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.

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.

## **Transaction management in a microservice architecture**

Almost every request handled by an enterprise application is executed within a database transaction. Enterprise application developers use frameworks and libraries that simplify transaction management. Some frameworks and libraries provide a programmatic API for explicitly beginning, committing, and rolling back transactions. Other frameworks, such as the Spring framework, provide a declarative mechanism. Spring provides an @Transactional annotation that arranges for method invocations to be automatically executed within a transaction. As a result, it’s straightforward to write transactional business logic.

Or, to be more precise, transaction management is straightforward in a monolithic application that accesses a single database. Transaction management is more challenging in a complex monolithic application that uses multiple databases and message brokers. And in a microservice architecture, transactions span multiple services, each of which has its own database. In this situation, the application must use a more elaborate mechanism to manage transactions.

**The need for distributed transactions in a microservice architecture**

Imagine that you’re the FTGO developer responsible for implementing the createOrder() system operation. This operation must verify that the consumer can place an order, verify the order details, authorize the consumer’s credit card, and create an

Order in the database. It’s relatively straightforward to implement this operation in the monolithic FTGO application.

All the data required to validate the order is readily accessible. What’s more, you can use an ACID transaction to ensure data consistency. You might use Spring’s @Transactiona

annotation on the createOrder() service method.

In contrast, implementing the same operation in a microservice architecture is much more complicated. The needed data is scattered around multiple services. The createOrder() operation accesses data in numerous services. It reads data from Consumer Service and updates data in Order Service, Kitchen Service, and Accounting Service.

Shape, polygon

Description automatically generated

Because each service has its own database, you need to use a mechanism to maintain data consistency across those databases.

### **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

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 interprocess 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. 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.

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. 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. 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.

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**

Sagas are mechanisms to maintain data consistency in a microservice architecture without having to use distributed transactions. You define a saga for each system command that needs to update data in multiple services. A saga is a sequence of local transactions. Each local transaction updates data within a single service using the familiar ACID transaction frameworks and libraries mentioned earlier.

Pattern: Saga

Maintain data consistency across services using a sequence of local transactions that are coordinated using asynchronous messaging. See <http://microservices.io/patterns/data/saga.html>

The system operation initiates the first step of the saga. The completion of a local transaction triggers the execution of the next local transaction. Coordination of the steps is implemented using asynchronous messaging. An important benefit of asynchronous messaging is that it ensures that all the steps of a saga are executed, even if one or more of the saga’s participants is temporarily unavailable. Sagas differ from ACID transactions in a couple of important ways. They lack the isolation property of ACID transactions. Also, because each local transaction commits its changes, a saga must be rolled back using compensating transactions.

*AN EXAMPLE SAGA: THE CREATE ORDER SAGA*

The Order Service implements the createOrder() operation using this saga. The saga’s first local transaction is initiated by the external request to create an order. The other five local transactions are each triggered by completion of the previous one.

Graphical user interface, application

Description automatically generated

Creating an Order using a saga. The createOrder() operation is implemented by a saga that consists of local transactions in several services.

This saga consists of the following local transactions:

Graphical user interface, text, application

Description automatically generated

A service publishes a message when a local transaction completes. This message then triggers the next step in the saga. Not only does using messaging ensure the saga participants are loosely coupled, it also guarantees that a saga completes. That’s because if the recipient of a message is temporarily unavailable, the message broker buffers the message until it can be delivered.

On the surface, sagas seem straightforward, but there are a few challenges to using them. One challenge is the lack of isolation between sagas. Another challenge is rolling back changes when an error occurs.

**Another challenge is rolling back changes when an error occurs.**

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. It executes a ROLLBACK statement, and the database undoes all the changes made so far. Unfortunately, sagas can’t be automatically rolled back, because each step commits its changes to the local database. This means, for example, that if the authorization of the credit card fails in the fourth step of the Create Order Saga, the FTGO application must explicitly undo the changes made by the first three steps. You must write what are known as *compensating transactions*.

Suppose that the (n + 1)th transaction of a saga fails. The effects of the previous n transactions must be undone. Conceptually, each of those steps, Ti, has a corresponding compensating transaction, Ci, which undoes the effects of the Ti. To undo the effects of those first n steps, the saga must execute each Ci in reverse order. The sequence of steps is T1 ... Tn, Cn ... C1, as shown:

When a step of a saga fails because of a business rule violation, the saga must explicitly undo the updates made by previous steps by executing compensating transactions.

Diagram, table

Description automatically generated

The saga executes the compensation transactions in reverse order of the forward transactions: Cn ... C1. The mechanics of sequencing the Cis aren’t any different than sequencing the Tis. The completion of Ci must trigger the execution of Ci-1

Graphical user interface, text, application, email

Description automatically generated

If a local transaction fails, the saga’s coordination mechanism must execute compensating transactions that reject the *Order* and possibly the *Ticket*.

It’s important to note that not all steps need compensating transactions. Read-only steps, such as verifyConsumerDetails(), don’t need compensating transactions. Nor do steps such as authorizeCreditCard() that are followed by steps that always succeed.

The compensating transactions for the Create Order Saga:

Table

Description automatically generated

The first three steps of the Create Order Saga are termed ***compensatable transactions*** because they’re followed by steps that can fail, how the fourth step is termed the saga’s ***pivot transaction*** because it’s followed by steps that never fail, and how the last two steps are termed ***retriable transactions*** because they always succeed.

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:

Text

Description automatically generated