The two types of messaging patterns are:

* **Point to Point** Messaging System
* **Publish-Subscribe** Messaging System.

In **Point to Point** messaging system, senders send messages to a queue and receivers consume messages from the queue, once consumed message is lost, but in topics once a consumer consumed the message it will not be deleted until all the subscribers read the messages,

The message disappears from the queue, once the message is consumed by the receiver.

* But there is a restriction that a particular message can be consumed by a maximum of only one receiver.

In **Publish-Subscribe** messaging system, senders, also known as **publishers**, classifies messages and publish them to a **topic**. Receivers, or **subscribers**, can receive messages only on subscribing to that **topic**.

Unlike point to point, in Publish-Subscribe messaging system,

* A message on a topic is broadcast to all subscribed consumers.
* Consumers can subscribe to multiple topics to receive messages

Like how a person can subscribe to multiple news papers or a person can subscribe to multiple tv channels in real time

# Meanings

Offset- means how far he has read the message

## **Kafka features**

* **Reliability**: Kafka's distributed design, topic partitioning, and data replication over servers make it reliable.
* **Scalability**: Kafka system exists as a cluster of brokers. The number of brokers can grow over time when more data comes. Any failure of an individual broker in a cluster is handled by the system providing uninterrupted service.
* **Durability**: Disk-based data retention makes Kafka durable. Messages remain on the disk based on the retention rule configured on a per-topic basis. Even if a consumer falls backs due to any reason, the data continue to reside in the Broker till the retention period and is not lost
  + A configurable **retention period** can be set to retain all published records in Kafka irrespective of whether they were consumed or not.
  + For example, if retention period is set as three days, a record will be available for consumption for three days after it is published, meaning that consumer can come back within 3 days after that It will be discarded after the retention period.
* It writes the data to disk and replicates the data for fault-tolerance.
* Data / reliability guaranteed because – incase if acks is set to all then kafka will send acknowledgement to broker only when all in Sync replicas / followers are in sync and when all followers got the latest messages from kafka then only it will send acks to producer saying I have received the message
* **High-Performance**: All the above features make Kafka a High-Performance messaging system.

Producers

Producers can also send messages to a partition of their choice based on the incoming key they will send messages to the partitions of a topic

Producers write to a single leader so that each write is served by a separate broker which helps in load balancing

##### What is a Consumer ?

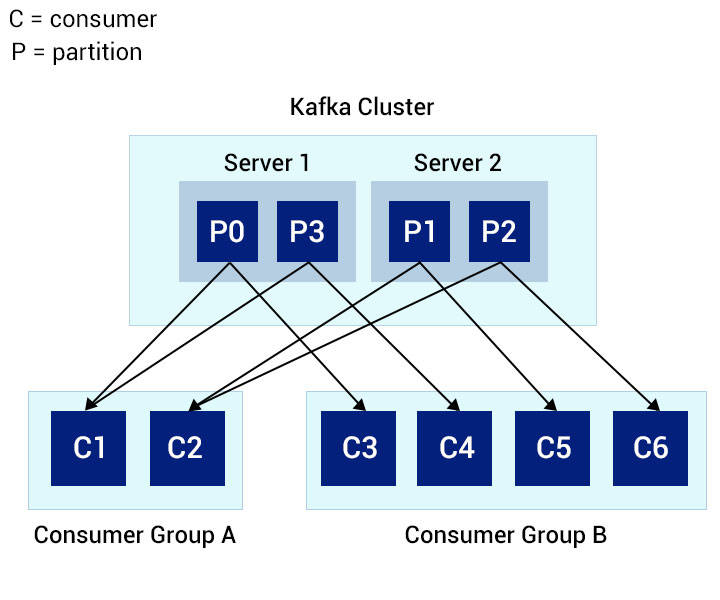
* Consumers are associated with ConsumerGroup using GroupId.
* Consumers having same GroupId belong to a ConsumerGroup.
* If all consumers belong to different consumer groups, then all the Consumer Groups will consume messages (This is a Publish-Subscribe model ).
* If all consumers belong to the same consumer group, then it is treated as single consumer, then the partitions of the same topic will be evenly distributed among consumers in the consumer group. (This is a Queuing Model ).means if the topic is having 6 partitions then 6 consumers of same topic will read from 6 different partitions of the same topic
* A consumer subscribes to a topic and consumes published messages by pulling data from the brokers.
* Consumers read from a single partition so that you can scale the throughput of message consumption similar to message production.
* If the number of consumers is more than the number of partitions then some consumers will remain idle as they have no partitions to read from.
* If the number of partitions is greater than the number of consumers, then each consumer will receive messages from multiple partitions.
* If the number of consumers is equal to the number of partitions, then each consumer reads messages in order from exactly one partition.

