SYSTÈMES DISTRIBUÉS POUR LE TRAITEMENT DES DONNÉES

Projet Hybride

Naima AMALOU

Amine KAMMAH

Kaoutar HAFYANE

Julien BITAILLOU

Janvier, 2021

Table des matières

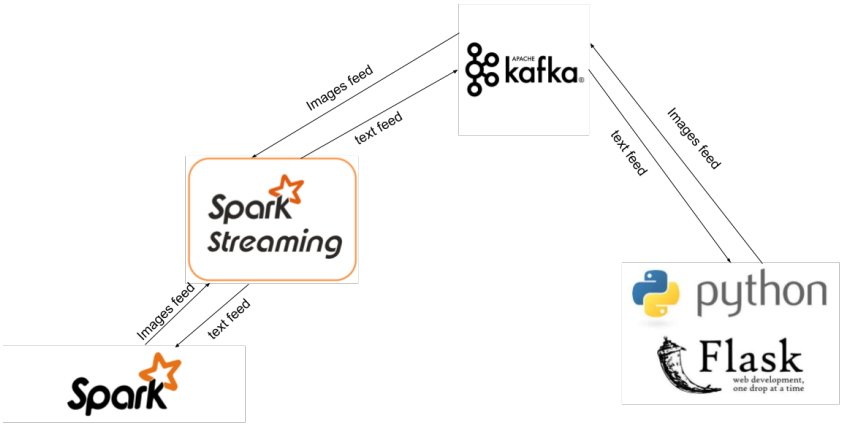
1 Résumé

Implémenter et déployer un service de reconnaissance de caractères optiques (OCR) et son infrastructure en utilisant :

— Terraform pour automatiser la création d'un cluster Kubernetes sur AWS EC2
— Kubernetes (en utilisant KubeADM) comme orchestrateur de service.
— Flask/JS pour gérer l'interaction avec les utilisateurs.
— Spark/Python pour un traitement distribué des données.
— Kafka comme agent de message entre Flask - Spark.

2 Architecture

Lorsqu'un utilisateur envoie des images depuis notre application de démonstration, Flask écrit les images envoyées sur le topic 'images-feed' de Kafka. A l'autre bout, Spark Streaming est connecté à ce topic de Kafka. Il reçoit donc l'image et commence le traitement. Une fois fini, l'exécuteur de Spark écrit le texte extrait dans un autre topic Kafka 'text-feed'. Flask reçoit cette nouvelle entrée et met à jour notre front-end pour afficher le texte extrait pour que l'utilisateur puisse le récupérer. Kubernetes permet d'orchestrer toutes ces composantes et garantit une haute disponibilité et tolérance aux pannes.



3 Automatisation

Pour démarrer le service, on utilise Terraform pour créer des instances EC2 M2.Medium et installer le cluster Kubernetes. Une fois terminé, Terraform lance automatiquement un script Bash sur le nœud Master permettant de créer les différentes composantes logicielles et fournit le lien d'accès externe à notre application de démonstration.

Grâce aux auto-scalers que nous avons mis en place, notre application peut s'ajuster automatiquement à la charge sur le service, sauf pour Spark. On explique ci-bas les limitations qui nous ont empêché d'atteindre une auto scalabilité de Spark.

4 Composantes créées

Notre pile logicielle est constitué de :

- 2 déploiements de Kafka limités à 0.5 cœur chacun.
- 2 déploiements de Zookeeper limités à 0.5 cœur chacun.
- 1 déploiement de l'application de démonstration limité à 1 cœur.
- 1 Spark Driver et deux exécuteurs limités à 1 cœur chacun.
- 5 auto-scalers de Kubernetes permettant d'ajuster le nombre de pods des déploiements de Kafka, Zookeeper et l'application de démonstrations. Tous configurés pour maintenir un pod au minimum et 3 pods au maximum, en se basant sur le taux d'utilisation du processeur (80

5 Application de démonstration

L'application de démonstration est une interface web développée en utilisant Html, Javascript et Flask qui consiste à envoyer des images à notre service de traitement de données à travers Kafka et afficher par la suite le texte extrait des images envoyées. Pour faire simple, l'utilisateur ne doit pas envoyer ses propres images, il suffit de saisir le nombre d'images qu'il souhaite faire traité. Ces images seront choisies aléatoirement de notre base de test et envoyées au service de traitement de données.

