# Apache Kafka
# Course of Distributed Software Systems
# Prof. Paolo Ciancarini

Riccardo Scotti

## 1 Introduction

Apache Kafka is a distributed platform for event streaming, designed to handle real-time data, while also offering message exchange services. Originally developed at LinkedIn to serve as its main framework for event pipelining and data integration, it was later acquired by the Apache Software Foundation as part of the Apache Incubator project. It has nonetheless remained an open-source project, as it was originally conceived.

Kafka employs a distributed and scalable architecture, utilizing a publish-subscribe model for communication between data producers and consumers. Thanks to its key characteristics, such as its fault tolerant persistence method, low latency and high throughput, it has become an essential component of data pipelines.

## 2 Context

Kafka is a highly versatile and inter-operable platform; as such, its use cases can be in principle extremely heterogeneous. Below, it will be listed a short collection of the most significant scenarios where Kafka is, or could be, commonly employed.

### 2.1 Real-time data analytics

In general, Kafka is particularly well suited in all those situations where it is necessary to process and analyze large data streams in near real-time, coming from different sources within a distributed system or multiple systems. In Figure 1, it is possible to see Kafka within a data analytics pipeline, collecting real-time events from different sources in an IoT context.

An enormous amount of companies use Kafka in their systems to process real-time data [1, p. 8]. For example, at LinkedIn it is used for streaming of activity data and operational metrics, other than for delivering news feeds; at

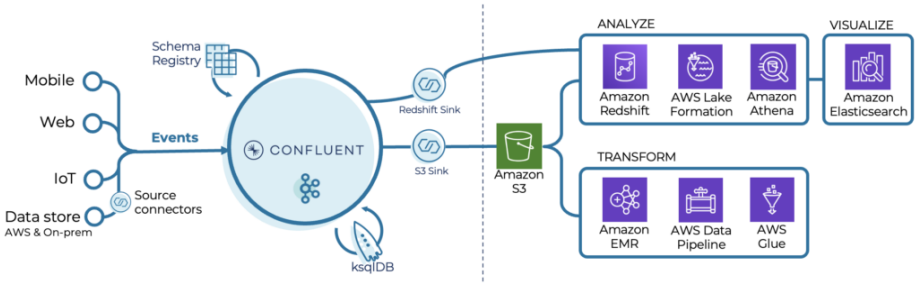


Figure 1: Example of Kafka (in its cloud version named Confluent) used in a data analytics pipeline, with data streams coming from different sources. (Source)

DataSift, it serves as a tracker of users' consumption of data streams in real time; at Twitter, it is part of the Twitter Storm data processing system.

### 2.2 Aggregated logs

Kafka's log-based architecture, which ensure a durable and consistent persistence of data, make it a perfect platform to handle the creation of aggregated logs from different data sources, a fundamental activity for any large system.

A concrete example is shown by Wang et al. in [6]. As part of the LinkedIn development team, they have built a distributed logging system by exploiting Kafka's inherent log-structured architecture; this is also a clear demonstration of the flexibility of Kafka, as it was already the backbone of many services of the social network and could be easily adapted to a newly arisen necessity.

### 2.3 Data broker

Another common use case for Kafka takes advantage of its publish-subscribe communication protocol, in order to build powerful and complex data brokers. As discussed in [4], Kafka's core mechanisms, such as topics, partitioning and replication, make it possible to create fault tolerant, asynchronous and parallel platforms which support the exchange of data between producers and consumers, allowing developers to fine tune data exchange policies.

In Figure 2, it is shown a possible scenario where two producers write data in two different topics; data is then replicated in the four Kafka brokers, which allow the two consumers to perform parallel read operations in their respective topics.

