Projet SDTD - Pile SMACK

Analyse de la corrélation entre données météorologiques et tweets

Documentation

19 / 01 / 2018

Cabrera - Carre - Colina - Deloche - Dubois - Lefoulon - Meyer - Parizot

1. Cas d'étude

Pour illustrer la pile SMACK (Spark Mesos Akka Cassandra Kafka), nous avons décidé d'étudier la corrélation entre la température de New York et la tendance sentimentale des tweets de cette ville. Par exemple, la pluie induit-elle des messages moins joyeux ?

Pour ce faire, notre application se découpe en quatre parties, chacune faisant appel à un ou plusieurs composants SMACK :

1. Récupération des données : à l'aide d'APIs web, nous récupérons la température et les tweets de New York.
2. Pré-traitement : les données sont nettoyées et pré-traitées. En particulier, nous effectuons ici l'analyse sentimentale des tweets.
3. Stockage : pour conserver un historique des données, nous les stockons en base.
4. Analyse : un serveur web permet aux clients d'obtenir pour une période donnée la liste des températures horaires, des sentiments moyens correspondants ainsi que leur corrélation.

Notre projet consistait à créer une telle application fonctionnelle, à en automatiser le déploiement sur Amazon Web Services (AWS) et à la rendre tolérante aux pannes.

2. Introduction aux composants de la pile SMACK

2.1. **S**park

Spark est un outil permettant de faire du **calcul réparti**. Son API fournit des abstractions pour découper un problème algorithmique et en paralléliser l'exécution.

Spark se base sur le **modèle Map-Reduce**, composé de deux étapes :

1. Map : le problème est décomposé de façon récursive en sous-problèmes envoyés sur des machines indépendantes
2. Reduce : après que chaque noeud a résolu son sous-problème, il remonte le résultat à son parent, et ce jusqu'à la racine

Un exemple classique est le calcul du minimum d'un tableau. Considérons par exemple `min(1, 2, 3, 4)` :

1. Spark découpe le tableau en sous-tableaux `[1, 2]` et `[3, 4]`, qu'il envoie à des machines indépendantes.
2. Chaque machine calcule le minimum de son sous-tableau (respectivement 1 et 3) et le retourne au maître.
3. Le maître calcule le minimum des minima, soit `min(1, 3)`, i.e. 1.

2.2. **M**esos

Mesos permet de gérer un **cluster de machines** et de répartir des tâches à effectuer de manière totalement transparente. Il repose sur un fonctionnement de type **maître-esclave** :

- Le maître reçoit les tâches à effectuer et va essayer de les attribuer à une des machines du cluster.

- Chaque esclave fournit au maître les ressources (RAM, CPU, espace disque, …) dont il dispose et effectue les tâches qui lui sont attribuées.

Pour garantir la haute disponibilité, plusieurs maîtres peuvent être lancés. Un seul d'entre eux est actif, les autres étant prêts à prendre le relais au cas où il viendrait à disparaître. L'élection d'un nouveau maître est gérée par le service **Zookeeper**.
La demande de ressources auprès du cluster Mesos se fait à travers un ordonnanceur de tâches: **Marathon**. Marathon gère le cycle de vie des services: lancement, passage à l'échelle, redéploiement en cas de panne et arrêt. Le rôle de service-discovery est joué par **Marathon-LB** (reverse-proxy basé sur HAProxy), qui permet aux différents services de communiquer à l'aide de leurs ports uniquement. Il fait également de l'équilibrage de charge entre les différentes instances d'un même service.
L'ensemble de ces services est déployé au travers de conteneurs **Docker**, à l'exception des **Mesos-Slave**. Cela permet une gestion simplifiée du cluster tout en évitant de déployer des conteneurs dans des conteneurs.

2.3. **A**kka

Le composant Akka permet le développement d'applications pilotées par messages, distribuées et concurrentes sur la JVM (java & scala), avec un haut niveau d'abstraction. Akka utilise le modèle des acteurs comme modèle concurrentiel. Les acteurs sont des entités isolées entre elles qui communiquent de manière asynchrone par messages et disposent chacune d'une boite aux lettres. Les messages sont traités de manière séquentiel et chaque acteur est restreint à un traitement spécifique (single responsability principle). La chaîne de traitement se traduit donc par une chaîne d'acteurs reliés par une relation père/fils : après avoir effectué son traitement l'acteur père supervise les fils qu'il a créé, au niveau de leur travail et de leur cycle de vie. Cette isolation permet la résolution efficace des traitements et assure une tolérance aux pannes.

