Detailed Design of Kafka

Understand the detailed design of each component in Kafka.

We'll cover the following

Kafka is a messaging system that includes a sender that publishes messages (data) not specifically to a receiver but assigns a type to them and a receiver that subscribes to a certain type of messages. Kafka also has a broker that facilitates both the sender and receiver publishing and subscribing to messages.

Kafka's Architecture

Kafka is composed of three main components, which are described as follows.

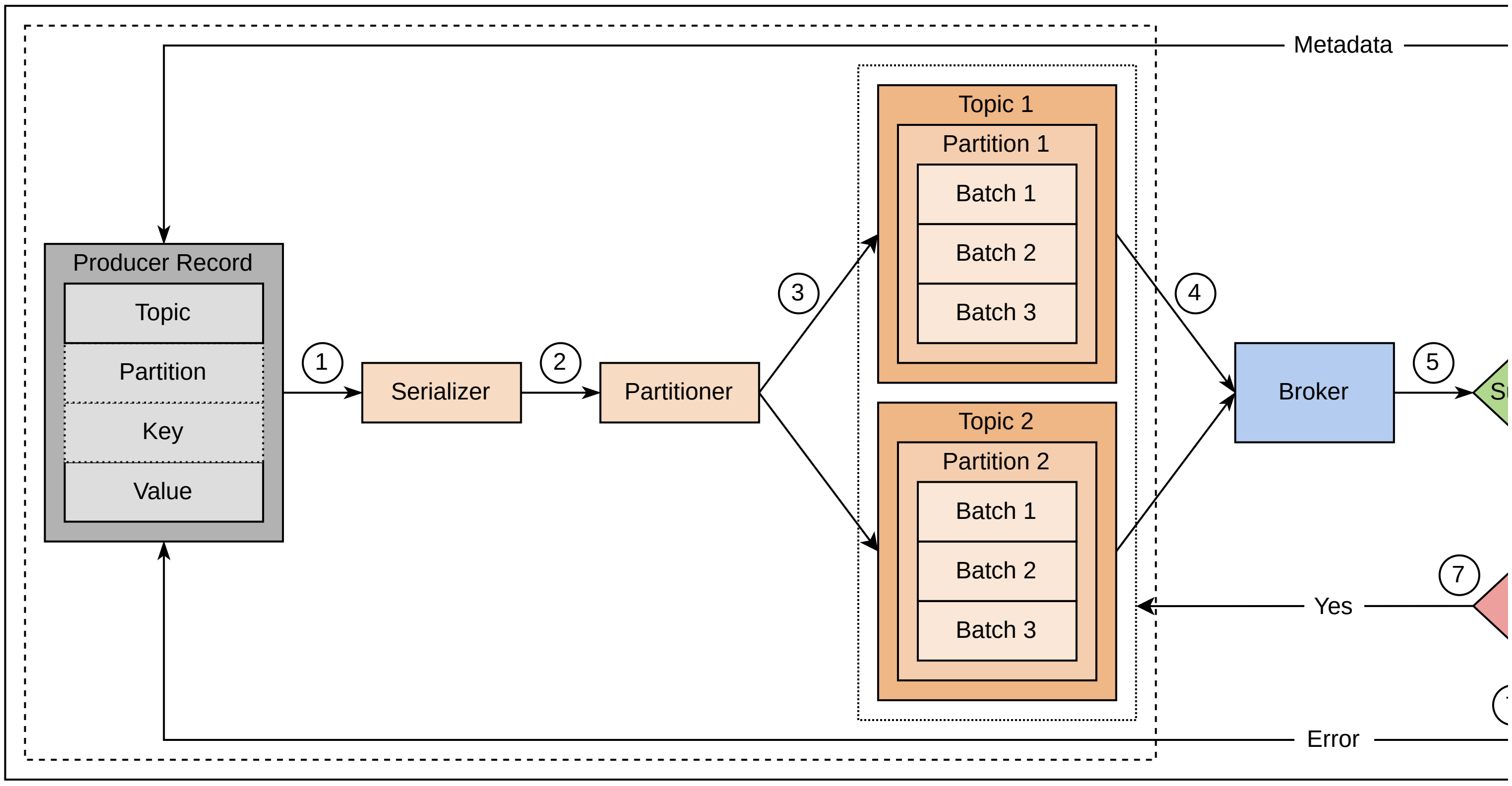
Producers

A **producer** is responsible for creating new messages. They contain the following components:

- **Producer record:** A record is kept for each message created in the producer record. It is composed of the topic that's supposed to get the message, a key or partition, and a value.
- **Serializer:** The first thing a producer does on a message is that it converts the key and value of a message into byte arrays (this process is sometimes called the **marshaling of data**).

- **Partitioner:** After being serialized, the message goes to the partitioner, which returns the partitions of a specific topic to which the message should be assigned to. Following are a few ways a partitioner can operate on messages:
  - If we specify a key in the producer record, the partitioner uses a hash function on the message key and maps it to a specific partition.
  - If we specify a partition in the producer record, the partitioner doesn't do anything, and the message is assigned to that specific partition.

