An Implementation of Service Composition for Enterprise Business Processes

IOP Conf. Series: Earth and Environmental Science **234** (2019) 012091 doi:10.1088/1755-1315/234/1/012091

An Implementation of Service Composition for Enterprise Business Processes

Jian Ke, Jian Bo Xu and Shu Feng

Hunan University of Science & Technology, Xiangtan 411201, Hunan, China
Email: hellokejian@163.com; jbxu@hnust.edu.cn; fengshu_gis@163.com

Abstract. The Microservice Architecture (MSA) is an advanced architecture with flexible technology selection, independent on-demand expansion, and high availability. It is one of the best solution for enterprise software systems to cope with cloud deployment. Moreover, Spring Cloud provides comprehensive technical support for microservice architecture; also, it is the best technical framework for implementing microservice architecture. The container technology represented by Dockers provides an independent and undisturbed deployment environment for microservice architecture. Based on the microservice architecture and lightweight container technology, this article propose an implementation idea for service composition.

1. Introduction

In the process of enterprise informatization construction, it is essential to integrate business processes with information technology. Lacking of overall planning and theoretical support leads to the following three features of the software architecture in the enterprise:

(1) The monolithic architecture became the main deploying mode [3]. Monolithic architecture software is easy to debug in the early stages of development, runs simply, easy to deploy. What we needs to do is just copy the packaged application to the server. By running multiple copies of the stateless service on the back end of the load balancer, we could easily realize the horizontal expansion of the application, and the operation or maintenance threshold is low.

(2) As demand changes, the system gradually becomes much more complex, new developers cannot figure out the business logic. Therefore, fixing bugs and adding new features is quite difficult and time-consuming. In the end, the system would fall into a huge, incomprehensible quagmire.

(3) Traditional development models have no advantages in cost and efficiency, which would limit the development of enterprises. Monolithic applications also make it quite difficult to adopt new architectures and programming languages. Eventually, we cannot achieve agile development or even rapid deployment with Non-expanding, low reliability applications.

2. Related Work

The Microservice Architecture (MSA) is an emerging cloud software system, which provides fine-grained, self-contained service components (Microservices) used in the construction of complex software systems. Facing the problem that SDLC-driven methods (SDLC: software development life cycle) are lacking to facilitate the migration of software systems from a traditional monolithic

architecture to MSA. Chen et al [1] proposed a migration process based on SDLC, including all of the methods and tools required during design, development, and implementation.

Well-designed microservice architecture with better quality relies on clear understanding of related quality attributes. However, current understanding of quality attributes in microservice architecture is deficient and not comprehensive. Reference [2] constructed knowledge of quality attributes in architecture through a Systematic Literature Review (SLR), the exploratory case study and the explanatory survey. By analyzing the influential factors and the corresponding tactics of related quality attributes, their research is aiming at providing a comprehensive guide on quality improvement in microservice architecture.

Mario Villamizar et al [8] presented a cost comparison of a web application developed and deployed using the same scalable scenarios with three different approaches [5]: 1) a monolithic architecture, 2) a microservice architecture operated by the cloud customer, and 3) a microservice architecture operated by the cloud provider. Test results show that microservices can help reduce infrastructure costs in comparison to standard monolithic architectures. Moreover, the use of services specifically designed to deploy and scale microservices reduces infrastructure costs by 70% or more. They also described the challenges that implementing and deploying microservice applications.

Based on IoT Clouds the container virtualization is becoming an even more prominent technology that allows them to deploy and manage, in a flexible fashion, micro-services within IoT devices. In reference [4], the authors focused on micro-service reliability in IoT devices and proposed a system based on container virtualization that allows IoT Clouds to carry out fault-tolerance when a microservice running on an IoT device fails.