##### **Consumers & Consumer Groups**

Consumers can be organized into **consumer groups** for a given topic.

Each message published on a topic will be delivered to one consumer instance within each subscribed consumer group. These consumer instances may either be in separate processes or on separate machines.

* If all the consumer instances are within the same consumer group, then the records will be load balanced over the instances.
* If all the consumer instances are within different consumer groups, then each record will be broadcast to all the consumer processes.



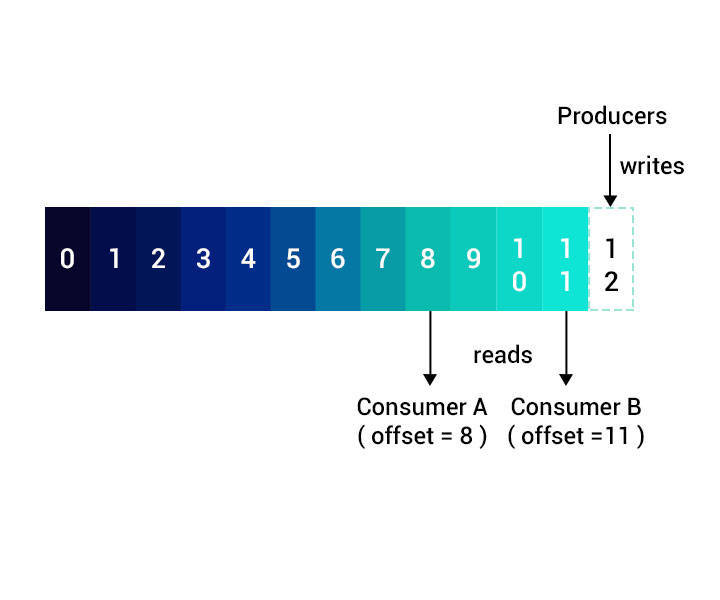
The image depicts a scenario with multiple partitions of a single topic.

Partitions 0 and 3 are kept in **server 1** and partitions 1 and 2 are kept in **server 2**.

There are two consumer groups - A and B. A is composed of two consumers, and B of four consumers.

* Consumer Group A consists of two consumers each reading two partitions and together reading all the four partitions of the topic.
* On the other hand, Consumer Group B has the same number of consumers and partitions, each reading exactly one partition from the topic.

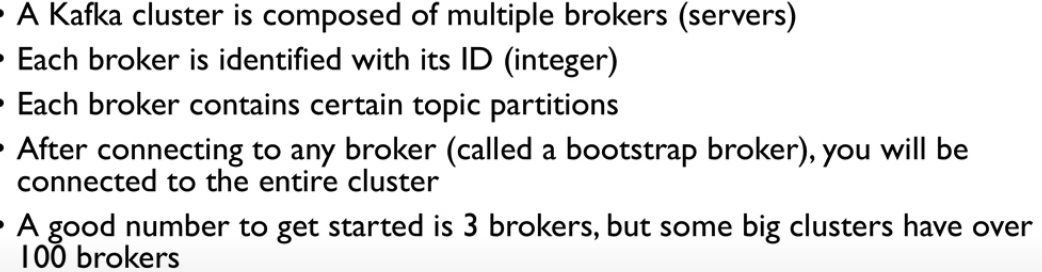
##### **Consumer Offset**



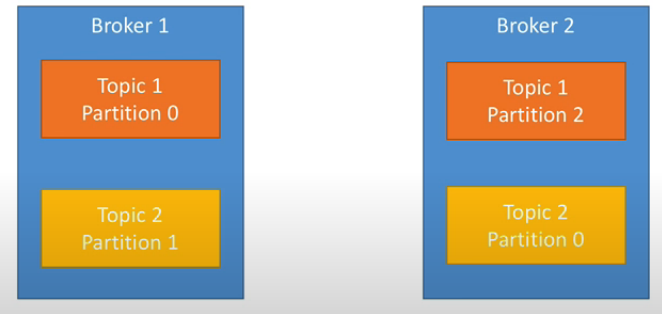
* The **Offset or position of a consumer** in the partition log is the only metadata retained for that consumer.
* The consumer controls the offset.
* When a consumer reads records, the offset advances linearly along the partition log.
* The consumer can read data from any position in the partition log - it can move back to an older offset to re-read older data or jump ahead to the latest record and start consuming from there.

