Distributed Transactions Patterns In Microservice Architecture

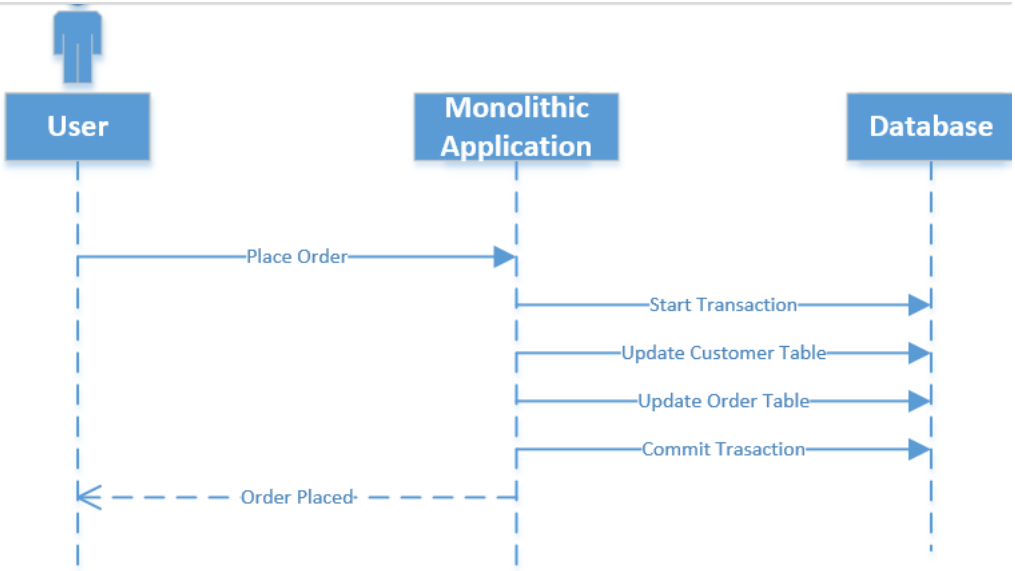

Hasan Shahjahan | Follow
Senior Backend Engineer at eatig...

While distributed transactions are important to microservices, they are often misunderstood. In this article, we look to iron out these misunderstandings.

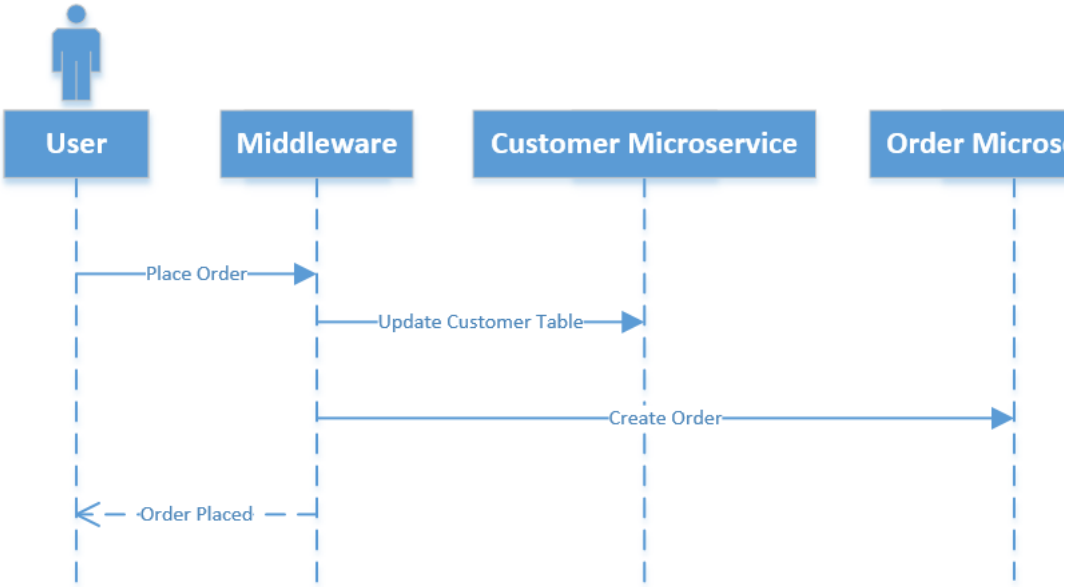
Microservices architecture is very popular in recent days to build big systems or applications. However, in this architecture, the most common problem is how to manage distributed transactions across multiple microservices. Here is a brief article on my real-time experience through my project, the actual problem and the possible solutions(patterns) that could solve it.

