Laboratoire #3

Groupe: 0x
Equipe: 0x

Membres de l'équipe:

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. Corrigez les problèmes des vues précédentes et montrez les changements apportés au laboratoire 3 pour résister aux nouvelles attaques, si applicables.

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.

Attribution des tâches

- Produisez une vue d'allocation de type attribution des tâches pour indiquer qui a travaillé sur quoi lors du laboratoire 3.

Questions

Question 1 : Lors de ce laboratoire, les queues de messagerie sont attaquées. Afin de ne pas perdre de messages, nous utilisons les quorum queues de RabbitMQ. Expliquez en détail comment fonctionnent ces quorum queues. Qu'arrive-t-il lorsqu'une quorum queue est détruite ? Qu'arrive-t-il lorsqu'elles sont toutes détruites?

Les **quorum queues** de RabbitMQ utilisent l'algorithme de consensus **Raft** pour garantir la cohérence des messages entre les répliques et assurer leur durabilité dans un cluster distribué.

Rappel : Algorithme Raft

Raft est conçu pour :

1. **Cohérence forte** : Toutes les répliques partagent le même état une fois les décisions prises.
2. **Disponibilité** : Le système reste opérationnel tant qu'une majorité (quorum) de répliques est disponible.
3. **Résilience aux pannes** : Le leader peut être remplacé sans perte de données confirmées.

Étapes principales du fonctionnement des Quorum Queues avec Raft

1. Élection du leader

- Une réplique est élue leader via un processus d'élection.
- Le leader coordonne toutes les opérations sur la queue, notamment la réception et la réplication des messages.

2. Écriture de messages

- Les producteurs envoient les messages au leader.
- Le leader inscrit chaque message dans un journal local (*log*).
- Ensuite, il propage les messages à toutes les répliques (followers).

3. Validation par le quorum

- Pour garantir la durabilité, le message doit être écrit dans le journal de la majorité (*quorum*) des répliques.
- Une fois qu'un quorum confirme la réception, le leader retourne un accusé de réception au producteur.

4. Réponse aux consommateurs

- Les consommateurs demandent les messages au leader.
- Le leader fournit les messages dans l’ordre confirmé par le quorum.

5. Changement de leader

- Si le leader tombe en panne, un nouveau leader est élu parmi les followers.
- Le nouveau leader est choisi parmi ceux possédant le journal le plus à jour.

Comportement en cas de panne ou défaillance

1. Panne d'un follower

- Si un follower devient indisponible, le quorum est maintenu tant que la majorité des répliques reste active.
- Le follower sera mis à jour par le leader lorsqu'il reviendra en ligne.

2. Panne du leader

- Si le leader tombe, un nouveau leader est élu parmi les répliques les plus à jour.
- Les messages confirmés ne sont pas perdus.

3. Perte de quorum

- Si moins de la moitié des répliques restent opérationnelles, la quorum queue devient indisponible.
- Aucune opération d'écriture ou de lecture ne peut être effectuée tant qu'un quorum n'est pas rétabli.

Avantages des Quorum Queues avec Raft

1. **Cohérence forte** : Les messages confirmés sont toujours disponibles.
2. **Durabilité** : Aucune perte de message confirmé grâce à la réplication.
3. **Tolérance aux pannes** : Le système reste fonctionnel malgré la perte de nœuds, tant que le quorum est maintenu.

Scénarios de défaillance

1. Perte du leader

- Un nouveau leader est élu parmi les répliques les plus à jour.
- La file d'attente reste disponible tant qu'un quorum est maintenu.

2. Destruction complète d'une quorum queue

- Si toutes les répliques d'une queue sont détruites, les messages non livrés sont perdus.
- Aucun nouveau message ne peut être envoyé ou consommé.

3. Destruction de toutes les quorum queues

- La destruction de toutes les quorum queues entraîne une perte totale des messages.
- Une récupération est possible uniquement via des sauvegardes externes (si configurées).

Synthèse

Les **quorum queues**, grâce à Raft, offrent une solution robuste pour garantir la durabilité des messages et la tolérance aux pannes. Cependant, pour éviter des pertes catastrophiques, il est essentiel de :

- Effectuer des sauvegardes régulières.

- Surveiller l'intégrité des nœuds du cluster.
- Configurer une haute disponibilité avec un nombre suffisant de répliques.

