Laboratoire #2

Groupe: 2 Equipe: 04

Membres de l'équipe: Jonathan Rodriguez Tames

Évaluation de la participation

> L'évaluation suivante est faite afin d'encourager des discussions au sein de l'équipe. Une discussion saine du travail de chacun est utile afin d'améliorer le climat de travail. Les membres de l'équipe ont le droit de retirer le nom d'un ou une collègue du rapport.

nom de l'étudiant	Facteur multiplicatif
Jean Travaillant	1
Joe Paresseux	0.75
Jules Procrastinateu	0.5
Jeanne Parasite	0.25
Jay Oublié	0

Introduction

> TODO: insérer votre introduction

Vues architecturales

- Au travers des différentes vues architecturales, montrez comment la redondance est présente dans vos microservices après l'implémentation du laboratoire 2. La présence des vues primaires et des catalogues d'éléments est nécessaire. Assurez-vous de bien présenter la tactique de redondance dans vos vues.

Vues architecturales de type module - redondance

Vues architecturales de type composant et connecteur - redondance

Vues architecturales de type allocation - redondance

> Note : Une légende est nécessaire dans les vues primaires pour assurer une compréhension claire des éléments représentés.

Alternatives envisagées

Expérience avec les Redondances pour TripComparator

Dans cette expérience, nous avions le choix d'implémenter deux types de redondances différentes pour **TripComparator** :

- Une **redondance passive** où un Leader est assigné pour envoyer les requêtes et les messages.
- Une **redondance active** où les deux instances envoient simultanément des requêtes et des messages.

Chaque implémentation avait ses particularités, ses avantages et ses inconvénients, que nous présenterons ici. Par la suite, nous expliquerons pourquoi nous avons choisi une approche plutôt qu'une autre.

Redondance Passive

La redondance passive pour le **TripComparator** présente certaines particularités :

- **Un seul Leader actif** est assigné pour envoyer les **requêtes** aux services et construire les **messages** à transmettre à RabbitMQ.
- L'autre instance reste en veille (*stand-by*) jusqu'à ce que le Leader cesse de fonctionner. À ce moment-là, l'instance en veille devient le nouveau Leader.

Fonctionnement

Pour permettre ce fonctionnement, nous avons implémenté un **service Manager**, dont le rôle principal est de :

1. Surveiller en permanence l'état des deux instances de TripComparator.
2. Garantir qu'il n'y a **qu'un seul Leader actif** à tout moment, pour éviter les requêtes et messages dupliqués dans RabbitMQ.

De plus, pour que le nouveau Leader puisse reprendre l'exécution après la défaillance de l'ancien, il est essentiel qu'il connaisse l'état du système. Nous avons utilisé une **base de données Redis** pour :

- Stocker les informations initiales de l'expérience.
- Suivre l'état du système en fonction des tâches accomplies.

Ainsi, le nouveau Leader peut reprendre précisément là où s'était arrêté son prédécesseur, **éviter les répétitions de tâches**, et prévenir les erreurs dans les requêtes ou les messages envoyés.

Redondance Active

L'approche active, quant à elle, présente des particularités distinctes :

- **Les deux instances** de TripComparator envoient simultanément des **requêtes** aux services et des **messages** à RabbitMQ.
- Un filtre est nécessaire pour sélectionner **un seul message** à afficher sur le tableau de bord.

Complexités

Cependant, cette approche pose certains défis :

- Chaque instance effectue des requêtes en continu auprès des services, ce qui peut entraîner une surcharge inutile.

- Le filtrage des messages pour garantir qu'un seul message est affiché sur le tableau de bord est complexe.
- Bien que nous ayons envisagé d'implémenter un filtre pour nettoyer les messages, cette solution a été abandonnée.

La principale raison était notre **compréhension insuffisante** du processus d'acheminement des messages vers RabbitMQ et des mécanismes nécessaires pour concevoir un filtre efficace.

Décision Finale

Nous avons opté pour la **redondance passive**, car :

1. Cette approche avait déjà été implémentée pour le service STM, ce qui nous a permis de réutiliser les **SideCars** existants pour communiquer avec le Manager concernant l'état du service STM.
2. Cela nous a fait **gagner un temps précieux**.
3. Nous avons pu capitaliser sur nos connaissances de Redis pour stocker les données essentielles au bon déroulement de l'expérience.

