What is Kafka?

We use Apache Kafka when it comes to enabling communication between producers and consumers using message-based topics. Apache Kafka is a fast, scalable, fault-tolerant, publish-subscribe messaging system. Basically, it designs a platform for high-end new generation distributed applications. Also, it allows a large number of permanent or ad-hoc consumers. One of the best features of Kafka is, it is highly available and resilient to node failures and supports automatic recovery. This feature makes Apache Kafka ideal for communication and integration between components of large-scale data systems in real-world data systems.

Moreover, this technology replaces the conventional message brokers, with the ability to give higher throughput, reliability, and replication like JMS, AMQP and many more. In addition, core abstraction Kafka offers a **Kafka broker**, a[**Kafka Producer**](https://data-flair.training/blogs/kafka-producer/), and a [**Kafka Consumer**](https://data-flair.training/blogs/kafka-consumer/). Kafka broker is a node on the Kafka cluster, its use is to persist and replicate the data. A Kafka Producer pushes the message into the message container called the Kafka Topic. Whereas a Kafka Consumer pulls the message from the Kafka Topic.

**a. Messaging System in Kafka**

When we transfer data from one application to another, we use the Messaging System. It results as, without worrying about how to share data, applications can focus on data only. On the concept of reliable message queuing, distributed messaging is based. Although, messages are asynchronously queued between client applications and messaging system. There are two types of messaging patterns available, i.e. point to point and publish-subscribe (pub-sub) messaging system. However, most of the messaging patterns follow pub-sub.

* **Point to Point Messaging System(QUEUE)**

Here, messages are persisted in a queue. Although, a particular message can be consumed by a maximum of one consumer only, even if one or more consumers can consume the messages in the queue. Also, it makes sure that as soon as a consumer reads a message in the queue, it disappears from that queue.

* **Publish-Subscribe Messaging System**

Here, messages are persisted in a topic. In this system, Kafka Consumers can subscribe to one or more topic and consume all the messages in that topic. Moreover, message producers refer publishers and message consumers are subscribers here.

History of Apache Kafka

Previously, LinkedIn was facing the issue of low latency ingestion of huge amount of data from the website into a lambda architecture which could be able to process real-time events. As a solution, Apache Kafka was developed in the year 2010, since none of the solutions was available to deal with this drawback, before.

However, there were technologies available for batch processing, but the deployment details of those technologies were shared with the downstream users. Hence, while it comes to Real-time Processing, those technologies were not enough suitable. Then, in the year 2011 Kafka was made public.

Why Should we use Apache Kafka Cluster?

As we all know, there is an enormous volume of data in [**Big Data**](https://data-flair.training/blogs/what-is-big-data/). And, when it comes to big data, there are two main challenges. One is to collect the large volume of data, while another one is to analyze the collected data. Hence, in order to overcome those challenges, we need a messaging system. Then Apache Kafka has proved its utility. There are numerous [**benefits of Apache Kafka**](https://data-flair.training/blogs/advantages-and-disadvantages-of-kafka/)such as:

* Tracking web activities by storing/sending the events for real-time processes.
* Alerting and reporting the operational metrics.
* Transforming data into the standard format.
* Continuous processing of streaming data to the topics.

Therefore, this technology is giving a tough competition to some of the most popular applications like ActiveMQ, RabbitMQ, AWS etc. because of its wide use.

As of now, we discussed the core concepts of Kafka. Let us now throw some light on the workflow of Kafka.

Kafka is simply a collection of topics split into one or more partitions. A Kafka partition is a linearly ordered sequence of messages, where each message is identified by their index (called as offset). All the data in a Kafka cluster is the disjointed union of partitions. Incoming messages are written at the end of a partition and messages are sequentially read by consumers. Durability is provided by replicating messages to different brokers.

Kafka provides both pub-sub and queue based messaging system in a fast, reliable, persisted, fault-tolerance and zero downtime manner. In both cases, producers simply send the message to a topic and consumer can choose any one type of messaging system depending on their need. Let us follow the steps in the next section to understand how the consumer can choose the messaging system of their choice.

## **Workflow of Pub-Sub Messaging**

Following is the step wise workflow of the Pub-Sub Messaging −

* Producers send message to a topic at regular intervals.
* Kafka broker stores all messages in the partitions configured for that particular topic. It ensures the messages are equally shared between partitions. If the producer sends two messages and there are two partitions, Kafka will store one message in the first partition and the second message in the second partition.
* Consumer subscribes to a specific topic.
* Once the consumer subscribes to a topic, Kafka will provide the current offset of the topic to the consumer and also saves the offset in the Zookeeper ensemble.
* Consumer will request the Kafka in a regular interval (like 100 Ms) for new messages.
* Once Kafka receives the messages from producers, it forwards these messages to the consumers.
* Consumer will receive the message and process it.
* Once the messages are processed, consumer will send an acknowledgement to the Kafka broker.
* Once Kafka receives an acknowledgement, it changes the offset to the new value and updates it in the Zookeeper. Since offsets are maintained in the Zookeeper, the consumer can read next message correctly even during server outrages.
* This above flow will repeat until the consumer stops the request.
* Consumer has the option to rewind/skip to the desired offset of a topic at any time and read all the subsequent messages.

