Apache Kafka vs RabbitMQ

In the previous section we discussed on how to build microservices that can communicate asynchronously with the help of RabbitMQ message broker.

Now, inside this section we are going to focus on how to build the **asynchronous communication** or **event driven communication** between microservices with the help of Apache Kafka.

So, before I try to explain what is Apache Kafka in detail, first let me give a quick introduction about

what are the differences between RabbitMQ and Apache Kafka, under which scenarios we need to choose Apache Kafka or RabbitMQ.

Like we discussed previously, both Kafka and RabbitMQ are popular messaging

systems, but they have some fundamental differences in terms of how they are designed architecture and use cases.

If you ask me what are these differences?

The very first difference is in terms of design.

**Kafka is a distributed event streaming platform, whereas RabbitMQ is a message broker.**

This means Kafka is designed to handle large volumes of data, while RabbitMQ is designed to handle

smaller volumes of data with more complex routing requirements.

So, if you are looking for complex routing requirements inside your application, then you need to go

with the RabbitMQ.

Whereas if you are trying to handle a large amount of data, then Apache Kafka is the best product for

you these days.

*RabbitMQ is also trying to become an event streaming platform in the recent versions but there is*

*a long way for them to achieve the capabilities provided by the Apache Kafka.*

The next important difference is around the data retention, how the data is going to be stored inside

the Apache Kafka and RabbitMQ.

So, whenever you send the messages or event details to these products, Kafka is going to store all

the data on the disk, whereas RabbitMQ is going to store the data inside memory.

Since Kafka is using disk capacity to store the data, it can store or retain any amount of data for

longer periods of time, while RabbitMQ since it is trying to store the data inside the memory, it is more suitable for the applications that require very low latency.

After the data retention there is also a difference in terms of performance.

Kafka is generally considered as faster than RabbitMQ, especially whenever you are trying to deal with large volumes of data.

However, RabbitMQ can have better performance in the scenarios where your application needs complex routing requirements.

After the performance, moving on to the scalability, Kafka is highly scalable, whereas RabbitMQ is more limited in its scalability.

Why Kafka is highly scalable.

Because whenever we are using Kafka, we can horizontally add any number of Kafka brokers to the Kafka cluster.

That means there is no limitations to you on how much data you want to process or how much Kafka brokers that you want to set up inside the cluster.

Whereas when you are dealing with RabbitMQ, definitely there will be some limitations on how much you want to scale RabbitMQ.

So overall, both products are good.

They support event streaming or asynchronous communication between your microservices.

So, whenever you are in the dilemma which product to use, it completely depends upon your requirements.

If you are looking for a high-performance messaging system that can handle any large volume of data, then Kafka is a good choice.

Whereas if you are not going to handle large volume of data and if you are looking for a message system with a complex routing requirement, then RabbitMQ is a good choice.

