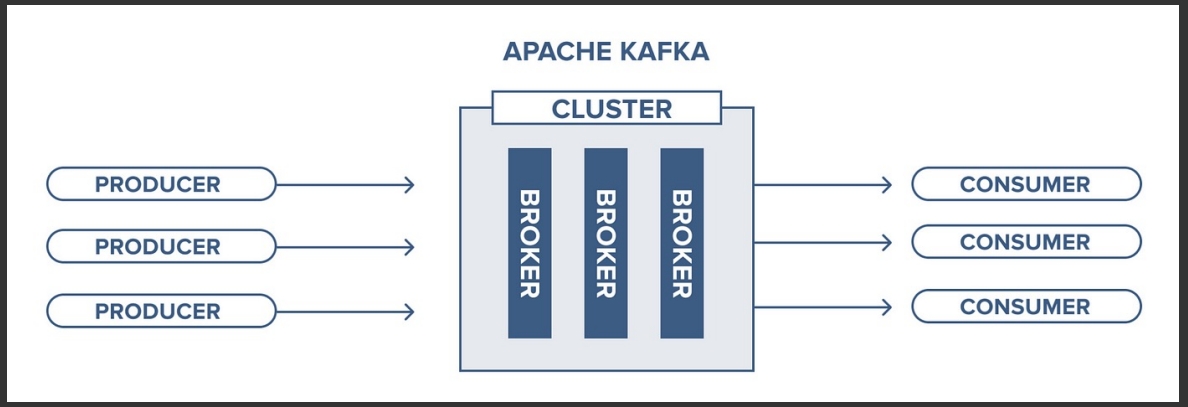
Kafka

Apache Kafka Architecture

The Kafka Architecture



Q. How does the Partitioning of Topics take place?

Q. How does Kafka ensure message ordering?

Q. Can the consumers read the same message again and again?

Q. What happens when new Consumers are added to the system?

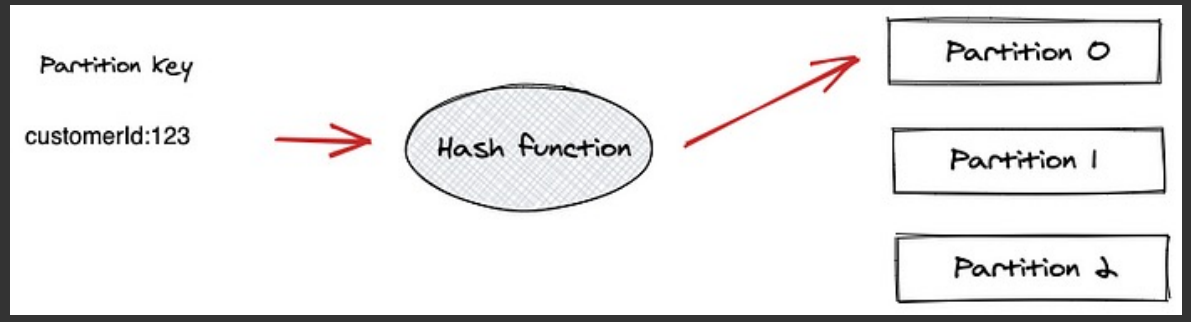
Q. What happens when some brokers go down?

Q. Can consumer read the messages after a few days they are produced?

Partitioning Logic

A producer can use a partition key to direct messages to a specific partition. A partition key can be any value that can be derived from the application context. A unique device ID or a user ID will make a good partition key. If a producer doesn't specify a partition key when producing a record, Kafka will use a round-robin partition assignment.

Messages with the same key are always written to the same partition, and Kafka ensures that consumers receive them in the order they were produced. However, if no partition key is used, the ordering of records can not be guaranteed within a given partition.



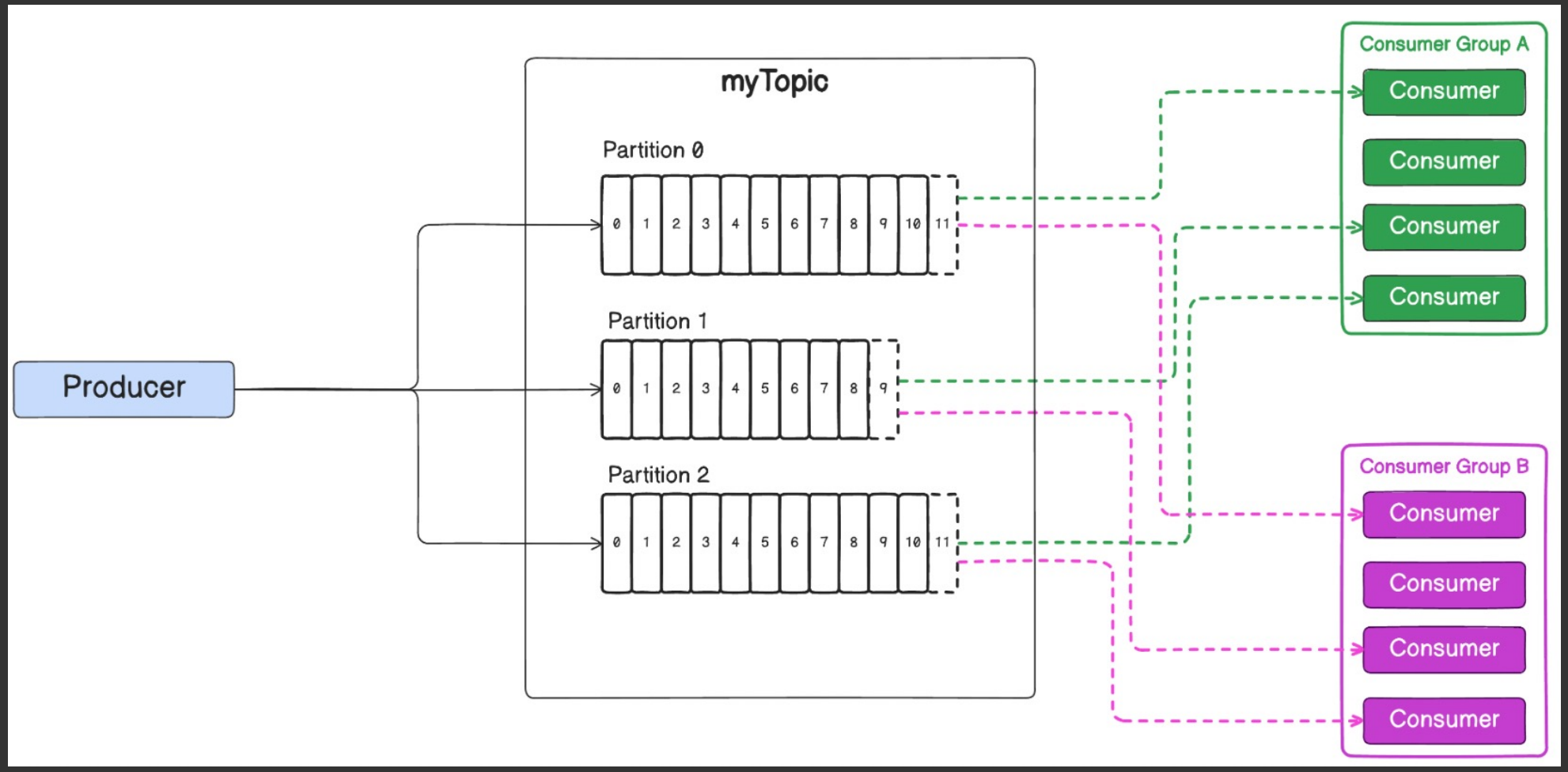
Reading records from Partitions

Unlike the other pub/sub implementations, Kafka doesn't push messages to consumers. Instead, consumers have to pull messages off Kafka topic partitions. A consumer connects to a partition in a broker, reads the messages in the order in which they were written.

By remembering the offset of the last consumed message for each partition, a consumer can join a partition at the point in time they choose and resume from there. That is particularly useful for a consumer to resume reading after recovering from a crash.

But this may create a problem where multiple consumers instances of the same type read the record of a Kafka topic. To avoid this, Kafka has a concept called Consumer Groups.

Consumer Groups



Consumer Groups

The consumer group concept ensures that a message is only ever read by a single consumer in the group.

When a consumer group consumes the partitions of a topic, Kafka makes sure that each partition is consumed by exactly one consumer in the group.

The maximum parallelism of a group will be equal to the number of partitions of that topic. The number of consumers don't govern the degree of parallelism of a topic. It's the number of partitions.

Rebalancing

Rebalancing is the re-assignment of partition ownership among consumers within a given consumer group such that every consumer in a consumer group is assigned one or more partitions. Rebalancing happens when:

