An overview of microservice architecture, recent trends and challenges

Aman Kumar Nirala
Reg. no. 23MCA0135
School of Computer Science Engineering, Vellore Institute of Technology, Vellore, India
amankumar.nirala2023@vitstudent.ac.in
Google-Scholar
ORCID

Abstract — Microservice architecture is one of the trending ways to build complex, highly scalable and high-performing IT infrastructure. This review article aims to provide a comprehensive overview of the microservice architecture, its evolution, needs, recent trends and problems related to it. It also discusses some of its implications in FaaS, server-less computing and technologies like Kubernetes. The article also aims to cover some design problems that come with such concurrent systems like race conditions, data inconsistency and deadlocks.

Keywords — *Microservice architecture, Concurrency, FaaS, Server-less computing, Kubernetes, Race Conditions, Cloud Functions*

I. INTRODUCTION

An architecture is the blueprint for any system under which it works, evolves and communicates with its components. The architecture plays an important role in the system's performance, quality, maintainability and overall success (1). Recently, microservice architecture has come up as a design and architecture of choice for modern software development (2). This development however took a lot of innovation and time (3). Before starting with microservice architecture, discussing other design paradigms like Monolithic Architecture would provide a better understanding and need for microservice architecture.

II. MONOLITHIC ARCHITECTURE

A monolith (a system that follows the monolithic architecture) is a system whose components are encapsulated and tightly coupled. The elements of a monolith cannot be executed independently. The mainstream server-side development languages like JAVA, C/C++, and Python are designed to create single executable artefacts, making deploying a monolith very easy (3, 4). A monolithic architecture has its disadvantages which are discussed in section 2.1. Monoliths are still widely used as many high business value services and applications today still follow this architecture due to its ease of deployment and ability to program small and simple tasks. Choosing this architecture for starting a software project is considered a good option as it allows for exploring the system's complexity and boundaries (2, 4, 5, 7). A diagrammatic representation of a monolithic system is represented in *Figure 1*.

2.1 Problems with monolithic architecture (3, 4)

This section provides a comprehensive list of the disadvantages that come with monolithic architecture that lead to the development of microservice architecture.

1. Monoliths are difficult to maintain as even a small change can lead to chaotic behaviour in the system and tracking down bugs will need lots of testing and changes to be made throughout the system.
2. Monoliths suffer from "dependency hell" i.e. changing or updating a dependency can lead to incompatibility with other dependencies, changes in the behaviour of the program or unable to compile the code completely.
3. Monoliths experience higher downtime as any maintenance or upgrades will need the whole application to be shut down.
4. Developers usually face a technology lock-in when using this architecture as all the elements of the system will have to be built and maintained in the language and framework used initially.
5. Monoliths are difficult to scale. To cater to more clients monoliths will have to be redeployed completely even in situations where only a few services are in demand. This adds extra operations expenses for the service provider.

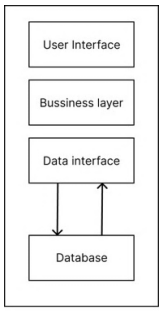

Figure 1: A diagrammatic representation of a monolithic software system

III. MICROSERVICE ARCHITECTURE (MSA)

Microservice architecture provides a design where the whole system is divided into small loosely connected systems called **microservices** that interacts with each other through message passing on request (1, 3, 4, 6, 7). They are built on the principles of **single responsibility**, **loose coupling**, **scalability**, **deploy-ability** and **independent execution** which are not the characteristics of monolithic systems (7). A very intuitive example of a graph plotting system using microservice architecture is provided in (3) where a user inputs to a microservice "Plotter" which in response passes the message/data to another microservice "Calculator" which is responsible for calculating the shape of the graph according to the provided function and returns it to the "Plotter", the "Plotter" then calls for another microservice "Display" which is responsible for displaying the received plot to the user. A diagrammatic representation of the same is represented in *Figure 2*. It's interesting to see that all these microservices are responsible for individual tasks and still coordinate via message passing as a system to provide meaningful service. One of the benefits of using MSA is the ability to use different technologies and deployment environments of each microservice according to their needs and demands. This architecture opens new doors for paradigms like distributed computing, cloud computing, clustered computing and FaaS (a product of the prior mentioned paradigms).

