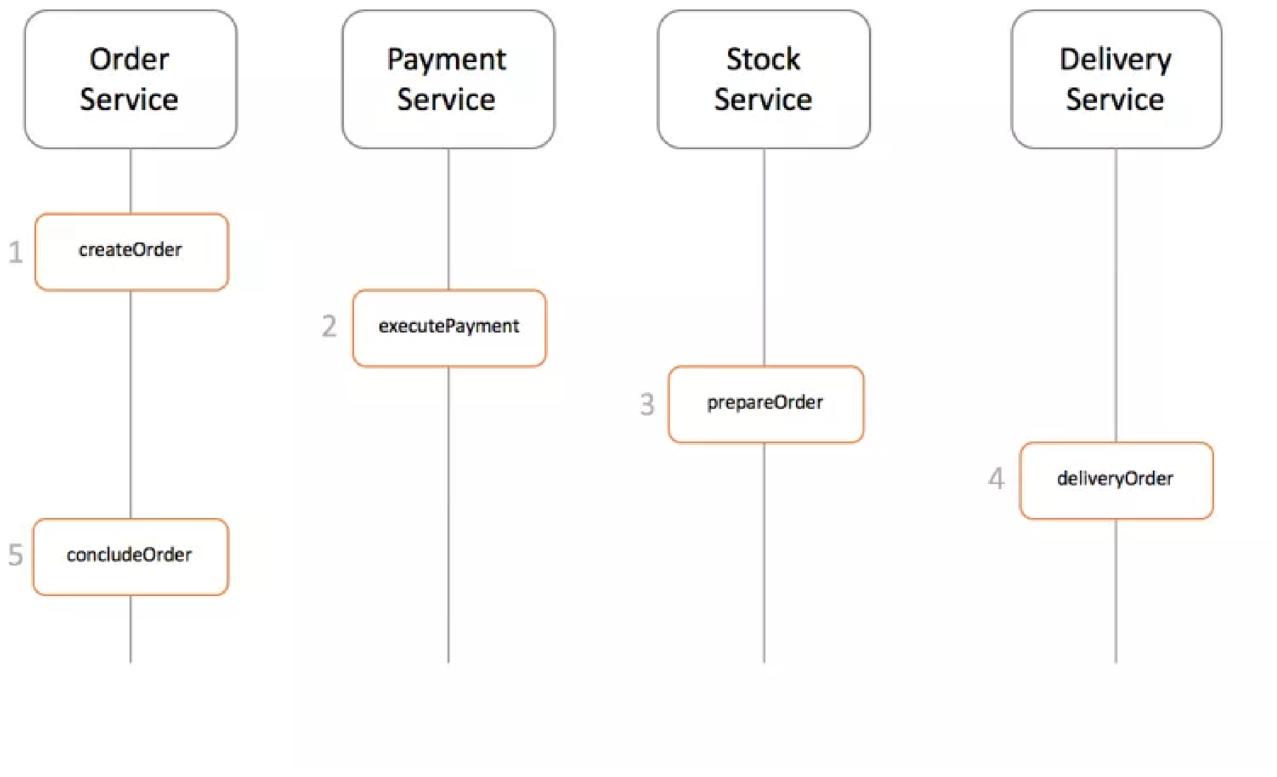
The SAGA Pattern

One of the most well-known patterns for distributed transactions is called Saga. The first paper about it was published back in 1987 and has it been a popular solution since then.

A saga is a sequence of local transactions where each transaction updates data within a single service. The first transaction is initiated by an external request corresponding to the system operation, and then each subsequent step is triggered by the completion of the previous one.

Using our previous e-commerce example, in a very high-level design a saga implementation would look like the following:



There are a couple of different ways to implement a saga transaction, but the two most popular are:

- **Events/Choreography:** When there is no central coordination, each service produces and listen to other service's events and decides if an action should be taken or not.
- **Command/Orchestration**: when a coordinator service is responsible for centralizing the saga's decision making and sequencing business logic

