Data management patterns in microservices architecture

Giovanni Jiayi Hu
Department of Mathematics
University of Padua, Italy I-35121
Email: giovannijiayi.hu@studenti.unipd.it

***Abstract*—A microservice architecture is a service-oriented architecture composed of loosely coupled elements that have bounded contexts. Each service has a focused, cohesive set of responsibilities. It comes then with no surprise that a key characteristic of the architecture is that the services are loosely coupled and communicate only via APIs. To achieve loose coupling, each service is required to to have its own datastore. However, this introduces a new range of obstacles. We must, in fact, implement transactions that work across multiple distributed services. An operation that spans services must indeed use what's known as a saga, a message-driven sequence of local transactions, to maintain data consistency. However, sagas lack the isolation feature of traditional ACID transactions. As a result, an application must use what are known as countermeasures, design techniques that prevent or reduce the impact of concurrency anomalies caused by the lack of isolation. On top of that, writing queries in a microservice architecture reveals to be likewise challenging. Implementing queries in the existing monolithic application is relatively straightforward because it has a single database, but in a microservice architecture queries often need to retrieve data that are scattered among the databases owned by multiple services. Two different patterns are presented for implementing query operations: the more straightforward API composition pattern and the more powerful Command query responsibility segregation (CQRS) pattern.**

I. Introduction

A microservice is as a small self-contained application that has a single responsibility, a lightweight stack, and can be deployed, scaled and tested independently. [1] The central aspect of this definition is the independence, the ability for a service to change without affecting any other. In order to achieve this praised independence, the database architecture must respect the following requirements:

- Services must be loosely coupled so that they can be developed, deployed and scaled independently
- Some business transactions must enforce invariants that span multiple services. For instance, before placing an order you must verify that a new order will not exceed the customer's credit limit.
- Some business transactions need to query data that is owned by multiple services. For example, displying the orders require querying the Orders service to gather the cost and the Restaurant service to know if an order has been accepted
- Databases must be replicated and sharded in order to scale.
- Different services have different data storage requirements. For some services, a relational database is the best choice. Other services might need a NoSQL database such as MongoDB, which is good at storing complex, unstructured data.

A common solution to these requirements is to keep each microservice's persistent data private to that service and accessible only via its API. The transactions of a service involve only its database. Using a database per service ensures that the services are loosely coupled and guarantees higher resilience. Changes to one service's database does not impact any other services. Besides each service can use the type of database that is best suited to its needs. For example, a service that does text searches could use ElasticSearch. [2]

However using a database per service introduces severe drawbacks which must be carefully evaluated: implementing business transactions that span multiple services is no more straightforward and implementing queries which join data that is now in multiple databases is challenging.

II. Distributed transactions

ACID (Atomicity, Consistency, Isolation, Durability) transactions greatly simplify the job of the developer by providing the illusion that each transaction has exclusive access to the data. In a microservice architecture, transactions that are within a single service can still use ACID transactions. The challenge, however, lies in implementing transactions for operations that update data owned by multiple services.

In many ways, the biggest obstacle that developers will face when adopting microservices is moving from a single database with ACID transactions to a multi-database architecture with ACD sagas. They're used to the simplicity of the ACID transaction model. [3]

The traditional approach to maintaining data consistency across multiple services, databases, or message brokers is to use distributed transactions. The de facto standard for distributed transaction management is the X/Open Distributed Transaction Processing (DTP) Model[1], which uses two-phase commit (2PC) to ensure that all participants in a transaction either commit or rollback.

Several problems arise however when a 2PC protocol is used in a system where failures occur. For instance, the coordinator

[1] https://en.wikipedia.org/wiki/X/Open_XA

as well as the participants have states in which they block waiting for incoming messages. [4]

Nevertheless the main problem with traditional distributed transactions is that they reduce availability. In order for a distributed transaction to commit, all the participating services must be available.

The availability is the product of the availability of all of the participants in the transaction. If a distributed transaction involves two services that are 99.5% available, then the overall availability is 99%, which is significantly less. Each additional service involved in a distributed transaction further reduces availability.

On top of that, the famous CAP theorem provides another proof of the unsuitability of distributed transactions:

> It is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees: consistency, availability, partition tolerance.
>
> Eric Brewer

Many systems nowadays drop consistency in favor of availability, settling for eventual consistency. Inconsistency can be tolerated for two reasons: for improving read and write performance under highly concurrent conditions and for handling partition cases where a majority model would render part of the system unavailable even though the nodes are up and running. [5]

It is clear at this point that to solve the complex problem of maintaining data consistency in a microservice architecture, an application must use a different mechanism that builds on top of the concept of loosely coupled, asynchronous services. This is where the Saga pattern comes in.

III. The Saga pattern

A saga is a sequence of local transactions. Each local transaction updates data within a single service using the familiar ACID transaction.