2.4. **C**assandra

Cassandra est une **base de données distribuée**. Les données sont réparties sur plusieurs machines ayant toutes le même rôle dans le cluster : il n'y a donc aucun point individuel de défaillance. Les noeuds communiquent entre eux via un protocole propre à Cassandra (Gossip), et échangent des informations sur leur état, sur les données stockées… L'utilisateur peut régler le niveau de réplication des données (un plus haut niveau de réplication entraîne un coût de stockage plus grand mais permet de résister à la perte de plus de noeuds). Lorsqu'on veut ajouter un noeud à un cluster d'instances Cassandra existantes, on lui fournit des IP de noeuds à contacter (les seeds). Il existe de plus un langage pour effectuer des requêtes à Cassandra, CQL (Cassandra Query Language), dont la syntaxe est proche du SQL.

Cassandra est gérée sur notre application indépendamment de Marathon-LB. En effet, lors de l'association de nouvelles machines au cluster, les messages envoyés via Gossip contiennent l'IP de la machine source. Comme les Cassandra sont dans des Dockers, l'IP hôte est inaccessible et ne peut pas être fournie par Marathon. Notre solution est donc basée sur le lancement de Cassandra sur les noeuds master (on peut au besoin ajouter des Cassandra sur des noeuds slaves)

dont on aura précisé les IPs à Spark afin qu'il puisse communiquer avec Cassandra. Les données sont ainsi répliquées 3 fois, ce qui permet de résister à la perte de 2 masters.

2.5. **K**afka

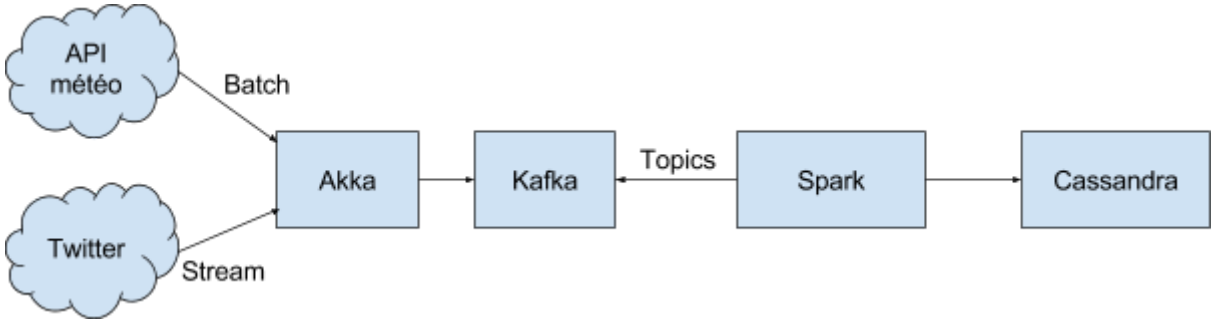
Kafka est un outil de **publication de messages entre applications.** Il permet à une application A de transmettre des données à des applications B et C sans en connaitre l'existence. On peut ainsi faire une analogie entre Kafka et un flux RSS.

Il est important de comprendre que Kafka ne fait pas du publish/subscribe classique, dans le sens où **il ne notifie pas les abonnés lui-même** (push). Au contraire, ces derniers doivent lui demander à intervalle régulier (pull) si de nouveaux messages sont disponibles. Ce choix d'architecture permet à Kafka de passer facilement à l'échelle puisqu'il n'a pas à retenir lui-même la liste des abonnés.

En plus de faire de la transmission de message, **Kafka joue le rôle de tampon**. Comme il ignore quand les abonnés réclameront les derniers messages, il est obligé de les stocker. Cela permet de facilement faire communiquer deux applications non synchronisées. Par exemple, si A envoie des messages toutes les minutes mais que B ne peut les traiter que toutes les heures, ce n'est pas gênant vu que Kafka les conservera en attendant.

