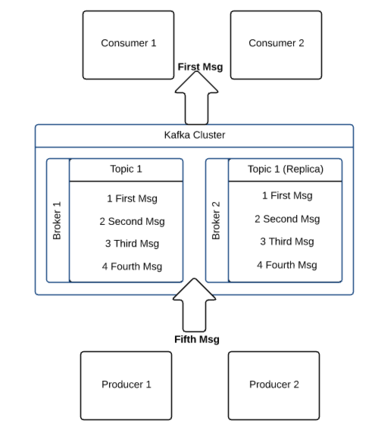


When the big data movement started it was mostly focused on batch processing. Distributed data storage and querying tools like MapReduce, Hive, and Pig were all designed to process data in batches rather than continuously. Businesses would run multiple jobs every night to extract data from a database, then analyse, transform, and eventually store the data. More recently enterprises have discovered the power of analysing and processing data and events as they happen, not just once every few hours. Most traditional messaging systems don't scale up to handle big data in real-time, however. So engineers at LinkedIn [built and open-sourced Apache Kafka](https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying): a distributed messaging framework that meets the demands of big data by scaling on commodity hardware.

* It's designed to scale horizontally, by adding more commodity servers.
* It provides much higher throughput for both producer and consumer processes.
* It can be used to support both batch and real-time use cases.
* It doesn't support JMS, Java's message-oriented middleware API.



Apache Kafka's architecture is very simple, which can result in better performance and throughput in some systems. Every topic in Kafka is like a simple log file. When a producer publishes a message, the Kafka server appends it to the end of the log file for its given topic. The server also assigns an offset, which is a number used to permanently identify each message. As the number of messages grows, the value of each offset increases; for example if the producer publishes three messages the first one might get an offset of 1, the second an offset of 2, and the third an offset of 3.

In Kafka, the client is responsible for remembering the offset count and retrieving messages. The Kafka server doesn't track or manage message consumption. By default, a Kafka server will keep a message for seven days. A background thread in the server checks and deletes messages that are seven days or older. A consumer can access messages as long as they are on the server. It can read a message multiple times, and even read messages in reverse order of receipt. But if the consumer fails to retrieve the message before the seven days are up, it will miss that message.

**What is Apache Kafka?**

Apache Kafka is a distributed streaming system with publish and subscribe the stream of records. In another aspect it is an enterprise messaging system. It is highly fast, horizontally scalable and fault tolerant system. Kafka has four core APIs called,

**Producer API:**

This API allows the clients to connect to Kafka servers running in cluster and publish the stream of records to one or more Kafka topics.

**Consumer API:**

This API allows the clients to connect to Kafka servers running in cluster and consume the streams of records from one or more Kafka topics. Kafka consumers PULLS the messages from Kafka topics.

**Streams API:**

This API allows the clients to act as stream processors by consuming streams from one or more topics and producing the streams to other output topics. This allows to transform the input and output streams.

