[**https://**kafka**.apache.org/documentation.html**](https://kafka.apache.org/documentation.html)

**https://www.projectpro.io/article/apache-kafka-architecture-/442**

[Java Message Service](https://en.wikipedia.org/wiki/Java_Message_Service%22%20/t%20%22_blank) is an API that supports the formal communication called **messaging between computers** on a network. JMS provides a common interface for standard message protocols and message services in support of the Java programs.

JMS provides the **facility to create, send and read messages**. The JMS API reduces the concepts that a programmer must learn to use the messaging services/products and also provides the features that support the messaging applications.

JMS helps in building the communication between two or more applications in a loosely coupled manner. It means that the applications which have to communicate are not connected directly they are connected through a common destination.

**1. Why We Need JMS?**

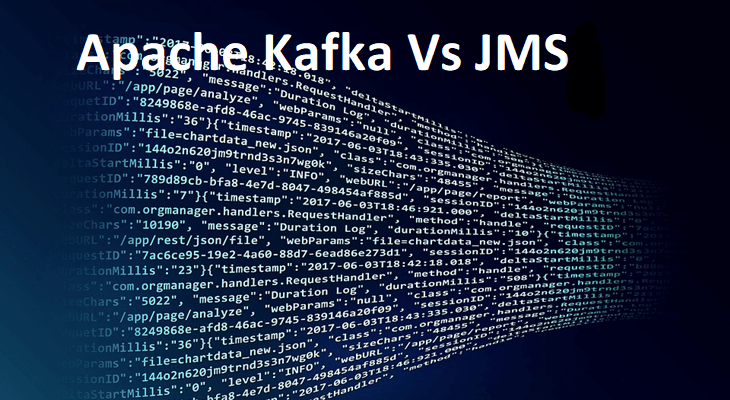
When one application wants to send a message to another application in such a way that both applications do not know anything about each other; and even they may NOT be deployed on the same server.

For example, one application A is running in India and another application B is running in the USA, and A is interested in sending some updates/messages to B – whenever something unique happens on A. There may be N number of such applications that are interested in such updates to B.

In this scenario, Java provides one of the solutions in form of JMS – and solves the exact same problem discussed above.

JMS is also useful when we are writing any event-based application like a [chat server](https://howtodoinjava.com/java/library/java-aiml-chatbot-example/) where it needs a publish event mechanism to send messages between the server and the clients.

Note that JMS is different from RMI so there is no need for the destination object to be available online while sending a message. The publisher publishes the message and forgets it, whenever the receiver comes online, it will fetch the message. It’s a very powerful solution for very common problems in today’s world.



[Apache Kafka vs. JMS: Difference Explained](https://www.knowledgenile.com/blogs/apache-kafka-vs-jms/)

Apache Kafka is a pub-sub tool that is commonly used for message processing, scaling, and handling a huge amount of data efficiently.

Whereas Java Message Service aka JMS is a message service that is designed for more complicated systems such as Enterprise Integration Patterns.

Both Apache Kafka and JMS-based services have been pretty successful and helped the organizations to effectively communicate through servers to the internal teams and customers alike.

Our primary focus in this blog is to understand how these tools differ from each other in detail.

**Key Differentiating Factors Between Apache Kafka and JMS**

1. [**Style of Programming**](https://www.knowledgenile.com/blogs/apache-kafka-vs-jms/#ProgrammingStyle)
2. [**Segregating the content**](https://www.knowledgenile.com/blogs/apache-kafka-vs-jms/#SegregatingContent)
3. [**Type of Message Programming**](https://www.knowledgenile.com/blogs/apache-kafka-vs-jms/#MessageProgramming)
4. [**Filter Method**](https://www.knowledgenile.com/blogs/apache-kafka-vs-jms/#FilterMethod)
5. [**Routing System**](https://www.knowledgenile.com/blogs/apache-kafka-vs-jms/#RoutingSystem)
6. [**Storage**](https://www.knowledgenile.com/blogs/apache-kafka-vs-jms/#Storage)
7. [**Queuing**](https://www.knowledgenile.com/blogs/apache-kafka-vs-jms/#Queuing)
8. [**Partitioning of topics**](https://www.knowledgenile.com/blogs/apache-kafka-vs-jms/#TopicPartitioning)
9. [**Message Logs**](https://www.knowledgenile.com/blogs/apache-kafka-vs-jms/#MessageLogs)
10. [**Scalability and Availability**](https://www.knowledgenile.com/blogs/apache-kafka-vs-jms/#Scalability&Availability)

Apache Kafka vs. JMS: Programming Style

The key differentiator between Apache Kafka and JMS is their programming styles. Apache Kafka is a reactive programming style while on the other hand, JMS is an imperative programming style.

