Asynchronous Microservices : Fundamental Principle

[Quick Bite] Let's learn the below points in detail

- How microservices can communicate asynchronously.
- Which protocols can be used for asynchronous communication.
- How events and asynchronous communication are linked.
- The advantages and disadvantages of asynchronous communication.

A microservice is **asynchronous** if:

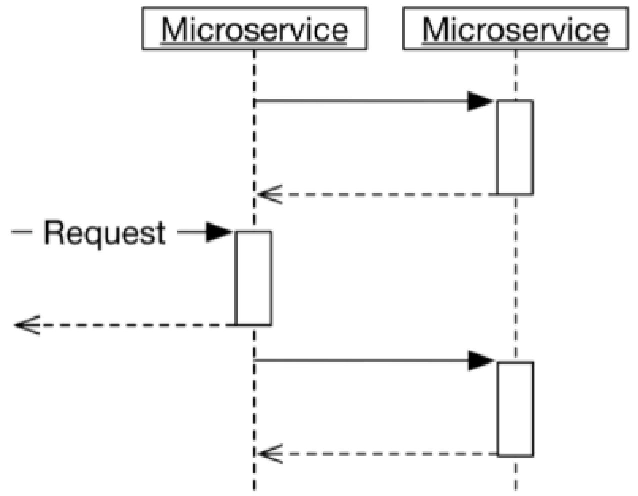
(a) It does not make a request to other microservices while processing requests. OR

(b) It makes a request to other microservices while processing requests and does not wait for the result.

(a) **No communication**

The microservice **does not communicate at all with other systems** while processing a request.

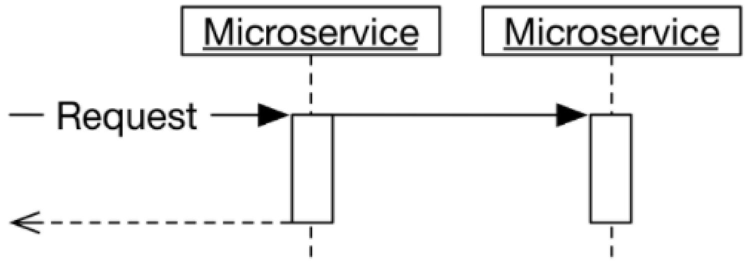
In that case, the microservice will typically communicate with the other systems at a different time, see the drawing below.



Asynchronous Microservices | Data Replication & bounded context

(b) **Does not wait for response**

The microservice sends a request to another microservice but **does not wait for a response**, see the drawing below.



Data replication and bounded context

Asynchronous communication becomes more complicated **if data is required** to execute a request.

The data specific for the bounded context should be stored in the bounded context in its own database schema.

The data should be accessed only by the logic in the bounded context and its interface. And eventually consistency in Data across the different bounding context should be maintained.

Communication Protocol

1. **Synchronous communication Protocol**

Synchronous communication protocol require response to every request of service done to server. For ex: REST and HTTP each request expecting some response.

2. **Asynchronous communication Protocol**

An asynchronous communication protocol sends messages and does not expect responses. For ex: Messaging system like Kafka uses this approach.

Asynchronous Microservices | Events

Rest vs Messaging

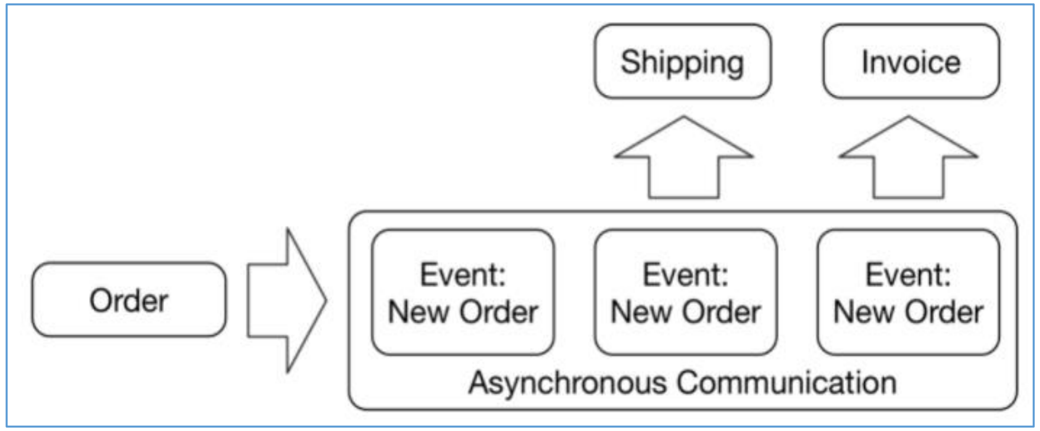
Conclusion: Event-driven microservices should be considered more often by developers and architects as they provide the foundation to build awesome systems and applications.

