Architecture Microservices pilotée par les données

Eddy Caron, Zeina Houmani

Architecture Microservices pilotée par les données

Eddy Caron[1] et Zeina Houmani[1] *

Univ Lyon, CNRS, ENS de Lyon, Inria, Université Claude-Bernard Lyon 1,
LIP UMR5668. Lyon. France.

Résumé

Les mécanismes de découverte de microservices classiques sont normalement basés sur les besoins des utilisateurs *(Goal-based Approches)*. Cependant, dans les architectures actuelles qui évoluent fréquemment, les clients ont besoin de découvrir les fonctionnalités dont ils peuvent bénéficier avant de rechercher dans leur domaine les microservices disponibles. Cet article présente une architecture microservices pilotée par les données qui permet aux clients de découvrir, à partir d'objets spécifiques, les fonctionnalités qui peuvent être exercées sur ces objets ainsi que l'ensemble des microservices qui leur sont dédiés. Cette architecture, basée sur les composants principaux des architectures microservices classiques, adopte une stratégie de communication particulière entre les clients et les registres permettant d'atteindre l'objectif recherché. Cet article contient une représentation du modèle de données d'un microservice qui couvre les détails de la fonctionnalité fournie ainsi que les caractéristiques de la machine sur laquelle il est déployé. En plus, il présente un modèle pair à pair (peer-to-peer) qui permet de transformer notre architecture en un système robuste et évolutif. Enfin, nous terminons cet article avec une discussion des avantages et des problématiques rencontrées, ainsi que les travaux futurs.

1 Introduction

Les Architectures MicroServices (AMS) sont de plus en plus utilisées dans le développement des applications, surtout depuis l'émergence du *Cloud computing* et du *Fog computing*. Ce paradigme est une approche relativement récente consistant à développer une application distribuée en tant que suite de composants modulaires et autonomes, appelés microservices. Chaque microservice [2] est responsable d'une seule fonctionnalité et peut être réutilisé dans le cadre de n'importe quelle application puisqu'il s'exécute dans son propre processus et gère sa propre base de données. Grâce à ces caractéristiques, l'AMS est devenue aujourd'hui l'architecture idéale pour les applications où l'évolutivité, la résilience et la disponibilité sont requises, comme c'est le cas pour Netflix, Amazon, Ebay et récemment, les applications de l'Internet of Things [3, 4].

Dans la plupart des architectures microservices, certains composants notables sont toujours utilisés : (1) l'*API Gateway* qui est responsable de la gestion des microservices du système interne par l'implémentation des fonctionnalités telles que l'authentification, la transformation de données et la transformation de protocoles ; (2) le *service registry* qui représente la base de données contenant les descriptions des microservices disponibles. L'implémentation de ces composants dépend principalement des besoins de chaque application, ce qui offre plus de flexibilité dans la mise en œuvre d'architectures microservices répondant à des objectifs spécifiques telles que les architectures pilotées par les données responsables de la mise en place de mécanismes efficaces de stockage et de récupération des données, ainsi de la gestion des interactions complexes entre les clients et les composants du système.

L'objectif de cet article vise à présenter une architecture microservices pilotées par les données qui permet aux clients de découvrir, en garantissant une bonne performance du système,

*Merci à Daniel Balouek-Thomert (Rutgers University, NJ) pour nos discussions sur le sujet

les différentes fonctionnalités qui peuvent être appliquées à leurs objets ainsi que les microservices existants capables de les exécuter.

2 État de l'art

Grâce à l'indépendance des microservices et à leurs autonomies, l'AMS est devenue le choix pour la détection des synergies dans les réseaux de recherche. Un projet récemment publié [7], basé sur la science des données, utilise des techniques d'extraction de connaissances telles que l'exploration de données, la classification et la visualisation pour générer des graphiques permettant aux acteurs du réseau de découvrir les relations thématiques entre les différentes publications existantes. Il définit une architecture microservices pilotée par les données qui prend en entrée la publication du client et renvoie un graphique contenant toutes les publications dans le réseau de recherche ayant des thèmes en commun avec cette publication. L'idée de la découverte des publications dans ce projet ressemble à notre approche qui à partir d'un ensemble des microservices qui partagent entre eux des *topics* en commun (fonctionnalité, objet en entrée, paramètres, etc.), renvoi au client la liste des microservices qui correspondent aux *topics* demandés dans sa requête.