Conclusion

Cette solution s'est avérée :

- Plus rapide à mettre en œuvre.
- Mieux adaptée à notre contexte technique.
- Évitant les complexités liées à la redondance active.

Diagrammes de séquence pour expliquer le fonctionnement des tactiques de redondance

- Vous devez fournir les diagrammes de séquence démontrant le fonctionnement de l'architecture avant le début du laboratoire 2.
- Vous devez fournir les diagrammes de séquence démontrant le fonctionnement après la réalisation du laboratoire 2.
- Représentez les différents moments de vie du système, si applicables:
    - L'état au démarrage des conteneurs,
    - La mécanique de détection d'un problème,
    - La mécanique de rétablissement du service, de récupération ou de reconfiguration.

Questions

- **Question 1** Discuter de l'interopérabilité des microservices, tout en fournissant des exemples d'une ou plusieurs tactiques utilisées. Si vous pensez qu'aucune tactique n'est utilisée dans le code, nommez et décrivez une ou plusieurs tactiques qui pourraient être utilisées.

- **Question 2** Qu'est-ce que l'injection de dépendance? Fournissez des exemples de l'utilisation de l'injection de dépendance dans un des microservices du projet. Comment l'injection de dépendance améliore-t-elle la testabilité du système?

- **Question 3** : Dans le laboratoire, nous utilisons une technologie semblable à Kubernetes. Qu'est-ce que Kubernetes et en quoi cette technologie est-elle pertinente à la redondance ? Quels sont les éléments et les concepts de Kubernetes qui sont présents dans le projet du laboratoire ?

Kubernetes : Une plateforme pour orchestrer les conteneurs

Kubernetes est une plateforme open-source qui automatise le déploiement, la gestion et l'évolutivité des applications conteneurisées. Elle permet d'orchestrer les conteneurs dans un cluster composé de plusieurs machines (physiques ou virtuelles), garantissant la fiabilité, la scalabilité et la performance des applications.

Concepts clés de Kubernetes

1. **Pod**

- L'unité de base dans Kubernetes est le **Pod**, qui regroupe un ou plusieurs conteneurs partageant des ressources telles que le réseau et le stockage.

2. **ReplicaSets**

- Les **ReplicaSets** s'assurent qu'un nombre défini de copies (*réplicas*) d'un Pod est toujours actif.
- Si un Pod échoue ou devient indisponible, Kubernetes crée automatiquement un nouveau pour le remplacer.
- Cela garantit une disponibilité continue des instances fonctionnelles.

3. **Deployments**

- Les **Deployments** permettent de gérer les mises à jour et le déploiement d'applications de manière déclarative, facilitant les transitions entre différentes versions.

Fonctionnalités pertinentes à la redondance

1. **Gestion des ReplicaSets**

- Kubernetes maintient automatiquement le nombre requis de réplicas pour un Pod.
- En cas de panne d'un Pod, un nouveau est instantanément créé pour le remplacer.
- Cette approche garantit que les demandes des utilisateurs sont toujours satisfaites.

2. **Équilibrage de charge**

- Kubernetes distribue le trafic réseau entre les Pods disponibles via un service d'équilibrage de charge intégré.
- Cela répartit efficacement la charge et évite de surcharger un seul Pod.
- Si un Pod devient indisponible, Kubernetes redirige automatiquement le trafic vers les Pods restants.

3. **Surveillance et auto-récupération**

- Kubernetes surveille en permanence l'état des Pods et des nœuds.
- Si un Pod, un conteneur ou un nœud entier tombe en panne, Kubernetes redémarre ou recrée automatiquement l'instance concernée sur un autre nœud.
- Cela garantit une **disponibilité continue** des services.

Similitudes entre Kubernetes et NodeController

Le **NodeController** partage plusieurs concepts et fonctionnalités similaires à Kubernetes, bien qu'il ne l'utilise pas directement. Voici une analyse des similitudes clés :

Concepts Partagés

1. **Terminologie des Pods**

- **Kubernetes** : Les Pods sont les unités de base qui regroupent un ou plusieurs conteneurs partageant des ressources comme le réseau et le stockage.
- **NodeController** : Les **pod types** et **pod instances** imitent cette idée en regroupant plusieurs services (conteneurs) sous une même entité. Cela permet de gérer les dépendances, le scaling et le statut des services comme un tout.