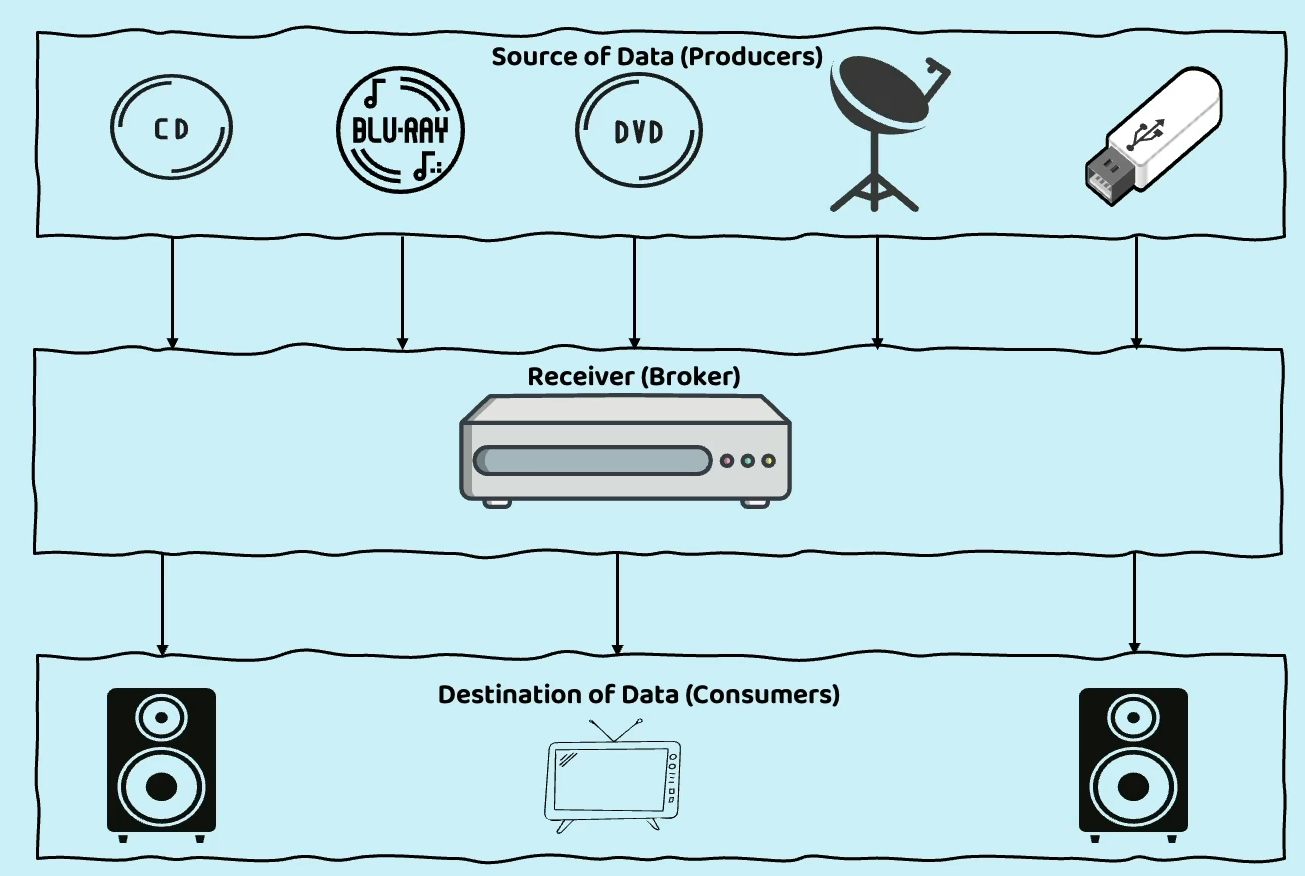
Moreover, RabbitMQ is very easy to maintain compared to the Apache Kafka.

In simple words, if your organization is going to handle a small amount of data on day-to-day basis,

then RabbitMQ is a good choice.

Otherwise, you can always go with the Apache Kafka.

Introduction to Apache Kafka



Here we have producers at the top who holds the source of data.

So, they are going to continuously stream the data to the consumers, like television and speakers with the help of this receiver.

So, this receiver is going to act as a broker.

So now if you try to correlate this to the Apache Kafka, the Apache Kafka is going to act as a broker

in between the producers and consumers, the same we have seen inside a RabbitMQ scenario also.

But Apache Kafka can stream any amount of data, whereas RabbitMQ it cannot stream large

amount of data.

It can only process very limited amount of data.

So that is why I am trying to compare the receiver that we have in this scenario to the Apache Kafka,

but not to the RabbitMQ. With this example, at least, I am assuming you understand that Apache Kafka is an event streaming broker which always accepts the data from the producers and send the same to the consumers.

So, with that quick introduction, let us go to the actual definition of Apache Kafka.

So, Apache Kafka is an open-source distributed event streaming platform like we discussed previously.

It is designed to handle large scale of data at real time and it is also capable of streaming the

data at real time.

Apart from streaming the data, it is also capable of high throughput, fault tolerant and scalable

data processing.

Apache Kafka is mainly used to build the real time streaming data pipelines and applications that adopt to that data streams.

So, you can have a producer application continuously sending the data in the format of logs or in the

format like Json, XML.

So regardless of whatever format that you are trying to use the producers, they can continuously send that data.

And the other side, the consumers, they are going to process the data by accepting the same from the Kafka.

So, inside the Apache Kafka, there are many important components like producers, topics, brokers,

partitions, offsets and very similarly, we also have replication consumer groups, streams.

So, these are all the important components available inside the Apache Kafka.

Let me try to explain about all these components by taking a visual example.

**Producers**

We all know the very first important component inside any event streaming platform is the producers.

So, the producers are the application who are going to responsible to produce a data or to produce an event.

