Apache Kafka - A distributed messaging system for log processing

Prakhar Srivastav (ps2894)

Summary

Kafka is a distributed messaging system that was developed at LinkedIn for collecting and delivering high volumes of log data with low latency. The primary use-case that led to the development of the project was to have the ability to do near real-time analysis on the log data collected by LinkedIn - a need which was not being met by the available solutions in the log analysis ecosystem.

Kafka is different from other systems in the sense that it is distributed, scalable and offers a high throughput while providing an API similar to a messaging system which allows applications to consume log events in real time.

Architecture

The key insight that underlies the architecture of Kafka stemmed from the differences in requirements from other messaging systems. For instance, the high delivery guarantees provided by messaging systems was at the cost of high throughput. Since having a very high throughput was of primary importance, the design of Kafka put that as their top-most design constraint. Lastly, in comparison to other systems, a key aspect of Kafka's design was the **pull architecture**.

> *At LinkedIn, we find the pull model more suitable for our applications since each consumer can retrieve the messages at the maximum rate it can sustain and avoid being flooded by messages pushed faster than it can handle.*

At a high level, Kafka provides two primary abstractions through its simple API

- *Topic*: A stream of messages of a particular type is defined by a topic. A publisher can publish messages to a topic.
- *Broker*: The published messages are stored on a set of servers called brokers. The consumer can subscribe to one or more topics from the brokers and consume the subscribed messages by pulling data from the brokers.

Each message stream provides an iterator interface over the continual stream of messages being produced. The consumer then iterates over every message in the stream and processes the payload of the message. Kafka supports both point-to-point delivery model in which multiple consumers jointly consume a single copy of all messages in a topic, as well as the publish/subscribe model in which multiple consumers each retrieve its own copy of a topic.

Lastly, since Kafka is distributed in nature, a cluster consists of multiple brokers each of which store one or more partitions of a topic.

Single Partition

The designers of Kafka took important decisions regarding storage, statelessness and transfer which form an important component of the system's design.

Storage: Segment files are flushed to disk only after a configurable number of messages have been published or a certain amount of time has elapsed. A message is only exposed to the consumers after it is flushed. Each message is addressed by its logical offset in the log which avoids the overhead of seek-intensive random-access index structures.

Transfer: The consumer API iterates one message at a time and pulls request from a consumer which retrieves multiple messages up to a certain size, typically hundreds of kilobytes. Instead of using memory for caching messages at the Kafka layer, the system uses the underlying the file system page cache.

Statelessness: In Kafka, the information about how much each consumer has consumed is not maintained by the broker, but by the consumer itself. It also uses a simple time-based SLA for the retaining messages after which a message is automatically deleted. An important side-benefit was that a consumer can deliberately rewind back to an old offset and re-consume data.

Distributed Coordination

Since Kafka is distributed it has a set of producers and consumers talking to multiple brokers each subscribed to a particular topic. Each publisher can speak to either a random broker or a static node decided by using a partitioning key. For consumers, Kafka has the concept of consumer groups. Each consumer group consists of one or more consumers that jointly consume a set of subscribed topics, i.e., each message is delivered to only one of the consumers within the group.

The key goal of the designers was to divide the messages stored in the brokers evenly among the consumers, without introducing too much coordination overhead. To facilitate this goal, the team took the following decisions

1. Make a partition within a topic the smallest unit of parallelism. This means that at any given time, all messages from one partition are consumed only by a single consumer within each consumer group.
2. Not have a central "master" node, but instead let consumers coordinate among themselves in a decentralized fashion. This was made possible by using Zookeeper for distributed coordination.

Delivery Guarantees

Delivery guarantees are an important part of any messaging system. Exposing clean semantics regarding the delivery (exactly-once, at-least once, at most once) to the users of the system is of utmost importance. In Kafka's case, Kafka guarantees at-least-once delivery. Most of the time, a message is delivered exactly once to each consumer group. However, in the case when a consumer process crashes without a clean shutdown, the consumer process that takes over those partitions owned by the failed consumer may get some duplicate messages.

Regarding the ordering of the messages Kafka guarantees that messages from a single partition are delivered to a consumer in order. However, there is no guarantee on the ordering of messages coming from different partitions.

Lastly, if a broker goes down, any message stored on it not yet consumed becomes unavailable. If the storage system on a broker is permanently damaged, any unconsumed message is lost forever.