**Apache Kafka**

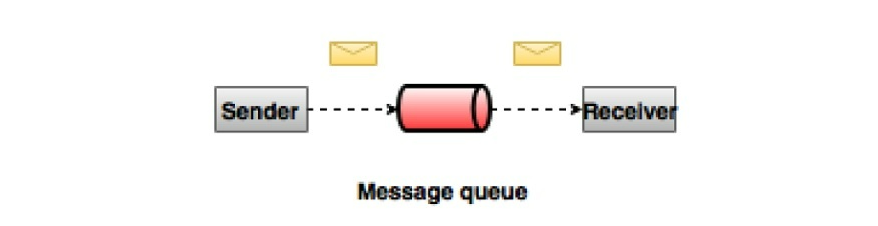
**1.. Messaging System**

In Big Data, an enormous volume of data is used. Regarding data, we have two main challenges. The first challenge is how to collect large volume of data and the second challenge is to analyze the collected data. To overcome those challenges, you must need a messaging system.

A Messaging System is responsible for transferring data from one application to another, so the applications can focus on data, but not worry about how to share it. Distributed messaging is based on the concept of reliable message queuing. Messages are queued asynchronously between client applications and messaging system. Two types of messaging patterns are available − one is point to point and the other is publish-subscribe (pub-sub) messaging system. Most of the messaging patterns follow pub-sub.

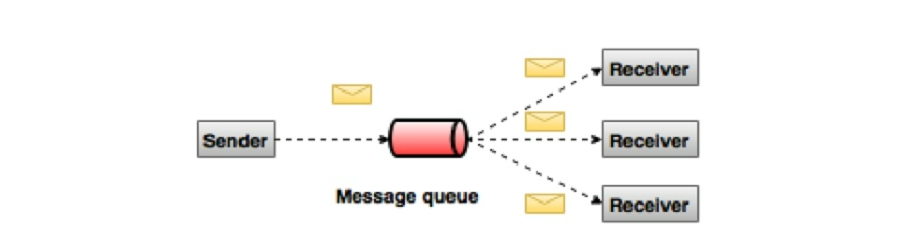
**1.1 Point to Point Messaging System**

In a point-to-point system, messages are persisted in a queue. One or more consumers can consume the messages in the queue, but a particular message can be consumed by a maximum of one consumer only. Once a consumer reads a message in the queue, it disappears from that queue. The typical example of this system is an Order Processing System, where each order will be processed by one Order Processor, but Multiple Order Processors can work as well at the same time. The following diagram depicts the structure.



**1.2 Publish-Subscribe Messaging System**

In the publish-subscribe system, messages are persisted in a topic. Unlike point-to-point system, consumers can subscribe to one or more topic and consume all the messages in that topic. In the Publish-Subscribe system, message producers are called publishers and message consumers are called subscribers. A real-life example is Dish TV, which publishes different channels like sports, movies, music, etc., and anyone can subscribe to their own set of channels and get them whenever their subscribed channels are available.



**2.. Apache Kafka**

Apache Kafka is Just like a Communication System which helps different parts of any web or any other application exchange data by publishing and subscribing to topics. Apache Kafka is an example of Publish-Subscribe Messaging System.

**2.1 Main Features of Apache Kafka**

* **Reliability** - Kafka is distributed, partitioned, replicated and fault tolerance.
* **Scalability** - Kafka messaging system scales easily without down time.
* **Durability** - Kafka uses Distributed commit log which means messages persists on disk as fast as possible, hence it is durable.
* **Performance** - Kafka has high throughput for both publishing and subscribing messages. It maintains stable performance even many TB of messages are stored.
* Kafka is very fast and guarantees zero downtime and zero data loss.

