**Saga Design pattern**:

**What is Saga:**

A saga is a sequence of local transactions. Each service in a saga performs its own transaction and publishes a message (or) event. The other services listen to that event (or) message and perform the next local transaction. If one transaction fails for some reason, the saga also executes compensating transactions to undo the impact of the preceding transactions.

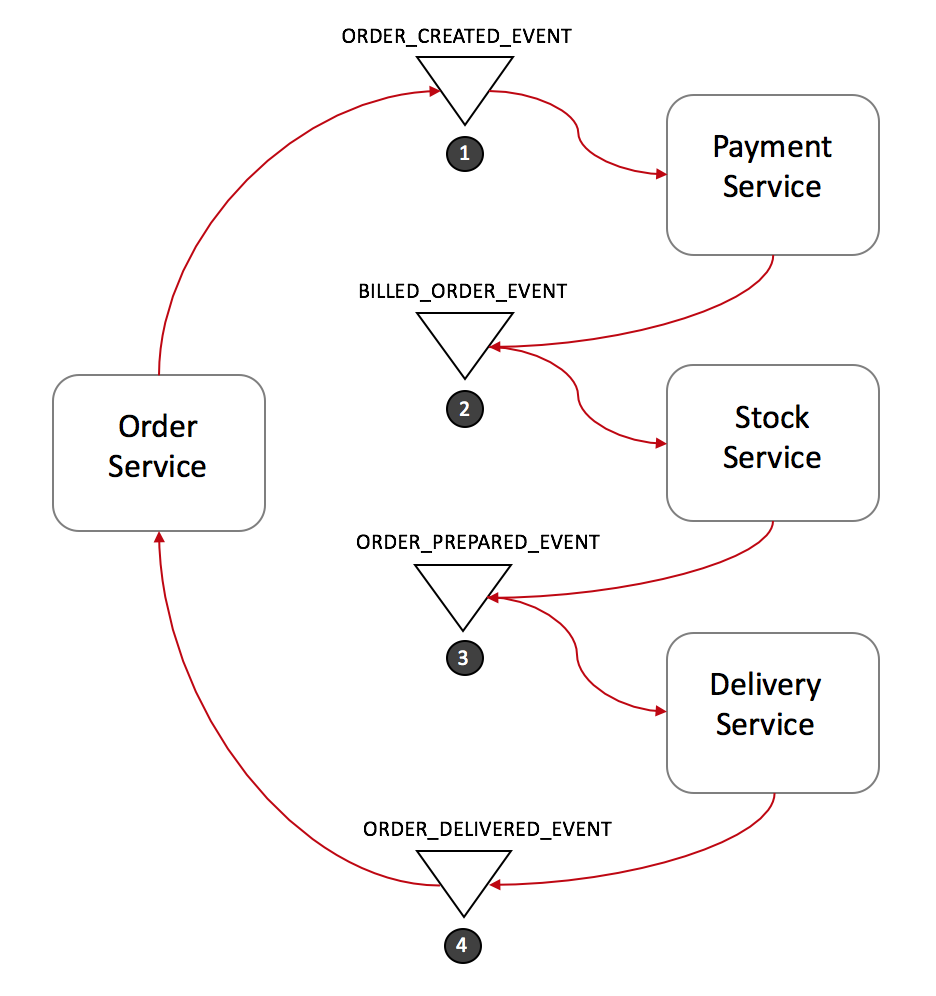
**There are two ways of coordination sagas:**