3. Architecture de l'application

3.1. Architecture côté API



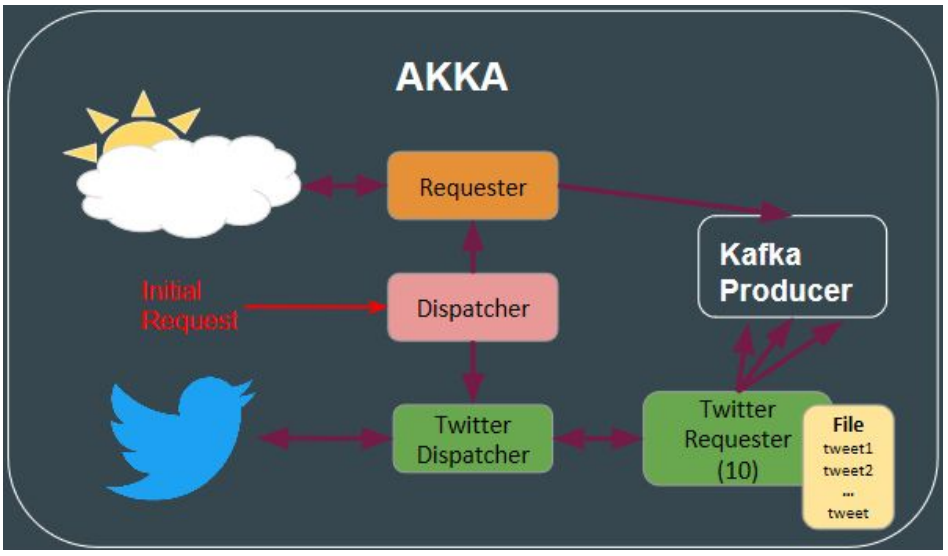
Modélisation de la pile côté API

Pour commencer, Akka récupère les données via les API. Pour pour ce faire, un acteur principal, le *dispatcher*, reçoit le message initial et le transfère à l'acteur fils correspondant.

Pour les températures, il existe un acteur fils par API, qui transmet la requête à l'API correspondante et récupère les température chaque minute en micro-batch. Un traitement est alors effectué sur les données reçues au format JSON pour ne garder que les informations pertinentes et le résultat est envoyé à un *producer* Kafka en le labellisant avec le topic *temperature*.

Pour les tweets, l'acteur fils envoie la requête à l'API Twitter pour ouvrir le stream, puis transmet les réponses sous forme de json au *twitterDispacher.* Celui-ci

répartit les messages entre 10 acteurs fils pour mettre en forme le json et envoyer le résultat au producer Kafka, avec le topic *tweet*.



Une instance de Spark est créée par topic. Elle s'y abonne puis demande régulièrement à Kafka les derniers messages grâce à la fonctionnalité de *streaming*. Les messages (températures ou tweets) sont alors prétraités puis stockés dans la base de données.

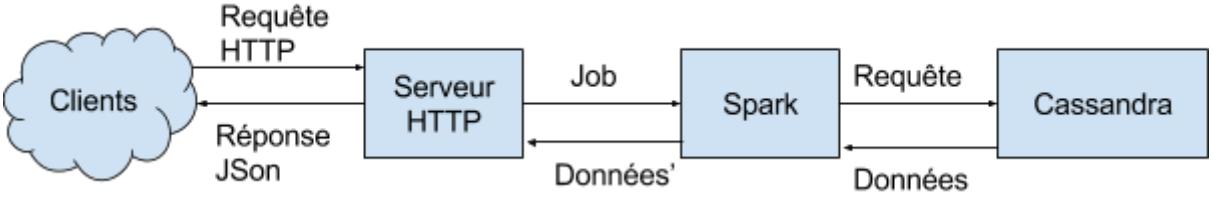
Le prétraitement des températures est inexistant. Par contre, quand on reçoit un tweet, on en extrait le sentiment général sous forme d'un entier entre 0 et 4 :

0	1	2	3	4
VERY NEGATIVE	NEGATIVE	NEUTRAL	POSITIVE	VERY POSITIVE

Pour extraire ce sentiment, nous utilisons la librairie Stanford CoreNLP, fournissant des ressources pour le traitement du langage naturel, avec entre autre l'extraction de sentiment d'un texte.

Une fois nos données pré-traitées, elles sont prêtes à être stockées. Spark se connecte à une instance de Cassandra et y ajoute alors les données, qui sont ensuite répliquées automatiquement à travers les noeuds du cluster Cassandra.