**2.2 Use Cases of Apache Kafka**

Kafka can be used in many Use Cases. Some of them are listed below –

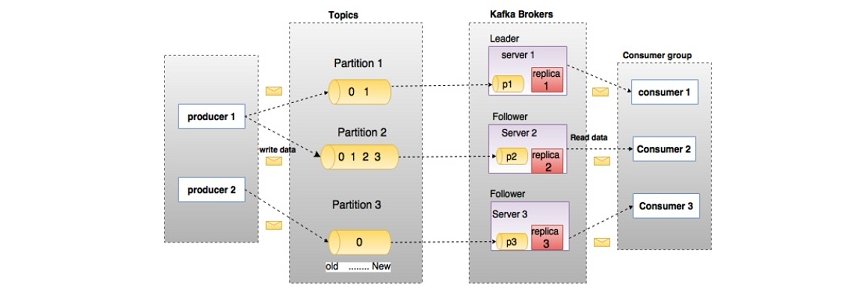
* **Metrics** − Kafka is often used for operational monitoring data. This involves aggregating statistics from distributed applications to produce centralized feeds of operational data.
* **Log Aggregation Solution** − Kafka can be used across an organization to collect logs from multiple services and make them available in a standard format to multiple consumers.
* **Stream Processing** − Popular frameworks such as Storm and Spark Streaming read data from a topic, processes it, and write processed data to a new topic where it becomes available for users and applications. Kafka’s strong durability is also very useful in the context of stream processing.
* **Live Location Tracking** – Kafka Can be used for Live Location Tracking of Delivery Boy in Application like Swiggy/Zomato.

**2.3 Need for Kafka**

Kafka is a unified platform for handling all the real-time data feeds. Kafka supports low latency message delivery and gives guarantee for fault tolerance in the presence of machine failures. It has the ability to handle a large number of diverse consumers. Kafka is very fast, performs 2 million writes/sec. Kafka persists all data to the disk, which essentially means that all the writes go to the page cache of the OS (RAM). This makes it very efficient to transfer data from page cache to a network socket.

**3.. Architecture of Apache Kafka**

Before moving deep into the Kafka, we must aware of the main terminologies such as topics, brokers, producers and consumers. The following diagram illustrates the main terminologies and the table describes the diagram components in detail.



In the above diagram, a topic is configured into three partitions. Partition 1 has two offset factors 0 and 1. Partition 2 has four offset factors 0, 1, 2, and 3. Partition 3 has one offset factor 0. The id of the replica is same as the id of the server that hosts it. Assume, if the replication factor of the topic is set to 3, then Kafka will create 3 identical replicas of each partition and place them in the cluster to make available for all its operations. To balance a load in cluster, each broker stores one or more of those partitions. Multiple producers and consumers can publish and retrieve messages at the same time.

**3.1 Topics**

A stream of messages belonging to a particular category is called a topic. Data is stored in topics. Topics are split into partitions. For each topic, Kafka keeps a minimum of one partition. Each such partition contains messages in an immutable ordered sequence. A partition is implemented as a set of segment files of equal sizes.

**3.2 Partition**

Topics may have many partitions, so it can handle an arbitrary amount of data.

**3.3 Partition offset**

Each partitioned message has a unique sequence id called as offset.

**3.4 Replicas of partition**

Replicas are nothing but backups of a partition. Replicas are never read or write data. They are used to prevent data loss.

**3.5 Brokers**

Brokers are simple system responsible for maintaining the published data. Each broker may have zero or more partitions per topic. Assume, if there are N partitions in a topic and N number of brokers, each broker will have one partition. Assume if there are N partitions in a topic and more than N brokers (n + m), the first N broker will have one partition and the next M broker will not have any partition for that particular topic. Assume if there are N partitions in a topic and less than N brokers (n-m), each broker will have one or more partition sharing among them. This scenario is not recommended due to unequal load distribution among the broker.

**3.6 Kafka Cluster**

Kafka’s having more than one broker are called as Kafka cluster. A Kafka cluster can be expanded without downtime. These clusters are used to manage the persistence and replication of message data.

**3.7 Producers**

Producers are the publisher of messages to one or more Kafka topics. Producers send data to Kafka brokers. Every time a producer publishes a message to a broker, the broker simply appends the message to the last segment file. Actually, the message will be appended to a partition. Producer can also send messages to a partition of their choice.

