Microservices Patterns: The Saga Pattern

Microservices Pattern Series — Part 04: Some insights into ACID Transactions, 2PC Pattern and finally the Saga Pattern in detail.

Crishantha Nanayakkara

Objective

Transaction handling is an integral part of any enterprise architecture. This allows for building proper data consistency across enterprise applications. As an Enterprise Architect, knowing how to handle transactions in a distributed architecture is essential.

This blog will give the reader some insights into ACID Transactions, 2PC Pattern and finally the Saga Pattern in detail. The primary objective is to give an overview of how Saga Pattern operates and how it can solve real-world distributed transactional issues.

The blog was written with some background reading on multiple literature resources mentioned under the *References*. If you really want to go into detail about this subject I highly recommend reading the book *Microservices Patterns: with Examples in Java — By Chris Richardson* [3].

ACID Transactions

In a traditional monolithic application, the system will create a local database transaction that works over multiple database tables. If any error occurs in between a transaction, it will be rolled back to its initial state. These transactions are known as ACID transactions (Atomicity, Consistency, Isolation, Durability). ACID transactions greatly ease the developer's task by having exclusive access to a

particular database. You could see this in many monolithic application architectures, which have a simple way of handling transactions with a single database. In Figure 01, you could see how both the Customer and the Order tables are updated via a single atomic ACID transaction.

Figure 01 — An ACID Transaction in a Monolithic Application

Distributed Transactions

Unlike single monolithic applications, distributed applications are dealing with multiple services. In such architectures, handling transactions could be a challenge.

Two-Phase Commit (2PC) is one of the distributed transaction strategies that we could apply.

Two-Phase Commit (2PC) Pattern

> *The 2PC Pattern is all about updating resources on multiple nodes in a single atomic operation.*

In 2PC, it carries out an update in two phases (Figure 02).

1. **Prepare**: Each node, participating in the transaction, whether it is able to carry out an update in the second phase. Once each node is able to ensure it, the coordinator will be notified. If any of the nodes are unable to make it, the coordinator is notified to roll back releasing any locks they have with nodes.
2. **Commit**: Carrying out the update and completing the transaction.

Figure 02–2PC Transaction on a Microservices Setup

2PC represents a synchronous strong consistency approach within a distributed transaction.

However, 2PC is not fully recommended for microservices-based applications due to its **synchronous blocking**. This protocol will need to block the required object that will be changed before the transaction completes. This prevents the relevant object from being used by a different transaction (deadlock situation) until the ongoing transaction is fully completed. This is not a good situation,

especially in a modern-day application.

Furthermore, it is not supported by many modern-day message brokers such as RabitMQ and Apache Kafka. In addition to that, some of the popular databases such as MongoDB and Cassandra are also not supported.

The Saga Pattern

As explained above, **your business logic can use ACID transactions within services. However, it must use Saga Pattern in order to maintain data consistency across services.**

> *Pattern: Maintain data consistency across services using a sequence of local transactions that are coordinated using asynchronous messaging [2].*

A Saga is a sequence of local transactions. Each local transaction updates the local database using the familiar ACID transaction frameworks and publishes an event to trigger the next local transaction in the Saga. If a local transaction fails, then the Saga executes a series of ***compensating*** *transactions* that undo the changes, which were completed by the preceding local transactions (Figure 03).

Figure 03 — Saga

It is an *asynchronous* and *eventually consistent* transactional approach, which is quite analogous to a typical microservices application architecture, where a distributed transaction is fulfilled by a set of asynchronous transactions on related microservices.

An important benefit of the asynchronous nature of a Saga message is that it ensures that all steps of a Saga are executed, even if one or more of the Saga's participants is temporarily unavailable. In addition to that, it can support long-lived transactions without blocking any other microservice or an object in the process.

However, Saga has its own weaknesses as well. Some of the key challenges are,

- The lack of isolation between Sagas.
- Rolling back changes when an error happens within a Saga.

Saga Coordination

Sagas can be implemented in "two ways" primarily based on the logic that coordinates the steps of the Saga.

1. **Choreography based sagas**
2. **Orchestration based sagas**

> *In a choreography based saga, a local transaction publishes events that trigger other participants to execute local transactions. In an orchestrated-based saga, a centralized saga orchestrator sends command messages to saga participants telling them to execute local transactions. [3].*
>
> *Generally in practice, simple sagas can leverage the concept of choreography and for complex sagas, it is recommended to use the orchestrated version. [3]*

Choreography based Sagas

In this approach, unlike orchestrator-based saga, there is no central coordinator to tell saga participants what to do. Saga participants subscribe to each other's events and respond accordingly (Figure 04).