6 Gestion des ressources et cycle de vie :

La gestion des instances EC2 nécessaires pour le fonctionnement du cluster se fait par Terraform. Avec "Terrafrom apply", on lance les nœuds "workers" et le nœud "Master",lorsque une instance est lancée par Terraform, elle entre dans l'état "pending". Une fois l'instance prête, elle entre dans l'état "running". Nous pouvons à ce niveau se connecter au nœud "Master" et lancer l'application. Lorsque nous n'avons plus besoin des instances, on peut les mettre hors service via Terraform. Les instances passent donc à l'état "terminated"

Tous les pods nécessaires pour notre service sont lancés grâce au script 'run_app.sh'. Ce script est appelé automatiquement par Terraform sur le nœud Master une fois qu'il démarre. En utilisant les auto-scalers et les spécifications des tolérances de Kubernetes, les pods sont automatiquement maintenus et optimisés pour assurer une haute disponibilité et un coût minimal. Pour arrêter le service, il suffit d'appeler le script 'destroy_app.sh' sur le nœud Master, ce qui permet de supprimer toutes les composantes Kubernetes créées avec 'run_app.sh'.

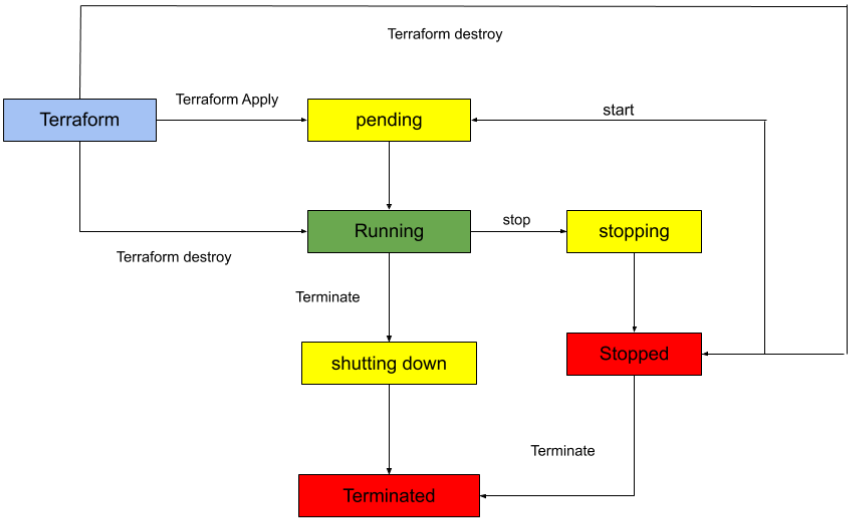


FIGURE 1 – Cycle de vie d'une instance EC2

7 Problèmes techniques et solutions

Après avoir développé une pile logicielle fonctionnelle, on s'est intéressé à l'aspect de scalabilité de Spark. Pour cela, on voulait utiliser l'allocation dynamique de Spark. Ceci peut-être réalisé en mettant 'spark.dynamicAllocation.enabled' à true.Cette solution est suffisante pour Spark 3. Dans notre cas, on utilise Spark 2.4.6. Cette version nécessite un 'External Shuffle Service' pour assurer la scalabilité des exécuteurs Spark. Ce problème n'est pas abordable à cause du manque de documentation. Malheureusement Spark Streaming depuis un Kafka Topic n'est toujours pas supporté en Python pour Spark 3, et donc on ne peut pas passer à utiliser Spark 3. Nous avons discuté des différentes possibilités avec l'équipe, et nous avons décidé d'ajouter des éléments supplémentaires à notre pile logicielle et de passer à un projet hybride à la place. Ainsi, nous pourrons apprendre une large sélection des technologies de systèmes distribués, ce qui sera utile pour nos stages/carrières.

