REST vs. Messaging for Microservices

Discover how to choose the right communication style for Microservices

Edited by **SANGEETHA MANJUNATH**

Microservices enable developers to develop applications that provide consistent user experiences across a range of platforms like web, mobile, IoT, wearable and fitness trackers. Microservices are autonomous by allowing code and state to be independently developed, versioned, deployed, and scaled. The popularity of microservices is that they can solve many current IT challenges such as increasing speed, quick deployment, scalability of applications and rapid test processes.

While identifying and designing microservices, it is highly essential to ensure that the services are as small as possible so that each microservice can be Continuously Integrated (CI) and Continuously Delivered (CD) for Deployment. The proper understanding of microservices is necessary for planning, documentation and testing in order to achieve the desired results.

REST Vs Messaging for Microservices

A microservices architecture is a well-established approach to build complex systems composed of loosely coupled modules. It has gained significant traction as one of the most prominent software architecture trends in recent years. The concept behind it is surprisingly straightforward: break down a large, interconnected system into multiple small, lightweight modules, which simplifies software management.

However, a crucial question arises once the monolithic application is divided into these smaller modules — how should they be effectively interconnected? While there isn't a definitive answer to this question, several approaches can be considered based on the specific application and use case. Two common protocols used in microservices are HTTP request/response with resource APIs and lightweight asynchronous messaging when communicating updates across several microservices. Let's explore these protocols.

Two common protocols used in microservices are HTTP request/response with resource APIs and lightweight asynchronous messaging when communicating updates across several microservices. This way the small, lightweight modules in MSA architecture can achieve the business domain process. MSA is the established pattern which can make software management easier with the agile development, fast delivery, highly scalable and maintain high availability.

Sunil Kumar A R
Technical Director
ar.sunilkumar@nic.in

Types of Communication

Microservices can communicate through many different modes of communication, each targeting a different use case. These types of communications can be primarily classified in two dimensions. The first dimension defines if the communication protocol is synchronous or asynchronous. The second dimension defines if the communication has a single receiver or multiple receivers. Refer to Table 11.1 and 11.2 to understand key difference between all these dimensions.

The most common type of communication between microservices is single-receiver communication with a synchronous protocol like HTTP/HTTPS when invoking a REST API. Microservices typically use messaging protocols for asynchronous communication between microservices. This asynchronous communication may involve a single receiver or multiple receivers depending on the application's needs.

Representational State Transfer

Representational State Transfer (REST) is a popular architectural style for request and response communication, and it can serve as a good example for the synchronous communication type. This is based on the HTTP protocol, embracing verbs such as GET, POST, PUT, DELETE, etc. In this communication pattern, the caller waits for a response from the server.

REST is the most commonly used architectural style for communication between services, but heavy reliance on this type of communication has some negative consequences when it comes to Microservices Architecture (MSA). The disadvantages include: 1. Multiple round trips (latency), 2. Blocking and Tight Coupling.

Asynchronous Messaging

Messaging is widely used in a microservices architecture, which follows the asynchronous protocol. In this pattern, a service sends a message