3. Implementation of Service Composition Based On Microservice Architecture

3.1 Description

To reduce the coupling between sub-systems in enterprise information systems, it is normal to split the system into multiple components, which helps to separate component boundaries and responsibilities. Programmers could upgraded or maintained the system independently by. The main purpose of service-oriented functional component is to encapsulate functional components implemented by different programming languages into services. After that, the client program written in different programming languages performs cross-language/environment call to the service interface, and the effect of the functional component service and the cross-language service interface call is in Fig. 1.

3.2 The Principles of Service Composition Solution

In the early stage of enterprise software development, Application normally adopted monolithic architecture that provides a list of services S (S1, S2...Sx), the schematic of the monolithic architecture is in Fig. 2, and a group of developers develops the code. As applications expands, more services or developers would be added, increasing the complexity and time required to launch new features or improvements.

The complexity of large application is solved by Service-Oriented Architectures [3] solutions, application is composed of a series of monolithic applications (a1, a2...ax) with each application providing service through different standard (such as Simple Objects Access Protocols). Some systems used routing mechanisms, such as Enterprise Service Bus (ESB) [5] to route/send messages between applications. The SOA strategy allows each application be developed by a team of developers T (T1, T2…Ty) (which are regularly grouped by business functions) and operated by the operator team O.

While SOA implementations could address the needs of certain companies, they are quite complex, expensive, and even time consuming [6]. One typical implementation of SOA is ESB, which designed for effectively support enterprise applications with numerous users. When facing the challenge of scaling ESB to hundreds of thousands or millions of users, they become the bottlenecks for creating

high latency and increasing the likelihood of single failure point. Therefore, it is complicated to add or remove servers to ESB as needed. As for agility, it requires quite a huge amount of configuration in ESB to feed new needs for end users, which will consume a lot of time.