**Connector API:**

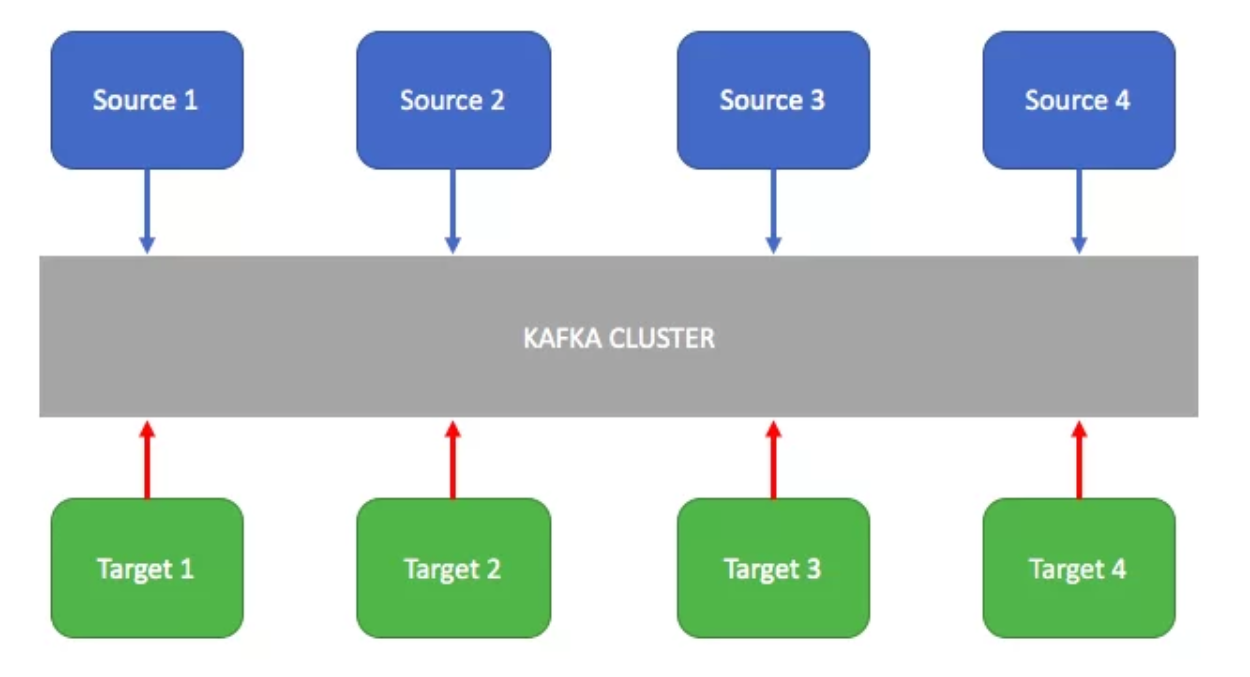
This API allows to write reusable producer and consumer code. For example, if we want to read data from any RDBMS to publish the data to topic and consume data from topic and write that to RDBMS. With connector API we can create reusable source and sink connector components for various data sources.

**What use cases Kafka used for?**

Kafka is used for the below use cases,

**Messaging System:**

Kafka used as an enterprise messaging system to decouple the source and target systems to exchange the data. Kafka provides high throughput with partitions and fault tolerance with replication compared to JMS.



**Web Activity Tracking:**

To track the user journey events on the website for analytics and offline data processing.

**Log Aggregation:**

To process the log from various systems. Especially in the distributed environments, with micro services architectures where the systems are deployed on various hosts. We need to aggregate the logs from various systems and make the logs available in a central place for analysis. Go through the article on distributed logging architecture where Kafka is used.

<https://smarttechie.org/2017/07/31/distributed-logging-architecture-for-micro-services/>

**Metrics Collector:**

Kafka is used to collect the metrics from various systems and networks for operations monitoring. There are Kafka metrics reporters available for monitoring tools like Ganglia, Graphite etc…

