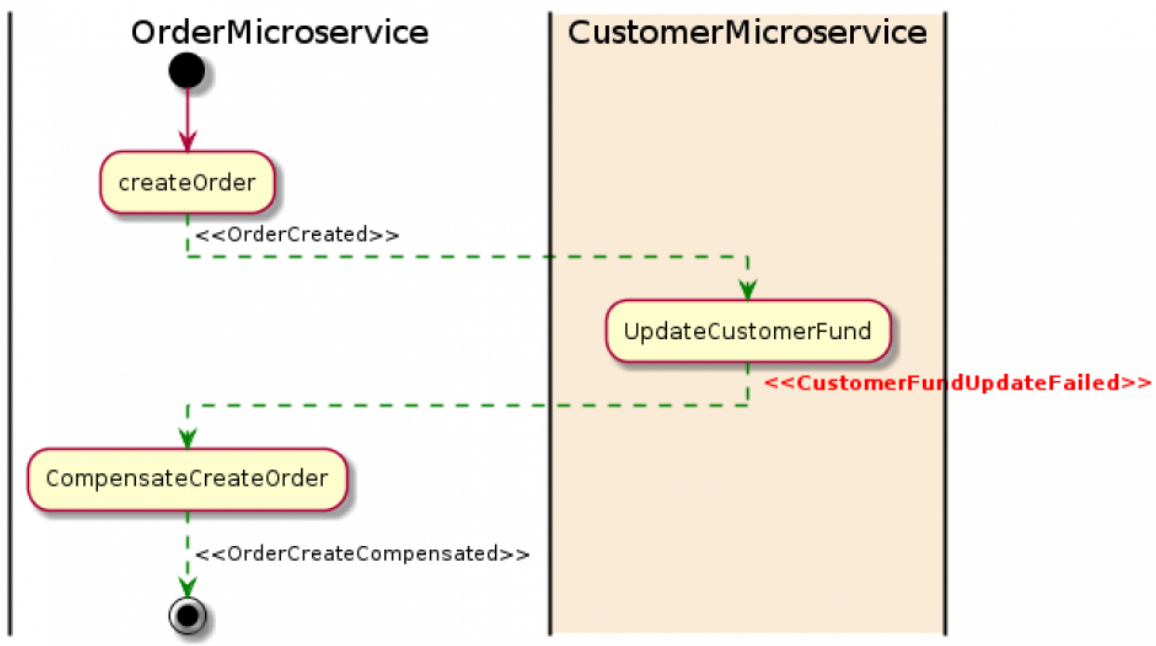
Article

Patterns for distributed transactions within a microservices architecture

October 1, 2018 Microservices, Event-Driven

Keyang Xiang
Senior Architect

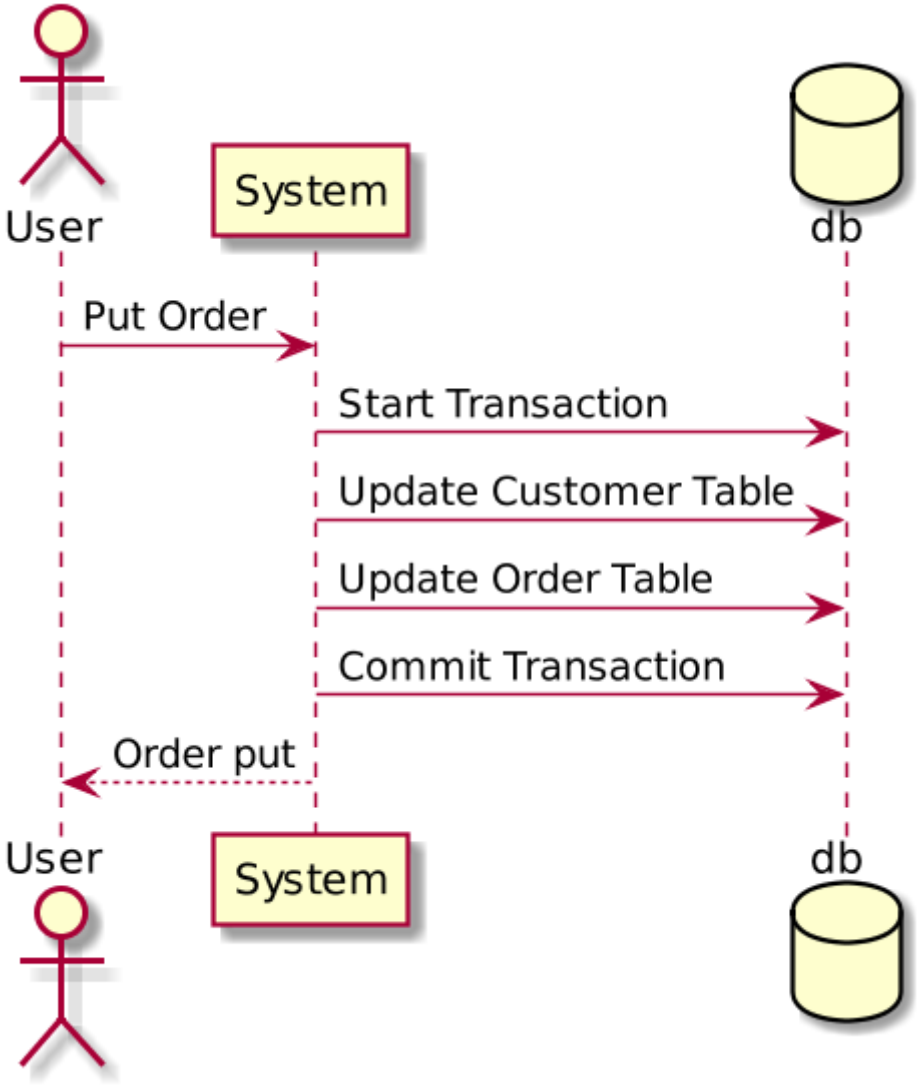


Microservices architecture (MSA) has become very popular.. However, one common problem is how to manage distributed transactions across multiple microservices. This post is going to share my