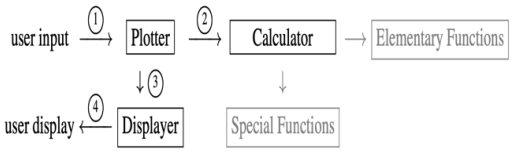

Figure 2: An example of a system that follows MSA (3)

3.1 Communication between microservices

A very intriguing question to ask is how microservices communicate with each other even though they can be deployed in totally different environments and developed different frameworks and standards. To approach this problem I will first discuss how the elements of a monolithic system communicate. In a monolithic application, the elements use language-level methods or procedure calls to communicate with other elements (8). Since, the elements of a monolithic system share the same environment, memory space and processor, the communicating is faster and secured (assuming that the integrity of the backend machine is intact). In contrast to the monolithic architecture, microservice-based systems are distributed systems as mentioned earlier, the microservices can be deployed on multiple machines in different environments. One can look at each of those instances running a microservice as a process which needs some IPC to work together.

3.1.1 IPC Interaction Styles

In this section, I will discuss some different ways these processes can communicate. Any inter-process communication can be categorized along two dimensions. The first dimension is the type of connection as mentioned below:

- **One-to-One –** Client/Service requests are catered by only one service
- **One-to-Many –** Client/Service requests are processed by multiple processes

The second dimension is the nature of interaction as mentioned below:

- **Synchronous –** Client/Service receives a periodic response from the service and the client might halt while waiting for the response
- **Asynchronous –** Client/Service doesn't block while waiting for a response and responses are not sent immediately.

Table 1 represents different kinds of IPC interaction styles. A brief description of those are mentioned below:

- **Request/Response –** A client sends a request to a service and waits for a response.
- **Notification –** A client sends a request to a service but no response is expected.
- **Request/Async response –** A client sends a request to a service and doesn't halt for a response. It assumes that the response might be delayed.
- **Publish/Subscribe –** A client broadcasts a message which is consumed by subscribed services.
- **Publish/Async responses –** A client broadcasts a message and waits for certain services to respond.

	One-to-One	**One-to-Many**
Synchronous	Request/Response	–
Asynchronous	Notification	Publish/Subscribe
	Request/Async response	Publish/Async responses

Table 1: Interaction styles (8)

3.1.2 IPC Technologies

A number of choices are available when it comes to different types of IPC implementations but two of the broadly classified technologies are **Asynchronous Message-based Communication** and **Synchronous-Response/Request IPC**. In message-based communication, a **message** is the data that is transmitted through a medium or channel between services. A message has headers (containing various metadata like the sender's details, protocols and standards involved and even authentication tokens) and a message body. Some of the open-source messaging systems (systems that facilitate message composition and transmission through the implementation of various protocols and standards) are RabbitMQ, Apache Kafka, Apache ActiveMQ, and NSQ. The message-based IPC add extra complexity as the whole messaging system is yet another element that becomes part of the system and adds extra dependency. On the other side in a response/request IPC, the client sends a request for services and waits for a response. This request could be made from a synchronous process i.e. blocking the thread that is making the request or an asynchronous process i.e. not blocking the thread while waiting for a response. Two of the important protocols used are **REST** and **Thrift**. REST is a design to implement API end-points that can be accessed through HTTP requests. Thrift is an alternative to REST. Apache Thrift is a framework used to write cross-language remote-procedure call clients and servers. The details and implementation of RESTful APIs and Thrift are out of the scope of this article.

3.2 Deployment and Orchestration

Deploying a microservice system and ensuring the integrity and health of all the components can be a very challenging task. Before the introduction of technologies used for the deployment of microservices, I will first define some basic entities and terms which will be used throughout the sections.

