An exploration of the evolving software architecture

Kun Wu
Hilaire Bouaddi
April 2021

1. Introduction

“Phoenix”, as a common metaphor, shows up frequently in software-related readings. The novel The Phoenix Project talks about how a collapsing project was brought back to life from the edge of breaking down. Martin Fowler also mentioned “Phoenix Server” and “Snowflake Server” lots of times in his interpretation of Continuous Delivery.

In any software engineering project, the development of a product is bound to meet unexpected failure as developers are prone to make mistakes, code could carry potential flaws, networks may be blocked and disconnected, etc. Just as the Murphy’s Law suggests, anything that can go wrong will go wrong. When designing large-scale distributed systems running simultaneously in massive server nodes, we need to face this curse squarely.

Various architecture styles spring up during the short history of software development. Recent years saw the increase in popularity of microservices as it simplifies the deployment, clarifies the business logic, adapts to the heterogeneous architecture, scales to large-scale nodes and so on. These advantages do help us to design better systems but most importantly, this trend reveals the essence of designing a stable system — individual services can fail and we need to replace them accordingly instead of the whole service.

In this essay, different architecture styles are summarized and compared to illustrate how they’re used to meet various requirements and addressing certain issues. Through a holistic view, we can see their connection and difference.

2. Primaeval distributed systems

The trial to build a large system through multiple independent distributed services showed up much earlier than people might believe. As the 1980s witnessed a trend from mainframe computers to minicomputers, one influential organization called OSF set up a whole set of regulations and reference implementations for distributed service components, called DCE(Distributed Computing Environment).

One important principle in setting these standards is transparency. A procedure call, resource access or data storage in a distributed environment should be the same whether it’s local or remote. Elegant and consistent with the UNIX philosophy, this ideal principle contains way too many technical complexities that can not be solved perfectly at that time. Various issues have to be taken into consideration, such as service discovery, load balance, network partition, serialization protocol, authentication, consistency over heterogeneous servers and so on.

Despite all the difficulty, DCE answered most of these questions from scratch by providing lots of relatively “transparent” distributed components and protocols.

3. Monolithic Application

Most software developers today have seen monolithic architecture themselves and taken its simplicity and efficiency for granted.

Vertically speaking, no modern information system in real production environments is not layered. Layered architecture, as a design pattern, is well accepted and adopted in almost all systems, let it be monolith or microservices. Through the vertical classification of modules, a request is delivered and handled in different forms until it reaches the database and a response is bounced back.