3.2. Architecture côté client



Modélisation de la pile côté client

Plusieurs solutions ont été envisagées pour effectuer les requêtes clients vers l'application. La solution retenue a été de mettre en place un **serveur HTTP** minimaliste en Scala par le biais de **Akka**. Ce serveur a l'avantage d'être extrêmement léger et hautement modulable.
Comme tout serveur Web, celui-ci fonctionne par routage, redirigeant les demandes clients vers la ressource associée. Une seule route est disponible pour le moment, permettant au client d'obtenir une corrélation entre températures et humeurs des tweets sur une période donnée en paramètre.

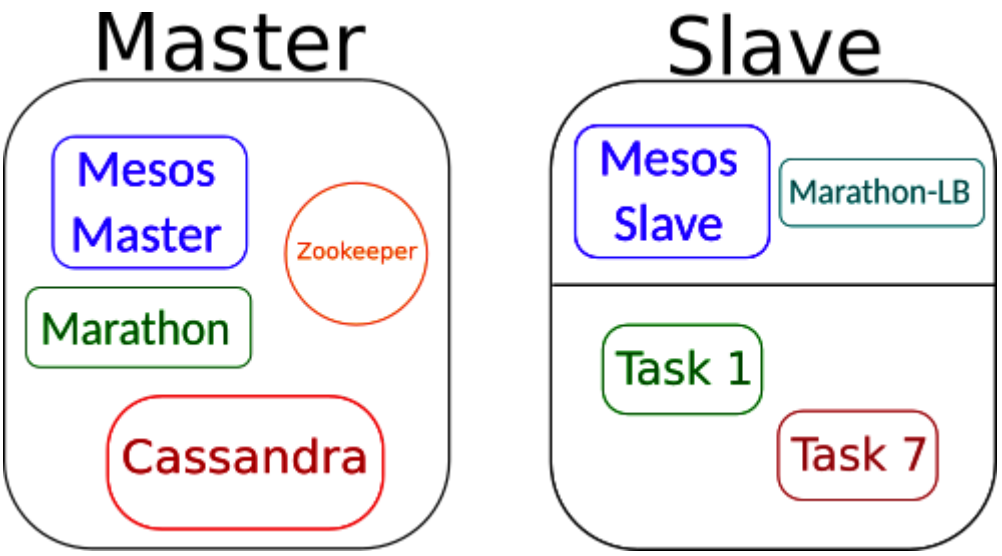
A la réception d'une requête, le serveur se connecte à un service **Spark** et lui transmet un job à accomplir. Tout le processus est ensuite effectué en boîte noire pour le serveur qui se contente d'attendre une réponse.
Le service Spark lui-même est connecté au service **Cassandra** afin d'obtenir toutes les données nécessaires à son travail.

3.3. Architecture du cluster Mesos

Le cluster Mesos est composé de deux types de machines : Master ou Slave.