## 3 Main architectural drivers

As already mentioned, Kafka was originally developed at LinkedIn as a platform that could handle in near real time the enormous quantity of data flowing across

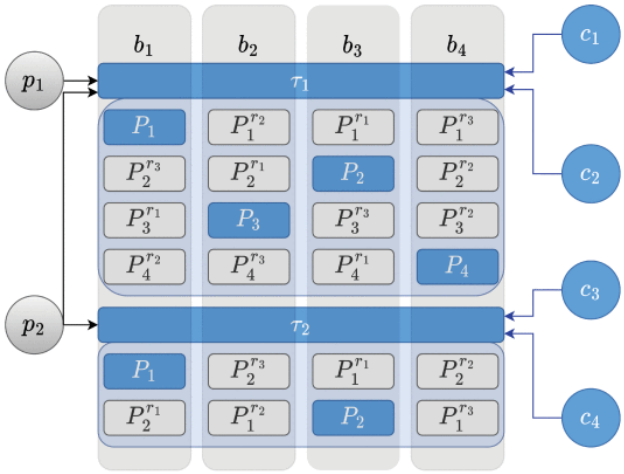


Figure 2: Kafka as a data broker. (Source: Raptis and Passarella [4])

its various services and modules; being able to ensure certain properties regarding has deeply influenced its architectural drivers, of which here we present a brief summary [1, p. 27].

**Low latency & high throughput** In order to be able to work with real time data, Kafka ensures that messages can flow as fast as possible, minimizing processing and communication delays.

**Scalability** Kafka was designed as a platform that could be integrated in large systems, receiving streams from a continuously changing pool of sources; for this reason, it is scalable by construction, meaning that it can handle larger volumes of data by easily exploiting a higher number of machines.

**Fault tolerance** As part of larger pipelines, often dealing with mission-critical, real-time data, it is not admissible to have single points of failure; therefore, Kafka is extremely resilient and can be operational even in the event of hardware failures or network issues.

**Consistency** Kafka ensures that replicas of data are kept consistent and enforces a message retention policy, which implies that multiple consumers subscribed to a same topic are guaranteed to read the same messages. Furthermore, it ensures that messages are delivered in the order they were sent.

**Interoperability** Given the necessity to integrate Kafka in complex systems composed of different technologies, its designers provided APIs for producers and consumers that support custom implementations in any language and framework.

# 4 Structure

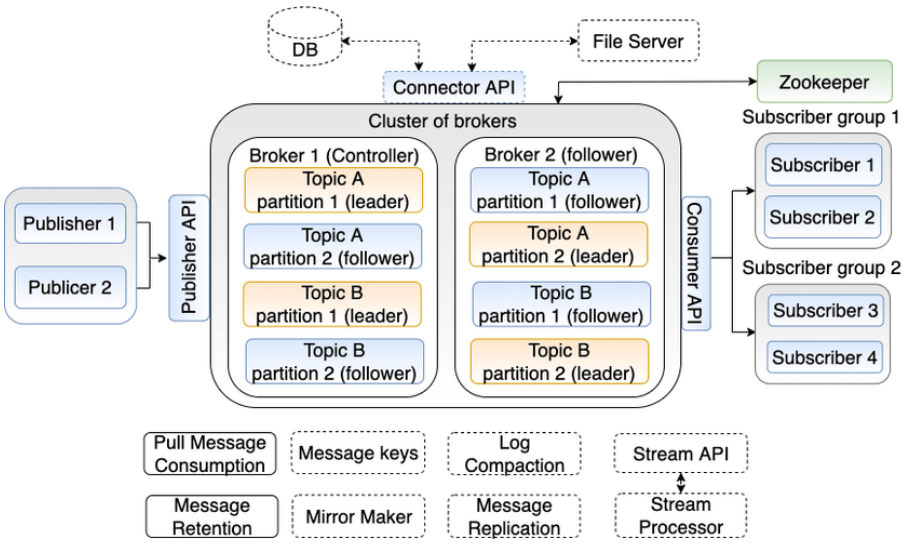


Figure 3: An overview of Kafka's architecture. (Source: Lazidis et al. [3])

We provide here a brief description of Kafka's architecture (shown in Figure 3) in terms of components and connectors.

First of all, Kafka's main architectural pattern is that of a publisher subscriber system, meaning that the sender (publisher) of messages does not specifically address a particular receiver, but it is rather the receiver (subscriber) that signals its interest for a certain class of messages (topic).

Kafka as a whole can also be described as a "distributed commit log" [5, p. 4]. It is a commit log, because it provides a durable record of all transaction, which makes it possible to replay them and build the state of the system, ensuring that data is stored durably and can be read deterministically in the order it was written. It is distributed, because data can be fragmented within the system, making the system fault tolerant and scalable.

## 4.1 Topics and partitions

A topic is a logical channel or feed name to which messages are published by producers and from which messages are consumed by consumers; it serves as a way of categorizing and organizing data streams.