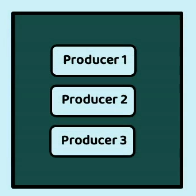
That is why we call them as producer.

So, inside your application you can have any number of producers.

So, these producers, they will connect to a Kafka cluster and they are going to be continuously pushing the messages into the Kafka cluster.

So, when I say that data is going to be continuously pushed by the producers into the Kafka cluster,

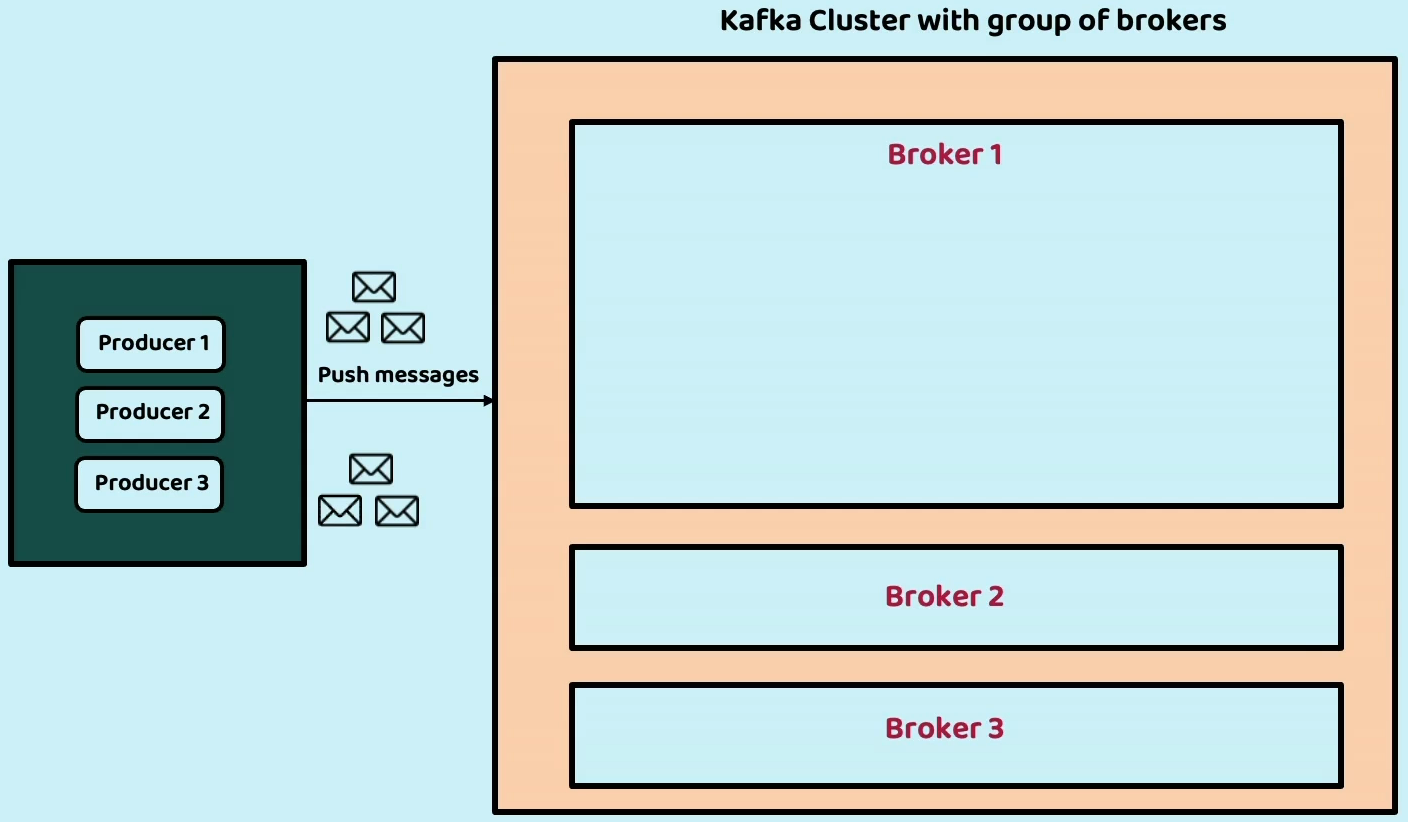
we need to understand how it is going to happen or how it is going to be handled within the Kafka cluster.



**Cluster**

So, what is a cluster?