experience from past projects and explain the problem and possible patterns that could solve it.

What is a distributed transaction?

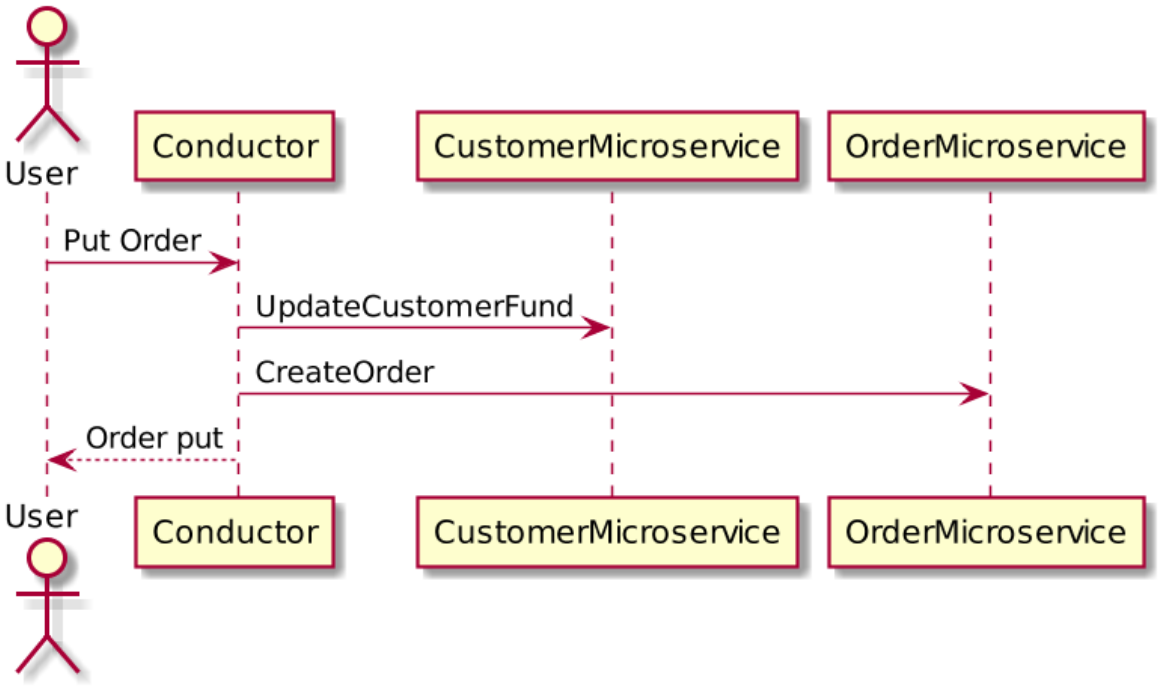
When a microservice architecture decomposes a monolithic system into self-encapsulated services, it can break transactions. This means a **local transaction** in the monolithic system is now **distributed** into multiple services that will be called in a sequence.

