newt|global
Aiming At Customer Delight

info@newtglobal.com

CONTINUOUS INTEGRATION & CONTINUOUS DELIVERY

EVENT DRIVEN MICRO SERVICES

AN INTRODUCTION

There is not much of complexity in terms of processes and communications between services in a Monolithic Application that deal with a single relational database. Most of the relational database use ACID transaction to process each request from the client. It means your database will have to process 'insert, update and delete' function quite often whenever there is a change or modification made. This also provides a greater benefit especially guaranteeing –

- Changes that are made atomically
- Consistency across the Database
- Serial execution of Transactions
- Durability

In a Micro Service architecture, each service has its own database and accessing those data gets more complex whenever there is a request. To make the matter worse, micro services are developed using a polyglot environment with different micro service use different kind of database that is apt for a particular service to ensure better scalability and performance.

Let us take an example of an Order update from a customer with regards to moving his internet speed to a high speed connection. The customer service maintains information about customers, including the internet scalability in that Geo. The Order service manages orders and must verify that the new order is available for the customer within that geo. In a monolithic version of the application, the Order Service can simply use an ACID transaction to check the availability and update the order.

In contrast, in a micro service architecture the ORDER and CUSTOMER tables are private to their respective services. The Order Service cannot access the CUSTOMER table directly. Micro Services communicate only with an API on any request. In this case, how do we implement queries to retrieve data from multiple services?

USING EVENTS TO COLLABORATE

An event driven Architecture provides a better solution in terms of decoupling of services by using event messages between the sender and receiver. The biggest advantage of using an Event driven Architecture is that it results in loose coupling that makes easier to add new components to the system without needing to modify existing components. A micro service distributes an event when there is an update, other micro services subscribe to those event by triggering the next step. However, the challenge here is to maintain consistency across multiple services.

Playing around with an Event Driven Architecture is hard. There are two main approaches for an event driven communication to take place: 1) a Feed to pull domain events and 2) a Broker to distribute events. In this case, the Order Service creates an Order with status UPDATE and publishes an Order Created event through the FEED process. The Customer Service consumes the Order Created event, reserves the order and publishes a Speed Reserved event.

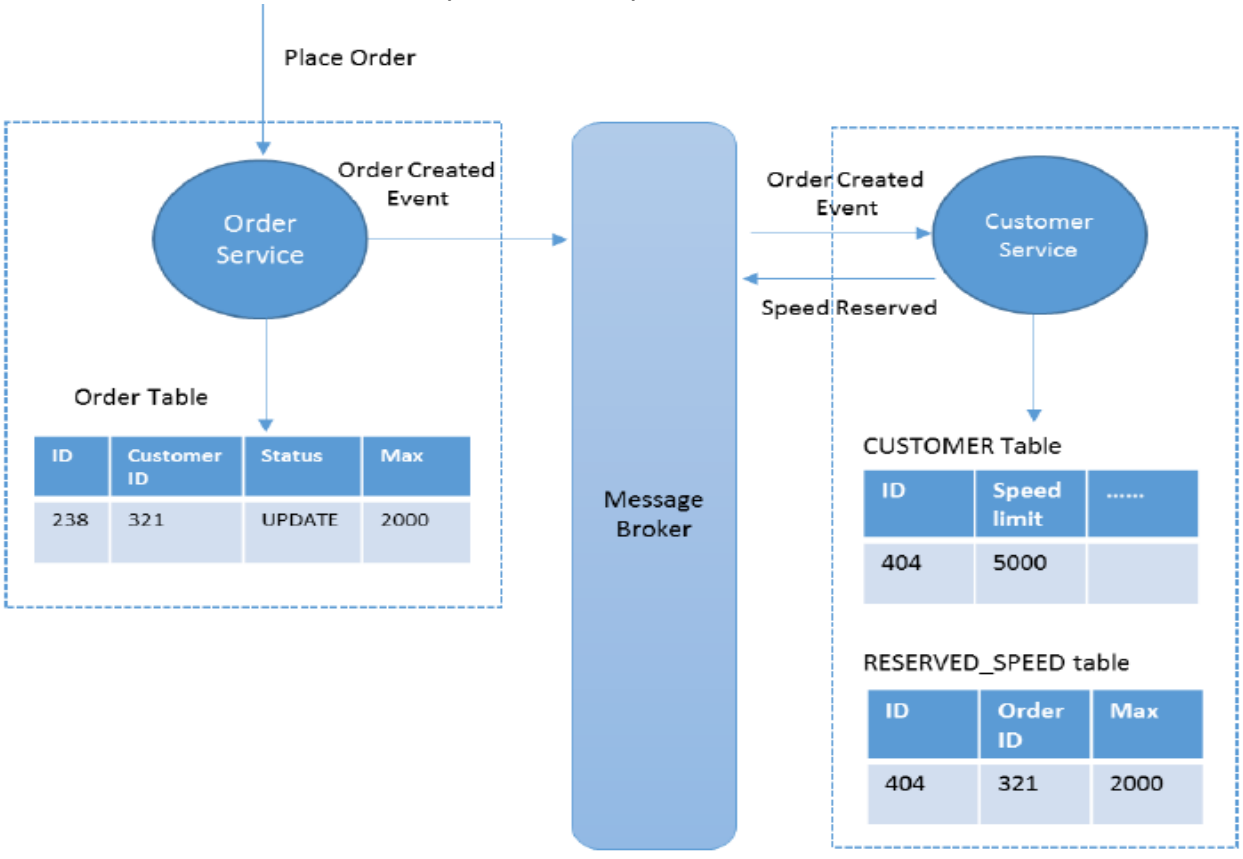

Fig.: Sample Event Driven Work Flow

A broker that is able to replay messages, that has persistent messages that is scalable and highly available – in short: you need Kafka and a Zookeeper. With an Event Driven Architecture, you need a messaging system that distributes, partitions, replicates commit log service and Kafka does that with a unique design model.

