ACIDBRAIN: Maintaining data consistency in microservices

Nana Pang*, Wode "Nimo" Ni †, and Xin Chen ‡

Columbia University

Email: *np2630@columbia.edu, †wn2155@columbia.edu, ‡xc2409@columbia.edu

Abstract

As software systems become more distributed and larger in scale nowadays, there is a growing need for novel architectures that are modularized, flexible, and scalable. Microservice architecture emerged as a popular software architecture where developers divide a large application into small, self-contained components (services). Although the separation significantly increases flexibility and reduces difficulty in development, maintaining data consistency across these isolated databases can still be be challenging. Our project targets maintaining data consistency among distributed microservices. To maintain data consistency, an event driven model called **Saga**[1] is often used to keep track of the data sources of each transaction. In this project, we investigate two predominant event sourcing frameworks: *Eventuate* and *Axon*, from an **end-users perspective**. We experiment with both of them by building data consistency guarantees on a dummy microservice system, and analyze the effectiveness of the two frameworks both quantitatively and qualitatively.

I. INTRODUCTION

Large software systems have been growing rapidly and becoming more distributed. As a result, the traditional monolithic software architecture, where modules of different functionalities are closely coupled and developed by the same group of developers, is now challenged by a new architectural pattern: **Microservices**. A software system that employs microservice architecture comprises of a suite of clearly defined small services, each running in its own process [2]. As a result, each service can be developed by completely separated teams and using distinct technology stack, thereby decoupling the modules and parallelizing development.

In a monolithic system, all modules typically share the same backend database and enjoy the ACID (Atomicity, Consistency, Isolation, Durability) properties provided by the database [3]. Under microservice architecture, however, each module would use a separate database to ensure good isolation among service modules, and each database might be using an entirely different technology. Therefore, large transactions that involves multiple modules, which used to run in the same database, now span across multiple databases with drastically different implementations. In this case, maintaining data consistency becomes a more complex task. **Two-phase-commit (2PC)** protocol [4] is a popular solution, where a coordinator process ensures all databases have the requested resource available before commiting the transaction. While 2PC ensures the correctness of distributed transactions, it requires all resources to be exclusively locked until the transaction finishes.

Unfortunately, in real world applications, many distributed transactions are long-lived, meaning they take much longer (in terms of hours and/or network gaps) to finish. For example, an e-commerce application may have ordering, billing, and shipping modules, and a complete transaction of purchasing an item would involve all components and would not complete until the item is shipped. In the case of these long-lived transactions, 2PC does not scale well because it locks all databases involved in the transaction and there can be many other transactions occurring in the system simultaneously.

Among the existing solutions to maintain data consistency in distributed databases of microservice systems, we chose to look at the **Saga pattern** [1]. There are a few arguments for Sagas over traditional consensus protocols such as 2PC. Firstly, the Saga pattern does not require synchronization of all databases in a transaction, making it more suitable for long-lived transactions. Also, Saga tends to be easier to implement than 2PC, whose logic is relatively complex.

II. Background

A saga is a sequence of local transactions. Each local transaction updates the database and publishes an event to trigger the next local transaction in the saga [5]. Using this pattern, a large distributed transaction is broken into multiple local transactions that update their local databases and publish events globally to notify others, which are then coordinated by a saga. To maintain data consistency, each local transaction theoretically must be accompanied by compensatory transaction. As a result, the saga can rollback the large transaction by executing these compensations in a reverse order.

In general, there are two types of sagas: *choreography-based* and *orchestration-based*. An orchestration-based saga is a standalone object that coordinates multiple service in a centralized manner, whereas choreography-based saga is implemented by each local transaction publishing domain events that trigger local transactions in other services [5].

The use of microservices and sagas also motivates the overall architectural pattern to deviate from the traditional layered architecture. **Event sourcing** and **Command Query Responsibility Segregation (CQRS)** are often used to fully implement sagas. Since sagas coordinate local transactions by publishing events after updating local datastores, it is vital for the publication and update to be atomic - otherwise the system ends up in an inconsistent state. Event sourcing is used to enforce atomicity of events publishing and datastore updates.

III. Existing Implementation of Sagas

Having decided to investigate Sagas, we looked into the existing implementations of Saga pattern for microservices. Among them, we picked Eventuate and Axon, which are the most popular and seemed to be well maintained. There are already comparisons online of the two frameworks [6], [7]. All of we could gathered, however, were written by implementers of transaction frameworks. For example, [7] was written by the a member of the Narayana team within Red Hat, which also has a similar implementation. Therefore, in this project, we would like to evaluate the end-user experience of using Saga frameworks to implement data consistency in microservice systems. During our early research on Microservices, we found a number of online documents about Microservices, but most of them only explain the architecture style of Microservices which more accessible but remain too high-level to guide implementation. For example, searching Microservices on Google gives around 1,700,000 results, Microservices example only gives about 706,000 results. Furthermore, Microservices data consistency gives only 62,300 results. Our project aims to find out more about what technology can be used in implementing Microservices with data consistency guarantees.

Which systems we have chosen: We found two major frameworks implementing the Saga pattern:

- **Axon framework** [8] is a Java based framework for building scalable and highly performant applications. The main notion is the event processing which includes the separated Command bus for updates and the Event bus for queries. According to an original Axon authors response on StackOverflow [6], Axon has been around for about 8 years and is being used by many systems in production since then. Axon has extensive support for Spring. Yet Spring is not required for Axon, it made configuration easy with Spring annotations.
- **Eventuate** [9] is a platform that provides an event-driven programming model that focus on solving distributed data management in microservices architectures. The framework stores events in the MySQL database and it distributes them through the Apache Kafka platform. Eventuate is a framework that has integration with Gradle and Maven project. Therefore, we have implemented our prototype using Spring and Maven.

Since both Eventuate and Axon frameworks themselves are freely available on the internet (it seems that some of other related services are not, we obtained them through their official websites [8], [9].

IV. Conclusion

References

[1] Hector Garcia-Molina and Kenneth Salem. *Sagas*, volume 16. ACM, 1987.

[2] James Lewis and Martin Fowler. Microservices: a definition of this new architectural term. *MartinFowler. com*, 25, 2014.

[3] Jim Gray et al. The transaction concept: Virtues and limitations. In *VLDB*, volume 81, pages 144–154. Citeseer, 1981.

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

[5] Chris Richardson. Pattern: Saga. *Microservices.io http://microservices.io/patterns/data/saga.html [last accessed on May 4, 2018]*, 2014.

[6] stackoverflow. Axon framework vs eventuate comparison. *https://stackoverflow.com/questions/43136291/axon-framework-vs-eventuate-comparison [last accessed on May 4, 2018].*

[7] Saga implementations comparison. *http://jbossts.blogspot.com/2017/12/saga-implementations-comparison.html [last accessed on May 4, 2018].*

[8] Axon. The axon framework. *http://www.axonframework.org/ [last accessed on May 4, 2018].*

[9] Chris Richardson. Eventuate - official website. *Eventuate.io http://eventuate.io/ [last accessed on May 4, 2018].*