The Saga Pattern

Maintaining Data Consistency in a Microservices Architecture

Arnthor Jonsson

April 30, 2019

Abstract

In a microservice architecture where each service owns its own data, the traditional ACID transactions using two-phase commit is often not an option. This report attempts to analyze how data consistency is kept in a microservice architecture using the Saga pattern.

The Saga pattern tries to solve distributed transactions without the two-phase-commit protocol by trading off atomicity for availability. Sagas split the entire work into a set of local transactions. In the case of a failure in one or more local transactions, the system executes compensating transactions to roll back the work that was already performed. The Saga pattern is a failure management pattern that forces developers to think about failure in their design in order to build more robust systems.

1 Introduction

In order to achieve deployability and modifiability, complex Internet services are increasingly being developed as distributed systems using a microservice architecture [5, 1]. By constraining a service to handle only a single part of the system, microservice architecture removes some of the complexity from the code. Therefore, making each service easier for a developer to understand [9].

But as with all things, they come at a cost. Developers must deal with the additional complexity of creating a distributed system. Furthermore, to ensure loose coupling and achieve deployability and modifiability, each service must handle and maintain its own data. As not all system can use traditional ACID transactions using two-phase commit, data consistency is one of the challenges when developing a distributed system [9]

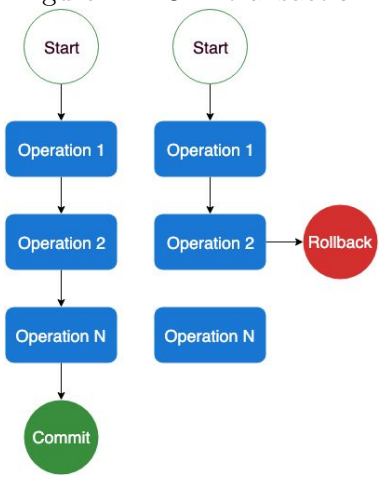
In section 5 we will analyze how a distributed system using a microservice architecture maintains data consistency across its services by using the Saga pattern.

2 ACID Transactions

A transaction is a group of operations which may span multiple database queries. They are an essential part of an application ensuring locks over resources in order to maintain consistency in the database. A transaction has the following properties (ACID):

- *Atomicity*: Either a transaction fully succeeds or is rolled back i.e. none of the operations persist.
- *Consistency*: The state of the database will remain consistent both before a transaction begins and after the transaction ends.
- *Isolation*: Transactions should not affect each other i.e. transactions can run in parallel.
- *Durability*: Any change made to the database should persist after the transaction completes [7].

Figure 1: ACID transaction



3 Microservice Architecture

Microservice architecture is an approach of designing software application as a set of loosely coupled, small collaborating services. Each service runs in its own process and communicates with other services with lightweight mechanisms, often over HTTP. These services may be written by multiple teams, in different programming languages and use different data storage technologies. Furthermore, each service may be deployed independently. The microservice architecture allows each service to be developed at its own rate and in parallel with other services, which enables rapid and frequent development. However, in order to be able to independently develop services, they must be loosely coupled. Therefore, each service must maintain it's own data in order to be loosely coupled from other microservices. If this is not the case i.e. if two microservices were to share data, then the development teams would need to carefully coordinate changes made to the data schema and thus slowing down development [4, 9, 8].

It is, therefore, important that in a microservice architecture each service owns its own data in order to get the full benefits from architecture. This is sometimes referred to as the database per service pattern. It does not necessarily mean that each service needs to provisions a database server but that data from a service should only be accessible to other microservices using the API from the service [8].

By setting these requirements the system loses some of the guarantees a single database and ACID transactions offer. Is there a way for the system to get similar guarantees with other methods?

4 Two-Phase Commit

Two-Phase Commit (2PC) is a distributed algorithm that tries to solve data consistency in a distributed transaction by coordinating all the processes in the transaction [2].

In execution of a single distributed transaction, the 2PC protocol consists of two phases:

1. Prepare phase (Voting phase): The coordinator attempts to prepare all participating in the transaction by asking if they are ready to commit. The participants respond either by *Yes* if the local transaction completed or *No* to abort if there was a problem with the local transaction.

2. Commit phase: The coordinator decides, based on the voting of the participants if the whole transaction should be committed. If and only if all of the participants vote *Yes* i.e. all of the local transactions completed, the coordinator instructs participants to commit their local transaction. Otherwise, the coordinator tells the participants to abort the transaction e.g. roll back the transaction.