Apache Kafka vs. JMS: Content Segregation

In the case of Apache Kafka, its system sorts the messages in the same order as they were sent from the partition level. But, in the case of JMS, such provision is not present, so you need to compartmentalize the messages as per the requirement.

Apache Kafka vs. JMS: Message Programming Type

Another factor which proves to be a key differentiator between Apache Kafka and JMS is the type of the messages.

Apache Kafka is a pull-type messaging platform where consumers pull the messages from the broker while JMS-based services are of push-type in nature where the providers push the messages to the consumers.

**Also Read:**[**SQL Vs. NoSQL Vs. NewSQL: What’s The Difference?**](https://www.knowledgenile.com/blogs/sql-vs-nosql-vs-newsql/)

Apache Kafka vs. JMS: Filter Method

In Apache Kafka, you cannot set the filters for exact terms at the broker level. If you want to set up the filters, you need to work that at the application level in Apache Kafka.

But, on the other hand, you can set the desired filters through JMS Message selectors. This functionality reduces an additional step of application-level filtering.

Apache Kafka vs. JMS: Routing System

Apache Kafka allows you to have a simplified and easy routing system; while JMS has a little complicated routing system due to its system design.

Apache Kafka vs. JMS: Storage

Messages are stored for a defined amount of time in Apache Kafka irrespective of whether they are received by the consumers or not.

And on the other hand, JMS provides the disk or in-memory based storage facility. And once the message read, it gets permanently deleted.

Apache Kafka vs. JMS: Queuing

Apache Kafka doesn’t allow you the queuing facility. The only way to send the messages is through the pub-sub model. But with JMS-based systems, you can queue up the messages through its routing system.

Apache Kafka vs. JMS: Partitioning of Topics

Apache Kafka allows you the functionality to segregate the topics as independent portioned logs. It ensures a high throughput for Kafka.

But, in the case of JMS-based tools, the segregation is not done in a sequential manner. This leads to lower throughput in the case of JMS-based tools.

Apache Kafka vs. JMS: Message Logs

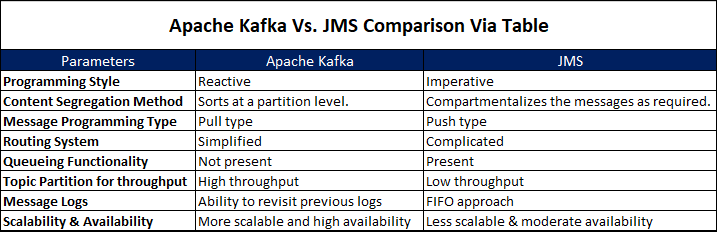
Apache Kafka implements a system that allows the brokers to determine which message to read first unlike in the case of JMS.

JMS uses First in First Out approach due to the queuing functionality. The ability to revisit and choice of reading is an important benefit of Apache Kafka, which gives it an edge over JMS.

Apache Kafka vs. JMS: Scalability and Availability

It is evident from the above-mentioned differences that Apache Kafka is more scalable in nature as compared to the JMS.

Also, the ability to auto-replicate the messages without compromising the simplicity results in the higher availability for Apache Kafka.



**So Which One is Better: Apache Kafka or JMS?**

Apache Kafka and JMS both are efficient tools, the key thing is to understand that the circumstances which help one of them to perform better than the other.

Apache Kafka is more suitable to handle a large volume of data due to its scalability and high availability while JMS systems are used when you need to work with multi-node clusters and highly complicated systems.

Also, Apache Kafka is used when there is a requirement of higher throughput (more than 100K/sec), and JMS is used when you need to work with the low throughputs.

JMS bases tools provide an HTTP API, CLI based operators that give JMS systems a faster deployment and ease of operation.

Apache Kafka uses partition-based operation with CLI and is more useful in the case of cases that require multiple changes in a short span.

Kafka is a distributed message streaming platform that uses publish subscriber mechanism to stream the records.

Messaging system is responsible for transferring data from one application to another so the application can focus on data without getting bogged down on data transmission and sharing.

Two types :-

1. Point to Point Messaging System.

* Messages are persisted in a queue.
* A particular message can be consumed by a maximum of one receiver only.
* There is no time dependency laid for the receiver to receive the messages.
* When the receiver receives the message, it will send an acknowledgement back to the Sender.

1. Publish Subscribe Messaging System.

* Messages are persisted in a topic.
* A particular message can be consumed by any number of consumers.
* There is time dependency laid for the consumer to consume the messages.
* When the subscriber receives the message, it does not send an acknowledgement back to the publisher.



Kafka Terminologies :-

* 1. Topics :- A stream of messages belonging to particular category is called a topic. It is a logical feed name to which records are published. Similar to a table in a database.

