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

Dans le cadre de ce laboratoire, notre objectif est d'améliorer la disponibilité de nos services. Pour ce faire, nous devons respecter certaines contraintes afin de garantir le succès de notre expérience. Pendant 5 minutes, un outil similaire à Chaos Monkey détruira nos services à une fréquence de 6 fois par minute. Notre but est de créer une infrastructure capable de se rétablir et de continuer à fournir le service de suivi des bus pour deux coordonnées spécifiques.

Critères de réussite

Pour considérer notre expérience comme réussie, nous devons respecter les critères suivants :

- **Actualisation continue** : Les messages affichant les informations de suivi ne doivent pas cesser de s'actualiser et doivent indiquer le bus correspondant aux coordonnées fournies.
- **Temps de réponse** : Notre temps de réponse (ms) doit être inférieur à 250 tout au long de l'expérience.
- **Stabilité** : Le temps nécessaire au système pour se rétablir doit également être inférieur à 250 ms.
- **Précision des trajets** : La durée correcte du trajet en voiture doit s'afficher correctement.
- **Réduction des erreurs** : Nous devons minimiser le nombre d'erreurs, telles que les messages répétés ou en retard.

Plan du rapport

Dans ce rapport, nous présenterons les stratégies mises en place pour répondre à ces exigences. Pour cela, nous aborderons les points suivants :

- Vues architecturales de type module pour le systeme de redondance
- Vues architecturales de type composant et connecteur pour le systeme de redondance
- Vues architecturales de type allocation pour le systeme de redondance
- Diagrammes de séquence avant et après l'implémentation de nos stratégies de redondance
- Alternatives envisagées
- Réponses à des questions concernant l'interopérabilité, les concepts de Kubernetes liés au projet et l'utilisation de l'injection de dépendance

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

Analyse des types de redondances pour TripComparator

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

- Une **redondance passive** pour l'assignation du Leader.
- Une **redondance active** pour l'envoi simultané des messages de chaque TripComparator.

Chaque implémentation avait ses **particularités**, ses **avantages** et ses **inconvénients**, que nous détaillerons ici. Enfin, nous expliquerons pourquoi nous avons choisi une approche plutôt qu'une autre.

Redondance Passive

La redondance passive repose sur un **modèle maître-esclave** dans lequel une seule instance de **TripComparator** est active à la fois, tandis que l'autre reste en attente (**stand-by**). Voici les particularités et la logique derrière cette approche :

Particularités

1. **Un seul Leader actif** :

- Une seule instance de **TripComparator** (le Leader) est responsable de la **construction des messages** et de leur envoi à **RabbitMQ**.
- L'autre instance reste en veille jusqu'à ce que le Leader devienne inactif.

2. **Prise de relais** :
   - Si le Leader cesse de fonctionner (panne ou arrêt), l'instance en veille devient automatiquement le nouveau Leader.
3. **Supervision par un Manager** :
   - Un service **Manager** surveille constamment l'état des deux instances de TripComparator. Son rôle est double :
     - Vérifier qu'un seul Leader est actif à un moment donné.
     - Éviter que plusieurs Leaders simultanés provoquent l'envoi de **messages dupliqués** dans RabbitMQ.
4. **Redis pour la continuité** :
   - Une base de données Redis est utilisée pour stocker :
     - Les informations initiales de l'expérience.
     - L'état courant des tâches exécutées.
   - Cela permet au nouveau Leader de **reprendre exactement là où le précédent s'était arrêté**, réduisant ainsi les **erreurs** et **redondances** dans les messages envoyés.

Avantages

- **Simplicité du flux de messages** :
  - Un seul Leader envoie des messages, ce qui élimine le besoin de filtrage au niveau de RabbitMQ.
- **Reprise sans perte de données** :
  - Grâce à Redis, le système peut reprendre l'exécution sans répéter les tâches déjà accomplies.
- **Utilisation efficace des ressources** :
  - Une seule instance est active, réduisant ainsi la consommation de ressources par rapport à une redondance active.

Inconvénients

- **Temps de basculement** :
  - Lorsque le Leader devient inactif, il y a un léger **temps de latence** avant que l'instance en veille prenne le relais.
- **Dépendance à Redis** :
  - Si Redis est indisponible, la continuité de l'exécution est compromise.

Redondance Active

La redondance active adopte une approche où **les deux instances de TripComparator fonctionnent simultanément** pour envoyer des messages à RabbitMQ.