2. **Services**

- **Kubernetes** : Les services permettent de connecter des Pods entre eux et d'exposer des applications à l'extérieur du cluster. Ils sont souvent associés à des configurations DNS.
- **NodeController** : Les services sont définis comme des **service types**. Ils contiennent des informations DNS et peuvent être regroupés ou reliés à d'autres services à travers des **Pod_Links**. Le NodeController utilise ces services pour gérer le routage et la découverte.

Gestion des Conteneurs

1. **Redondance et Scalabilité**

- **Kubernetes** : Kubernetes gère la redondance avec des **ReplicaSets**, permettant de maintenir un certain nombre de Pods actifs. Il offre également un mécanisme de scaling horizontal (autoscaling).
- **NodeController** : Il permet de définir le nombre de réplicas d'un pod via des labels comme `Replicas`. Lorsqu'un pod échoue, une nouvelle instance est créée pour maintenir l'intégrité du système.

2. **Sidecars**

- **Kubernetes** : Kubernetes utilise souvent des sidecars pour ajouter des fonctionnalités complémentaires (ex. : monitoring, logs, etc.).
- **NodeController** : Les conteneurs secondaires d'un pod sont considérés comme des **sidecars**, avec des DNS privés par défaut. Cela simplifie l'ajout de fonctionnalités sans modifier directement le conteneur principal.

Routage et Découverte

1. **Règles DNS**

- **Kubernetes** : Les configurations DNS sont utilisées pour limiter ou permettre l'accès à certains services dans un cluster.
- **NodeController** : Les labels comme `DNS: Private`, `DNS: Public`, et `DNS: Partial` définissent les règles de routage et d'accessibilité entre les services.

2. **Load Balancing**

- **Kubernetes** : Kubernetes inclut un mécanisme d'équilibrage de charge au niveau réseau et application, redirigeant les requêtes vers les Pods disponibles.
- **NodeController** : Offre un équilibrage de charge au niveau L4 (Transport Layer) pour les connexions TCP (comme RabbitMQ ou PostgreSQL) et un service de découverte au niveau L7 (Application Layer), fournissant des adresses aux clients sans agir comme un proxy complet.

Avantages et Inconvénients

Similitudes dans les Avantages

- **Multi-conteneurs (Pods)** :
  - **Kubernetes** : Simplifie la gestion de services interdépendants.
  - **NodeController** : Permet de regrouper dynamiquement des conteneurs pour des scénarios créatifs comme l'élection de leaders ou le partage de fonctionnalités via des sidecars.
- **Scalabilité** :
  - Les deux systèmes permettent de gérer dynamiquement les réplicas pour répondre à la charge.

Similitudes dans les Inconvénients

- **Complexité** :
  - Les deux introduisent une couche de complexité supplémentaire, nécessitant une configuration fine et une compréhension des interactions entre services.
- **Problèmes liés aux pods multi-conteneurs** :
  - Si un conteneur échoue, tous les autres conteneurs du pod sont affectés, ce qui peut entraîner un gaspillage de ressources.

Autres Points Clés

1. **Volumes Partagés**

- Similaire à Kubernetes, le NodeController permet de partager des volumes entre les réplicas avec le label `SHARE_VOLUMES`.

2. **Alias pour Services**

- NodeController propose une fonctionnalité semblable aux services de Kubernetes en utilisant des alias pour regrouper plusieurs instances sous un seul nom logique.

3. **Modes de Routage**

- NodeController permet des routages avancés basés sur des configurations DNS et des stratégies de service mesh, rappelant les fonctionnalités de Kubernetes Ingress et ServiceMesh.

Conclusion

Bien que le **NodeController** ne soit pas basé sur Kubernetes, il s'en inspire fortement dans la gestion des conteneurs, des pods, du routage, et de la découverte des services. Cette approche permet de bénéficier des concepts avancés de Kubernetes tout en adaptant la complexité à des environnements plus spécifiques ou contrôlés.

Conclusion

> TODO: insérer votre conclusion

- N'oubliez pas d'effacer les TODO.
- Générer une version PDF de ce document pour votre remise finale.
- Assurez-vous du bon format de votre rapport PDF.
- Créer un tag git avec la commande "git tag laboratoire-2".

\newpage

Annexes