What is a distributed transaction?

A distributed transaction is a type of transaction with two or more engaged network hosts. Generally, hosts provide resources, and a transaction manager is responsible for developing and handling the transaction.This means that all transactions in the monolithic system are now distributed into multiple services.Here is a customer order example with a monolithic system:



In the above example if a user sends an **Order** action to a monolithic system, the system will create database trasaction 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 decomposing above monolithic system into microservice, we created 2 services **CustomerMicroservice** and **OrderMicroservice**. Both of these have separate databases. Here is a customer order example with microservices:



When an **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 are the Major Problems in Microservices?

To ensure ACID principle (Atomicity, Consistency, Isolation, Durability) in monolithic system, we have database sytem. But in microservice, we need to clarify the following key problems in ACIDity.

have a global transaction coordinator by default. In the microservice example above, if the CreateOrder method fails, how do we roll back the changes we applied by the CustomerMicroservice?

Keep transactions isolate - Before completing all the distributed transaction in microservice, if the user sends another request, should the object return old data or the updated one?. In the above microservice example, Update Customer table succeeds. But it is still waiting for a response from CreateOrder service. Should requests for the current customer's table return the updated amount or not?

Ways to Solve the Major Microservices Problems

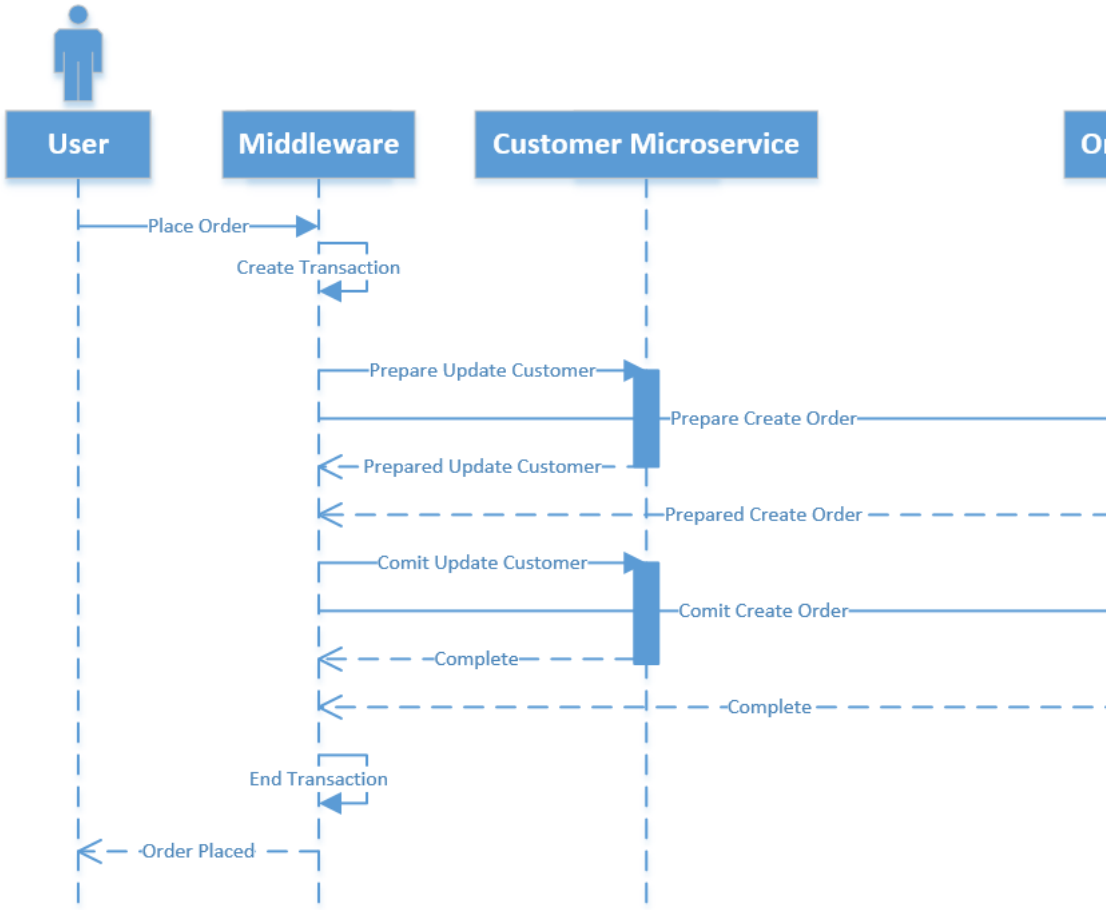
The following two patterns can resolve the above problem:
Two-phase commit pattern (2PC)
Saga pattern (3PC)

