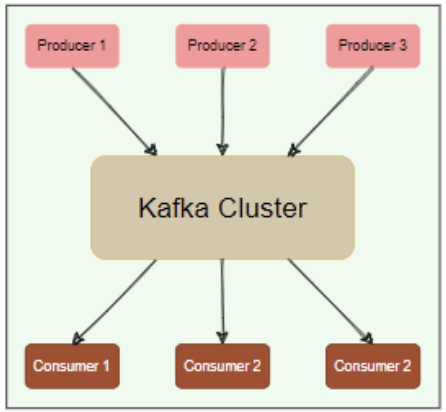
Kafka : Introduction

Kafka First view as a Learner

This is the basic view of Kafka Messaging System.

For understanding you can divide this into 03 parts

- Producers
- Kafka Cluster
- Consumers/Consumer Group



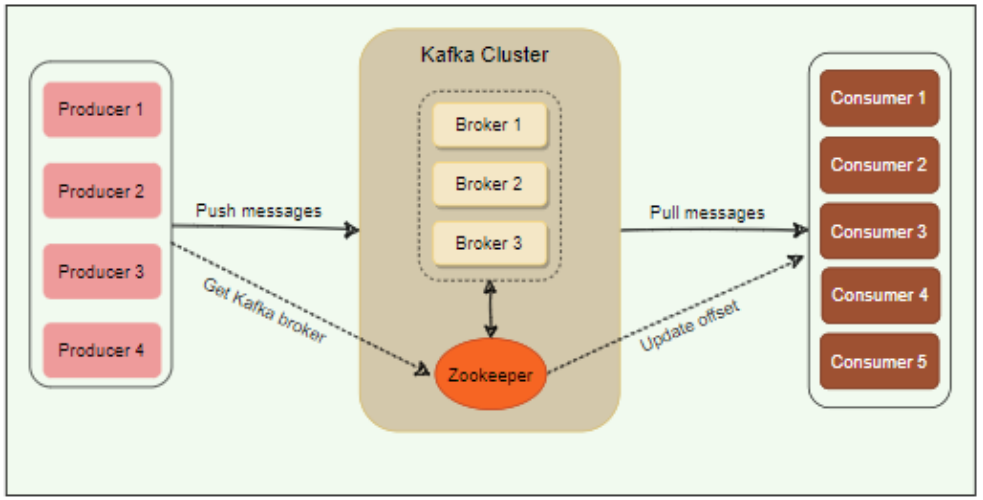
At higher level, major use-cases where Kafka is very useful are listed below, but note that
It is just small list, there are n number of addition in use-cases possible.
So, I am no way saying that it is the exhaustive list.
1.Stream processing
2.Metrics
3.Log Aggregation
4.Commit Log
5.Website activity tracking
6.Product suggestion/recommendations
7.Microservice communications etc.
And so on…

Kafka high level architecture overview
So here in this diagram, I added some interaction between the parts and Zookeeper.

At a high level, applications (producers) send messages to a Kafka broker, and these messages are read by other applications called consumers. Messages get stored in a topic, and consumers subscribe to the topic to receive new messages.

Kafka cluster
Kafka cluster deployed as one or more servers, where each server is responsible for Running one Kafka broker.

Zookeeper
Zookeeper is a distributed key-value store for managing the coordination of Kafka brokers majorly along with some level communication between Kafka parts.
Will see more detailed analysis in coming pages.



Kafka : Deep Dive

Kafka: Deep Dive
Kafka is collection of topics in simple term. As topic can go bigger in size so the best approach is to partition them for better performance & scalability.

Topic Partitions
As explained above, for one topic we may have n numbers of partition. Now next is how & who will decide that the current message or record will be pushed in which partition?

Now, as per Kafka architecture, producer have ownership in deciding the topic in which it will be pushed. Next thing, how they will decide? The decision will be based on data of message or record.