**3.8 Consumers**

Consumers read data from brokers. Consumers subscribes to one or more topics and consume published messages by pulling data from the brokers.

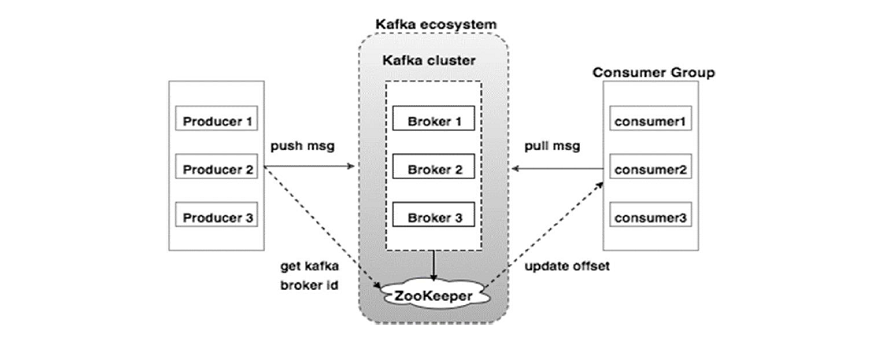
**3.9 Leader**

Leader is the node responsible for all reads and writes for the given partition. Every partition has one server acting as a leader.

**3.10 Follower**

Node which follows leader instructions are called as follower. If the leader fails, one of the followers will automatically become the new leader. A follower acts as normal consumer, pulls messages and up-dates its own data store.

**4.. Kafka Cluster Architecture**



**4.1 Broker**

Kafka cluster typically consists of multiple brokers to maintain load balance. Kafka brokers are stateless, so they use Zookeeper for maintaining their cluster state. One Kafka broker instance can handle hundreds of thousands of reads and writes per second and each broker can handle TB of messages without performance impact. Kafka broker leader election can be done by Zookeeper.

**4.2 Zookeeper**

Zookeeper is used for managing and coordinating Kafka broker. Zookeeper service is mainly used to notify producer and consumer about the presence of any new broker in the Kafka system or failure of the broker in the Kafka system. As per the notification received by the Zookeeper regarding presence or failure of the broker then producer and consumer takes decision and starts coordinating their task with some other broker.

**4.3 Producers**

Producers push data to brokers. When the new broker is started, all the producers search it and automatically sends a message to that new broker. Kafka producer doesn’t wait for acknowledgements from the broker and sends messages as fast as the broker can handle.

**4.4 Consumers**

Since Kafka brokers are stateless, which means that the consumer has to maintain how many messages have been consumed by using partition offset. If the consumer acknowledges a particular message offset, it implies that the consumer has consumed all prior messages. The consumer issues an asynchronous pull request to the broker to have a buffer of bytes ready to consume. The consumers can rewind or skip to any point in a partition simply by supplying an offset value. Consumer offset value is notified by Zookeeper.

**5.. Workflow of Apache 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.

**5.1 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.

**5.2 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 consumers reach the number of partitions configured for that particular topic.
* Once the number of consumers exceeds the number of partitions, the new consumer will not receive any further message until any one of the existing consumers 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.

**5.3 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.

**6.. Installation**

* Download Apache Kafka of Version which we want from

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

Recommended is to download the Apache Kafka of Latest Version.

* Zip File Will be downloaded. Extract the Zip files into some folder.

**7.. Starting and Stopping Zookeeper Server**

To Start the Zookeeper, run the below command on CMD.

**bin\zookeeper-server-start.bat config\zookeeper.properties**

To Stop the Zookeeper, run the below command on CMD:

**bin\zookeeper-server-stop.bat**

**8.. Starting and Stopping Kafka Server**

To Start the Kafka Server, run the below command on CMD.