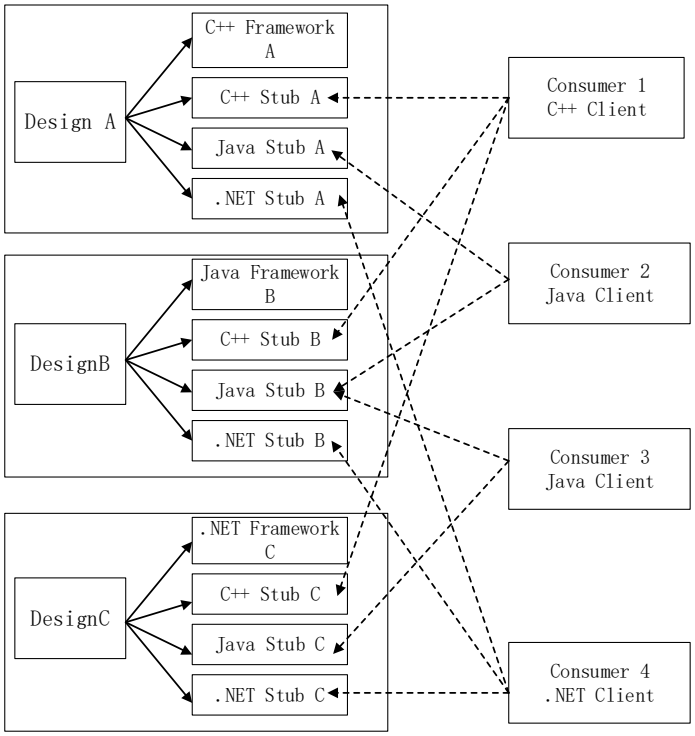

Figure 1: service-oriented functional component

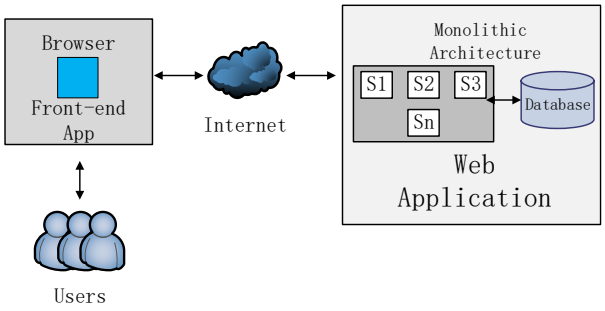

Figure 2: Monolithic architecture

As a lightweight subset of the SOA, MSA (Microservice Architecture) absorbs the advantages of the SOA architecture and avoids the corresponding problems of the monolithic architecture. MSA is a solution of building applications using a set of microservices. It is composed of multiple services in the form of separate business units and implemented around the specified business through appropriate technologies. Each microservice runs in a separate process and relies on independently automatic deployment mechanism, and forming a high cohesive autonomous unit with clear boundaries; Microservices communicate through some lightweight communication mechanisms, such as RPC, HTTP and so on.

It is proposed to turn the application into a set of microservices mS (mS1, mS2 ... mSn), each of which provides a subset of the services S (S1, S2...Sx). Development team mTi independently develope and test these microservices using the technology stack (including presentations, services,

and persistence layers) to be more applicable to services provided by microservices. The team mTi is also responsible for deploying, extending, operating, and upgrading microservices on the cloud computing IaaS/PaaS solution. In presentation layer, the service is released that using the Representational State Transfer (REST) [7]. The schematic diagram of the microservice architecture is in Fig. 3:

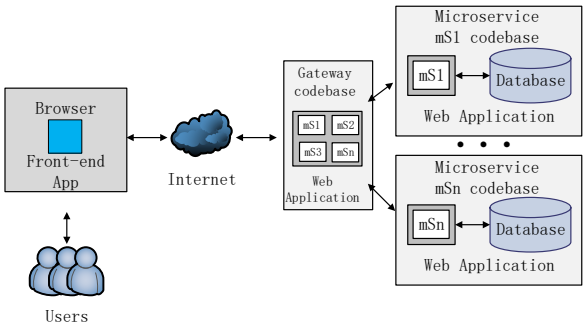

Figure 3: Microservice architecture

Spring Cloud is a brand new web framework by the Pivotal team. Its main feature is to simplify the developing/deploying process. Spring Cloud contains a set of well-functioning and lightweight micro-service components based on Spring Boot. The key characteristics of Spring Cloud are as follows: service discovery management, service fault tolerance, service gateway, and service configuration, load balancing, and messaging. There are also well-tested and mature components in terms of bus and service tracking.