It is a set of servers which are going to work together to produce a desired output. Very similarly inside Kafka cluster. We can have any number of brokers.



So, a broker represents a server inside the Kafka cluster that can accept the data from

the producers and sending the same to the consumers.

So, inside this cluster, I have given three brokers like broker1, broker2, broker3 in

production environment.

The recommendation is to have at least three brokers so that your messages can be replicated at least in two different brokers.

Even if we lost one of the brokers due to some earthquake or fire accident, your data is going to be

safely available as a backup inside the broker2.

And these brokers, they are not going to be deployed in the same geographical location.

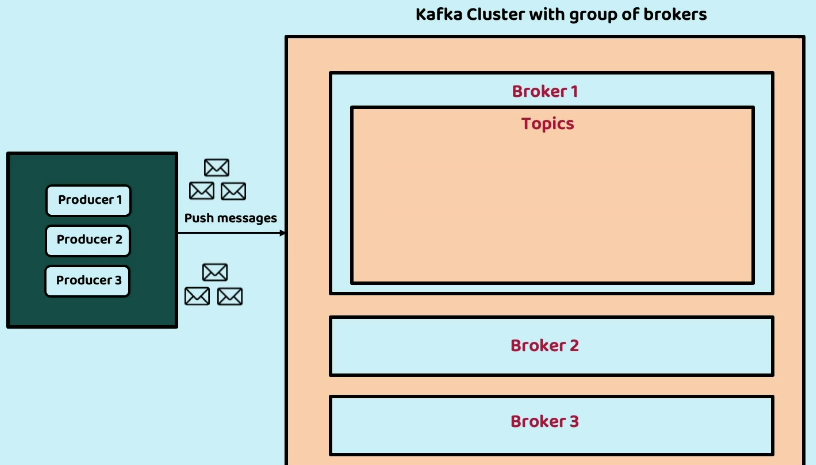
Multiple brokers will be deployed at multiple geographical locations so that your replicated messages

at data are going to be stayed in different geographical locations.

So now think like I have a cluster and inside the cluster I have three brokers.

So now let us try to understand what is going to happen inside a broker.

Inside a broker, we can have topics.



A topic is very similar to exchange that we have inside the RabbitMQ.

So, my producers, they are going to connect with a topic and whatever messages that they want to send, they are going to send a topic.

So, inside an organization we can have various topics like to send communication or to process the refund of the payment.

So based upon your scenarios, you can have multiple topics and accordingly you can configure a producer to a specific topic.

And whenever that event happens inside a producer, it is going to push a message or event into a topic.

So, a broker can have any number of topics.

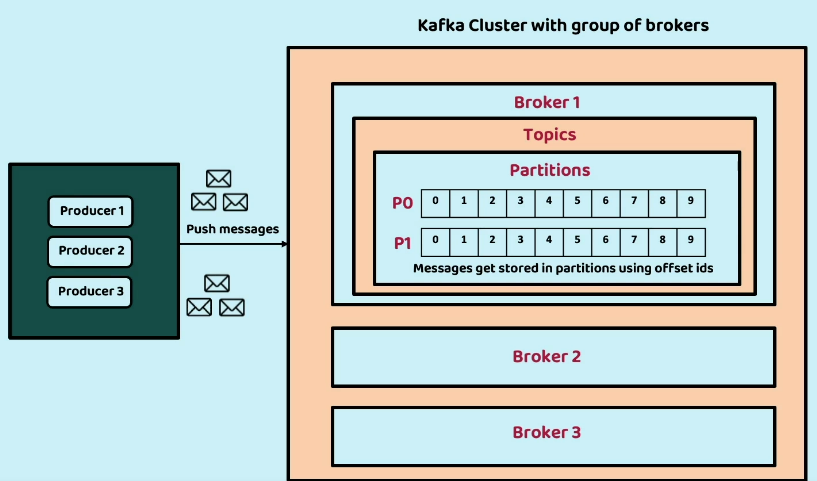
Just now I said, all your producers, they are going to send the messages and the events to the topics.

Does that mean all the data that I pushed into the topics are going to be stored directly inside the

topics?

No, it is not going to work that way.

Inside the topics we are going to have multiple partitions.



So, what is the purpose of partitions?

Like we discussed previously, Kafka can store any amount of data.

Does that mean I can store all data related to a topic inside a single broker?