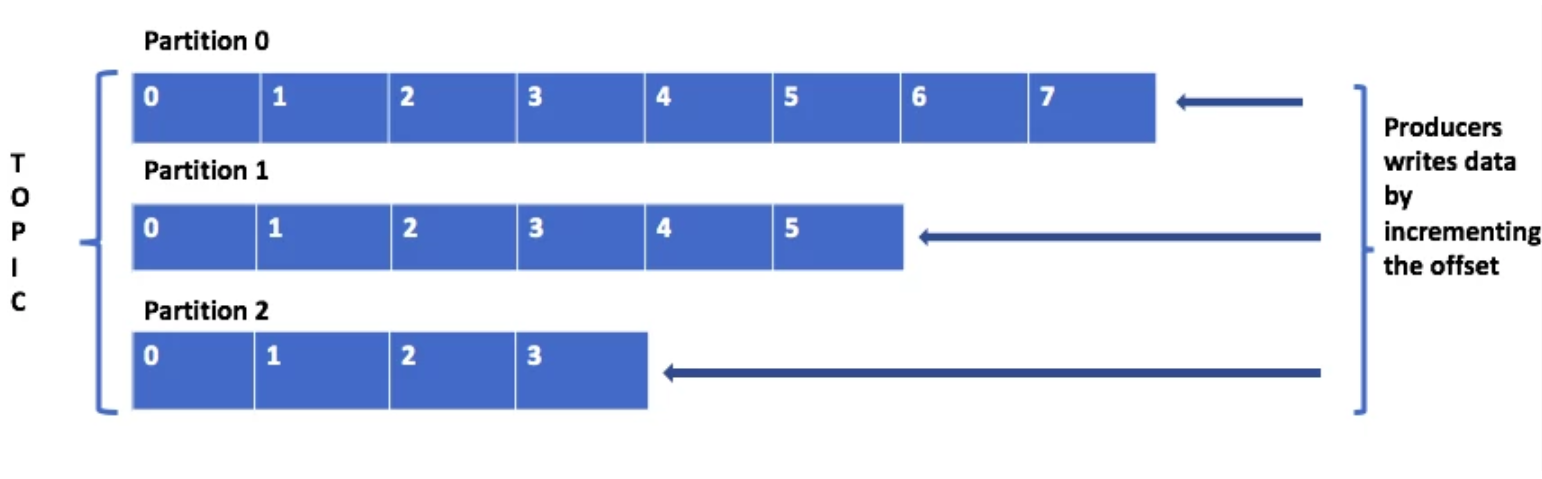
Some references on this https://github.com/stealthly/metrics-kafka

**What is broker?**

An instance in a Kafka cluster is called as broker. In a Kafka cluster if you connect to any one broker you will be able to access entire cluster. The broker instance which we connect to access cluster is also known as bootstrap server. Each broker is identified by a numeric id in the cluster. To start with Kafka cluster three brokers is a good number. But there are clusters which has hundreds of brokers in it.

