Kafka Basics

Vivek Bansal
Senior Software Engineer at Uber | Ex-Grab | Ex-Directi
Published Feb 18, 2024

Today, we are living in a web of microservices. Usually, one microservice talks to other microservices in different ways like REST API or RPC calls. These are synchronous ways of communication, meaning that the call happens immediately.

Thanks for reading Curious Engineer! Subscribe for free to receive new posts and support my work.

But, what if a service (let's say 'A' service) does some business logic and ultimately wants to update all other microservices in the ecosystem (whoever needs that information)?

Now, if service ‘A’ would take time to call all services individually, you can imagine that the service responsibility would be impossible to manage. As the responsibility of the service increases, it is prone to more errors and causes a serious challenge as the number of services in the ecosystem increases.

That’s why we have asynchronous communication across microservices. It says after a service (let’s say ‘A’ service) has done some work, and few other services should know about that, we can send in the information which ‘A’ service has done through some channel and then it’s no longer service A’s responsibility to manage that information.

What is Kafka?

Apache Kafka’s website explains that Apache Kafka is an open-source “distributed” event “streaming platform”. In simple terms, this means that it helps us deliver information from one service to another in an async manner.

Question: What do you mean by “distributed”?

Answer: Kafka can store or process huge amounts of data. Since all this data cannot be accommodated on one server, that is why it has more than one server and thus it is called distributed.

Question: What do you mean by “streaming platform“?

Answer: It simply means that data in Kafka acts as a stream. These data streams are continuous flows of information generated by various sources, then Kafka stores it for some time and then it eventually gets consumed by the receivers (whoever needs to consume that information).

Basic Terminology

Let’s take a deep look into the basic terminologies for understanding Kafka.

Message

Message is the atomic unit of information for passing any piece of information passing from one system to another. Imagine that you need to emit application logs from your service for observability. The log could be of any JSON format like this:

```
{
  "key": "some-random-key",
  "value": "some-random-value",
  "created_date": "2024-02-18"
}
```

When we talk about pushing one message into Kafka, we are talking about pushing a JSON object as a message into Kafka. Pushing a JSON object means you can add any information like an integer, string, a JSON object inside the main object, etc.

You can think of a single message in Kafka as similar to a single row in a table in a relational database.

Topic

Even in a simple architecture, there are so many kinds of events being emitted. It only makes sense to logically group these different kinds of messages into one category each called a *Topic*.

In Kafka, a topic is an ordered log of events. When any external system/service writes an event to Kafka, it gets appended to the end of the topic. You can't insert any message in the middle of the Topic. You can have multiple topics in your Kafka cluster each storing a specific category of events. For example: you can have an "application-events" topic for all events emitted during user interaction with the app and a "database-events" topic for all events emitted during any database operation. You can think of a Topic in Kafka as similar to a table in a relational database.

Also, note that the Topics in Kafka are proper logs and *not* queues. They are durable and persist on the disk in the form of data files. You can configure the messages in the Kafka topic to expire after a certain time (TTL) but you can also choose to let the messages stay forever in the Kafka cluster.

Thanks for reading Curious Engineer! Subscribe for free to receive new posts and support my work

Partitions

Since the data in a Kafka cluster can infinitely grow, we cannot just rely on one server's memory and I/O capabilities for overall Kafka's performance. So, Kafka provides us an option of breaking the topics into partitions (similar to breaking a table into multiple shards).

Partitions are a systematic way of breaking one topic log file into many logs, each of which can be hosted on a separate server. This gives us the capability to scale out forever and get more performance out of the Kafka cluster. Now, the tasks of storing messages, reading messages, and writing new messages from the partitions can be split across many nodes that host that partition in the Kafka cluster.

The above image depicts the idea of partitioning a topic into multiple partitions. Each partition contains a different set of data. The union of all the data in all partitions is everything that got ingested into Kafka (assuming only 1 topic is there).

"Concept of Offset": An offset in Kafka is a unique identifier assigned to each message within a partition. It essentially acts as a pointer, indicating the position of a message within the partition.

When the consumer starts consuming the message from the topic, they use the last committed offset to know from where they should start reading next.

Producers

Producers are the external applications that write messages to a Kafka topic. The producer is responsible for taking care of which message should go to which partition (if there needs to be some specific assignment)

For this, Kafka provides you the capability to assign which message should go to which partition. The rules are simple: If there is no "key" in the message JSON, then Kafka uses a round-robin algorithm to distribute messages to all partitions evenly. If there is a "key" field in the message JSON, then the partition number of the message is found using a hash function.

Partition number to which message belongs = Hash(key) mod P P = number of partitions

The messages having the same key always get stored in the same partition and therefore they are always stored in order. You can also choose to mention the partition

number in the message or use some custom logic to find the destination partition number.

Consumers

Consumers are the external applications that consume messages from a Kafka topic and then do some work on the events consumed. This work could be like filtering the required events, calling an external service, or even updating a database upon consuming the event.