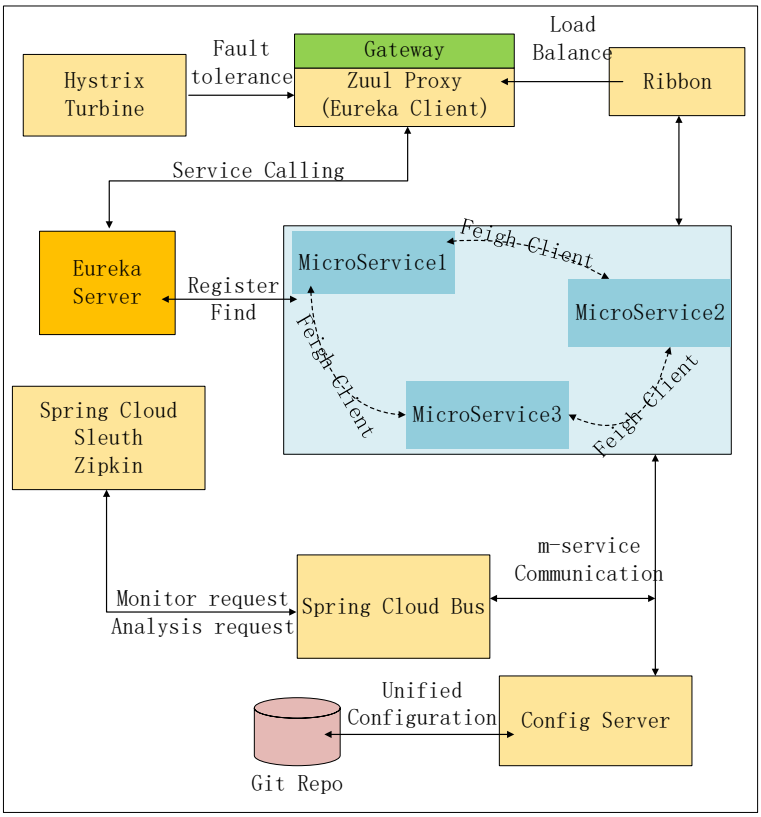

Figure 4: The architecture of Spring Cloud

Fig. 4 shows the complete architecture diagram of Spring Cl-oud. Eureka implements the automatic registration and dis-covery of microservices in Spring Cloud. Zuul is used for dynamic routing and

request filtering. Ribbon is client-side load balancing based on HTTP and TCP that gets a list of services from the Eureka Registry. HyStrix is a fuse that improves the system's fault tolerance. Turbine is a tool introduced to monitor microservice clusters. Feign integrates Ribbon to provide a declarative HTTP API to clients. Spring Cloud Config provides unified configuration management for the Spring Cloud framework system, and provides support for the server (Config Server) and the client (Config Client). The role of the Spring Cloud Bus is to connect the service nodes with a lightweight message broker (such as RabbitMQ) and broadcast the communication between the dynamic information of the configuration file and the service. Spring Cloud Sleuth integrates ZipKin to implement monitoring link analysis of microservices.

Microservice is an advanced architecture, but there are unavoidable drawbacks in terms of system complexity and continuous integration of services. Therefore, we introduced Docker technology. Docker is an open source container engine that complies with the Apache 2.0 protocol. It uses lightweight virtualization technology to achieve resource isolation and package various environment dependencies and applications to facilitate application porting and deployment. We package the microservices into separate Docker images, and then push them into the private image repository. Each time the service deployed, we pull the corresponding image from the private image library, and the image run according to the scheduled microservice.

3.3 Implementation Technology

Since Spring cloud platform is based on the Java language, to publish programs written in different languages into microservices with unified communication standards, such as C++, .NET, Python, Matlab and other languages or tools, we can use the corresponding technology. We decorated the underlying system as a Java program using technology such as JNI (solving C++ and JAVA communication problems), inter-process communication, and RPC (Remote Procedure Calling), thus solving the problem of making functional component service.

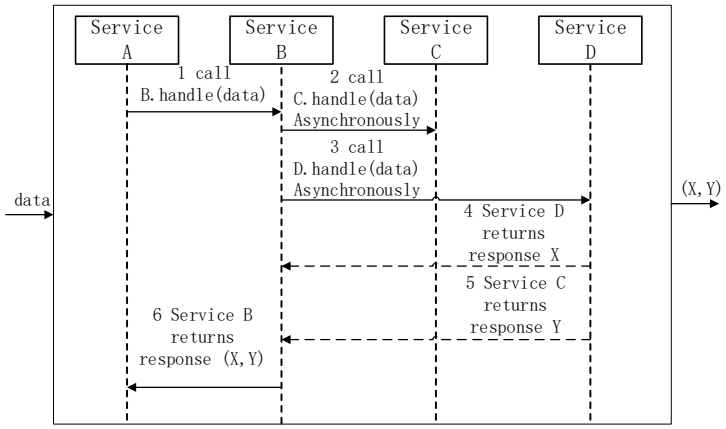

Figure 5: The service composition

Fig 5 illustrates a simple example of service composition，and the example consists of four services A, B, C and D.