- **Choreography**

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

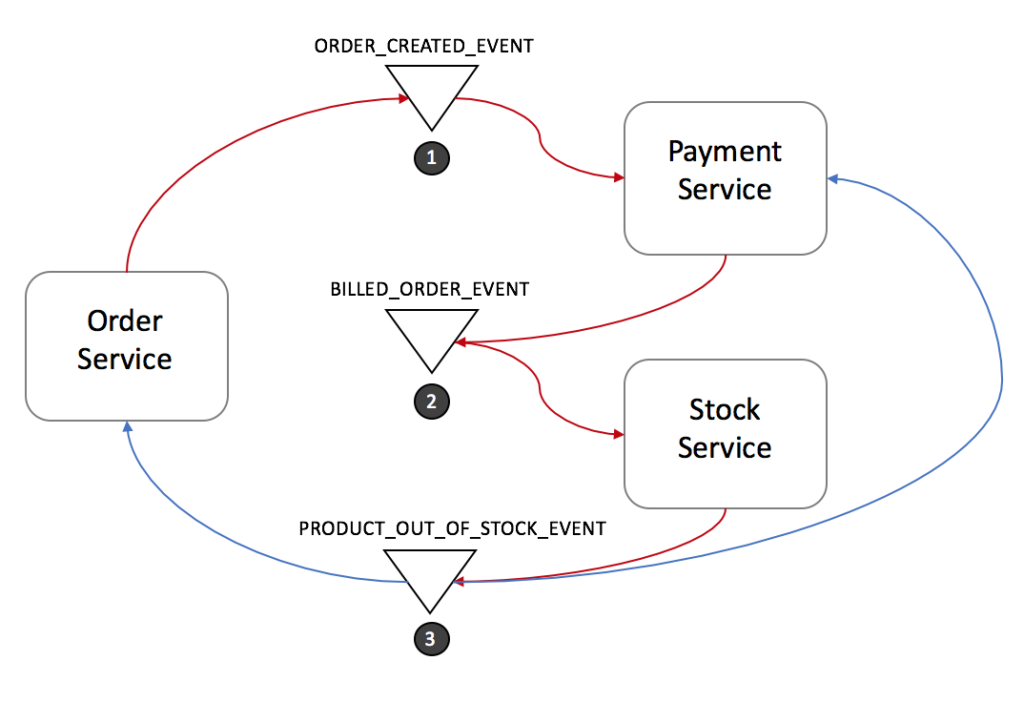
3) The Stock Service listens to BILLED\_ORDER\_EVENT, update the stock, prepare the products bought in the order and publish ORDER\_PREPARED\_EVENT.

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.

**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.



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.

**Advantages of Choreography:**

1) Easy to implement.

2) Appropriate if the number of local transactions is small.

3) Loosely coupled endpoints.

4) Suitable for agile organizations.

5) Enables fast processing. As no dependency on the central controller.

**Disadvantages of Choreography:**

1) It gets very complicated when the number of local transactions increases.

2) Cyclic dependency may occur between services.

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

**- Orchestration:**

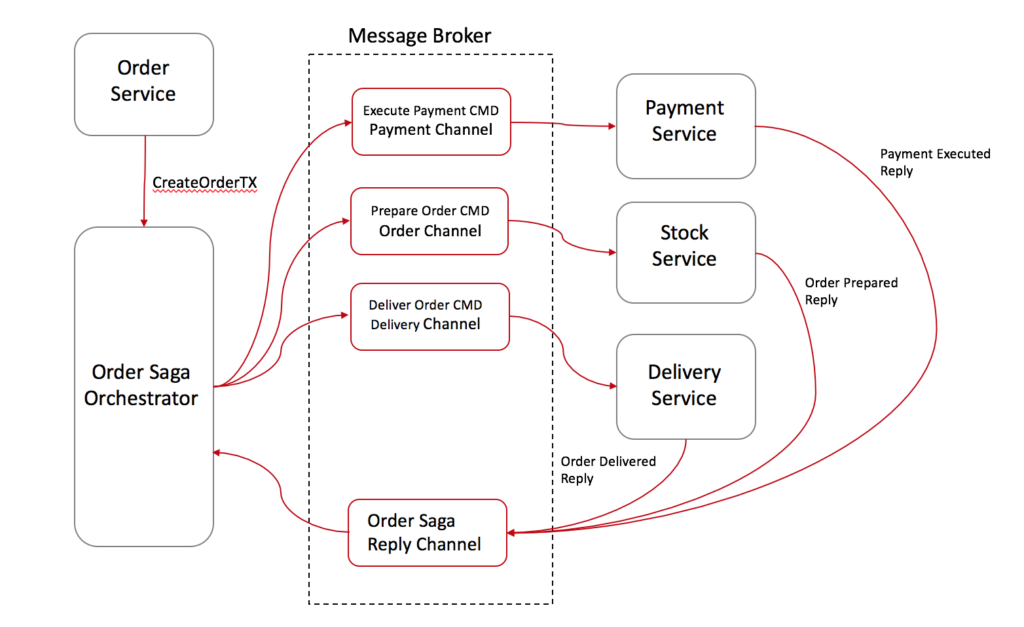
— an orchestrator (object) tells the participants what local transactions to execute.