A saga is initiated by a command event and it reacts to subsequent command events, generating new ones and thus allowing command handlers to be kept independent.

While handling a request, the service doesn't synchronously interact with any other services. Instead, it asynchronously sends messages to other services which are not required to be available at the same time. Eventually, any unavailable service will come back up and process queued messages.

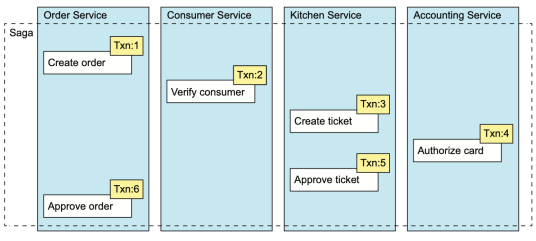

Fig. 1. Creating an Order using a saga

The example saga used throughout this section is the `Create Order Saga`, which is shown in Figure 1. The saga's first local transaction is initiated by a command event to create an order. The other five local transactions are each triggered by command events signaling the completion of the previous one.

This saga consists of the following local transactions:

1) Order Service: Create an Order in an `APPROVAL_PENDING` state;
2) Consumer Service: Verify that the consumer can place an order;
3) Kitchen Service: Validate order details and create a Ticket in the CREATE `_PENDING`;
4) Accounting Service: Authorize consumer's credit card;
5) Kitchen Service: Change the state of the Ticket to `AWAITING_ACCEPTANCE`;
6) Order Service: Change the state of the Order to APPROVED.

A. Compensating transactions

A great feature of traditional ACID transactions is that the business logic can easily roll back a transaction if it detects the violation of a business rule. Unfortunately, sagas can't be automatically rolled back, because each step commits its changes to the local database.

To see how compensating transactions are used, imagine a scenario where the authorization of the consumer's credit card fails. In this scenario, the saga executes the following local transactions:

1) Order Service: Create an Order in an `APPROVAL_PENDING` state;
2) Consumer Service: Verify that the consumer can place an order;
3) Kitchen Service: Validate order details and create a Ticket in the `CREATE_PENDING` state;
4) Accounting Service: Authorize consumer's credit card, which fails;
5) Kitchen Service: Change the state of the Ticket to `CREATE_REJECTE`D;
6) Order Service: Change the state of the Order to `REJECTED`.

The fifth and sixth steps are compensating transactions that undo the updates made by Kitchen Service and Order Service, respectively.

A saga's coordination logic is responsible for sequencing the execution of forward and compensating transactions. [3]

B. Lack of isolation

The I in ACID stands for isolation. The isolation property of ACID transactions ensures that the outcome of executing multiple transactions concurrently is the same as if they were executed in some serial order. The database provides the illusion that each ACID transaction has exclusive access to the data.

The challenge with using sagas is that they lack the isolation property of ACID transactions. That's because the updates made by each of a saga's local transactions are immediately visible to other sagas once that transaction commits. This

behavior can cause two problems. First, other sagas can change the data accessed by the saga while it's executing. And other sagas can read its data before the saga has completed its updates, and consequently can be exposed to inconsistent data.

This lack of isolation potentially causes what the database literature calls *anomalies*. An anomaly is when a transaction reads or writes data in a way that it wouldn't if transactions were executed one at time. When an anomaly occurs, the outcome of executing sagas concurrently is different than if they were executed serially.

The lack of isolation can cause the following three anomalies [3]:

1) *Lost updates*: One saga overwrites without reading changes made by another saga;
2) *Dirty reads*: A transaction or a saga reads the updates made by a saga that has not yet completed those updates;
3) *Fuzzy/non-repeatable reads*: Two different steps of a saga read the same data and get different results because another saga has made updates;

It's the responsibility of the developer to adopt a set of countermeasures for handling anomalies caused by lack of isolation that either prevent one or more anomalies or minimize their impact on the business. [6]

C. Reliable events

It's essential that the database update and the publishing of the event happen atomically. Consequently, to communicate reliably, the saga participants must use transactional messaging. The second issue you need to consider is ensuring that a saga participant must be able to map each event that it receives to its own data. The solution is for a saga participant to publish events containing a correlation id, which is data that enables other participants to perform the mapping.

IV. Communication

The microservice architecture structures an application as a set of services which must often collaborate in order to handle a request. Because service instances are typically processes running on multiple machines, they must interact using Inter-Process Communication (IPC).

The choice of IPC mechanism is an important architectural decision since it can impact application availability and it intersects with transaction management. As we have seen with the saga pattern, we favor loosely coupled services that communicate with one another using asynchronous messaging.

Synchronous protocols such as REST and RPC are used mostly to communicate with other applications. The problem with REST is that it's a synchronous protocol: an HTTP client must wait for the service to send a response. Whenever services communicate using a synchronous protocol, the availability of the application is reduced.

