Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture

Chaitanya K. Rudrabhatla
Executive Director - Solutions Architect
Media and Entertainment domain
Los Angeles, USA

***Abstract*—Microservice Architecture (MSA) is an architectural design pattern which was introduced to solve the challenges involved in achieving the horizontal scalability, high availability, modularity and infrastructure agility for the traditional monolithic applications. Though MSA comes with a large set of benefits, it is challenging to design isolated services using independent Database per Service pattern. We observed that with each micro service having its own database, when transactions span across multiple services, it becomes challenging to ensure data consistency across databases, particularly in case of roll backs. In case of monolithic applications using RDBMS databases, these distributed transactions and roll backs can be handled efficiently using 2 phase commit techniques. These techniques cannot be applied for isolated No-SQL databases in micro services. This research paper aims to address three things: 1) elucidate the challenges with distributed transactions and rollbacks in isolated No-SQL databases with dependent collections in MSA, 2) examine the application of event choreography and orchestration techniques for the Saga pattern implementation, and 3) present the fact-based recommendations on the saga pattern implementations for the use cases.**

***Keywords*—Microservice architecture; database per service pattern; Saga pattern; orchestration; event choreography; No-SQL database; 2 phase commit**

I. Introduction

According to Martin Flower, the microservice architectural style [2], [3] is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. MSA defines each service to be totally independent [4] with its own database. When MSA is defined with completely isolated No-SQL databases [6], and when the business transactions span across multiple services, the state changes in one database entity are not visible to state changes in the other. The application cannot use the local ACID transactions as the entities are now spread into multiple databases. Also, if the transaction gets rolled back because of a failure in one of the micro services, state recovery cannot be attained using the standard 2PC [8] as these are distributed entities. The scenario becomes even more challenging when there are dependent entities with one to many relationships.

To handle this scenario, saga pattern can be used [5]. The services which alter the state can be written in the form of a Saga. In a saga, each service which changes the state of the database in a distributed transaction [1], [11], can generate an event which can trigger the next micro service. In case of a failure, the saga triggers a sequence of compensating roll back events from one service to the other in the reverse direction. These sagas can be designed using two techniques: (1) Event choreography, in which each service can trigger other service's event without a central coordinator. (2) Orchestration, in which a central coordinator makes the decision of triggering the relevant events in the saga. Both these techniques have pros and cons based on the use case which is being implemented. In the past some researchers have suggested the use cases for which these approaches are suitable, but a quantitative analysis has not been performed. In this research, we tried to come up with the recommendations on which saga technique to pick up in which scenario by examining the performance and complexity using the factual data generated by simulating a variety of use cases using a custom project developed on spring boot based micro services and Mongo DB and ActiveMQ based java messaging service queue, which is explained in the later sections.

The rest of the paper is organized as follows. In Section II, we explain the challenges involved in the distributed transactions in the MSA with no-SQL databases by bringing up the use cases in an e-commerce application. In Section III, we explain how the event choreography and orchestration can be implemented for these use cases. In Section IV, we go through the relevant work conducted in the research project and outline the results. In Section V, the conclusions are presented.

II. Distributed Transactions in MSA

In a traditional monolith application based on relational databases, the transactions originate and progress within the scope of the container hosting the application. So, it becomes easy to handle the roll backs. But it is different in case of micro services running with database per service pattern [7]. Since the entities and the databases are isolated, the traditional rollback approaches cannot be applied. We have taken an example of a standard e-commerce application flow (Fig. 1) to explain the complexity of distributed transactions.

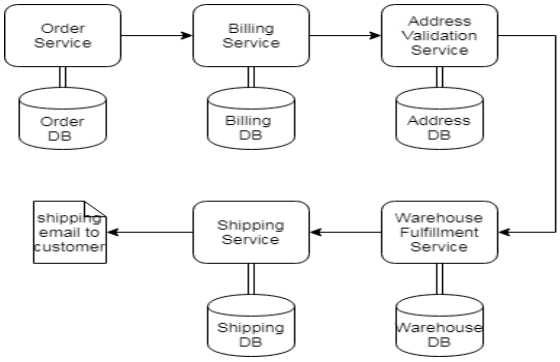

Fig. 1. Micro services in an e-commerce application.