(or)

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

The microservice Orchestration is more like a centralized service. It calls one service and waits for the response before calling the next service. This follows a request/response type paradigm.

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



1) Order Service saves a pending order and asks Order Saga Orchestrator (OSO) to start a create order transaction.

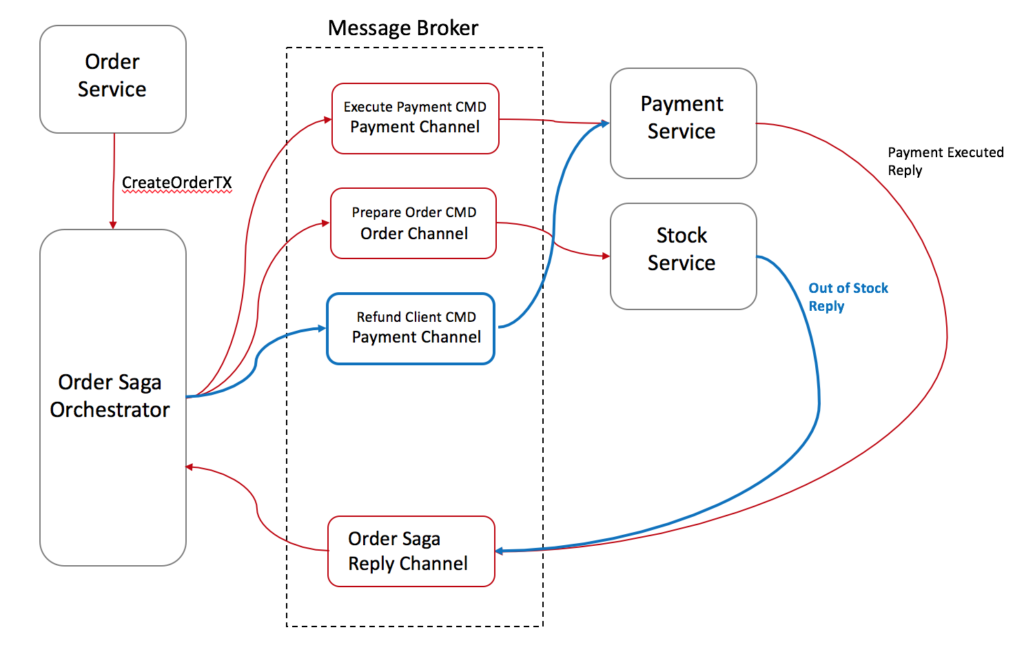
2) OSO sends an Execute Payment command to Payment Service, and it replies with a Payment Executed message

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 the flow is 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.

**Rolling Back in Saga's Command/Orchestration**



**Advantages of Orchestration:**

1) Avoid cyclic dependencies between services, as the saga orchestrator invokes the saga participants, but the participants do not invoke the orchestrator

2) Centralize the orchestration of the distributed transaction

3) Easier to be implemented and tested

4) The transaction complexity remains linear when new steps are added.

5) Rollbacks are easier to manage.

6) Reduce participants complexity as they only need to execute/reply commands.

**Disadvantages of Orchestration:**

1) This method creates a situation that the communication channel becomes intelligent and the endpoints become a dummy.

2) All the business logic is implemented inside the orchestrator. So the orchestrator becomes the monolith.

3) Creating dependency due to coupled services. For example, if service A is down, service B will never be called.

4) Working is very slow.

**Advantages of the Saga pattern:**

1)The main benefit of the Saga Pattern is that it helps maintain data consistency across multiple services without tight coupling. This is an extremely important aspect for a microservices architecture.

2)It is support for long-lived transactions. Because each microservice focuses only on its own local atomic transaction, other microservices are not blocked if a microservice is running for a long time.

**Disadvantages of the Saga pattern:**

1) The Saga pattern is difficult to debug, especially when many microservices are involved.

2) The event messages could become difficult to maintain if the system gets complex.

3) The Saga pattern is it does not have read isolation. For example, the customer could see the order being created, but in the next second, the order is removed.

**Saga Pattern Tips:**

1. **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.

1. **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.

1. **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.

1. **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.