Laboratoire #1

Groupe: 02

Equipe: 04

Membres de l'équipe: Ali Dickens Augustin, Jean-Philippe Lalonde, Jonathan Rodriguez Tames et Alexandre Roy

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
Ali Dickens Augustin	1
Jean-Philippe Lalonde	1
Jonathan Rodriguez Tames	1
Alexandre Roy	1

Introduction /2

Dans ce laboratoire, l'objectif principal est de comprendre et mettre en œuvre les concepts de microservice du projet LOG430-STM. De plus, chaque microservice joue un rôle distinct dans le but d'interagir afin de fournir des fonctionnalités telles que la collecte, le traitement et la comparaison de données liés au transport en commun des bus de la STM, et ce, en temps réel. L'architecture du projet implique entre autres une série de conteneurs Docker, qui doivent être déployés ainsi que configurés correctement dans le but de garantir la communication efficace entre les différents microservices. Ainsi, l'équipe est amenée à explorer ensemble diverses tâches, dont la configuration initiale du système, de l'exécution de microservices sur Docker, d'interagir avec des API externes et de manipuler des données réelles, et ce, en passant par l'optimisation des performances grâce à l'ajout de bases de données, l'ajout d'un ping echo et donc à la mise en œuvre de tactiques de résilience.

Explication générale du fonctionnement du système /2

- **NodeController** : Ce dernier est considéré comme étant le chef d'orchestre des tests de chaos, assurant ainsi la perturbation contrôlée des microservices afin de tester entre autres leur résilience. Ensuite, celui-ci a pour but de surveiller la durée de vie des différents conteneurs Docker et de fournir des mécanismes pour gérer, équilibrer ou même créer la charge de ces derniers. Ainsi, le NodeController a pour but de faciliter la connexion aux serveurs de l'ÉTS ce qui permet donc de communiquer avec le restant du système et son API permet d'ajuster dynamiquement les différents

services en cours d'exécution. Finalement, son équilibrage de charge utilise deux stratégies étant le Round Robin étant décrit comme distributeur de requêtes de manière aléatoire, et ce, parmi les instances disponibles. Le second étant Broadcast, qui lui envoie plutôt la requête à toutes les instances.

- **STM** : Le microservice STM agit de son côté comme étant un adaptateur pour l'API de la Société de Transport de Montréal (STM). Ce dernier fournit entre autres des fonctionnalités supplémentaires par rapport à l'API originale, dont l'obtention de données à intervalles réguliers (50 ms) et le suivi en temps réel des bus entre deux coordonnées spécifiques. Le STM est basé sur une architecture en couche, respectant donc le principe d'inversion des dépendances dans le but de faciliter toutes modifications et de maximiser la modularité. Ainsi, afin d'améliorer les performances de ce service, l'équipe doit donc configurer une base de données afin que celui-ci stocke ses données statiques GTFS dans cette dernière (PostgreSQL) plutôt que de la conserver en mémoire ce qui réduit alors l'utilisation de la RAM.
- **RouteTimeProvider** : Le microservice RouteTimeProvider a pour but de fournir des informations sur le temps de trajet basé sur les routes et les horaires disponibles pour les différents bus de la STM ou voitures, et ce, grâce à son utilisation de l'API TomTom. Au début, ce microservice n'est pas fonctionnel dans le projet et donc la création d'un fichier Dockerfile est obligatoire afin de rendre ce dernier opérationnel permettant ainsi la possibilité d'exécuter ses différentes fonctions. De plus, lorsque ce dernier configuré selon les normes du projet, il exposera une route nommée ping echo ayant pour but simple de vérifier son état de fonctionnement tous les 0.5 seconde, et ce, grâce à son instance. Si cette dernière est active, la route transmettra un message isAlive, permettant aux autres microservices de s'assurer qu'il est bien fonctionnel.
- **TripComparator** : Finalement, le TripComparator a pour mission de comparer les différentes options de trajet. Son fonctionnement est qu'il surveille en permanence l'état du RouteTimeProvider afin de vérifier sa disponibilité grâce à l'envoi du ping echo configuré préalablement. De plus, si le microservice RouteTimeProvider est redéployé ou bien détruit, ce dernier va détecter ce changement et donc identifier un nouveau port sur lequel l'instance sera possible. Ainsi, cette fonctionnalité de surveillance continue assure la réactivité et le fonctionnement opérationnel du système le rendant ainsi plus dynamique aux différents changements.