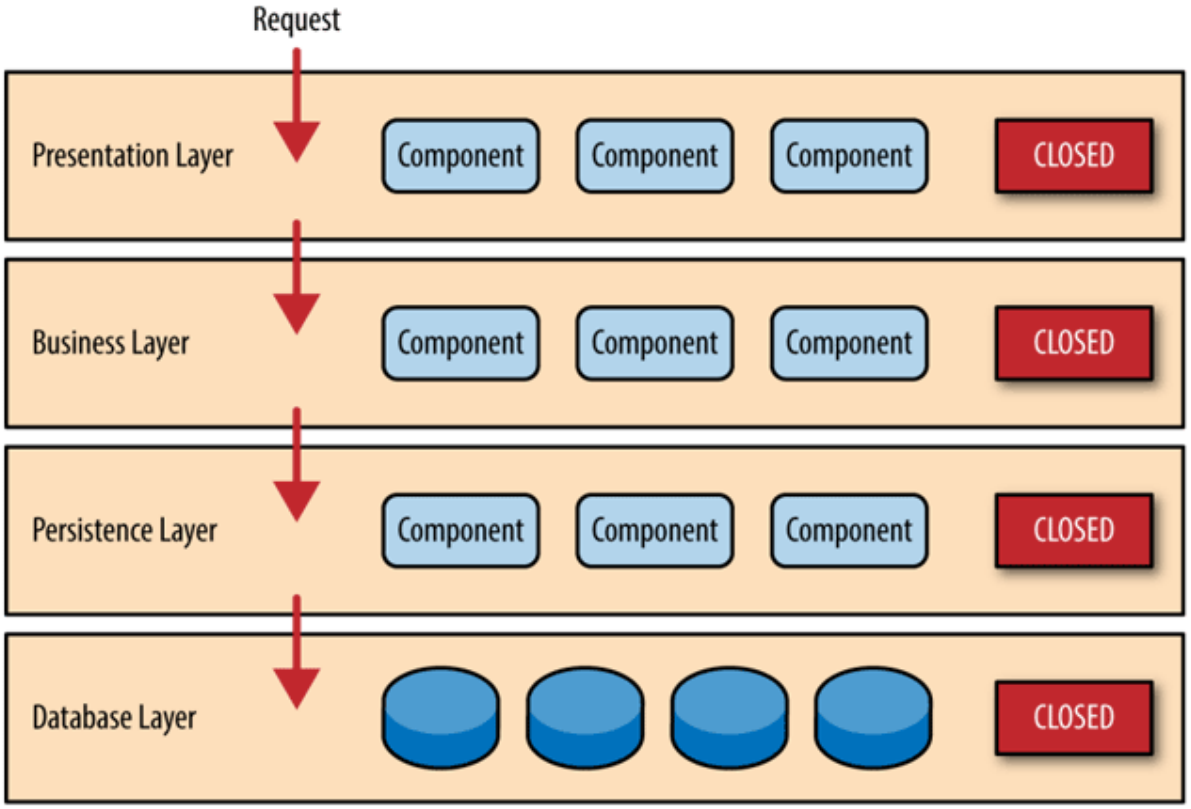

Figure 4-1. Request in a layered architecture, from《Software Architecture Patterns》

On a horizontal view, monolithic applications could be classified into modules according to different technique, functionality or responsibility. Such an application doesn't mean just one executable.

Despite that composability, it is often the lack of isolation and autonomy that drags monolithic applications down. If all parts of the codes are running in the same process space, then there is no need to worry about network partition or performance loss. With this great simplicity and efficiency, here entails the cost that if just a tiny part of the code goes wrong, the entire system breaks down. Issues like memory leaks, deadlock, endless loops will wreck the whole application. Overconsuming other common resources like port or database connection could even endanger all other copies running on the server.

In short, this architecture assumes that each gear of the system should be reliable everytime. This assumption may hold for small-scale applications but fail as the scale of an application grows. Accepting the fact that errors are inevitable is a big step forward as people experiment with different

ways to divide services into smaller independent parts running in different processes, one of which is Service-Oriented Architecture.

4. Service-Oriented Architecture

To divide a large system into several independent subsystems, developers have tried lots of different solutions, two of which are listed below.

Microkernel Architecture provides its core functionality through a kernel or core system and extends its business logic through plug-ins. In this way, shared data or services are gathered in one place while other functionalities are encapsulated and could be extended flexibly. As figure 5-1 shows, this architecture is very popular among desktop application or web services since customized development is quite convenient. However, the presumption of independence between different plug-in components in the application may not always be true. After all, in most usage scenarios, these components need to communicate and cooperate.

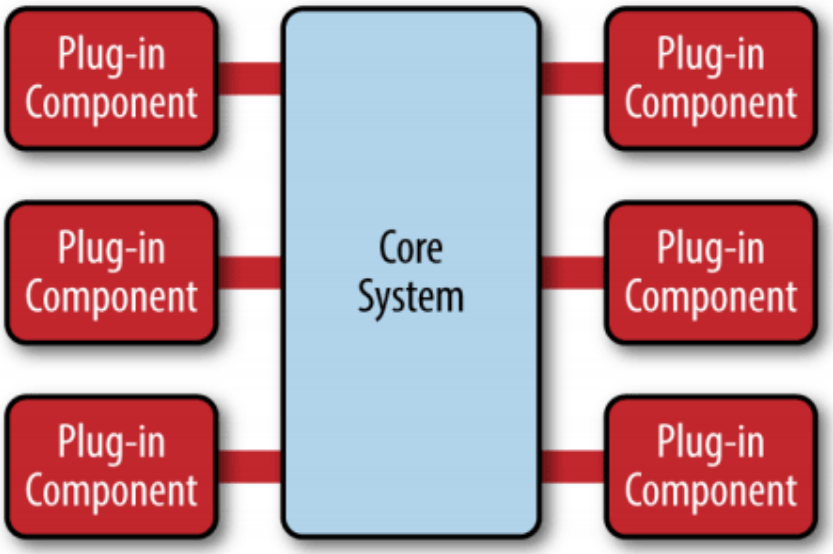

Figure 5-1 Illustration of Microkernel Architecture, from《Software Architecture Patterns》

Event-Driven Architecture, at the same time, describes another solution that subsystems communicate through a set of event queues. In this highly independent and decoupled way, events are sent to the event channel and every subsystem subscribes to their interested topics and handles events respectively.

