Tweetoscope

Abel CAPITANT, Sabrina MOCKBEL, Paul BREHAT

Novembre 2024

1 Introduction

L'objectif du projet Tweetoscope est d'appliquer les concepts vus lors du cours d'Ingéniérie logicielle en créant une application de composée de microservices qui analyse et traite les données de Twitter, afin de générer des statistiques en temps réel sur les hashtags.
Cette application utilisera les différentes composantes logicielles (Kafka, Docker, Kubernetes...) et organisationnelles (bonnes pratiques DevOps). En raison de changements récents dans les politiques d'accès à l'API de Twitter, nous utiliserons des Tweets préenregistrés pour simuler des données en direct.

2 Choix de l'architecture

Pour l'architecture nous avons decidé de faire 5 microservices :
— TweetsProducer : le service de production de tweets (offline à cause des changements de politique de l'API Tweeter)
— Filters : le service de filtration des tweets selon des critères specifiques
— HashtagExtractor : extraction des hashtags de la partie texte des tweets filtrés
— HashtagAggregator : aggregation des tweets afin de les compter
— HashtagVisualiser : partie client de l'application qui affiche un leaderboard des hashtags

Ces services sont en majorité stateless ce qui permet de paralleliser au maximum le pipeline de traitement des tweets. En efffet, les services de filtrage et d'extraction des tweets sont stateless. Cependant le service d'aggregation est stateful puisqu'il enregistre un état du pipeline.

La solution middleware Kakfa nous permet d'élaborer des stratégies de résilience en lien avec ces microservices.

Pour commencer nous avons besoin d'une instance de zookeeper afin de pouvoir initialiser les Brokers Kafka. Nous avons décidé de mettre en place 2 brokers Kafka sur les port 9092 et 9093. Cela nous permet d'augmenter le facteur de réplication de notre architecture à 2 afin d'être plus résilient en cas d'échec d'un des deux brokers. En effet, en cas d'échec du leader, le follower prend la place de leader et permet de continuer le service sans interruption.

Au niveau des topics, nous avons en avons 4 :
— raw-tweets : 2 partitions, facteur de réplication 2

— filtered-tweets : 2 partitions, facteur de réplication 2
— hashtags : 1 partition, facteur de réplication 2
— hashtag-counts : 1 partition, facteur de réplication 2

Au niveau des instances, nous avons :

— 1 TweetsProducer initialisé avec un scénario particulier (scenario, recorded ou random) qui est producer du topic raw-tweets
— 2 Filters (consumer-producer) appartenant au même groupe de consumers afin de distribuer le traitement des tweets du topic raw-tweets qui sont ensuite envoyés sur le topic hasthags
— 2 Extractors (consumer-producer) appartenant au même groupe afin de paralleliser (fonctionnement de load balancing de Kafka)
— 1 Aggregator (consumer-producer avec une Topology) qui créée une Ktable afin de conserver un état du système est abonné au topic hashtags. Ce topic est composé d'une seule partition donc tous les tweets sont lus par tous les groupes de consommateurs. L'Aggregator écrit ensuite dans le topic hastag-counts.
— 1 Visualiser abonné au topic hashtag-counts qui possède une seule partition afin que tous les groupes de consumers puissent lire la même partition.

Voici ci-dessous une représentation de notre architecture :

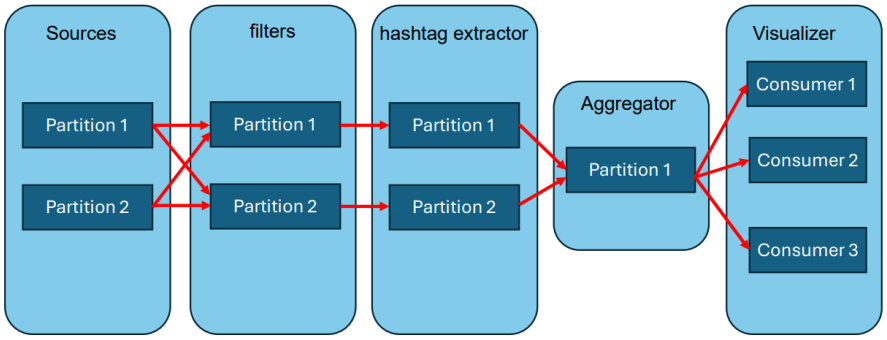

FIGURE 1 – Representation of the Architecture

3 Risques initiaux

Cette architecture est donc résiliente face aux menaces de pannes suivantes :

