Isuru Jayakantha

111 Followers About Follow

Microservices : The SAGA Pattern for distributed transactions

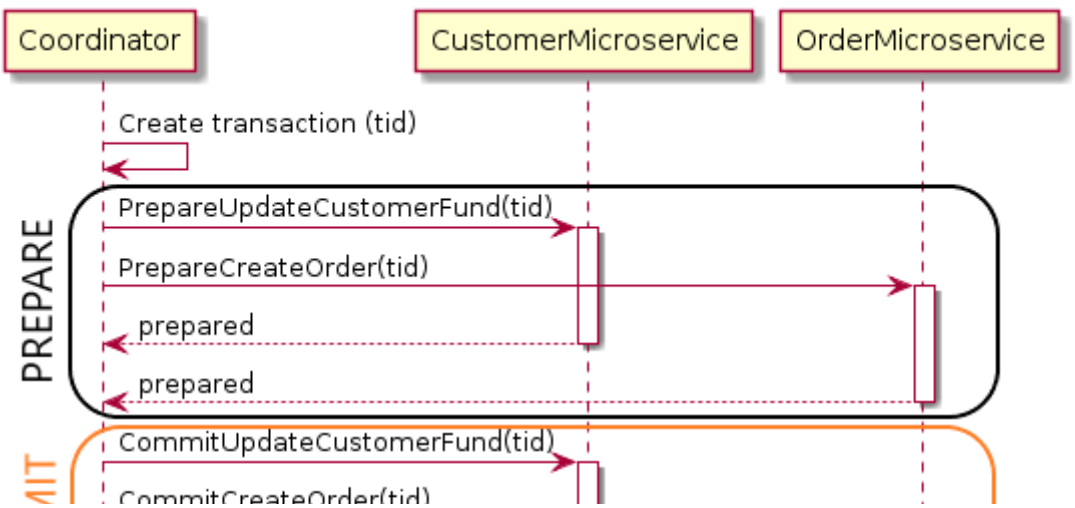
Isuru Jayakantha Aug 14, 2019 · 3 min read

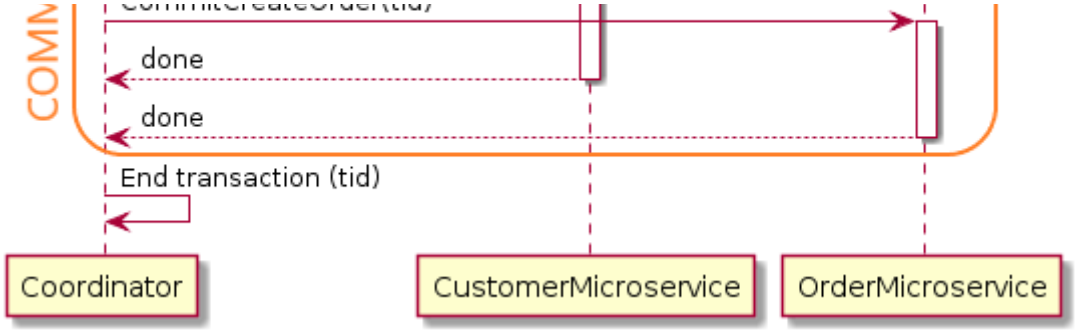
In microservice world, it is common that transactions may take long journey for it to complete. having multiple databases and communicating with multiple services makes more complex and make it difficult to keep the consistency of data. local ACID transactions also wont help if the communication happens between separate services with multiple databases.

> *How do we maintain the consistency of data across multiple services?*

Two-Phase Commit ?

Two phase commit (2PC) is a well known algorithm to achieve the benefits of ACID. Handling 2PC is not that simple. 2PC consists of two stages, one is “PREPARE” and “COMMIT”.





Example for 2PC — Create Customer Order

This solves the problem up to a certain extend, but still unable to solve below.

- There is no mechanism to rollback the other transaction if one micro service goes unavailable in commit phase.
- Others have to wait until the slowest resource finish its confirmation.

We can't just place an order, update the customer > update the stock > send delivery in a single ACID transaction. Then what are the other design patterns available to solve these problems ? Let's go back to 1987.

SAGA Pattern ?

SAGA is one of the best way to ensure the consistency of the data in a distributed architecture without having a single ACID transaction. SAGA commits multiple **compensatory transactions** at different stages ensuring to rollback when required. SAGA introduce the master process called "**Saga Execution Coordinator**" or SEC.