### Kafka Broker

* Each broker may hold zero or more partitions of a topic. For example, if you have a topic with 24 partitions and a cluster with 3 Kafka brokers, each one will hold 8 partitions of the topic.
* In Kafka, brokers maintain their membership in the cluster through **Zookeeper**. Every broker in a cluster has a **unique ID** specified in their configuration file.
* The broker uses message partitioning strategy storing each partition in the corresponding broker in a cluster.
* Broker-1 have topic-chiru ,partition –A, Topic-Nags’s Partition-B



Each broker will not have all the partitions of a topic , 1st broker will have topic-1 p-1,topic-2 p-2,topic-3 p-3, and second broker will have topic-1 2nd partition… so the intent is all the partitions of a topic will not be in same broker



Therefore the data is distributed across 3 different brokers, so that if that broker goes down also it should not have any impact

## **Workflow**

**1.** Producers send messages to a topic at regular intervals.

**2.** Kafka broker stores all messages to the partitions configured for that topic,

Based on the message key, the kafka broker decides the partition number of that topic, such that the messages are equally divided among the partitions of the topic.

**3.** Consumer subscribes to the topic.

**4.** On subscription, Kafka will send the current offset of the topic to the consumer. It then saves a copy of the current offset in \_\_consumer\_offsets topic.

**5.** The consumer then requests polls Kafka for new messages at regular intervals.

**6.** Once received from the producer, Kafka forwards the message to the consumer.

**7.** The consumer receives the message and processes it.

**8.Consumer after reading successfully he should commit the offsets, because suddenly if the consumer went shutdown and came back online, first he must know**

**How far he has read right, so periodically he will commit the offsets, like we after reading the book, before closing the book we will keep some object so that if we come back on next day we can exactly know how many pages we have read.**

 Once processed, the consumer sends an acknowledgment to Kafka broker.

**9.** On receiving the acknowledgment, Kafka broker changes offset to the new value and updates it in \_\_consumer\_offsets topic.

**10.** The above flow goes on repeating until the consumer stops the request.

**11.** At any time, the consumer can rewind/skip to the desired offset and get subsequent messages.

1. Kafka broadcasts messages to all subscribing consumer groups, as with Publish-Subscribe.
2. **Thus, Kafka combines the strength of both these message models, enabling it to easily scale.**
3. Kafka assigns topic partitions to each consumer within the consumer group in such a way that each partition is consumed by only one consumer in the group. like 1st partition to the first consumer of the consumer group
4. This guarantees that the consumer of that consumer group is the sole reader of that partition, consuming the data in order.

## **Kafka Producer**

* In a Kafka system, the producer decides on the topic partition to be published. A message is sent to the leader replica of the brokers in cluster.
* The Producer sends these requests only to the **leader** replica. Otherwise, it will receive an error response saying "Not a Leader for Partition".

#### Produce Requests

* On receiving a Produce request, the leader replica will check if the respective user has write privilege to the topic partition.
* If the privilege is available, it will write the data and check 'ack' (acknowledgment configuration).
  + If it is 0, the leader replica will immediately acknowledge the Producer application.
  + If it is 1, then the leader replica acknowledges to the producer immediately after the request is written in the leader replica.
  + If it is set to all, the produce request is responded by the leader replica only after it receives acknowledgement from all the follower replicas. The request will be stored in a buffer called **purgatory** buffer till then.