For instance, in the above mentioned diagram, the Order Service is the 'Producer', Customer Service is the 'Consumer' as part of step 1. However, as step 2, the Customer Service becomes the Producer and Order Service becomes the consumer when there is a message passing from the former end. This entire event of update is a 'Topic'. Kafka maintains the feeds of messages in categories called Topics.

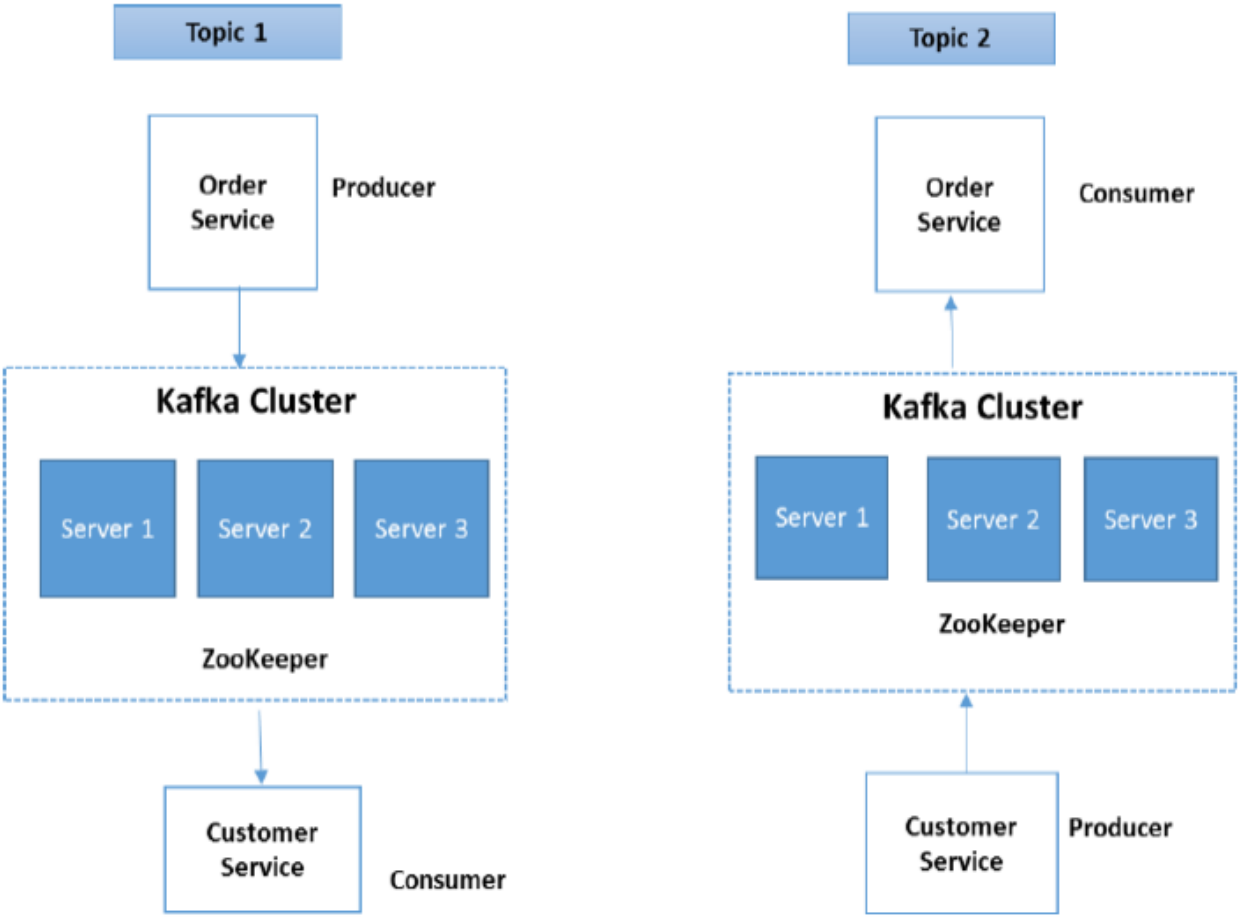

Fig.: Kafka Messaging System

Whenever, an event is triggered, Kafka runs through the Topics to make the changes happen over a distributed message system. To make this happen, you need a 'ZooKeeper' to start Kafka. Zookeeper is a server that stores all the Topics to run any message through the Kafka cluster.

CONCLUSION

Each approach has its up- and downsides. When it is not possible to run a SQL query with a micro service architecture each time when there is an update or new addition, it becomes necessity to use an Event Driven Management solution that could interact with various databases under each micro service. However, the challenge is to maintain consistency across multiple services which is very well addressed by a message queue system using a broker 'Kafka' and that can be integrated with CI / CD framework along with Jenkins, Github and SonarQube. With a broker you have more infrastructure to handle, but you also have a central place where your events are stored.