Deeper Considerations When Using REST	**Messaging for Event-Driven Microservices**
• Tight Coupling • Blocking • Error Handling	• Loose Coupling • Non-Blocking • Simple to scale • Greater Resiliency and Error Handling

Events

Each microservice decides for itself how it reacts to the events, see the drawing below.

If a microservice must react differently to a new order, the microservice can implement this change on its own.



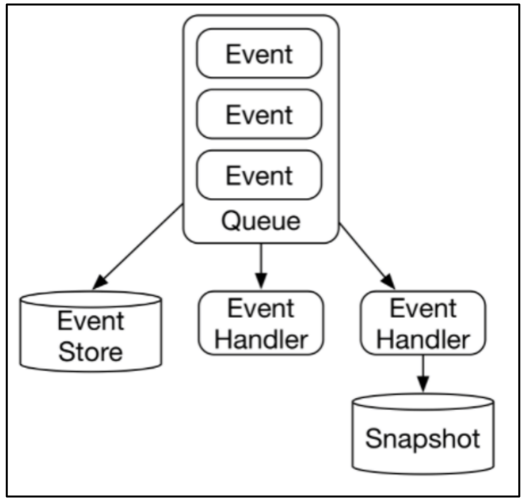
Asynchronous Microservices | Event Sourcing

Event Sourcing

These ideas form the basis for event sourcing.

The elements of an event sourcing implementation are shown in the drawing right:

- Event Queue
- Event Store
- Event handler



One important architectural decision comes Infront of architect here that where you put your event store?

It should inside the boundary of Micro Service or outside the bounding context. Off-course there is no one bullet for all kind of scenario; it depends your requirement. If somehow the possibility of Event schema is same across the micro-service than considering this outside the microservice bounding context is good idea but most of the time this is not the real situation.

Asynchronous Microservices | Challenges(Inconsistencies & CAP Theorem)

Conclusion:

*A microservice that relies on asynchronous communication, events, and data replication corresponds to an **AP system**.*

*The CAP theorem says that the only alternative is a **CP system**. This would be consistent but not available.*

Inconsistencies

Due to asynchronous communication, the system is not consistent. Some microservices already have certain information while others do not.

For example, *order process* might already have information about an order, but *invoicing* or *shipping* does not know about the order yet.

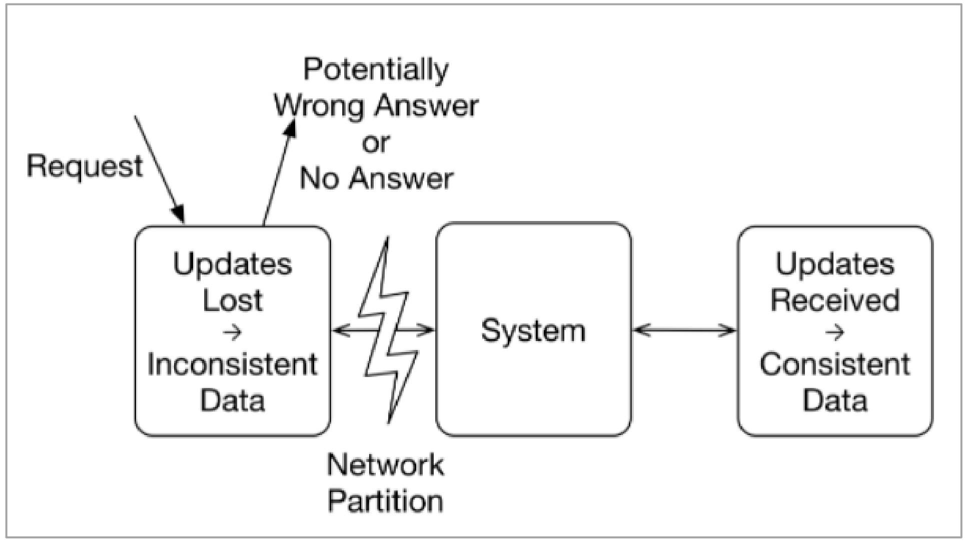
This problem cannot be solved. It takes time for asynchronous communication to reach all systems.

So, by the time you realized that this is not a problem, but it is an architectural choice and the same is covered under CAP theorem in next section.

CAP theorem

According to the CAP theorem, three characteristics exist in a distributed system:

- **Consistency (C)** means that all components of the system have the same information.
- **Availability (A)** means that no system stops working because another system failed.
- **Partition tolerance (P)** means that a system will continue to work in case of arbitrary package loss in the network.



Potentially Wrong Answer or No Answer
Request
Updates Lost → Inconsistent Data
System
Updates Received → Consistent Data
Network Partition

There are **only two options**:

1. **The system provides a response**. In this instance, the response can be wrong because changes have not reached the system; this is the *AP* case.
2. Alternatively, **the system returns no response**; this is the *CP* case.