Here is a customer order example with a monolithic system using a local transaction:



In the customer order example above, if a user sends a **Put Order** action to a monolithic system, the system will create a local database transaction that works over multiple database tables. If any step fails, the transaction can **roll back**. This is known as ACID (Atomicity, Consistency, Isolation, Durability), which is guaranteed by the database system.

When we decompose this system, we created both the `CustomerMicroservice` and the `OrderMicroservice`, which have separate databases. Here is a customer order example with microservices:



When a **Put Order** request comes from the user, both microservices will be called to apply changes into their own database. Because the transaction is now across multiple databases, it is now considered a **distributed transaction**.

What is the problem?

In a monolithic system, we have a database system to ensure ACIDity. We now need to clarify the following key problems.

How do we keep the transaction atomic?

In a database system, atomicity means that in a transaction either **all steps complete** or **no steps complete.** The microservice-based system does not have a global transaction coordinator by default. In the example above, if the `CreateOrder` method fails, how do we roll back the changes we applied by the `CustomerMicroservice`?

Do we isolate user actions for concurrent requests?

If an object is written by a transaction and at the same time (before the transaction ends), it is read by another request, should the object return old data or updated data? In the example above, once `UpdateCustomerFund` succeeds but is still waiting for a response from `CreateOrder`, should requests for the current customer's fund return the updated amount or not?

Possible solutions

The problems above are important for microservice-based systems. Otherwise, there is no way to tell if a transaction has completed successfully. The following two patterns can resolve the problem:

- 2pc (two-phase commit)
- Saga

Two-phase commit (2pc) pattern

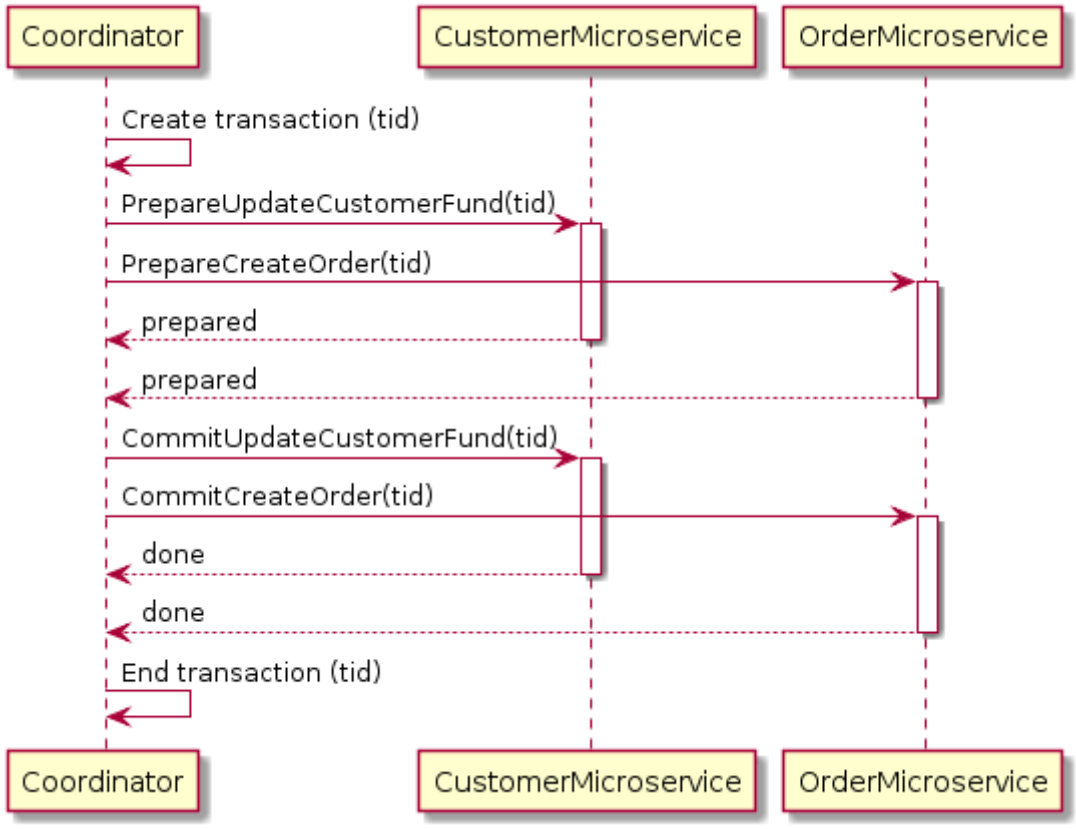
2pc is widely used in database systems. For some situations, you can use 2pc for microservices. Just be careful; not all situations suit 2pc and, in fact, 2pc is considered impractical within a microservice architecture (explained below).