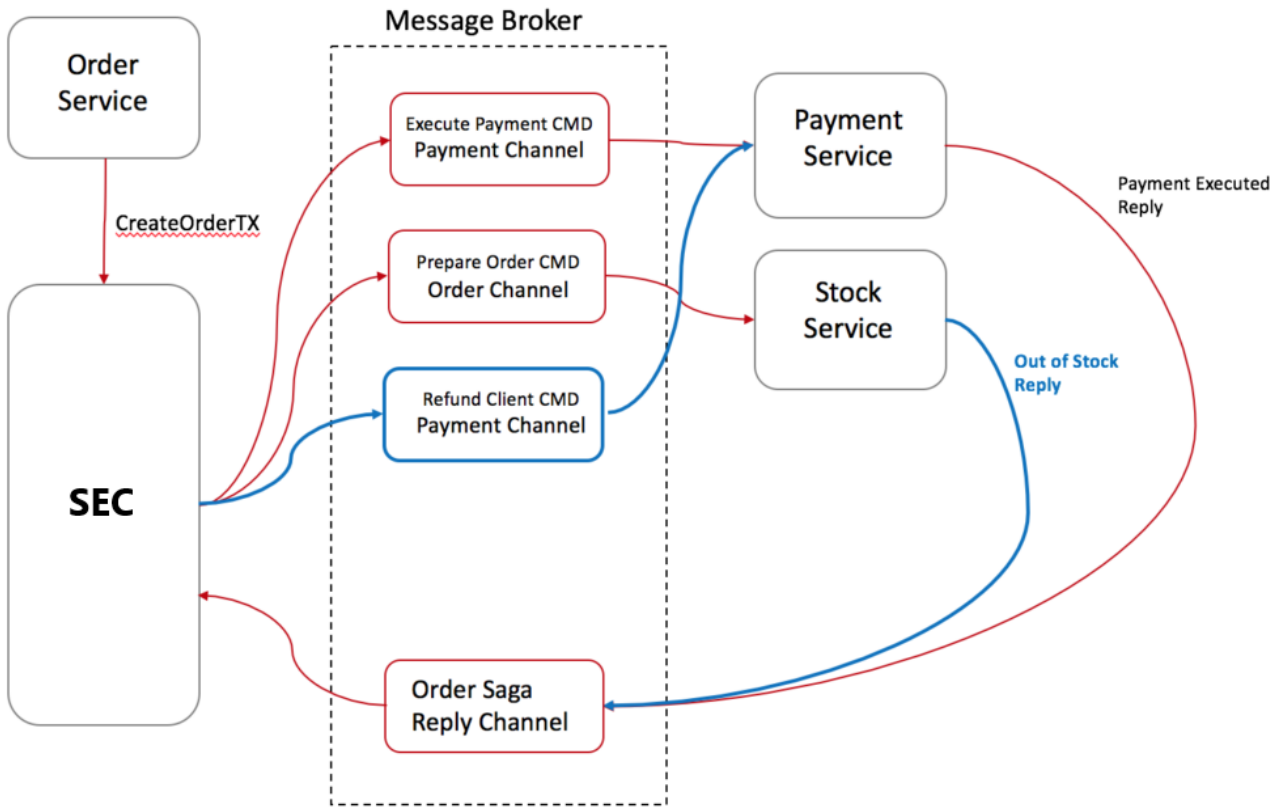
There are two ways to achieve sagas:

- Choreography : each local transaction publishes domain events that trigger local transactions in other services.
- Orchestration : an orchestrator (object) tells the participants what local transactions to execute.

Orchestration-based saga

The best option would be using an orchestration process. we define a new service (SEC) with the sole responsibility of telling each service what to do and when. SEC orchestrates sequence of commands notifying other services what operations are

required to perform. we also could use event based communication between services to accommodate resubmitting or rollback the local transactions.



Example of Orchestration-based saga for customer order creation

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

SAGA also has some disadvantages, like the SEC becomes more complex and becomes the core potion of the entire platform. Nevertheless, you will gain much more advantages over this. the most important advantage is that Rollbacks are easier to manage.

Important Tips when using SAGA

- Avoid cyclic dependencies between services. Do not let the participants to call SEC.
- Always create a Unique ID per transaction and make it flow through every local transactions which is easier for traceability and troubleshooting.
- Avoid Synchronous Operations.
- Log the request payloads for re-submission purpose.
- Make sure you capture all data required before handing over the process to SEC.

Downside the SEC becomes more complex. Therefore, beware of the trade offs.

JAVA Frameworks Available for SAGA Implementation

- Eventuate Tram Sagas : [http://eventuate.io]
- Axon framework : [http://www.axonframework.org/]
- MicroProfile LRA : [https://github.com/eclipse/microprofile-lra]

Microservices