— Failure d'un broker
— Failure d'un filtre
— Failure d'un extracteur

En effet, en cas de failure de ces différents composants, nous n'aurions pas de problèmes de disponibilité du service client mais seulement des ralentissement dans la chaîne de traitement.

Le point le plus critique concerne le service d'aggregation. Il pourrait être dissipé en conservant un fichier temporaire (monté dans un volume extérieur) par exemple.

4 Atténuation des risques avec Kubernetes

Le logiciel Kubernetes nous permet de déployer nos différents services de manière automatique sur plusieurs noeuds et surtout d'ajouter un mécanisme de tolérance aux pannes en cas de failure d'un pod.

4.1 Présentation de notre utilisation de Kuberneetes

Tout d'abord, on a utilisé Docker Compose afin d'assembler les différentes images Docker, ce qui permet de définir et d'exécuter des applications multi-conteneurs. En effet, nous avons mis chaque composant du projet (producer, consummer, filter) dans des containers distincts, afin de permettre une gestion simplifiée des dépendances pour une meilleure ochestration des services.

C'est à ce moment-là que l'utilisation de Kuberneetes est primordiale, en effet, celui-ci permet de déployer l'ensemble de l'application sur un cluster. Il garantit que les services restent disponibles et résilients, même en cas de défaillance d'un des composants (un pod par exemple) grâce à sa capacité à automatiser les tâches de déploiement en ayant une excellente gestion des pannes.

4.2 Gestion des risques avec Kubernetes

Kuberneetes déploie automatiquement les services sur plusieurs noeuds du cluster, les services peuvent être éxécutés de manière distribuée, ce qui permet de réduire les risques de pannes. En effet, en cas de défaillance d'un pod, Kuberneetes va reconstruire automatiquement un nouveau pod afin de le remplacer, ce qui assure une continuité de service sans intervention manuelle et donc une grande atténuation des risques.

Ainsi nous deployons un pod par service afin de profiter des mécanismes de résilience au pannes de Kubernetes.

5 Conclusion

5.1 Experience

Pour ma part (Paul) j'ai apprécié découvrir de nouveaux outils devOps extrêmement utiles (notamment docker et kubernetes). Le plus difficile a été de comprendre le fonctionnement de maven avec de nombreuses subtilités en terme de direction, pom etc... et le debugging de Kakfa, notamment à cause de zookeeper qui conserve des logs dans des directions incongrues.

Pour ma part (Sabrina), j'ai pu revoir beaucoup de points du cours en particulier pour le choix de l'architecture et la chaine CI/CD. D'un point organisationnel, il a été difficile de me consacrer autant que je l'aurais voulu au projet au vu de difficultés propres, et d'un point de vue pratique, j'ai eu du mal à impléménter CI/CD d'un point de vue des tests sans trouver de réels solutions. Néanmoins ce projet m'a permis d'appliquer ce qui avait été vu en cours, pour revoir toutes les grandes parties d'une gestion de projet en DevOps.

5.2 Points de difficulté

Nous avons rencontré de nombreuses difficultés, en particulier du point de vue organisationnel. Il était difficile de rendre compatible nos différents emplois du temps compte tenus des obligations de chacun (fac, partiels...). D'un point de vue technique, nous n'avons pas réussi à conservé les .jar dans les artifacts du projet car ils sont trop volumineux (100Mo par jar). En effet, afin de conserver tous les différents scénarios dans le fichier compilé maven, nous avons inséré les fichier dans un dossier ressources du projet. Pour paré à cela et effectuer la tâche 15, nous avons dû ajouter les .jar au repo afin que les krunners aient connaissance des différentes directions lors du run de la pipeline.