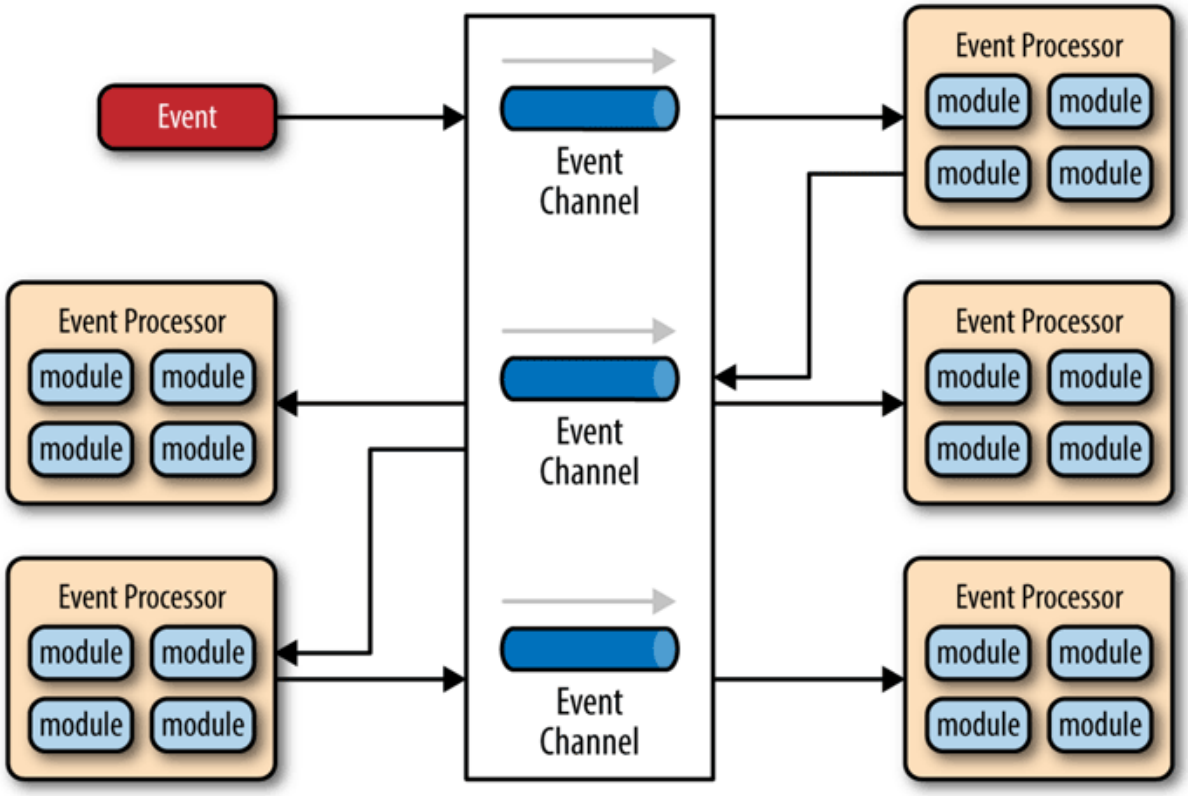

Figure 5-2 Illustration of Event-Driven Architecture, from《Software Architecture Patterns》

In the exploration of different architecture patterns, some familiar concepts in microservices like decoupling, registration, isolation, orchestration, and so on emerged. Around the 2000s, Service-oriented architecture (SOA) came to the stage. It represents a way of thinking in terms of services and service-based development and the outcomes of services. In other words, it allows users to combine large chunks of functionality to form applications that are built purely from existing services and combining them in an ad hoc manner.

5. Microservices

First put forward by Dr Peter Rodgers at Web Services Edge 2005 as Micro-Web-Service, microservices used to represent a kind of granular web services independent of programming languages. Later at 33rd Degree Conference, James Lewis delivered a speech《Microservices - Java, the Unix Way》, in which he talked about single responsibilities, Conway's law, automated scaling and addressed the Unix Philosophy of small and simple.

The real rise of microservices was from 2014 as《Microservices: A Definition of This New Architectural Term》written by Martin Fowler and James Lewis went viral. In that article, they defined today's microservices - an architectural style that structures an application as a collection of fine-grained and loosely coupled services. Such kinds of architecture allow different programming languages, various data storage techniques, light-weight communication mechanisms between services and automated maintenance. Apart from the definition, the article also lists several typical characteristics of microservices, such as Organized around Business Capability, Decentralized Governance, Design for Failure and Infrastructure Automation.

