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.

## How to improve kafka from coding point of view

* + The ack configuration **ensures the durability** of the message sent. It can have three values viz. 0, 1, and all.
  + Produce requests -- 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.
* Fetch requests -- 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).
* Don’t retry all the times, u don’t need any specific API , just configure few params it will automatically retry when a producer send a message to a broker

Then when leader is down , then a leader will be elected among the existing topic replications, during election time the messages will not be delivered so only at that time you have to retry that to it will only do with help of some params

* + Try setting these params

**buffer.memory, compression.type**, retries (when leader is down msg will not be delivered during the leader election among remaining 2 servers so we can configure retries)

**if the topic is having 4 partitions, then make sure to consume data from 4 partitions we should have 4 consumers in that same consumer group, else if we have only 1 consumer , then load will be there on that single consumer as it should read from 4 partitions**

**how do you identify that message is a duplicate—based on the correlation id, every message will have different correlation id and timestamp.**

Analogies

* + If Number of partitions =4, then have 4 consumers in that consumer group is like having more work in project then have more resources

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 ?

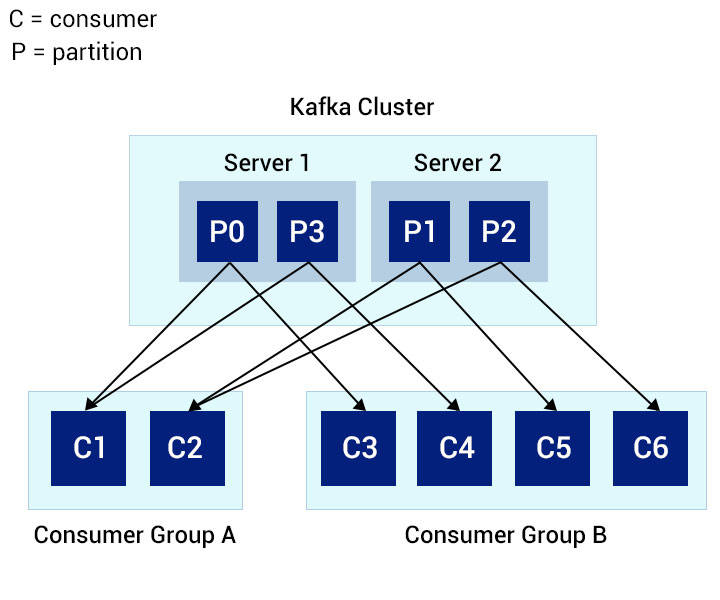
* Consumers are associated with ConsumerGroup using GroupId.
* Consumers having same GroupId belong to a ConsumerGroup.
* If all consumers belong to different consumer groups, then all the Consumer Groups will consume messages (This is a Publish-Subscribe model ).
* If all consumers belong to the same 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.



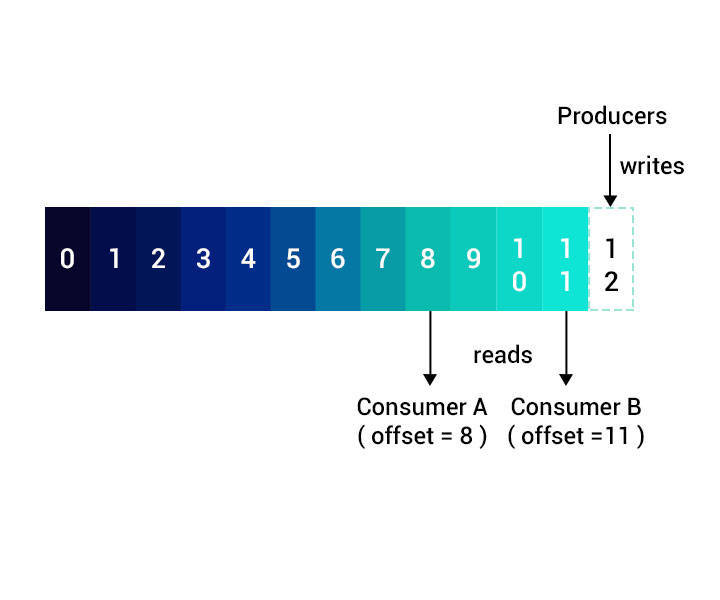
The image depicts a scenario with multiple partitions of a single topic.

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.