## **Workflow of Queue Messaging / Consumer Group**

In a queue messaging system instead of a single consumer, a group of consumers having the same Group ID will subscribe to a topic. In simple terms, consumers subscribing to a topic with same Group ID are considered as a single group and the messages are shared among them. Let us check the actual workflow of this system.

* Producers send message to a topic in a regular interval.
* Kafka stores all messages in the partitions configured for that particular topic similar to the earlier scenario.
* A single consumer subscribes to a specific topic, assume Topic-01 with Group ID as Group-1.
* Kafka interacts with the consumer in the same way as Pub-Sub Messaging until new consumer subscribes the same topic, Topic-01 with the same Group ID as Group-1.
* Once the new consumer arrives, Kafka switches its operation to share mode and shares the data between the two consumers. This sharing will go on until the number of con-sumers reach the number of partition configured for that particular topic.
* Once the number of consumer exceeds the number of partitions, the new consumer will not receive any further message until any one of the existing consumer unsubscribes. This scenario arises because each consumer in Kafka will be assigned a minimum of one partition and once all the partitions are assigned to the existing consumers, the new consumers will have to wait.
* This feature is also called as Consumer Group. In the same way, Kafka will provide the best of both the systems in a very simple and efficient manner.

## **Role of ZooKeeper**

A critical dependency of Apache Kafka is Apache Zookeeper, which is a distributed configuration and synchronization service. Zookeeper serves as the coordination interface between the Kafka brokers and consumers. The Kafka servers share information via a Zookeeper cluster. Kafka stores basic metadata in Zookeeper such as information about topics, brokers, consumer offsets (queue readers) and so on.

Since all the critical information is stored in the Zookeeper and it normally replicates this data across its ensemble, failure of Kafka broker / Zookeeper does not affect the state of the Kafka cluster. Kafka will restore the state, once the Zookeeper restarts. This gives zero downtime for Kafka. The leader election between the Kafka broker is also done by using Zookeeper in the event of leader failure.

Kafka Architecture

Four core APIs in this Apache Kafka :

**a. Kafka Producer API**

This Kafka Producer API permits an application to publish a stream of records to one or more Kafka topics.

**b. Kafka Consumer API**

To subscribe to one or more topics and process the stream of records produced to them in an application, we use this Kafka Consumer API.

**c. Kafka Streams API**

In order to act as a stream processor consuming an input stream from one or more topics and producing an output stream to one or more output topics and also effectively transforming the input streams to output streams, this Kafka Streams API gives permission to an application.

**d. Kafka Connector API**

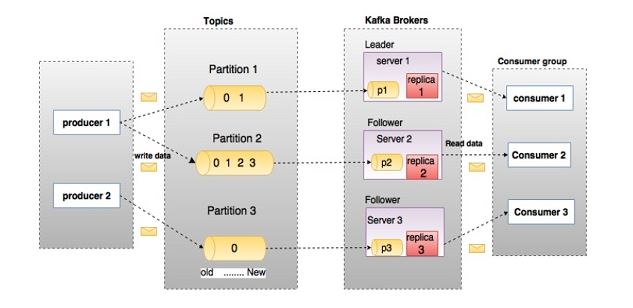
This Kafka Connector API allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems. For example, a connector to a relational database might capture every change to a table.

Kafka Components

Using the following components, Kafka achieves messaging:

a. Kafka Topic

Basically, how Kafka stores and organizes messages across its system and essentially a collection of messages are Topics. In addition, we can replicate and partition Topics. Here, replicate refers to copies and partition refers to the division. Also, visualize them as logs wherein, Kafka stores messages. However, this ability to replicate and partitioning topics is one of the factors that enable Kafka’s fault tolerance and scalability.



b. Kafka Producer

It publishes messages to a Kafka topic.

c. Kafka Consumer

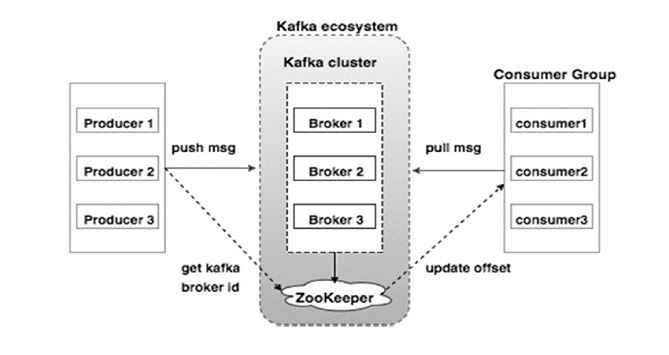
This component subscribes to a topic(s), reads and processes messages from the topic(s).

d. Kafka Broker

Kafka Broker manages the storage of messages in the topic(s). If Kafka has more than one broker, that is what we call a Kafka cluster.

e. Kafka Zookeeper

To offer the brokers with metadata about the processes running in the system and to facilitate health checking and broker leadership election, Kafka uses Kafka[**zookeeper**](https://data-flair.training/blogs/zookeeper-in-kafka/).