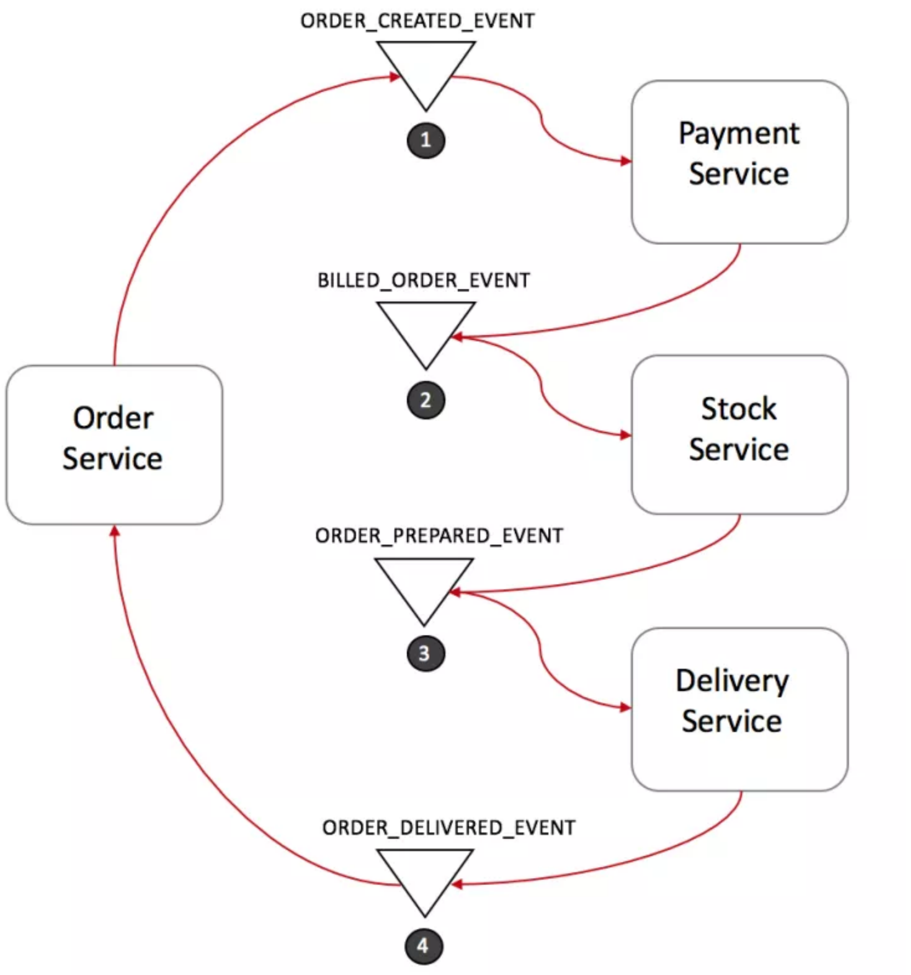
Let's go a little bit deeper in each implementation to understand how they work.

Events/Choreography

In the Events/Choreography approach, the first service executes a transaction and then publishes an event. This event is listened by one or more services which execute local transactions and publish (or not) new events.

The distributed transaction ends when the last service executes its local transaction and does not publish any events or the event published is not heard by any of the saga's participants.

Let's see how it would look like in our e-commerce example:



1. *Order Service* saves a new order, set the state as *pending* and publish an event called ***ORDER_CREATED_EVENT***.
2. The *Payment Service* listens to ***ORDER_CREATED_EVENT***, charge the client and publish the event ***BILLED_ORDER_EVENT***.
3. The *Stock Service* listens to ***BILLED_ORDER_EVENT***, update the stock, prepare the products bought in the order and publish ***ORDER_PREPARED_EVENT***.
4. *Delivery Service* listens to ***ORDER_PREPARED_EVENT*** and then pick up and deliver the product.