Unique identifier of a topic is it’s NAME. We can create as many as topics we want.

Topics are split into partitions. All the messages within a partition are ordered and immutable. Each messages within a partition has unique ID Associated known as Offset.

Replica are backup’s of a partition. Replica are never read and write data. They are used to prevent data loss.

A stream of messages that are a part of a specific category or feed name is referred to as a Kafka topic. In Kafka, data is stored in the form of topics. Producers write their data to topics, and consumers read the data from these topics.

* 1. Producer :- Producer are applications which write/publish data to the topics within a cluster using the Producing API’s.

Producers can write data either on topic level or specific partitions of the topics.

* 1. Consumers :- Consumers are applications which read/consume data from the topics

Within a cluster using the Consuming API’s. Consumer can read data either on the topic level or specific partitions of the topics.

Consumer are always associated with exactly one Consumer Group.

A Consumer Group is a group of related consumers that perform task.

* 1. Broker :- Brokers are simple software processes who maintain and manage the published messages. Also known as kafka servers. Brokers also manage the consumer offsets and are responsible for the delivery of messages to the right consumers. A set of brokers who are communicating with each other to perform the management and maintenance task are collectively knows as Kafka Cluster. We can add a more broker in already running kafka cluster without any downtime.

**Brokers**

A Kafka cluster comprises one or more servers that are known as brokers. In Kafka, a broker works as a container that can hold multiple topics with different partitions. A unique integer ID is used to identify brokers in the Kafka cluster. Connection with any one of the kafka brokers in the cluster implies a connection with the whole cluster. If there is more than one broker in a cluster, the brokers need not contain the complete data associated with a particular topic.

## ****Why is Apache Kafka so popular?****

So why is [Kafka](https://www.projectpro.io/article/kafka-vs-rabbitmq/451) so popular? And what makes it such a popular choice for companies?

* Scalability: The scalability of a system is determined by how well it can maintain its performance when exposed to changes in application and processing demands. Apache Kafka has a distributed architecture capable of handling incoming messages with higher volume and velocity. As a result, Kafka is highly scalable without any downtime impact.
* High Throughput: Apache Kafka is able to handle thousands of messages per second. Messages coming in at a high volume or a high velocity or both will not affect the performance of Kafka.
* Low Latency: Latency refers to the amount of time taken for a system to process a single event. Kafka offers a very low latency, which is as low as ten milliseconds.
* Fault Tolerance: By using replication, Kafka can handle failures at nodes in a cluster without any data loss. Running processes, too, can remain undisturbed. The replication factor determines the number of replicas for a partition. For a replication factor of ‘n,’ Kafka guarantees a fault tolerance for up to n-1 servers in the Kafka cluster.
* Reliability: Apache Kafka is a distributed platform with very high fault tolerance, making it a very reliable system to use.
* Durability: Data present on the Kafka cluster is allowed to remain persistent more on the cluster than on the disk. This ensures that Kafka’s data remains durable.

Ability to handle real-time data: Kafka supports real-time data handling and is an excellent choice when data has to be processed in real-time.

**New Projects**

[**Build Serverless Pipeline using AWS CDK and Lambda in Python**View Project](https://www.projectpro.io/project-use-case/aws-cdk-lambda-pipeline-example?utm_source=442&utm_medium=fold2)

[**dbt Snowflake Project to Master dbt Fundamentals in Snowflake**View Project](https://www.projectpro.io/project-use-case/dbt-snowflake-project-to-master-dbt-in-snowflake?utm_source=442&utm_medium=fold2)

[**AWS CDK Project for Building Real-Time IoT Infrastructure**View Project](https://www.projectpro.io/project-use-case/aws-cdk-project-example-for-building-real-time-iot-infrastructure?utm_source=442&utm_medium=fold2)

[**EMR Serverless Example to Build a Search Engine for COVID19**View Project](https://www.projectpro.io/project-use-case/emr-serverless-example-to-build-a-search-engine?utm_source=442&utm_medium=fold2)

[**AWS CDK and IoT Core for Migrating IoT-Based Data to AWS**View Project](https://www.projectpro.io/project-use-case/aws-cdk-project-for-migrating-iot-based-data?utm_source=442&utm_medium=fold2)

[**Learn to Create Delta Live Tables in Azure Databricks**View Project](https://www.projectpro.io/project-use-case/how-to-create-delta-live-tables-in-azure-databricks?utm_source=442&utm_medium=fold2)

[**Build a Data Pipeline with Azure Synapse and Spark Pool**View Project](https://www.projectpro.io/project-use-case/build-a-data-pipeline-with-azure-synapse-analytics-and-spark-pool?utm_source=442&utm_medium=fold2)