Just For example, a producer can decide that all messages related to a particular 'city' go to the same partition.

Essentially, a partition is an ordered sequence of messages. Producers continually append new messages to partitions.

@Note – Ordering of messages is maintained at the partition level, not at the topic level.

Offset: A unique sequence id called an offset gets assigned to every message that enters a partition.

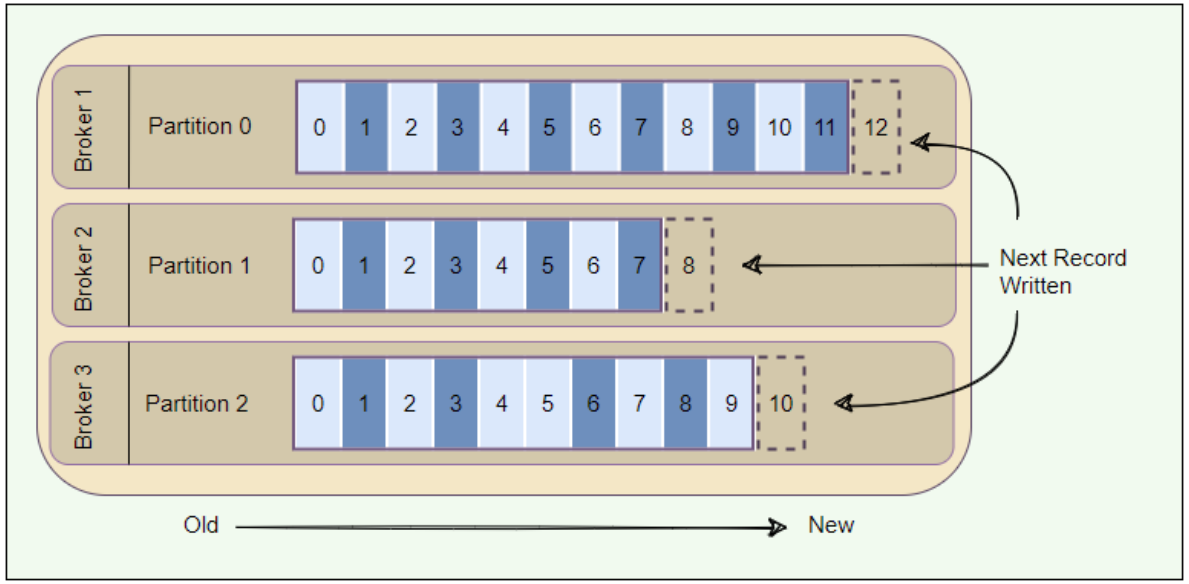
These numerical offsets are used to identify every message's sequential position within a topic partition.

Offset sequences are unique only to each partition. So, to get the message you must know 03 information, **topic**, **partition**, and **offset**.

Next is how producer will choose the partition for their message or record?

Messages once written to partitions are immutable and cannot be updated.

Each broker manages a set of partitions belonging to different topics.



Approach 01 (**Round robin**) : if message order within the partition is not important than best approach is Round robin for equal distribution.

Approach 02 ( ) : Let's think some scenario where ordering of message also matters within the Partition, in that case you must use some strategy to push messages within the same partition so that it will be in order as expected. And that might be the data of message itself.

In this case producer add key to message and every time producer make sure that message with the same key should go to the same partition.

Kafka : Deep Dive

Contd...

Kafka follows the principle of dumb broker and smart consumer.
It means that broker is not keeping track about the message consumed by consumer. Instead, consumer itself tracking the message and offset of their interest. Meaning every such control is with smart consumer itself. They should know from where they need to start fetching messages.

By this way broker and consumer are decoupled in this interaction.