At the end, it publishes an ***ORDER_DELIVERED_EVENT***

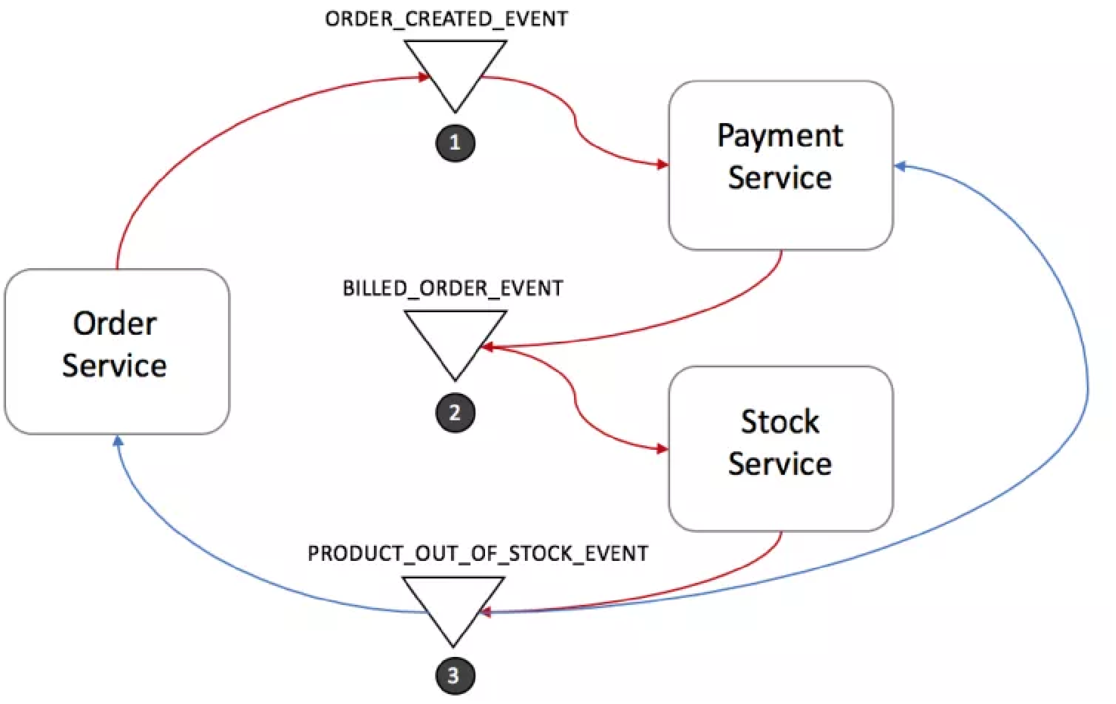
5. Finally, *Order Service* listens to ***ORDER_DELIVERED_EVENT*** and set the state of the order as concluded.

In the case above, if the state of the order needs to be tracked, Order Service could simply listen to all events and update its state.

Rollbacks in distributed transactions

Rolling back a distributed transaction does not come for free. Normally you have to implement another operation/transaction to compensate for what has been done before.

Suppose that Stock Service has failed during a transaction. Let's see what the rollback would look like:



1. *Stock Service* produces ***PRODUCT_OUT_OF_STOCK_EVENT***;
2. Both *Order Service* and *Payment Service* listen to the previous message:
   1. *Payment Service* refund the client
   2. *Order Service* set the order state as failed

Note that it is crucial to define a common shared ID for each transaction, so whenever you throw an event, all listeners can know right away which transaction it refers to.

Benefits and drawbacks of using Saga’s Event/Choreography design

Events/Choreography is a natural way to implement Saga’s pattern, it is simple, easy to understand, does not require much effort to build, and all participants are loosely coupled as they don’t have direct knowledge of each other. If your transaction involves 2 to 4 steps, it might be a very good fit.

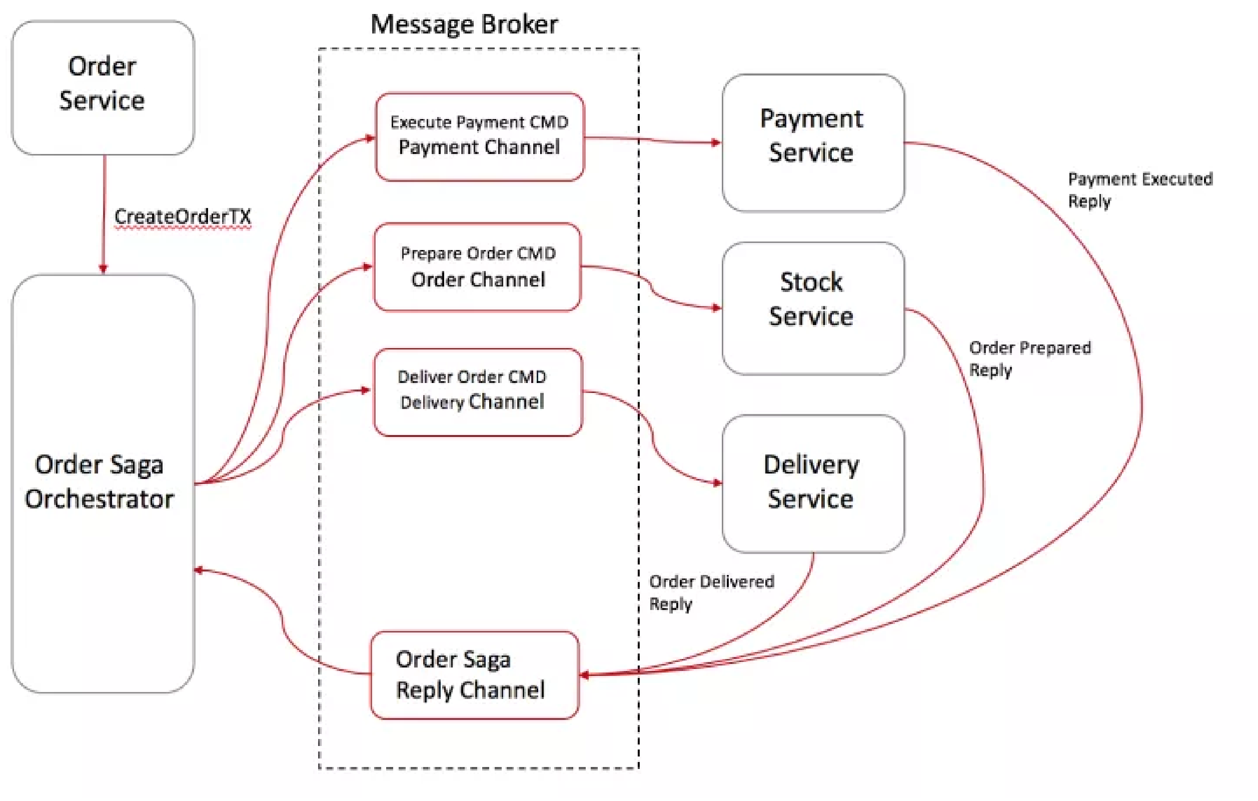
However, this approach can rapidly become confusing if you keep adding extra steps in your transaction as it is difficult to track which services listen to which events. Moreover, it also might add a cyclic dependency between services as they have to subscribe to one another’s events.