It is not going to be possible since I want to store the large amount of data with multiple brokers

inside multiple brokers, I am going to have a same topic but different partitions like P0, P1, P2,

P3..

For example, take a scenario where I am going to trigger an event whenever a new account is created inside a bank application.

So, this event is responsible to send the communication to my end user.

So, my bank application may have millions of customers and sending all the messages or data of all these millions of customers into a particular topic or into a particular broker is not going to be a recommended approach.

Because it may not be possible to store all the data inside a single broker.

So, to handle these kinds of scenarios, Kafka has partitions.

I can build logic such a way that all the communication that I want to send for the **New York** customers needs to be go to the partition **P0**.

And similarly, all the customer communication details related to the **Washington**, it must go to the other partition **P1**.

So, this way, based upon my business use case, I can separate my messages or events to various partitions in a single topic and this gives me flexibility to store any amount of data because I can add any number of brokers inside my cluster.

And whenever a message we are trying to store inside a partition, we are going to provide an offset

ID to the message right from 0 to 1, two, three, four, so on.

It will keep on increasing the offset number.

So, what is the purpose of this offset number or offset ID?

So, this will give flexibility to the Apache Kafka and consumers to uniquely identify a particular

message.

So, whenever my consumer says that it has processed all the messages up to the offset ID 9 inside

the P0 partition, then it indicates it has processed all the messages that we stored inside the

P0 partition.

So, in other words, these offset IDs are like sequence ID's that we have inside the database rows.

So, whenever we try to create a new row inside a database table, we are going to assign a sequence number.

Very similarly, we are going to assign the offset ID for each message.

And here you may have a question like under P0 and P1, the offset IDs are being duplicated like 0011.

Will this not create any issues?

Off course it is not going to create any issues because apart from the offset ID the consumers, they

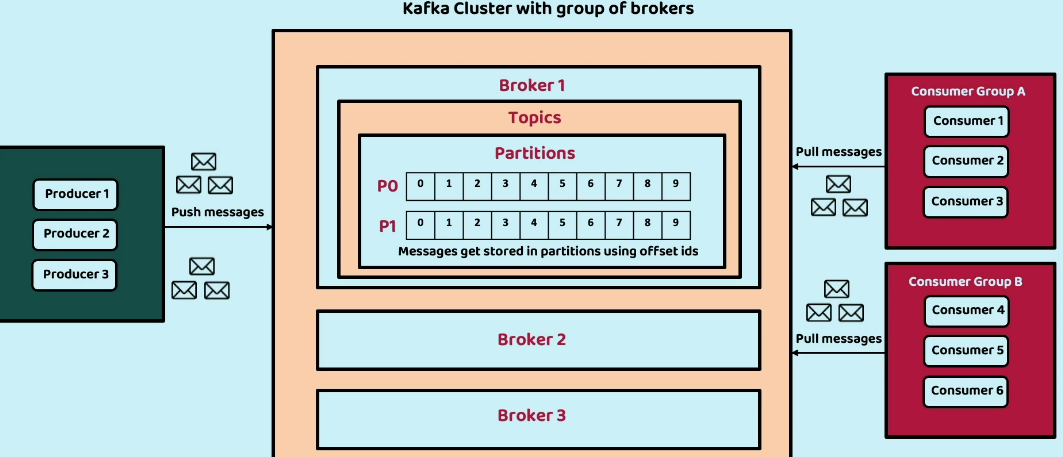
are also going to consider the partition number.

So, the combination of **topic**, **partition** and the **offset ID** is going to be a unique always inside your Apache Kafka.

So once my message is stored inside a topic with the help of partition and after assigning the offset

ID, these messages are ready to be consumed by my consumers.

If you try to see the story on the other side and the other side, I am going to have multiple consumers who are continuously trying to pull the messages from a particular topic and the partition that they have registered.



And here you can see I can also group my consumers with the concept of **consumer group** inside the Apache Kafka.

So, there are two groups here like **Consumer Group A** and **Consumer Group B**, so what is the purpose of this grouping of consumers?

I can group my consumers that are responsible to process the data available inside a particular topic.

Maybe I can have group of consumers which are responsible to pull the messages under a topic with the name send a communication.

So whatever messages that we are going to have in this send communication topic, all the messages are going to be processed by the set of consumers.

So here with the setup of consumer group, it is also possible to process the messages parallelly.

I can configure for the Consumer 1 to always process the messages available inside the Partition 0.