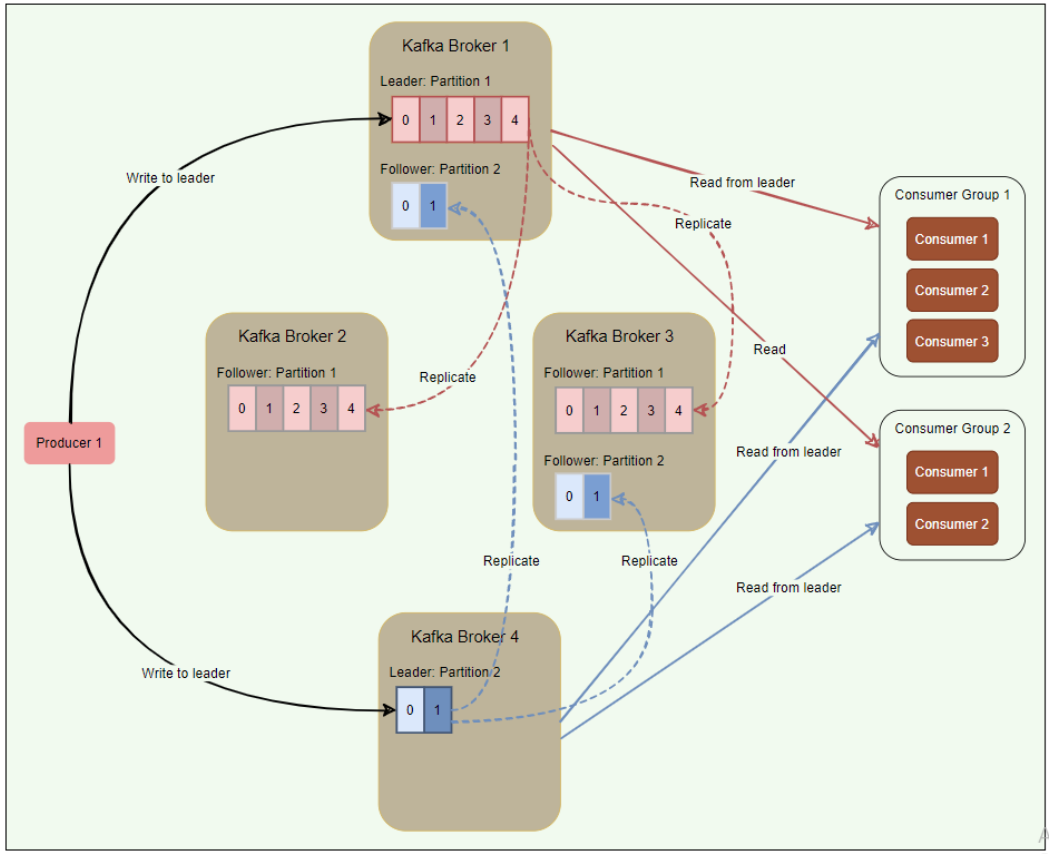
Every topic can be replicated to multiple Kafka brokers to make the data fault-tolerant and highly available. Each topic partition has one leader broker and multiple replica (follower) brokers.

Leader
A leader is the node responsible for all reads and writes for the given partition. Every partition has one Kafka broker acting as a leader.

Follower
To handle the SPF (Single Point of Failure), Kafka replicates partition and distribute across multiple broker servers are called followers. Also, if leader fails than out of followers any partition can be promoted as leader partition.

For now, Kafka stores the location of the leader of each partition in Zookeeper. As all writes/reads happen from leader partition so producers and consumers directly talk to find the partition leader.

There are some changes in latest version of Kafka regarding the leader finding approach, but for now assume this is as written in above paragraph.