Asynchronous Microservices | More on Inconsistencies

Conclusion:

Repairing inconsistencies - Guaranteed order of events using Event Sourcing

This one question, architect should always ask before deep diving in Asynchronous approach, **Are inconsistencies acceptable?**

The inconsistency of an asynchronous system is inevitable unless you want to give up availability.

Let's deep dive in it. Every customer wants a reliable system, data inconsistency seems to be contradictive of this. So here you should know, does the temporary data inconsistency is really a problem for system? After all inconsistency will disappear eventually so is this really a problem?

For example, if goods are listed days before the first sale, inconsistencies are initially acceptable and must only be corrected when the goods are finally being sold.

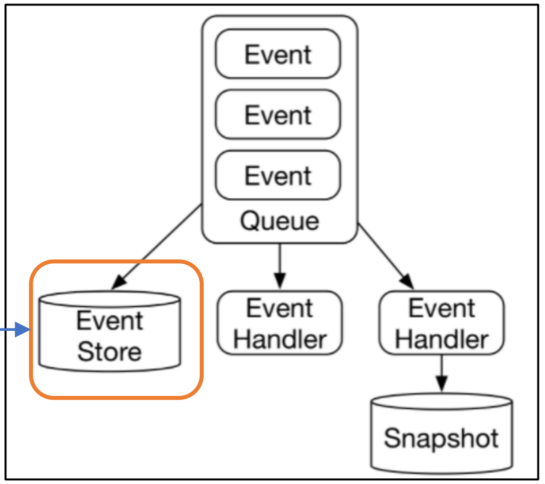
If **inconsistencies are not acceptable at all, asynchronous communication is not an option**. This means that synchronous communication must be used with all its disadvantages.

In the other hand if eventual consistency will work for system than how you make sure

that this will happen in all the circumstances for system.

How to deal with the Failure case which eventual syncing of data.

This Event store will help to handle the Failure case event handling

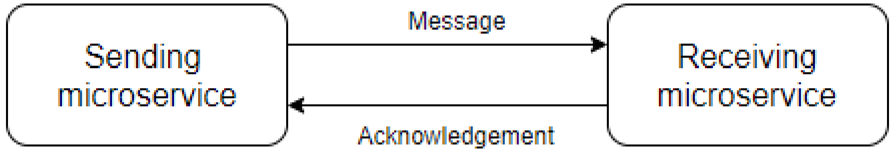


Asynchronous Microservices | More Challenges

Conclusion:

Challenges with Guaranteed delivery, Idempotency, One Recipient , Order of message & Test should require your architectural attention in system design process.

Let's discuss more challenges in Asynchronous system have
Guaranteed delivery – This is very difficult to maintain Guaranteed delivery and acknowledgement in case of Asynchronous microservices communication, but architect Must think about this in their system, may be using some notification and error handling for failure case.



Idempotency - Many time in asynchronous system happen that you are sending message multiple time because you feel that system has not processed the last message and so if at recipient side if they process both message that means system is in inconsistent state. So, architect must think about this and implement it like:

Design distributed systems in such a way that the microservices are **idempotent**. This means that a message can be processed more than once, but the state of the service no longer changes.

There is another challenge in this belt is **One recipient**

One Recipient - For example, it would be incorrect from a domain perspective when multiple instances of the invoice microservice receive the order and all of them generate an invoice. So multiple invoice generated for same order. It is wrong.

For this, messaging systems normally have an option to send messages only to a single recipient. This recipient then must confirm the message and process it. Such a communication type is called **point to point communication**.

And, in some cases order of messages also matter in your service. For example, it would not be good if changes to the billing address are processed after the invoice has been written; the invoice would still contain the wrong address.

For this reason, it may be important to guarantee the order of messages.

Test - With asynchronous microservices, the **continuous delivery pipelines must be independent** to enable independent deployment. To do this, **the testing of the microservices must be independent** of other microservices. Here Architect must do some smart planning because this test must be as same as real workflow. Challenging but worth to give time to plan. **Black box test** is the easy to adopt here.

Asynchronous Microservices | Advantages & Variations

Conclusion:

The below important Advantage and variation of the Asynchronous Microservice consideration

Advantages -

Especially for distributed systems, asynchronous communication has several decisive advantages:

- Guarantee of delivery in design makes system more resilient.
- Due to loose-coupling between sub parts of the system, scalability required is better catered here

Variations -

Apache Kafka as an example for a message-oriented middleware (MOM). Kafka offers the option to store messages for a very long time. This can be helpful for event sourcing. This feature distinguishes Kafka from other MOMs which are also good options for microservices.

Summary of Complete learning:

- **Asynchronous communication should be preferred over synchronous communication** between microservices due to the advantages concerning resilience and decoupling.
- **The only reason against this is inconsistency**. Therefore, it is important to know exactly what the requirements are, especially concerning consistency, in order to make the technically correct decision.
- Choosing asynchronous communication has **the potential to solve the essential challenges** of the microservices architecture and should, therefore, be considered in any case.