Towards service discovery and autonomic version management in self-healing microservices architecture

Yuwei Wang

Towards Service Discovery and Autonomic Version Management in Self-healing Microservices Architecture

Yuwei WANG
EDF Lab Paris-Saclay
Palaiseau, France
UMR SAMOVAR, CNRS, Télécom SudParis, Institut Polytechnique de Paris
Évry, France
yuwei.wang@telecom-sudparis.eu

ABSTRACT

Microservices architectures (MSAs) contribute to building complex distributed systems by decomposing monolithic systems into a set of independent microservices. This makes it possible to design, develop and deploy scalable and flexible systems. However, various unexpected changes could happen during execution, such as a service upgrade, a sudden increase of traffic, or an infrastructural failure. In this cases, how to react autonomously to these changes without outages becomes a challenge to consider. A PhD project has been launched to propose a self-healing microservices architecture, which can adapt dynamically to inside and outside changes without human intervention. In this paper, we present the first results of a systematic state of the art in the field of self-healing MSA systems. As an entry point of our research, we focus on self-healing triggered by upgrade changes. The initial contribution is a new component of a version manager in our self-healing MSA solution, in relation with service discovery elements. This approach can provide an autonomic version management on both the application level and the system level, and helps to control services upgrading changes. We plan to validate our proposition in a company project use case by deploying it in an emulated production environment, and applying a chaos engineering approach.

CCS CONCEPTS

• **Software and its engineering → Software architectures**.

KEYWORDS

Software architecture, microservices, self-healing, service discovery, version management, industrialization.

ACM Reference Format:
Yuwei WANG. 2019. Towards Service Discovery and Autonomic Version Management in Self-healing Microservices Architecture. In *ECSA 2019: 13th European Conference on Software Architecture, September 09–13, 2019, Paris, France.* ACM, New York, NY, USA, 4 pages.

ECSA '19, Sept. 09–13, 2019, Paris, France

1 INTRODUCTION

In recent years, microservices have become a new software architecture paradigm. In this approach, complex applications are divided into smaller, independent and loosely coupled services, and each service provides one functionality with one business objective. This type of architecture directly emerges from increased needs in terms of flexibility, scalability, and maintainability. The traditional service-oriented architecture (SOA) has been wildly used since 2000, but this architecture meets a bottleneck when systems need to scale up because of its centralized governance and integration mechanism such as enterprise services bus (ESB). A lot of companies have built up a technical debt over the last years. So, microservices architecture (MSA) is an alternative for them to become more effective at delivering into production environments [16].

However, the reliability of MSA systems decreases straightly with the growth of the number of microservices and the complexity of the systems [10]. Services can become temporarily unavailable because of the unexpected changes, such as network failures, suddenly increased service loads, release upgrades, configuration changes, etc. Moreover, unlike monolithic systems, microservices support the Conway's law [5] that means aligning microservices ownership to the structure of teams [12]. In other words, each microservice can be managed by different autonomous teams. Service dependencies will not be only controlled by their own teams. So, these unexpected changes across service boundaries become more difficult to deal with. In order to minimize the impact of partial outages of the system, it is important to design a microservices architecture with some self-healing capabilities [8] that can discover, study, and react to these changes. In addition, when changes happen in a system, how to reallocate and optimize the infrastructure resource usage is also a associated challenge to address.

In this context, a PhD research project has been launched and aims at proposing a microservices architecture with self-healing and autonomic management of resources. This architecture will answer the following main business requirements: (i) allowing services hot swapping and automated version upgrading of services without impacting the other services that depend on them, (ii) implementing automated microservices' migration following a failure of part of the infrastructure, (iii) making systems more resistant to the failure of one or more services, and making them possible to recover from an unhealthy state, (iv) taking into account load balancing, traffic routing, and resources allocation when system changes (including failures, scale changes, and evolutions).