[**Multilabel Classification Project for Predicting Shipment Modes**View Project](https://www.projectpro.io/project-use-case/multilabel-classification-project-example?utm_source=442&utm_medium=fold2)

[**Build an ETL Pipeline on EMR using AWS CDK and Power BI**View Project](https://www.projectpro.io/project-use-case/etl-pipeline-project-on-aws-emr-using-cdk-and-powerbi?utm_source=442&utm_medium=fold2)

[**Multilabel Classification Project for Predicting Shipment Modes**View Project](https://www.projectpro.io/project-use-case/multilabel-classification-project-example?utm_source=442&utm_medium=fold2)

[**Build Serverless Pipeline using AWS CDK and Lambda in Python**View Project](https://www.projectpro.io/project-use-case/aws-cdk-lambda-pipeline-example?utm_source=442&utm_medium=fold2)

[**dbt Snowflake Project to Master dbt Fundamentals in Snowflake**View Project](https://www.projectpro.io/project-use-case/dbt-snowflake-project-to-master-dbt-in-snowflake?utm_source=442&utm_medium=fold2)

[**AWS CDK Project for Building Real-Time IoT Infrastructure**View Project](https://www.projectpro.io/project-use-case/aws-cdk-project-example-for-building-real-time-iot-infrastructure?utm_source=442&utm_medium=fold2)

[**EMR Serverless Example to Build a Search Engine for COVID19**View Project](https://www.projectpro.io/project-use-case/emr-serverless-example-to-build-a-search-engine?utm_source=442&utm_medium=fold2)

[**AWS CDK and IoT Core for Migrating IoT-Based Data to AWS**View Project](https://www.projectpro.io/project-use-case/aws-cdk-project-for-migrating-iot-based-data?utm_source=442&utm_medium=fold2)

[**Learn to Create Delta Live Tables in Azure Databricks**View Project](https://www.projectpro.io/project-use-case/how-to-create-delta-live-tables-in-azure-databricks?utm_source=442&utm_medium=fold2)

[**Build a Data Pipeline with Azure Synapse and Spark Pool**View Project](https://www.projectpro.io/project-use-case/build-a-data-pipeline-with-azure-synapse-analytics-and-spark-pool?utm_source=442&utm_medium=fold2)

[**Multilabel Classification Project for Predicting Shipment Modes**View Project](https://www.projectpro.io/project-use-case/multilabel-classification-project-example?utm_source=442&utm_medium=fold2)

[**Build an ETL Pipeline on EMR using AWS CDK and Power BI**View Project](https://www.projectpro.io/project-use-case/etl-pipeline-project-on-aws-emr-using-cdk-and-powerbi?utm_source=442&utm_medium=fold2)

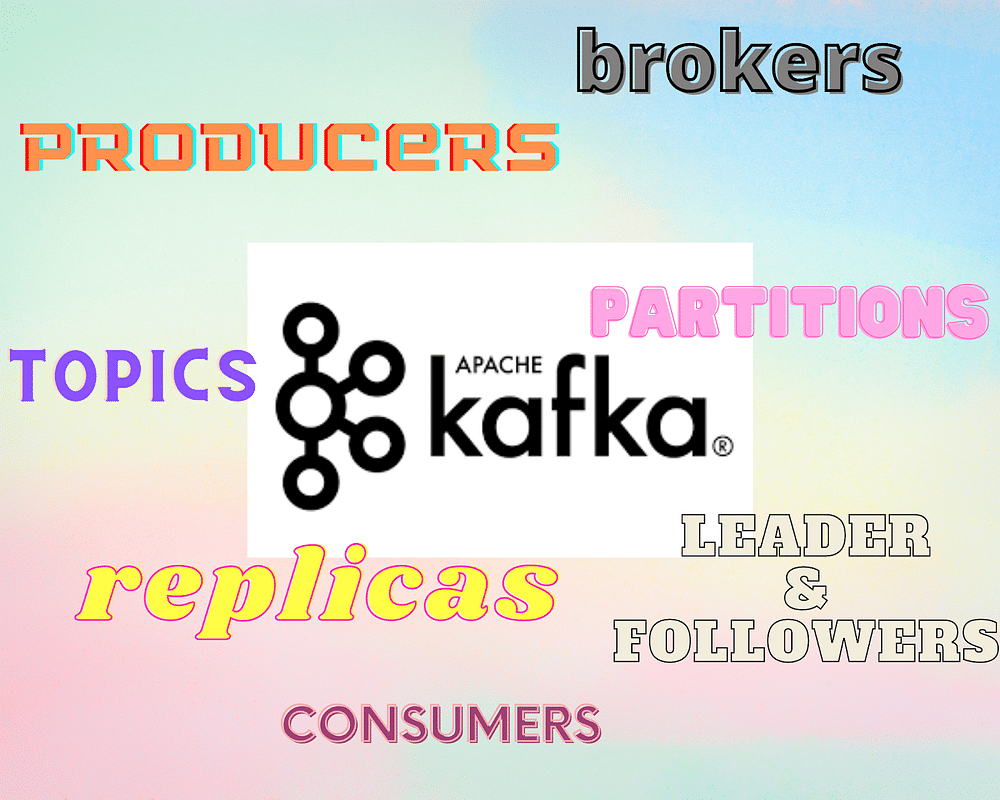
[**Multilabel Classification Project for Predicting Shipment Modes**View Project](https://www.projectpro.io/project-use-case/multilabel-classification-project-example?utm_source=442&utm_medium=fold2)