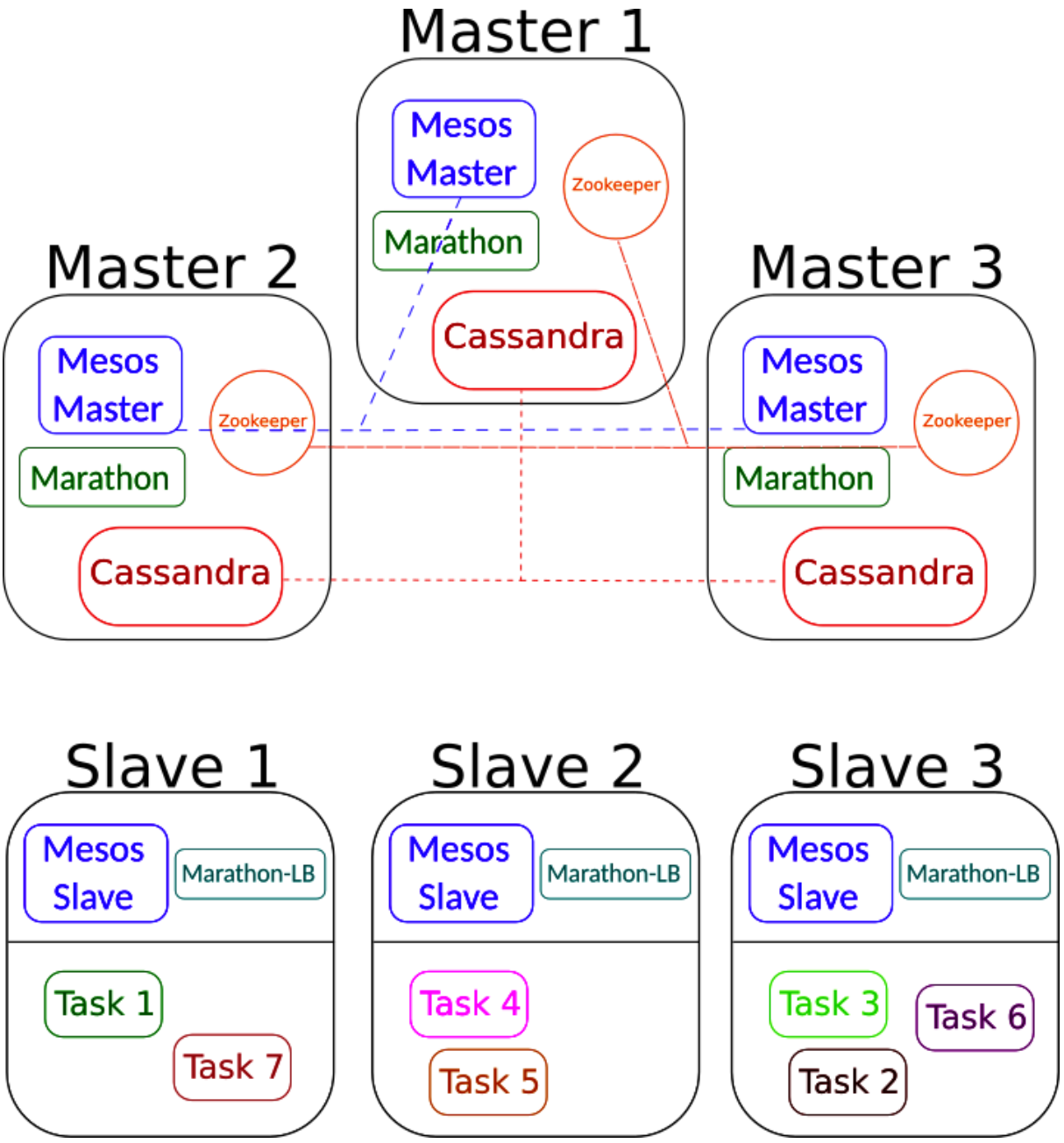
Une machine Master héberge une instance de Mesos Master, de Marathon, de Zookeeper et de Cassandra. Un seul des Mesos Master est actif, les deux autres sont prêts à prendre la relève en cas de panne. L'élection se fait par Zookeeper. Les trois Masters ensemble fournissent donc un service Mesos Master, Marathon et Cassandra résistant à deux pannes.

Une machine Slave héberge une instance de Mesos Slave et de Marathon-LB, ainsi que les tâches assignées par Mesos Master. Le service Mesos Slave permet à la machine de se raccorder au Mesos Master qui lui assignera des tâches à déployer. Marathon-LB permet aux services de communiquer entre eux de manière transparente, en s'identifiant uniquement par leurs ports.



Modélisation des machines du cluster Mesos

Le cluster final est composé de trois machines Master et de N (nombre variable, potentiellement dynamique) machines Slave. Les machines Master forment un cluster résistant aux pannes et (re)déployant dynamiquement tous les services (à l'exception de Cassandra) sur les machines Slave.



Modélisation du cluster Mesos

4. Gestion des ressources

4.1. Conteneurs Docker

Tous les composants de notre pile tournent dans des **conteneurs Docker**. Docker est un outil permettant de créer des conteneurs logiciels isolés du reste de la machine, fournissant des outils pour automatiser le déploiement de ces derniers sur différentes machines.

Nous avons créé un Dockerfile (fichier décrivant le fonctionnement d'un conteneur, les fichiers dont il dépend, ses variables d'environnement, les commandes à lancer…) par application, que nous avons hébergé publiquement sur un compte **Dockerhub**, ce qui permet de les rendre accessible depuis toutes les machines. Ce sont ces Dockerfiles que Marathon utilise pour déployer nos applications sur les machines AWS.

