir commencer une première installation sans avoir à effectuer des manipulations complexes.

2.2 Lancement des scripts depuis le cluster

Les différents scripts sont prévus pour être exécutés directement depuis l'une des machines du cluster de la stack. Cela permet d'en simplifier le fonctionnement.

2.3 Mise à jour de l'environnement d'installation

L'ensemble des scripts utilisés sont installés et mis à jour via le téléchargement et l'extraction d'un fichier ZIP dédié disponible en ligne. Cela permet de simplifier le processus d'installation.

2.4 Fonctionnement / organisation des scripts

2.5 Scripts d'installation / configuration

install_all.sh

Permet le lancement de l'installation sur toutes les machines en appellant le script `server_install.sh` sur chacune de celles-ci. Le script est appellé de manière différente selon l'hôte en question (master ou slave). Le script inverse est : `uninstall_all.sh`

update_setup.sh

Met à jour les différents scripts sur tous les noeuds du cluster.

server_install.sh

Permet de lancer l'installation sur une machine. Premièrement, le script vérifie si l'installation n'est pas encore faite avant d'installer et de configurer les hôtes, mmonit et les autres dépendances. L'installation est bien évidemment différente selon le type de l'hôte, ce contrôle est fait à l'aide d'un argument passé au script. Le script inverse est : `server_uninstall.sh`

install_hosts.sh

Permet de configurer le fichier `/etc/hosts` sur toutes les machines en appellant le script `declare_hosts.sh` sur chacune de celles-ci. Le script inverse est `uninstall_hosts.sh` .

declare_hosts.sh

Permet de configurer, ou de rétablir l'ancienne configuration des hôtes en cas de désinstallation.

install_node_deps.sh :

Permet d'installer les dépendances des différents frameworks et de les configurer selon le type de la machine (Master, esclave ou manager).

config_mmonit.sh

Permet de configurer les processus et hôtes monitorés par Monit.

install_cassandra.sh

Permet l'installation et la configuration de cassandra sur une machine. L'installation est identique sur toutes les machines (peer to peer).

2.6 Démarrage / arrêt des composants de la stack et automatisation

run_cassandra_cluster.sh

Permet de lancer les processus Cassandra sur tous les noeuds du cluster en appellant `run_cassandra_local.sh` sur chaque noeud. Le script inverse est `stop_cassandra_cluster.sh` .

run_cassandra_local.sh

Permet de lancer les processus Cassandra sur un hôte, il détecte aussi l'état du noeud Cassandra pour décider de la manière du lancement (start normal ou remplacement). Le script inverse est `kill_mesos_processes.sh` .

run_kafka_cluster.sh

Permet de lancer le scheduler kafka et de démarrer les brokers. Le nombre de ceux-ci est passé en argument. Le script inverse est `stop_kafka_cluster.sh` .

run_mesos_cluster.sh

Permet de démarrer le cluster mesos sur tous les maîtres et esclaves du cluster. Le script inverse est `stop_mesos_cluster.sh` .

run_mesos_local.sh

Permet de démarrer les daemons mesos. La commande de démarrage est différente selon le type de l'hôte. Le script inverse est `kill_mesos_processes.sh` .

add_kafka_topic.sh

Permet d'ajouter un topic en interactif selon les besoins de l'application à déployer sur le cluster.

2.7 Tolérance aux fautes

Pour l'aspect tolérance aux fautes et redémarrage automatique des services qui pourraient tomber en panne, nous avons utilisé l'outil demandé, Monit.

De plus, nous avons installé l'interface web M/Monit qui permet de facilement superviser l'intégralité du cluster et effectuer des actions de monitoring.

2.7.1 Installation

Pour l'installation, on procède d'abord à un arrêt de Monit au cas ou une ancienne version serait déjà en exécution.

Ensuite, on installe Monit depuis les packages de la machine. Seulement, pour M/Monit il faut au moins la version 5.20.0 de Monit, qui n'est pas disponible sur les packages de Ubuntu 16.04 LTS. Ainsi on télécharge la version 5.20 et on remplace l'exécutable installé par la bonne version. Après cela, on lance Monit.

Vient alors l'installation de M/Monit, pour cela on télécharge simplement l'archive que l'on extrait dans le répertoire home.

2.7.2 Configuration

Le travail de configuration est effectué au sein du script `config\ mmonit.sh` .

On installe d'abord un service systemd qui sera en charge de gérer M/Monit et on le lance.

Ensuite on configure Monit de manière à ce que toutes les machines machines puissent collecter les données Monit de chaque machines au sein de M/Monit.

Finalement on ajoute les différents services à monitorer à l'aide de Monit : Cassandra, Mesos, Kafka. Pour prendre on compte la nouvelle configuration de Monit, on effectue aussi un rechargement.

2.7.3 Architecture SPOF-Less

Le composant principal permettant l'absence de single point of failure dans notre architecture est le service de coordination ZooKeeper. Tous les mécanismes de tolérence aux fautes proposés par Mesos se basent sur ce service.

En effet, notre architecture comporte 3 masters, deux en standby et un master en marche. Ceci permet la tolérance d'une panne au niveau des masters selon le principe du quorum.

En cas de panne du master courant, ZooKeeper permet l'élection d'un nouveau master, et grâce au principe de checkpointing, les tâches courantes sont reprises sans perte d'avancement.