

Microservices communication is the heartbeat of any microservices architecture and designing them could make or break the system.

There are just 3 ways of communication between the services-

* **Synchronous Communication**
* **Asynchronous Communication**
* **Hybrid Communication**

The article went quite long, and I decided to split it up into 2 parts. In the first part of this article, we will see the Synchronous Communication between microservices, and we discuss different protocols like REST, **gRPC**, and **GraphQL**. Then we will see the important aspects to be considered when designing the synchronous communication between microservices. Let's get started.

**What is Microservices Communication?**

During Monolith days, communication means connecting our application with **external systems**like payment gateways. Ever since the [microservices architecture](https://medium.com/javarevisited/50-microservices-interview-questions-for-java-programmers-70a4a68c4349) evolved, the services got distributed and communication between them forms the heartbeat of the whole architecture. To add to the complexity, every microservice has **multiple instances**running, and making them talk to the right instance of the right service to get the job done is the most important aspect of any [microservices architecture](https://medium.com/javarevisited/8-best-online-courses-to-learn-service-oriented-soa-and-microservices-architecture-94c01d6b94e6). Let us look at the various ways of communication between the microservices, the important aspects in the following section.

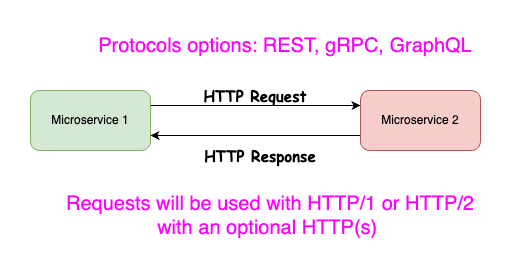
# Types of Microservices Communication

This is one of the confusing aspects when we talk about microservices communication but it's quite easy. There are only 3 ways the microservices can communicate as follows

* **Synchronous Communication**
* **Asynchronous Communication**
* **Hybrid Communication**

In this article, we will only look at**Synchronous communication**. In the next part of the article, we will see about **Asynchronous and Hybrid communication** between the microservices.

**1. Synchronous Communication between Microservices**



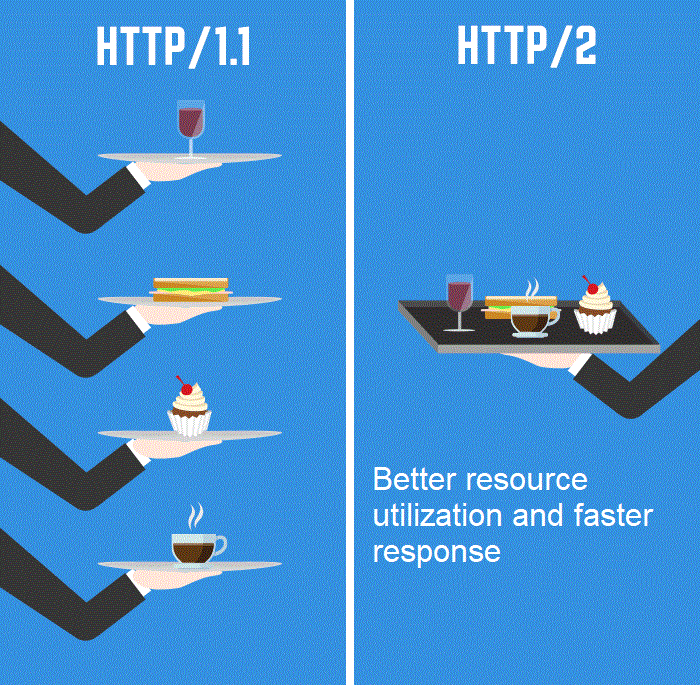
In synchronous communication, microservices call each other directly to their IP address and this works on an HTTP request-response model using REST API. Microservice 1 either gets the static IP of the Microservice upfront or gets the dynamic IP address (usually private IP) from the service registry like Eureka where all the instances register themselves. If Microservice 2 is down, Microservice 1 will get an error and then the action cannot be performed, and the message is lost. We cannot do a retry mechanism in this direct communication.

**a. REST API with HTTP/2**

The synchronous communication happens over HTTP/1 or HTTP/2 using protocols like REST, gRPC, and GraphQL. HTTP/1 is very slow, and it adds some latency hence it is better to use the HTTP/2 to get the benefit of multiplexing (less bandwidth and more efficiency).