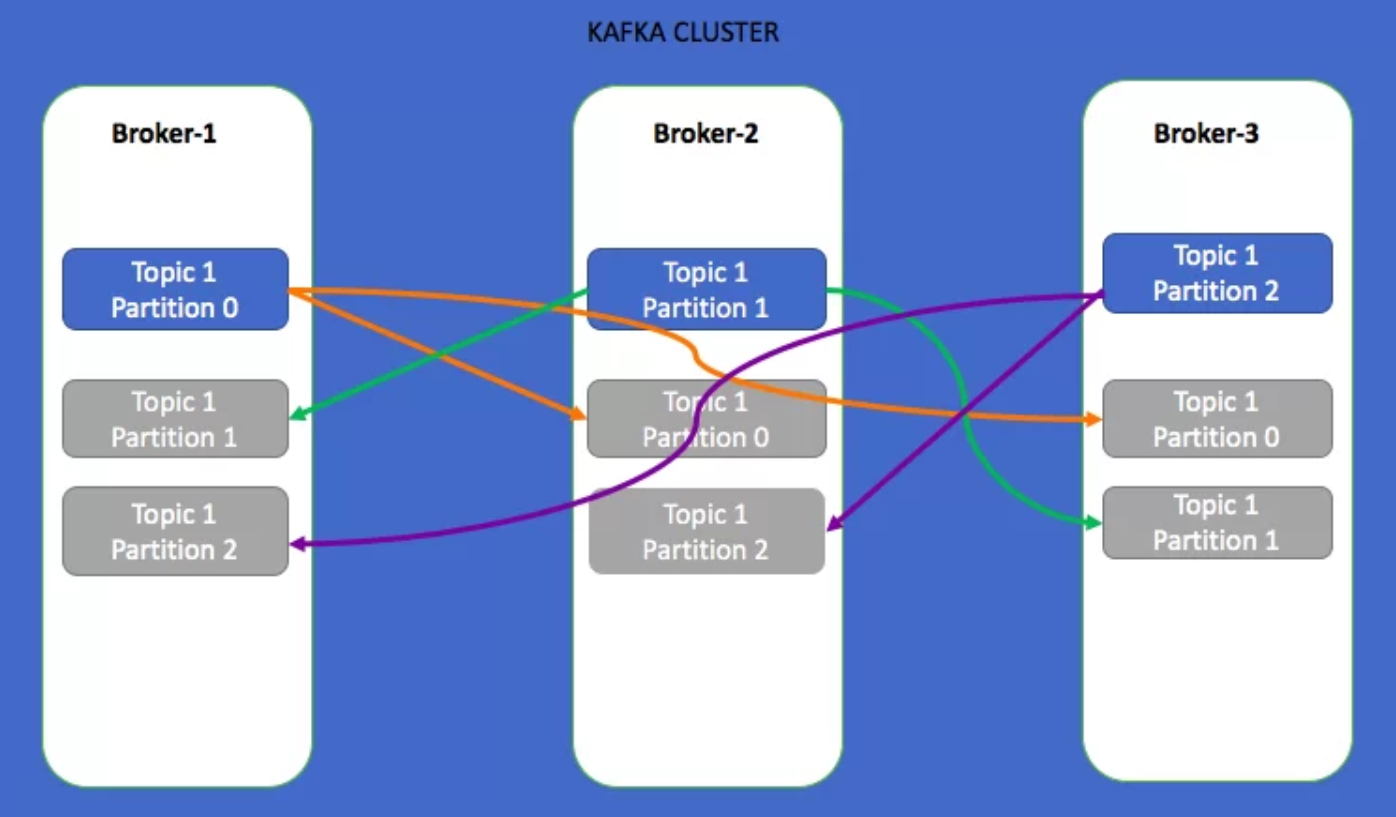
**What is Topic?**

A topic is a logical name to which the records are published. Internally the topic is divided into partitions to which the data is published. These partitions are distributed across the brokers in cluster. For example if a topic has three partitions with 3 brokers in cluster each broker has one partition. The published data to partition is append only with the offset increment.



Below are the couple of points we need to remember while working with partitions.

* Topics are identified by its name. We can have many topics in a cluster.
* The order of the messages is maintained at the partition level, not across topic.
* Once the data written to partition is not overridden. This is called immutability.
* The message in partitions are stored with key, value and timestamp. Kafka ensures to publish the message to same partition for a given key.
* From the Kafka cluster, each partition will have a leader which will take read/write operations to that partition.



In the above example, I have created a topic with three partitions with replication factor 3. In this case as the cluster is having 3 brokers, the three partitions are evenly distributed and the replicas of each partition is replicated over to another 2 brokers. As the replication factor is 3, there is no data loss even 2 brokers goes’ down. Always keep replication factor is greater than 1 and less than or equal to number of brokers in the cluster. **You cannot create topic with replication factor more than the number of brokers in a cluster.**

In the above diagram, for each partition there is a leader (glowing partition) and other in-sync replicas(grey out partitions) are followers. For partition 0, the broker-1 is leader and broker-2, broker-3 are followers. All the reads/writes to partition 0 will go to broker-1 and the same will be copied to broker-2 and broker-3.

Source: <https://www.javacodegeeks.com/2017/11/introduction-apache-kafka.html>

<https://data-flair.training/blogs/apache-kafka-tutorial/>

<https://dzone.com/articles/kafka-producer-and-consumer-example>

https://www.javacodegeeks.com/2018/05/spring-apache-kafka-tutorial.html

With the up rise of Microservices, the necessity of asynchronous communication between the involved services became a mainstream requirement. Actually, that is how Apache Kafka came into existence at LinkedIn. The main requirements of the new asynchronous communication system they needed were message persistence and high throughput. Once LinkedIn was able to make a project in the face of Kafka, they donated the project to Apache Software foundation where it came to be known as Apache Kafka.

* **Distributed**: Kafka is a distributed system where all the messages are replicated across various nodes so that each server is capable of responding to the client for the messages it contains. Also, even if one node fails, other nodes can quickly take over without any downtime.
* **Fault-tolerant**: As Kafka does not have **a Single Point of Failure**, even if one of the node goes down, the end-user will hardly notice this as the other parts take responsibility of the messages which are lost due to the failed node.
* **Horizontally-scalable**: Kafka allows us to add more machines to the cluster with zero downtime. This means that if we start to face lag in messages due to a low number of servers in the cluster, we can quickly add more servers and maintain the performance of the system.
* **Commit Log**: A commit log refers to a structure similar to a Linked List. The order of insertion of messages is maintained and data cannot be removed from this log until a threshold time is reached.