Particularités

1. **Envoi simultané des messages** :
    - Les deux instances de TripComparator **envoient des messages en parallèle** à RabbitMQ.
2. **Nécessité d'un filtre** :
    - Étant donné que les deux instances produisent des messages similaires, un **filtre** est nécessaire pour choisir quel message sera affiché sur le tableau de bord.
3. **Requêtes STM continues** :
    - Chaque instance envoie des **requêtes en continu** aux services STM pour récupérer les informations nécessaires, augmentant la charge sur STM.
4. **Complexité accrue** :
    - La conception et l'implémentation d'un filtre efficace pour éliminer les doublons nécessitent une compréhension approfondie du **processus d'acheminement des messages** vers RabbitMQ.

Avantages

- **Tolérance accrue aux pannes** :
    - Même si une instance devient inactive, l'autre continue d'envoyer des messages, assurant ainsi une **disponibilité maximale**.
- **Réduction des temps de basculement** :
    - Contrairement à la redondance passive, il n'y a pas de délai pour basculer entre les instances puisque les deux fonctionnent simultanément.

Inconvénients

- **Complexité du filtrage** :
    - La conception d'un filtre pour éviter les doublons est techniquement exigeante.
- **Charge réseau et système** :
    - Les requêtes continues de chaque instance vers STM augmentent la **charge réseau** et peuvent impacter les performances globales.
- **Efforts d'implémentation plus élevés** :
    - La mise en œuvre de la redondance active est plus complexe et nécessite davantage de tests.

Décision Finale

Nous avons opté pour la **redondance passive** pour les raisons suivantes :

1. Réutilisation des composants existants

- 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.

2. Simplicité et rapidité d'implémentation

- La redondance passive est plus simple à mettre en œuvre.
- En utilisant Redis pour stocker les états du système, nous avons pu **accélérer le développement** et garantir une reprise correcte après une panne.

3. Charge réseau réduite

- Contrairement à la redondance active, la redondance passive limite le nombre de requêtes simultanées envoyées aux services STM.

4. Adaptation au contexte technique

- Dans notre contexte technique, où nous avons une compréhension limitée des mécanismes de filtrage et d'acheminement de message vers la RabbitMQ, la redondance passive était plus adaptée.

Synthèse

Bien que la redondance active offre des avantages en termes de disponibilité, sa complexité accrue et son impact sur les performances nous ont poussés à privilégier la redondance passive. Cette solution s'est avérée plus rapide à mettre en œuvre et plus adaptée à nos connaissances techniques.

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.

L'**interopérabilité des microservices** est essentielle pour garantir que les différents services au sein d'un système distribué peuvent communiquer efficacement, partager des données et collaborer pour accomplir des tâches. Cette interopérabilité repose sur des **tactiques** spécifiques qui permettent de découvrir, de gérer et de synchroniser les interactions entre services.

Dans ce cas, nous explorons l'interopérabilité entre le **SideCard** et le **LeaderController** dans le contexte du service STM.

Tactiques d'Interopérabilité utilisées

1. Découverte des Services

Définition

La découverte des services est une tactique d'interopérabilité qui consiste à localiser dynamiquement les services nécessaires pour établir une communication. Cette approche est essentielle dans les environnements où les adresses des services ne sont pas fixes ou prédéterminées.

Application dans le Système

Dans l'interaction entre **SideCard** et **LeaderController**, la découverte est réalisée grâce à des mécanismes comme :

- L'utilisation d'un **PodLeaderId** (identifiant dynamique du leader du pod) pour identifier le service principal dans un environnement multi-pods.
- La possibilité de récupérer dynamiquement l'adresse d'un leader via une logique de résolution des services.

Impact

- Réduit la dépendance aux configurations statiques.
- Renforce la flexibilité en permettant aux services de s'adapter dynamiquement à des changements d'infrastructure.

Exemple de mise en oeuvre dans le code

```
[HttpGet]
[ActionName(nameof(PromoteToLeader))]
[EnableRateLimiting("fixed")]
public async Task<ActionResult<string>> PromoteToLeader()
{
    var podLeaderID = await ServiceMeshInfoProvider.PodLeaderId;

    try
    {
        var res = await RestController.Get(
            new GetRoutingRequest()
            {
                TargetService = podLeaderID,
```

```
                Endpoint = "Leader/LeaderPromotion",
                Mode = LoadBalancingMode.Broadcast // Assurez-vous que ce mode est
pris en charge
            });
    }
    catch (Exception e)
    {
        _logger.LogError(e, "Error promoting to leader");
        return StatusCode(500, "Error promoting to leader");
    }

    return Ok("Promoted to leader");
}
```

2. Gestion des Interfaces

Définition

La gestion des interfaces consiste à organiser les interactions entre services de manière efficace en définissant clairement les points d'accès, les types de données échangées et les règles de communication.