Definition 1: **Cloud Computing –** Cloud computing is a kind of computing technique where computing services are provided by low-cost computing units connected via IP networks (9). This definition however may vary as there seems to be no agreement on what is cloud computing and what its scope is as it's expanding continuously (9, 10).

Definition 2: **Container –** Containers are entities that contain the execution environment, VMs, OS, physical hardware everything encapsulated that is needed for the microservice to execute (11).

Definition 3: **POD –** A Pod is a unit computing entity that is copied multiple times to achieve scalability. A Pod can have multiple containers (12, 13).

Definition 4: **Cluster –** A Cluster is a collection of resources like storage, network, and computing systems that are used to distribute workload (12).

Definition 5: **Node –** A Node is a single host that can be a physical machine or a VM. Nodes are responsible for executing pods (12).

Definition 6: **Master –** A master is a special node that is responsible for handling other nodes and their resources (12).

The distributed nature of MSA requires it to be deployed using cloud computing. There are multiple cloud service providers

today which provide a substantial and reliable infrastructure for the deployment of an MSA-based application. Some of the noted and important cloud service providers are Google Cloud Service (GCP), Amazon Web Service(AWS), Microsoft Azure, etc. There are multiple strategies for deploying these services according to the design and need.

- **Design –** Breaking the application into microservice by following the principles mentioned in Section III. Choose a proper IPC design as discussed in section 3.1.
- **Containerization –** Technologies like **Docker** can be used to containerise the application environment which provides a level of virtualization, isolation and consistency between microservices and their deployment. These containers can be managed using services like **Kubernetes** which is discussed in Section 3.3.
- **Cloud Provider Selection –** Choosing a CSP according to the need is one of the crucial steps of deploying a microservice. Some important aspects to look into are pricing, the region the services are catered and the ecosystem.
- **Orchestration –** Orchestration refers to services like logging, scaling, cost and resource optimisation and error handling. Most of the cloud services provide built-in orchestration services while Kubernetes can be used to manage clusters of Pods. Services like AWS EKS, and GCP GKE can be used to deploy Kubernetes clusters.

3.3 Kubernetes

Kubernetes, an open-source container orchestration platform, oversees the deployment, monitoring, and scaling of containerized applications across numerous hosts. Initially developed by Google, it was subsequently contributed to the Cloud Native Computing Foundation (CNCF) in March 2016 (13). Kubernetes allows the user to declare states of application using concepts of "deployments" and "services" (13). For example, a user can specify they need to deploy 5 instances of Python3 with Flask. Kubernetes will continuously monitor and manage the instances and maintain the integrity of their states. A detailed definition of Kubernetes architecture is given in (13) which is represented in *Figure 3.*

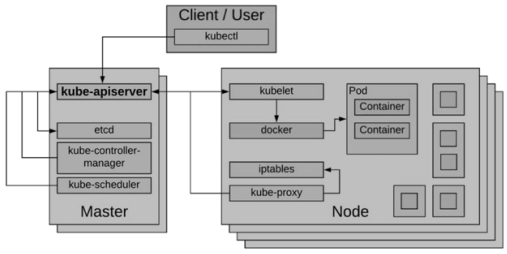

Figure 3: A diagrammatic representation of Kubernetes architecture (13)

The Master in the architecture controls and manages the cluster. The kube-apiserver is used to modify the resources running in the cluster. The etcd is a distributed key-value store used to store cluster states. The kube-controller-manager is a monitoring system that looks into the elasticity of the components. The kube-scheduler is used to schedule pod initialization based on node configuration. More details on the configuration and deployment of Kubernetes clusters can be referred to in (13). Docker plays an important role in the deployment of Kubernetes clusters. The nodes in the cluster have a docker container which is like a stripped-down OS with only packages installed that are needed for the execution of the microservice application.

