Constructing a Service Software with Microservices

Feng-Jian Wang
Department of Computer Science
National Chiao Tung University
Hsinchu, Taiwan
fjwang@cs.nctu.edu.tw

Faisal Fahmi
EECS International Graduate Program
National Chiao Tung University
Hsinchu, Taiwan

Abstract— The microservice architecture is a variation of a Service-Oriented Architecture (SOA) that allows a service to be broken down into a number of smaller but independently concurrent running units so that both performance and maintainability of the application can get a big improvement. In this paper, we introduce an approach to constructing a software with layered and distributed microservices using object-oriented specifications. The proposed approach could increase the concurrency of the application system and thus improve the performance and could increase the reusability of microservices.

Keywords— Microservice, Systematic Approach, Object-Oriented Specification.

I. Introduction

SOA is one of the most popular software architecture. Services in SOA are frequently implemented as monolithic applications [15] on a single code and deployed as a single unit. Consequently, the service scalability and maintainability are inhibited since they cannot be performed in isolation and have large granularity, e.g., the change of a single service could affect the system entirely. In contrast to SOA, a microservice architecture allows a service to be divided into a number of smaller size but running concurrently. It enables the micro-service to be scaled and maintained independently in smaller granularity so that the performance [2, 3] and maintainability [10, 16] of the system can be given a big improvement.

In this paper, we present an approach to constructing a microservice-based software using object-oriented requirement specification [1], adopted due to the completeness and consistency of its requirements specification with object orientation. Using the approach, the system is decomposed into several subsystems recursively, and then define the microservice accordingly. After comparing with the existing approaches, the proposed approach could increase the concurrency of the system and thus could improve the performance and increase the reusability of the microservice with object orientation.

II. Microservices and Architecture Characteristic

Microservice can be defined as a programmable unit bounded by the contexts and can communicate with each other through message passing. Different microservices can be managed, deployed, and scaled independently in various platforms [10, 16], e.g., physical machines, virtual machines, or containers in different levels. Consequently, to improve performance and maintainability, the microservice architecture should have the following characteristics at least [2, 3]: bounded by context, smaller in size, and independence. For example, the characteristic of smaller size brings the major benefit in maintainability, e.g., it can be modified easily, and the independent microservice allows it to be scaled or modified without affecting others.

A distributed application can be comprised of distributed microservices of which each runs its own process(es). For example, consider an online system of product selling. It owns multiple functions such as creating an order, making a payment, …, etc. Each of them can be developed as a microservice, a distinct code to be deployed independently to improve the speed.

III. Constructing A Microservice-Based Software

In this section, our approach called MOOS is introduced. Using the approach, the system is decomposed into several subsystems recursively, where a subsystem provides interfaces, and its architectural style is similar to a microservice naturally. Thus, the details of MOOS are shown below:

Step 1: Describe each subsystem based on object-based analysis specification. The object-oriented requirements specification [1] include functional, such as use case, and nonfunctional, such as performance. A subsystem is described by encapsulating a set of related objects (e.g., in the same use case or subject) under its interfaces. Typical considerations to describe subsystems include [12]:

1. a subsystem can be developed independently, as long as the interfaces remain unchanged,
2. a subsystem can be deployed independently across a set of distributed computational nodes, and
3. a subsystem can be changed independently without breaking other parts of the systems.

Step 2: Map the subsystems to platforms. Each subsystem is mapped to a platform, which might contain more than one subsystems. A platform configuration includes the hardware (e.g., CPU and memory) and software (e.g., Jolie, JSP, etc.). The mapping is used to identify concurrency among subsystems and address their nonfunctional requirements, e.g., performance and reliability. The mapping considerations could include:

1. If two subsystems could run concurrently, each of them could be mapped to a distinct platform.
2. If a subsystem is given a special performance requirement, e.g., it needs to support 1000 concurrent users, it could be mapped to a distinct platform.
3. If a subsystem is given a special reliability requirement (e.g., it needs to apply circuit breaker [10, 16] to access external services), it could be mapped to a distinct platform.
4. If there are two related subsystems, the mapping consideration could include:
   a. In a tight-relation, e.g., they frequently communicate with each other, if one or both of them are not given special performance or reliability requirements, they could be mapped to the same platform to minimize the communication cost [8]. Otherwise, each of them could be mapped to a distinct platform.
   b. In a loose-relation, each of them could be mapped to a distinct platform.

Step 3: Identify global resources and create their access control mechanisms correspondingly. To identify the global resources, the objects that must be persistent are identified first.

978-1-5386-7374-4/18/$31.00 ©2018 IEEE
DOI 10.1109/SERVICES.2018.00035

With the requirements specification adopted in Step 1, the objects shared among users in different use cases could be identified. If these objects are included in different distributed subsystems, to simplify the maintenance of data consistency, they could be separated into a new subsystem as a *shared data* with access controls. The same rule is applied to the shared functions, to increase the reusability, these functions could be put into a new subsystem (and treated as *utility functions*).

Step 4: Describe the concurrency control of each subsystem. In distributed systems, a subsystem is need to deal with the external requests and respond in time. Since the external requests may be accepted concurrently, Jolie [9] is adopted to reduce the complexity of applying concurrency control for each subsystem. Jolie could perform the work either concurrently or sequentially, e.g., Jolie performs the concurrent works by supporting multithread mechanism [11], which allows each subsystem to serve multiple requests at the same time by creating a new thread for each request accepted at the point.