Les architectures microservices, comme tous les systèmes distribués, posent toujours des problèmes de sécurité. Ces problèmes ont été amplifiés avec ce paradigme puisqu'il consiste à exposer des interfaces API au réseau public pour assurer une communication avec toutes les entités. Afin de protéger nos microservices contre les accès illégitimes de clients non autorisés, une solution d'autorisation [6] a récemment été proposée qui consiste à implémenter dans les APIs de microservices, des politiques d'autorisation dynamiques composées d'ensembles de règles permettant de contrôler l'accès des clients aux données et aux fonctionnalités fournies. Cette autorisation peut être directement appliquée par le microservice ou déléguée de manière sécurisée à un autre. Dans cette solution, la logique de l'autorisation est complètement indépendant de la logique de traitement de données grâce à l'utilisation de *Feature toggles* (souvent appelées Feature Flags) et de *circuit breakers*.

3 Une architecture microservices pilotée par les données

3.1 Objectifs

Dans une architecture microservice classique, le mécanisme de découverte de services consiste à découvrir la liste des instances pouvant fournir une fonctionnalité déjà connue par le client. Cette liste contient normalement des informations de base sur la fonctionnalité et l'emplacement de chacun de ces microservices. Cependant, cette découverte ne semble utile que dans un système où les types de services fournis sont constants et toujours connus des utilisateurs. En effet, dans la plupart des architectures actuelles qui évoluent rapidement, de nouvelles fonctionnalités sont toujours mises en place. Afin que les microservices récents soient découvert, on souhaite étudier un système qui détermine les microservices disponibles selon un type de donnée, c'est-à-dire si un client à un objet de type image et de taille 50Mo, le système doit lui renvoyer toutes les fonctionnalités existantes qu'il peut appliquer sur cette image, comme par exemple `crop`, `resize`, etc .

Cette découverte de services peut être résumée en 4 étapes : d'abord, le client envoie une requête contenant au moins le type et le format de son objet à la base de données du système, le *service registry*. Ensuite, ce registre filtre l'ensemble des microservices stockés et renvoie au

client une liste de tous les microservice disponibles pouvant fonctionner sur ce type d'objet. Avec cette liste, le client est maintenant capable d'envoyer une requête en spécifiant la fonctionnalité qui lui convient avec les caractéristiques de son objet comme la taille, la dimension, etc . Ces caractéristiques permettent au registre de filtrer la liste précédente pour garder uniquement les microservices qui supportent ces valeurs. À ce stade, le client a découvert tous les microservices existants et peut contacter l'instance qui lui semble la plus appropriée en termes de performance de la machine, performance du réseau, paramètres demandés, etc .

Pour réaliser cet objectif, nous définirons en premier lieu une description de microservice contenant les informations principales sur les instances et leurs fonctionnalités. En deuxième lieu, nous proposerons un nouveau modèle d'une architecture microservices basée sur les composants principaux des architectures microservices classiques mais qui supporte un flux piloté par les données. En dernier lieu, nous allons proposer une architecture pair à pair (peer-to-peer) pour connecter les architectures microservices distribuées.

3.2 Modèle de données du microservice

Aujourd'hui, JSON et XML sont les formats les plus utilisés pour décrire les méta-données de services. Ces formats, qui peuvent être pris en charge par de nombreux langages de programmation, offrent une structure hiérarchique très flexible permettant aux services d'utiliser les balises les plus appropriées pour leurs catégories.

Afin d'atteindre l'objectif principal de notre architecture, on est besoin d'une description assez complète de chaque microservice disponible qui couvre non seulement les informations des machines sur lesquelles les microservice sont déployés, mais aussi des données détaillées sur les fonctionnalités qu'ils fournissent.