From the definition and characteristics mentioned above, we see that microservices architecture essentially follows the Unix philosophy of "Do one thing and do it well". It is common for microservices architectures to be adopted for cloud-native applications, serverless computing, and applications using lightweight container deployment. According to Fowler, because of the large number (when compared to monolithic application implementations) of services, decentralized continuous delivery and DevOps with holistic service monitoring are necessary to effectively develop, maintain, and operate such applications. A consequence of (and rationale for) following this approach is that the individual microservices can be individually scaled.

6. Cloud Native

Problems like service registration and discovery, load balance, transmission protocol have been existing ever since the primaeval distributed era. They are inevitable for any distributed system. Traditionally, they're addressed on the software layer. However, does it have to be like that?

Since the 2010s, Container techniques like Docker, have been contributing to building up a lightweight running environment for microservices. In 2017, as Kubernetes won the war of containers, it became obvious in the industry field that virtual infrastructure could be used in a distributed architecture. The boundary between software and hardware evaporates as a container for a single service is extended to a service cluster consisting of multiple containers. This flexibility of virtual hardware enables software to focus more on business logic and to leave other unrelated issues to infrastructure.

Despite the victory of the container war, Kubernetes still fails to provide a perfect solution for distribution problems. Imagine a case where microservice A calls two services B1 and B2 of microservice B, B1 behaves as expected while B2 returns errors all the time. To avoid a snowball effect, connection to B2 should be cut off after a certain number of failures. If we leave this problem to infrastructure, we're stuck in a dilemma that if the connection between A and B is cut off, then A will not be able to call B1 and that otherwise the failure from B2 consistently exists.

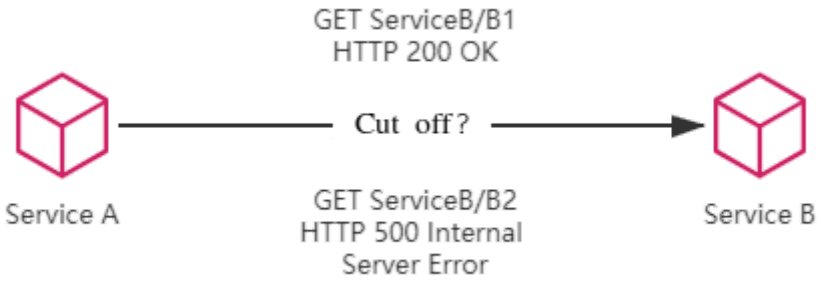

Figure 7.1 Example of possible failure handle

To handle these kinds of problems, virtual infrastructure introduced the sidecar pattern, namely Service Mesh. A service mesh is a configurable, low-latency infrastructure layer designed to handle a high volume of network-based interprocess communication among application infrastructure services using application programming interfaces (APIs). A service mesh ensures that communication among

containerized and often ephemeral application infrastructure services is fast, reliable, and secure. The mesh provides critical capabilities including service discovery, load balancing, encryption, observability, traceability, authentication and authorization, and support for the circuit breaker pattern.

