**Inter-service communication**

Some of the things that we'll cover in this module are:

1. How do services interact with each other
2. When should you prefer which communication mode
3. What are the tradeoffs that you need to consider when making this decision

Why is this important? Well, when you have a huge system with a lot of microservices interacting with each other, their communication needs to be efficient to provide the best user experience and also to avoid any cascading effects across the system.

Modes of communication

There are primarily two modes of communication between services:

**Synchronous:** When a service waits for a downstream system to respond before responding back to the client with a success or failure response.

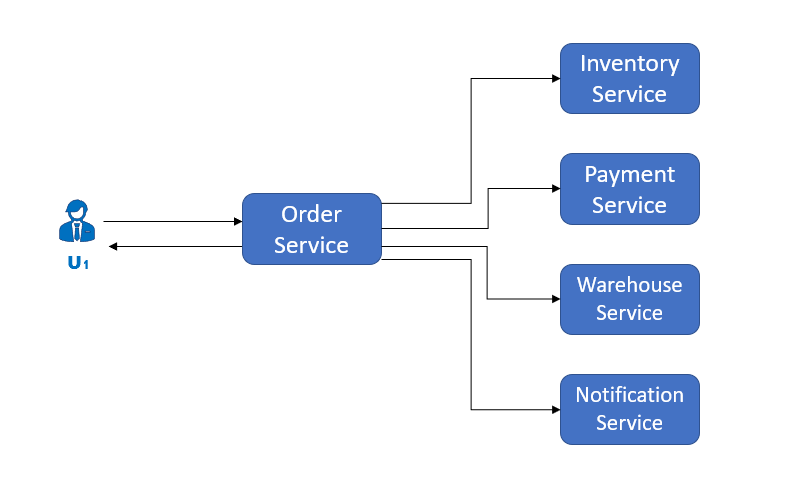
**Asynchronous:** This is a more of a fire and forget approach. A service will fire a call to the downstream system and won’t track it further.

Let us look at how these communications would work in a real-world scenario.

Synchronous Approach

Let’s say you are building Amazon. You have a user U1 trying to place an order. U1 will reach out to the Order Service. Order Service will now talk to the Inventory Service to find out if a sufficient quantity of the product is available. If that is the case, Inventory Service will send a success response. Otherwise, it will respond with an error, and Order Service will respond to the user saying the order could not be placed.

Now if the inventory response was a success, the Order Service will talk to the Payment Service to process the payment. Once the payment is successful, the Order Service will now talk to the Warehouse Service asking it to start packing and prepare for shipping the product to the user. Once Warehouse Service responds with a success, the Order Service will talk to a Notification Service to send an email to the user saying their order has been placed, with so and so payment details and sharing an ETA for the delivery of the product.



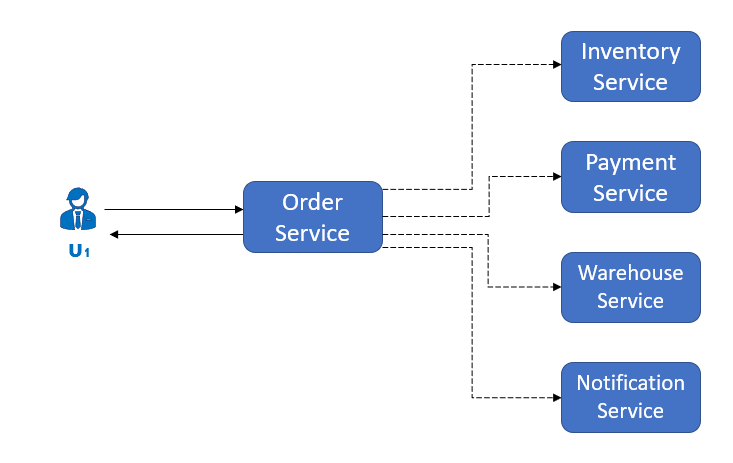
Now, this is a happy scenario. What happens when one of the calls fails? Well, it depends on which call fails. If the call to Notification Service fails, does it make sense to cancel the order? No. We shouldn't cancel an order just because the Notification Service failed. However, what if payment fails? Now we definitely need to cancel the order. But now we need to update the Inventory again to undo the change to the product quantity. What if the call to Inventory Service fails?

So as you can see, there are some loopholes in a purely synchronous approach.

* It has very high latency as the user does not get notified until all the calls have come back with a success or failure response.
* The system is tightly coupled, and any failure will have cascading effects across the board.
* The code becomes very complex since we need to handle all the cascading error scenarios.
* Due to complexity, it requires extremely high maintenance.

Asynchronous Approach

Let us see what happens in a purely asynchronous approach.