Finally, testing would be tricky to implement using this design, in order to simulate the transaction behavior you should have all services running.

Saga’s Command/Orchestration sequencing logic

In the orchestration approach, we define a new service with the sole responsibility of telling each participant what to do and when. The saga orchestrator communicates with each service in a command/reply style telling them what operation should be performed.

Let’s see how it looks like using our previous e-commerce example:



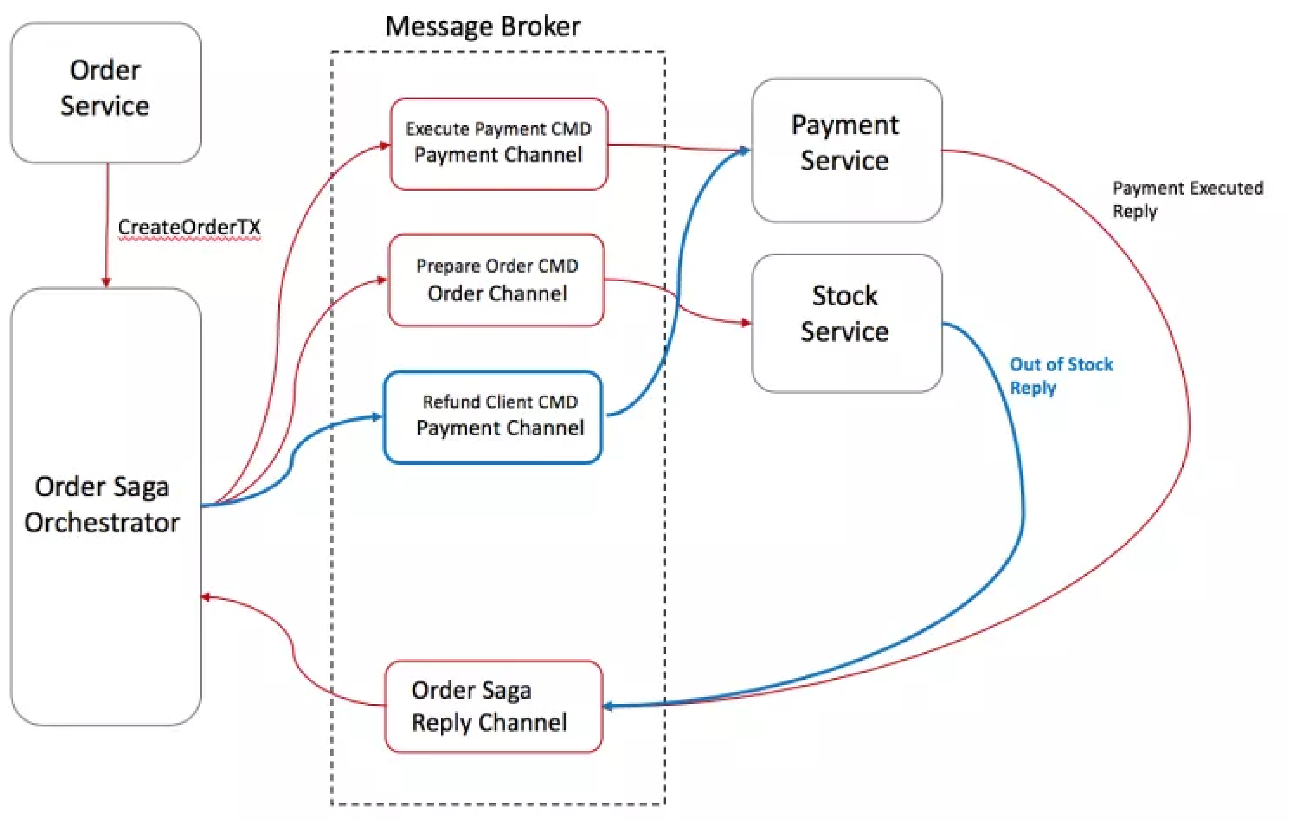
1. *Order Service* saves a pending order and asks Order Saga Orchestrator (OSO) to start a *create order transaction*.
2. *OSO* sends an ***Execute Payment*** command to *Payment Service*, and it replies with a ***Payment Executed*** message
3. *OSO* sends a ***Prepare Order*** command to Stock Service, and it replies with an ***Order Prepared*** message
4. *OSO* sends a ***Deliver Order*** command to Delivery Service, and it replies with an ***Order Delivered*** message