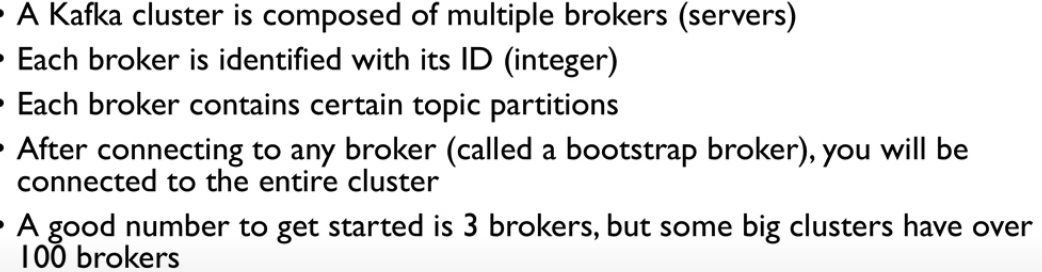
##### **Consumer Offset**



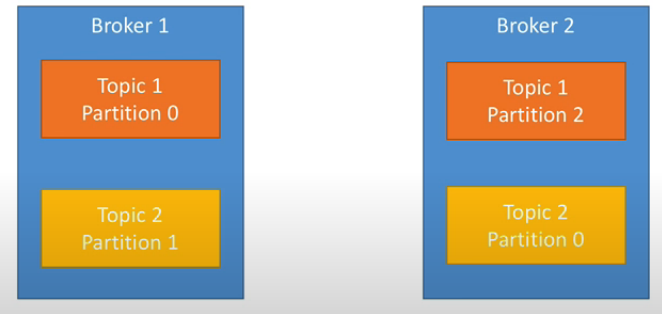
* The **Offset or position of a consumer** in the partition log is the only metadata retained for that consumer.
* The consumer controls the offset.
* When a consumer reads records, the offset advances linearly along the partition log.
* The consumer can read data from any position in the partition log - 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

Kafka broker allows the consumer to read data from any available offset.

* To help brokers to locate the given partition quickly, Kafka maintains an **index** for each partition.
* An index is a mapping of **offsets to segment files** and its **position within the file**. may be something like the it index will have consumer name

|  |  |
| --- | --- |
| Consumer name | Consumer offset along with segment (file name) and how much he read in that file |

* When **messages are deleted**, corresponding offset entries will be deleted from the index.
* Indexes are split into segments.
* When an index becomes **corrupted**, it will get **automatically regenerated** by re-reading the messages and recording the offsets and corresponding locations.

#### Broker threads flow

1. The broker runs an **acceptor thread** on each port it is listening to.
2. The acceptor thread creates a client connection, and hands over the connection to the **processor thread**, also called **network thread**.
3. The processor thread takes requests from client connections and places them in a **Request Queue**.
4. From the Request Queue, the **I/O thread** picks up the request, processes them and places them in a **Response Queue**.
5. From the Response Queue, the process thread picks up the response and sends back to clients.

## **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.

## **Topics**

In topics 🡪 we have partitions and in each partition 🡪 we have segments.

Lets say , if that topic is having 4 partitions , then No two partitions of a topic should remain in the same broker. Means like all married son’s will not be in same home , similarly all partitions of a topic will not be in same broker they will be distributed among all brokers

* **Retention Period of Topic**: Messages that are older than the retention period will be deleted.
* **Compaction**: This is done only on messages with **key-value** data inside the **inactive partition** segments. Compaction allows storing only the most recent value of each key in a message.

## **Partitions**

* Each partition is split into **segments** of a particular size, say 1 GB or configured to be retained for a proper amount of time, say one week. Each partition segment is stored in a separate file.
* A message from the Producer is written to multiple partitions. we can decide the partition number where the message should go. Its like 1 apartment (1 topic) have many number of flats(topics) whereas each topic still have segments(like 1 flat have many rooms)
* Each partition has numerous segments. When any of the above limits (size/retention period) exceeds while writing data to a partition segment, then that segment is closed, and a new one is started.
* A segment to which Broker is currently writing is called an **active segment**.
* An active segment will never be deleted until any of the segment's limits are exceeded.
* A broker may open multiple file handles for every segment for all of its partitions, irrespective of active or inactive.

