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. [1] 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. Each service is required to have its own data store to achieve loose coupling. However, this introduces a new range of obstacles. We must, indeed, 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 countermeasures 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. [2] 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, displaying 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, but other services might need a NoSQL database such as MongoDB, which is good at storing unstructured or geospatial data.

A standard 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 involves 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 do not impact any other services. Besides each service can use the type of database that is best suited for its needs. For example, a service that has text searches could use ElasticSearch. [3]

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

II. Queries

Two problems are commonly encountered when implementing queries in a microservice architecture:

1) Retrieving data scattered across multiple services can result in expensive, inefficient in-memory joins without a proper solution;
2) The service can store the information in a database that doesn't efficiently support the required query, for instance traditional relational databases are not efficient for geospatial queries.

The most straightforward approach is the API composition pattern, which implements a query operation by invoking the services that own the data and combining the results. The role of API composer is usually taken by a service such as an API gateway.

However, this approach has several drawbacks. It can result in a very inefficient in-memory join of large datasets and it requires more computing and network resources, increasing the cost of running the application. There is a risk also that a query operation will return inconsistent data and moreover there are many queries that can't be implemented using this pattern. [4]

The Command query responsibility segregation (CQRS) pattern is instead more powerful than the API composition pattern, but it's also more complex. It maintains one or more view databases whose sole purpose is to support queries.

A. CQRS pattern

Command Query Responsibility Segregation, as the name suggests, is all about segregation, or the separation of concerns. As figure 1 shows, it splits a persistent data model and the modules that use it into two parts: the command side and the query side.

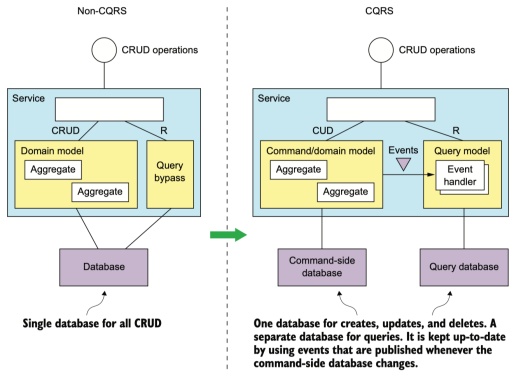

Fig. 1. On the left is the not-CQRS version of the service, and on the right is the CQRS version. [5]

- Write side (or the command side): the command side implements create, update, and delete operations (CUD), ensuring business rules and handling commands. The command side publishes events whenever its data changes;
- Read side (or a query side): it takes events produced by the write side and uses them to build and maintain a model that is suitable for answering the client queries. It uses whatever kind of database makes sense for the queries that it must support.

CQRS enables the efficient implementation of queries that retrieve data owned by multiple services, especially compared to the API composition pattern. Another benefit of CQRS is that it enables an application or service to efficiently implement a diverse set of queries. Attempting to support all queries using a single persistent data model is also often challenging and in some cases impossible.

Even though CQRS has several benefits, there is the additional operational complexity of managing and operating the extra datastores, which might even be of different types. Besides, as you might expect, there's a delay between when the command side publishes an event and when the query side processes that event and the view updated. A client application that updates a model and then immediately queries a view may see the previous version of the model.

As we have seen how to query data spanning multiple services, we proceed to analyse more complex transactions that invoice several services.

III. 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.

The traditional approach to maintaining data consistency across multiple services or databases 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, as well as the participants, have states in which they block waiting for incoming messages. [9]

Nevertheless, the main problem with traditional distributed transactions is that they reduce availability. 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 favour 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. [10]

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.

IV. 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 service. 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.

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

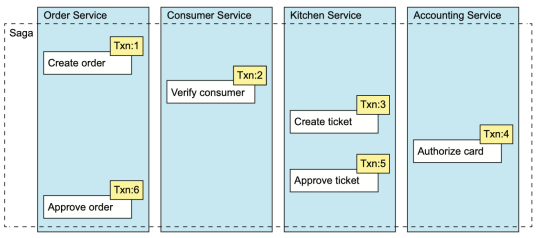

Fig. 2. Creating an Order using a saga. [6]

The example saga used throughout this section is the `Create Order Saga`, which is shown in Figure 2. 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 signalling the completion of the previous one.

This saga consists of the following local transactions:

1) Order Service: it creates an order in an `APPROVAL_PENDING` state;
2) Consumer Service: it verifies that the consumer can place an order;
3) Kitchen Service: it validates order details and create a ticket in the `CREATE_PENDING` state;
4) Accounting Service: it authorises the consumer's credit card;
5) Kitchen Service: it changes the state of the ticket to `AWAITING_ACCEPTANCE`;
6) Order Service: it changes 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 authorisation of the consumer's credit card fails. In this scenario, the saga executes the following local transactions:

1) Order Service: it creates an order in an `APPROVAL_PENDING` state;
2) Consumer Service: it verifies that the consumer can place an order;
3) Kitchen Service: it validates order details and creates a ticket in the `CREATE_PENDING` state;
4) Accounting Service: it authorises consumer's credit card, which fails;
5) Kitchen Service: it changes the state of the ticket to `CREATE_REJECTED`;
6) Order Service: it changes the state of the order to `REJECTED`.