In the case above, Order Saga Orchestrator knows what is the flow needed to execute a “create order” transaction. If anything fails, it is also responsible for coordinating the rollback by sending commands to each participant to undo the previous operation.

A standard way to model a saga orchestrator is a State Machine where each transformation corresponds to a command or message. State machines are an excellent pattern to structure a well-defined behavior as they are easy to implement and particularly great for testing.

Rolling back in Saga's Command/Orchestration

Rollbacks are a lot easier when you have an orchestrator to coordinate everything:



1. *Stock Service* replies to OSO with an ***Out-Of-Stock*** message;
2. OSO recognizes that the transaction has failed and starts the rollback
    1. In this case, only a single operation was executed successfully before the failure, so *OSO* sends a ***Refund Client*** command to *Payment Service* and set the order state as failed

Benefits and drawbacks of using Saga's Command/Orchestration design

Orchestration-based sagas have a variety of benefits:

- Avoid cyclic dependencies between services, as the saga orchestrator invokes the saga participants but the participants do not invoke the orchestrator
- Centralize the orchestration of the distributed transaction

- Reduce participants complexity as they only need to execute/reply commands.
- Easier to be implemented and tested
- The transaction complexity remains linear when new steps are added
- Rollbacks are easier to manage
- If you have a second transaction willing to change the same target object, you can easily put it on hold on the orchestrator until the first transaction ends.

However, this approach still has some drawbacks, one of them is the risk of concentrating too much logic in the orchestrator and ending up with an architecture where the smart orchestrator tells dumb services what to do.

Another downside of Saga's Orchestration-based is that it slightly increases your infrastructure complexity as you will need to manage an extra service.

Saga Pattern Tips

Create a unique Id per Transaction

Having a unique identifier for each transaction is a common technique for traceability, but it also helps participants to have a standard way to request data from each other. By using a transaction Id, for instance, Delivery Service could ask Stock Service where to pick up the products and double check with the Payment Service if the order was paid.

Add the reply address within the command

Instead of designing your participants to reply to a fixed address, consider sending the reply address within the message, this way you enable your participants to reply to multiple orchestrators.

Idempotent operations

If you are using queues for communication between services (like SQS, Kafka, RabbitMQ, etc.), I personally recommended you make your operations Idempotent. Most of those queues might deliver the same message twice.

It also might increase the fault tolerance of your service. Quite often a bug in a client might trigger/replay unwanted messages and mess up with your database.

Avoiding Synchronous Communications

As the transaction goes, don't forget to add into the message all the data needed for each operation to be executed. The whole goal is to avoid synchronous calls between the services just to request more data. It will enable your services to execute their local transactions even when other services are offline.

The downside is that your orchestrator will be slightly more complex as you will need to manipulate the requests/responses of each step, so be aware of the tradeoffs.