**bin\kafka-server-start.bat config\server.properties**

To Stop the Kafka Server, run the below command on CMD:

**bin\kafka-server-stop.bat**

If we want to change any configuration of Kafka Server and Zookeeper we can do so in server.properties and zookeeper.properties file.

**9.. Kafka Commands**

**9.1 Creating a Kafka Topic**

Kafka provides a command line utility named kafka-topics.bat to create topics on the server. Open new terminal on CMD and type the below example.

bin/kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1

--partitions 1 --topic Hello-Kafka

We just created a topic named Hello-Kafka with a single partition and one replica factor. The above created output will be similar to the following output

Output − Created topic Hello-Kafka

Once the topic has been created, you can get the notification in Kafka broker terminal window and the log for the created topic specified in “/tmp/kafka-logs/“ in the config/server.properties file.

**9.2 List Out Kafka Topics**

To get a list of topics in Kafka server, you can use the following command –

bin/kafka-topics.sh --list --zookeeper localhost:2181

Output

Hello-Kafka

Since we have created a topic, it will list out Hello-Kafka only. Suppose, if you create more than one topic, you will get the topic names in the output.

**9.3 Producing Messages By Producer**

**Syntax**:

bin/kafka-console-producer.bat --broker-list localhost:9092 --topic topic-name

From the above syntax, two main parameters are required for the producer command line client:

**Broker-list** − The list of brokers that we want to send the messages to. In this case we only have one broker. The Config/server.properties file contains broker port id, since we know our broker is listening on port 9092, so you can specify it directly.

**Topic name** − Here is an example for the topic name.

bin/kafka-console-producer.bat --broker-list localhost:9092 --topic Hello-Kafka

The producer will wait on input from stdin and publishes to the Kafka cluster. By default, every new line is published as a new message then the default producer properties are specified in config/producer.properties file. Now you can type a few lines of messages in the terminal as shown below.

**9.4 Consuming Messages By Consumer**

Similar to producer, the default consumer properties are specified in config/consumer.properties file. Open a new terminal and type the below syntax for consuming messages.

Syntax:

bin/kafka-console-consumer.bat --zookeeper localhost:2181 —topic topic-name --from-beginning

Example:

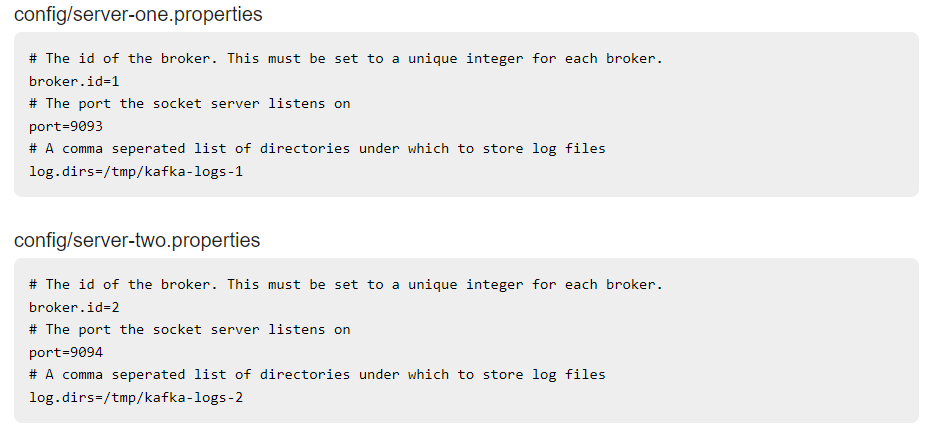
bin/kafka-console-consumer.bat --zookeeper localhost:2181 —topic Hello-Kafka --from-beginning

Finally, We are able to enter messages from the producer’s terminal and see them appearing in the consumer’s terminal. As of now, We have a very good understanding on the single node cluster with a single broker. Let us now move on to the multiple brokers configuration.

**Single Node Multiple Brokers Configuration**

Before moving on to the multiple brokers cluster setup, first start Zookeeper server.