And similarly Partition 1 messages can be processed by the Consumer 2.

So, this way I can have my consumers to parallelly process the messages as they are coming into the

topic from the producer side.

So, this is the plain story that is going to happen behind the scenes whenever you try to implement Apache Kafka.

Apart from these one more advantage of Apache Kafka is, it is going to replicate your data in multiple

brokers.

Like if you try to save a message into one of the brokers, the same is also going to be replicated in

one more broker, like broker2 or broker3.

This way, your Kafka data processing is also fault tolerant, even if you lose the message available

inside the broker1 you are going to have a copy of it inside the broker2 or broker3.

I hope now you are clear with all the important components available inside the Apache Kafka.

Now if you try to look at the definitions that I have provided, they are going to make sense to you.

So, let us try to revisit the definitions very quickly.

So first we have producers like we discussed.

Producers are responsible to produce the messages or events by writing them into a specific topic.

So, what is a topic?

Kafka organizes data into topics.

A topic is a particular stream of data that can be divided into multiple partitions.

And whenever we are trying to write a message inside a partition, it is going to be identified by an

offset ID.

Next, we have brokers.

Brokers are nothing but Kafka servers which are going to store the data and replicate the data that

we receive, and they are also responsible to receive the messages from producers and assigning the

offset IDs to the messages and serving the messages to the consumers. And post the brokers

we already discussed what is partitions, what is offset ID, what is the importance of them.

Now moving on to the next important components replication.

Like we discussed with the help of replication, Kafka is going to replicate your data across multiple

brokers to ensure fault tolerance.

So, with the help of this replication, your data is going to be maintained at multiple locations, allowing that you are ready for failover and high availability.

After replication, we have consumers, consumer groups we already discussed what are those now moving on to the last component, which is streams.

So, Kafka Streams is a client library that enables stream processing within Kafka. With the help of

the libraries available inside the streams.

Any application they can produce data in real time and send the same to the Kafka server and at the

same time, on the consumer side, any kind of application.

It can also connect to the Kafka server and consume the data and process the same.

Producer and Consumer side streams

Now, inside this lecture, let me share more details on what is going to happen whenever a producer

is trying to send a message to the Kafka server.

And at the same time, whenever a consumer is trying to read the message from the Kafka server.

The very first action that needs to happen on the producer side is the producer must be configured, which means inside my application, a producer needs to be configured.

And this involves setting up properties like what is the endpoint URL of Kafka broker, what is the serialization format for messages and other optional configurations like whether you as a

producer must follow any compression or batching to send the messages.

Once the producer related configuration is completed.

Now my producer must select the topic that it needs to send the messages, which means whenever a producer wants to send a message to the Kafka, it needs to mention what is the topic to which it wants to stream the data within the Kafka.

If the topic does not exist based upon your configurations on the broker side, the topic can be created dynamically.

So now think like producer side everything is configured and topic selection is also completed.

As a next step, my producer can send the message to the Kafka by using the Kafka client Libraries APIs.

The producer must specify what is the target topic and what is the serialized message that it is trying

to send to the Kafka broker.

And if needed, the producer also can provide what is a partition key that it wants to consider while

saving the message inside the Kafka.

This is completely optional.

As a next step, as soon as the message is received by my Kafka broker, the Kafka is going to assign your message to one of the partitions.

If the partition key is provided by the producer the same, it is going to be used by the Kafka broker

and inside the same partition it is going to store.

If no partition key is provided, then Kafka uses various approaches like Round Robin or hashing algorithm to distribute your messages evenly across various partitions.

Once my Kafka broker identified to which partition, the message must be stored, the Kafka is going

to do an offset assignment, which means it is going to assign an offset ID to the message that it received from the producer and post that it is going to append the message to the partition that it has identified.

And once the message is successfully stored inside a partition with the help of offset ID as a next

step, if you have enabled the replication, the message replication will be done by the Kafka broker.

So, the Kafka is going to make sure that your message is going to be copied in other Kafka brokers based upon your configurations.

And this message replication can happen asynchronously or synchronously based upon your configurations.

Once the message replication is completed as a last step, your Kafka broker is going to give an acknowledgment to your producer.

If there is some error happen the same, it is going to convey to the producer.

So, the producer must receive the acknowledgment from the Kafka.

If there is some error the same, it must handle inside the business logic.

Otherwise, the producer can continue processing its next business logic without worrying about who is going to process your message.