Since building a complete architecture for self-healing covering all the desired use cases at once can be difficult, we will firstly focus

on the changes caused by microservices' version upgrading (the first consideration above) as a breakthrough. Because of upgrades, the network locations of microservices may change dynamically. A service discovery mechanism is therefore necessary in our architecture. Moreover, the problem of versioning has also come up due to the explosion of microservices.

The contributions of this paper are mainly the following: (i) We have started a systematic state of the art on self-healing, service discovery and versioning in MSA systems. This study considers the guidelines identified in [13]; (ii) We propose a preliminary reference architecture for microservices clusters with business nodes and manager nodes; (iii) We introduce a new version manager component able to provide an appropriate microservice version and that works jointly with service discovery.

The rest of the paper is organized as follows. Section 2 presents an overview of the related work. Section 3 provides the proposed solution. The excepted results are discussed in Section 4. Section 5 focuses on the plan for evaluation of the proposed microservices architecture. Section 6 finally concludes the paper.

2 RELATED WORK

This section discusses some works related to self-healing microservices systems, service discovery and versioning for microservices.

2.1 Self-healing in MSA

MSA systems are composed of a set of independent services to achieve a collective business goal, which makes them possible to realize run-time healing and dynamic adaptation of systems.

Nowadays in industry, the technologies of containerization such as Docker and the cluster orchestration platforms for automating operations such as Kubernetes simplify the resource management and supports basic self-healing for MSA systems.

Some papers also present their propositions about this topic. [17] describes an architecture for self-managing microservices to build a scalable and resilient system in a cloud environment. This architecture implements the management logic within the managed MSA applications rather than as a third-party service. It draws support from the distributed hierarchical storage and a consensus algorithm to elect a leader to provide health management and auto-scaling functionality. When failures happen, the services can be restarted on another node that shares service configurations. [6] introduced an external and decentralized management approach called GRU to integrate an autonomic manager into Docker-based MSA systems. GRU is based on a multi-agent system and creates a new abstraction level with agents units implementing a decentralized MAPE-K [9] autonomic loop on the top of Docker containers. GRU agents allow to manage a set of containerized microservices and take decisions according to their states and also to the status of their neighbors when the environment changes.

Throughout the existing researches, we find that most of the works were published in 2015 and 2016; there are only a few new results coming out in the last years directly addressing self-healing microservices architectures. When mentioning self-healing systems, most papers talk about the capacity of detecting and then recovering from failures. However, in our work, we intend to extend self-healing to a broader meaning of adapting dynamically to more various changes.

2.2 Service discovery in MSA

Service discovery is not a new middleware service. DNS (Domain Name Server) is one of the simplest approach that maps domain names with IP addresses. But in this case, the address is often hard-written in applications. This does not allow to meet new challenges when it comes to microservices and an increasing number of services resulting from dynamic changes. So, we need more advanced mechanisms to solve this problem. There are some existing solutions , mostly used in the industry. Consul [7] is a distributed key-value store that provides service discovery and integrates monitoring as well as health checking. It uses client agents and RAFT quorum algorithm to ensure consistency between instances. It also provides service membership management and message broadcasting through a gossip protocol. Eureka [11], from Netflix, is a RESTful service discovery tool with a client/server architecture. There is a Eureka server per data center and Eureka clients use an embedded Java component or a sidecar to register and discover services. It provides also heartbeats and TTL (Time to Live) to achieve high availability. Synapse [1], from Airbnb SmartStack, is based on HAProxy to route requests to microservices. It includes watcher components to frequently check for changes of service address and then to update configuration in HAProxy.

[15] mainly addresses service discovery in MSA systems and proposes a solution called Serfnode, which is based on Docker and the Serf project. A Serfnode agent encapsulates one or more Docker images and a supervisor instance, and it communicates with an event system via a Gossip protocol. It provides a simple monitoring and self-healing mechanism through a third-party process control system called Supervisor.

In the state of the art, few of the works associate service discovery with self-healing. It has to be noted that a wide gap appears between academic research and industrial solutions. In addition, we consider that service discovery should work together with a version management component to control the changes of services versions.