IV. SERVERLESS ARCHITECTURE

Serverless architecture is a product of cloud computing and an amalgamation of two types of services i.e. **BaaS (Backend as a Service)** and **FaaS (Function as a Service)** (1). In serverless architecture, the developer doesn't need to bother about setting up and maintenance of the backend environment and hardware. All they need to do is use a cloud service like Firebase from GCP that provides a ready-to-use infrastructure for the deployment of services and an interface/API to interact with it. These same services provide a way to deploy simple lightweight endpoints without setting up any servers, network, port configs, etc. These are called FaaS, some examples are Cloud Function by Firebase(GCP), and AWS Lambda. These FaaS services can be used for RESTful IPC which I discussed in section 3.1.2. *Figure 4* shows an image of the Cloud Functions Dashboard on Firebase.

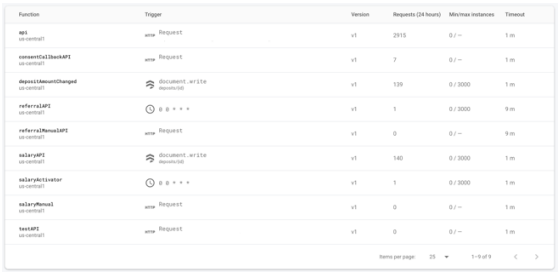

Figure 4: Cloud Function from Firebase Dashboard

V. PROBLEMS WITH CONCURRENCY

The MSA comes with a feature that allows developers to make use of parallel computing. Multiple nodes on a cluster can be used as processing units that can work together to solve a single problem. Such systems face the issues of concurrency. For example, let's say I have two services "Alice" and "Bob" which share the same source of truth i.e. a database which stores a variable "Count". The instruction I am trying to execute requires "Alice" to first access the value from the variable "Count" and then increment the value by 1. Post this I want "Bob" to access "Count" and multiply the value by 2. At the end of the instruction, I read the value of "Count" as the output.

```
ALICE ( COUNT ) :
        COUNT = COUNT + 1
END


BOB ( COUNT ) :
        COUNT = COUNT * 2
END


START:
        COUNT = 5

        THREAD 1 :
                ALICE(COUNT)
        END

        THREAD 2 :
                BOB(COUNT)
        END
END
```

Figure 5: Instruction for Alice and Bob

In *Figure 5*, you can see the pseudocode for the same. But here the method calls of Alice and Bob are taking place concurrently. The expected output is first Alice increments the value by **1** i.e. $\mathbf{5+1=6}$ and then Bob multiplies it by 2 i.e. $\mathbf{6*2=12}$, resulting in the value being **12**. But let's assume that for some reason the thread or process that is responsible for running Alice got delayed or suspended, the process responsible for executing Bob will get the preference and the whole condition will change as first Bob multiplies Count by **2** i.e. $\mathbf{5*2=10}$ and then Alice increments the value by **1** i.e. $\mathbf{10+1=1}$, resulting Count into **11** which is not what was expected. This whole situation is termed as **Race Condition** (14 − 16). Many techniques and methods to avoid such conditions are discussed in (14, 15). Discussing that will be out of the scope of this article as it's better to discuss them under concurrent systems and not just MSA. Another unwanted situation these systems can get into is referred to as Deadlock. **Deadlock** is a situation where elements of a situation halt in a situation where they wait for a condition to fulfil which never does (17). Generally, it is seen with elements that share a circular conditional dependency. This deadlock situation persists until some special action is taken from the outside. A very intuitive example of such a situation is provided in (17). A very elegant solution to the problem of race conditions and deadlocks using Communication Sequential Processes is provided in (14).

VI. CONCLUSION

In the article, I discussed and explored different architectures which are used for the development of any IT infrastructure. I briefly described the Monolithic architecture and the problems associated with it that led to the development of the MSA. I discussed the core principle of MSA i.e. **single responsibility**, **loose coupling**, **scalability**, **deploy-ability** and **independent execution.** I have also provided a detailed example of how an MSA works. I have then provided detailed information on how microservices communicate using different IPC styles and technologies like REST and Thrift. I have also discussed how a microservice is deployed and what role Kubernetes and Docker play in the modern tech stack. I have then briefly discussed what serverless architecture is and how MSA lead to the development of new paradigms like BaaS and FaaS. Lastly, I have discussed the problems and limitations like race conditions and deadlocks related to such concurrent systems. MSA has been a revolutionary design that led to the development of modern IT infrastructure and brought a generational innovation of the Internet.