The Kafka is going to take care of that by assigning that to one of the consumers.

Here based upon the acknowledgment and the error received by the producer from the Kafka broker.

If needed, they can also try multiple retry attempts to send the message to the Kafka broker and these acknowledgments you are going to receive based upon your mode of configuration.

The producer may wait for the acknowledgment from all the replicas or it can simply wait for only

the leader replica.

So, what is leader replica when your message is stored inside a broker very first time it is going to

be the leader replica.

Once the leader replica is completed, the other replicas Kafka broker is going to do behind the scenes.

So here the producer has an option to completely wait for all the replicas to complete or to simply

wait for the leader replica to complete.

So, this is what is going to happen on the producer side.

Now think like our message right now stored inside the Kafka broker successfully. As a next step,

let us try to understand what is going to happen on the consumer side.

Like we discussed before, all our consumers, they are going to assign to a consumer group and each

consumer inside a consumer group, they need to subscribe to one of the topics available inside the Kafka broker. Which means before a consumer tried to read a message from a Kafka broker.

It needs to join a consumer group and all the consumers assigned to a consumer group. They are going to subscribe to one or more topics.

And this subscription also specifies which topics the consumer wants to consume, messages from the Kafka Broker.

Once a consumer is assigned to the consumer group and it subscribed to one of the topics as a next step, my Kafka is going to assign the partitions of the subscribed topics to the consumers available within the consumer group.

So, each partition can be consumed by only one consumer in the group.

This will ensure a balanced distribution of partition among consumers and to achieve the parallel processing.

So now, from the first two steps, the consumer is assigned to a consumer group.

It also assigned to a topic and inside the topic it also assigned to a partition.

As a next step, the offset management must be done by my consumer.

That means the consumer should maintain its offset details for each partition it is trying to consume.

Initially, the offset number is going to be the null because my consumer never consumed any messages inside a partition as it is trying to process the messages from a partition, the consumer needs to update its offset to keep track of the progress.

Otherwise, my consumer will never know up to which offset number it has processed previously.

So once my consumer understands like up to which offset number it has processed previously as a next step, it can try to send a fetch request to the Apache Kafka broker.

So, inside this the consumer is going to send a **fetch request** and as part of this fetch request, it

will mention what is the **topic**, what is the **partition** and what is the **offset number** from which the

consumer want to read the message.

And inside this request also the consumer can mention what is the number of messages that it is trying to fetch in each request.

Unlike RabbitMQ inside Apache Kafka, a consumer can read multiple messages at a time, like my consumer can try to fetch 100 messages at a time inside each fetch request.

This is going to improve the performance of your Kafka setup whenever you are trying to handle the large amount of data.

Once my Kafka broker receives this fetch request, the Kafka broker is going to retrieve the requested

messages from the corresponding partitions log and the same messages it is going to send back to the consumer as a fetch response.

The response contains all the messages, their associated offsets, and metadata information.

As a next step, obviously, the messages will be processed on the consumer side based upon the business logic that we have written.

This process can involve transforming the messages or aggregation, calculations, or any other operations based upon the business requirements.

After message processing, my consumer must commit the offset number to the broker, saying that I have processed the messages up to this offset number.

This confirms to the Apache Kafka broker that my consumer processed all the messages up to this offset number.

So, committing these offset number ensures that consumers progress is persisted inside the Kafka broker, and the same can be resumed from that point in case of failure or restart is going to happen.

So, this process of fetching requests, processing them, committing the offset is going to continuously

happen as part of the polling loop.

So, the consumer continuously is going to repeat the step four, five, six, seven.

And this way, as soon as the messages are going to be arrived into a partition inside a topic, immediately my consumer is going to process them in near real time.

Installation of Apache Kafka

<https://kafka.apache.org/>

<https://kafka.apache.org/documentation/#docker>

Docker image can be pulled from Docker Hub using the following command:-

$ docker pull apache/kafka:3.7.1

If you want to fetch the latest version of the docker image use following command:-

$ docker pull apache/kafka:latest

**Start kafka with default configs**

Run docker image on default port 9092:-

$ docker run -p 9092:9092 apache/kafka:3.7.1

Implement & Demo of Async communication or events streaming using Kafka

Install VSCode Kafka plugin

Tools for Apache Kafka

Demo of Async comm or event streaming using Docker containers and Docker Compose