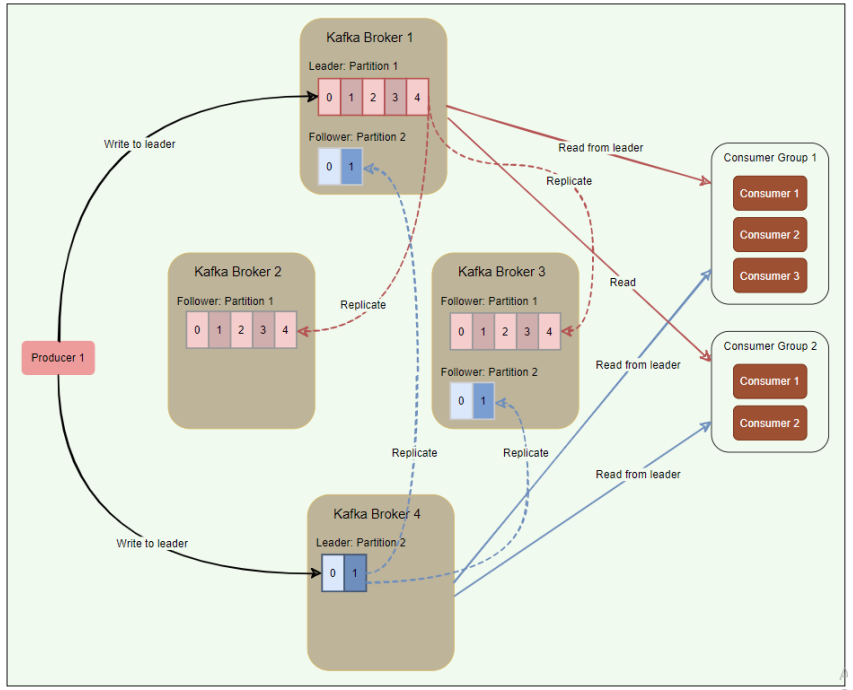


Kafka : Deep Dive

Contd...

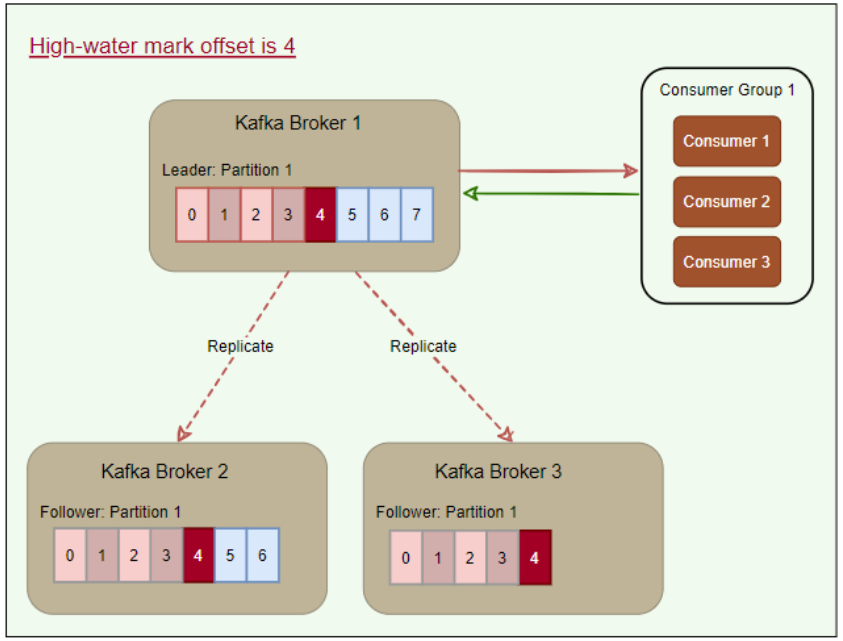
In Sync replica (ISR) :

ISR is a broker having latest data for a given partition. So, by this logic **Leader is always an ISR**. A follower is an in-sync replica only if it has fully caught up to the partition it is following. **Another key rule here is that only ISR is eligible to become leader partition**.



High-water mark (a system design pattern) :

To ensure data consistency, the leader broker never returns (or exposes) messages which have not been replicated to a **minimum set of ISRs.** For this, brokers keep track of the high-water mark, which is the highest offset that all ISRs of a particular partition share. The leader exposes data only up to the high-water mark offset and propagates the high-water mark offset to all followers.