By default, we get one partition for each topic. However, the number of partitions is a parameter that can be altered by the user (each partition gets its own ID). A good practice is to select the number of partitions equal to or a multiple of the number of brokers in the cluster. This allows equal distribution of partitions to the brokers.



Components of the Kafka producer

The producer knows which topic and partition the message should be written to after the partitioner gives out the partition. The producer then keeps these messages as batches in partitions of topics in a buffer.

The producers send **produce requests** that are responsible for sending the buffered batches to specific Kafka brokers.

Brokers

Multiple servers are working in Kafka. They receive messages from the producers and deliver them to the consumers upon a pull request. An individual server is also known as a **broker**. When a broker receives messages from producers, it performs the following tasks:

- Assigns an offset to the message
- Stores them on a disk
- Sends a response back to the producer
    - If a message is successfully written, it sends metadata of its record that includes topic, partition, and offset.
    - If a message is not successfully written, it sends back an error to the producer. The producer tries to send the message again a few more times. If still unsuccessful, it returns an error.

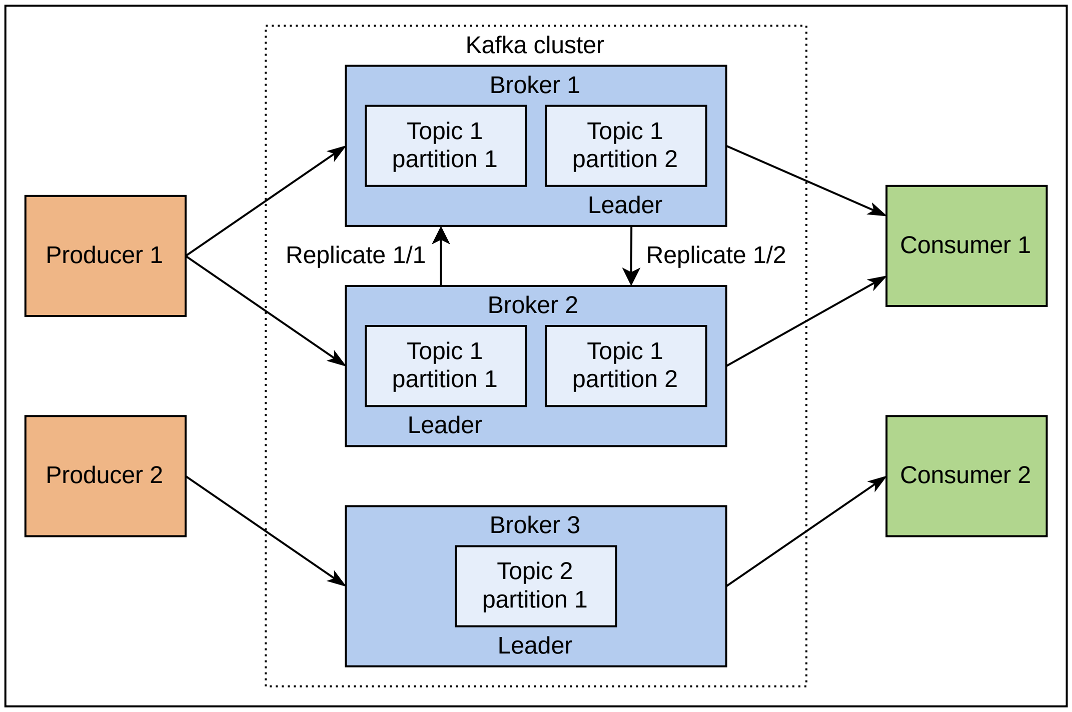
A broker can handle millions of messages and thousands of partitions per second, given that it matches specific hardware requirements. Brokers are operating in a cluster. One of the brokers in a cluster is elected as a cluster controller by the ZooKeeper employed by Kafka, the details of which are discussed later. A cluster controller performs the following functions:

- Assigns partitions to brokers
- Elects partition leaders
- Monitors broker failures

There are some benefits we get by putting multiple brokers in a cluster.

- **Scalability:** Data load can be scaled across multiple servers/brokers.
- **Replication:** Partitions can be replicated to provide fault tolerance in case a server fails. The number of times a partition is replicated is decided by the

user.



A Kafka cluster

Replicas of a topic partition can be assigned to multiple brokers. Each partition should be assigned to a different broker to ensure fault tolerance. However, only one of the brokers that has the partition interacts with each consumer and producer, and that broker is called the leader of that partition. If a leader broker fails, another broker is assigned as the partition's leader. All the producers and consumers working on that partition must be connected to the partition leader.

Consumers

Consumers are also known as subscribers or readers in some other messaging systems because they can subscribe to multiple topics, and they are used to read messages from the partitions of a topic. They read the messages in the following way:

- They subscribe to one or more topics.
- They send pull requests of subscribed topics to the brokers, which have the information regarding the data location, and it responds with messages it has kept on a disk.

- They read messages from the partitions of a topic in the order they were written.
- They keep track of the messages they have already consumed through offset.