Topics are broken down in partitions, which can be thought of as a single log of the distributed commit log model; they guarantee message ordering within the partition, whereas there is no guarantee of message ordering within the topic.

## 4.2 Producers and consumers

Producers and consumers are processes that, respectively, write and read data to and from one or multiple Kafka topics.

## 4.3 Brokers and clusters

A broker is a single Kafka server that receives messages from producers and services consumers by fetching published messages in the various partitions. Brokers can work in clusters, in which a single broker is automatically assigned the role of controller and is responsible for assigning and monitoring failures.

## 4.4 ZooKeeper

A core element of Kafka is the ZooKeeper module, a service for coordinating processes in distributed application, introduced in [2] by Hunt et al.

A fundamental design choice of ZooKeeper is that, instead of implementing primitives on the server side, it exposes an API that enables application developers to implement their own primitives. The so called *coordination kernel* of ZooKeeper makes it a service with remarkable flexibility, which is reflected in the flexibility of Kafka itself.

Kafka uses ZooKeeper for various tasks related to distributed systems, including maintaining configuration information, providing distributed synchronization in the form of distributed locks, and electing leaders within a Kafka cluster.

Starting with Kafka 2.8.0 in 2019, there is ongoing work to remove the dependency on ZooKeeper, replacing it with the a new controller named KRaft [5, p. 137-139]. The reason for this change is to simplify Kafka deployment architecture and reduce operational overhead, by employing a log-based controller which integrates better with Kafka and solves many of the pitfalls of ZooKeeper, such as bottlenecks. Additionally, ZooKeeper is a systems as complex as Kafka itself, and requires additional knowledge to be configured and used properly, which makes it less palatable for the community of developers who are planning to adopt Kafka.

## 4.5 Connectors

We provide here a short overview on some of Kafka's main connectors.

**Kafka Connect** Provides a scalable and fault-tolerant way to integrate Kafka with various data sources and external systems, such as databases and messaging systems.

**Source and sink connectors** Provide APIs to implement producers and consumer in external systems.

**Connect workers** Processes responsible for running connectors and managing their life cycles, handling configuration, execution and scaling.

# 5 Behavior

This section delves into the internal dynamics of Kafka, examining the mechanisms that regulate the behavior of its most relevant components.

## 5.1 Production

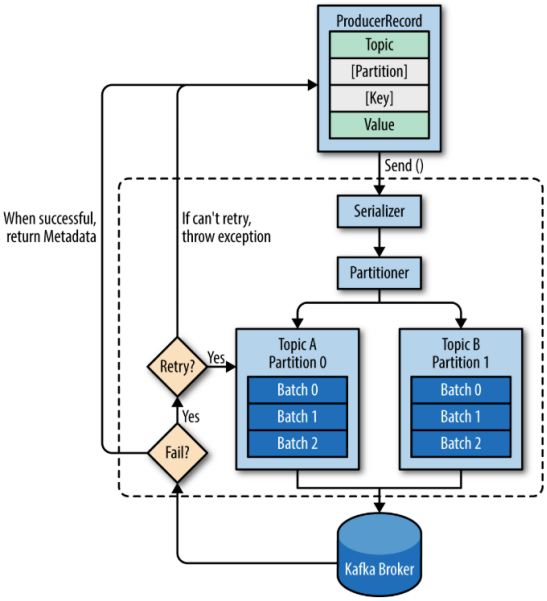


Figure 4: Production of a message in a topic (Source: Shapira et al. [5])

The process of production, as illustrated in Figure 4, starts with the creation of a `ProducerRecord`, a data structure that, apart from the payload, contains information about the topic and the partition. A producer can either choose to balance messages across all partitions or target a specific one; it is also possible to specify ad hoc partitioners, when a more complex business logic requires so.

When the broker receives the message, it sends back a response containing information regarding topic, partition and offset within the partition; in case of failure, it sends back an error, which triggers a retry in the producer.

## 5.2 Consumption

Consumers subscribe to one or more topics and read messages in the order they were produced in each partition. To do so, each message has an associated offset, related to the order of production; by keeping track of the offset of the last message consumed for each partition, a consumer can resume its work in case of stop or failure.