Figure 2: 2PC Prepare Phase

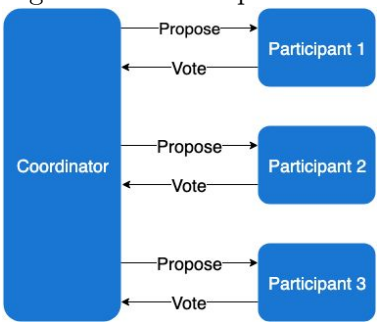
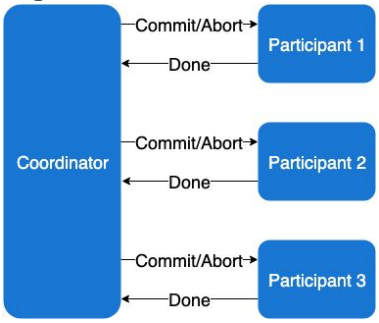


Figure 3: 2PC commit Phase



While 2PC solves the data consistency problem to a decent extent, however, it has some drawbacks. 2PC concentrates the logic of the coordinator to a single node. By doing so it makes that node a single point of failure, which is something distributed systems want to avoid [3]. Furthermore, by introducing phases participants are dependent on each other, thus the transaction is bounded by the slowest participant since participant P_1 will have to wait for confirmation from P_2 in order to commit its transaction. This brings us to the greatest disadvantage of the protocol, namely that 2PC is a blocking protocol. What this means is that a participant will have to block its resources while waiting for a message from the coordinator. Other processes accessing the same resources as locked by the participant will have to wait for the transaction to be over and the locks released before accessing the resources. By this, we can see that 2PC and distributed transactions can hamper both the availability and performance of the system. The question then becomes can we trade off atomicity for availability?

5 Sagas

The Saga pattern tries to solve the distributed transactions problem without the two-phase-commit protocol. In their paper "Sagas" Hector Garcia-Molina and Kenneth Salem introduced the notion of a saga. They described a saga as a "long lived transaction that can be written as a sequence of transactions which in turn can be interleaved" [6]. A sequence of transactions is defined as $T_1, T_2, ...T_n$. Each T_i should have a compensating transaction C_i, thus for the sequence we have $C_1, C_2, ...C_n$ compensating transactions. A compensating transaction is defined as a transaction "that undoes, from the semantic point of view, any of the actions performed by T_i" [6].

With this in place, the system can make the following guarantees. Either sequence $T_1, T_2, ...T_n$ runs or sequence $T_1, T_2, ...T_m, C_m, C_{m-1}, ..., C_1$ runs. This means that either all transactions in the sequence complete successfully or compensating transactions are ran to amend a partial execution [6].

By requiring a compensating transaction for each transaction the Saga pattern forces developers to think about failure in their system design. Therefore, the Saga pattern is described as a failure management pattern [11].

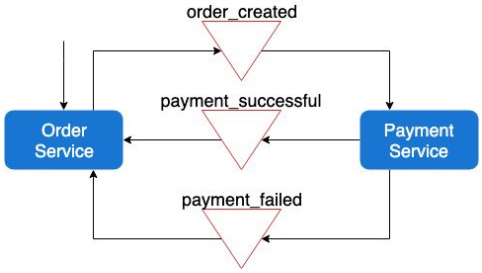
In a microservice architecture "a saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga" [10]. That is, after the completion of transaction T_i something must decide what step to execute next. In the case of success the system should execute T_{i+1}, otherwise execute compensating transactions $C_i, ..., C_1$ to roll back. There are a couple of ways to implement this pattern. In this article, we will explore the two most popular.

Choreography

After completing a local transaction a service publishes an event that triggers local transactions in other services. The sequence ends when the service does not publish an event [10].

1. Order Service creates a new order and sets the state to *pending*.
2. After finishing its local transactions Order Service publishes an event *order_created*.
3. Payment Service listens to the *order_created* event and tries to bill the client.
4. Based on if the Payment Service is successful in billing the client it publishes either *payment_successful* or *payment_failed* event.
5. Order Service listens to both *payment_successful* and *payment_failed* events.
   (a) *payment_successful*: Order Service changes the state of the order to *approved*.
   (b) *payment_failed*: Order Service changes the state of the order to *canceled*.