So what is a two-phase commit?

As its name hints, 2pc has two phases: A *prepare phase* and a *commit phase*. In the prepare phase, all microservices will be asked to prepare for some data change that could be done atomically. Once all microservices are prepared, the commit phase will ask all the microservices to make the actual changes.

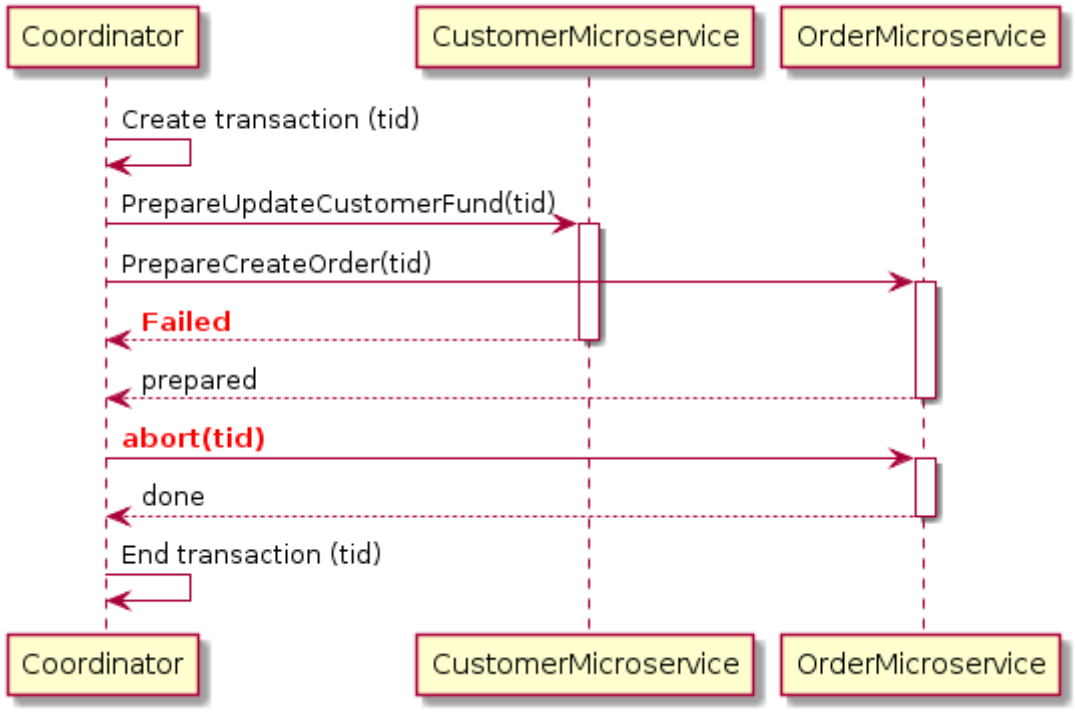
Normally, there needs to be a global coordinator to maintain the lifecycle of the transaction, and the coordinator will need to call the microservices in the prepare and commit phases.