[View all New Projects](https://www.projectpro.io/projects?utm_source=442&utm_medium=fold2)

## ****Apache Kafka Architecture - Overview of Kafka Components****

Let’s look in detail at the architecture of Apache Kafka and the relationship between the various architectural components to develop a deeper understanding of Kafka for distributed streaming.

**Components of Apache Kafka and Its Architectural**Concepts



**Topics**

A stream of messages that are a part of a specific category or feed name is referred to as a Kafka topic. In Kafka, data is stored in the form of topics. Producers write their data to topics, and consumers read the data from these topics.

**Brokers**

A Kafka cluster comprises one or more servers that are known as brokers. In Kafka, a broker works as a container that can hold multiple topics with different partitions. A unique integer ID is used to identify brokers in the Kafka cluster. Connection with any one of the kafka brokers in the cluster implies a connection with the whole cluster. If there is more than one broker in a cluster, the brokers need not contain the complete data associated with a particular topic.

**Consumers and Consumer Groups**

Consumers read data from the Kafka cluster. The data to be read by the consumers has to be pulled from the broker when the consumer is ready to receive the message. A consumer group in Kafka refers to a number of consumers that pull data from the same topic or same set of topics.

**Ace Your Next Job Interview with Mock Interviews from Experts to Improve Your Skills and Boost Confidence!**

**[](https://www.projectpro.io/resources/data-science-interview-preparation)**

**Producers**

Producers in Kafka publish messages to one or more topics. They send data to the Kafka cluster. Whenever a Kafka producer publishes a message to Kafka, the broker receives the message and appends it to a particular partition. Producers are given a choice to publish messages to a partition of their choice.

**Partitions**

Topics in Kafka are divided into a configurable number of parts, which are known as partitions. Partitions allow several consumers to read data from a particular topic in parallel. Partitions are separated in order. The number of partitions is specified when configuring a topic, but this number can be changed later on. The partitions comprising a topic are distributed across servers in the Kafka cluster. Each server in the cluster handles the data and requests for its share of partitions. Messages are sent to the broker along with a key. The key can be used to determine which partition that particular message will go to. All messages which have the same key go to the same partition. If the key is not specified, then the partition will be decided in a round-robin fashion.

**Partition Offset**

Messages or records in Kafka are assigned to a partition. To specify the position of the records within the partition, each record is provided with an offset. A record can be uniquely identified within its partition using the offset value associated with it. A partition offset carries meaning only within that particular partition. Older records will have lower offset values since records are added to the ends of partitions.

**Replicas**

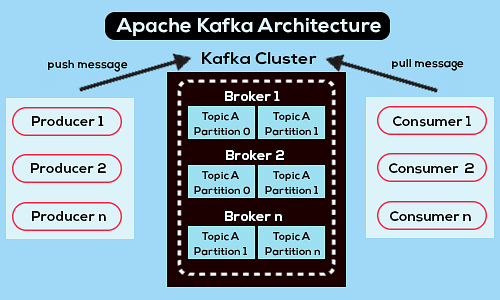
Replicas are like backups for partitions in Kafka. They are used to ensure that there is no data loss in the event of a failure or a planned shutdown. Partitions of a topic are published across multiple servers in a Kafka cluster. Copies of the partition are known as Replicas.

**Leader and Follower**

Every partition in Kafka will have one server that plays the role of a leader for that particular partition. The leader is responsible for performing all the read and write tasks for the partition. Each partition can have zero or more followers. The duty of the follower is to replicate the data of the leader. In the event of a failure in the leader for a particular partition, one of the follower nodes can take on the role of the leader.