Two-phase commit (2PC) pattern:

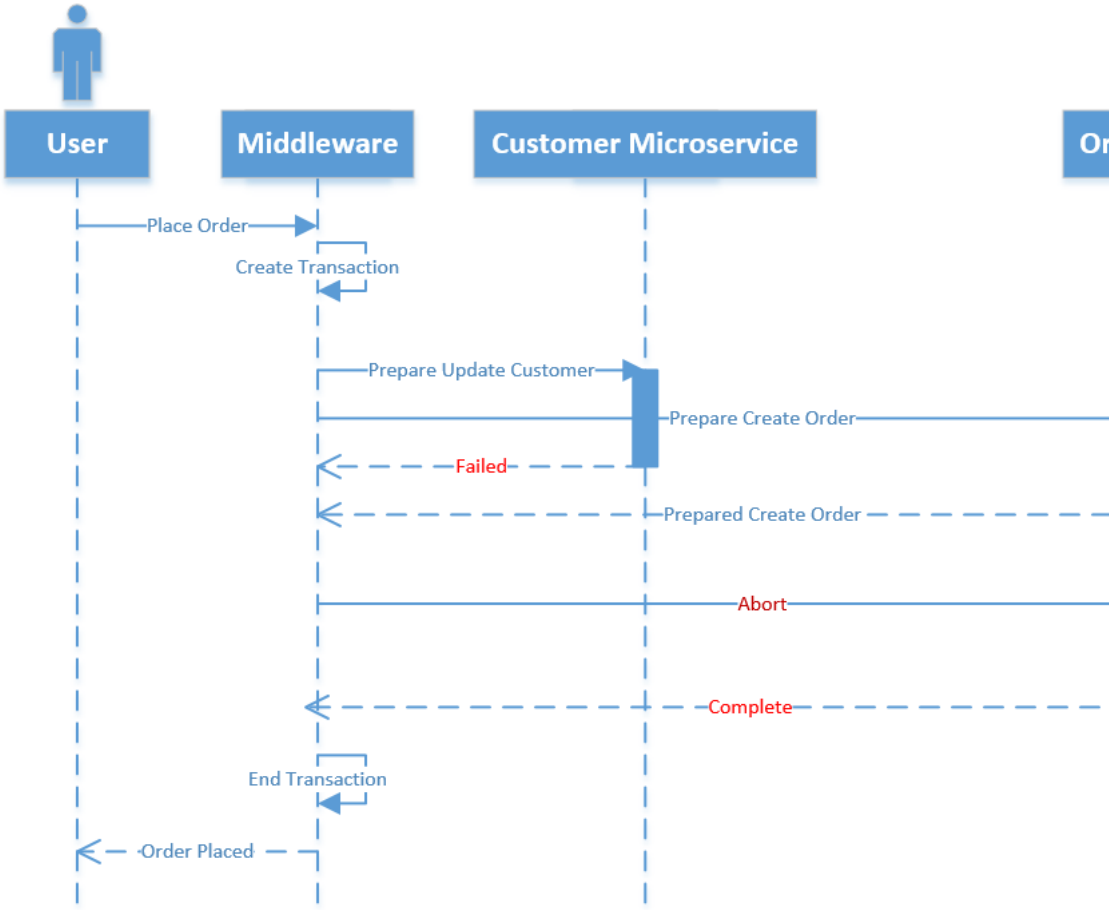
2PC is widely used in database systems. For some situations, you can use 2PC for microservices. 2PC has two phases:

Prepare Phase - All the microservices will ask you to prepare for some data change that could be done atomically.

Commit Phase - Once all microservices are prepared for data change, the commit phase will ask all the microservices to make the actual changes in DB.



