CentraleSupélec

Metz

Ingénierie des applications logicielles

Compte-rendu

Tweetoscope

BARRÉE Guillaume
CARNEIRO BARBOSA ROCHA João Felipe
COLOMBO Daniel

Table des matières

Enseignants : Virginie GALTIER & Michel IANOTTO

3 Décembre 2021

1 Architecture de la solution

Nous avons développé notre solution en nous basant sur l'architecture qui nous avait été proposé. Elle est donc constituée de différents micro-services qui communiquent entre eux via Kafka.

Vous trouverez dans le dossier courant un README présentant le projet ainsi que les liens utiles vers les différentes documentations du projet.

Git : Chaque étape du développement a directement été gérée avec l'outil *Issues* de Gitlab pour suivre au mieux l'état d'avancement du projet. La structure des messages des *commits* avec les balises permet de comprendre rapidement la modification apportée et aide à la gestion du projet. A chaque version intermédiaire de notre développement, nous avons associé un tag Gitlab qui permet de facilement visualiser les étapes importantes du projet.

Solution finale :

tweet-generator : Il était fourni au début du projet. Néanmoins nous avons noté quelques problèmes de fonctionnement que nous avons corrigé pour garantir la croissance des temps au sein d'une même source lorsque plusieurs partitions étaient utilisées. Nous avons ajouté une option pour compiler le code avec cmake. Cela permet de vérifier les dépendances et faciliter la procédure de build.

tweet-collector : Développé en C++. Le code est disponible dans *./src/tweet_collector*. Son développement s'est fait en plusieurs étapes clés. Des tests ont été réalisés pour valider l'implémentation de chaque bloc qui le compose (Consumer, Producer, Cascade, Processor). La compilation du code et des tests est géré par cmake pour vérifier les dépendances et faciliter la procédure de build.

hawkes-estimator : Développé en Python. Le code est disponible dans *./src/hawkes_estimator*. Nous nous sommes basés sur le TP de ModStat pour son implémentation.

predictor : Développé en Python. Le code est disponible dans *./src/predictor*. Il a été implémenté en orienté objet afin de faciliter la gestion des différentes fenêtres de temps et des différentes cascades. La façon de gérer les messages a évolué à la fin du projet afin de faciliter son passage à l'échelle et améliorer la gestion de la mémoire.

learner : Développé en Python. Le code est disponible dans *./src/learner*. Il a été implémenté en orienté objet afin de faciliter la gestion des différentes fenêtres de temps. Un système de buffer a été mis en place pour éviter de trop souvent mettre à jour les modèles. Nous attendons que 5 nouveaux messages arrivent avant de mettre à jour la *RandomForest*.

logger : Il était fourni au début du projet. Il nous permet d'afficher les logs envoyés par les différents micro-services cités ci-dessus. Chacun des micro-services envoie donc des messages sur le topic *logs* pour informer l'utilisateur lorsqu'un tweet est traité, qu'un message est envoyé ...

dashboard : Développé en Python. Le code est disponible dans *./src/dashboard*. Il utilise les différentes API Python proposées par Loki et Prometheus. L'affichage est géré par Grafana. Tous les fichiers de configuration pour Loki, Prometheus et Granafa sont disponibles dans *./src/dashboard/config*.

2 Pipeline et déploiement

2.1 Docker

Chaque micro-service (*tweet_generator*, *tweet_collector*, ...) est associé à une image Docker. Chaque image docker se structure de la façon suivante. L'objectif était de minimiser la taille de chaque image.

```
1 FROM image (Alpine ou python3.9-slim)
2 RUN install dependancies
3 RUN build (si necessaire)
4 CMD launch application
```

En utilisant Alpine et python3.9-slim à la place de gcc et python, nous sommes parvenus à réduire drastiquement la taille de nos images (le *tweet_collector* est passé de *1.73*GB avec gcc à *554*MB avec Alpine). Nous avons également crée nos propres images Loki, Prometheus et Grafana afin d'y appliquer toutes les configurations nécessaires à leur fonctionnement. Les images docker Kafka, Zookeeper sont quant à elles téléchargées depuis le DockerHub.

2.2 CICD

Une fois tous ces *Dockerfile* crées, nous avons utilisé la fonctionnalité d'intégration et de déploiement continu de Gitlab. Ainsi, lorsqu'un fichier impactant un micro-service est modifié, l'image correspondante est directement construite et partagée sur le Container Registry de Gitlab.

2.3 Docker Compose

La première façon de déployer notre solution est d'utiliser *docker-compose*. Une documentation est disponible dans *./docker*. Trois fichiers *.yml* sont présents. Le premier sert à déployer Zookerper et Kafka, le second permet de lancer les différents micro-services et le dernier permet de lancer toutes les images liées au dashboard.

2.4 Kubernetes

Le déploiement sur *Kubernetes* est également possible (soit en utilisant *Minikube*, soit en utilisant le cluster de l'école pour utiliser plusieurs noeuds). Le répertoire Gitlab étant privé, la gestion de token d'accès et la mise en place de secret dans Kubernetes ont été nécessaires pour donner l'accès au Container Registry où sont stockées nos images. La documentation liée au déploiement sur *Kubernetes* est disponible dans *./kubernetes*

3 Passage à l'échelle et tolérance aux pannes

Nous avons mis en place deux stratégies pour le passage à l'échelle.

Code source : Les messages publiés dans les topics Kafka ont deux structures distinctes : soit la fenêtre d'observation sert de clé au message, soit il n'y a pas de clé. Tous les micro-services ont été développés en prenant en compte cette spécificité pour garantir un passage à l'échelle facile ainsi qu'une meilleure gestion de la mémoire.
Par exemple, la conception du *hawkes_estimator* permet de passer à l'échelle directement en augmentant le nombre de répliques jusqu'au nombre de partitions disponibles. Pour le *prédictor* et le *learner*, des groupId ont été spécifiés dans les Consumers. Ainsi, si nous répliquons les *predictor* et *learner*, les Consumers vont se partager les partitions disponibles. Les objets de notre code ayant été conçus de manière à ne dépendre que de la fenêtre de temps (donc à une partition), chaque réplique du *predictor/learner* va gérer une fenêtre de temps spécifique lors d'un passage à l'échelle.

Minikube & Intercell : L'utilisation de Minikube et Intercell est disponible pour assurer le passage à l'échelle d'un point de vue architecture logicielle.
Chaque micro-service est déployé sur un pod kubernetes qui peut être répliqué pour avoir autant de Consumers du même groudId qu'il y a de partitions.
Si une panne intervient et qu'un pod tombe, un nouveau pod se relance automatiquement (sur Intercell) pour relancer le micro-service en question assurant ainsi une robustesse de l'application.

4 Visualisation des résultats avec Gafana

Le fonctionnement ainsi que les explications des différents graphiques de notre dashboard sont disponibles dans *./src/dashboard*.