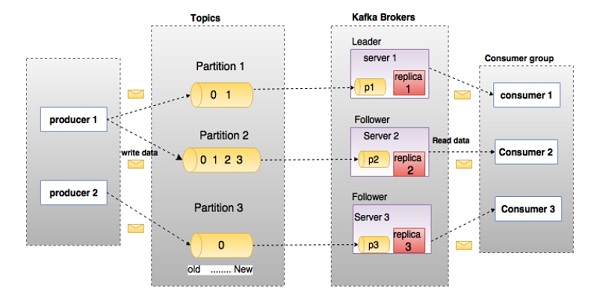
# Kafka

**Intro**

* Kafka is designed for distributed high throughput systems and has better throughput, built-in partitioning, replication and inherent fault-tolerance
* 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.
* Kafka is suitable for both offline and online message consumption. Kafka messages are persisted on the disk and replicated within the cluster to prevent data loss.
* Kafka is built on top of the ZooKeeper synchronization service. It integrates very well with Apache Storm and Spark for real-time streaming data analysis.
* Kafka supports low latency message delivery and gives guarantee for fault tolerance in the presence of machine failures.



|  |  |
| --- | --- |
| **S.No** | **Components and Description** |
| 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 mini-mum 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. |
| 2 | **Partition**  Topics may have many partitions, so it can handle an arbitrary amount of data. |
| 3 | **Partition offset**  Each partitioned message has a unique sequence id called as offset. |
| 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. |
| 5 | **Brokers**   * Brokers are simple system responsible for maintaining the pub-lished 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 distri-bution among the broker. |
| 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. |
| 7 | **Producers**  Producers are the publisher of messages to one or more Kafka topics. Producers send data to Kafka brokers. Every time a producer pub-lishes 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. |
| 8 | **Consumers**  Consumers read data from brokers. Consumers subscribes to one or more topics and consume published messages by pulling data from the brokers. |
| 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. |
| 10 | **Follower**  Node which follows leader instructions are called as follower. If the leader fails, one of the follower will automatically become the new leader. A follower acts as normal consumer, pulls messages and up-dates its own data store. |
| 11 | **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 pro-ducer and consumer takes decision and starts coordinating their task with some other broker. |

**KafkaProducer API**

* The central part of the KafkaProducer API is KafkaProducer class.
* 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).
* The central part of the Producer API is Producer class.
* ProducerRecord is a key/value pair that is sent to Kafka cluster.ProducerRecord class constructor for creating a record with partition, key and value pairs using the following signature.

public ProducerRecord (string topic, int partition, k key, v value)

public ProducerRecord (string topic, k key, v value)

public ProducerRecord (string topic, v value)

Example

kafkaTemplate.send(KafkaProperties.getKafkaDgaTopicOut(), message);

**KafkaConsumer API**

* KafkaConsumer API is used to consume messages from the Kafka cluster
* The ConsumerRecord API is used to receive records from the Kafka cluster. This API consists of a topic name, partition number, from which the record is being received and an offset that points to the record in a Kafka partition

public ConsumerRecord(string topic,int partition, long offset,K key, V value)

* ConsumerRecords API acts as a container for ConsumerRecord. This API is used to keep the list of ConsumerRecord per partition for a particular topic

public ConsumerRecords(java.util.Map<TopicPartition,java.util.List

<Consumer-Record>K,V>>> records)