U1 sends a call to the Order Service which makes asynchronous calls to all the downstream systems. In such a case, even if Inventory Service responds with an error code, or even if the payment fails, the order would get placed. Which is an even bigger mess! So how do we go about this?

Well, as we can see, some parts of this process must be mandatory, and some can be done on a best-effort basis. If the Inventory Service or Payment Service responds with an error, we cannot place the order. But if the notification does not go through or the Warehouse Service is temporarily down, we don’t need to cancel our order. So we can follow a hybrid approach here; use a synchronous approach for the mandatory steps and an asynchronous approach for the rest.

Best of Both Worlds

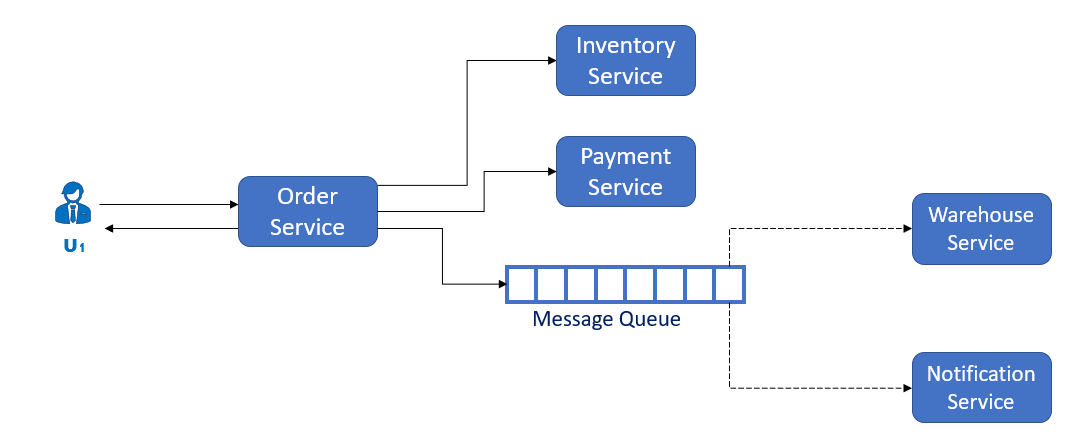
The Hybrid approach suggests that the mandatory tasks need to be performed in a synchronous manner and everything else can be done asynchronously.

So Order Service will send out a synchronous call to Inventory Service, and wait for a response. In case of success, it will call the Payment Service. If the Payment Service gives a successful response, Order Service will make parallel asynchronous calls to the Warehouse Service and Notification Service and, at the same time, respond to the user saying the order has been placed. If the Payment Service call had failed, Order Service would send an asynchronous call to the Inventory Service reverting the quantity change.

So this looks like a much better solution. There are still some misses here though. What if the asynchronous call to Warehouse Service failed? It would lose the details for that order. This is where we would use Message Queues.

Message Queues

Message Queues are highly fault-tolerant and persist messages for some time. How a message Queue works is, it has some Publishers adding messages to it, and some Subscribers listening to it and picking up the events meant for them at their own pace. Since these queues store messages for some time, if a subscriber is temporarily down, the messages will remain in the queue and will be picked up when the subscriber is running again.



So now, when Order Service wants to make asynchronous calls to Warehouse and Notification services, it will instead put an event in the Message Queue. Warehouse Service and Notification Service, which will be listening to the queue, will pick up the events meant for them. If one of the systems is down, the messages will remain in the queue until the service is back up and ready to receive messages again. This way, none of the data gets lost.

Benefits of Message Queues

In addition to the data not getting lost, there are a few more upsides to this approach.

**Increased Stability**: As the events will not get lost, even when a service is not available, the system becomes substantially more stable.

**Enhanced Scaling**: Since a part of the system is communicating asynchronously and the events are being handled via a message queue, when there is a surge in the number of requests, only the services that are communicating synchronously need to be scaled up by adding additional hardware. Warehouse Service and Notification Service keep working at their own pace, picking up events from the message queues.

**Reduced Cost**: This enhanced scalability also saves cost since hardware resources are the most expensive resources. Since you now don’t need to add hardware for Warehouse Service and Notification Service, it is a major win for the wallet.

**Reduced Complexity**: Since we do not need to handle cascading effects anymore, the code becomes less complex and easier to maintain, which leads to fewer bugs.

Message Queue Options

So now you know Message Queues are a lifesaver. Do you really need to build them from scratch? Not really, and it is not recommended either. You could build your own message queue but it would not be as battle-tested as some of the solutions out there. Some of the most popular ones are:

* Kafka
* RabbitMQ
* ActiveMQ

These are some of the most commonly used Message Queues that are highly tested, reliable and have a great support community.