Here is a 2pc implementation for the customer order example:



In the example above, when a user sends a put order request, the `Coordinator` will first create a global transaction with all the context information. It will then tell `CustomerMicroservice` to prepare for updating a customer fund with the created transaction. The `CustomerMicroservice` will then check, for example, if the customer has enough funds to proceed with the transaction. Once `CustomerMicroservice` is OK to perform the change, it will lock down the object from further changes and tell the `Coordinator` that it is prepared. The same thing happens while creating the order in the `OrderMicroservice`. Once the `Coordinator` has confirmed all microservices are ready to apply their changes, it will then ask them to apply their changes by requesting a commit with the transaction. At this point, all objects will be unlocked.

If at any point a single microservice fails to prepare, the `Coordinator` will abort the transaction and begin the rollback process. Here is a diagram of a 2pc rollback for the customer order example:



In the above example, the `CustomerMicroservice` failed to prepare for some reason, but the `OrderMicroservice` has replied that it is prepared to create the order. The `Coordinator` will request an abort on the `OrderMicroservice` with the transaction and the `OrderMicroservice` will then roll back any changes made and unlock the database objects.

Benefits of using 2pc

2pc is a very strong consistency protocol. First, the prepare and commit phases guarantee that the transaction is atomic. The transaction will end with either all microservices returning successfully or all microservices have nothing changed.

Secondly, 2pc allows read-write isolation. This means the changes on a field are not visible until the coordinator commits the changes.

Disadvantages of using 2pc

While 2pc has solved the problem, it is not really recommended for many

Table of contents: Conclusion

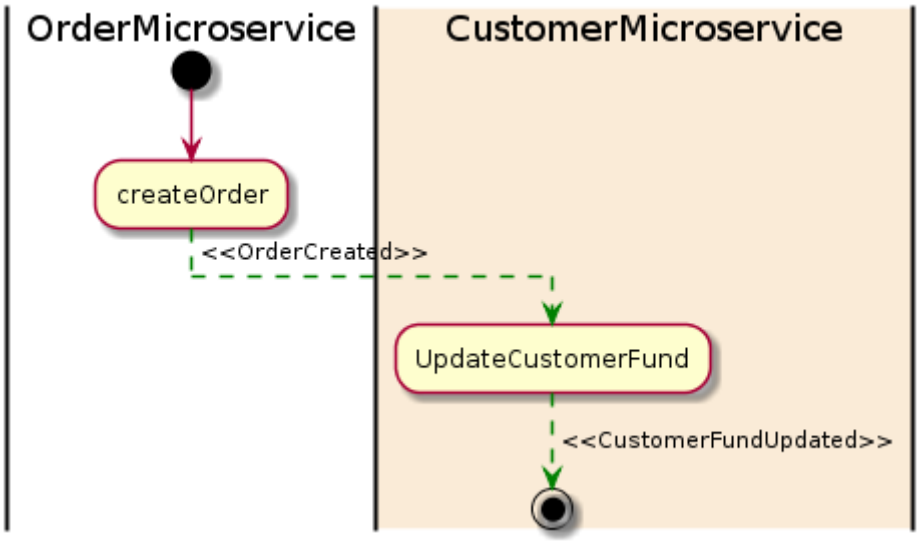
will need to lock the object that will be changed before the transaction completes. In the example above, if a customer places an order, the "fund" field will be locked for the customer. This prevents the customer from applying new orders. This makes sense because if a "prepared" object changed after it claims it is "prepared," then the commit phase could possibly not work.

This is not good. In a database system, transactions tend to be fast—normally within 50 ms. However, microservices have long delays with RPC calls, especially when integrating with external services such as a payment service. The lock could become a system performance bottleneck. Also, it is possible to have two transactions mutually lock each other (deadlock) when each transaction requests a lock on a resource the other requires.