As it can be seen, the order placement, credit card billing, address validation, fulfillment and inventory update, shipping are the various micro services which have their own databases and entities. It is not possible to capture all the steps in a single ACID transaction. To ensure the data consistency [13], we need to implement distributed transaction. Since there is no direct linking of the entities or databases, when the distributed transaction progresses few steps and encounters an issue, it becomes challenging to handle the consistency in the entity states by performing the roll backs. For example, when an order is placed successfully, and the customer's credit card is charged, but if the address validation fails, the transaction must be rolled back correctly so that the customer is not charged for the unfulfilled item. That means the transaction must be rolled back in the proper reverse order. To handle this flow of events in forward and reserve directions by triggering the relevant events, Saga pattern can be used. Saga pattern can be implemented using Event choreography and orchestration techniques as mentioned below.

III. Event Choreography vs. Orchestration

Some researchers already explored how event choreography and Orchestration [12] techniques for implementing sagas in micro service architecture. We are going to explain it in detail with the use case of e-commerce application mentioned above. In Event choreography approach, when a micro service executes a local transaction, it publishes an event which can be subscribed by one or other micro services to trigger their local transactions. This process proceeds till the last service which doesn't publish any more events, there by marking the end of transaction. It can be visualized in Fig. 2 given below.

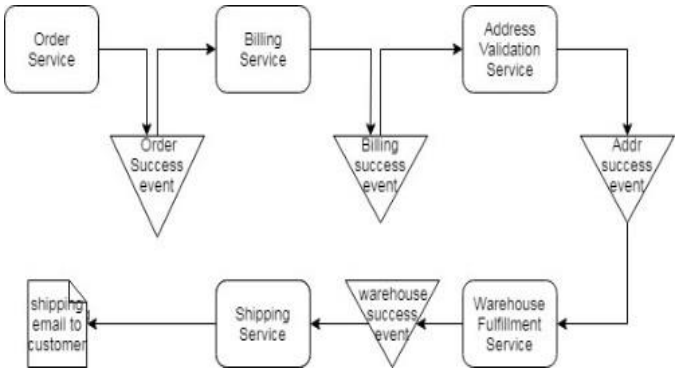

Fig. 2. Event choreography flow.

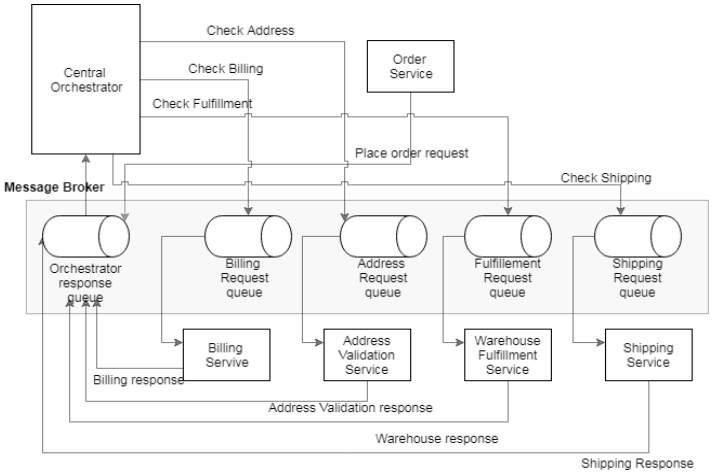

Fig. 3. Orchestration flow.

In this approach there is no central coordinator which listens to the events and triggers the relevant micro service local transaction.

The other technique to implement sagas is called orchestration. In this, there is a central coordinator which listens to all the events emitted by any of the micro service local transaction. Based on the incoming event, it triggers the next local transaction in a different micro service or services. This pattern is depicted in Fig. 3 below.

The scenarios mentioned above are depicted using a single entity at each local transaction level. It can be complex if there are dependent collections in each of those data sources. When a transaction needs to rolled back, the dependent collections state needs to be reverted as well. Both the techniques mentioned above have pros and cons based on the scenario that needs to be implemented. In the next section, we are going to simulate various scenarios and understand the suitability of these techniques.

IV. Related Work: Research Project

To determine which saga implementation technique is more suitable under which scenario, we have implemented a research project and simulated various circumstances. We have implemented micro services in spring boot technology. A service discovery component called Eureka [9], [14] is used to register and discover the micro services running. This is similar to the other API gateways like Kong or Apigee which are available in the market. The entities are represented as collections in an open source no-SQL database called Mongo. Each micro service -MS1, Ms2.MSn has an isolated instance of Mongo DB- DB1, DB2.DBn, respectively with a collection –C1, C2 Cn, respectively running on each of those database instances. These micro services and database instances run on Linux based virtual machines. First the event choreography technique is executed with 2 micro services, MS1 and MS2 having DB1 and DB2 as databases for each micro service with C1 and C2 as collections in each database respectively as depicted in Fig. 4. Each collection has an attribute called state which describes the state of the entity with the possible values of S1 and S2 and an attribute called timestamp which records the time stamp when the state change occurred.