Kafka : Deep Dive

Consumer Group (multiple smart consumer) :
A consumer group is basically a set of one or more consumers working together in parallel to consume messages from topic partitions.

Messages are equally divided among all the consumers of a group, with no two consumers receiving the same message.

Kafka ensures that **only a single consumer reads messages from any partition within a consumer group**.

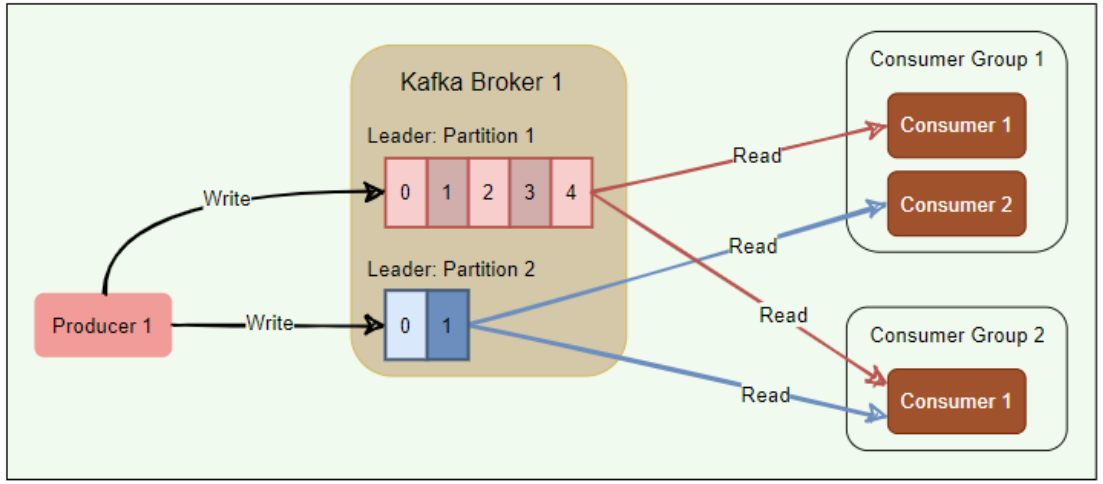
Consumers pull messages from topic partitions. Different consumers can be responsible for different partitions.

By using consumer groups, consumers can be parallelized so that multiple consumers can read from multiple partitions on a topic, allowing a very high message processing throughput.

The number of partitions impacts consumers' maximum parallelism, as there cannot be more consumers than partitions.

Kafka stores the current offset per consumer group per topic per partition, as it would for a single consumer. This means that unique messages are only sent to a single consumer in a consumer group, and the load is balanced across consumers as equally as possible.

Kafka uses any unused consumers as failovers.



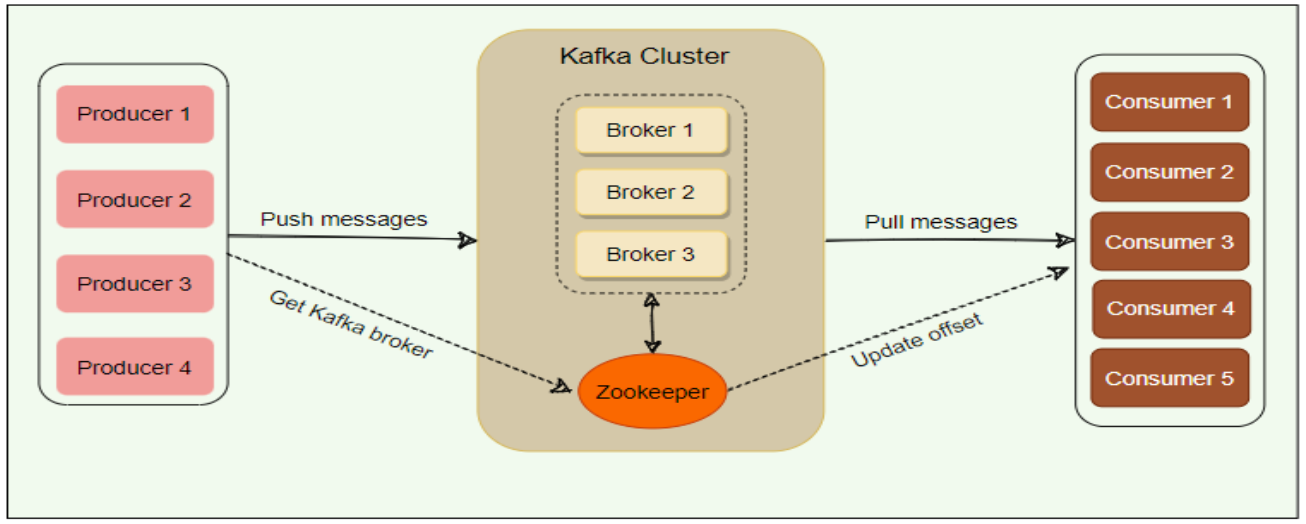
Here is a summary of how Kafka manages the distribution of partitions to consumers within a consumer group:

Number of consumers in a group = number of partitions: each consumer consumes one partition.

Number of consumers in a group > number of partitions: some consumers will be idle.

Number of consumers in a group < number of partitions: some consumers will consume more partitions than others.

Kafka : Role of Zookeeper

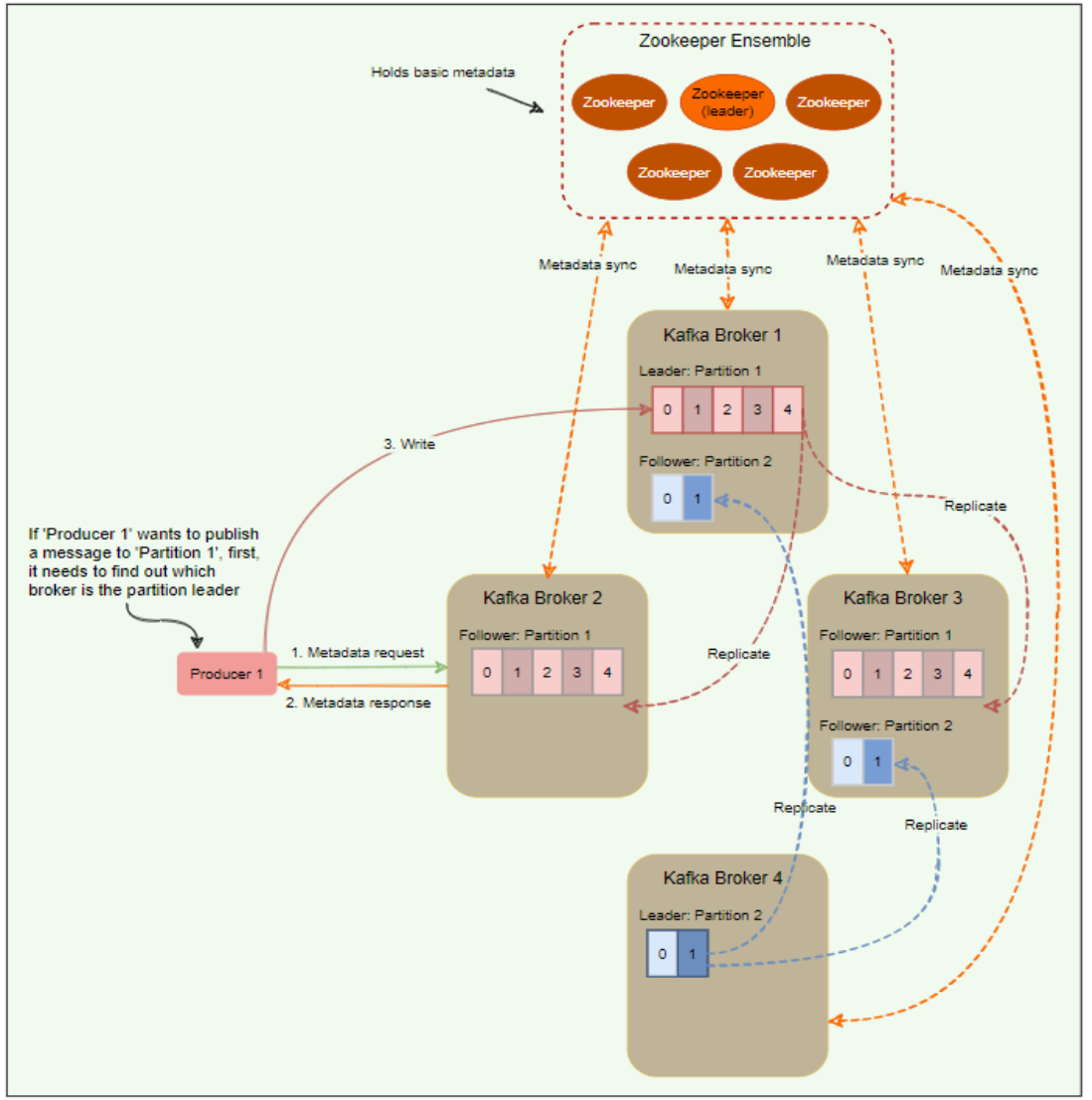


Zookeeper serves as the coordination interface between the **Kafka brokers, producers, and consumers.** Kafka **stores basic metadata in Zookeeper**, such as information about brokers, topics, partitions, partition leader/followers, consumer offsets, etc.

Zookeeper is used for storing all sorts of metadata about the Kafka cluster:

- ✓ It maintains the last offset position of each consumer group per partition, so that consumers can quickly recover from the last position in case of a failure **(although modern clients store offsets in a separate Kafka topic).**
- ✓ It tracks the topics, number of partitions assigned to those topics, and leaders'/followers' location in each partition.
- ✓ It also manages the access control lists (ACLs) to different topics in the cluster. ACLs are used to enforce access or authorization.

How to identify the leader of the partition?



Kafka : Role of Zookeeper

How to identify the leader of the partition? Contd…

In the older versions of Kafka, all clients (i.e., producers and consumers) used to directly talk to Zookeeper to find the partition leader. Kafka has moved away from this coupling, and in Kafka's latest releases, clients fetch metadata information from Kafka brokers directly; brokers talk to Zookeeper to get the latest metadata.

All the critical information is stored in the Zookeeper and Zookeeper replicates this data across its cluster, therefore, failure of Kafka broker (or Zookeeper itself) does not affect the state of the Kafka cluster. Upon Zookeeper failure, Kafka will always be able to restore the state once the Zookeeper restarts after failure. Zookeeper is also responsible for coordinating the partition leader election between the Kafka brokers in case of leader failure.

Role of Controller Broker in Kafka

Within the Kafka cluster, one broker is elected as the Controller. This Controller broker is responsible for admin operations, such as creating/deleting a topic, adding partitions, assigning leaders to partitions, monitoring broker failures, etc.

It also communicates the result of the partition leader election to other brokers in the system.

What is Split Brain problem?

Suppose your controller broker observed as nonresponsive for some temp intermediate network low bandwidth and Zookeeper just promoted another broker as controller. After some time, your earlier controller again come-up and start working properly. Now in this situation the earlier controller become Zombie controller. So, at this point we have two controller working simultaneously and this is the situation which is considered as Split-Brain problem. And in this situation our system state is very inconsistent. **So, how do we handle this situation?**

Split-brain is commonly solved with a **generation clock**, which is simply a monotonically increasing number to indicate a server's generation. In Kafka, the generation clock is implemented through an **epoch number**. If the old leader had an epoch number of '1', the new one would have '2'. This epoch is included in every request that is sent from the Controller to other brokers. This way, brokers can now easily differentiate the real Controller by simply trusting the Controller with the highest number. The Controller with the highest number is undoubtedly the latest one, since the epoch number is always increasing. This epoch number is stored in Zookeeper.

Kafka : Message delivery Semantics from producer

There are two angle of delivery semantics

- ✓ **Producer delivering message to Kafka partitions**
- ✓ **Consumer group is reading message from Kafka partitions**

Async: Producer sends a message to Kafka and does not wait for acknowledgment from the server. This means that the write is considered successful the moment the request is sent out. This fire-and-forget approach gives the best performance as we can write data to Kafka at network speed, but no guarantee can be made that the server has received the record in this case.

Committed to Leader: Producer waits for an acknowledgment from the leader. This ensures that the data is committed at the leader; it will be slower than the 'Async' option, as the data must be written on disk on the leader. Under this scenario, the leader will respond without waiting for acknowledgments from the followers. In this case, the record will be lost if the leader crashes immediately after acknowledging the producer but before the followers have replicated it.

Committed to Leader and Quorum: Producer waits for an acknowledgment from the leader and the quorum. This means the leader will wait for the full set of in-sync replicas to acknowledge the record. This will be the slowest write but guarantees that the record will not be lost if at least one in-sync replica remains alive. This is the strongest available guarantee.

Kafka : Message delivery Semantics to consumer group

There are three ways of providing consistency to the consumer:

At-most-once (Messages may be lost but are never redelivered): In this option, a message is delivered a maximum of one time only. Under this option, the consumer upon receiving a message, commit (or increment) the offset to the broker. Now, if the consumer crashes before fully consuming the message, that message will be lost, as when the consumer restarts, it will receive the next message from the last committed offset.

At-least-once (Messages are never lost but maybe redelivered): Under this option, a message might be delivered more than once, but no message should be lost. This scenario occurs when the consumer receives a message from Kafka, and it does not immediately commit the offset. Instead, it waits till it completes the processing. So, if the consumer crashes after processing the message but before committing the offset, it has to reread the message upon restart. Since, in this case, the consumer never committed the offset to the broker, the broker will redeliver the same message. Thus, duplicate message delivery could happen in such a scenario.

Exactly-once (each message is delivered once and only once): It is very hard to achieve this unless the consumer is working with a transactional system. Under this option, the consumer puts the message processing and the offset increment in one transaction. This will ensure that the offset increment will happen only if the whole transaction is complete. If the consumer crashes while processing, the transaction will be rolled back, and the offset will not be incremented. When the consumer restarts, it can reread the message as it failed to process it last time. This option leads to no data duplication and no data loss but can lead to decreased throughput.

Kafka : Performance

Here are a few reasons behind Kafka's performance and popularity:

Scalability: Two important features of Kafka contribute to its scalability.

A Kafka cluster can easily expand or shrink (brokers can be added or removed) while in operation and without an outage.
A Kafka topic can be expanded to contain more partitions. Because a partition cannot expand across multiple brokers, its capacity is bounded by broker disk space. Being able to increase the number of partitions and the number of brokers means there is no limit to how much data a single topic can store.

Fault-tolerance and reliability: Kafka is designed in such a way that a broker failure is detectable by Zookeeper and other brokers in the cluster. Because each topic can be replicated on multiple brokers, the cluster can recover from broker failures and continue to work without any disruption of service.

Throughput: By using consumer groups, consumers can be parallelized, so that multiple consumers can read from multiple partitions on a topic, allowing a very high message processing throughput.

Low Latency: 99.99% of the time, data is read from disk cache and RAM; very rarely, it hits the disk.