Pour assurer l'auto scaling des nœuds workers au niveau d'aws, nous créer un groupe d'auto scaling avec les politiques suivantes :

— La création d'un nouveau nœud lorsque les ressources d'un nœud dépasse un seuil que nous avons fixé à 80

— La suppression d'un nœud lorsque ses ressources sont inférieures à un seuil qui a été fixé à 10

L'auto-scale fonctionne correctement, mais malheureusement nous avons un problème au niveau de "kubeadm" pour tous les nœuds créer par ce groupe et donc sont incapable d'être lier au cluster.

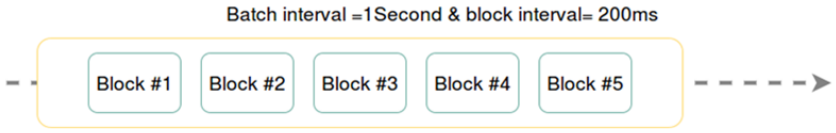
Suite au passage à un projet hybride, on a décidé d'intégrer MongoDB dans notre pile logicielle. Malheureusement, notre effort n'a pas été fructueux. L'intégration utilisait le concept de side-car, on n'a pas pu comprendre comment établir la communication entre les deux conteneurs dans le pod de MongoDb. Aussi on n'a pas pu établir la connexion à notre base de données en dehors du pod. Le deadline de projet s'approchait et on n'avait pas suffisamment de temps pour débugger et finir l'intégration. On a donc décidé de nous concentrer plutôt sur l'évaluation de la tolérance aux pannes de notre infrastructure.

Nous avons aussi voulu ajouter Kibana pour le monitoring de l'infrastructure. La solution nécessite de déployer Elasticsearch. Nous avons des problèmes de communication entre Kibana et Elasticsearch que nous n'avons pas réussi à résoudre dans les temps impartis.

Nous avons rencontré un autre problème avec l'installation des auto-scalers. Kubernetes n'arrive pas à retrouver les métriques liées aux différents pods. Ceci a été résolu en fin de compte avec un serveur de métriques qui communique les différentes mesures aux auto-scalers.

8 Benchmarking

Pour mesurer les capacités du système, on a choisi de mesurer le temps de traitement moyen d'une image dans un batch de 100 images choisies aléatoirement, Dans un premier temps, on a envoyé 100 images à Spark l'une après l'autre. Par conséquent, toutes les images ont été envoyées au même exécuteur Spark même si on disposait de 5 exécuteurs. Ceci a révélé une limitation de Spark Streaming, en fait la répartition des tâches sur les exécuteurs est déterminée par 2 paramètres, un intervalle de Batch, qui est la durée du micro-batch, et un intervalle de block. Toutes les images qui seront reçues pendant le même intervalle de block seront regroupées dans le même RDD, et traitées par la suite par le même exécuteur. Ce qui explique pourquoi toutes les 100 images envoyées n'ont pas été distribuées.



Dans un deuxième Benchmark, nous avons envoyé une image au service toutes les 200ms. C'est-à-dire nous avons assuré que chaque image envoyée sera contenue dans un RDD différent. Effectivement, nous avons réussi à distribuer le traitement de données, mais les gains en performances étaient non satisfaisants. On s'attendait à des gains en performances presque parfaits vu que l'application est facilement parallélisable.