Tactiques Appliquées

1. **Orchestration**
    - **Description** : Coordination centralisée des interactions pour accomplir une tâche complexe.
    - **Application dans le système** : Le **SideCard** joue un rôle dans l'orchestration des communications en interagissant avec le **STM** pour déterminer ou mettre à jour le rôle de leader.
2. **Adaptation d'Interface**
    - **Description** : Modification ou enrichissement des capacités d'une interface pour s'adapter aux besoins spécifiques d'un service.
    - **Application dans le système** : Le **STM** expose des capacités qui permettent au **SideCard** de vérifier ou de signaler des changements d'état liés au leader.

Exemple de mise en oeuvre dans le code

```
    [HttpGet]
    [ActionName(nameof(CheckIfLeader))]
    [EnableRateLimiting("fixed")]
    public async Task<IActionResult> CheckIfLeader()
    {
        _logger.LogInformation("Starting CheckIfLeader endpoint...");

        const int pingIntervalMs = 50; // Intervalle de ping en millisecondes

        // Obtenez l'ID du pod leader
        var podLeaderID = await ServiceMeshInfoProvider.PodLeaderId;
        _logger.LogInformation($"Pod Leader ID: {podLeaderID}");
```

```
        try
        {
            // Utilisez HttpContext.RequestAborted comme CancellationToken
            var cancellationToken = HttpContext.RequestAborted;

            // Appeler la route 'isLeader' en mode Broadcast
            var res = await RestController.Get(
                new GetRoutingRequest()
                {
                    TargetService = podLeaderID,
                    Endpoint = $"Leader/IsLeader",
                    Mode = LoadBalancingMode.Broadcast // Utiliser Broadcast pour
le ping
                });

            // Itérer sur les résultats asynchrones
            await foreach (var result in res!.ReadAllAsync(cancellationToken))
            {
                // Vérifier la réponse
                if (result.Content != null &&
JsonConvert.DeserializeObject<string>(result.Content) == "isLeader")
                {
                    _logger.LogInformation("Pod leader confirmed as leader.");
                    return Ok("Pod leader is confirmed as the leader.");
                }
                else
                {
                    _logger.LogInformation("Pod leader is not the leader.");
                    return Problem("Pod leader is not the leader.");
                }
            }
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, $"Service is not available. Detailed error:
{ex.GetType().Name} - {ex.Message}");
            return StatusCode(500, "Service is not available");
        }

        // Si aucune réponse n'est reçue, attendre et essayer à nouveau (en
boucle)
        await Task.Delay(pingIntervalMs, HttpContext.RequestAborted);

        // En cas d'absence de réponse, retournez une réponse par défaut
        return StatusCode(500, "No response from the leader check.");
    }
```

Synthèse

L'interopérabilité entre le **Sidecar** et le **LeaderController** du service STM repose sur des tactiques clés telles que la **découverte des services** et la **gestion des interfaces**. Ces approches favorisent une communication dynamique et flexible, essentielle au bon fonctionnement des microservices dans un environnement distribué.

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?

Définition de l'injection de dépendance

L'**injection de dépendance** (Dependency Injection, DI) est un modèle de conception appartenant au principe d'**inversion de contrôle** (Inversion of Control, IoC). Ce modèle consiste à déléguer la gestion des dépendances d'une classe à un composant externe, plutôt que de laisser cette classe les créer ou les gérer elle-même.

Une **dépendance** est un objet dont une classe a besoin pour accomplir ses tâches. Avec l'injection de dépendance :

- Une classe ne dépend plus directement de la création ou de la configuration de cet objet.
- Les dépendances sont fournies depuis une source externe (comme un conteneur DI ou du code utilisateur).
- Cela favorise un code **modulaire**, **maintenable** et **testable**.

Analyse : Rôle de l'injection de dépendance dans `StmClient` et `CompareTimes` dans le service TripComparator

---

1. StmClient

La classe `StmClient` utilise l'injection de dépendance pour gérer ses composants critiques.

Dépendances injectées dans `StmClient`

1. **ILogger** :
    - Fournit un mécanisme centralisé pour enregistrer les logs, gérer les erreurs et suivre les événements importants.
2. **IBackOffRetryPolicy** :
    - Implémente une politique de retry avec un délai progressif pour les opérations critiques.
    - Exemple : Réessaie jusqu'à 10 fois avec un délai croissant en cas d'échec.

3. **IInfiniteRetryPolicy** :
    - Implémente une politique de retry infini, garantissant que les opérations critiques aboutissent malgré des échecs temporaires.