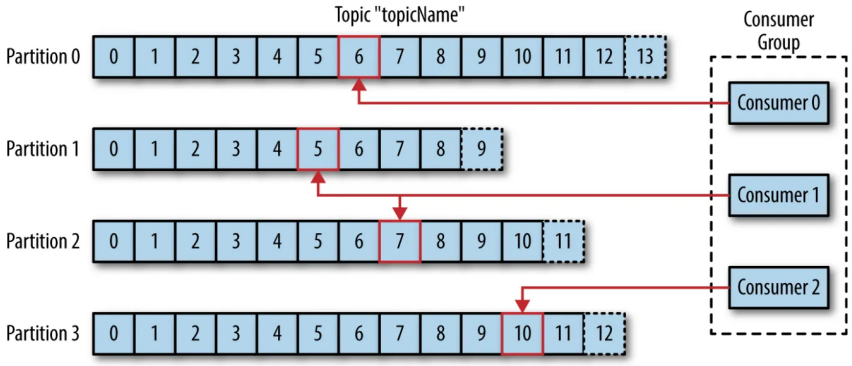


Figure 5: Consumer group working on a topic. (Source: Shapira et al. [5])

A consumer group is a set of consumers that read messages from a same topic, with each partition consumed by no more than one consumer, although a single consumer may have access to multiple partitions, as shown in Figure 5. In the event of a failure of one of the consumers in the group, its partitions will be redistributed to the remaining ones.

## 5.3 Brokers and clusters

Brokers play a central role in managing storage and retrieval of messages in Kafka topics; as such, each broker is responsible for one or more partitions. A broker that owns a certain partition is called *leader* of that partition, while brokers that contain replicas of that partition are named *followers*, as shown in Figure 3 and Figure 6.

In order to write in a specific partition, producers must connect to the leader, while consumers can read messages by connecting to any follower broker of that partition. Brokers are also responsible for consumer coordination, a load balancing task that assigns different partitions to different consumers within a group, ensuring that each message is processed by only one consumer in the group.

Apart from serving requests and exchanging data among producers and consumers, Kafka brokers are also able to handle persistence, according to a configurable parameter called *message retention*; messages are thus kept in memory for a certain amount of time before being deleted.

Finally, brokers maintain metadata about topics, partitions, and consumers. This metadata is used for coordination, routing, and maintaining the overall health of the Kafka cluster.

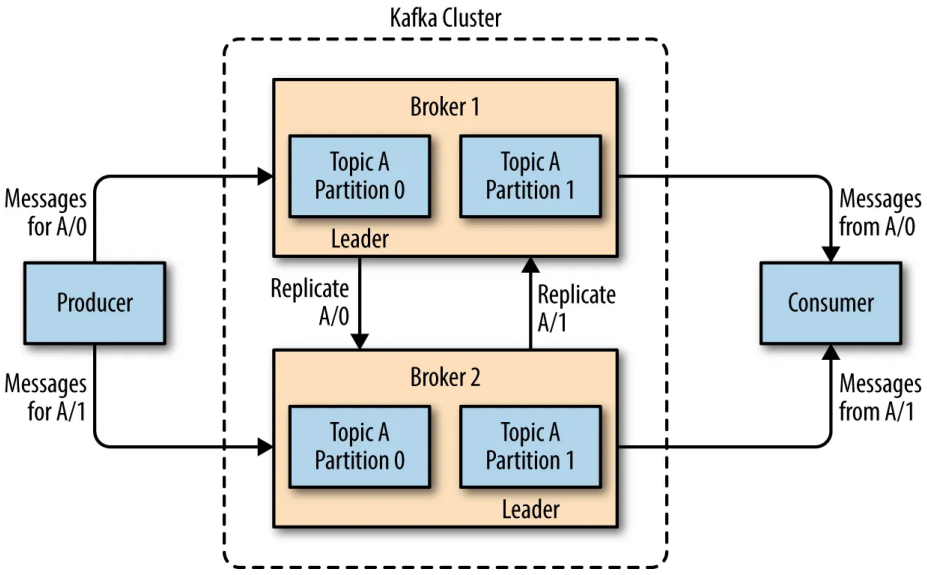


Figure 6: Replication of partitions in a cluster. (Source: Shapira et al. [5])

### 5.4 ZooKeeper

The ZooKeeper module takes care of coordination between the other components of Kafka. For example, it keeps track of which brokers are controllers, and in case of failure elects a new one; it stores metadata so that, when producers and consumers need them, they can query it and finally it assigns partitions

Despite being such a critical component for Kafka, ZooKeeper does not have a fixed role and its behavior has often changed, depending on what was popular at that time in the Kafka community. For instance, ZooKeeper used to take care of helping consumers to keep track of their received offsets, but later on it became commonplace to store this information in a dedicated topic; it was also necessary to query ZooKeeper to create new topics, but in newer versions of Kafka it is possible to do so by directly querying a broker [5, p. 148].