Service A gets data as input, data can be represented by a combination of multiple basic data types (integer, floating point number) and Complex data types (array), for example: data = (integer, floating point, byte []); output (X, Y) can be a combination of two basic data types. After the client invokes the service composition A remotely (A .handle (data)), the internal invoking process of the service is as follows:

1) Service A calls B.handle (data);

2) Service B calls C.handle (data) asynchronously;

3) Service B calls D.handle (data) asynchronously;
4) Service C returns response X;
5) Service D return response Y;
6) Service B returns a response (X, Y) after both C and D return a response.
The algorithm used for building the microservice is as follows:

Algorithm for building microservices

1 : mS $\leftarrow$ *InitMicroServiceList*
2 : t_s
$\leftarrow The\ Threshold\ of\ Dependcy\ for\ Micro\ Services$
3 : $S \leftarrow FunctionalityList$
4 : $C \leftarrow CodeLibrary$
5 : $for\ s\ in\ S$
6 : $c \leftarrow codes\ that\ accomplish\ in\ C$
7 : $mS.add(c)$
8 : $end\ for$
9 : $for\ mS_i, mS_j\ in\ mS$
10 : $if\ Correlation(mS_i, mS_j) > t_s$
11 : $mS_i = merge(mS_i, mS_j)$
12 : $end\ if$
13 : $end\ for$

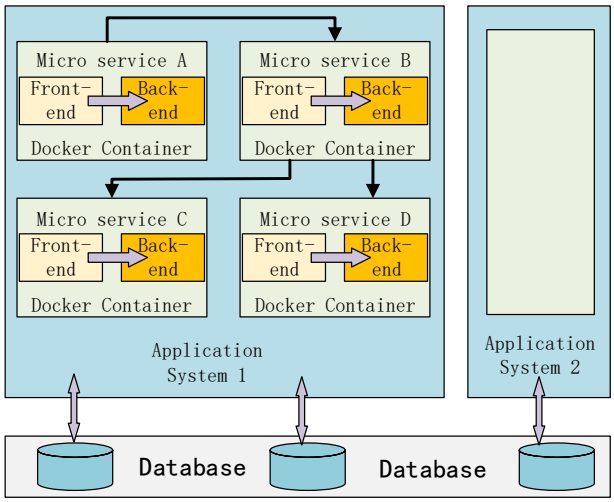

Figure 6: Microservice deployed in Docker

Fig. 6 shows the frame diagram of the microservices section after using Docker. These four microservices A, B, C, and D are independently deployed in the Docker container, the microservice A initiates a request to invoke the microservice B, and the microservice B asynchronously invokes the microservices C and D. This creates a complex and complete business process. We split a complex application system into multiple services with a single function and simple business logic. Each microservice is registered in Eureka Server, and microservices can be invoked through a declarative RESTful API.

4. Experimental Evaluations

Our experimental environment consists: Xeon processors, 16GB RAM, for microservice, the machine runs Docker 1.6 and Spring Cloud Dalston.SR5 on Linux 3.9, and each Container host runs separately.

In addition, we choose WSO2 product as Enterprise Service Bus environment. To test and compare the performance of all the three architectures. We measured that the time consumption for service includes (1) time used for requesting a service, (2) time consumed for receiving a response message (3) time used for parsing the service response message and description. In order to obtain accurate data, we launch multiple repeated invocation requests.

As mentioned before, we refactored the enterprise software architecture into uncoupled microservices using Spring Cloud Framework, and deployed them on servers through Docker technology. In this case study, we used four microservices as experimental conditions. The microservice A of the enterprise is responsible for inquiring about the cost and selling price of enterprise products. Microservice B is used for the statistics of the sales volume of related products. Microservice C product overview information, such as the profit of the quarter products. Microservice D specify the next quarter's production plan according to the profit/ loss information.

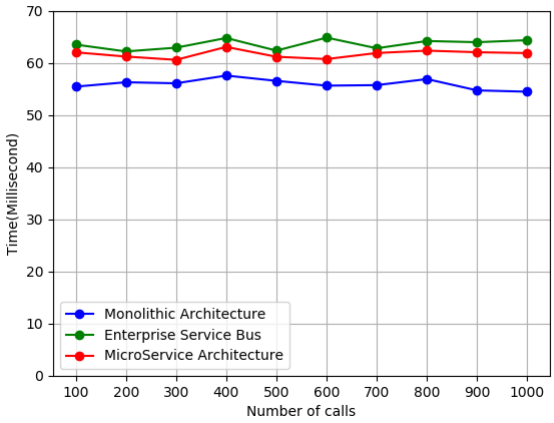

Figure 7: Time consumption of three scenarios

In order to get the benchmark performance from each of these programs, we run them repeatedly under conditions where the system is not loaded. Under the different number of repeated tests of the program, we got the experimental data. The result of performance test is shown as below. It is worth mentioning that the time consumption of microservice A represents the final performance of the whole microservice scheme.