However, to use HTTP/2 it is mandatory to use it over HTTP(s). In order to use HTTP(s), we need to generate a self-signed SSL certificate and load it to the microservice instances. Then we can easily make REST API calls using HTTP/2 instead of HTTP/1 calls which makes the communication very effective.

You can read my below article to understand in detail about HTTP/2



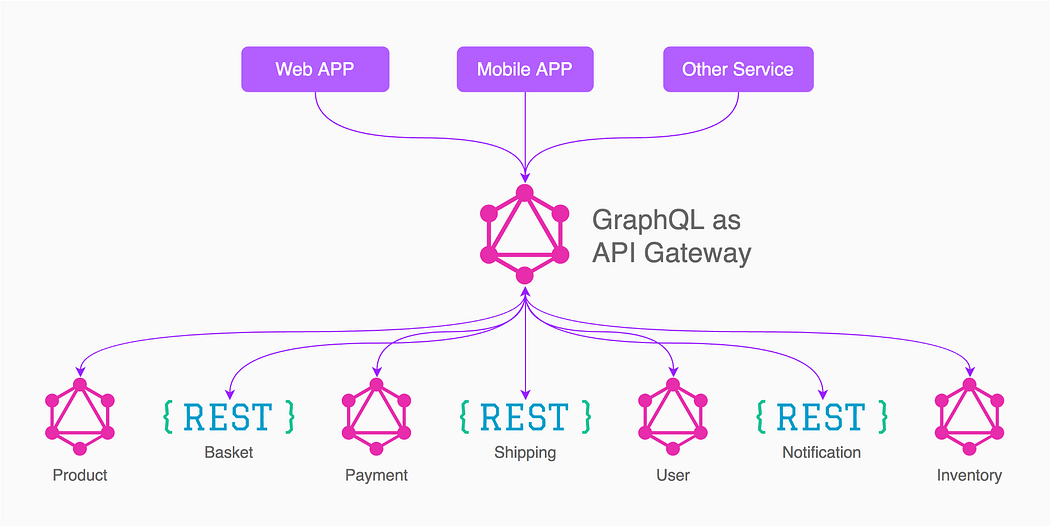
**b. gRPC Protocol**

Let's say you have a lot of API calls happening in the system and your application is of an e-commerce or finance nature, then you might need to use gRPC protocol. gRPC is designed for low latency and high throughput communication and it is great for lightweight microservices where latency is critical. gRPC has excellent support for bi-directional streaming and it uses two powerful things under the hood i.e HTTP/2 and Protobuf.

The only problem in gRPC is that debugging gets complicated as we use the binary files called Protobuf. Using a normal HTTP/1 or HTTP/2 REST calls, debugging is easy as we use JSON format in general. But if you weigh the pros and cons, then gRPC should be the ideal choice for the microservices that communicate quite often and needs a low latency.

**c. GraphQL Protocol**

Another important protocol used in synchronous microservices communication is GraphQL. This also works on top of HTTP but in a different aspect.



GraphQL is the perfect technology to bring your microservices together for aggregating the data as a unified API. We use GraphQL as a data layer for the microservices and combine the data from all these services into a single API that responds to the client. This is synchronous as the GraphQL Server/gateway needs to wait for the response from the other microservices, but it aggregates all the data into a single response to the client. This means the client just makes a single API call to get the data it needs without making multiple round trips. This saves a lot of bandwidth at a cost of little latency.

**Important Aspects of Synchronous Communication in Microservices**

**1. Non-Blocking IO**

When microservice 1 calls microservice 2, an HTTP Request is made and microservice 1 waits for the HTTP Response. The thread will keep waiting until the response is received and it uses up the system resource. Imagine there are around 100 API calls at the same time from service 1 to 2, then service 1 is gonna be down soon. An example of this is the RestTemplate library in SpringBoot which is synchronous and blocking.

In most programming languages these days we have non-blocking IO libraries available. One such library is the **Spring WebClient**. It is still synchronous, but it will not block the thread until the response is received. Instead, the thread does another job and a notification is received once the response is ready from Microservice 2.

In some use cases, the non-blocking approach uses much fewer system resources compared to the blocking one. So, you can consider using a nonblocking IO library depending on the language you use.