##### **Specifying Topic Partition**

* Producers create messages by creating a **ProducerRecord** that includes the **topic to which the data should be sent**, and the **corresponding data**.

producer.send(new ProducerRecord<>(topic,messageKey,messageValue)

* ProducerRecord can have a **Key** and **Partition**.
* After sending the ProducerRecord, the producer **serializes the key** and the value objects to ByteArray.
* The serialized data is then sent to a **partitioner**.
* Two options are there 1) Specifying a partition in the **ProducerRecord** will send the data to that partition by the partitioner. 2) If u don’t specify the partition number then it will calculate a partition based on the key specified in the **ProducerRecord**.
* Once a partition is chosen for the data, the producer will know which partition of the topic the data should go.
* The partitioner then adds the record to a batch of records that will be sent to a specific partition of a topic.

##### **Segment File**

Each partition segment is stored in a **single data file**. The file contains **messages** and their **offsets**.

Each message consists of its:

* key
* value
* offset
* message size
* **checksum code** - helps to detect message corruption.
* **magic byte** - indicates the version of the message format.
* **compression codec** - indicates message compression format (snappy, GZip or LZ4).
* **timestamp** - denotes the time message was sent from the producer or received by the consumer.

## **Offsets**

Generally offsets are committed to the broker, the consumer should commit to the broker , because the consumer may go down , but the broker will not go down,

## **Cleaning Kafka/Compaction**

Compaction is a process of removing the records / messages based on the key .

Sometimes message will come with the same key, then if multiple messages came with same key deletion of previous messages of same key is called Key compaction, this procedure will retain the messages with the newer key

When compaction is enabled, the following things happen:

* The broker starts a **compaction manager** thread and multiple **compaction threads**. Compaction threads are responsible for the compaction task.
* The compaction thread starts the compaction task by choosing the partition, with the **highest ratio of dirty messages** to the total size of the partition and starts cleaning.
* The compaction thread reads the dirty portion of the partition and creates an **in-memory heap** of map entries. A map entry comprises of **hash of each message** (a key-value pair) of 16 bytes **and the offset of the previous message** that had the same key which is 8 bytes. Thus, each map entry is **24 bytes** in size.
* After building the offset map for the dirty segments of the partition log, the compaction thread starts **reading the clean segments**, starting from the oldest. It checks their content against the offset map.
* It checks if the key of the message is present in the offset map.
  + If the key is present, the message in the segment is ignored, since there will be a newer value for the key later in the partition.
  + If the key is not present, the corresponding message will be copied to a replacement segment, since it is the newest message for that key.
* Once the replacement segment is built with all latest messages for their keys, the original segment is replaced.
* Finally, only clean segments exist with one message per key – each keys having the latest value.

##### **Deletion of Messages from Segments**

* To **delete the message** of a key completely from the system, the **application needs to produce** a message with that **key with a NULL value**.
* When the compaction thread finds the key with the NULL value, regular compaction is done replacing the segment with the key with the NULL value.
* This message with the key with the NULL value is also called a **tombstone message** and is retained for a configured time.
* When a consumer sees a tombstone message, it knows that the value is deleted.
* After the configured time, the compaction thread will delete the tombstone message, and the key will be completely removed from the partition log.

## **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.
* **Produce requests** are sent from the producer application and contain messages that need to be written to the Broker.
* The Producer sends these requests only to the **leader** replica. Otherwise, it will receive an error response saying "Not a Leader for Partition".
* Messages sent from the producer are **retained in the Kafka broker** till the **time configured for each topic** of the message, referred as **retention period**. If the retention period is 20 days, the consumer can come back online within these many number of days then till then it will it be holded in that
* The administrator configures retention period for each topic regarding
* the amount of **time the message should be kept** before deleting them.
* the amount of **data that can be stored** before the older messages are purged.

#### 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).

##### **Configuring ACKS**

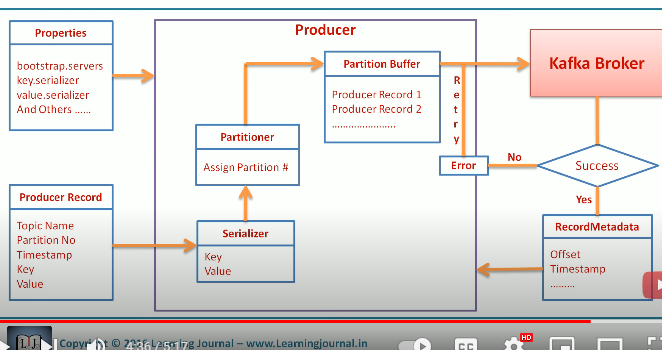
**i**. acks=0:

