Transaction Management in Microservices

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.

For example, as described in chapter 2, the createOrder() operation spans numerous services, including Order Service, Kitchen Service, and Accounting Service. Operations such as **these need a transaction management mechanism that works across services.**

**the traditional approach to distributed transaction management isn’t a good choice for modern applications**.

Instead of an ACID transactions, an operation that spans services must use what’s known as a *saga*, **a message-driven sequence of local transactions, to maintain data consistency.**

**One challenge with sagas is that they are ACD (Atomicity, Consistency, Durability). They 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.**

*Probably the good news is:*

in reality, even monolithic applications such as the FTGO application typically don’t use textbook ACID transactions. For example,

* **many applications use a lower transaction isolation level in order to improve performance.**
* **Also, many important business processes, such as transferring money between accounts at different banks, are eventually consistent.** Not even Starbucks uses two-phase commit ([www.enterpriseintegrationpatterns.com/ramblings/18\_starbucks.html](http://www.enterpriseintegrationpatterns.com/ramblings/18_starbucks.html)).

# The trouble with distributed transactions

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 (X/Open XA—see https://en.wikipedia.org/wiki/X/Open\_XA).

XA uses *two-phase commit* (2PC) to ensure that all participants in a transaction either commit or rollback. An XA-compliant technology stack **consists of XA-compliant databases** **and message brokers**, **database drivers,** **and messaging APIs**, and **an inter-process communication mechanism that propagates the XA global transaction ID.** Most SQL databases are XA compliant, as are some message brokers. Java EE applications can, for example, use JTA to perform distributed transactions.

As simple as this sounds, there are a variety of problems with distributed transactions.

* One problem is that **many modern technologies, including NoSQL databases** such as MongoDB and Cassandra, **don’t support them**.

Also**,** distributed transactions aren’t supported by modern message brokers such as RabbitMQ and Apache Kafka(now I think it does). As a result, if you insist on using distributed transactions, you can’t use many modern technologies.

* Another problem with distributed transactions is that **they are a form of synchronous IPC, which reduces availability.** In order for a distributed transaction to commit, all the participating services must be available. As described in chapter 3, 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.

*Get back to this paragraph later:*

There is even Eric Brewer’s CAP theorem, which states that a system can only have two of the following three properties:

consistency, availability, and partition tolerance (https://en.wikipedia.org/wiki/CAP\_theorem). Today, architects prefer to have a system that’s available rather than one that’s consistent.

On the surface, distributed transactions are appealing. From a developer’s perspective, they have the same programming model as local transactions.

But because of the problems mentioned so far, distributed transactions aren’t a viable technology for

modern applications.

Chapter 3 described how to send messages as part of a database transaction **withou**t using distributed transactions.

To solve the more complex problem of maintaining data consistency in a microservice architecture, an application must use a different mechanism that **builds on the concept of loosely coupled, asynchronous services. This is where sagas come in.**

# Using the Saga pattern to maintain data consistency