**Get FREE Access to**[**Data Analytics Example Codes for Data Cleaning, Data Munging, and Data Visualization**](https://www.projectpro.io/recipes?utm_source=Blg442&utm_medium=RcpLink&utm_campaign=TXTCTA1)

## ****Apache Kafka Event-Driven Workflow Orchestration****



**Kafka Producers**

In Kafka, the producers send data directly to the broker that plays the role of leader for a given partition. In order to help the producer send the messages directly, the nodes of the Kafka cluster answer requests for metadata on which servers are alive and the current status of the leaders of partitions of a topic so that the producer can direct its requests accordingly. The client decides which partition it publishes its messages to. This can either be done arbitrarily or by making use of a partitioning key, where all messages containing the same partition key will be sent to the same partition.

Messages in Kafka are sent in the form of batches, known as record batches. The producers accumulate messages in memory and send them in batches either after a fixed number of messages are accumulated or before a fixed latency bound period of time has elapsed.

**Explore Categories**

[**Apache Hive Projects**](https://www.projectpro.io/projects/big-data-projects/apache-hive-projects?utm_source=442&utm_medium=fold3) [**Apache Hbase Projects**](https://www.projectpro.io/projects/big-data-projects/apache-hbase-projects?utm_source=442&utm_medium=fold3) [**Apache Pig Projects**](https://www.projectpro.io/projects/big-data-projects/apache-pig-projects?utm_source=442&utm_medium=fold3) [**Apache Oozie Projects**](https://www.projectpro.io/projects/big-data-projects/apache-oozie-projects?utm_source=442&utm_medium=fold3) [**Apache Impala Projects**](https://www.projectpro.io/projects/big-data-projects/apache-impala-projects?utm_source=442&utm_medium=fold3) [**Apache Flume Projects**](https://www.projectpro.io/projects/big-data-projects/apache-flume-projects?utm_source=442&utm_medium=fold3) [**Apache Sqoop Projects**](https://www.projectpro.io/projects/big-data-projects/apache-sqoop-projects?utm_source=442&utm_medium=fold3) [**Spark SQL Projects**](https://www.projectpro.io/projects/big-data-projects/spark-sql-projects?utm_source=442&utm_medium=fold3) [**Spark GraphX Projects**](https://www.projectpro.io/projects/big-data-projects/spark-graphx-projects?utm_source=442&utm_medium=fold3) [**Spark MLlib Projects**](https://www.projectpro.io/projects/big-data-projects/spark-mllib-projects?utm_source=442&utm_medium=fold3) [**Apache Zepellin Projects**](https://www.projectpro.io/projects/big-data-projects/apache-zepellin-projects?utm_source=442&utm_medium=fold3) [**Neo4j Projects**](https://www.projectpro.io/projects/big-data-projects/neo4j-projects?utm_source=442&utm_medium=fold3) [**Redis Projects**](https://www.projectpro.io/projects/big-data-projects/redis-projects?utm_source=442&utm_medium=fold3) [**Microsoft Azure Projects**](https://www.projectpro.io/projects/big-data-projects/microsoft-azure-projects?utm_source=442&utm_medium=fold3) [**Google Cloud Projects GCP**](https://www.projectpro.io/projects/big-data-projects/google-cloud-projects-gcp?utm_source=442&utm_medium=fold3) [**AWS Projects**](https://www.projectpro.io/projects/big-data-projects/aws-projects?utm_source=442&utm_medium=fold3)

**Kafka Brokers**

In Kafka, the cluster usually contains multiple nodes, that are known as brokers, to maintain the load balance. The brokers are stateless, and hence their cluster state is maintained by the ZooKeeper. One Kafka broker is able to handle hundreds of thousands of reads and writes per second. For one particular partition, one broker serves as the leader. The leader may have one or more followers, where the data on the leader is to be replicated across the followers for that particular partition. The role of leader for partitions is distributed across brokers in the cluster.

The nodes in a cluster have to send messages called Heartbeat messages to the ZooKeeper to keep the ZooKeeper informed that they are alive. The followers have to stay caught up with the data that is in the leader. The leader keeps track of the followers that are “in sync” with it. If a follower is no longer alive or does not stay caught up with the leader, it is removed from the list of in-sync replicas (ISRs) associated with that particular leader. If the leader dies, a new leader is selected from among the followers. The election of the new leader is handled by the ZooKeeper.

**Kafka Consumers**

In Kafka, the consumer has to issue requests to the brokers indicating the partitions it wants to consume. The consumer is required to specify its offset in the request and receives a chunk of log beginning from the offset position from the broker. Since the consumer has control over this position, it can re-consume data if required. Records remain in the log for a configurable time period which is known as the retention period. The consumer may re-consume the data as long as the data is present in the log.