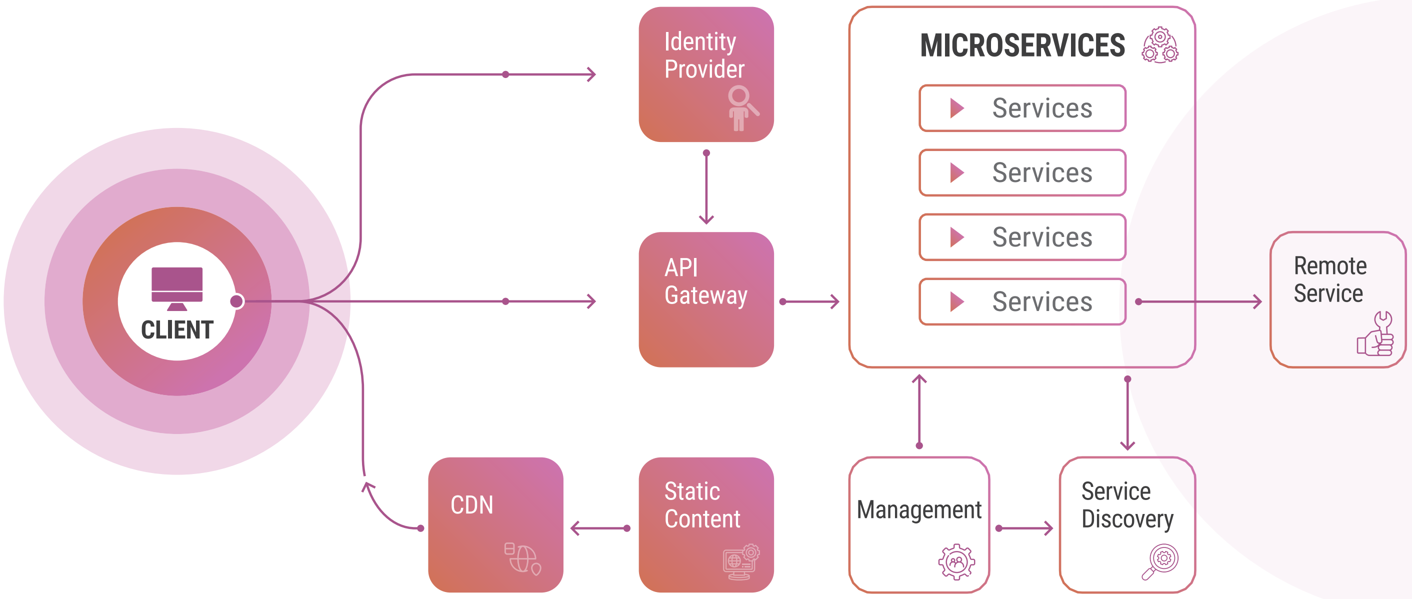


Fig 11.1 Architecture of Microservices

without waiting for a response, and one or more services process the message asynchronously. Asynchronous messaging provides many benefits but also brings challenges such as idempotency, message ordering, poison message handling, and complexity of message broker, which must be highly available. It is important to note the difference between asynchronous I/O and the asynchronous protocol.

Asynchronous I/O means that the calling thread is not blocked while the I/O operations are executed. This is an implementation detail in terms of the software design. It also means that the sender does not need to wait for a response.

Asynchronous messaging has matured into a number of messaging patterns. These patterns apply to scenarios when several parts of a distributed system must communicate with one another in a dependable and scalable way. Let's take a look at some of these patterns.

Pub/Sub Pattern

The pub/sub pattern implies that a publisher sends a message to a channel on a message broker. One or more subscribers subscribe to the channel and receive messages from the channel in an asynchronous manner. This pattern is useful when a microservice needs to broadcast information to a significant number of consumers.

Advantages

- Decouples Publishers and Subscribers
- Increases Scalability
- Improves Responsiveness
- Separation of Concerns

Disadvantages

- High Semantic Coupling
- Difficult to Gauge the Health
- Becomes a Bottleneck for scaling

Queue-Based Pattern

In the queue-based pattern, a sender posts a message to a queue containing the data required by the receiver. The queue acts as a buffer, storing the message until it is retrieved by the receiver. The receiver retrieves messages from the queue and processes them at its own pace.

This pattern is useful for any application that uses services that are subject to overloading.