- Interaction Type: Publisher / Subscriber Messaging

Figure 04 — Choreography based saga

In Figure 04, each saga participant (here it is the microservice) communicates with each other by exchanging events. In this diagram,

- Event Topic 1 is subscribed to Microservice 2 and 3
- Event Topic 2 is subscribed to Microservice 3
- Event Topic 3 is subscribed to Microservice 1 and 2

Each saga participant updates its local database and publishes an event that triggers the next participant. For example, in Figure 04, once microsrevice 01 completes its local database update, it publishes an event to *Event Topic 1* in the message broker. Saga participants, which are subscribed to *Event Topic 1* are now triggered to execute any other actions within the microservice and publish further events (if any) to connect with other saga participants. Likewise, the flow can continue until the saga is fully completed.

Simplicity and *Loose coupling* are some of the key benefits of using choreography-based sagas. However, choreography-based sagas are

coupled with some significant drawbacks as well. Some of the key drawbacks are:

- *Difficulty in understanding the flow* — Generally this type distributes the implementation of the saga among the services. In this method, there is no centralized place for it to define the flow of the saga.
- *Cyclic dependencies between services* — Saga participants have the possibility of having cyclic dependencies such as *microservice 01 -> microservice 02 -> microservice 01*
- *Risk of tight coupling* -> Having to subscribe to all events that affect them could end up having a tight coupling among services.

Due to the above drawbacks and how it is being architect, it is advisable to say that choreography-based sagas are fine with a few services but with complex services setup.

Orchestration based sagas

In this type, a central Saga orchestration class is responsible to tell saga participants what to do.

- Interaction Type: Asynchronous Request/ Response

Here, an Asynchronous Request also names a “Command Message”

Figure 05 — Orchestrator based saga

As you see in Figure 05, the “Saga Orchestrator” which is implemented within Microservice 01, initiates the saga transaction (e.g. Create Order Saga). As shown above, the interaction style is asynchronous request/ response and the request is forwarded as a “command message”. Since it uses the asynchronous request/ response style, it does use separate request and response channels within the message broker.

In Figure 05, when microservice 01 receives the create() request, it creates the saga orchestrator. This will set the service request to the PENDING state until the saga orchestrator gets the complete approval. Meantime, the saga orchestrator sends request commands to both microservice 2 and 3 and gets the responses back via the saga orchestrator response channel in the message broker. Based on the outcome of both responses, the orchestrator will approve or reject the request.

Some of the key benefits are,

- *Simpler dependencies* — Always orchestrator invokes saga participants not vice versa. Therefore, there are no cyclic

dependencies.
- *Less coupling* — Unlike a choreography-based saga, it does not need to know about events published or the business logic implemented by other saga participants. This improves coupling and greatly simplifies the business logic.

There are a few drawbacks as well.

- *Less business logic at the saga orchestrator level* — Try not to have business logic within the saga orchestrator allowing to have more decoupled architecture. Having business logic outside of the relevant service is “not” recommended as it is advised not to load business logic in the saga orchestrator.
- *Less Isolation* — As a whole, microservices architecture lacks *Isolation* compared to traditional ACID transactions. This is due to the fact that saga participants do commit changes as local transactions before completing the whole transaction. As a result at the database level *anomalies* could happen.

Anomalies

There are three types of anomalies found in a typical saga.

1. **Lost Updates** — One saga overwrites an update made by another saga.
2. **Dirty Reads** — One saga reads data that is in the middle of being updated by another saga.
3. **Fuzzy / Non-repeatable Reads** — Two different sets of a saga read the same data and get different results because another saga has made updates.

Out of these three, *lost update* and *dirty read* scenarios are the most common.

In order to rectify the anomalies, it is required to implement countermeasures in your designs. There are multiple

countermeasure approaches in the literature and some of the important ones are as follows.

1. **Semantic Lock** — This is an application-level lock, in which saga's compensable transactions set a flag (e.g. Creating an Order can have flag status such as APPROVAL_PENDING, REVISION_PENDING, etc.) in any record that it creates or updates. This flag indicates that the record is not committed and that it has the potential to change. This could be cleared by a retriable transaction or a compensating transaction.
2. **Commutative Updates** — Designing the system to have more its update operations to be commutative (updates in an orderly manner). This can basically eliminate *lost updates*.
3. **Pessimistic View** — Reordering saga participants/services to minimize the effect of dirty reads.
4. **Reread Values** — This countermeasure reread values before updating it to further to re-verify the values are unchanged during the process. This will minimize *lost updates*.
5. **By Value** — This strategy will select concurrency mechanisms based on the business risk. This can help to execute low-risk requests using sagas and execute high-risk requests using distributed transactions.

Conclusion

That's it! Hope you got some good insight into the Saga Pattern and other related concepts by reading this blog. You can find my other blogs related to Microservices Patterns and Concepts using the following links.

1. Microservices Patterns: Inter Process Communication
2. Microservices Patterns: The Circuit Breaker Pattern
3. Microservice Patterns: The Service Discovery Patterns

Happy coding!

References

1. Two-Phase Commit: https://martinfowler.com/articles/patterns-of-distributed-systems/two-phase-commit.html
2. The Saga Pattern: https://microservices.io/patterns/data/saga.html
3. Microservices Patterns: with Examples in Java [Book] — By Chris Richardson, Manning Publications, 2018