**Apache Kafka:**

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 these challenges, we need a messaging system.

**Messaging system:**

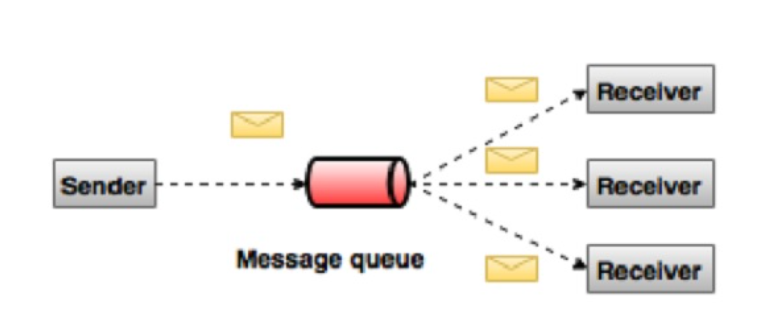
A messaging system is responsible for transferring data from one application to another, So the applications can focus on data, but not worry about how to share it. Distributed messaging is based on the concept of reliable message queuing. Messages are queued asynchronously between client applications and messaging system.

**Public subscribe messaging system:**

In the publish-subscribe system, messages are persisted in a topic. 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.

Ex:

Dish Tv, which publishes diff channels, and anyone can subscribe to their own set of channels and get them.



What is kafka?

Apache kafka is a distributed publish- subscribe messaging system and a robust queue that can handle a high volume of data and enable us to pass messages from one endpoint to another. Kafka is suitable for both offline and online message consumption.

Kafka messages persisted on the disk and replicated within the cluster to prevent data loss.

ADVANTAGES OF KAFKA:

Reliability: - Kafka is distributed, partitioned, replicated and fault tolerance.

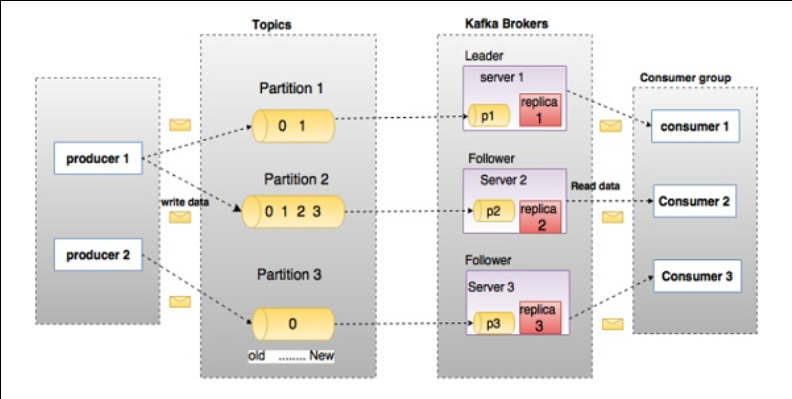
Scalability: -Kafka messaging system scales easily without down time.

Durability: -Kafka uses “Distributed commit log” which means messages persist 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.

Note:

Kafka is very fast and guarantees zero down time and zero data loss.



In the above diagram, First there is a topic then within topics are partitions, and finally the ordering of the messages in the topics is referred to as the offset. a topic configured into 3 partitions. Partition 1 has two offset factors 0 and 1. Partition 2 has four offset factors 0,1,2,3. Partition 3 has one offset factor 0.

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 cluster to make them available for all its operations. Multiple producers and consumers can publish and retrieve messages at the same time.

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.

PARTITION:

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

PARTITION OFFSET:

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

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.

BROKERS:

->A Kafka cluster is a group of multiple kafka brokers.

->A kafka broker is a server in the cluster this will receive and send the data.

->Each kafka broker is identified with an ID.(Integer)

->Each broker will have certain topic partitions.

->All the topic partitions data is distributed across all brokers(load balanced).

**EXAMPLE:**

Imagine a post office system that handles the distribution of messages (data) from one place to another. In this analogy:

* **Kafka Cluster:** The entire post office system.
* **Kafka Brokers:** The individual post offices in different locations.