Chaque microservice, caractérisé par son identifiant et son adresse, doit spécifier aux clients les types d'entrées et de sorties supportées, les caractéristiques des objets d'entrée tels que la taille maximum et le format, le nombre minimal et maximal d'objets acceptés, ainsi que les paramètres à envoyer dans les demandes des clients pour bien utiliser cette fonctionnalité. Un exemple simplifié d'une description d'un microservice permettant de réaliser un `crop` (un recadrage) sur des images est présentée dans la Figure 1.

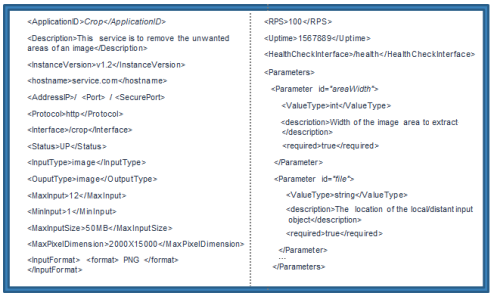

Figure 1 – Modèle de données pour un microservice `crop`

Dans cet exemple, le microservice `crop` v1.2 supporte une entrée de type *image* avec un format PNG uniquement et une taille qui ne dépasse pas le 50MB et la dimension 2000x15000. Pour profiter de ce microservice, certains paramètres doivent être spécifiés dans les demandes des clients, tels que l'emplacement de l'image, la largeur de la partie de l'image à supprimer, . En outre, selon la description ci-dessus, cette instance est déployée sur une machine qui traite

100 requêtes par seconde et qui fonctionne depuis 1567889 millisecondes sans aucune interruption due à une défaillance du système.
Toutes ces informations décrivent d'une part les performances de la machine sur laquelle ce microservice est déployé et d'autre part les détails de la fonctionnalité apportée par ce microservice.

3.3 Architecture

Afin de réaliser notre découverte de services en garantissant un accès sécurisé et authentifié au registre et en évitant la dégradation des performances au sein de notre *API Gateway* en raison de l'énorme trafic échangé entre les clients et les composants internes du système, nous allons utiliser la stratégie illustrée dans la FIGURE 2.

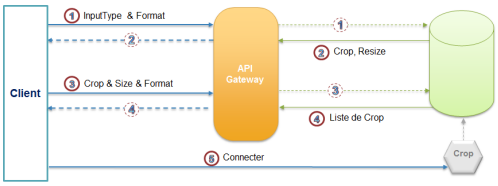

FIGURE 2 – Workflow de la découverte de services gérée par les données.

Dans cette stratégie, l'*API Gateway* est le composant intermédiaire entre le client et le registre uniquement : il est d'abord chargé d'envoyer au registre la demande de client (1) et de lui retourner la liste des fonctionnalités disponibles reçue (2). Ensuite, l'*API Gateway* est responsable de faire passer au client les descriptions complètes de tous les microservices (4) capables d'appliquer la fonctionnalité choisie sur l'objet indiqué (3). Afin d'éviter le goulot d'étranglement au niveau de la passerelle, le client communique directement avec le microservice choisi. Cette communication directe peut nécessiter la mise en place d'un composant intermédiaire tel que le système RabbitMQ chargé de transformer les protocoles et le format de données utilisés par les entités communicantes.