The number of Spark executors	The average response time for processing one image	Gain in performance
2	6.782 seconds	-
3	4.797 seconds	39%
4	3.646 seconds	46%
5	3.484 seconds	49%

Avec 5 exécuteurs, nous avons à peine pu mesurer une amélioration de 49% comparée à 2 exécuteurs. Dans ce deuxième Benchmark, nous envoyons une image chaque 200ms, Spark assigne un exécuteur pour chaque image une fois reçue, c'est-à-dire au bout de 20 secondes, toutes les 100 images sont distribuées sur tous les exécuteurs disponibles. On finit généralement par une distribution uniforme des images, par exemple avec 4 exécuteurs, chacun traite 25 images. Le problème c'est que quelques exécuteurs finissent le traitement en 5 minutes, d'autres reçoivent un batch qui contient beaucoup plus de texte et prennent 15 mins pour finir l'exécution. Ceci entraîne une grande perte de performance. Ceci ne sera pas un problème avec une application d'un très haut flux de données avec un petit temps de traitement. Mais vu qu'on ne peut pas configurer le temps d'exécution moyen dans Spark Streaming, il serait mieux d'utiliser une solution de distribution personnalisé autre que Spark Streaming.

9 Évaluation de la tolérance aux pannes

Les différents scénarios testés :

9.1 Kubernetes - Disponibilité - Blackhole un nœud Kubernetes

9.1.1 Description

Blackhole est une technique que nous pouvons utiliser pour rendre les nœuds et les pods indisponibles . C'est une action moins destructrice que l'arrêt. Nous avons posé les questions suivantes :

— Mon cluster Kubernetes peut-il gérer correctement un nœud devenu indisponible ?

9.1.2 Hypothèses

Il s'agit d'un scénario de disponibilité pour Kubernetes. Ce scénario rendra un nœud de notre cluster Kubernetes indisponible. Nous prévoyons que l'application pourra toujours servir le trafic utilisateur et fonctionner comme prévu.

9.1.3 Résultats

La majorité des pods ont pu récupérer vu qu'on utilise des déploiements de Kubernetes, sauf pour Spark. Mais les pods prenaient très longtemps pour récupérer :

— Quand le worker 1 qui contient le pod exécuteur de Spark a été arrêté. l'application arrête de traiter les images pour quelques minutes et ensuite il reprend son fonctionnement

— Quand le worker 2 qui contient le driver de Spark, application de démonstration et un pod de ZooKeeper a été arrêté :

    - Le pod de Zookeeper a pu récupérer à terme.
    - Le driver de Spark n'a pas pu récupérer.
    - Vu qu'on a un seul pod de l'application de démonstration, elle a arrêté de fonctionner pour quelques minutes puis elle a pu récupérer. Mais le traitement d'images ne s'effectuait plus à cause de l'absence de Spark
- Quand le worker 3 qui contient les pods kafka et CoreDNS a été arrêté :
    - L'application a arrêté le traitement des images vu que les deux pods de Kafka n'étaient plus fonctionnels.
    - Les deux pods ont pu récupérer sur un autre nœud.
    - Spark a recommencé le traitement de quelques images déjà reçues vu que Kafka ne garantit que la livraison 'au moins une fois'.
- Quand le worker 4 qui contient le deuxième pod de Zookeeper et CoreDNS a été arrêté, Le pod de Zookeeper a pu récupérer à terme.

9.1.4 Les mesures suivies pour résoudre le problème

- Pour ajuster le temps nécessaire des pods pour récupérer, nous avons changé les paramètres par défaut durant l'initialisation du cluster (not-ready tolérance et unereachable tolérance à 30 secondes) les pods sont attribuées d'une façon plus rapide à d'autre nœuds.
- De même, on a ajouté une entrée 'tolérances' dans le Yaml file de chaque déploiement pour décrire le comportement des pods en cas de panne et permettre une récupération des pods plus rapide.
- Pour éviter d'avoir un downtime pour notre application de démonstration, on peut tout simplement ajouter une deuxième réplication du pod.
- L'application est capable de récupérer son fonctionnement normal après l'arrêt de n'importe quel nœud , sauf dans le cas d'un nœud qui contient le pod data processing driver.