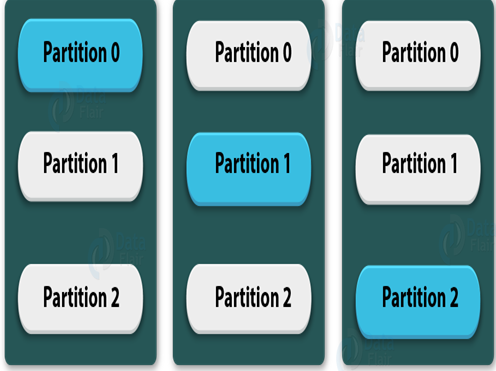


Kafka Data Log

By Kafka, messages are retained for a considerable amount of time. Also, consumers can read as per their convenience. However, if Kafka is configured to keep messages for 24 hours and a consumer is down for time greater than 24 hours, the consumer will lose messages. And, messages can be read from last known offset, if the downtime on part of the consumer is just 60 minutes. Kafka doesn’t keep state on what consumers are reading from a topic.

Kafka Partition in Kafka

There are few partitions in every Kafka broker. Moreover, each partition can be either a leader or a replica of a topic. In addition, along with updating of replicas with new data, Leader is responsible for all writes and reads to a topic. The replica takes over as the new leader if somehow the leader fails.



Importance of Java in Apache Kafka

Apache Kafka is written in pure [**Java**](https://data-flair.training/blogs/java-tutorial/) and also Kafka’s native API is java. However, many other languages like C++, [**Python**](https://data-flair.training/blogs/python-tutorial/)**,** .Net, Go, etc. also support Kafka. Still, a platform where there is no need of using a third-party library is Java. Also, we can say, writing code in languages apart from Java will be a little overhead.

In addition, we can useJavalanguage if we need the high processing rates that come standard on Kafka. Also, Java provides a good community support for Kafka consumer clients. Hence, it is a right choice to implement Kafka in Java.

Kafka Use Cases

There are several[**use Cases of Kafka**](https://data-flair.training/blogs/kafka-use-cases-and-applications/) that show why we actually use Apache Kafka.

* **Messaging**

For a more traditional message broker, Kafka works well as a replacement. We can say Kafka has better throughput, built-in partitioning, replication, and fault-tolerance which makes it a good solution for large-scale message processing applications.

* **Metrics**

For operational monitoring data, Kafka finds the good application. It includes aggregating statistics from distributed applications to produce centralized feeds of operational data.

* **Event Sourcing**

Since it supports very large stored log data, that means Kafka is an excellent backend for applications of event sourcing.

Kafka — Comparisons in Kafka

Many applications offer the same functionality as Kafka like ActiveMQ, RabbitMQ, [**Apache Flume**](https://data-flair.training/blogs/apache-flume-tutorial/), Storm, and [**Spark**](https://data-flair.training/blogs/spark-tutorial/). Then why should you go for Apache Kafka instead of others?

Let’s see the comparisons below:

a. Apache Kafka vs Apache Flume

Apache Kafka vs Flume

***i. Types of tool***

**Apache Kafka**– For multiple producers and consumers, it is a general-purpose tool.

**Apache Flume**– Whereas, it is a special-purpose tool for specific applications.

***ii. Replication feature***

**Apache Kafka**– Using ingest pipelines, it replicates the events.

**Apache Flume-** It does not replicate the events.

b. RabbitMQ vs Apache Kafka

One among the foremost Apache Kafka alternatives is RabbitMQ. So, let’s see how they differ from one another:

***i. Features***

**Apache Kafka**– Basically, Kafka is distributed. Also, with guaranteed durability and availability, the data is shared and replicated.

**RabbitMQ**– It offers relatively less support for these features.

***ii. Performance rate***

**Apache Kafka**— Its performance rate is high to the tune of 100,000 messages/second.

**RabbitMQ**— Whereas, the performance rate of RabbitMQ is around 20,000 messages/second.

***iii. Processing***

**Apache Kafka**— It allows reliable log distributed processing. Also, stream processing semantics built into the Kafka Streams.

**RabbitMQ** — Here, the consumer is just FIFO based, reading from the HEAD and processing 1 by 1.

c. Traditional queuing systems vs Apache Kafka

Traditional queuing systems vs Apache Kafka

***i. Messages Retaining***

**Traditional queuing systems**— Most queueing systems remove the messages after it has been processed typically from the end of the queue.

**Apache Kafka**— Here, messages persist even after being processed. They don’t get removed as consumers receive them.

***ii. Logic-based processing***

**Traditional queuing systems**— It does not allow to process logic based on similar messages or events.

**Apache Kafka**— It allows to process logic based on similar messages or events.