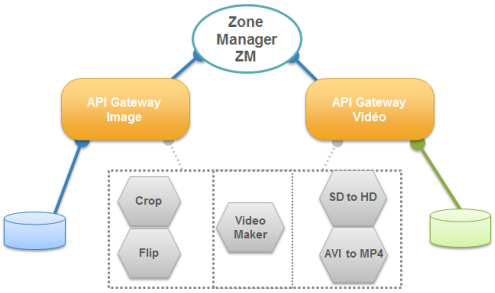

FIGURE 3 – L'architecture microservices multi-gateway d'une zone

Pour améliorer la performance de cette architecture dans la découverte de microservices, un modèle piloté par les données doit être créer. Ce modèle, illustré dans la FIGURE 3, consiste à implémenter des *API Gateways* basées sur le modèle *Backend For Frontend* (BFF) de Sam Newman [5] qui fournit une « *single purpose API Gateway* » spécifique à chaque type de client. Dans notre architecture, une API Gateway dédiée à chaque catégorie de microservice est implémentée tel que l'*image API Gateway* responsable de la gestion des microservices qui prennent en entrée ou donnent en sortie un objet de type image tels que, les microservices `flip`, `crop`, `rotate`, `video maker`, etc. Ce dernier qui nécessite d'avoir un ensemble d'images en entrée pour renvoyer une vidéo, appartient à deux catégories différentes de microservices : la catégorie des microservices *image* et celle des microservices *vidéo*. Dans ce cas, le microservice *video maker* est accessible à la fois via la passerelle spécifique de chacune de ces deux catégories.

Pour acheminer les requêtes des clients vers les *API Gateways* appropriées, un composant nommé "*Zone Manager (ZM)*" est utilisé. Ce composant qui représente le point d'entrée de notre architecture, vérifie les demandes des clients pour déterminer à quelle *API Gateway* ces requêtes doivent être envoyées. Une fois que l'*API gateway* choisie a reçu la liste des microservices disponibles de son registre dédié, il renvoie les résultats au gestionnaire de zone ZM qui, à son tour, les transmet aux clients.

Cette architecture permet d'une part de réduire le nombre de requêtes à traiter par l'*API Gateway* qui est devenue responsable d'une seule catégorie au lieu de tous les microservices du système ce qui améliore la performance de notre architecture. D'autre part, elle permet de bien gérer les descriptions des microservices : grâce à ce modèle utilisé, on est capable de mettre en place un cluster de *service registry* responsable uniquement des données des microservices de la même catégorie, comme par exemple un cluster de *service registry image* géré par l'*API Gateway image* et indépendant de tous les autres registres de la zone ce qui facilite le mécanisme de découverte de microservices dans notre architecture.

Lors de la découverte de microservices, il y a toujours une possibilité que le client ne trouve dans sa zone aucun service capable de traiter sa demande. Dans ce cas, cette demande doit être propagée vers les autres zones pour trouver le microservice adéquat. Afin de réaliser une communication entre les différentes zones, une connexion entre les gestionnaires de zones doit être gérée : le gestionnaire de zone qui ne reçoit de sa zone aucun microservice capable d'exécuter la requête du client, envoie en diffusion (ou multicast) cette demande à tous les gestionnaires des autres zones qui seront chargés de vérifier leurs registres dédiés puis renvoyer la liste des microservices capables de traiter la requête du client (s'ils existent) vers le ZM initial. Quand ce dernier reçoit la liste complète de toutes les zones, il renvoie cette liste au client qui, à son tour, choisit le microservice le plus adapté.

Cette architecture pair à pair entre les différents gestionnaire de zones, présentée dans la Figure 4, permet la création d'un système évolutif qui permet de relier les points d'entrée de toutes les zones en formant un système robuste pour le traitement des requêtes.

4 Discussions et problématiques

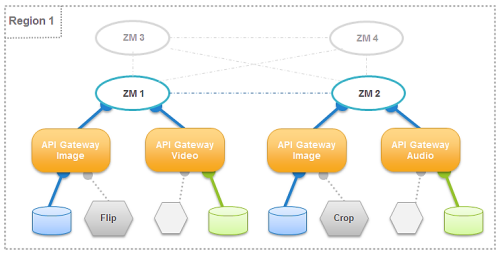

Figure 4 – La connexion inter-zone

Grâce à notre architecture microservices évolutive, les clients sont en mesure de découvrir toutes les fonctionnalités existantes pouvant être appliquées à leurs objets, même dans le cas de systèmes dynamiques où de nouvelles fonctionnalités sont régulièrement mises en place, ce qui n'est pas le cas avec les architectures microservices traditionnelles qui nécessitent d'avoir une connaissance préalable des services fournis pour tirer parti de ces fonctionnalités. Cette architecture proposée a réduit le goulot d'étranglement au niveau des *API Gateways* grâce à l'implémentation de *single purpose API Gateways* qui partagent entre elles les demandes des clients en fonction de la catégorie de leurs objets. En plus, cette réduction est réalisée grâce à la participation de ces passerelles uniquement dans la communication entre le client et le registre, ce qui nous a permis ainsi de profiter de la haute performance de la communication directe entre clients et microservices.