Advantages

- Maximize Scalability
- Maximize Availability

Disadvantages

- No Longer Available after receipt
- Operational Complexity

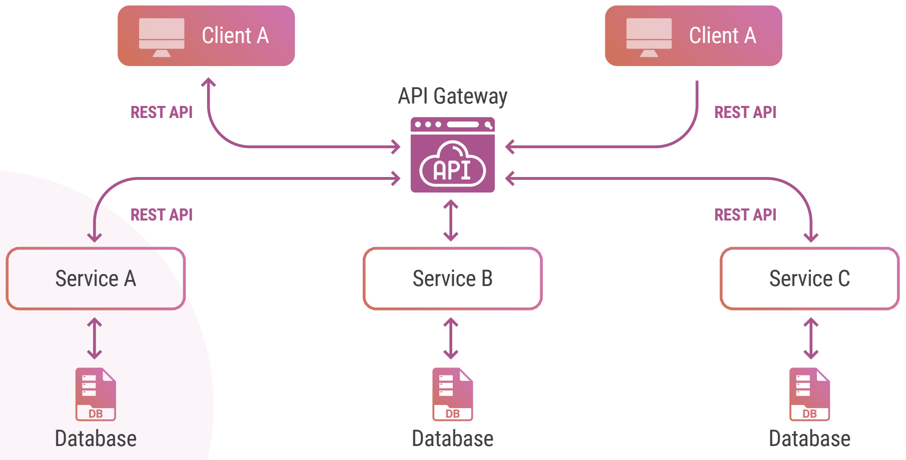


Fig 11.2 Rest API-based Communication

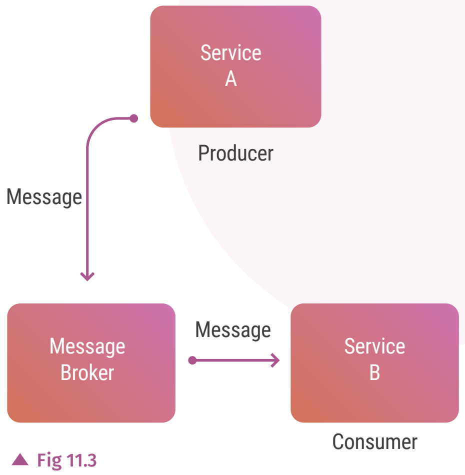
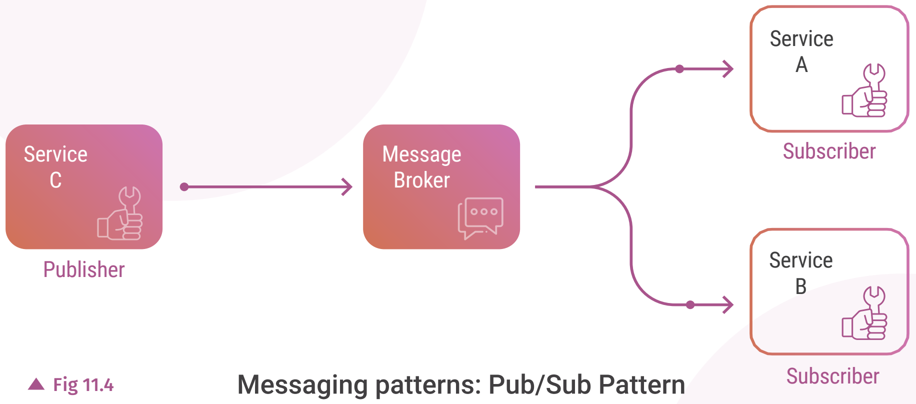


Fig 11.3 Messaging-based communication

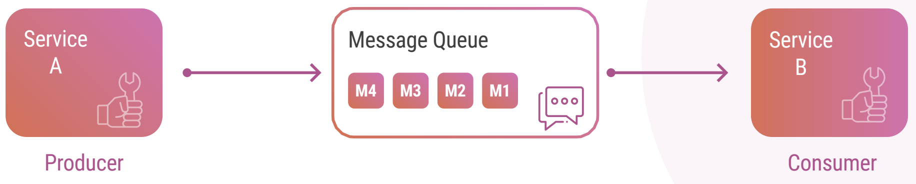
Keys to Streamlined Messaging Infrastructure

Asynchronous communication is usually managed through a message broker. There are some factors to consider when choosing the right messaging infrastructure:

- **Scalability –** ability to scale automatically when there is a load surge on message broker
- **Data persistency –** ability to recover messages in case of reboot or failure
- **Consumer capability –** whether the broker can manage one-to-one and/or one-to-many consumers
- **Monitoring –** whether monitoring capabilities are available
- **Push and pull queue –** ability to handle push and pull delivery by message queues
- **Security –** proper authentication and authorization for messaging queues and topics
- **Automatic failover –** ability to connect to a failover broker automatically when one broker fails without impacting publisher / consumer



▲ Fig 11.4 Messaging patterns: Pub/Sub Pattern



▲ Fig 11.5 Messaging patterns: Queue-Based Pattern

Synchronous Vs. Asynchronous Communication

What are the various concerns with respect to the first dimension: synchronous or asynchronous are tabled below.

	Communication Pattern	Protocols	Coupling	Failure Isolation
Synchronous	The client sends a request and waits for a response from the server.	HTTP/ HTTPS	The client code can only continue its task further when it receives the server response.	It requires the downstream server to be available or the request fails.
Asynchronous	Communication is not in sync, which means it does not happen in real time.	AMQP, MQTT	In the context of distributed messaging, coupling implies that request processing will occur at an arbitrary point in time	If the consumer fails, the sender can still send messages. The messages will be picked up when the consumer recovers.

▲ Table 11.1

Communication via Single Vs. Multiple Receivers

With respect to the communication receiver's instances the concerns related are tabled below.

	Communication Pattern	Use Case
Single Receiver	It implies that there is point-to-point communication that delivers a message to exactly one consumer that is reading from the channel, and that the message is processed only once.	It is well-suited for sending asynchronous commands from one microservice to another.
Multiple Receivers	Communication from the sender is available to multiple receivers.	The publish/subscribe mechanism is where a publisher publishes a message to a channel and the channel can be subscribed by multiple subscribers/receivers to receive the message asynchronously.

▲ Table 11.2

Conclusion

Microservices are becoming the de facto approach for designing scalable and resilient systems. It cannot be defined discretely the best and suitable approach for communications among the microservices.

Restful APIs provide a request-response model for communication between services, whereas asynchronous messaging offers a more scalable producer-consumer relationship among different services. Both messaging and REST APIs can be utilized for communication between microservices.

Messaging architectures, in particular, are highly beneficial for enhancing agility and facilitating rapid development. They are commonly employed among the modern applications that employ microservices or any application featuring decoupled or distributed components.

When selecting the appropriate communication style for microservices, it is crucial to ensure a harmonious alignment between the requirements of the consumer and one or more communication types. This alignment guarantees the provision of a robust interface for the services.

Contact for more details

Sunil Kumar A R
Technical Director
NIC Centre of Excellence on Microservices
A-Block, 3rd Floor, Kendriya Bhavan, CSEZ P.O, Kochi
Kerala - 682037
Email: ar.sunilkumar@nic.in, Phone: 0484-2423769