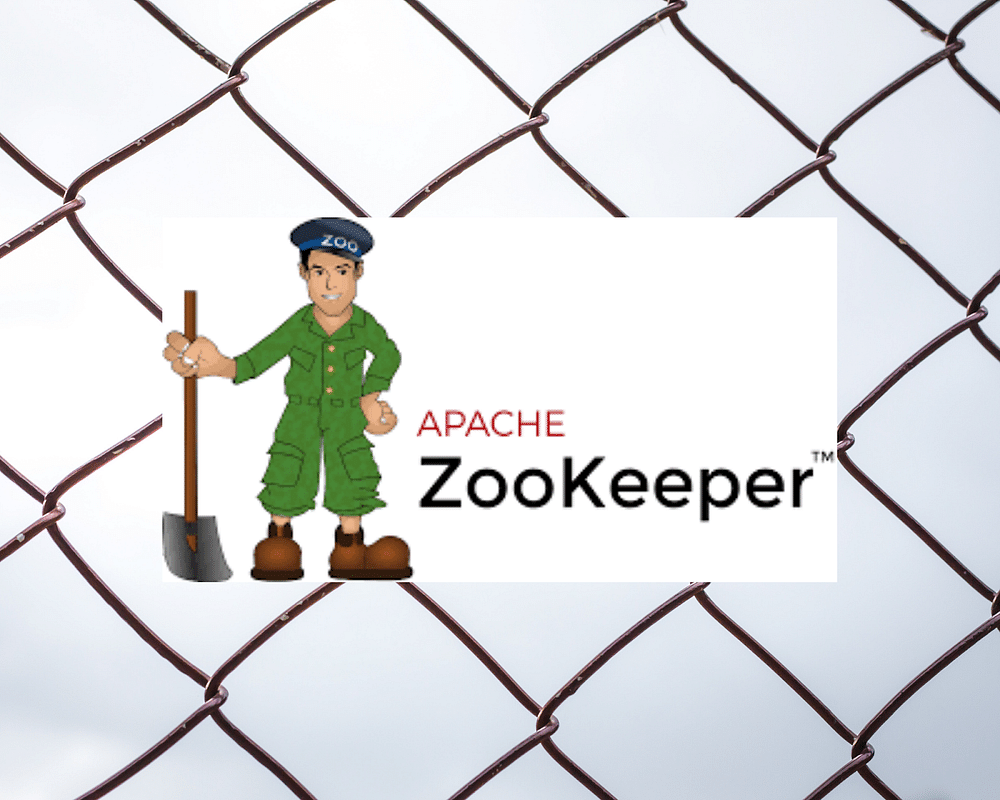
In Kafka, the consumers work on a pull-based approach. This means that data is not immediately pushed onto the consumers from the brokers. The consumers have to send requests to the brokers to indicate that they are ready to consume the data. A pull-based system ensures that the consumer does not get overwhelmed with messages and can fall behind and catch up when it can. A pull-based system can also allow aggressive batching of data sent to the consumer since the consumer will pull all available messages after its current position in the log. In this manner, batching is performed without any unnecessary latency.

**End to End Batch Compression**

To efficiently handle large volumes of data, Kafka performs compression of messages. Efficient compression involves compressing multiple messages together instead of compressing individual messages. For the reason that Apache Kafka supports an efficient batching format, a batch of messages can be compressed together and sent to the server in this format. The batch of messages here get written to the broker in a compressed format and continue to remain compressed in the log until they are extracted and decompressed by the consumer.

**Get confide**

## ****The Role of ZooKeeper in Apache Kafka Architecture****



[Apache ZooKeeper](https://www.projectpro.io/hadoop-tutorial/zookeeper-tutorial) is a software developed by Apache that acts as a centralized service and is used to maintain the configuration of data to provide flexible yet robust synchronization for distributed systems. The ZooKeeper is used to manage and coordinate Kafka brokers in the cluster. It maintains a list of the brokers and manages them using this list. It is used to notify the producer and consumer about the presence of new brokers or about the failure of brokers in the Kafka cluster. Using this information, the producer and consumer can make a decision and accordingly coordinate with some other broker in the cluster. The ZooKeeper is also used to store information about the Kafka cluster and the various details regarding the consumer clients. The ZooKeeper is also responsible for choosing a leader for the partitions. In the event of a failure in the leader node, it is the duty of the ZooKeeper to coordinate the leader election and choose the next leader for a partition.

Up until Kafka 2.8.0, it was not possible to run a Kafka cluster without the ZooKeeper. However, in the 2.8.0 release, the Kafka team is rolling out an alternative method where users can run a Kafka cluster without ZooKeeper but instead using an internal implementation of the Raft consensus algorithm. The changes are outlined in KIP-500 (Kafka Improvement Proposal - 500). The goal here is to move topic metadata and configurations out of ZooKeeper and into a new internal topic, named @metadata, which is managed by an internal Raft quorum of controllers, and replicated to all brokers in the cluster.