Now, let's break down the analogy:

1. **Post Office System (Kafka Cluster):**
   * This is the entire Kafka ecosystem, consisting of multiple components working together.
   * It includes producers (people sending messages), consumers (people receiving messages), topics (types of messages), and brokers (post offices).
2. **Post Office (Kafka Broker):**
   * Each post office is like an independent Kafka broker.
   * Just as there can be multiple post offices in different locations, a Kafka cluster can have multiple brokers distributed across different servers or nodes.
3. **Mail Sender (Producer):**
   * People who want to send messages (data) go to the post office to drop off their mail.
   * In Kafka, producers are responsible for sending data (messages) to Kafka topics.
4. **Mail Receiver (Consumer):**
   * People who want to receive messages go to the post office to pick up their mail.
   * In Kafka, consumers are responsible for subscribing to topics and receiving messages.
5. **Message (Data):**
   * The letters or packages being sent represent the data in Kafka.
   * Messages in Kafka are organized into topics, and each message has a key and a value.
6. **Post Office Staff (Kafka Internals):**
   * Inside each post office, there are staff members who manage the incoming and outgoing mail.
   * Similarly, Kafka brokers have internal mechanisms for managing the flow of messages.
7. **Post Office Address (Topic):**
   * To send or receive mail, you use a specific address.
   * In Kafka, a topic is like an address. Producers send messages to a topic, and consumers subscribe to topics to receive messages.

**8.Mail Sorting (Partitioning):**

* Inside the post office, there's a sorting process to ensure mail goes to the right recipients.
* Kafka partitions messages within a topic to distribute the load and ensure efficient processing.

In summary, Kafka brokers are like post offices in a distributed messaging system. They receive, store, and distribute messages within the Kafka cluster, allowing producers to send data and consumers to receive data from specific topics. Just as post offices are distributed in different locations for better efficiency, Kafka brokers are distributed across servers to handle large-scale data processing.

KAFKA CLUSTER:

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

PRODUCERS:

Producers are the publishers 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 last segment file.

CONSUMERS:

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

LEADER:

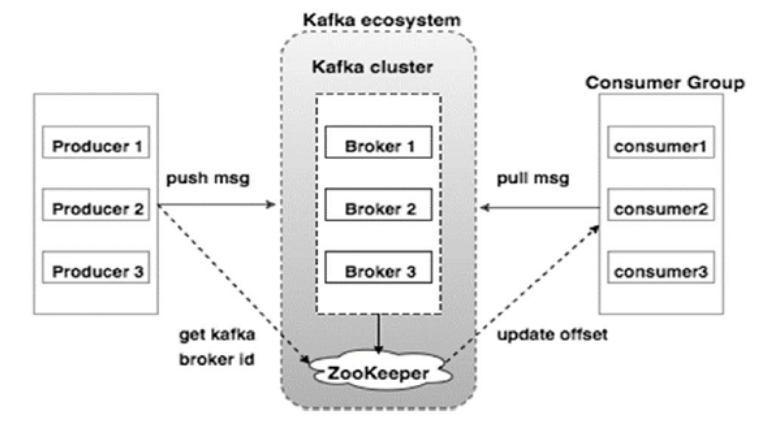
Leader is the node responsible for all reads and writes for the given partition. Every partition has one server acting as a leader.

FOLLOWER:

Node which follows leader instructions are called follower. If the leader fails, one of the follower

Will automatically become the new leader. A follower acts as normal consumer, pulls messages and updates its own data store.

**Apache kafka – cluster architecture:**

****

**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 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.

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.

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. Consumer offset value is notified by zookeeper.

**Apache Kafka – Workflow**

->Producers send messages 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 first partition and second message in second partition.

->Consumer subscribe to a specific topic.

->once the consumer subscribes to a specific topic, Kafka will provide the current offset of the topic to the consumer and 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 messages 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 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.

**MICROSERVICES:-**

**Circuit breaker:**

Use of the Circuit Breaker pattern can allow a microservice to continue operating when a related service fails, preventing the failure from cascading and giving the failing service time to recover.

Spring Reactive Web (for the service application) or Spring Reactive Web and Resilience4J (for the client application).

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-circuitbreaker-reactor-resilience4j</artifactId>

</dependency>

Spring Cloud Circuit Breaker provides an interface called ReactiveCircuitBreakerFactory, which we can use to create new circuit breakers for our application. An implementation of this interface is auto-configured, based on the starter that is on your application’s classpath. Now we can create a new service that uses this interface to make API calls to the client application: