Microservice Event-sourcing

Loic Niragire

TIM – 8110 Programming Languages and Algorithms

Dr. Phillip David

Table of Contents

1. Introduction

Advancement in data structures and algorithms lies at the center of modern-day technology, from smart homes to autonomous vehicles and beyond. Successful technical applications often leverage these advancements to solve problems in a more appealing approach. Companies such as Apple, Google, Facebook, and the likes exemplify this model, where suddenly what seemed farfetched just a few years ago is now expected as the norm. For instance, your GSP system provides you directions and alerts you of live events such as accidents on your route. Apple Watch can monitor your heart and alerts you of high or low heart rates and irregular heart rhythms. Thanks to Facebook, you can now live stream events to friends all across the globe. This article explores the algorithms enabling events live-processing examples above. First, we describe an event-driven architecture with live streaming capability using Kafka, followed by a deeper exploration of Kafka's algorithms.

2. Asynchronous Event-Driven Microservice Architecture

Microservice architecture structures an application as a collection of independently deployable and loosely coupled services where each service is a process. Communication among these services is achieved either through synchronous request/response models such as HTTP-based REST or message-based asynchronous mechanisms such as AMQP (F5, 2021). Figure 1 below illustrates these communication options.

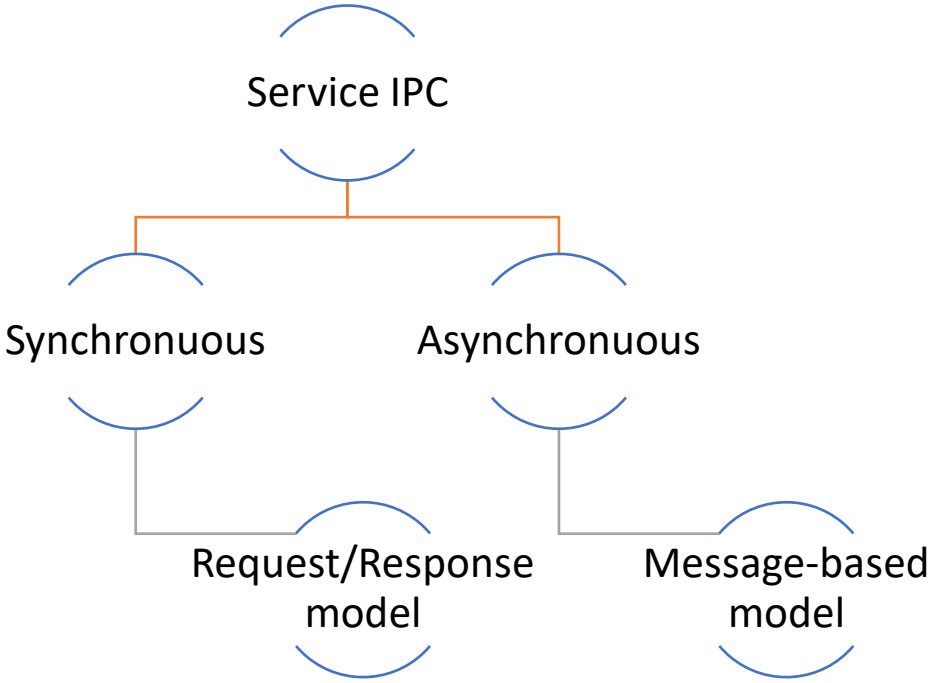

Figure 1 Service to service IPC options

In an event-based architecture, services communicate asynchronously through a message-based broker by publishing and subscribing to events. In other words, a service publishes events to a broker who is responsible for delivering those messages to all interested services. For service to receive notifications from the broker, services subscribe to specific topics on the broker.

A typical pattern applied in an event-driven microservice is for each service to maintain its database and copy the dependent services' data to increase its self-reliance. Suppose we have a "Customer" service that relies on a "Reservation" service to book a restaurant reservation. This approach suggests that the "Customer" service keeps an up-to-date copy of the "Reservation" data, thus increasing fault tolerance (Richardson, 2019). Data redundancy is one drawback of this approach - although space is relatively cheap today.

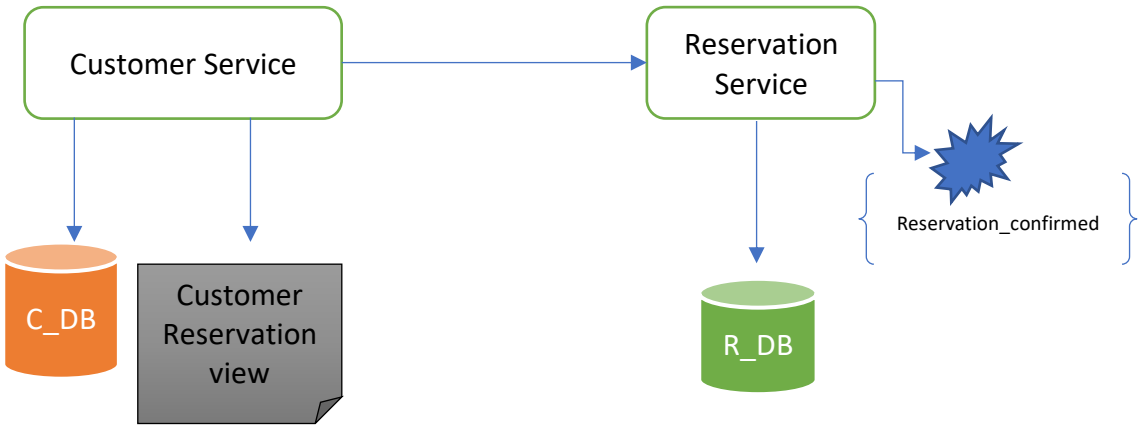

Figure 2 Service data dependency

Microservices, in event-driven architecture, use event-stores to maintain data consistency from other services. Figure 2 above illustrates a customer reservation view used to keep relevant data from the Reservation service. It is essential to highlight that this data is held for faster reads that require data from the Reservation service.