In the example above, when a user sends an **Order** request, the Middleware or Coordinator will first create a global transaction (Create Transaction) with all the context information. It will then tell **CustomerMicroservice** to prepare for updating a customer table 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 Middleware that it is prepared. The same thing happens while creating the order in the **OrderMicroservice**. Once the Middleware 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. Once both phases are completed, records will be unlocked.



In the above sequence diagram, the Customer microservice fails to prepare for some reason, but the Order microservice prepares to create the order. The Middleware will ask to abort all the prepared transactions i.e roll back changes and unlock the records.

Advantages of 2PC

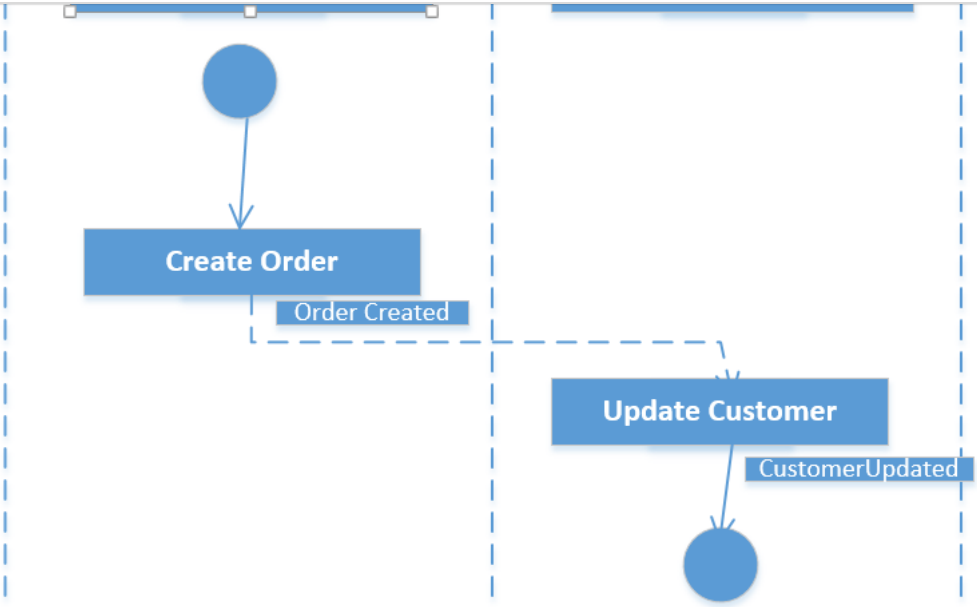
More atomic(A)
Data consistency(C)
Read-write Isolation(I)
Durability(D)