* This setting can be used for achieving high throughput and is the fastest one.
* When acks is set to 0, the Producer will not wait for an acknowledgment from any of the brokers to assume the message is sent successfully.
* **This setting provides the least durability** as there could be message loss - sometimes the message is not sent to the broker, and the Producer will not know about it. If u keep acks=0 , since the producer never waits for the acknowledge from the broker , even if the message fails it assumes message is not lost and it wont even retry. Generally the inbuilt retry works on this configurations

**ii**.acks=1:

* With this setting, the Producer will receive a success response **from the broker** as soon as the message is written to the leader replica.
* If the leader replica crashes in between and the message could not be written to the leader, the producer will receive an error response, and it will resend the message avoiding loss of data.
* There can still be a message loss if the leader crashes and a replica gets elected as a new leader without this message.
  + With acks = 1, there will be a tradeoff between throughput and latency depending on whether the send() method sends data **synchronously** or **asynchronously**.
  + If the data is sent synchronously, waiting for a reply from the server using the FutureObject get() method, there will be an increase in latency.
  + If the data is sent asynchronously with a **callback**, the throughput depends on the number of messages sent from the Producer before receiving a reply from the server.
  + **iii**. **acks=all**:
  + The Producer will receive a success response from the broker, once all in-sync replicas received the message.
  + This setting ensures the safest message publish, as it ensures the data had reached at more than one replicas in the cluster.
  + But it has the highest latency as the broker is waiting for the message to be written to more than one replicas.

Total flow

* Producers create messages by creating a **ProducerRecord** that includes the **topic to which the data should be sent**, and the **corresponding data**.
* ProducerRecord can have a **Key** and **Partition number**.
* After sending the ProducerRecord, the producer **serializes the key** and the value objects to ByteArray (so make sure both are serializable).
* The serialized data is then sent to a **partitioner**.
  + 

|  |
| --- |
| Producer Record |
| Topic- mandatory |
| Partition- optional-if partition is not specified then it will decide partition based on the key |
| Key-optional – if u don’t specify the key then msgs are evenly distributed across all partitions |
| Value-mandatory |

### **Decide the partition**

Always remember, all the partitions of a topic will not be in same broker, consider if topic-A has 4 partitions then

Topic –a ‘s partition 0 will be in broker -1 , p1-will be in broker -2… means partitions will be splitted across brokers

If u specify the partition num, then that message will go to that same partition. If u don’t specify the partition number, then partition will be decided based on the hash value of that key, then messages will be sent

1. If u define **same** key then all messages, then all messages will go to the same partition, because partitioner will decide the partition number based on the key
2. If u **don’t specify the key** then the default partitioner will send all the messages to all partitions evenly in a topic, it will uses round robin algorithm

Once the partition is decided, messages will not be send immediately, all those messages will be sent to partition buffer. They will be sent in batch to the broker, once messages are successfully received by the broker then the broker will send the Record Metadata object as an acknowledgement back to the broker

if so many producers are there they might be sending data to diff partitions of the same topic

### **Writing Message and acknowledging**

* Using a separate thread, the Producer will send each record from the batch of records to the appropriate Kafka Broker.
* If the record is completely written to the broker, it will respond back to the Producer with a **RecordMetadata** object. This object contains the topic, partition, and the offset of the record within the partition.
* If the broker fails to write to the partition log, it will send back an error.
* When the producer receives the error, it will try resending the record.

### Creating Producer Object

The following properties are mandatory for a Kafka Producer.

* **bootstrap.servers**: This is the list of **host:port** pairs of brokers needed for Producer to establish an initial connection to the Kafka cluster. The list must have at least two pairs so that the producer can connect to the cluster during failures.

Since both key and value objects must be sent over network

Both the objects must be serializable by the serializer

**key.serializer**: The producer interface accepts parameterized types and objects to be sent as key-value pairs. However, brokers accept keys and values of messages as byte arrays. **key.serializer** is again the name of the class that **serializes the keys** of the records sent from Producer to byte arrays. It is the class that implements org.apache.kafka.common.serialization.Serializer interface.