An **offset** is a unique integer value attached to every message as metadata in a partition. It is incremented every time Kafka gets a new message in a partition, and that incremented value is then attached to the new message as metadata. By keeping the offset of the last consumed message in Kafka, stopping and restarting the process is possible without losing track of the messages that are supposed to be read. The offset is discussed in detail in the following lessons.

The consumers send **pull requests** that are responsible for getting the buffered batches from specific Kafka brokers.

The produce and pull requests should be sent to the leader partition, which is a single partition residing on only one of the brokers in the cluster. If the wrong broker gets the requests, it gives an error stating, "Not a leader for partition." So, both the producers and consumers should send requests to the broker that contains the leader partition for a specific partition.

Question

How do produce and pull requests know where (that is, which broker) to send the request?

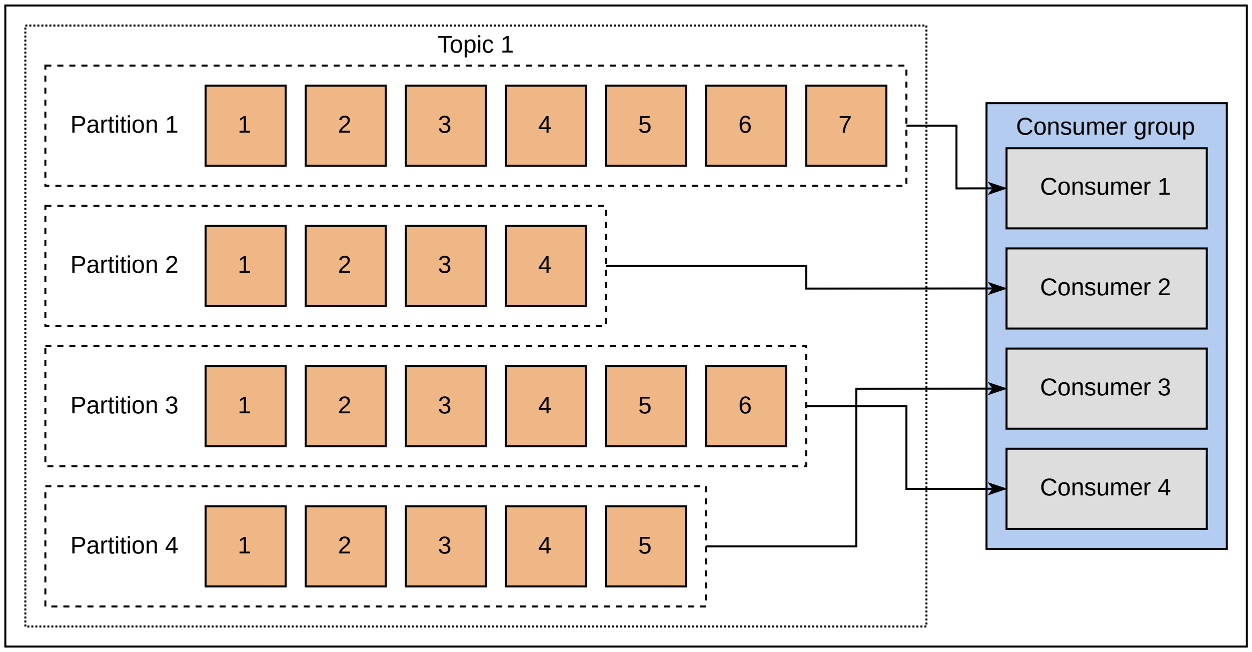
Hide Answer

Producers and consumers send a metadata request to the brokers, including a list of topics of interest. The brokers give back a response, including partitions in the topics, replica partitions, and leader partitions. This request can be sent to any broker because they all have cached metadata. The producers and consumers sending this request also cache this metadata to avoid sending requests to the wrong brokers and direct future requests correctly.

Consumer groups

Multiple consumers operate in consumer groups. A consumer group operates in the following way:

- A topic is consumed by one or more consumers working in a consumer group.
- It ensures that each partition is consumed only by a single member of the consumer group. Therefore, a member can consume messages from multiple partitions. Hence, for a topic that has multiple partitions, one of many consumers will read each of them, but for one partition, only one consumer will read it.
- A partition being consumed by a member comes under the ownership of that particular member.



Consumption of partitions

Question

Why do we need consumer groups when a single consumer can consume messages?

Hide Answer

Suppose that we have to get data from a topic. We'll have to create a consumer object, subscribe it to the topic and start fetching the data. However, there are two reasons why consuming data with a single consumer might be a bad idea:

- What if the producer is publishing messages at a rate higher than the rate at which the consumer is fetching messages? This will cause the system to lag further and further behind and Kafka will need to store more messages in itself as well.
- What if a member of a consumer group fails? (**failures are also detected by the Zookeeper**, which is explained later on)

To cater to these two problems having a consumer group is necessary because they enable horizontal scaling of consumers for consumption of topics with a large number of messages. Furthermore, in case of failures, other consumers can take over and divide the partitions consumed by the missing member.

In this lesson, we learned how the data is published by the producers and stored by the brokers. We also discussed how it is readily available for consumers to consume.

Back

High-level Design of K...

Next

Efficiency of Kafka

Mark as Completed