Services communicate events to an event-store such as Kafka and can subscribe to be notified of specific events. Therefore, assuming that the Reservation Service publishes a "Reservation_confirmed" event, the Customer Service could subscribe to such event and store associated data in its "Customer Reservation Data." The benefit, in this case, is that the Customer service does not need to call the Reservation service for confirmed reservations.

3. Event-Sourcing

Whenever a domain-model, or an aggregate, changes state, it must emit an event. The event store then becomes then the sore source of truth. From a functional language perspective, this is equivalent to applying events to domain models; therefore, the event must contain the data necessary to perform the state change. The business logic handles a request by applying a series of events to a domain model. The first step in this process is to determine state changes needed to

handle a given request and return a list of events representing the state changes (Richardson, Microservice Patterns, 2019). The second step in this process is to apply produced events to the original domain model.

4. Kafka: Topics and Partitions

Kafka is a distributed append-only log that acts as a broker to solve data sharing problems between services (Bejeck Jr., 2018). Whereas message brokers, such as RabbitMQ, can provide similar functionalities, Kafka maintains events in a timely, orderly fashion and retains them even after being consumed by subscribers. In other words, RabbitMQ does not guarantee that messages are received in their sending order. Kafka supports multiple logs by segregating them into topics and replicating them across the cluster. Moreover, Kafka uses "Partitions" to route incoming messages such that data with the same keys will be sent to the same consumer and in order (Richardson, Microservices Pattern, 2019).

4.1.Partition Algorithm

Partition numbers are obtained by hashing the bytes of the message key, modulus the number of partitions. Therefore, for a given message to store in a specific topic, Kafka has to first compute the partition to append the message to. Using the formular below guarantees that messages with the same key will always end in the same partition. It is worth noting that users can provide custom partitioners to Kafka.

$$\text{partition} = \text{hashCode}(\text{messageBytes}) \,\%\, \text{Number\_of\_partitions}$$

4.2.Number of Partitions

Currently, there is no specific strategy as to how to choose the number of partitions. This is a critical aspect of the success of Kafka. Moreover, when used in cluster mode, Kafka partitions from the same topic are not kept on the same machine but instead spread across devices in a cluster. With each node in the cluster referred to as a broker. This allows consumer to read a topic in parallel and thus sets Kafka up for high message throughput.

5. Statement of the Problem

Real-time data processing presents a data partitioning challenge due to the variety and velocity of data. Furthermore, according to the CAP theorem, in the presence of partitions where one set of nodes in a distributed system is not reachable from the other, access to nodes from different partitions may result in inconsistent data (Mehta & Sahni, 2018). In other words, to achieve both consistency and availability, no partitions should be allowed in the distributed system. Hence why NoSQL data stores are prevalent in distributed systems; they favor partitioning and either consistency or availability, whereas RDBMS are designed to be consistent and available. Is there a minimum number of partitions that guarantee consistency for efficient processing in a Kafka Cluster? If so, can this be determined dynamically to support real-time processing?

A Kafka Cluster consists of a leader node and one or more in-sync replicas responsible for staying up-to-date with the leader node's data. In the case a leader node goes down, the cluster elects a new leader from one of its replicas. We continuously add new replica nodes to scale out the cluster. However, the number of partitions remains static. Kafka cluster handles auto scaling out through replicas, but the number of partitions in a given node is set at the

creation of the cluster and does not auto-update as we scale-out. There are two caveats to this problem. While partitions increase throughput, having too many leads to increased replication latency, as it increases the number of open file handles. Second, by changing the number of partitions, it is hard to guarantee that keyed messages will map to the same partitions.

5.1.Research Questions & Justification of the Problem

The ability to scale a Kafka cluster and maintain high throughput while simultaneously guaranteeing that keyed messages map to the same partition. Currently, this involves a guessing strategy where the user has to predetermine future target throughput and compute the number of partitions based on that value. *Consistent hashing* is a commonly used partition scheme in distributed data stores. It is used in NoSQL databases such as Riak, Cassandra, and CouchDB. Under consistent hashing, when the number of nodes in a distributed data storage changes, all the data must be reshuffled to be compatible with the new mapping scheme (Mehta & Sahni, 2018).

A closely related concept is that of *eventual vs. strict consistency* models. In an eventual consistency model, all nodes in a cluster are guaranteed to be updated eventually. This is considered a weak form of consistency as it adds potential for data loss (Raynal, 2018). Whereas a *strictly consistent* model ensures data resiliency from node failure and always returns the latest write from any node. Table one below highlights execution steps between these data consistency models in a distributed system. Proposed partition algorithm for the problem posed in this article will have to account for both data consistency models.

Table 1 Distributed systems data consistency model

Consistency Model	Execution steps
Strict Consistency	1. Write from client 2. Write propagated through cluster 3. Internal acknowledgment 4. Acknowledged to client
Eventual Consistency	1. Write from client 2. Acknowledge to client 3. Eventual write propagation

Reference List

Bejeck Jr., W. P. (2018). Kafka Streams in Action. NY: Manning.

F5. (2021, March). NGINX. Retrieved from nginx.com: https://www.nginx.com/blog/building-microservices-inter-process-communication/

Mehta, D. P., & Sahni, S. (2018). Handbook of Data Structures and Applications. Boca Raton: CRC Press.

Raynal, M. (2018). Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach. Cham, Switzerland: Springer.

Richardson, C. (2019). Microservice Patterns. NY: Manning.