* **value.serializer**: This is the name of the class that serializes the values of the records produced from Kafka.It is the class that implements org.apache.kafka.common.serialization.Serializer interface.
* Properties kafkaProps = new Properties();
* kafkaProps.put("bootstrap.servers","localhost:9092,localhost:9093");
* means even if 1 server was down it will connect to another server
* kafkaProps.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
* kafkaProps.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
* KafkaProducer<String, String> producer = new KafkaProducer<String, String>(kafkaProps);

### **Creating Producer Record Object**

Kafka producer accepts **ProducerRecord** object which is created using **ProducerRecord constructor**.

It needs two parameters, the **topic name** to which we want to send the record and the **key and value** of the record we are sending to Kafka Broker (which are of string type).

The following code snippet displays the creation of a ProducerRecord object in Java.

ProducerRecord<String, String> record =

new ProducerRecord<>("CustomerCountry", "Precision Products","France");

In the above code snippet, **CustomerCountry** is the **topic** name, **Precision Products** is the message **key** and **France** is the message **value**.

### **Sending Record to Kafka Broker**

The producer object's send method is used to send the ProducerRecord object, to the Kafka broker.

The below code snippet is used to send the ProducerRecord to Kafka broker.

producer.send(record);

**Synchronous Send**: Here, the send() method returns a future object, and we use the get() method of that object to wait and see if the send() was successful.

**Asynchronous send**: Here, the send() method is called with a callback function which will be triggered on getting a response from the Kafka broker.

### Configuring other params

* **buffer.memory**: This is used to set the total amount of memory the Producer can use, to buffer messages waiting to be sent to Broker. The producer will block sends for max.block.ms time and throw an exception, if the messages are sent faster than the brokers could receive them.
* **compression.type**: This is used to set the compression format for the messages sent from the Producer. The default value is **None** - meaning no compression. For compressing messages before sending to brokers, gzip, snappy, or lz4 algorithm can be specified.
* **retries**: The message from the Producer sometimes will not be written to broker due to transient errors (no leader for a partition). The value of **retries** decides the number of times the producer resends data to the broker on receiving a transient error. It is set to a value **greater than zero**.

### Sample kafka producer program

public static void produceMessages(){

String topic = "MyTopic"; //Use this topic.

Properties kafkaProps = new Properties();

kafkaProps.put("bootstrap.servers","localhost:9092");

kafkaProps.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");

kafkaProps.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

kafkaProps.put("acks","all");

//If the request fails, the producer can automatically retry,

kafkaProps.put("retries", 0);

KafkaProducer<String, String> producer = new KafkaProducer<String, String>(kafkaProps);

ProducerRecord<String, String> record = new ProducerRecord<>(topic,"Hello");

//in above topic is topic name and hello is the value

producer.send(record);

producer.close();

## **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.

To consume messages from topics in the broker,

* The application will create a consumer object, subscribe to the required topic.
* Consume messages from the topic, validate them and write them to another application or data store.

##### **Consuming Messages**

* To consume message from Kafka broker, the **consumer subscribes** to the required topic on the broker.
* While subscribing, the consumer **connects to any of the live nodes** in the cluster to get metadata about leaders of partitions.
* Once getting metadata about leader partition, the consumer **sends a fetch request** to the broker with the **leader partition giving the offset** of the message.
* Each topic is **divided into multiple** partitions, and each consumer in a consumer group will be assigned to a different set of partitions of the topic.
* Thus a whole consumer group reads the topic.
* Once a partition is consumed by the consumer, it changes the offset to the next partition to be read.it should oftenly commit the offsets , same like when we read a book daily night we keep some object in that last paper we read so that if we come back on next day we can know till how many pages we have read, hence consumer should commit the offsets very frequently
* A consumer can **change to an older offset, so that as offset changed to previous position indicates he want to read those messages again** thus re-consuming the message.

### Consumer Group:

In real time, all consumers in a consumer group will be treated as same consumer

Each consumer will read from that one partition,

Consider the situation when Producer is sending data to the broker at a faster rate than your application could read. Your application will fall back **failing to keep up** with the rate of messages coming from Producer.

* To scale the consumption of messages from the topic, more consumers will be added to the same consumer group, so that the reading will be balanced between the consumers.
* Multiple consumers subscribed to a topic can join a group called **Consumer Group** so that each consumer in the group will have the **same id** which will be that of the **consumer group id**.

By default, **a consumer** will be part of a consumer group.