La stratégie de communication client/registre a permis de réduire le trafic réseau lors de la découvertes de services en renvoyant au client les descriptions complètes uniquement des

microservices qui exécutent la fonctionnalité déjà choisie.

La dégradation de la performance des instances est toujours un problème dans les architectures microservices puisqu'elle entraîne normalement la désactivation des instances jusqu'à ce que leur temps de réponse atteigne une valeur acceptable. Cela peut provoquer une chute excessive du nombre d'instances de certaines fonctionnalités, ce qui empêche les clients de bénéficier de ces microservices, même si elles existent.

Dans notre AMS, sacrifier la disponibilité de *service registry* n'est jamais une option puisqu'il est un composant essentiel pour la gestion des méta-données des microservices disponibles et c'est pourquoi, selon le théorème du *PACELC* [1], nous devons assurer une cohérence à terme des données enregistrées et alors des mécanismes de gestion des données incohérentes doivent être mis en œuvre.

5 Conclusion

Dans cet article, nous avons présenté une AMS dont la particularité est d'être pilotée par les données. Nous avons alors créé une description spécifique des microservices et adopté une stratégie de communication particulière avec la base de données du système. Cette architecture, caractérisée par son évolutivité et sa robustesse grâce au modèle pair à pair mis en place, permet aux clients de découvrir les fonctionnalités ainsi que les microservices disponibles dans un environnement dynamique et en évolution rapide.

Nos travaux futurs vont se focaliser sur le problème de la dégradation des performances des microservices, afin de mettre en place un système capable de s'auto-régler pour avoir une plateforme efficace selon le temps de réponse des instances : lorsque le temps de réponse des instances augmente au-delà d'un seuil, fixé par chaque microservice, ces instances doivent être désactivées afin qu'elles soient inaccessibles aux clients (elles sont alors mises en état de veille). Ces désactivations excessives entraîneront une baisse du nombre de services pour certaines fonctionnalités. À ce stade, le système doit être capable de déployer immédiatement de nouvelles instances ou de les réactiver lorsque cela est possible afin de gérer les nouvelles demandes des clients. Ce travail va être testé sur des microservices réels par un simulateur qui sera développé.

Références

[1] Daniel Abadi. Consistency tradeoffs in modern distributed database system design : Cap is only part of the story. *Computer*, 45(2) :37–42, 2012.

[2] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices : yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[3] Kamill Gusmanov, Kevin Khanda, Dilshat Salikhov, Manuel Mazzara, and Nikolaos Mavridis. Jolie good buildings : Internet of things for smart building infrastructure supporting concurrent apps utilizing distributed microservices. *arXiv preprint arXiv :1611.08995*, 2016.

[4] Kevin Khanda, Dilshat Salikhov, Kamill Gusmanov, Manuel Mazzara, and Nikolaos Mavridis. Microservice-based iot for smart buildings. In *Advanced Information Networking and Applications Workshops (WAINA), 2017 31st International Conference on*, pages 302–308. IEEE, 2017.

[5] Sam Newman. Pattern : Backends for frontends. https://samnewman.io/patterns/architectural/bff/, nov 2015.

[6] Davy Preuveneers and Wouter Joosen. Access control with delegated authorization policy evaluation for data-driven microservice workflows. *Future Internet*, 9(4) :58, 2017.

[7] Thomas Thiele, Thorsten Sommer, Sebastian Stiehm, Sabina Jeschke, and Anja Richert. Exploring research networks with data science : A data-driven microservice architecture for synergy detection. In Muhammad Younas 0001, Irfan Awan, and Joyce El Haddad, editors, *4th IEEE International Conference on Future Internet of Things and Cloud Workshops, FiCloud Workshops 2016, Vienna, Austria, August 22-24, 2016*, pages 246–251. IEEE Computer Society, 2016.