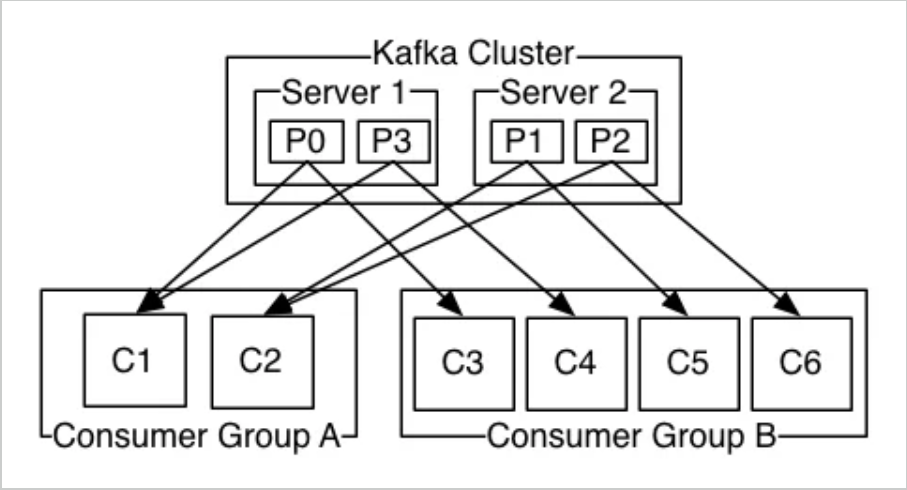
The producer needs to establish a TCP connection to publish message. Same is needed in consumer app to consume data from the Kafka cluster.

Kafka runs as a cluster and the nodes are called brokers. Brokers can be leaders or replicas to provide high-availability and fault tolerance. Brokers are in charge of partitions, being the distribution unit where messages are stored. Those messages are ordered and they’re accessible by an index called offset. A set of partitions forms a topic, being a feed of messages. A partition can have different consumers, and they access to the messages using its own offset.

**Queuing vs publish-subscribe**

Consumer groups is another key concept and helps to explain why Kafka is more flexible and powerful than other messaging solutions like RabbitMQ. Consumers are associated to consumer groups. If every consumer belongs to the same consumer group, the topic’s messages will be evenly load balanced between consumers; that’s called a ‘queuing model’. By contrast, if every consumer belongs to different consumer group, all the messages will be consumed in every client; that’s called a ‘publish-subscribe’ model.

You can have a mix of both approaches, having different logical consumer groups, for different needs, and several consumers inside of every group to increase throughput through parallelism. Again, another diagram from Kafka’s documentation:



Kafka Cluster consists of multiple brokers and a Zookeeper. Broker is an instance of Kafka that communicate with Zookeeper. Each broker holds partition(s) of topic(s). Some of those partitions are leaders and others are replicas of leader partitions from other brokers. Zookeeper is used to automatically select a leader for a partition. In case of any broker shutdown, an election is held, by Zookeeper, for leader position of partitions (that went down with the broker). Also, metadata like, in which broker a leader partition is living, etc., are held by Zookeeper. Producers that stream data to topics, or Consumers that read stream data from topics, contact Zookeeper for the nearest or less occupied broker.

**Kafka Stream Processors**

Stream Processor is an application that enrich/transform/modify the entries or records of a Topic (sometimes write these modified records to a new Topic) in Kafka Cluster.

* Stream Processors first act as sink and then as source in Kafka Cluster.
* Stream Processors are scalable. Multiple Stream Processing applications could be connected to the Kafka Cluster.
* A single Stream Processor can subscribe to the records of multiple Topics [based on configuration] and then write records back to multiple Topics.

**Kafka Connector**

Connectors are those which allow the integration of things like Relational Databases to the Kafka Cluster and automatically monitor the changes. They also help to pull those changes onto the Kafka cluster. Connectors provide a single source of ground truth data. Which means that we have a record of changes, a Topic has undergone.