Figure 4: Choreography



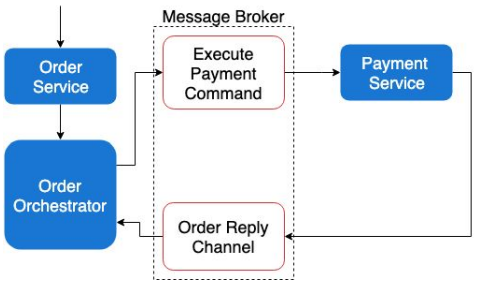
Choreography offers a way to implement the Saga pattern and is easy to understand. However, this approach can introduce cyclic dependencies between services as services subscribe to each other's events. Furthermore, as our system grows and our transactions increase this implementation becomes more and more confusing as it may be difficult to track which service listens to which event.

Orchestration

In this approach, the system creates an orchestrator with the sole responsibility of coordinating what local transaction to execute next. Messages are usually sent using a message broker that guarantees, at least one's delivery. This means that the message broker will deliver that message eventually [10]. Furthermore, it is useful to make requests idempotent as a message broker could deliver the same message twice.

1. Order Service creates a new order, sets the state to *pending* and creates a new Order Orchestrator.
2. The Order Orchestrator sends an *ExecutePayment* command to the Payment Service.
3. Payment Service tries to execute the command and replies either with a *PaymentSuccessful* or *PaymentFailed* message.

Figure 5: Orchestration



4. Order Orchestrator receives the message
   from Payment Services and sends either a *SuccessfulOrder* or *FailedOrder* command to the Order Service

5. Order Service receives the message

   (a) *SuccessfulOrder*: Order Service changes the state of the order to *approved*.

   (b) *FailedOrder*: Order Service changes the state of the order to *canceled*.

By centralizing the orchestration of the saga i.e. having an orchestrator invoking participants, Orchestration sagas are more loosely coupled then Choreography sagas. This, in turn, reduces participants complexity as participants only need to handle execution and reply commands. Furthermore, by orchestrating the work, the system can better avoid cyclic dependencies.

The orchestrator should not maintain any business logic. This is preferable since in the case of an unexpected failure in the orchestrator the system can create a new orchestrator. The system, however, has to be able to determine if the saga was in a safe state or not when the orchestrator failed.

- Safe state: Either that all transactions up to this point have completed and the system can, therefore, continue with the saga. Or the saga has been aborted, then we simply proceed with compensating transactions to roll back.

- Un-Safe state: Is when a transaction has started but has not notified if it ended or not before the failure of the orchestrator. If a transaction is idempotent it may be safe to replay that transaction.

One of the drawbacks of Orchestration sagas is that the system will need to manage an extra service. This will increase the complexity of the system.

6 Conclusion

In this article, we have gone over some of the aspects of dealing with data consistency in a microservice architecture using the Saga pattern. As stated in section 3, a microservice architecture should apply the database per service pattern, in order to ensure loosely coupled services. Because of drawbacks, some systems cannot use traditional ACID transactions using two-phase commit to solve data consistency and are therefore forced to explore other methods. An alternative to using 2PC is to use Sagas to maintain data consistency across microservices.

Sagas enable the system to maintain data consistency across multiple services without using distributed transactions. This is done by the trading of atomicity for availability. At its core sagas are a failure management pattern forcing developers of the system to think about failure in their design in order to build more robust systems.

References

[1] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, May 2016.

[2] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. Concurrency control and recovery in database systems. 1987.

[3] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. Addison-Wesley Publishing Company, USA, 6th edition, 2010.

[4] M. Fowler and J. Lewis. Microservices. `https://www.martinfowler.com/articles/microservices.html`. Accessed: 2019-04-28.

[5] Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andrew Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, and Ion Stoica. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28(13):2009, 2009.

[6] Hector Garcia-Molina and Kenneth Salem. Sagas. *SIGMOD Rec.*, 16(3):249–259, December 1987.

[7] J. Kennedy and M. Satran. What is a transaction? `https://docs.microsoft.com/en-us/windows/desktop/Ktm/what-is-a-transaction`, May 2018. Accessed: 2019-04-28.

[8] Chris Richardson. Pattern: Database per service. `https://microservices.io/patterns/data/database-per-service.html`. Accessed: 2019-04-28.

[9] Chris Richardson. Pattern: Microservice architecture. `https://microservices.io/patterns/microservices.html`. Accessed: 2019-04-28.

[10] Chris Richardson. Pattern: Saga. `https://microservices.io/patterns/data/saga.html`. Accessed: 2019-04-28.

[11] Clemens Vasters. Sagas. `http://vasters.com/archive/Sagas.html`. Accessed: 2019-04-30.