L'intérêt d'utiliser des Dockers est d'une part de **faciliter le déploiement**, et d'autre part de **garantir la sécurité** et le bon fonctionnement des applications : le fait qu'ils soient dans des conteneurs garantit qu'un composant ne peut pas agir sur la machine hôte ni être en conflit avec un autre composant. Cela permet enfin de **limiter les ressources allouées à une application**, et donc à en faire tourner plusieurs sur une même machine sans craindre qu'une ne monopolise les ressources.

4.2. Automatisation du lancement et de l'arrêt

L'automatisation du lancement et de l'arrêt du système sont gérés par des scripts Python dans le dossier `deployment`.

Le script de lancement est séparé en quatre parties distinctes et nécessite au préalable d'avoir configuré un compte AWS (cf. `deployment/README.md`):

- Création des instances sur AWS
- Déploiement des frameworks nécessaires sur les maîtres (Zookeeper, Mesos-Master, Marathon & Cassandra-Seed)
- Déploiement des frameworks nécessaires sur les esclaves (Mesos-Slave, Marathon-LB)
- Lancement des composants de la pile dans un ordre cohérent (exemple : Zookeeper avant Kafka) via l'API REST de Marathon

Le script d'arrêt est lui constitué de deux parties :

- Arrêt des services dans un ordre cohérent (ex : Kafka avant Zookeeper) à l'aide de l'API REST de Marathon
- Extinction des machines AWS

Le dossier comporte également des scripts utilitaires (par exemple pour lister les machines allumées), décrits dans le README.

4.3. Haute disponibilité

La **haute disponibilité** des composants peut être garantie de deux façons :

- **Par réplication** : plusieurs instances d'un même service peuvent être présentes sur le cluster. La perte d'un nombre restreint d'instances de ce service peut donc être supportée, au moins temporairement.
- **Grâce à Marathon** : les services lancés par Marathon tournent dans des containers Docker, et sont relancés dès que leur mort est détectée.

Certains composants ne sont pas critiques : le redéploiement par Marathon en cas de panne suffit.

Service	Criticité	Nombre d'instances	Redéployable par Marathon
Mesos Master	Forte	3	Non
Mesos Slave	Forte	N	Non
Zookeeper (Mesos)	Faible	3	Non
Marathon	Moyenne	3	Non
Akka (Température)	Forte	1	Oui
Akka (Tweets)	Forte	1	Oui
Kafka	Forte	3	Oui
Zookeeper (Kafka)	Faible	1	Oui
Spark (Température)	Forte	3	Oui
Spark (Tweets)	Forte	3	Oui
Cassandra	Forte	3	Non
Serveur Web (Spark + Akka)	Forte	1	Oui

N : Nombre de machines esclaves dans le cluster

5. Fonctionnement de l'application (de démonstration)

5.1. But de l'application

Le but de cette application est d'étudier la corrélation entre les données météorologiques et les tweets reçus dans un même intervalle de temps. Un traitement est effectué sur les données et son résultat est stocké dans la base de données. Lorsqu'un utilisateur souhaite connaître la corrélation entre tweet et température entre deux dates, il fait une requête et le résultat lui est envoyé, avec notamment le résultat de la corrélation.

5.2. API et données

5.2.1. Données météorologiques

Les données météorologiques utilisées sont tirées de 3 API différentes, OpenWeatherMap, WeatherBit et Apixu. Ainsi on peut croiser les données obtenues pour s'assurer de leur pertinence.

On récupère uniquement la température, la source et la date de réception des données, une fois par minute et par application, en micro batch, et on stocke cette information dans Cassandra.

5.2.2. Tweets

Les tweets sont streamés en continu grâce à l'API Twitter. On récupère l'identifiant du tweet, le texte et la date de réception, puis on effectue une analyse de sentiment sur le texte et on stocke le résultat obtenu dans Cassandra.