**Kafka Use Cases**

Kafka can be used for various purposes in an organization, such as:

**Messaging service:** Millions of messages can be sent and received in real-time.

**Real-time stream processing:** Kafka can be used to process a continuous stream of information in real-time and pass data to stream processing systems such as Storm.

**Log aggregation:** Kafka can be used to collect physical log files from multiple systems and store it in a central location such as HDFS.

**Commit log service:** Kafka can be used as an external commit log for distributed systems.

**Event sourcing:** A time ordered sequence of events can be maintained through Kafka.

**Aggregating User Activity Using Kafka - Example**

Kafka can be used to aggregate user activity data such as clicks, navigation, and searches from different websites of an organization; such user activities can be sent to a real-time monitoring system and Hadoop system for offline processing. The information from customer-facing portals is sent in real-time to the Kafka cluster. The Kafka cluster consists of one or more servers that process the messages in parallel. The information is sent to a real-time monitoring system to monitor the user clicks, navigation, and searches. The information is also saved in a Hadoop system for offline processing.

**Topics in Apache Kafka**

A topic is a category of messages in Kafka. The producers publish the messages into topics and the consumers read the messages from topics. A topic is divided into one or more partitions.

A partition is also known as a commit log. Each partition contains an ordered set of messages. Each message is identified by its offset in the partition. Messages are added at one end of the partition and consumed at the other.

The image below illustrates a topic ‘simple’ that is divided into two partitions.

The writes are completed at one end and the reads are completed at the other. It shows six messages in partition 0 and five messages in partition 1.

The offset of message one in partition 0 is zero as it is the first message. The offset of message six in partition 0 is five.

The messages are written in the order 1, 2, 3, 4, 5 and 6, whereas, they are read in the same order as 1, 2, 3, 4, 5 and 6. The next message in partition 0 will be message 7 which will be written at offset 6. The next message for partition 1 will be message 6 which will be written at offset 5.

**Partitions in Apache Kafka**

Topics are divided into partitions, which are the unit of parallelism in Kafka. Partitions allow messages in a topic to be distributed to multiple servers or brokers so that the messages in a topic can be processed in parallel.

A topic can have any number of partitions. Each partition should fit in a single Kafka server. **The number of partitions in a topic decide the parallelism of the topic.**

The image below illustrates two partitions of a topic ‘simple.’

Partition 0 consists of six messages, whereas, partition 1 consists of five messages.

**Partition Distribution in Apache Kafka**

Partitions can be distributed across the Kafka cluster. Each Kafka server or broker may handle one or more partitions.

A partition can be replicated across several servers for fault-tolerance.

One server is marked as a leader for the partition and the others are marked as followers. The leader controls the read and writes for the partition, whereas, the followers replicate the data. If a leader fails, one of the followers automatically become the leader.

Zookeeper is used for the leader selection as explained in the previous lesson.

The image below illustrates the partitions of a topic ‘simple’.

Here, the partition 0 is assigned to server 1 and partition 1 is assigned to server 2. These servers process the messages in **parallel to increase throughput.**

**Kafka Architecture**

Kafka architecture consists of brokers that take messages from the producers and add to a partition of a topic. Brokers provide the messages to the consumers from the partitions. The producers create the messages and send them to a particular topic and a partition of a Kafka cluster. A topic is divided into multiple partitions. The messages are added to the partitions at one end and consumed in the same order. Each partition acts as a message queue.

**Types of Messaging Systems in Apache Kafka**

Kafka architecture supports two types of messaging systems known as publish-subscribe and queue system. The publish-subscribe system is also called pub-sub.

In this system, one system broadcasts the messages and the consumers subscribe to receive the messages.

Each message is received by all the subscribers. So, if there are 100 messages published, each subscriber receives all the 100 messages in the same order that they are produced.

In the queue system, each message has to be consumed by only one consumer. If there are multiple consumers, each message is consumed by any one of the available consumers, in the same order that they are received.