When a consumer starts consuming the message from a partition, it stores the offset in the Kafka cluster denoting to what position it has read the messages. If something breaks or the consumer goes down, the consumer can reuse this value stored in the Kafka cluster to restart consuming where it left off.

Also, a consumer can reset its offset value to start replaying the past events stored in the Kafka cluster.

A consumer can be a single instance or it can be many instances of the same program which is called a consumer Group. There are only two rules here:

1. A single consumer group can have multiple consumers

2. A consumer in a consumer group should at least consume data from one partition. It can consume data from multiple partitions as well. That's why, the number of consumers in a consumer group cannot be greater than the number of partitions in a topic.

Question: How is the ordering of messages maintained in a consumer group?

Answer: One partition can not be read by multiple consumers in the same consumer group. So, a single partition gets processed by only one consumer. Thus, all the messages in a single partition are guaranteed to be delivered in the same order in which they arrived in the partition.

Usually, there is out-of-the-box library support available in all popular languages for interacting with the Kafka cluster to produce/consume messages and you don't have to worry about writing all the implementation from scratch.

Thanks for reading Curious Engineer! Subscribe for free to receive new posts and support my work.

Brokers

Kafka is composed of a network of machines called brokers. Each broker manages some partitions, handles read requests for reading data from existing partitions as well as writes requests for storing data in these partitions. Brokers also manage the replication of partitions among other broker nodes.

ZooKeeper

Zookeeper in Kafka serves as:

1. Coordinator: It managed the cluster membership, tracking which brokers were active and available.

2. Leader election tool: It facilitates the selection of a leader broker for each partition, ensuring only one broker handles writes for that partition.

3. Configuration store: It stores important Kafka configuration details, including topics, partitions, and access control rules.

4. Coordination mechanism: It enables communication and synchronization between different Kafka components, like producers, consumers, and brokers.

Note: Zookeeper adds an extra management layer for Kafka. It's just an external tool for metadata management in the Kafka cluster. To keep things simple and more scalable from future perspective, Kafka is under process to shift to Kraft. Read more ***here****.*

Delivery Semantics

Delivery semantics refers to how the broker, producers, and consumers agree to share the messages stored in partitions. There are three types of message sharing:

1. At most once: Messages are delivered at most once. This means a message might be lost due to failures but will never be delivered twice. This is the simplest option which provides high throughput and low latency.

2. At least once: Messages are delivered at least once, but may be delivered multiple times due to retries on failures. This also guarantees no data loss and a good balance between throughput and data integrity. Since there can be multiple retry attempts for the same event, so there is duplicate processing possible and thus it requires handling duplicates in your application logic.

3. Exactly once: Messages are delivered exactly once, even in the presence of failures. The strongest guarantee ensures data integrity. This one is the most complex to achieve and might impact the performance of your Kafka cluster.

The above-discussed semantics can be applied to both producers and consumers. These semantics provide tradeoffs between latency (lowest in at-most-once and higher in other options) and message durability. Knowing which delivery semantic to use on the producer as well as the consumer depends on your use case.

Applications of Kafka

Here are some popular applications of Kafka across multiple industries

1. **Messaging:** Kafka can be used as a high-throughput, low-latency messaging system for exchanging data between different parts of an application or between different applications. This can be used for things like:Sending real-time updates to users, such as stock prices or social media notifications.Triggering microservices to perform actions based on events, such as processing a new order or updating a user profile.

2. **Log aggregation:** Kafka can be used to collect and aggregate logs from different sources, such as applications, servers, and network devices. This can be used for:Centralized log storage and analysis.Troubleshooting and debugging issues

3. **Metrics collection:** Kafka can be used to collect and store operational metrics from applications and systems. This can be used for:Monitoring the performance and health of applications and systems.Identifying and troubleshooting performance bottlenecks.Capacity planning and resource optimization

4. **Stream processing:** Kafka can be used as a platform for building real-time stream processing applications. This can be used for:Fraud detection in real-time.building recommendation

engines for movies/food/videos.Sensor data analysis for Internet of Things (IoT) applications.Real-time analytics for marketing and advertising.

5. **Event sourcing:** Kafka can be used to store and replay events that happen within a system. This can be used for:Auditing and debugging system behavior.Recovering from failures.Building applications that are eventually consistent.

That's it, folks for this edition of the newsletter. Please consider liking and sharing with your friends as it motivates me to bring you good content for free. If you think I am doing a decent job, share this article in a nice summary with your network. Connect with me on Linkedin or Twitter for more technical posts in the future!

Thanks for reading Curious Engineer! Subscribe for free to receive new posts and support my work.

Resources

Intro to Apache Kafka: How Kafka Works by Confluent

Message Delivery Guarantees by Confluent

Kafka Applications by Apache

Kafka Basics and Core Concepts