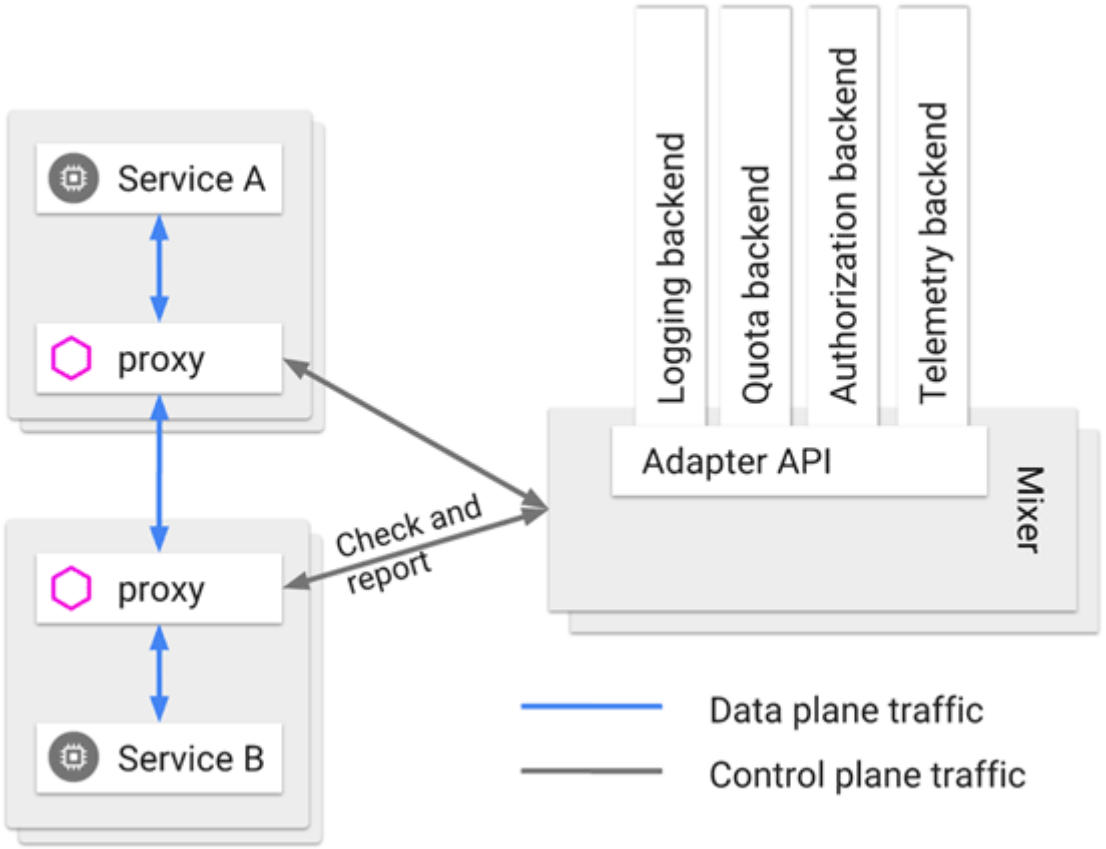

Figure 7.2 Illustration of sidecar pattern, from the document of Istio1.5

7. Serverless

Research on distributed architecture originated from the growing computing need beyond a single machine. Later on, fault tolerance, heterogeneity and function encapsulation also come into the minds of software architects.

In 2012, Iron.io first put forward the idea of "Serverless". Since the release of Lambda by Amazon in 2014, serverless compute service gets noticed and accepted by developers. Some Chinese companies like Alibaba, Tencent also followed this trend and released their counter products in 2018.

In the academic world, UC Berkeley predicted in 2009 the evolution and prevalence of cloud computing, which was verified over the last decade, in《Above the Clouds: A Berkeley View of Cloud Computing》. Ten years later in 2019, a second essay《Cloud Programming Simplified: A Berkeley View on Serverless Computing》was published. It predicts that serverless computing will grow to dominate the future of cloud computing.

Seeming promising, serverless computing may not become prevalent given some of its principles. Some applications like online games which require low latency and keep a long connection, serverless computing is inappropriate. This is because the actual scale of service is decided by the amount of time and memory it consumes.

As the last section of this essay, we want to point out that we are not archaeologists. We sort out this short history to understand the purpose and flaws of different architectures and to find solutions for issues we face today or in the future. After all, one greatest challenge of software development is to decide how to solve the problem with limited information given, as Alan Turing says - We can only see a short distance ahead, but we can see plenty there that needs to be done.

References

1. Jez Humble, and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*.
2. Lewis, James. “Microservices - Java, the Unix Way.” *http://2012.33degree.org/speaker/show/48*.
3. Eric Jonas, et al. “Cloud Programming Simplified: A Berkeley View on Serverless Computing.” https://arxiv.org/abs/1902.03383.
4. Michael Armbrust, et al. “Above the Clouds: A Berkeley View of Cloud Computing.” https://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf.
5. Fowler, Martin. “PhoenixServer.” https://martinfowler.com/bliki/PhoenixServer.html.
6. Gene Kim, and Kevin Behr. *The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win*.
7. Lewis, James. “Microservices - a definition of this new architectural term.” https://martinfowler.com/articles/microservices.html.
8. Amazon. “AWS Lambda.” https://aws.amazon.com/cn/lambda/.
9. Richards, Mark. *Software Architecture Patterns*. O'Reilly Media, Inc., February 2015, https://learning.oreilly.com/library/view/software-architecture-patterns/9781491971437/.
10. The Linux Foundation. Containers Are Not Lightweight VMs. url: https: //www.linuxfoundation.org/blog/2017/05/containers- are- notlightweight-vms/.
11. Docker Inc. What is a Container? url: https : / / www . docker . com / resources/what-container.