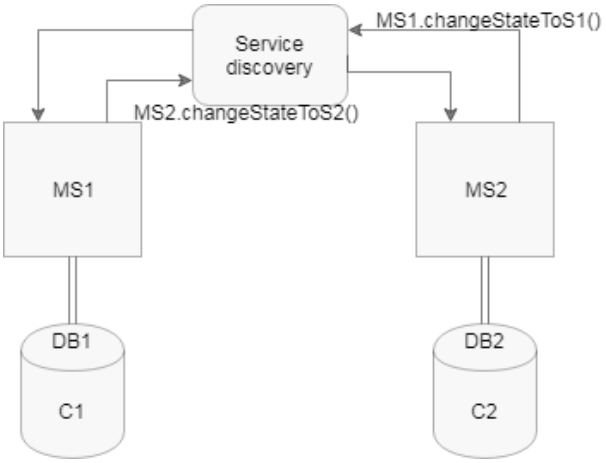

Fig. 4. Event choreography with 2 micro services.

A. Performance Analysis

Here is the sequence of steps which are executed as a part of the project to compare the performances:

- Micro service MS1 method is called which changes the value of state attribute of collection C1 from 'S1' to 'S2'. This method saves the time stamp T1 of the update action in the timestamp attribute of C1.
- Once the update is complete, MS1 triggers an event called 'MS1_state_change_success' which calls the method 'changeStateToS2' on micro service MS2.
- MS2 executes a logic to update state of C2 from 'S1' to 'S2'. But we simulate the transaction failure with which the state change of C2 fails.
- Now due to transaction failure, MS2 creates an event called 'MS2_state_change_failure' which rolls back the transaction in MS2 and calls the method 'changeStateToS1' on micro service MS1.
- MS1 then rolls back the state of C1 from 'S2' and 'S1' and updates the time stamp to new value T2.
- The difference between T2 and T1 tells us the time taken to execute the Saga with Event choreography of 2 micro services. These values are noted down as time taken for 2 micro service event choreography.
- Similarly, this exercise is repeated 3 more times by taking 4 micro services, 6 micro services and 8 micro services in each attempt. The exercise is executed in the same fashion as described in the steps above where the transaction progresses in a series of events from MS1 to MSn-1. At MSn-1 it triggers the event 'MSn-1_state_change_success' and calls the 'changeStateToS2' method on MSn. MSn fails the transaction and rolls back the transaction by calling the 'changeStateToS1' on MSn-1. This rolls back the state of Cn-1 to S1 and triggers the method 'changeStateToS1' on MSn-2. This happens till it reaches 'changeStateToS1' on MS1 which rolls back the state to S1 and calculates the time difference.

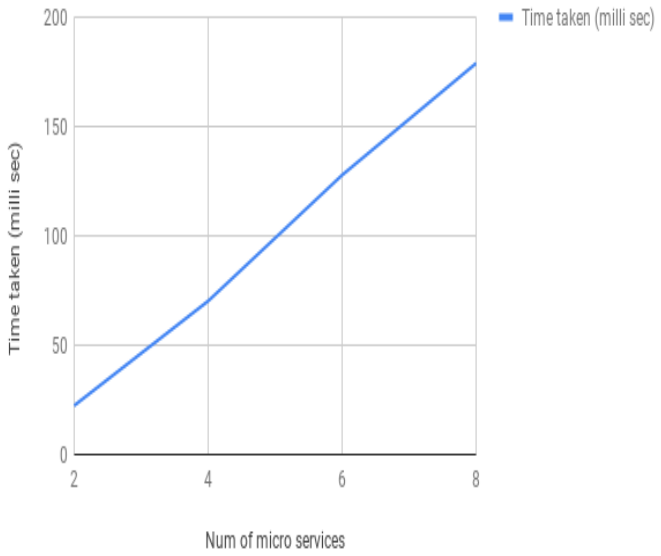

Fig. 5. Correlation of time taken vs. Micro services in event choreography.

We have executed 5 test runs and noted down the time taken in each instance and calculated the average. The graph in Fig. 5 shows the time taken vs number of microservices in the event choreography.