Exemple d'injection de dépendance dans `StmClient`

Le constructeur de la classe `StmClient` montre comment ces dépendances sont injectées via l'injection par constructeur :

```
public class StmClient : IBusInfoProvider

    private readonly ILogger<StmClient> _logger;
    private readonly IBackOffRetryPolicy<StmClient> _backOffRetry;
    private readonly IInfiniteRetryPolicy<StmClient> _infiniteRetry;

    public StmClient(
        ILogger<StmClient> logger,
        IBackOffRetryPolicy<StmClient> backOffRetry,
        IInfiniteRetryPolicy<StmClient> infiniteRetry)
    {
        _logger = logger;
        _backOffRetry = backOffRetry;
        _infiniteRetry = infiniteRetry;
    }
```

Configuration dans le conteneur d'injection de dépendances

Ces dépendances sont également enregistrées dans le conteneur DI. Par exemple, dans la méthode `ConfigureServices` :

```
services.AddSingleton<IRouteTimeProvider, RouteTimeProviderClient>();
services.AddTransient<IBusInfoProvider, StmClient>();
services.AddTransient<IDataStreamWriteModel, DataStreamWriteModel>();
services.AddTransient<CompareTimes>();
```

Utilisation des dépendances injectées

- **Gestion des échecs avec les politiques de retry** :
    - Exemple : Dans `GetBestBus`, la politique `InfiniteRetryPolicy` gère automatiquement les erreurs :

```
return await _infiniteRetry.ExecuteAsync(async () => {
    // Exécution d'une logique pour obtenir les données de STM ou STM2
});
```

- **Logger pour la surveillance et les diagnostics** :
    - Enregistre les erreurs ou les réponses non valides :

```
_logger.LogWarning("Ignored response from non-leader instance.");
```

---

2. CompareTimes

La classe `CompareTimes` repose également sur l'injection de dépendance pour gérer les interactions avec les services et la logique métier.

Dépendances injectées dans `CompareTimes`

1. **IRouteTimeProvider** :
    - Fournit le temps moyen de trajet en voiture.
    - Exemple d'implémentation : `RouteTimeProviderClient`.
2. **IBusInfoProvider** :
    - Interagit avec STM et STM2 pour :
        - Trouver le meilleur bus (`GetBestBus`).
        - Démarrer le suivi d'un bus (`BeginTracking`).
        - Obtenir les mises à jour de suivi (`GetTrackingUpdate`).
    - Exemple d'implémentation : `StmClient`.
3. **IDataStreamWriteModel** :
    - Permet d'écrire les mises à jour de position dans un flux de données.
4. **ILogger** :
    - Enregistre les états critiques et les exceptions pour un diagnostic rapide.

Exemple d'injection de dépendance dans `CompareTimes`

Le constructeur de la classe `CompareTimes` montre comment ces dépendances sont injectées via l'injection par constructeur :

```
public class CompareTimes {
    private readonly IRouteTimeProvider _routeTimeProvider;
    private readonly IBusInfoProvider _iBusInfoProvider;
```

```
    private readonly IDataStreamWriteModel _dataStreamWriteModel;
    private readonly ILogger<CompareTimes> _logger;

    public CompareTimes(
        IRouteTimeProvider routeTimeProvider,
        IBusInfoProvider iBusInfoProvider,
        IDataStreamWriteModel dataStreamWriteModel,
        ILogger<CompareTimes> logger)
    {
        _routeTimeProvider = routeTimeProvider;
        _iBusInfoProvider = iBusInfoProvider;
        _dataStreamWriteModel = dataStreamWriteModel;
        _logger = logger;
    }
}
```

Enregistrement des dépendances dans `ConfigureServices` pour `CompareTimes`

Pour que les dépendances soient injectées dans `CompareTimes`, elles doivent être enregistrées dans le conteneur d'injection de dépendances dans la méthode `ConfigureServices` de `Startup.cs` :

```
public void ConfigureServices(IServiceCollection services)
{
    // Enregistrement des services nécessaires
    services.AddSingleton<IRouteTimeProvider, RouteTimeProviderClient>(); //
Fournisseur de temps de trajet
    services.AddTransient<IBusInfoProvider, StmClient>(); // Fournisseur
d'informations sur les bus
    services.AddTransient<IDataStreamWriteModel, DataStreamWriteModel>(); //
Modèle d'écriture pour le flux
    services.AddTransient<CompareTimes>(); // Enregistrement de la classe
CompareTimes
}
```

Utilisation des dépendances injectées

- **Étapes principales** :