5.3. Requêtes client

Pour interagir avec l'application, il suffit au client d'entrer dans un navigateur ou à l'aide de la commande curl l'adresse du serveur HTTP et de son port, ainsi que la route associée à sa requête.
Pour le moment seule une route est disponible, permettant au client d'obtenir une corrélation entre températures et humeur des tweets sur une durée définie via des paramètres POST. La réponse obtenue est en JSon.
Exemple de requête pour sélectionner une période allant du 1 janvier à 2h au 1 janvier à 3h :

```
curl -H "Content-Type: application/json" -X POST -d '{"begin":"2018-01-01 02, "end";"2018-01-01 03"}" "http://@serveur:8080/weather"
```

Cette requête renvoie au client la réponse suivante :

```
{"res":[
  {
    "hourSlot":"2018-01-01 02",
    "feeling":1.67,
    "temperature":7.0
  },
  {
    "hourSlot":"2018-01-01 03,
    "feeling":2.0,
    "temperature":7.0
  }],
  "r2":0.0
}
```

6. Tests et mesures

Les temps de déploiement et d'arrêt dépendent principalement de la quantité d'instances lancées sur AWS ainsi que de la duplication des services. Le script de déploiement (`deployment/start.py`) mesure le temps d'exécution de chaque étape. De plus, les comptes AWS dont nous disposons rendant difficile le lancement d'un grand nombre de machines, nous avons décidé de tester ces temps pour un déploiement minimaliste qui consiste ici en un cluster de 7 machines (3 maîtres et 4 esclaves) avec des services non dupliqués. Il s'agit donc d'une pile minimaliste n'assurant la haute disponibilité que via le redéploiement des tâches fautives par Marathon. Pour une telle pile, **le lancement** avec nos scripts d'automatisation **prend en moyenne 8,5 minutes**, dont un total de 3 minutes d'attente pour certaines tâches afin d'assurer une bonne mise en place (ex:

attente après la création des instances AWS avant de les configurer). **L'arrêt s'effectue quand à lui en 4 secondes en moyenne**.
Toujours avec cette configuration, nous pouvons également tester le temps de redéploiement lorsqu'un service ou une instance meurt. Dans le cas où une des machines maîtres devient fautive, la réélection via Zookeeper se fait presque instantanément. Le changement de maître n'influe pas sur les services lancés par Marathon et notre application reste donc active.
Dans le cas où un service unique meurt, Marathon va relancer ce service. Le temps nécessaire va varier en fonction du service en question et, dans notre cas de test, l'application sera à l'arrêt jusqu'au redéploiement du service. **Le temps de relancement** va être identique au temps de lancement pour chaque service **dans le cas où l'image Docker n'est pas encore présente sur la machine** :

- Akka : **27s**
- Cassandra : **18s**
- Kafka : **12s**
- Spark : **28s**
- Zookeeper : **9s**
- Client : **13s**

Quand Marathon relance le service **sur une machine qui possède déjà l'image Docker utilisée**, **le redéploiement d'un service ne prend alors que 5 secondes en moyenne**.
Enfin, dans le cas ou une machine meurt, le temps de redéploiement dépendra des services qui étaient initialement situés sur l'instance en question. Le script `deployment/benchmark.py` mesure le temps de redémarrage du client par Marathon. Il est possible de le compléter facilement pour mesurer le temps de redémarrage de chaque service.

Côté réception des données les mesures de benchmarking s'avèrent compliquées à réaliser. Pour les température, il n'y a pas de test à réaliser étant donné qu'on se contente de faire une requête par minute et par application. Pour les tweets, on réalise un stream des tweets fournis par l'API Twitter. Seul un certain pourcentage des tweets émis sont récupérés via le stream, il n'y a donc pas de flux théorique quantifiable. Ensuite, la quantité de tweets reçus dépend de l'activité au moment du stream et varie dans le temps.

Pour 1 acteur réalisant le streaming, en journée et en se focalisant sur une tranche ouest-est des Etats Unis, on récupère en moyenne 9,703 tweets par seconde. La mise en forme et l'envoie au consumer Kafka prennent plus de temps. Pour compenser on utilise un pool de 10 acteurs. Cela suffit si on se limite à la localisation de la ville de New York. En revanche pour suivre le débit en charge haute (tranche ouest-est des état unis en journée) il faut jusqu'à 100 acteurs et la vitesse de traitement varie. Le traitement et l'envoie d'un tweet au consumer prend en moyenne 4,504 secondes.

Côté client, nos tests nous ont permis d'obtenir **un temps de réponse moyen d'environ 6 secondes avec une variance d'environ 1 seconde** pour la ressource '/test', qui est le chemin réservé aux tests et traitant un nombre fixe de données connues. Ce temps relativement élevé est dû au fait que le serveur se connecte à la machine Spark à chaque requête, qui elle-même en retour doit se connecter à la machine Cassandra. C'est cette connection et la mise en route du job qui prend le plus de temps.