**Queue System - Example**

Consumer 1, consumer 2, and consumer 3 belong to the same consumer group. So out of the six messages, two messages are received by consumer 1, two messages by consumer 2, and two messages by consumer 3.

Note that the messages are received in the same order that they are produced.

So, consumer 1 receives message 1, consumer 2 receives message 2, and consumer 3 receives message 3. After this, consumer 1 receives message 4, consumer 2 receives message 5, and consumer 3 receives message 6.

**Publish-Subscribe System - Example**

Consumer 1, Consumer 2, and Consumer 3 belong to three separate consumer groups. So, all the six messages are sent to all the three consumer groups called consumer group 1, consumer group 2, and consumer group 3.

Since there is only one consumer in consumer group 1, it receives all the six messages in the order 1, 2, 3, 4, 5, and 6. Similarly, consumer 2 and consumer 3 also receive all the six messages in the same order.

**Brokers in Apache Kafka**

Each machine in the cluster can run one broker. The brokers coordinate with each other using Zookeeper. One broker acts as a leader for a partition and handles the delivery and persistence, whereas, the others act as followers. Brokers receive the message from the producer and send it to consumer groups.

**Kafka Guarantees**

**Guarantee 1:** Messages sent by a producer to a topic and a partition are appended in the same order. This ensures that the messages produced earlier do not get ahead of the messages produced later. The time order is maintained very strictly.

**Guarantee 2:** A consumer instance gets the messages in the same order as they are produced, which means that the messages are never out of order.

If the messages are produced in the order 1, 2, 3, 4, 5, 6, they will be received in the order 1, 2, 3, 4, 5, and 6. This is important in messaging systems, as the dependency is on the time order of messages.

**Guarantee 3:** A topic with replication factor N, tolerates up to N-1 server failures.

For example, when the replication factor is specified as 3, there will be no loss of messages even if two machines fail.

**Offset:** The offset is a unique identifier of a record within a partition. It denotes the position of the consumer in the partition.

<https://www.cloudkarafka.com/blog/2016-11-30-part1-kafka-for-beginners-what-is-apache-kafka.html>

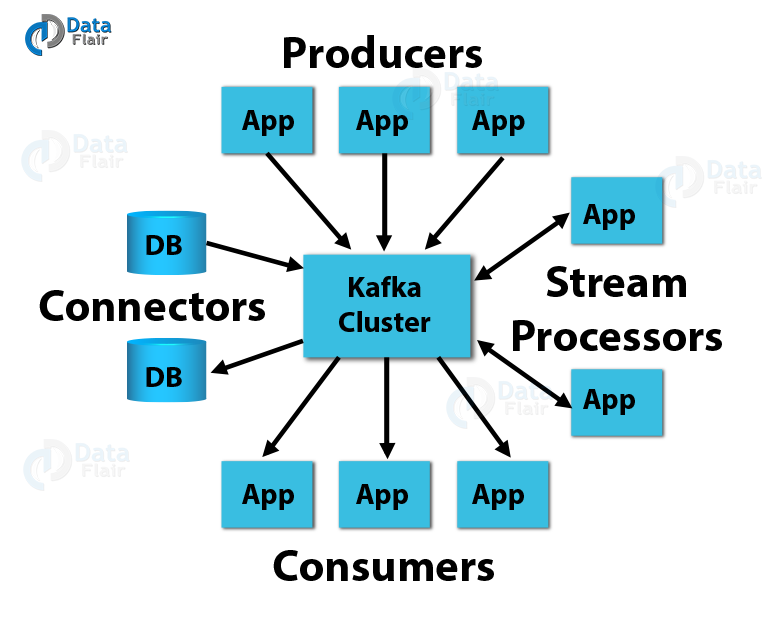
A Kafka Producer pushes the message into the message container called the Kafka Topic. Whereas a Kafka Consumer pulls the message from the Kafka Topic. There are two types of messaging patterns available, i.e. point to point and publish-subscribe (pub-sub) messaging system. However, most of the messaging patterns follow pub-sub.

**Point to Point Messaging System**

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.



1. **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.