**Create Multiple Kafka Brokers** − We have one Kafka broker instance already in con-fig/server.properties. Now we need multiple broker instances, so copy the existing server.properties file into two new config files and rename it as server-one.properties and server-two.properties. Then edit both new files and assign the following changes.



**Start Multiple Brokers**− After all the changes have been made on three servers then open three new terminals to start each broker one by one.

Broker1

bin/kafka-server-start.bat config/server.properties

Broker2

bin/kafka-server-start.bat config/server-one.properties

Broker3

bin/kafka-server-start.bat config/server-two.properties

Now we have three different brokers running on the machine.

**Creating a Topic**

Let us assign the replication factor value as three for this topic because we have three different brokers running. If We have two brokers, then the assigned replica value will be two.

Syntax:

bin/kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 3 -partitions 1 --topic topic-name

Example:

bin/kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 3 -partitions 1 --topic Multibrokerapplication

The Describe command is used to check which broker is listening on the current created topic as shown below –

bin/kafka-topics.bat --describe --zookeeper localhost:2181 --topic Multibrokerapplication

Producing and Consuming Messages to/from Topic Remains same as we did in Single Broker Communication.

**Modifying a Topic**

Syntax

bin/kafka-topics.sh —zookeeper localhost:2181 --alter --topic topic\_name --partitions count

Example

We have already created a topic “Hello-Kafka” with single partition count and one replica factor.

Now using “alter” command we have changed the partition count.

bin/kafka-topics.sh --zookeeper localhost:2181 --alter --topic Hello-Kafka --partitions 2

**Deleting a Topic**

Syntax

bin/kafka-topics.sh --zookeeper localhost:2181 --delete --topic topic\_name

Example

bin/kafka-topics.sh --zookeeper localhost:2181 --delete --topic Hello-Kafka

Output

> Topic Hello-Kafka marked for deletion

Note −This will have no impact if delete.topic.enable is not set to true.

**Simple Producer Consumer Application**

Let us create an application for publishing and consuming messages using a Java client. Kafka producer client consists of the following API’s.

**KafkaProducer API**

Let us understand the most important set of Kafka producer API in this section. The central part of the KafkaProducer API is KafkaProducer class. The KafkaProducer class provides an option to connect a Kafka broker in its constructor with the following methods.

KafkaProducer class provides send method to send messages asynchronously to a topic.

The signature of send() is as follows

producer.send(new ProducerRecord<byte[],byte[]>(topic, partition, key1, value1) , callback);

**ProducerRecord** − The producer manages a buffer of records waiting to be sent.

**Callback** − A user-supplied callback to execute when the record has been acknowl-edged by the server (null indicates no callback).

KafkaProducer class provides a flush method to ensure all previously sent messages have been actually completed. Syntax of the flush method is as follows −

public void flush()

KafkaProducer class provides partitionFor method, which helps in getting the partition metadata for a given topic. This can be used for custom partitioning. The signature of this method is as follows −

public Map metrics()

It returns the map of internal metrics maintained by the producer.

public void close() − KafkaProducer class provides close method blocks until all previously sent requests are completed.

**Producer API**

The central part of the Producer API is Producer class. Producer class provides an option to connect Kafka broker in its constructor by the following methods.

**The Producer Class**

The producer class provides send method to send messages to either single or multiple topics using the following signatures.

public void send(KeyedMessaget<k,v> message)

- sends the data to a single topic, partitioned by key using either sync or async producer.

public void send(List<KeyedMessage<k,v>>messages)

- sends data to multiple topics.

Properties prop = new Properties();

prop.put(producer.type,”async”)

ProducerConfig config = new ProducerConfig(prop);

There are two types of producers – Sync and Async.

The same API configuration applies to Sync producer as well. The difference between them is a sync producer sends messages directly, but sends messages in background. Async producer is preferred when you want a higher throughput. In the previous releases like 0.8, an async producer does not have a callback for send() to register error handlers. This is available only in the current release of 0.9.

public void close()

Producer class provides close method to close the producer pool connections to all Kafka brokers.