Question 2 : Quel type de « Test Double », tel qu'identifié par Martin Fowler, est utilisé dans les différents tests du projet? Identifiez un type de «Test Double» présent et présentez comment ce « Test Double » est utilisé dans le code. Montrez les extraits de code pertinents.

Le code suivant représente un **test double**, plus précisément un **stub** :

```
using Application.BusinessObjects;
using Application.DTO;
using Application.Interfaces;

namespace Configuration.Stubs;

public class StmClientStub : IBusInfoProvider
{
    public Task<RideDto> GetBestBus(string startingCoordinates, string
destinationCoordinates)
    {
        return Task.FromResult(new RideDto("0000", "0001", "0002"))!;
    }

    public Task BeginTracking(RideDto bus)
    {
        return Task.CompletedTask;
    }

    public Task<IBusTracking?> GetTrackingUpdate()
    {
        return Task.FromResult((IBusTracking) new BusTracking()
        {
            Duration = 1,
            Message = "i'm a tripcomparator stm client stub",
            TrackingCompleted = false
        })!;
    }
}
```

Un **stub** est un type de **test double** qui remplace une dépendance réelle pour fournir des données ou des comportements prédéfinis. Dans ce cas, le `StmClientStub` implémente l'interface `IBusInfoProvider` et remplace la logique réelle par des retours fixes.

Caractéristiques des stubs dans ce code

1. **Renvois prédéfinis**

- La méthode `GetBestBus` retourne toujours un objet `RideDto` avec des valeurs fixes : `"0000"`, `"0001"`, `"0002"`.
- Cela permet de tester le reste du système sans dépendre d'une implémentation réelle ou de données externes.

2. **Absence de logique complexe**

- Les méthodes, comme `BeginTracking`, ne font rien d'autre que retourner une tâche terminée (`Task.CompletedTask`).
- La méthode `GetTrackingUpdate` fournit des valeurs fixes pour tester des scénarios spécifiques, comme :
  - Une durée fixe (`Duration = 1`).
  - Un message défini : `"I'm a tripcomparator stm client stub"`.
  - Un état : `TrackingCompleted = false`.

3. **Isolation des tests**

- En remplaçant une dépendance réelle (potentiellement lente ou non disponible) par ce stub, le système peut être testé de manière isolée.

Différences entre ce stub et les comportements réels

1. **Absence de logique réelle**

- Le stub ne calcule pas réellement le meilleur bus.
- Il ne contacte aucun service externe ou API.
- Les valeurs sont fixes et servent uniquement à tester les interactions ou les comportements du système testé.

2. **Pas de dépendance externe**

- Contrairement au composant réel, le stub n'a besoin ni de réseau, ni d'accès à des données en temps réel.

Utilité du `StmClientStub`

Le `StmClientStub` est un exemple classique de **stub**, utilisé pour :

- Simplifier les tests.
- Isoler les tests des dépendances externes.
- Fournir des valeurs prédéfinies en remplacement des appels réels à un fournisseur de services de bus (`IBusInfoProvider`).

Il est particulièrement utile pour tester les interactions et les cas spécifiques sans dépendre d'un environnement réel.

Synthèse

Le `StmClientStub` est un exemple classique de **stub**, utilisé pour simplifier et isoler les tests.

Il fournit des **valeurs prédéfinies** en remplacement des appels réels à un fournisseur de services de bus (`IBusInfoProvider`), ce qui en fait un **outil utile** pour :

- Tester les interactions.
- Gérer des cas spécifiques.
- Réduire la dépendance à un environnement réel.

En isolant les tests des dépendances externes, le `StmClientStub` améliore la fiabilité et la rapidité des tests tout en garantissant une expérience de développement plus fluide.

- **Question 3** : Lorsqu'un utilisateur entre les coordonnées de deux points à Montréal entre lesquels ne circule pas un autobus, que se passe-t-il? Donnez un attribut de qualité concerné par cette situation. Identifiez une ou plusieurs tactiques pouvant être utiles dans cette situation et justifiez pourquoi.

Situation

Lorsqu'un utilisateur entre les coordonnées de deux points à Montréal entre lesquels **aucun autobus ne circule**, le système doit gérer cette situation de manière appropriée pour éviter une mauvaise expérience utilisateur.