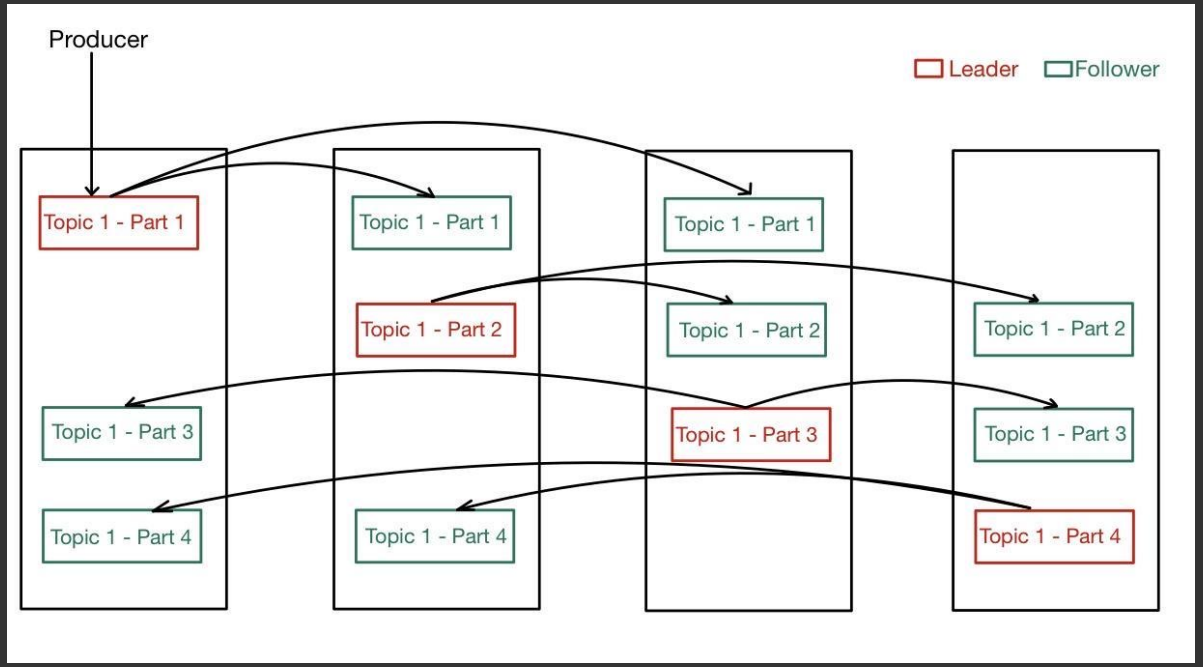
- A new consumer joins the consumer group
- An existing consumer goes down
- New partitions are added
- An existing consumer is considered dead by the Group coordinator

The first consumer that joins a consumer group is called the Group Leader of that consumer group

Partition Replication

Replication is making a copy of a partition available in another broker.

Replication enables Kafka to be fault tolerant. When a partition of the topic is available in multiple brokers then one of the partitions in a broker is elected as leader and rest of the replication of partition are followers.

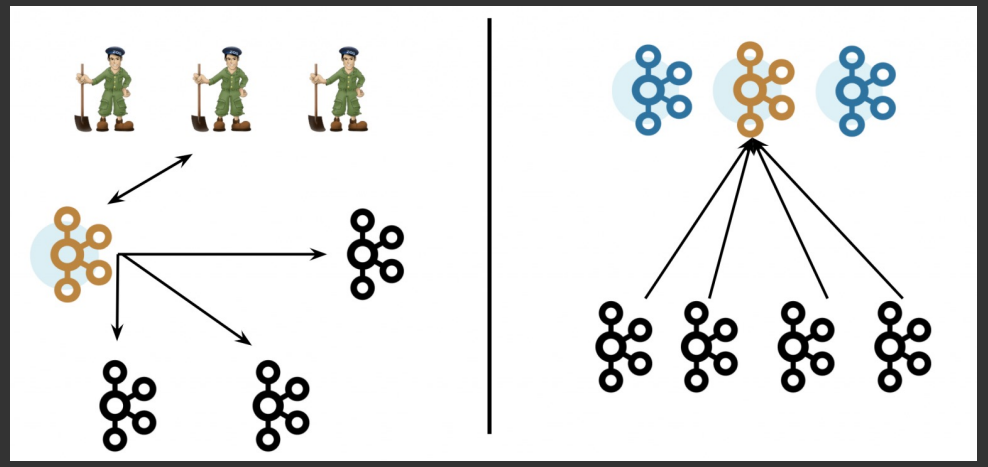


Kafka Brokers Management

Different brokers store partition replicas, with one of the brokers being the designated leader for each partition. A controlling mechanism that keeps track of the state of topics, partitions, and brokers is needed for such an architecture to work.

Kafka now uses the Raft consensus protocol to manage the metadata and the state of the Kafka brokers.

In KRaft mode, brokers coordinate directly with each other to elect a leader and replicate state across the cluster, eliminating the need for a separate system like Zookeeper.



Kafka Log Retention

Kafka's log retention controls how long messages are kept in a topic. Messages are stored on disk, and Kafka can retain them based on time (retention.ms) or size (retention.bytes), whichever comes first. Once the retention limit is reached, old messages are deleted in the background. Kafka can also be configured to retain messages indefinitely (retention.ms=-1), but this could lead to high storage costs if not managed carefully.

Acks and Retries

acks

The acks configuration determines the number of acknowledgments the producer requires from the broker before considering a request complete. It has three possible settings: 0, 1 and all (-1)

0: The producer does not wait for any acknowledgment from the broker, fastest, but offers no guarantee that the message has been received or written to the broker.

1: The producer will receive an acknowledgment as soon as the leader broker has received the message.

All: The producer will wait for acknowledgments from all in-sync replicas (ISRs) before considering the message sent.

Acks and Retries

retries

If a send request fails (for example, due to a temporary network issue or a broker being down), the producer will automatically retry sending the message up to the number specified by the retries configuration.
This is useful for ensuring that transient issues do not result in message loss.

When combined with `acks=all`, the producer can retry sending messages that have not been acknowledged by all replicas. This ensures that even if there are issues with one of the brokers, the message can still be sent to the other available brokers.