A similar exercise is performed using orchestration technique. In this, a central orchestration service is added which listens to various events and takes the necessary action. We have used an Apache ActiveMQ [10] as the JMS broker. Here is the sequence of steps which take place.

- MS1, Ms2 MSn are the microservices, each having a mongo DB instance DB1, DB2 DBn. Each of the databases has collections C1, C2 Cn. Like the setup described in event choreography.
- For Orchestration technique we hosted a new micro service MSn+1.
- We have n different queues running on Active MQ broker Q2, Q3…Qn+1 with MS2, MS3…MSn+1 subscribing to each of them, respectively.
- When state change happens from S1 to S2 on MS1, it triggers an event 'MS1_state_change_success' on the orchestrator MSn+1.
- Orchestrator posts a message on Q2, which MS2 listens and executes 'changeStateToS2' method and changes state to S2. Upon state change, MS2 posts a message 'MS2_state_change_success' on the Qn+1 which is subscribed orchestrator MSn+1.
- This forward transaction continues till it reaches the last micro service MSn. At MSn we fail the transaction, roll back the state to S1 on Cn and post the message 'MSn_state_change_failure' on the Qn+1 which is subscribed orchestrator MSn+1.
- Orchestrator listens to this roll back event from MSn and posts a rollback message on Qn-1. MSn-1 listens to this message and rolls back the state to S1 on MSn-1.

- This rollback continues till it reaches MS1 which rolls backs the state to S1, notes the time difference and posts no more messages.
- This exercise is also performed 4 times, with 2,4,6,8 micro services and orchestrator and the timestamps are noted.

In Fig. 6 given below, the graph shows the time taken vs number of microservices in the orchestration technique.

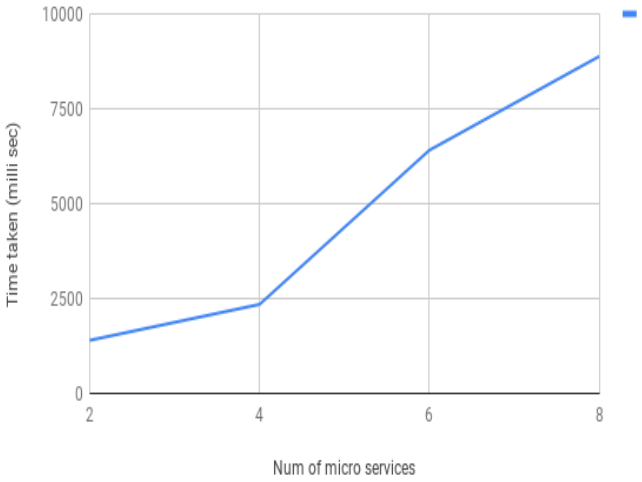

Fig. 6. Correlation of time taken vs. Micro services in orchestration.

Now it can be clearly seen that event choreography takes much faster in performance when compared to Orchestration. Event choreography can be well suited in the scenarios where the number of micro service calls are limited, and the response time is critical.

B. Complexity

The same experiment is repeated with a scenario which is more complex. When a state change is occurred in one micro service, we want to test both the techniques by triggering multiple events in more than one micro service. To do this, we implemented the following pattern.

- When state changes from S1 to S2 in C1, MS1 triggers 2 events which changes the state of C2 in MS2 from S1 to S2 and state of C3 in MS3 from S1 to S3.
- Upon successful change of state from S1 to S2 in C2, MS2 triggers 2 events which changes the state of C3 in MS3 from S3 to S2 and state of C4 in MS4 from S1 to S3.
- When a transaction fails at MS4, it rolls back the state of C4 to S1 and triggers 2 events which changes the state of C3 in MS3 from S2 to S1 and state of C2 in MS2 from S2 to S3.
- Upon successful rollback of state from S2 to S1 in C3, MS3 triggers 2 events which changes the state of C2 in MS2 from S3 to S1 and state of C1 in MS1 from S2 to S3.
- Finally, upon successful rollback of state from S3 to S1 in C2, MS2 triggers an event which changes the state of C1 in MS1 from S3 to S1.

This pattern is performed for 4 micro services and 6 micro services in both event choreography and orchestration for 5 test runs. It was observed that the time taken for orchestration technique is approximately 40 times more than the event choreography. But it was noted that as the number of events increased, it became more and more complex to handle the code in individual micro services. Whereas orchestrator proved to be more elegant in handling multiple events with less confusion as the event handlers are orchestrator at a single location.