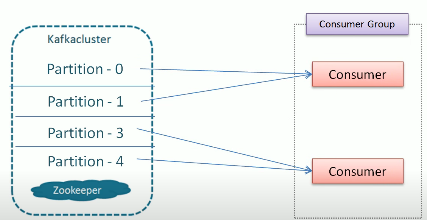
In case of 4 partitions and 2 consumers in a consumer group then those 2 consumers will read from 4 partitions, means there is a load on those 2 consumers , because each consumer will be reading from 2 partitions

Partitions vs consumers

If your application has **only one consumer instance** consuming and processing partitions, it will be **unable to keep up** with the rate of messages flowing into topics.

To **scale message consumption**, more consumers are added so that the topic partitions are split between multiple consumers. Here any of the four situations may occur:

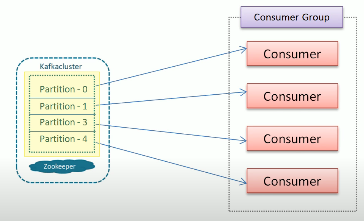
* 1. When multiple consumers are subscribed to a topic, and all of them belong to the same consumer group. Each consumer in the consumer group will get different set of partitions of the topic and the consumer group as a whole reads the topic. like 1st consumer in a consumer group will read from partition -0, second consumer will read from partition-1, 3rd consumer from partition -2



**2**. As per above when the topic has multiple partitions, and only 2 consumers subscribed to the topic, and the consumer group has only 2 consumers. Then each consumer have to read from 2 topics parallelly which causes load on consumers.so best is if we have 4 consumers for 4 partitions , then there will be no load on those consumers

**3**. When the **number of consumers in the consumer group** grows to the number of topic partitions, then each consumer in the consumer group gets exactly one partition of the topic.

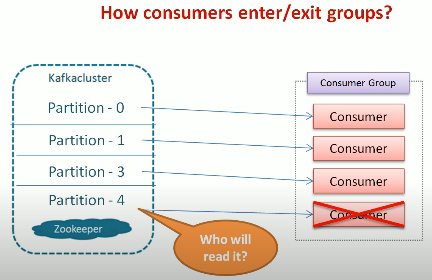
**4**. When the **number of consumers in the consumer group grows to a number greater than the number of topic partition**s, some of the consumers in the consumer group **remain idle** and get no message partitions. Like having 5 consumers for 4 partitions, 1 consumer will be idle. Analogy\_ its like more people in project than the work



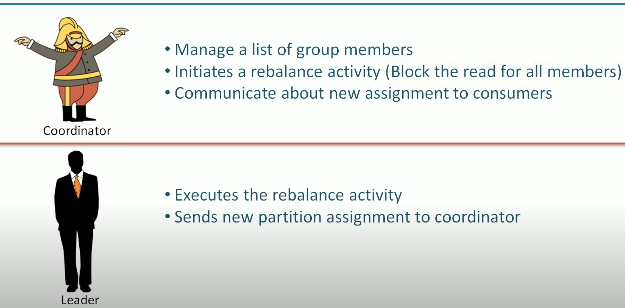
Consumer group rebalancing/

Rebalance :- means assigning partitions to individual consumers

* + Leader is responsible for rebalancing activity



In the above case , when an existing consumer dies or when any new consumer add to the consumer group , then the group will rebalance means suddenly if the consumer died who will take care of his work of reading from any partition same like if any person in a team has kept resignation then who will take care of his work, so at that time group/ team should re balance so that existing consumers/team members should bear that load untill they find a replacement or until a new consumer joins the group , meanwhile the existing consumer should understand till where that died consumer have read that message (means leader should know the offset), so that they can give that offset number to the new/existing consumer and that new consumer will read from that position

* + **During rebalancing , messages will not be read by any consumer from any partition** , means all consumers in a consumer group will remain silent and they wont read any data from any partition
  + 

Consumer groups:- used to read and process data in parallel.

Partitions:- 1 consumer may read from 2 partitions, but 2 consumers will not read from same partition

Group coordinator:- maintains a list of active consumers

Rebalance :- means assigning partitions to individual consumers. Initiates when the list of consumers are modified, like when a consumer exited or when a new consumer joins a group

Group leader – executes a rebalance activity

##### **Consumer Rebalance – Introduction**

Transferring the partitions of one consumer, to another consumer is called a **rebalance**. Like when a person kept resignation some one of the team members should take care of his work right, it is called balancing the team, if that work is not handled by other the project will imbalance

A consumer in a consumer group reads different sets of partitions of a topic.