Attribut de qualité concerné

L'attribut de qualité concerné ici est **l'utilisabilité**. Cela inclut :

- **Efficacité** : La capacité du système à fournir une réponse rapide et pertinente.
- **Satisfaction** : L'expérience utilisateur lorsqu'aucun trajet n'est disponible.

Comportement attendu

Lorsque le système détecte qu'aucun autobus ne circule entre les deux points fournis :

1. **Informer l'utilisateur** :
    - Afficher un message clair indiquant qu'il n'y a pas de liaison directe.
    - Fournir des explications ou des alternatives.
2. **Proposer des solutions alternatives (si possible)** :
    - Suggérer des trajets nécessitant des correspondances.
    - Recommander d'autres modes de transport ou des arrêts à proximité.

Tactiques pouvant être utilisées

1. **Cancel : Annuler la requête de recherche**

- **Tactique** : Permettre à l'utilisateur de stopper une recherche en cours.

- **Exemple** :
  - Ajouter un bouton **"Annuler"** pour interrompre une recherche de trajet longue ou non pertinente.
  - Envoyer une requête HTTP d'annulation au serveur (`POST /cancel` ou `DELETE /trip/{requestID}`).
- **Justification** :
  - Renforce l'**efficacité** et le **contrôle utilisateur**.
  - Réduit la frustration en cas d'attente inutile.

2. **Undo : Revenir à un état précédent**

- **Tactique** : Fournir la possibilité de revenir à une action précédente.
- **Exemple** :
  - Ajouter un bouton **"Annuler l'entrée précédente"** pour modifier les coordonnées ou réinitialiser les données saisies.
  - Stocker l'état précédent des données saisies pour permettre leur récupération.
- **Justification** :
  - Améliore la **tolérance aux erreurs** et la **satisfaction utilisateur**.
  - Offre une interaction fluide en permettant des corrections rapides.

3. **Assistance contextuelle**

- **Tactique** : Fournir des indications supplémentaires pour aider l'utilisateur.
- **Exemple** :
  - Une section d'aide expliquant comment utiliser le système.
  - Un lien vers une carte interactive ou un service d'assistance.
- **Justification** : Offrir un support pour guider l'utilisateur, améliorant ainsi son expérience globale.

4. **Gestion des erreurs**

- **Tactique** : Identifier les cas où aucun trajet direct n'est disponible et informer l'utilisateur avec un message explicite.
- **Exemple** :

  > "Aucun service d'autobus ne relie les emplacements sélectionnés."

- **Justification** : Réduire la frustration de l'utilisateur en expliquant la situation de manière claire.

Synthèse

Lorsqu'un utilisateur entre des coordonnées entre lesquelles aucun autobus ne circule, il est essentiel que le système gère cette situation de manière à maintenir une expérience utilisateur positive.

Pour ce faire, plusieurs tactiques peuvent être utilisées :

1. **Cancel et Undo** : Offrir des options pour annuler une recherche en cours ou revenir à un état précédent permet de renforcer l'efficacité et le contrôle utilisateur tout en réduisant les frustrations.
2. **Assistance contextuelle** : Fournir une aide claire et des suggestions alternatives guide l'utilisateur et améliore la tolérance aux erreurs.

3. **Gestion des erreurs** : Informer l'utilisateur avec des messages explicites garantit une meilleure compréhension de la situation et évite la confusion.

En combinant ces approches, le système devient plus utilisable, satisfaisant et adaptable, tout en offrant une expérience utilisateur fluide et intuitive.

- **Question 4** : Imaginez qu'on vous demande de réaliser un 4e laboratoire lors duquel votre système devra résister à des attaques visant les bases de données. Ces attaques comprennent des « Hardware Failures », donc le contenu des bases de données sera perdu à la suite des attaques. Décrivez-nous la stratégie que vous allez employer pour résister à ces attaques. Comment est-ce que les résultats obtenus lors de la démonstration 4 seront impactés?
- **Question 5** : Imaginez qu'on vous demande également d'implémenter la tactique d'utilisabilité « Annuler » (Cancel). Nous désirons permettre d'envoyer une requête HTTP « Annuler » au TripComparator pour annuler la comparaison du temps de trajet entre deux coordonnées. Expliquer les changements devant être apportés au projet pour pouvoir annuler la comparaison du temps de trajet. Créez un diagramme de séquence pour présenter visuellement les changements suggérés.

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-3".

\newpage

Annexes