C. Load based Test

The same setup is repeated one more time with a scenario where the frequency of events which are triggered are increased by 5-fold and 10-fold. This is obtained by writing a test client which fires parallel requests. We calculated the ratio of response times with the frequency of 1 vs 5 vs 10. We observed that the event model began to respond slowly as the frequency increased, whereas the orchestrator was able to handle the load better. The response times varied as 1:3.6:8.2 for event, whereas the ratios for orchestration came out as 1:3.9:6.4. These results might have been different if we ran multiple instances of each micro service rather than a single instance by horizontally scaling them using auto scaling techniques available in the cloud. This can be an element of future research.

V. Conclusion

In this paper, we performed a quantitative analysis of performance of both event choreography and orchestration techniques used for implementing the saga design pattern to handle the distributed transactions in isolated no-SQL databases in micro service architecture. We were able to clearly determine that event choreography is much faster in performance when compared to orchestration. However, event choreography becomes very complex to code and handle if there are multiple events triggered from each micro service**.** It is also evident that handling multiple actions for the triggers without a central orchestrator is tough as one developer or team working on a micro service may not be aware of the other. This shows that event choreography is a suggested approach when there are less number of micro services participating in the distributed transaction, or the number of event triggers are not too many or when the trigger actions are not too complex. Orchestration is slow, but it is useful when the transaction scenarios are complex.

Future work includes working on scenarios involving transaction rollbacks involving dependent collections where the states are distributed in multiple collections and recording the performance metrics in various saga patterns. We also plan to do research around the areas where the triggered actions are bi-directional or cyclic rather than unidirectional and record the metrics around them. Author is thankful to anonymous reviewers for their valuable feedback.

REFERENCES

[1] R.K. Batra, M. Rusinkiewicz & D. Georgakopoulos, A decentralised deadlock-free concurrency control method for multidatabase transactions,Proc. 12th Int. Conf. on Distributed Computing Systems, 1992.

[2] N. Alshuqayran - A Systematic Mapping Study in Microservice Architecture. In Proc. of the 9th International Conference on Service-Oriented Computing and Applications. IEEE, IEEE, 2016.

[3] Paolo Di Francesco- Architecting Microservices. 2017 IEEE International Conference on Software Architecture Workshops.

[4] H. Kang, M . Le, and S. Tao, "Container and microservice driven design for cloud infrastructure DevOps," in 2 0 1 6 IEEE International Conference on Cloud Engineering (IC2E). IEEE, 2 0 1 6 , pp. 202-2 1 1.

[5] Hector Garcia-Molina, Kenneth Salem - Proceedings of the 1987 ACM SIGMOD international conference on Management of data, pages 249-259. https://dl.acm.org/citation.cfm?id=38742

[6] A.K. Elmagarmid & W. Du, A paradigm for concurrency control in heterogeneous distributed database systems, Proc. 6th Int. Conf. on Data Engineering, 1990.

[7] Messina, Antonio & Rizzo, Riccardo & Storniolo, Pietro & Tripiciano, Mario & Urso, Alfonso. (2016). The Database-is-the-Service Pattern for Microservice Architectures. 9832. 223-233. 10.1007/978-3-319-43949-5_18.

[8] 2 phase commit - https://en.wikipedia.org/wiki/Two-phase_commit_protocol.

[9] P. Bak, R . Melamed, D . Moshkovich, Y . Nardi, H. Ship, and A . Yaeli, "Location and context-based microservices for mobile and internet of things workloads," in 2 0 1 5 IEEE International Conference on Mobile Services, 2 0 1 5 , pp. 1-8.

[10] Apache Active MQ JMS - http://activemq.apache.org/

[11] P.A. Bernstein & N. Goodman, Concurrency control in distributed database systems, AGM Computing Surveys 13(2) pp. 185-222, 1981.

[12] J. P. Macker and I. Taylor. Orchestration and analysis of decentralized workflows within heterogeneous networking infrastructures. Future Generation Computer Systems, 2017.

[13] J. Tang, Using dummy reads to maintain consistency in heterogeneous database systems, Proc. Third Workshop on Future Trends of Distributed Computing Systems, 1992.

[14] Tasneem Salah, M. Jamal Zemerly, Chan Yeob Yeun, Mahmoud AI-Qutayri, Yousof AI-Hammadi.The Evolution of Distributed Systems Towards Microservices Architecture.The 11th International Conference for Internet Technology and Secured Transactions (ICITST-2016)