Diagramme de séquence de haut niveau

Diagramme de séquence /3

> TODO: Insérer le diagramme de séquence de haut niveau pour illustrer ce qui arrive lorsqu'une requête de comparaison de trajet provenant du dashboard arrive au système.

Diagrammes de contexte de haut niveau

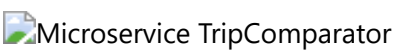
Microservice STM /3


Microservice STM

Microservice RouteTimeProvider /3


Microservice RouteTimeProvider

Microservice TripComparator /3



Questions générales /12

1. L'ajout de la base de donnés est une amélioration au système. Quel attribut de qualité est amélioré par l'ajout d'une base de données? Expliquez votre réponse. **/3**

Tout d'abord, l'ajout d'une base de données a pour d'améliorer plusieurs attributs de qualité dans un système en général. Dans notre cas, l'attribut de qualité ayant été amélioré lors de ce projet est principalement la performance. Celle-ci mesure entre autres la rapidité ainsi que l'efficacité avec laquelle le système va répondre aux différentes requêtes et traiter les données stockées. Dans le cadre du laboratoire, l'équipe a utilisé DBeaver afin d'ajouter une base de données PostgreSQL permettant d'optimiser l'interaction avec le microservice STM favorisant comme dit plutôt la performance. Ainsi, cette dernière permet de réduire les temps de réponse, et ce, en remplaçant la gestion en mémoire des données par une base de données optimisée, ce qui permet alors de récupérer les données GTFS plus rapidement. De plus, la base de données que l'équipe doit gérer est énorme et contient des millions de données provenant de la STM et donc une base de données comme PostgreSQL est plus optimale pour cela puisqu'elle est conçue de manière à gérer de grandes quantités de données, et ce, efficacement ce qui accélère donc le traitement des nombreuses requêtes de suivi de bus. Le temps de traitement entre les différents microservices dont l'envoi d'une requête de TripComparator et la réponse reçu par STM est ainsi réduit. Ensuite, l'instauration de la base de données aide aussi à l'optimisation des ressources système permettant un accès sélectif aux données pertinentes, évitant donc de charger toutes les données GTFS statiques en mémoire ce qui pourrait augmenter considérablement la RAM et entraîner des ralentissements dans le traitement de celles-ci. Ainsi, la charge sur le système sera diminuée, ce qui va permettre et aider une exécution plus fluide des autres microservices. Une autre amélioration serait au niveau de la scalabilité comme vue dans le cours. L'implémentation d'une base de données rend plus facile de gérer un grand nombre de requêtes en simultanées puisque PostgreSQL permet avec efficacité ce traitement de plusieurs connexions.

2. Lorsqu'une requête de comparaison du temps de trajet faite sur le Dashboard est envoyée au NodeController, différentes requêtes sont échangées sur le réseau. Pendant une minute, combien de requêtes sont échangées entre les microservices **NodeController**, **TripComparator**, **RouteTimeProvider** et **STM** et avec les API externes de la STM et de TomTom ? Décrivez également rapidement les différentes requêtes échangées. **/4**

Tout d'abord, à la suite de l'envoi de la requête, une série de ces dernières seront échangées entre les différents microservices et les API externe de STM et TomTom. Le NodeController va d'abord agir comme un superviseur. Il va donc superviser l'envoi des requêtes initiales et à chaque requête de comparaison de temps de trajet lancée, ce dernier va transmettre cette demande au TripComparator puis effectuer des tâches d'équilibrage de charge ainsi que de monitoring. Si nous évaluons le nombre de requêtes en 1 minute ayant été envoyées par ce microservice afin de superviser la création et le suivi des analyses de trajet, on pourrait en compter 2-3 voir plus si un conteneur est déployé ou bien détruit. Ensuite, le TripComparator, quant à lui, va recevoir la requête de comparaison de temps et celui-ci va l'a diviser en deux parties. Premièrement, il va envoyer une requête au STM dans le but d'obtenir les informations nécessaires sur la position des bus en temps réel et en deuxième lieu, il va transmettre une requête au RouteTimeProvider afin d'obtenir le temps de trajet en voiture, et ce, via l'API TomTom. Ce dernier va aussi faire certaines vérifications régulières toutes les secondes afin d'être sûr que le suivi des positions des bus est effectif et va de fait même traiter les mises à jour si nécessaire. Ainsi, cela peut donc impliquer une série de requêtes vers RabbitMQ en ce qui concerne la