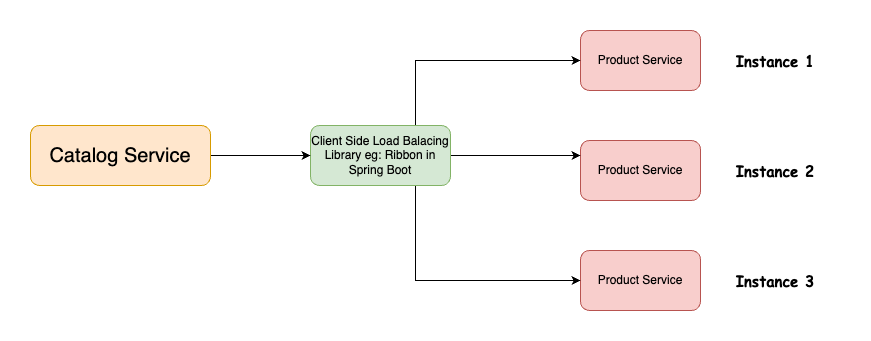
**2. Load Balancing**

Earlier, we discussed each microservices running multiple instances when its scales up on the load. In such cases, it is very very important for us to send the requests in a load-balanced manner to each of the instances. Failing to do so will result in a single instance of a microservice bombarded and hinder the whole system.

**here are two ways we can use a load balancer when we call another microservice.**

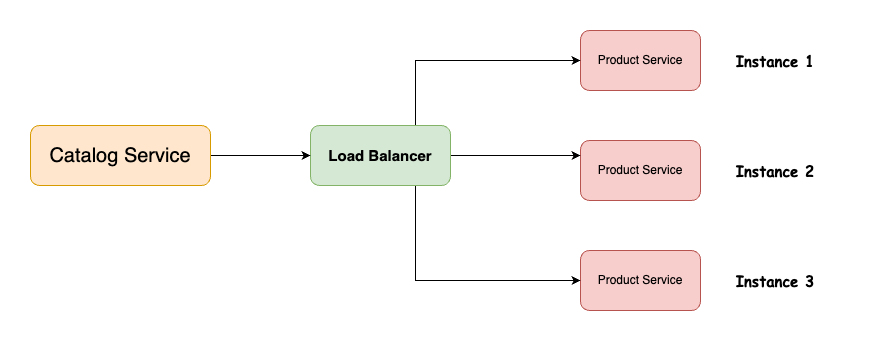
**a. Client-Side Load Balancing**

In a Client-Side load balancing, the Catalog Service requests the details of the Product Service from the Service registry and obtains the IP of all the 3 instances. Every Microservice instance has a Client-Side load balancer running inside which decides which instance of Product Service to call. Eg: We have a library called Ribbon in Spring Boot. This does the work of Client-Side load balancing.



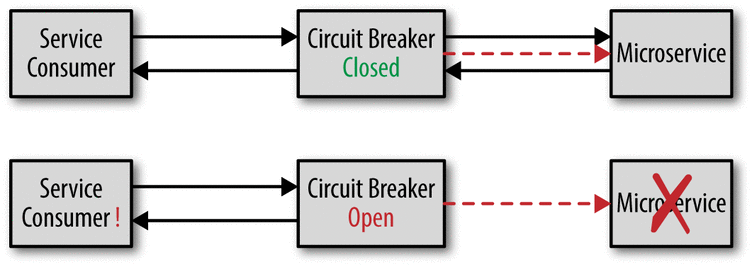
**b. Server-Side Load Balancing**

In Server-Side load balancing, we use the domain name of the load balancer that we define for the product service. Under this domain, there are three instances of Product Services registered now. So, when Catalog Service calls the Product Service domain name, the load balancer uses its algorithm to route the requests to each of the instances. A good example of such a load balancer is AWS ELB or Nginx etc.



**3. Circuit Breaker**

Another important aspect of synchronous communication is the Circuit Breaker. Circuit Breaker pattern can allow a microservice to continue operating when a related service fails, preventing the failure from cascading and giving the failing service time to recover. If microservice B is down, the circuit is open and there won't be any calls made from Service A to B. Then service A will check for service B and closes the circuit once service B is up.