Saga pattern

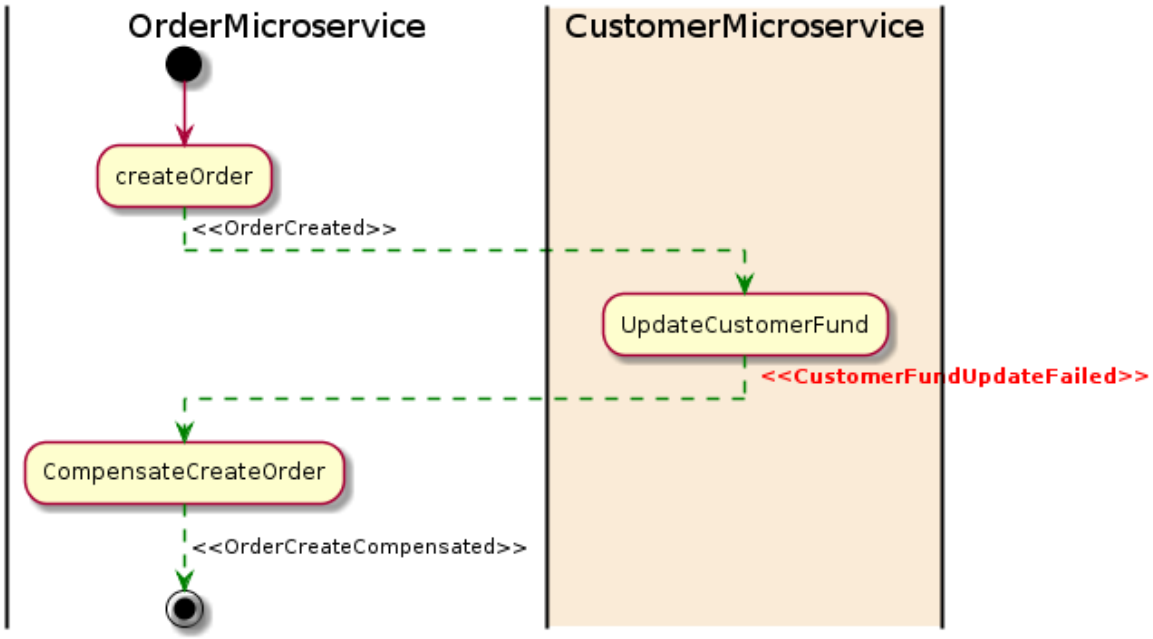
The Saga pattern is another widely used pattern for distributed transactions. It is different from 2pc, which is synchronous. The Saga pattern is asynchronous and reactive. In a Saga pattern, the distributed transaction is fulfilled by asynchronous local transactions on all related microservices. The microservices communicate with each other through an event bus.

Here is a diagram of the Saga pattern for the customer order example:



In the example above, the `OrderMicroservice` receives a request to place an order. It first starts a local transaction to create an order and then emits an `OrderCreated` event. The `CustomerMicroservice` listens for this event and updates a customer fund once the event is received. If a deduction is successfully made from a fund, a `CustomerFundUpdated` event will then be emitted, which in this example means the end of the transaction.

If any microservice fails to complete its local transaction, the other microservices will run compensation transactions to rollback the changes. Here is a diagram of the Saga pattern for a compensation transaction:



In the above example, the `UpdateCustomerFund` failed for some reason and it then emitted a `CustomerFundUpdateFailed` event. The `OrderMicroservice` listens for the event and start its compensation transaction to revert the order that was created.

Advantages of the Saga pattern

One big advantage of the Saga pattern is its 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. This also allows transactions to continue waiting for user input. Also, because all local transactions are happening in parallel, there is no lock on any object.

Disadvantages of the Saga pattern

The Saga pattern is difficult to debug, especially when many microservices are involved. Also, the event messages could become difficult to maintain if the system gets complex. Another disadvantage of 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 due to a compensation transaction.

Adding a process manager

To address the complexity issue of the Saga pattern, it is quite normal to add a process manager as an orchestrator. The process manager is responsible for listening to events and triggering endpoints.

Conclusion

The Saga pattern is a preferable way of solving distributed transaction problems for a microservice-based architecture. However, it also introduces a new set of problems, such as how to atomically update the database and emit an event. Adoption of the Saga pattern requires a change in mindset for both development and testing. It could be a challenge for a team that is not familiar with this pattern. There are many variants that simplify its implementation. Therefore, it is important to choose the proper way to implement it for a project.

Last updated: October 10, 2019