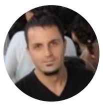
SAGA Pattern Briefly



Osman Başkök
Sep 2, 2019 · 3 min read

What is SAGA Pattern?

SAGA pattern has been around for over 30 years and is widely used for distributed transaction management. it's simply one of the ways to maintain data consistency for distributed systems. When you are working with micro services based architectures, data consistency easily becomes a headache for developers. Since each service can have its own database and be in a different environment, it's no longer possible to take advantage of the ACID transactions. Saga simply sacrifices atomicity and relies on eventual consistency.

When to use it?

When atomicity is not a necessity. In distributed systems, like micro services based architectures that are using database per service. For example; When you book an online holiday reservation, the system first returns a success message to the customer and then sequentially performs the flight (connecting flights, if there are), hotel, car etc. reservations using different sub-systems.

SAGA vs. 2PC

2PC works as single commit, aims to perform ACID transactions on distributed systems. It is used in where strong consistency is important. SAGA on the other hand works sequentially, not as a single commit. Each operation gets committed before the following one which makes the data eventually consistent. Thus Saga consists of multiple steps whereas 2PC acts like a single request.

One important consideration with 2PC is that it could be a single point of failure in the system and cause a "blocking problem".

Choreography-Based Saga

In this approach, there would be no orchestrator in the system. Each service performs its own process and if the result is successful, it fires a success event for the next step to continue with. In case of a failure it fires a failure event for the previous step. So the services that have worked before this step can sequentially perform rollbacks.

This approach is useful for the cases which consist of only a few steps. As it includes more steps, the events and the design could get more complex to manage.

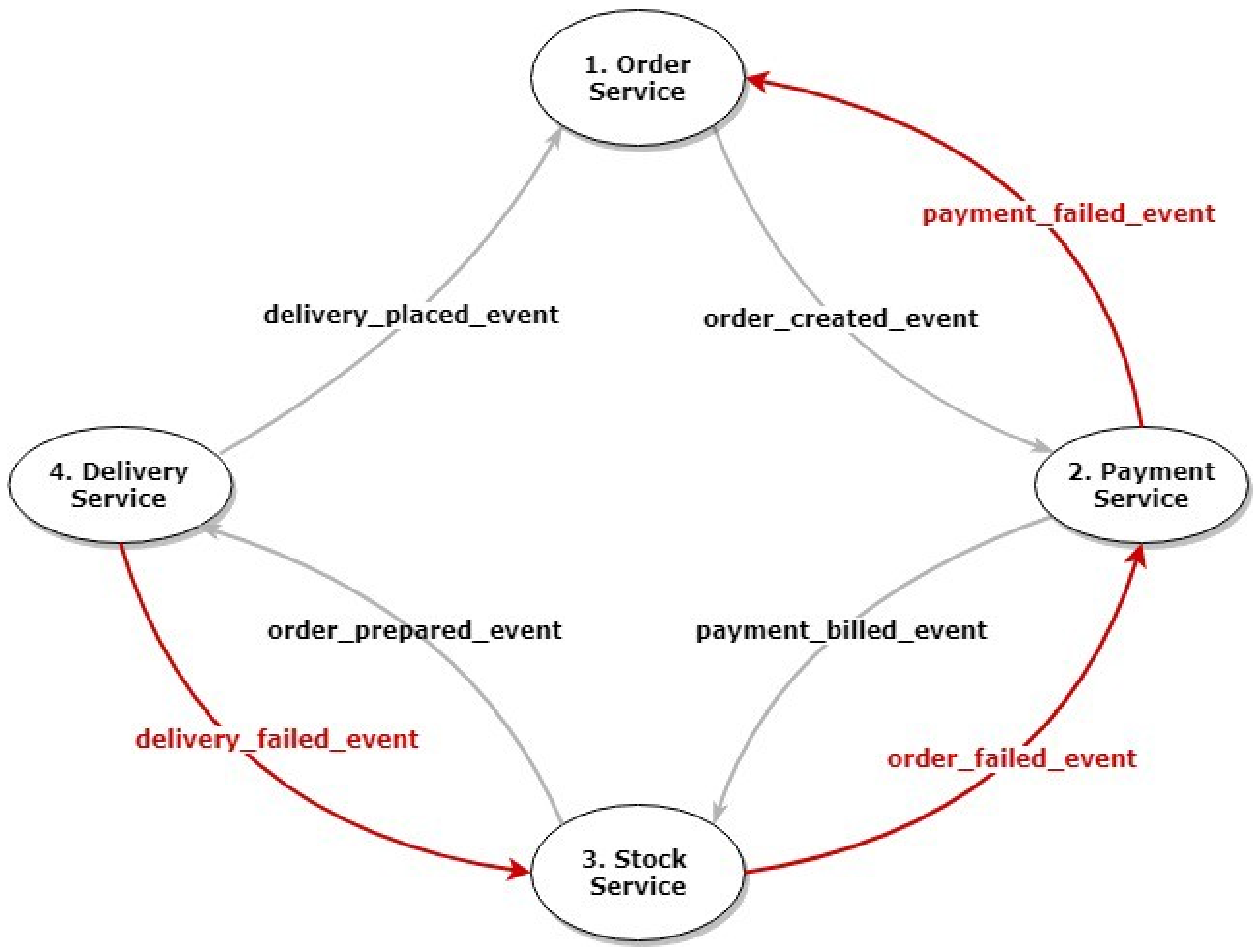


Figure 1: A sample flow of a Choreography-Based Saga Pattern implementation

1. Order Service creates the order and sends “order_created_event” into the message queue.
2. Payment Service first receives the message and creates the payment and then sends “payment_billed_event” into the message queue.
3. Stock Service receives this message from Payment Service, performs required processes and sends “stock_prepared_event” to the message queue.