## 6 Rationale

In this section, there will be an examination regarding the reasons behind some of the design choices of Kafka.

**Publisher/subscriber model** Makes Kafka suitable for building real-time data pipelines, enabling seamless communication between producers and consumers and providing space decoupling.

**Partitions and distributed commit log** Ensure horizontal scalability, high availability, fault tolerance and allow read operations in parallel access.

**Consumer groups** Enables consumers to scale horizontally, in order to consumer topics with a high message volume.

**Log-based architecture** Such representation of data as stream of events allows multiple consumers to quickly catch up to the latest states by replaying events, for example after a failure. The log establishes a clear ordering between events and ensures that the consumers always move along a single timeline.

**Retention policy** Ensures data durability, which may be useful for certain application that require to store data for a certain amount of time (e.g. aggregated logs and monitoring).

# 7 Similar or competing middlewares

As shown in section 2, Kafka offers a wide variety of functionalities and can be employed in different scenarios; therefore, there are numerous middlewares that are similar in terms of use cases.

## 7.1 RabbitMQ

RabbitMQ is a messaging framework that, similarly to Kafka, acts as a broker between producers and consumers, providing point-to-point message queues and offering also load balancing functionalities.

RabbitMQ does not have the so called message retention policy, meaning that, differently from Kafka, multiple consumers cannot subscribe to a same topic and read the same message, as messages are deleted after consumption. Furthermore, it possesses a simpler architecture (message queues) than Kafka's partitioned log model. Lastly, in RabbitMQ it is possible to flag messages as "high priority", while in Kafka message ordering is always strictly respected.

## 7.2 Apache Pulsar

Apache Pulsar is a cloud-native, publisher-subscriber based platform for message exchange, streaming and queueing.

It has a similar but more complex architecture than Kafka; on top of the Pulsar servers and the ZooKeeper, it has also a RocksDB database and a BookKeper, which takes care of long-term storage of data. Because of this, Pulsar is easily scalable, but performances tend to be poorer if compared to the simpler and therefore faster Kafka architecture.

As a newer, less mature platform, with fewer resources and documentation and a smaller community, Pulsar is still behind Kafka in terms of usage in industry.

### 7.3 NATS

NATS is a lightweight, cloud-native, message-oriented middleware, specifically designed for message exchange between publishers and subscribers in scenarios where low latency is crucial.

Developed with high performances and quality of service as a driving principle, it offers the bare minimum in terms of functionalities, to ensure reliable message exchange between applications with extremely low latency. Since NATS is primarily message-oriented, it does not offer message persistence; NATS Streaming, built on top of NATS, while providing message persistence, does not scale up as well as Kafka does.

Despite being more limited and less versatile, NATS' high performances and fast message delivery make it a perfect middleware for mission-critical applications, communication between microservices and IoT pipelines.

## References


[1] Garg, N. *Apache kafka*. Packt Publishing Birmingham, UK, 2013.

[2] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. {ZooKeeper}: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)* (2010).

[3] Lazidis, A., Tsakos, K., and Petrakis, E. Publish-subscribe approaches for the iot and the cloud: Functional and performance evaluation of open-source systems. *Internet of Things 19* (05 2022), 100538.

[4] Raptis, T. P., and Passarella, A. A survey on networked data streaming with apache kafka. *IEEE Access 11* (2023), 85333–85350.

[5] Shapira, G., Palino, T., and Sivaram, R. *Kafka - the definitive guide*, 2 ed. O'Reilly Media, Sebastopol, CA, Nov. 2021.

[6] Wang, G., Koshy, J., Subramanian, S., Paramasivam, K., Zadeh, M., Narkhede, N., Rao, J., Kreps, J., and Stein, J. Building a replicated logging system with apache kafka. *Proc. VLDB Endow. 8*, 12 (aug 2015), 1654–1655.