Step 5: Define and reuse microservices based on subsystems. A Layered Microservice Model (LMM) is proposed to reduce the management complexity of the derived microservices. An LMM includes *Domain*, *Shared*, and *Utility Microservice* Layers, where each of them contains the microservices related to the application domain, shared data, and utility functions respectively, shown in Fig. 1. The arrow in Fig. 1 indicates the allowed access between microservices in different layers, where microservices in the same layer could communicate each other.

Inside LMM, a strategy from [4, 14] is adopted to define five types of microservice, called *core*, *support*, *generic*, *shared*, and *utility microservice*. Thus, the detailed strategy includes:

1. Define core, support, and generic microservices in the application domain as the following steps sequentially:
   - Collect microservices whose corresponding subsystems have the highest corresponding priority in reliability and performance, and define them as core microservices.
   - Collect microservices that will be implemented by using open sources, not defined as core, and define them as generic microservices.
   - The rest microservices in the application domain are defined as support microservices.
2. Define *shared* and *utility* microservices based on shared data and utility functions derived in Step 3 respectively.

For example, in an online product selling system, a core could be the one that analyze the products recommended for each costumer immediately, a support could be the one that manages stock or sales, and a generic could be the one that manages employee or payroll.

Step 6: Check and update the system design. In this step, we check whether every requirement and design issue addressed contains any contradiction. If any error found, go back to the corresponding step, perform necessary updates, and continue.

IV. A Comparison with the Existing Approaches

The layered architecture could reduce the management complexity of the derived microservices. From the existing approaches [4, 6, 7, 10, 13, 16], there is currently one approach with the structural capability called Erl's approach [4]. The approach includes four layers namely *task service*, *microservice*, *entity service*, and *utility service* layers respectively. Using Erl's approach, microservices are used to implement services which need special requirement(s), e.g., runtime, deployment, ..., etc.

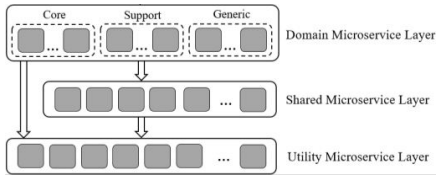

Fig. 1: A Layered Microservice Model (LMM)

After applying MOOS and Erl's approaches on the same study case adapted from [5], it is found that the deployment of Erl's approach may face with a bottleneck since all services, except microservices, may be deployed onto a single platform. Moreover, a microservice constructed with Erl's approach keeps for a special purpose only, and called as *non-agnostic*, it thus could reduce the reusability. Besides, LMM is composed by microservices only, MOOS could increase the concurrency of the system and reduce the occurrence possibility of a bottleneck. With generic microservices implemented for open sources, it could increase the reusability of the microservices.

V. Conclusions

In the paper, we introduce MOOS, a systematic approach to constructing a software with layered microservise using object-oriented specifications. Compared with the existing construction approaches, MOOS could increase the concurrency of the application system and thus improve the performance and reusability of microservices.

References

[1] B. Brueggle and A.H. Dutoit, Object-Oriented Software Enginering: Using UML, Pattrens, and Java, Prentice Hall, 2004.

[2] N. Dragoni, et al., "Microservices: How to Make Your Application Scale," Perspectives of System Informatics, vol. 10742, pp. 95-104, 2017.

[3] N. Dragoni, et al., "Microservices: Yesterday, Today, and Tomorrow," Present and Ulterior Software Engineering, pp. 195-216, 2017.

[4] T. Erl, Service-Oriented Architecture: Analysis and Design for Services and Microservices, Prentice Hall, 2016.

[5] K. Fakhroutdinov. "Online Shopping: UML Use Case Diagram Example," https://www.uml-diagrams.org/examples/online-shopping-use-case-diagram-example.html

[6] G. Granchelli, et al. "Towards Recovering the Software Architecture of Microservice-Based System," Proc. IEEE ICSAW, pp. 46-53, 2017.

[7] C. Guidi, et al., "Microservices: a language-based approach," Present and Ulterior Software Engineering, pp. 217-225, 2017.

[8] K. Huang and B. Shen, "Service deployment strategies for efficient execution of composite SaaS applications on cloud platform," Journal of Systems and Software, vol. 107, pp. 127-141, 2015.

[9] Jolie. "The first language for Microservices," http://www.jolie-lang.org/

[10] S. Newman, Building Microservice: Designing Fine-Grained Systems, O'Reilly Media, 2015.

[11] M. Papathomas, Concurrency in Object-Oriented Programming Languages, in Object-Oriented Software Composition, Prentice Hall, pp. 31-68, 1995.

[12] M. Richardson. "Guideline: Design Subsystem," http://www.michael-richardson.com/processes/rup_for_sqa/core.base_rup/guidances/guidelines/design_subsystem_B26FD609.html

[13] L. Sun, Y. Li, and R.A. Memon, "An open IoT framework based on microservices architecture," China Communications, vol. 14, pp. 154-162, 2017.

[14] V. Vernon, Implementing Domain-Driven Design, Pearson, 2013.

[15] Wikipedia. "Monolithic Application," https://en.wikipedia.org/wiki/Monolithic_application

[16] E. Wolff, Microservices: Flexible Software Architecture, Addison-Wesley, 2016.