CRV

Projet Observabilité

L'objectif de ce projet est d'utiliser l'infrastructure créé pour le projet précédent (kubernetes, prometheus, etc).

Pour simuler différents scénarios et détecter les limites de perfomances du projet.

Rappel de l'infrastructure

Redis

Une base de données redis implémentant un pattern main/replicas, une base redis principale accepte toutes les opérations et des replicas recopient les données de cette base pour accepter plus de lecture en parallèle.

Lorsqu'une écriture survient sur la base principale propage la modifications aux réplicas. Les réplicas n'acceptent pas d'écriture.

Dans un premier temps seul les replicas pourront monter à l'échelle. Mais il est possible d'implémenter d'autres stratégies plus élaborées si le travail initial est terminé.

https://hub.docker.com/_/redis

Nodejs

Un serveur nodejs stateless (vu en TME) qui appel la base redis et ses replicas.

Grâce à la caractéristique stateless de ce serveur (ne possède pas d'état persistant) il est possible de le dupliquer sans modifier le comportement de l'application.

https://github.com/arthurescriou/redis-node

https://hub.docker.com/r/arthurescriou/node-redis

React

Le frontend ne s'executant que très peu sur les serveur backend (transfert de fichier statique). On ne s'intéressera pas à ses performance.

Prometheus/grafana

Il faut aussi ajouter un système pour récupérer les logs: prometheus/grafana. Afin d'enregistrer les différentes variations de l'application (nombre d'appel, delai des requêtes, mémoire, etc).

Au minimum le prometheus devra récupérer des données provenant du serveur **Nodejs** et de la base de données **Redis** principale.

Il est également possible pour étoffer votre rapport de le configurer pour récupérer des données du cluster **kubernetes**.

L'affichage pourra se faire depuis prometheus ou en ajoutant un grafana.

Scénarios intéressants

Ressources allouées

Pour obtenir des résultats réalistes et intéressants on veut limiter les ressources de conteneurs. En effet dans le cas de déploiement sur des serveurs dans le cloud ou chez un fournisseurs de machine virtuelle les machines ont rarement autant de ressources que nos ordinateurs.

Par exemple les VM d'aws EC2 de base ont 2 giga de ram et 1 seul CPU. On veut donc limiter de la même manière les ressources données à notre serveur et à sa base à l'aide des conteneur kubernetes.

Charge

L'autre chose que l'on veut simuler c'est l'utilisation de l'application, et donc simuler des utilisateurs qui se connectent et utilise le serveur. Pour ça le plus simple et de créer un script (bash, python ou js) qui lance des requêtes HTTP sur le endpoint du serveur.

Pensez bien à varier les types d'utilisations : Le comportement du serveur et de sa base peuvent varier si les utilisateurs font seulement de la lecture ou si il y a également des écritures en base de données. Pour ça veillez bien à utiliser les différents endpoints du serveur (create, request, update, delete).

Monté à l'échelle

Lorsque que le serveur est beaucoup sollicité il est possible de dupliquer l'instance de pod du serveur. Il est possible de le faire manuellement ou avec une configuration qui l'automatise avec les deploiement kubernetes. (Il est très interessant d'observer quelles conditions d'utilisations déclenchent ce genre de monté à l'échelle automatique).

Rendu attendu

Il est attendu un rapport mettant en forme les résultats de vos observations de différents scénarios que vous trouvez pertinents. Un soin particulier doit être porté à la présentation des résultats (graphiques scientifiques, explicitation du contexte du scénario).

De plus il est important d'y ajouter votre analyse et déductions par rapport au comportement observés.

Le rendu attendu concerne plus les résultats obtenus par différents scénarios plutot que la configuration du code.

Cependant il est utile aussi de joindre ces configurations de code pour expliciter les scénarios lancés.

Exemple de graphiques qui pourrait être pertinent :

- charge limite de l'application : (pour des ressources données, trouver le nombre et type d'appel par secondes maximum)
- charge optimal : (comportement jugé comme optimal : temps de réponses minimum, pas d'erreur du système, avec un maximum d'utilisateurs par secondes)
- montée à l'échelle : (comportement lors de la création d'un nouveau conteneur)

Vous pouvez ajouter tout les autres scénarios que vous trouvez pertinent.

Scripts fourni

Pour vous aider à réaliser ces situations vous pouvez utiliser les scripts fourni dans le dossier `loadTest`

Les scripts fournissent plusieurs scénarios :

- "server" ping le serveur sans qu'il intéragisse avec la base de données

  arguments :

  - nombre d'appels total (défaut: 10000)
  - nombre d'appels simultanés (défaut: 100)
- "writeRead" fait des appels sur le serveur pour qu'il écrive et lise dans la base de données

  arguments :

  - nombre d'appels total (défaut: 10000)
  - nombre d'appels simultanés (défaut: 100)
- "pending" ouvre des connections avec le serveurs (attention il faut la version `arthurescriou/node-redis:1.0.6` pour que la route existe (si vous utilisez votre propre image vous avez besoin de mettre à jour le code)).

  arguments :

  - nombre d'appels simultanés (défaut: 200)
  - temps de réponse de la requête (défaut: 10000 (ms))

Exemples :

```
node fetchData.js server 10000 100
node fetchData.js writeRead 10000 100
node fetchData.js pending 200 10000
```