When a **new consumer is added** (like when a new person joins a team) to the consumer group, it starts reading messages from partitions read early by another previous consumer (like it will take the existing members work).

When a consumer in a consumer group crashes (like when a existing member left the team), it **leaves the consumer group** and its corresponding partitions will be consumed by any other remaining consumers by knowing the offset.

Partition re-assignment also happens when a topic currently being consumed by a consumer group is modified.

Rebalance makes the consumer group highly available and scalable.

During a consumer rebalance, consumers cannot consume messages or change their current state until rebalance is complete.

Consumers ensure their membership in a consumer group by sending **heartbeats** to a broker that is declared as the **group coordinator** for the consumer group and will be different for each consumer group.

**Heartbeats are sent** when the **consumer is retrieving messages** from its topic partitions. Heart beats are like how we fill time sheets in so that manager is aware that we are in team.

As long as a consumer sends **heartbeats at regular intervals** to the broker, it is considered to be **live, and processing** messages from the partitions it is assigned with.

If the consumer stops sending heartbeats for a long time, the group coordinator will consider the consumer is **dead** and triggers a consumer **rebalance**.

When the consumer is crashed, it makes the group coordinator **wait for some time** without heartbeats, and decide that it is dead and a rebalance should be triggered.

When the consumer is completely closed, it notifies the coordinator that it is going to leave the group, and the group coordinator will immediately trigger a rebalance.

##### **Assigning Partitions to Consumers**

A consumer that needs to join a group sends **JoinGroup** request to the group coordinator like when we join the project .

The first consumer to join the group becomes the **leader** (Team lead) of consumers in the consumer group. The leader gets a list of all consumers in the group from the **group coordinator**.

The leader consumer assigns unique partitions to each consumer in the consumer group. The leader uses **PartitionAssigner** implementation for this purpose.

Like if there are too many user stories (or too much of work) then as like Team lead assigns 2 /3 stories for each developer, similarly if there are more partitions in a topic and less consumers in a consumer group , each consumer will get 2-3 partitions like how each developer gets 2-3 stories

Once deciding the partition assignment, the consumer leader sends a list of partition assignments to group coordinator.

Group coordinator sends the assignment information to corresponding consumers.

The consumer leader will have the complete list of consumers in the group and the partitions assigned to them.

## **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.

## **Kafka Retry**

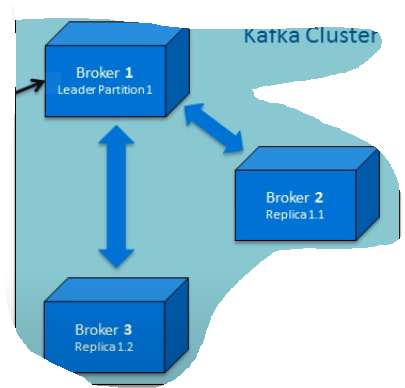
Some errors are recoverable with retry

Suppose if the leader /leader node of the partition is down , if we retry after few millis then meanwhile leader would have got elected in this scenario retry is worth.

So only in case of recoverable errors producers will retry after some milliseconds, we just need to configure them using Producer config parameters

If error is not recoverable then producer will not retry like retryable exception

## **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

## Kafka properties

Spring:

kafka:

consumer:

bootstrap-servers: ent-kafka-dev.wellsfargo.com:49092 properties:

bootstrap-servers: ent-kafka-dev.wellsfargo.com: 49092

spring.json.trusted.packages: "com.wellsfargo.common.kafka"

auto-offset-reset: latest

key-deserializer: org.apache.kafka.common.serialization. StringDeserializer

value-deserializer: org.springframework.kafka.support.serializer.JsonDeserializer

producer:

bootstrap-servers: ent-kafka-dev.wellsfargo.com: 49092

key-serializer: org.apache.kafka.common.serialization. StringSerializer

value-serializer: org.springframework.kafka.support.serializer.JsonSerializer

properties:

acks: all //means producer will get acknowledgement for the message only when all in sync Replicas received the messages

retries: 3

retry.backoff.ms: 200000 // in case if broker is rebalancing then message may not be accepted by the broker, in that case if we retry after the specified milliseconds