gestion des messages. Finalement, on conclut en 1 minute qu'une moyenne de 6 requêtes est répartie entre les interactions avec RouteTimeProvider, STM et le message broker. RouteTimeProvider va donc utiliser l'API TomTom pour calculer le temps de trajet en voiture et transmettra une requête vers celle-ci à chacune des demandes de trajet. Ainsi, 1 à 2 requêtes est transmis vers l'API TomTom, et ce, dépendant le nombre de trajets comparés. En ce qui concerne le microservice de la STM, ce dernier s'occupe plutôt de trouver l'autobus étant le plus proche des coordonnées transmises et de suivre sa position en temps réel. Ces tâches impliquent donc d'envoyer une requête à l'API de la STM afin de recevoir des mises à jour à peu près toutes les secondes. Il va aussi régulièrement adresser des commandes afin de mettre à jour les informations des bus suivis et donc parmi les nombreuses requêtes envoyées constamment, en 1 minute, il y a environ 6 requêtes dont une, initialement pour trouver l'autobus puis d'autres selon un intervalle de 10 secondes afin de suivre sa position. Finalement, au niveau des API externes, celui de la STM effectue différents appels avec le microservice de la STM comme énoncé plus tôt, et ce, dans le but d'obtenir les informations sur les trajets des différents bus. TomTom va plutôt interagir avec le microservice RoueTimeProvider pour faire le calcul du temps en voiture et donc on conclut une moyenne de 1 requête pour chacun.

2.1 Dans le code source du projet, nous pouvons ajuster le taux de lancement des requêtes. Expliquez comment nous pouvons nous y prendre pour ce faire en donnant un ou des extraits spécifiques de code. **/2**

Il est possible d'ajuster le taux de lancement des requêtes, et ce, en contrôlant la fréquence avec Task.Delay. Tout d'abord, dans le service de suivi des trajets, l'intervalle de temps entre tous les envois de la commande UpdateRidesTrackingCommand est géré par un délai nommé Task.Delay. Ce dernier est défini par la variable UpdateIntervalInMs de sorte que la commande soit exécutée par une boucle infinie tant que le CancellationToken n'a pas arrêté le service. En voici un exemple :

while (!stoppingToken.IsCancellationRequested) { await commandDispatcher.DispatchAsync(new UpdateRidesTrackingCommand(), stoppingToken); await Task.Delay(TimeSpan.FromMilliseconds(UpdateIntervalInMs), stoppingToken); }

Ainsi, afin d'envoyer des requêtes de manière plus fréquentes, il suffit de changer et donc diminuer la valeur de UpdateIntervalInMs par exemple de 10 000 ms à 5 000 ms réduira de moitié le temps d'attente entre chaque requête. En ce qui concerne la fréquence de ces dernières, il faut augmenter la valeur afin que les requêtes soient espacées dans des périodes plus longues. Finalement, cela permet entre autre de contrôler avec précision comment le taux de requêtes envoyées par le microservice est géré tout en assurant un équilibre entre la nécessité d'actualiser les données et la charge du système.

De plus, afin d'ajuster le taux de lancement des requêtes, il est possible de limiter le débit des requêtes avec Rate Limiting. Ceci est un autre mécanisme de contrôle mis en place dans un des fichiers Program.cs où une certaine politique de limitation de débit y est configurée. Le nombre de requêtes est alors limité lorsqu'elles sont envoyées à une API tierce comme le démontre le code suivant :

builder.Services.AddRateLimiter(_ => _ .AddFixedWindowLimiter(policyName: "fixed", options => { options.PermitLimit = 2; options.Window = TimeSpan.FromSeconds(10); options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst; options.QueueLimit = 0; }));

Ainsi, en regardant le code ci-haut, on remarque plusieurs choses dont le PermitLimit ayant pour but de définir le nombre maximal de requêtes autorisées dans la fenêtre de temps définie. Ce dernier a été défini sur 2 donc seulement 2 requêtes sont autorisées. Window, quant à lui, va plutôt indiquer la durée de la fenêtre de temps pendant laquelle le nombre de requêtes est limité et donc 10 secondes dans notre cas. Finalement,

QueueLimit et QueueProcessingOrder permettant de mieux gérer les requêtes excédentaires et donc ici, aucune requête ne sera mise en file d'attente, car il est défini à 0.