2.3 Version management in MSA

In regard to version management in MSA systems, [4] discusses three patterns to handle different service versions: immutable server pattern, blue green deployment, and canary release. Immutable server pattern means that once the server is deployed in production environment, when changes happen, it should be replaced with a new updated instance rather than be modified. Blue-green deployment and canary release resolve the problem of introducing the new versions without downtime. When deploying a new version of a microservice, the old one and the new one are run in parallel. Blue-green deployment routes traffic immediately to the new version but canary release routes it iteratively. In [12], the author describes two possibilities for the coexistence of different versions: maintaining old and new versions of microservices resources, or old and new interfaces in the same running microservice. [14] proposes

a system-level autonomous healing tool called "app-bisect" to troubleshoot and repair MSA applications in production environments. The idea is that the dependencies of microservices are represented as a graph change; a particular past version is searched and deployed automatically; the candidate version is sped up to be chosen by using canary testing and version-aware routing techniques.

Although several articles have mentioned this subject and its models, they do not provide more details than the main idea. They only consider the impact of upgrading the service in a single application, which means the versioning at the application level. In the context of a business application, the microservices in another application developed by another team are often considered as external components. As an example, consider the scenario in Figure 1. An application A is composed of a set of microservices and is managed by an internal or an external team A in the company. An application B also includes a set of microservices managed by team B. For reasons of cost reduction and infrastructure reuse, application B can use a subset of microservices and also its infrastructure underlying Application A. In fact, both applications are developed and maintained independently. This reused subset, like MS1 in figure, is considered as an external component by the development team B. If the microservices on which the application B depends are updated, there is a risk of breaking the application. Thus, the overall management of the microservices version among several applications at the company level becomes an important issue. Our solution will take into account versioning both at the application level and at the company level.

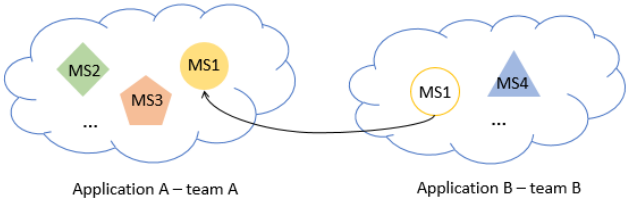

Figure 1: Example of versioning problem in microservices

3 PROPOSED SOLUTION

This section presents the preliminary design of the solution we propose in the context of industry.

To meet the main expected business requirements discussed in Section 1, in the context of industry, virtualizing the microservices by mixing and matching containers with virtual machines is a major step in our design of MSA infrastructures. Figure 2 illustrates a reference architecture of clusters for our self-healing MSA. In this approach, the microservices are containerized as business nodes that run within a group of virtual machine (VM) instances. Another group of VMs is allocated for some manager nodes that help independent microservices for working together in a distributed environment and for making complex system changes easier to manage. To avoid any single point failure, there are several manager nodes. To facilitate the communication between instances of each node and reduce complexity, we can use a message broker.

Figure 3 shows an outline of our cluster manager node. It takes care of orchestrating, scheduling and controlling the microservices

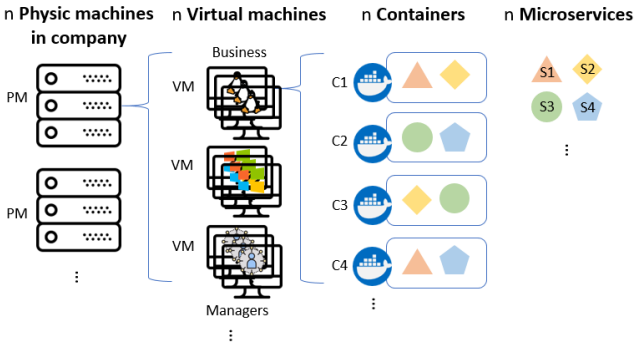

Figure 2: A reference architecture for clusters

in the clusters. We introduce a new component for managing version upgrades. In the following, we present the main components in a manager node.