**5. Distributed Tracing**

When we have a chain of synchronous communication happening in microservices, it is very important for us to trace the requests to understand where they fail, or which service takes a long time to process. Understanding this will help us to build resilient microservices architecture and we need to use the pattern called Distributed Tracing.

For distributed microservices tracing we can use Sleuth and Zepkin( which uses Trace Id and Span Id).

Zipkin- Is an GUI to shows them all MS details.

TraceId- It will be same among complete life cycle from source service to destination service.

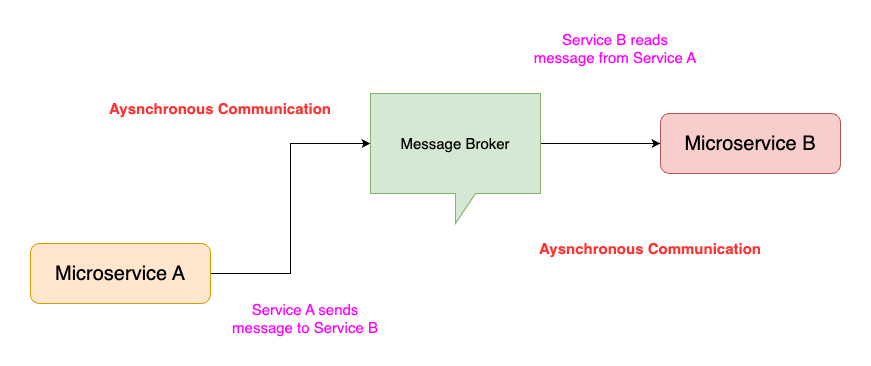
SpanId- it will be different for each and every MS .

**Summary:** In the first part of the Microservices Communication article, we saw about 3 ways of communication between the services i.e Synchronous, Asynchronous, and Hybrid. We dived deep into the Synchronous Communication between microservices and we discussed using different protocols like REST, gRPC, and GraphQL. Then we have seen the important aspects to be considered when designing the synchronous communication between microservices.

**2. Asynchronous Communication**

In this article, we are going to explore Asynchronous communication between microservices. We will see about three types of patterns used in async i.e one-to-one, one-to-many, and event-driven. Then we will see hybrid communication that uses the combination of sync and async communication Toward the end of the article, we are going to see the use cases, benefits, and drawbacks of these different communication mechanisms.

**In Asynchronous communication, a**[**microservice**](https://medium.com/javarevisited/10-microservices-design-principles-every-developer-should-know-44f2f69e960f)**sends a request to another microservice but it doesn’t wait for a response. Hence microservice A will not have blocked a thread while waiting for a response from service B. It is a one-way communication and hence non-blocking in nature. This will save a lot of resources and helps to perform efficient parallel processing.**



We can easily handle multiple consumers at a time because it is loosely coupled. Service A does not know the whereabouts of Service B and its instances. It just pushes to the specific [Queue](https://javarevisited.blogspot.com/2017/03/difference-between-stack-and-queue-data-structure-in-java.html)and continues its job while Service B will have consumer polling for messages in the Queue. As soon as a message arrives in the Queue, it will be picked up by Service B and processed. Handling failures in this pattern is also easy as we have ways to reprocess the messages.

The Message Queues are called Message Brokers and work as middleware between the microservices. There are various Asynchronous protocols like MQTT, STOMP, and AMQP are handled by platforms like [Apache Kafka](https://medium.com/javarevisited/top-10-apache-kafka-online-training-courses-and-certifications-621f3c13b38c) Stream, and RabbitMQ. The most popular protocol for this Asynchronous communication is AMQP (Advanced Message Queuing Protocol).

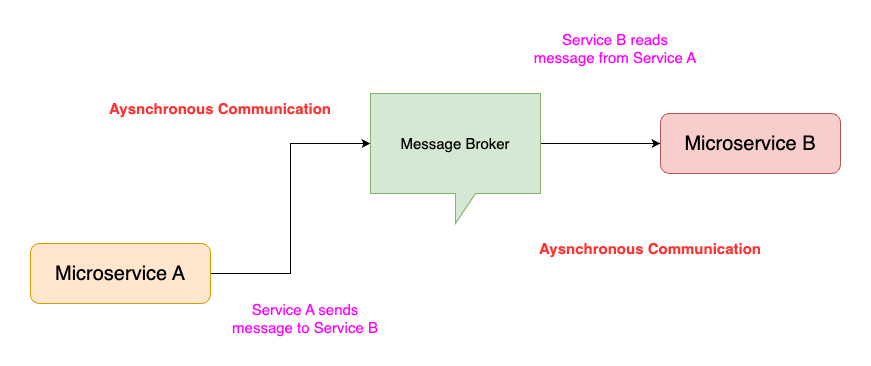
By using AMQP protocols, the client sends the message using message broker systems like Kafka and RabbitMQ queue.