Afin d'augmenter le nombre de requêtes ou bien ajuster la période, il faut respectivement augmenter la valeur PermitLimit ou bien modifier la valeur de Window. Cette approche permet ainsi de contrôler le volume de requêtes envoyé à des API externes comme ceux utilisés dans le projet soient TomTom ou STM assurant donc d'éviter les surcharges en plus de respecter les limites d'utilisation des services tiers.

3. Proposez une tactique permettant d'améliorer la disponibilité de l'application lors d'une attaque des conteneurs de computation (**TripComparator**, **RouteTimeProvider**, et **STM**) lors du 2e laboratoire. **/3**

Il y a plusieurs tactiques différentes qui permettraient entre autres d'améliorer la disponibilité de l'application lors d'une attaque des conteneurs de computation. Certaines ont été vues à travers la théorie du cours dont la redondance passive ou bien de son nom anglais, Standby Redundancy qui implique de maintenir des conteneurs de secours en veille tout en étant prêts à être déployés en cas de défaillance ou d'attaque des conteneurs principaux. Cela va donc permettre entre autres de garantir une transition rapide et fluide vers un conteneur opérationnel sans néanmoins compromettre la continuité du service. Pour ce faire, chaque microservice critique possédera un ou plusieurs conteneurs de secours configurés et donc en cas de défaillance détectée dans un de ces derniers, un conteneur de secours sera alors déployé et activé afin de reprendre la charge de travail, et ce, de manière transparente pour les utilisateurs dans le but de minimiser l'impact de l'attaque sur le service. Ainsi, cette approche a pour avantage de réduire le temps de récupération (MTTR) puisque le déploiement d'un conteneur de secours se fait immédiatement lorsqu'un autre est échoué et le système reste toujours disponible en tout temps améliorant donc la tolérance aux pannes. Une autre tactique serait la détection des défaillances en les surveillant, et ce, par l'utilisation de ping echo et/ou heartbeat. Pour ce faire, le ping echo enverrait un signal régulier à chaque conteneur dans le but de vérifier leur état de fonctionnement. Ainsi, si un de ces derniers ne répond pas, cela déclenche un basculement vers un conteneur de secours. Sinon, l'implémentation d'un système Heartbeat pourrait être effectué permettant un envoi périodique de chacun des conteneurs afin de montrer qu'ils sont toujours bien en vie. Si l'un d'eux cesse d'envoyer ces signaux, il sera considéré comme défaillant et donc le système activera une instance de secours. Finalement, une dernière tactique serait l'automatisation de la reprise qui aide à restaurer rapidement le service si une défaillance à lieu. Ce dernier a pour but de détecter la défaillance en surveillant activement le système, activer les conteneurs de secours après la défaillance prise en compte ainsi que réorienter le trafic vers le conteneur de secours afin de ne pas causer d'interruption sans le système et d'assurer une continuité fluide et rapide.

Conclusion du laboratoire /2

En conclusion, ce projet offre une expérience pratique permettant davantage la compréhension du déploiement et la gestion des microservices. De plus, en travaillant sur la performance du système ainsi que l'amélioration de la disponibilité, l'équipe a acquis plusieurs connaissances et différentes compétences notamment en configuration Docker, en surveillance de l'état des services par la configuration entre autres du ping echo et en intégration de bases de données. Finalement, le travail effectué sur les microservices sur la comparaison de trajets en temps réel a permis de développer une certaine expertise en ce qui concerne le fait de maintenir et concevoir des systèmes complexes, et ce, basés sur une architecture de microservices.

- Créer un tag git avec la commande "git tag laboratoire-1"