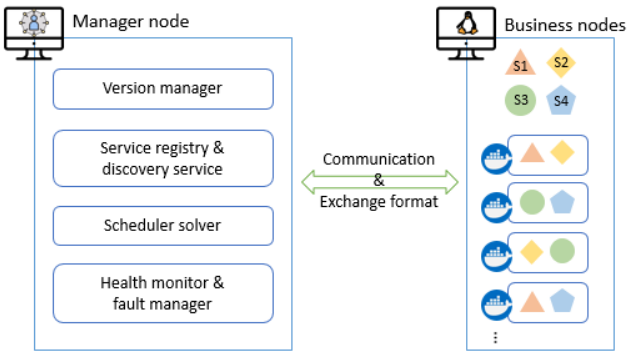

Figure 3: Main concerns in the design of the manager node

The version manager focuses on the version changes due to microservices' upgrades at the application level and company level. At the application level, the changes introduced could be minor, which means that their dependent services do not need to modify the called APIs, and the version manager can redirect immediately to the new version. Or they could be major, in which case these changes are destructive for other microservices, and the version manager should gradually route the traffic to the new version. At the company level, the version manager also takes in charge the control of the utilization of each version and when the older one should be considered as obsolete.

The service registry and discovery service is a mechanism for dependent services to locate each other dynamically on the network. It comprises a service registry, which is a database to maintain microservices information including IP address, port number and other useful metadata such as version, service name, and so on. Microservices exchange with the service registry to record their locations and discover other registered microservices.

The schedule solver plays the role of a planner and a load balancer. It takes decisions for microservices scheduling by launching specific constraints to optimize clusters resource usage, and to provide high availability. It considers the status of each microservice

and route the traffic to relative health instances. This component also includes a priority engine that identifies a priority level for each system changes based on different metrics—i.e., a higher priority level means a more urgent situation to solve.

The health monitor and fault manager address the system quality attribute of fault tolerance. This component refers to the ability of detecting, correcting and even preventing failures. It takes responsibility for collecting and checking microservices status at run time. It relies on a score engine to calculate scores for each microservice on the basis of defined criteria and weight factors, such as the load of services, response time, resource usage, failures, etc. The higher the score obtained by a microservice, the healthier it is. It also carries out different strategies to minimize the impact of partial outages. Common architectural patterns and techniques to isolate failures and to achieve graceful service degradation are considered, such as failover caching, circuit breakers, rate limiters, etc. [2].

The problem of communication and data exchange format relates to interactions in the clusters among different nodes and among different components in a node. MSA applications are distributed systems running on multiple machines. So how to stitch them together with different communication styles for different scenarios and goals should also be considered in our architectural design: e.g. the communication protocol can be synchronous or asynchronous; the interaction can be one-to-one or one-to-many, and the exchange message formats can be text-based formats such as JSON or binary format such as Protocol Buffers.

In this paper, we give a bird eye view of our self-healing MSA design. The next step will be entering into the details of each part of the architecture, starting from the problems of version management.

4 EXPECTED CONTRIBUTIONS

The expected contributions of our work include both theory and practice. The first contribution is expected as a systematic state of art study, following guidelines proposed in [13], performed on self-healing microservices architectures. It provides a comprehensive overview of the existing challenges in the field and also points out the current gap between academia and industry. The second contribution concerns an architectural proposition for microservices-based systems that have the ability to react autonomously to changes (including failures, variations in scale and evolutions). Especially, this architecture enables a new approach to manage the versions of microservices without human intervention. The third contribution could be an experience report or an evaluation report that applies our proposition to an industrial case. We aim to help companies improve robustness as well as performance of systems and reduce their maintenance costs.