4. Delivery Service runs into an error when performing this event and sends back “delivery_failed_event” message to rollback the entire process.

5. Stock Service receives this failure message and using the transaction_id provided in it, performs a compensation process and sends back “stock_failed_event”.

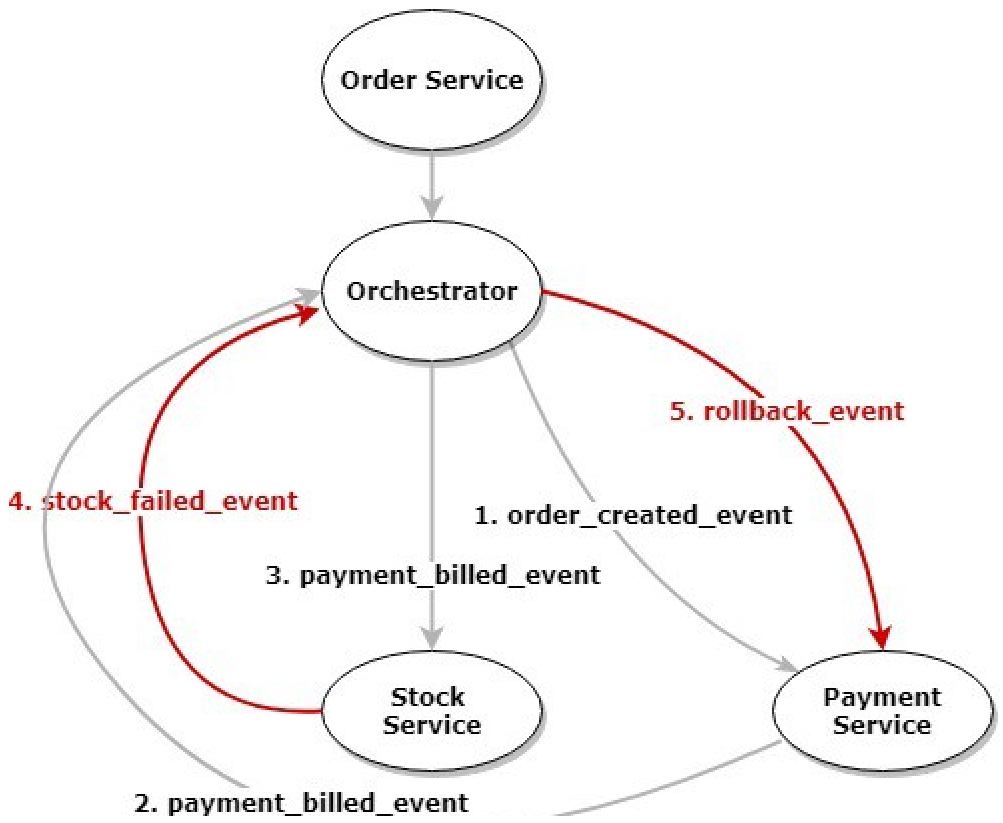
6. Finally Payment Service receives the failure message and performs its own compensation process and sends “payment_failed_event” for Order Service.

7. Order Service for such cases can use a retry mechanism with some delay and if the error still persists, it can use some warning mechanisms for a manual check.

Orchestration-Based Saga

In this approach on the other hand, there would be an orchestrator to manage the entire operation from one center. An orchestrator receives a start command from a source and begins calling related services sequentially. After each successful response, it makes the next call to the following service. If one of the steps fails and the service returns a failure message, the orchestrator makes rollback calls for each previous step/service.

As it also brings along some scalability issues and the risk of a single point of failure, developers should consider necessary recovery actions or simply go for Choreography-Based architecture if they can.



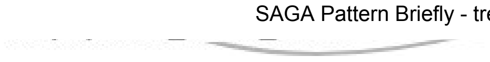


Figure 2: An example flow of an Orchestrator-Based Saga Pattern implementation

1. Order Service creates the order and employs Orchestrator.
2. Orchestrator sends “order_created_event” for Payment Service.
3. Payment Service creates the payment and sends “payment_billed_event” for Orchestrator.
4. Orchestrator this time sends “payment_billed_event” for Stock Service.
5. Stock Service runs into an error when performing this event and sends “stock_failed_event” for Orchestrator.
6. Orchestrator starts rollback cycle and sends “rollback_event” for the previous service which is Payment Service in this scenario.
7. Payment Service performs a compensation process after receiving this message.

Microservices Distributed Systems Design Patterns