VII. REFERENCES

1. Jaiswal, Manishaben, Software Architecture and Software Design (November 5, 2019). International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056, p-ISSN: 2395-0072, Volume: 06 Issue: 11, s. no -303 , pp. 2452-2454 , Nov 2019 Available at: https://www.irjet.net/archives/V6/i11/IRJET-V6I11303.pdf, Available at SSRN: https://ssrn.com/abstract=3772387 or http://dx.doi.org/10.2139/ssrn.3772387

2. Selmadji A, Seriai AD, Bouziane HL, Oumarou Mahamane R, Zaragoza P, Dony C. From Monolithic Architecture Style to Microservice one Based on a Semi-Automatic Approach. 2020 IEEE International Conference on Software Architecture (ICSA). 2020 Mar;

3. Dragoni N, Giallorenzo S, Lafuente A, Mazzara M, Montesi F, Mustafin R, et al. Microservices: yesterday, today, and tomorrow [Internet]. 2017 [cited 2023 Sep 20]. Available from: https://arxiv.org/pdf/1606.04036.pdf

4. 1. F. Ponce, G. Márquez and H. Astudillo, "Migrating from monolithic architecture to microservices: A Rapid Review," 2019 38th International Conference of the Chilean Computer Science Society (SCCC), Concepcion, Chile, 2019, pp. 1-7, doi: 10.1109/SCCC49216.2019.8966423.

5. Gos K, Zabierowski W. The Comparison of Microservice and Monolithic Architecture. 2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH). 2020 Apr;

6. Ramu VB. Performance Impact of Microservices Architecture. The Review of Contemporary Scientific and Academic Studies. 2023;3(6).

7. Abgaz Y, McCarren A, Elger P, Solan D, Lapuz N, Bivol M, Jackson G, Yilmaz M, Buckley J, Clarke P. Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review. IEEE Transactions on Software Engineering. 2023 Jun 23.

8. Richardson C. Building Microservices: Inter-Process Communication [Internet]. NGINX. 2015. Available from: https://www.nginx.com/blog/building-microservices-inter-process-communication

9. Qian L, Luo Z, Du Y, Guo L. Cloud computing: An overview. InCloud Computing: First International Conference, CloudCom 2009, Beijing, China, December 1-4, 2009. Proceedings 1 2009 (pp. 626-631). Springer Berlin Heidelberg.

10. Kim W. Cloud computing: Today and tomorrow. J. Object Technol.. 2009 Jan 2;8(1):65-72.

11. Turin G, Borgarelli A, Donetti S, Johnsen EB, Tapia Tarifa SL, Damiani F. A formal model of the kubernetes container framework. InInternational Symposium on Leveraging Applications of Formal Methods 2020 Oct 20 (pp. 558-577). Cham: Springer International Publishing.

12. Sayfan G. Mastering kubernetes. Packt Publishing Ltd; 2017 May 25.

13. Kubernetes T. Kubernetes. Kubernetes. Retrieved May. 2019;24:2019.

14. Martin JM. Designing and Verifying Microservices Using CSP. In2021 IEEE Concurrent Processes Architectures and Embedded Systems Virtual Conference (COPA) 2021 Apr 25 (pp. 1-4). IEEE.

15. Beckman NE. A survey of methods for preventing race conditions. May. 2006 May 10;10:1-20.

16. Bruce KB, Danyluk A, Murtagh T. Introducing concurrency in CS 1. InProceedings of the 41st ACM technical symposium on Computer science education 2010 Mar 10 (pp. 224-228).

17. Holt RC. Some deadlock properties of computer systems. ACM Computing Surveys (CSUR). 1972 Sep 1;4(3):179-96.