5 PLAN FOR EVALUATION

In order to evaluate our propositions, we will use them for the design and development of a company project use case. This project was developed several years ago with monolithic architectures and now has difficulty to be maintained and scaled. We will help to optimize this system by using MSA and demonstrate the feasibility of our design. We plan to deploy this system on physical infrastructures and a private cloud platform in the company, which follows the reference architecture described in Figure 2. We will also pilot it into production environment. Moreover, we will introduce chaos engineering [3] into our evaluation, which means proactively testing how our system responds to various changes in a production environment. This implementation and testing will help to validate the performance and resiliency of our system. In addition, how to evaluate by benchmarking and which benchmark suite to use are open issues to be discussed.

6 CONCLUSION

In this paper, we presented a research summary of a PhD project for discussing the topic of self-healing microservices systems. Our project is still in a very early stage. We defined the context of our research and the potential problems. As self-healing involves different types of changes: upgrades, failures and scales, we identified an entrance for our work that is the problem of service discovery and versioning in microservices systems. We will evaluate our future solutions by applying into a company project use case.

ACKNOWLEDGMENTS

This work is supported by a CIFRE convention of the ANRT and the French Ministry of Higher Education, Research and Innovation.

REFERENCES

[1] Airbnb. 2012. Synapse. Retrieved May 2018 from https://github.com/airbnb/synapse
[2] N. Alshuqayran, N. Ali, and R. Evans. 2016. A Systematic Mapping Study in Microservice Architecture. In *Proc. IEEE 9th International Conference on Service-Oriented Computing and Applications*. Macau China.
[3] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. 2016. Chaos Engineering. *IEEE Software* 33, 3 (May 2016), 35–41.
[4] B. Butzin, F. Golatowski, and D. Timmermann. 2016. Microservices approach for the internet of things. In *Proc 21st IEEE International Conference on Emerging Technologies and Factory Automation*. Berlin Germany.
[5] M.E. Conway. 1968. How Do Committees Invent? *Datamation* 33, 3 (April 1968). http://www.melconway.com/research/committees.html
[6] L. Florio and E. Di Nitto. 2016. Gru: An Approach to Introduce Decentralized Autonomic Behavior in Microservices Architectures. In *Proc. 2016 IEEE International Conference on Autonomic Computing*. Wurzburg, Germany.
[7] HashiCorps. 2014. Consul by HashiCorp. Retrieved May 2019 from https://www.consul.io/
[8] M.C. Huebscher and J.A. McCann. 2008. A survey of autonomic computing-degrees, models, and applications. *Comput. Surveys* 40, 3 (Aug. 2008), Article No. 7.
[9] J.O. Kephart and D.M. Chess. 2003. The vision of autonomic computing. *IEEE Computer* 36, 1 (Jan. 2003), 41–50.
[10] T. Killalea. 2016. The hidden dividends of microservices. *Commun. ACM* 59, 8 (Aug. 2016), 42–45.
[11] Netflix. 2012. Eureka at a glance. Retrieved Dec 2014 from https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance
[12] S. Newman. 2015. *Building microservices: designing fine-grained systems*. O'Reilly, USA.
[13] K. Petersen, S. Vakkalanka, and L. Kuzniarz. 2015. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology* 64, C (Aug. 2015), 1–18.
[14] S. Rajagopalan and H. Jamjoom. 2015. App–Bisect: Autonomous Healing for Microservice-Based Apps. In *Proc. 7th USENIX Workshop on Hot Topics in Cloud Computing*. Santa Clara, CA, USA, 16–16.
[15] J. Stubbs, W. Moreira, and R. Dooley. 2015. Distributed Systems of Microservices Using Docker and Serfnode. In *Proc. 7th IEEE International Workshop on Science Gateways*. Budapest, Hungary.
[16] J. Thönes. 2015. Microservices. *IEEE Software* 32, 1 (Jan. 2015), 116–116.
[17] G. Toffetti, S. Brunner, M. Blöchlinger, F. Dudouet, and A. Edmonds. 2015. An architecture for self-managing microservices. In *Proc. 1st International Workshop on Automated Incident Management in Cloud*. Bordeaux France, 19–24.