Table 1: Average running time of microservices

Service	A	B	C	D
Average Time	61.24ms	50.56ms	20.41ms	17.68ms

Table 2: Average running time of monolithic architecture and ESB

Architecture	Monolithic	ESB
Average Time	56.25ms	63.42ms

By analyzing the data in Table 1 and 2, we conclude that the average time overhead of monolithic architecture is minimal due to the nearly zero internal communication cost, and its average running time is 63.24ms. As for the microservice architecture and ESB, the microservice architecture has an

average time loss of 54.25ms and is better than ESB's 65.42ms, which makes it very reasonable to use the microservice architecture to reconstruct enterprise business processes.

5. Conclusion

As cloud-computing technology evolves from concept to implementation, it becomes a new choice of providing deployment, providing software developers with flexible software construction methods on demand. In this deploying model, traditional integrated architecture or even SOA model cannot cope with much more frequent software updates and shorter delivery cycles, and the microservice architecture model that emerges with the development of Docker container technology can better response to the need for frequent delivery. Application platform built by Spring Cloud and Docker could fully demonstrates the advantages of the micro-service architecture, and implements component-oriented and service-oriented management of services, which enhances the continuous integration and expansion capabilities of services. As technology advances, microservices architecture systems will be adopted more, and microservices systems built based on Spring Cloud and Docker will be the best solution for microservices. Of course, with the advancement of technology, changes in ideas, the idea of microservice architecture still needs constant exploration and improvement.

6. Acknowledgement

This work is partially supported by National Natural Science Fund (61872138)
This work is partially supported by Natural science fund project in Hunan province (2016JJ2058).

7. References

[1] Fan, C. Y., & Ma, S. P. (2017). Migrating Monolithic Mobile Application to Microservice Architecture: An Experiment Report. IEEE International Conference on Ai & Mobile Services (pp.109-112). IEEE

[2] Li, S. (2017). Understanding Quality Attributes in Microservice Architecture. Asia-Pacific Software Engineering Conference Workshops (pp.9-10). IEEE Computer Society

[3] Wang, H., Zou, S., Lin, J., Feng, G., & Lv, H. (2016). A Dependable Service Path Searching Method in Distributed Virtualized Environment Using Adaptive Bonus-Penalty Micro-Canonical Annealing. IEEE, International Conference on Cyber Security and Cloud Computing (pp.530-539). IEEE.

[4] Celesti, A., Carnevale, L., Galletta, A., Fazio, M., & Villari, M. (2017). A Watchdog Service Making Container-Based Micro-services Reliable in IoT Clouds. IEEE, International Conference on Future Internet of Things and Cloud (pp.372-378). IEEE.

[5] Guidi, C., Lanese, I., Mazzara, M., & Montesi, F. (2017). Microservices: a language-based approach.

[6] Ma, S. P., Lin, H. J., Lan, C. W., Lee, W. T., & Hsu, M. J. (2018). Real World RESTful Service Composition: A Transformation-Annotation-Discovery Approach. IEEE, International Conference on Service-Oriented Computing and Applications (pp.1-8). IEEE.

[7] Wang, S., Urgaonkar, R., Chan, K., He, T., Zafer, M., & Leung, K. K. (2015). Dynamic service placement for mobile micro-clouds with predicted future costs. IEEE International Conference on Communications (Vol.28, pp.5504-5510). IEEE.

[8] Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., & Verano, M., et al. (2016). Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures. Ieee/acm International Symposium on Cluster, Cloud and Grid Computing (pp.179-182). IEEE.