    1. Récupération du temps de trajet en voiture via `IRouteTimeProvider` :

        ```
        _averageCarTravelTime = await
        _routeTimeProvider.GetTravelTimeInSeconds(startingCoordinates,
        destinationCoordinates);
        ```

    2. Recherche du meilleur bus via `IBusInfoProvider` :

```
_optimalBus = await _iBusInfoProvider.GetBestBus(startingCoordinates,
destinationCoordinates);
```

3. Suivi des bus via `IBusInfoProvider` :

```
await _iBusInfoProvider.BeginTracking(_optimalBus);
```

4. Écriture dans un flux via `IDataStreamWriteModel` :

```
await _dataStreamWriteModel.Produce(busPositionUpdated);
```

- **Logger pour le suivi des erreurs et des états** :
  - Exemple d'enregistrement de l'état actuel via Redis :

```
_logger.LogInformation($"Refreshed CurrentState: {state}");
```

Testabilité grâce à l'injection de dépendance

Avantages communs pour `StmClient` et `CompareTimes`

1. **Facilité d'isolation pour les tests unitaires** :
   - Les dépendances comme `ILogger`, `IBackOffRetryPolicy`, `IInfiniteRetryPolicy`, `IRouteTimeProvider`, et `IBusInfoProvider` sont injectées via des interfaces.
   - Cela permet d'utiliser des **mocks** ou des **stubs** pour simuler les comportements attendus, sans dépendre des implémentations réelles ou des services externes.
2. **Simulation de scénarios complexes** :
   - Les politiques de retry (comme `BackOffRetryPolicy`) peuvent être simulées pour valider la gestion des pannes dans des scénarios complexes (pannes réseau, indisponibilité des services).
   - Exemple pour `StmClient` : Simuler une défaillance temporaire des services STM/STM2 et valider que les politiques de retry réagissent correctement.
3. **Validation des interactions avec les dépendances** :
   - Grâce à des frameworks comme Moq ou NSubstitute, il est possible de vérifier que les dépendances injectées sont appelées avec les bons paramètres et au bon moment.
   - Exemple pour `CompareTimes` : Valider que `IDataStreamWriteModel` écrit correctement dans le flux après le suivi d'un bus.

4. **Réduction de la dépendance aux systèmes externes** :
    - Les services comme `RouteTimeProviderClient` ou `StmClient` peuvent être remplacés par des implémentations simulées, ce qui réduit la complexité et les temps d'exécution des tests.

Exemples spécifiques

Pour `StmClient`

- Lors d'un test, injecter un mock pour `IInfiniteRetryPolicy` permet de simuler un service STM qui échoue plusieurs fois avant de réussir. Cela valide que `StmClient` continue d'exécuter ses tâches malgré les pannes temporaires.

Pour `CompareTimes`

- En remplaçant `IRouteTimeProvider` par un mock, il est possible de tester comment `CompareTimes` réagit à des durées de trajet simulées (par exemple, un trajet en voiture plus court qu'un trajet en bus).

Impact sur la qualité et la maintenance du code

1. **Rapidité des tests** :
    - Les mocks permettent d'éviter les appels réseau ou les interactions coûteuses, rendant les tests plus rapides et fiables.
2. **Débogage facilité** :
    - En isolant les dépendances, les tests ciblent directement la logique métier, ce qui simplifie l'identification des bugs.
3. **Flexibilité pour les changements futurs** :
    - Les dépendances injectées peuvent être remplacées par de nouvelles implémentations sans modifier le code des classes `StmClient` ou `CompareTimes`.

Synthèse

L'injection de dépendance dans `StmClient` et `CompareTimes` joue un rôle crucial pour :

- Réduire le couplage.
- Améliorer la testabilité.
- Gérer efficacement les erreurs et les échecs temporaires.
- Rendre les classes plus flexibles et extensibles.

Ces avantages permettent à `StmClient` et `CompareTimes` de se concentrer sur leur logique métier principale, tout en déléguant la gestion des aspects spécifiques (retry, logging, interactions avec les services) à des composants dédiés.

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. **Équilibrage de Charge**

- **Kubernetes** : Kubernetes inclut un mécanisme d'équilibrage de charge au niveau réseau et application, redirigeant les requêtes vers les Pods disponibles.
- **NodeController** : Offre deux modes d'équilibrage de charge :
  - **RoundRobin** : Répartit les requêtes de manière séquentielle entre les services disponibles.
  - **Broadcast** : Envoie une requête à toutes les instances de service disponibles, ce qui est utile pour des cas spécifiques où tous les services doivent traiter la même demande.

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.

Synthèse

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

Annexes