**Types of Asynchronous communication**

Let us look at different types of patterns used in Asynchronous Microservices communication.

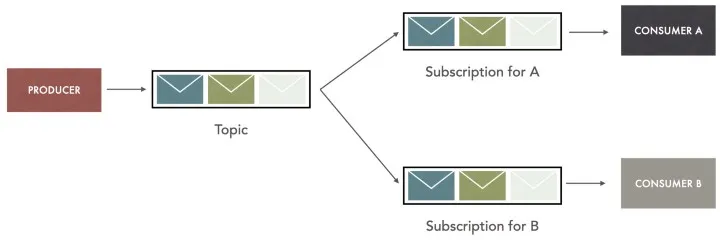
**a. One-to-One — Sender Receiver Communication**

This is a very common pattern used in async communication. Here only 2 services are involved. Service A pushes a message to the Message Queue and Service B reads the messages and does the processing. An example of this could be sending email notifications after Order processing.

[[](https://javarevisited.blogspot.com/2018/02/top-5-spring-microservices-courses-with-spring-boot-and-spring-cloud.html)](https://javarevisited.blogspot.com/2018/02/top-5-spring-microservices-courses-with-spring-boot-and-spring-cloud.html)

**b. One to Many — Publish & Subscribe**

This pattern is called Publish and Subscribe or broadcast or Fan-out pattern. In this pattern, the Message Broker has a concept called [Topic](https://javarevisited.blogspot.com/2020/05/top-16-jms-java-messaging-service-interview-questions-answers.html) which is subscribed by the multiple message queues inside the message broker. Then each of these message queues is consumed by different microservices.

[[](https://www.java67.com/2022/08/java-spring-boot-microservices-example.html)](https://www.java67.com/2022/08/java-spring-boot-microservices-example.html)

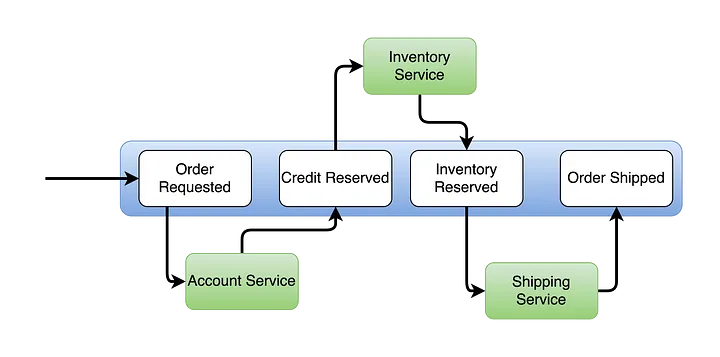
Example: We an order is completed, the Order service can push to the topic which will be received by other microservices that will generate payment receipts, send an email, etc at the same time using different microservices.

Can I use a Queue instead of a Topic that is listened to by multiple Services?

The answer is No because, if you use a queue, then the first consumer will receive the message and other services won't be getting the message. We have to use a Topic that will be read by multiple queues and each of them gets the same copy of the message.

**c. Event-Driven Communication**

Event Driven communication is slightly different from the above 2 patterns. In an event-driven system, different parts of the system respond to specific events that occur. It is easy to set up a system where different parts of the system can subscribe to specific events or different events, and then take appropriate action when those events occur. This can be especially useful in systems where are many different components that need to respond to the same or different events, such as in microservices.



In the above diagram, we can see that Order Service emits an event on Order Requested and Account Service receives it, does some processing, and emits emit back to Order Service. Then Order Service reserves the Credit and emits another event to Inventory Service Likewise, the chain flows across multiple services until the order is shipped. This avoids complex integration platforms such as ESB used in traditional SOA design.

**Important Aspects of Asynchronous Communication**

* Distributed Transaction Management
* Rollbacks in Distributed Transaction Management