**Recommended Reading:**

* [**Apache Spark Architecture Explained in Detail**](https://www.projectpro.io/article/apache-spark-architecture-explained-in-detail/338/)
* [**Hadoop Architecture Explained in Detail**](https://www.projectpro.io/article/hadoop-architecture-explained-what-it-is-and-why-it-matters/317/)
* [**Top 75 Data Engineer Interview Questions and Answers for 2021**](https://www.projectpro.io/article/data-engineer-interview-questions-and-answers/456)

**Achieving Performance Tuning in Apache Kafka**

Optimum performance involves the consideration of two key measures: latency and throughput. Latency refers to the time taken to process one event. Hence a lower latency is required for better performance. Throughput denotes the number of events that can be processed in a specific amount of time, and hence, the goal is to always have a higher throughput. Many systems tend to optimize one and end up compromising the other, but Kafka attains a perfect balance of the two.

Tuning Apache Kafka for optimal performance involves:

* Tuning Kafka Producer: Data that the producers publish to the brokers is stored in a batch and sent only when the batch is ready. To tune the producers, two parameters are taken into consideration -
  + Batch Size: The batch size has to be decided based on the nature of the volume of messages sent by the producer. Producers which send messages frequently will work better with larger batch sizes so that throughput can be maximized without compromising heavily on the latency. In cases where the producers do not send messages frequently, smaller batch size is preferred. In such cases, if the batch size is very large, it may never get full or take a long time to fill up. This will increase the latency and hence, compromise performance.
  + Linger Time: The linger time is added to create a delay to allow more records to be filled up in the batch so that larger batches can be sent. A longer linger time allows more messages to be sent in one batch but can result in compromising latency. On the other hand, a reduced linger time results in fewer messages getting sent faster, and as a result, there is lower latency but reduced throughput too.
* Tuning Kafka Brokers: Every partition has a leader associated with it and zero or more followers for the leader. While the Kafka cluster is running, due to failures in some of the brokers or due to reallocation of partitions, an imbalance may occur among the brokers in the cluster. Some brokers might be overworked compared to others. In such cases, it is important to monitor the brokers and ensure that the workload is balanced across the various brokers present in the cluster.
* Tuning Kafka Consumers: While tuning consumers, it is important to keep in mind that a consumer can read many partitions, but one consumer can only read one partition. A good practice to follow is to keep the number of consumers equal to or lower than the partition count. If the customers are lower than the partition count, the number of partitions can be an exact multiple of the number of consumers. More consumers than partitions will result in some consumers remaining idle.

**Get More Practice, More**[**Big Data and Analytics Projects**](https://www.projectpro.io/projects/big-data-projects/apache-spark-projects?utm_source=Blg442&utm_medium=ConvertLink&utm_campaign=TXTCTA2)**, and More guidance.Fast-Track Your Career Transition with ProjectPro**

## ****Drawbacks of Apache Kafka****

We have already seen some interesting reasons that make Apache Kafka a popular tool for distributed streaming, but like every other big data tool, there are a few downsides to using Apache Kafka-

* Tweaking messages in Kafka results in performance issues. Kafka is well-suited for cases where the message does not have to be changed.
* In Kafka, there is no support for wildcard topic selection. The topic name has to be an exact match.
* Certain message paradigms such as point-to-point queues and request/reply features are not supported by Kafka.
* Large messages require compression and decompression of messages. This results in an effect on the throughput and performance of Kafka.

## ****Apache Kafka Use Cases****

**Message Broker**

Kafka serves as an excellent replacement for traditional message brokers. Compared to traditional massage brokers, Apache Kafka provides better throughput and is capable of handling a larger volume of messages. Kafka can be used as a publish-subscribe messaging service and is a good tool for large-scale message processing applications.

Commands : -

Run these commands from your Kafka root folder:

* cd bin\windows

Then run Zookeper server:

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

Then run Kafka server:

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

To create Topics

* kafka-topics.bat --bootstrap-server localhost:9092 --create --topic myTopic --partitions 1 --replication-factor 1

To see the lists of topics available

* kafka-topics.bat --bootstrap-server localhost:9092 –list

Create Kafka Producer

* kafka-console-producer.bat --bootstrap-server localhost:9092 --topic myTopic

On Second Window Create Kafka Consumer

* kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic myTopic –from-beginning

To know all the consumer groups

* kafka-consumer-groups.bat --bootstrap-server localhost:9092 --lists

Describe all the consumer groups

* kafka-consumer-groups.bat --bootstrap-server localhost:9092 --describe --group console-consumer-12605