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

A saga's coordination logic is therefore responsible for sequencing the execution of both forward and compensating transactions. [6]

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 of 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 behaviour can cause two problems. First, other sagas can change the data accessed by the saga while it's executing. And second, 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 database *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 [7]:

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;
3) *Fuzzy/non-repeatable reads*: two different steps of one saga read the same data and get different results because another saga has made updates;

It's then 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. [11]

One example of countermeasure is the usage of semantic locks: a saga's compensatable transaction sets a flag in any record that it creates or updates. The flag indicates that the record isn't committed and could potentially change. The flag can either be a lock that prevents other transactions from accessing the record or a warning that indicates that other transactions should treat that record with suspicion.

C. Reliable events

It's essential for the database update and the publishing of the event to happen atomically. Consequently, to communicate reliably, the saga participants must use transactional messaging. Otherwise, a service might update the database and then crash, for example, before sending the message. If the service doesn't perform these two operations atomically, a failure could leave the system in an inconsistent state.

A straightforward way to reliably publish messages is to apply the Transactional outbox pattern. This pattern uses a database table as a temporary message queue, where the service inserts messages. Atomicity is guaranteed because this is a local ACID transaction. Then, an effortless way to publish the messages is to poll the table for unpublished messages although frequently polling the database can be expensive.

A sophisticated solution is to tail the database transaction log (also called the commit log). Every committed update made by an application is represented as an entry in the database's transaction log. A transaction subscriber can read the transaction log and publish each change as a message to the message broker. [12]

V. Communication

As we have seen with the previous patterns, the services within a microservice architecture must frequently collaborate in order to complete a saga or to handle a query 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 favour 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

On the other hand, the basic idea behind a message-queuing system is that applications communicate by inserting messages in specific queues by using a message broker, an infrastructure service through which all messages flow.

A critical 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 behaviour 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. [8]

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 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.

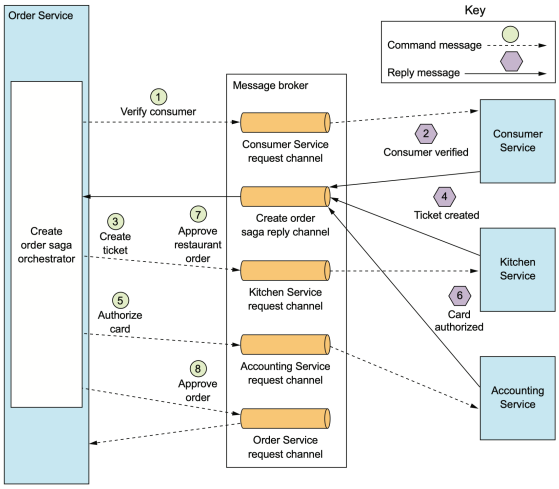

Fig. 3. Order Service implements a saga orchestrator, which invokes the saga participants using asynchronous request/response. [6]

VI. Conclusion

We started the paper by showing the reasons behind a database-per-service solution in a microservice architecture. Unfortunately, no technology is a silver bullet and this solution has several significant drawbacks and issues that must be kept in mind before having services with their own database.

One issue is that developers must deal with the additional complexity of managing databases in a distributed system. It's crucial to use an interprocess communication pattern aware of the asynchronous nature of the communication and the transience of the system, such as message queuing. Missing a message like a command event could indeed leave the system in an inconsistent state.

Moreover, implementing use cases that span multiple services requires the use of unfamiliar techniques like CQRS and sagas. One big obstacle that developers will have to face when

adopting microservices is moving from a single database with ACID transactions to a multi-database architecture with ACD sagas, because they're used to the simplicity of the ACID transaction model.

CQRS and sagas have their limitations as everything in life, but we believe that knowing how to use them properly will empower the developer to find a good compromise between truly independent and scalable services on the one hand and maintainability on the other hand. Otherwise having a database-per-service will make your life a living hell.

REFERENCES

[1] Adrian Cockcroft, "State of the Art in Microservices", pp. 41-44, Decembre 2014. [Online]. Available: https://www.slideshare.net/adriancockcroft/dockercon-state-of-the-art-in-microservices

[2] Alberto Simioni, Tullio Vardanega, "In Pursuit of Architectural Agility: Experimenting with Microservices" in *2018 IEEE International Conference on Services Computing (SCC)*, July 2018. [Online] Available: https://ieeexplore.ieee.org/document/8456408

[3] Chris Richardson, "Database per service" in "Microservice Architecture". [Online]. Available: https://microservices.io/patterns/data/database-per-service.html

[4] Chris Richardson, "Querying using the API composition pattern" in "Microservices Patterns", pp. 221-228, October 2018.

[5] Chris Richardson, "Overview of CQRS" in "Microservices Patterns", pp. 232-233, October 2018.

[6] Chris Richardson, "Using the Saga pattern to maintain data consistency" in "Microservices Patterns", pp. 114-117, October 2018.

[7] Chris Richardson, "Handling the lack of isolation" in "Microservices Patterns", pp. 126-131, October 2018.

[8] Chris Richardson, "Overview of broker-based messaging" in "Microservices Patterns", pp. 92, October 2018.

[9] Maarten van Steen, Andrew S. Tanenbaum, "Distributed Systems", pp. 483-490, February 2017.

[10] Werner Vogels, "Eventually Consistent". [Online]. Available: https://queue.acm.org/detail.cfm?id=1466448

[11] Lars Frank, Torben U. Zahle, " Semantic ACID properties in multi-databases using remote procedure calls and update propagations". [Online]. Available: https://dl.acm.org/citation.cfm?id=284472.284478

[12] Oded Shopen, "Listen to Yourself: A Design Pattern for Event-Driven Microservices". [Online]. Available: https://medium.com/@odedia/listen-to-yourself-design-pattern-for-event-driven-microservices-16f97e3ed066