Disadvantages of 2PC

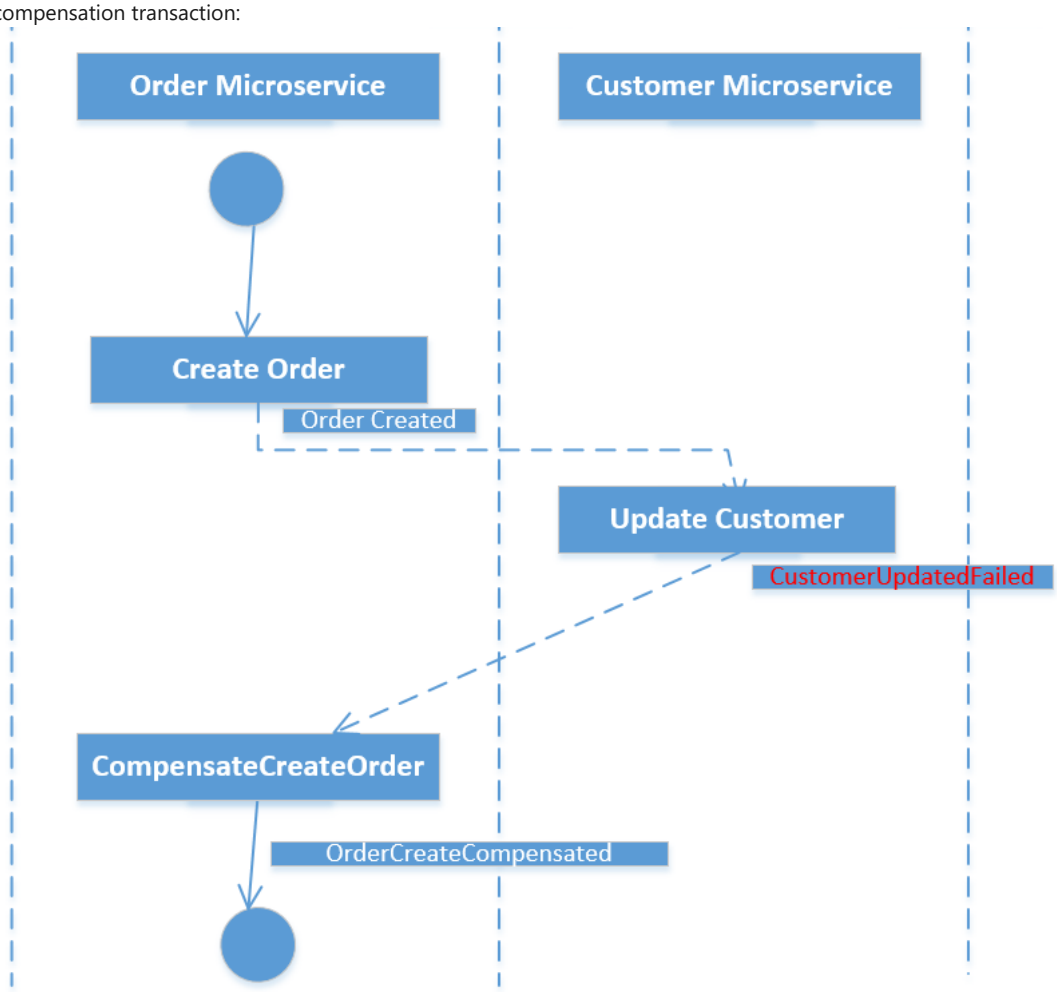
Request is Synchronous (Blocking)
Lock records/object until transaction completes.
Possibility of Deadlock between transactions.

SAGA (3PC) Pattern:

For distributed transactions, The Saga pattern is another widely used pattern.The Saga pattern is asynchronous and more reactive. It is different from 2PC, which is synchronous.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 CustomerUpdated 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 UpdateCustomer failed for some reason and it then emitted a **CustomerUpdateFailed** event. The OrderMicroservice listens for the event and start its compensation transaction to revert the order that was created.

Request are asynchronous
No lock for DB object

Disadvantages of the Saga pattern

Difficult to debug/test especially if we have multiple microservices
It does not have read isolation i.e user can see the order created but in next second, the order is removed due to failure transaction

Conclusion

The Saga pattern is the most preferable way of solving distributed transaction problems for a microservice-based architecture. As a team if we have to adopt the Saga pattern, it requires a change in mindset for both development and testing. Sometimes It could be a challenge for a team that is not familiar with this pattern. Therefore, it is very important to choose the proper way to implement it for a project.

Published By

Hasan Shahjahan
Senior Backend Engineer at eatigo (...

Follow

#paymentprocessing #paymentsolutions #onlinepayments #financialservices #microservices #softwarearchitecture

4 comments

Sign in to leave your comment

Pragnesh Patel
Azure | M365 | SharePoint | Power Platform Solution Architect at Zensar Technologies

4d ···

Very well explained.

Like Reply | 1 Like 1 Reply

Hasan Shahjahan
Senior Backend Engineer at eatigo (Tripadvisor-backed)

4d

Thank you very much for appreciating.

Like Reply

Abebe Hailu
Lead Enterprise Architect bei Mercedes-Benz AG

2mo

So well explained. Thanks!

Like Reply | 1 Like 1 Reply

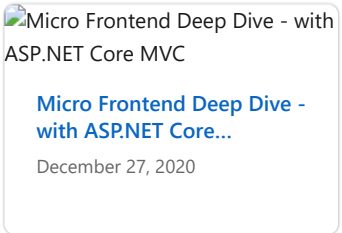
Hasan Shahjahan
Senior Backend Engineer at eatigo (Tripadvisor-backed)

4d

Thanks for appreciating.

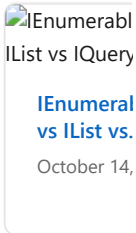
Like Reply

More from Hasan Shahjahan 11 articles



Micro Frontend Deep Dive - with ASP.NET Core...
December 27, 2020

Building Micro Frontends
December 26, 2020



IEnumeral vs IList vs.
October 14,