Pour mettre en valeur le poids de cette connection sur le temps de réponse, le cas du test de connection - i.e. un 'ping' retournant une chaîne de caractère - fournit un temps de réponse aller-retour moyen de 38ms, avec une variance de 1.5ms.

7. Principaux obstacles et solutions techniques apportées

Le principal problème sur un cluster Mesos est que **les différents services ne peuvent pas facilement connaître leurs adresses IP respectives**. Ils sont en effet créés par Marathon dans un docker sur une des machines du cluster, **choisie de manière aléatoire** (ayant tout de même assez de ressources disponibles). L'adresse IP d'un service n'est donc **pas fixée** au lancement de l'application, et **pourra changer** au cours du cycle de vie s'il vient à être redéployé.

Il a donc fallu implanter un système de **service discovery**. Nous avons opté pour **Marathon-LB** car il fonctionne en lien avec Marathon et se met à jour dynamiquement suivant les redéploiements des différents services. Il permet également de faire de l'**équilibrage de charge** entre plusieurs instances d'un même service. De manière à le rendre **hautement disponible**, ce service a été installé sur tous les noeuds esclaves du cluster.

Cette solution fonctionne très bien pour tous les services, sauf Cassandra. En effet, à cause de son implémentation et de ses protocoles de gossip, Cassandra a besoin de connaître à la fois sa propre adresse IP et celles des noeuds Cassandra *seeds*. Or, cela n'est pas réalisable avec notre choix d'implémentation, dû au fait que les noeuds Cassandra tournent dans des containers Docker et à cause des (re)déploiements dynamiques de Marathon. Nous avions plusieurs possibilités:

- Ne faire tourner qu'un seul noeud Cassandra avec Marathon. Cela enlève le besoin d'autoriser le protocole de gossip entre les noeuds, mais cette solution est très fragile: si ce noeud venait à tomber, toutes les données stockées seraient perdues.
- Faire tourner Cassandra en dehors du cluster Mesos. Cette solution permet d'avoir plusieurs machines Cassandra communiquant entre elles et donc de garantir l'intégrité et l'accessibilité des données, mais rend impossible le redéploiement dynamique et l'équilibrage de charge.

Lors de la soutenance, nous avions implémenté la première solution. Cependant, l'intégrité des données étant primordiale, nous avons finalement opté pour la deuxième solution.

Nous avons également rencontré des problèmes avec l'utilisation d'AWS, les comptes dont nous disposions limitant le nombre et la taille des instances pouvant être lancées. Il a donc été difficile de tester notre application dans son intégralité, avec une réplication systématique de tous les services.

8. Evolutions possibles

Notre application assure la haute disponibilité via Marathon, en relançant les tâches qui échouent pour une quelconque raison, et via Zookeeper afin d'assurer la présence d'une instance maître. Néanmoins, les machines défaillantes n'étant pas automatiquement relancées, à partir d'un certain nombre de fautes le service sera incapable de se maintenir.

Pour résoudre cela, il serait possible de mettre en place un système de HealthCheck réguliers (avec le framework Chronos par exemple) permettant de vérifier l'état d'une instance et de la relancer ou d'en ajouter au cluster en cas de problème. De plus, cela permettrait également de contrôler la charge des différents frameworks et de les scaler si nécessaire.

En ce qui concerne le déploiement et l'arrêt de notre pile, il serait également intéressant de mettre en place des vérifications afin de s'assurer du bon lancement des instances et des services plutôt que d'utiliser des sleep(). Cela permettrait d'optimiser le temps de lancement de l'application. La même chose peut être implantée pour l'arrêt où l'on pourrait vérifier la bonne terminaison des services avant d'effectuer le shutdown des instances.

Bibliographie

Akka
http://adiln.com/la-tolerance-aux-pannes-avec-akka-1-les-exceptions-regulieres/
http://bigdata-madesimple.com/smackspark-mesos-akka-kafka/
Client
https://doc.akka.io/docs/akka-http/current/introduction.html
Mesos
http://mesos.apache.org/documentation/latest/
Marathon
https://mesosphere.github.io/marathon/docs/
Marathon-LB
https://github.com/mesosphere/marathon-lb/blob/master/README.md