## **Kafka consumer**

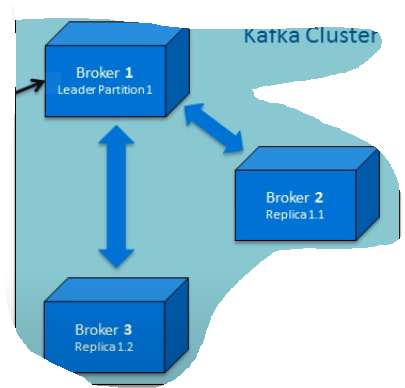
##### **Fetch /Consume Requests**

* **Fetch requests** are sent by the **consumer to Broker** to read messages from an **offset of a topic's leader** partition. It is also used among Brokers to **replicate partitions**.
* The client sends an **upper limit** (maximum amount of data the client can accept from the broker), and a **lower limit** (minimum amount of data needed for a data transfer to happen).whereas in case of message it could be fairly small but still it will be received by the consumer na, so here size is not considered na?
* The Fetch request should have an upper limit. Otherwise, the broker will send a large amount of data making the client run out of memory.
* Likewise, setting a minimum limit will ensure that broker sends the response only when there is sizable amount of data available to be sent to the client. Else, smaller amount of data processing happens resulting in resources being utilized inefficiently.
* As soon as a leader replica receives a Fetch request, it **validates the request** by checking if the offset specified in the **request exists for that particular** partition. Means u should tell the exact offset like nenu intha varaku messages chadivesanu, we should tell correctly
  + If it is valid, the leader replica will **read messages from that offset** till the specified upper limit and return the messages to the client.
  + The broker will **respond with an error** message if it is an old one or if it does not exist.
* To avoid any *inconsistency*, the leader replica always sends messages that are replicated to all its followers.

## **Leader concept**

* **Follower** are the replicas other than the leader that replicate messages from the leader and remain updated with the recent message. In the event of a leader crash, one follower will be promoted as **leader**.
* One replica of a topic partition acts as a **leader**. All producer and consumer requests go through the leader for that topic partition.

## **How Replication Works?**



* **Fetch** requests are constantly sent by the follower replicas to the leader to consume recent messages.
* That Fetch request contains an **offset of the message** the replica wants to receive. The leader responds to the request with a message at that offset.

If offset is not mentioned then how server knows how many messages have been read by consumer? , so of it is mentioned then only he can read messages from there in that topic

* Message requests to the leader are in **sequential order**. The offset requested by the **replica**, helps the leader to **identify the last message** received by the **follower replica**.
* A **replica** is considered **out of sync** when it fails to update with recent messages **within 10 seconds** and loses credibility to become a leader in the event of some failures with the current leader in the cluster.

Kafka supports **two forms** of replication :

* Synchronous Replication
* Asynchronous Replication

### **Synchronous replication**

1. The Producer identifies the lead replica from the Zookeeper and publishes a message.
2. The published message is written to the log of the lead replica, and all of its followers in **ISR (In-sync Replica)** start to pull the messages. Please note that every lead replica maintains In-Sync Replicas (ISR) in the cluster. These are the minimum set of replicas that ought to be in sync with the lead replica whenever there is a change in the lead replica.
3. Once the message is written to its respective logs, each of the follower replicas send an acknowledgment to the lead replica.
4. Once all expected acknowledgments are received, and the replication is complete, the lead replica sends an acknowledgment to the Producer.
5. The consumer pulls the message from the lead replica.

### Asynchronous Replication

In an **asynchronous replication**, once the published message is written to the lead replica log, it **acknowledges** the message producer **without waiting** for an acknowledgment from follower replicas.

In such replication, in the event of broker failure, there is no guarantee that all the follower replicas would have committed the published message.

### Leader Replication

There are chances that leader can go offline

### Handling Broker Failures

1. As soon as a **follower** in the ISR (in-sync replica) fails, the leader removes it from the ISR and continues writing to other followers in the ISR.
2. Once it comes back, it will immediately truncate its log to the offset position of the last message committed to its log.
3. Then it starts reading the messages from that point from the leader.
4. Once it is fully synced with the leader, it is added back to the current ISR by the leader.

### **Handling Leader Failures**

* When a leader fails while writing a message to its log(similar to giving or helping team members to real time) or acknowledging the producer (like a TL not responding to Manager ), a new leader is elected.
* To elect the new leader, all the follower ISRs register themselves with the Zookeeper.
* The first one to register becomes the new leader replica, and the offset of its log end becomes the offset of the last committed message.
* All other replicas become the follower of the new leader.
* Each replica registers a listener in the Zookeeper so that it gets notified of any leader change.
* Once there is a leader change, the replica truncates to the offset of the last committed message and starts to catch up with the new leader.
* The new leader waits until all live replicas get in sync, write the new ISR to the Zookeeper and opens itself for any read/write from producer or consumer.

### **Request Header**

All the request headers should have:

* **Request Type** - This is the API Key.
* **Request Version** - enable brokers to handle clients independent of their versions and respond accordingly.
* **Correlation ID** - This is a number which can **uniquely identify requests**. It is also present in the response and error logs.
* **Client ID** - This **identifies the application** from which a request is sent.

##### **Request Types Handled by Broker**

There are **three** types of requests managed by the broker. They are:

* Produce
* Metadata
* Fetch