A. Message Queuing

The basic idea behind a message-queuing system is that applications communicate by inserting messages in specific queues.

An important aspect of message-queuing systems is that a sender is generally given only the guarantees that its message will eventually be inserted in the recipient's queue. No guarantees are given about when, or even if the message will actually be read, which is completely determined by the behavior of the recipient.

These semantics permit communication to be loosely coupled in time. When selecting a message broker, you have various factors to consider, including the following:

- Messaging ordering: Does the message broker preserve ordering of messages?
- Delivery guarantees: What kind of delivery guarantees does the broker make?
- Persistence: Are messages persisted to disk and able to survive broker crashes?
- Durability: If a consumer reconnects to the message broker, will it receive the messages that were sent while it was disconnected?
- Scalability: How scalable is the message broker?
- Latency: What is the end-to-end latency?

Each broker makes different trade-offs. For example, a very low-latency broker might not preserve ordering, make no guarantees to deliver messages, and only store messages in memory. A messaging broker that guarantees delivery and reliably stores messages on disk will probably have higher latency. Which kind of message broker is the best fit depends on your application's requirements.

After picking the proper broker solution for your application, we can see how the saga orchestrator communicates with the participants using async reply-style interaction based on message queues. To execute a saga step, it sends a command message to a participant telling it what operation to perform. After the saga participant has performed the operation, it sends a reply message to the orchestrator. The orchestrator then processes the message and determines which saga step to perform next.

V. Queries

API composition via API Gateway: very inefficient in-memory join. Can use caching with Redis or Memcached to avoid querying other databases too often. Also more computing and network resources are required, increasing the cost of running the application. Another drawback of this pattern is reduced availability. There's a risk also, that a query operation will return inconsistent data. - CQRS: Have duplicated data and use events to keep track. Drawbacks here are: you only have eventual consistency as the data duplication is asynchronous.(Better availability than consistency)

CQRS Three problems that are commonly encountered when implementing queries in a microservice architecture:

1. Using the API composition pattern to retrieve data scattered across multiple services results in expensive, inefficient in-memory joins. 2. The service that owns the data stores the data in a form or in a database that doesn't efficiently support the required query. 3. The need to separate concerns means

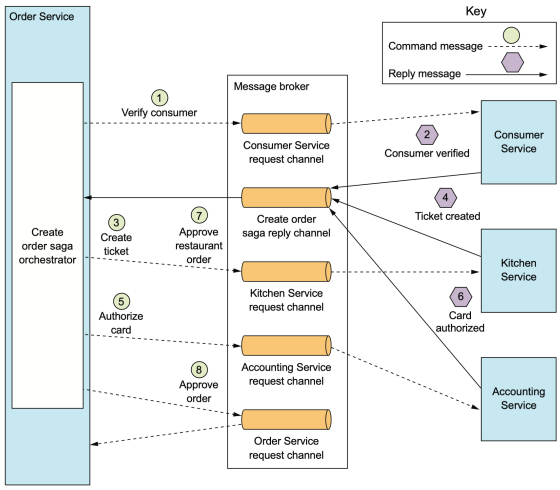

Fig. 2. Order Service implements a saga orchestrator, which invokes the saga participants using asynchronous request/response.

that the service that owns the data isn't the service that should implement the query operation.

- **write side** (or a command side) - The command side modules and data model implement create, update, and delete operations (CUD), ensuring business rules and handling commands. The command-side domain model handles CRUD operations and is mapped to its own database. It may also handle simple queries, such as nonjoin, primary key-based queries. **The command side publishes domain events whenever its data changes.** - **read side** (or a query side) - takes events produced by the write side and uses them to build and maintain a model that is suitable for answering the client's queries. The query side uses whatever kind of database makes sense for the queries that it must support. The query side has event handlers that subscribe to domain events and update the database or databases.

CQRS has both benefits and drawbacks. The benefits are as follows:

- Enables the efficient implementation of queries in a microservice architecture - Enables the efficient implementation of diverse queries - Makes querying possible in an event sourcing-based application - Improves separation of concerns

Even though CQRS has several benefits, it also has significant drawbacks:

- More complex architecture - Dealing with the **replication lag**: there's delay between when the command side publishes an event and when that event is processed by the query side and the view updated.

A service has an API that provides its clients access to its functionality. There are two types of operations: commands and queries. The API consists of commands, queries, and events.

VI. Conclusion

The conclusion goes here.

References

[1] Exploring microservices
[2] Available: https://microservices.io/patterns/data/database-per-service.html
[3] Available: https://microservices.io/book
[4] Available: https://microservices.io/book
[5] Available: https://www.infoq.com/news/2008/01/consistency-vs-availability
[6] Available: https://dl.acm.org/citation.cfm?id=284472.284478