9.1.5 Limitation

La faible tolérance de Spark Driver revient au fait qu'il est déployé comme un pod sur Kubernetes. Changer le type de déploiement à un Kubernetes Job ou un Kubernetes Deployment ou même créer un deuxième pod Driver pourra améliorer sa tolérance à cette panne. Mais on n'a pas pu trouver une méthode pour appliquer ces solutions.

9.2 Kubernetes - Disponibilité - Blackhole un pod Kubernetes

9.2.1 Description

Arrêter plusieurs pods et regardez si le système pourrait récupérer rapidement.

9.2.2 Hypothèses

Il s'agit d'un scénario de disponibilité pour Kubernetes. Ce scénario rendra certains pods du cluster Kubernetes indisponibles. Nous prévoyons que l'application pourra toujours servir le trafic utilisateur et fonctionner comme prévu.

9.2.3 Résultats

— Après l'arrêt du Spark Driver, ses exécuteurs se sont arrêtés automatiquement et le traitement des données s'est arrêté. Le système n'a pas pu récupérer et créer une nouvelle instance.
— Après avoir supprimé le pod de l'application de démonstration, le système est capable de créer un autre pod similaire et continuer son fonctionnement de façon normale.
— La suppression de l'un des pods kafka, zookeeper n'influence pas le système, le système est capable de créer immédiatement un remplaçant.

point de défaillance : La suppression du pod Spark driver.

9.2.4 Les mesures suivies pour résoudre le problème

— Spark-submit crée un pod pour contenir le driver de Spark. L'option RestartPolicy de ce Pod est par défaut mise à Never. Cette valeur n'est pas modifiable en utilisant spark-submit. Par contre, Sparkapplication de Helm permet l'accès à ce paramètre. On a donc changé notre application pour utiliser Sparkapplication de Helm ce qui nous a permis d'améliorer la tolérance à ce type de faille.
— En conséquence, tous nos pods sont maintenant tolérants à cette faille.

9.3 Kubernetes - Disponibilité - CPU attack d'un pod Kubernetes

9.3.1 Description

Stresser l'utilisation du processeur d'un pod pour voir si le service continuera de fonctionner.

9.3.2 Hypothèses

Il s'agit d'un scénario de disponibilité pour Kubernetes. Ce scénario rendra certains pods du cluster Kubernetes indisponibles à cause de l'utilisation de toutes les ressources processeur du pod.

9.3.3 Résultats

— Après avoir appliqué cette attaque aux différents déploiements de notre service, le cluster peut continuer de fonctionner grâce aux auto-scalers qui créent un nouveau pod qui ne souffre pas, lui, de cette attaque
— Spark arrête de fonctionner.

point de défaillance : L'attaque du pod Spark driver.

9.3.4 Les mesures suivies pour résoudre le problème

— Les auto-scalers nous ont permis de résoudre ce problème.
— Même en activant l'auto-scaling de Spark en passant à Spark 3, le driver ne supporte pas la réplication. Spark reste donc vulnérable à cette attaque.

9.4 Conclusion Tolérance aux pannes :

En conclusion, toutes les composantes de notre pile logiciel peuvent récupérer de ces scénarios de test sauf pour Spark. Il paraît que Spark Driver constitue un **single point of failure** de notre infrastructure. On peut penser à pallier ce problème en créant un pod qui envoie des Health Check régulier au pod Spark Driver. Si jamais le pod ne répond pas après un timeout déterminé, une nouvelle application Spark est lancée en tuant l'ancienne.

10 Coût de l'infrastructure

Pendant la phase du développement de notre infrastructure, nous avons dépensé au total 120 dollars partagés entre les membres de l'équipe.
Dans le cas normal, notre système utilise 6 instances t2.medium situées dans la région US - East (N.Virgenia) . En utilisant "Amazon EC2 estimate" le coût total de notre infrastructure est de 95.80 dollars par mois.

Amazon EC2 estimate

Amazon EC2 Instance Savings Plans instances (monthly) :125.71 USD
Amazon Elastic Block Storage (EBS) pricing (monthly) : 18.00 USD
Total monthly cost : 143.71 USD