Apache Kafka

A high-throughput distributed messaging system.

Kafka 0.9.0 Documentation

1. Getting Started

1.1 Introduction

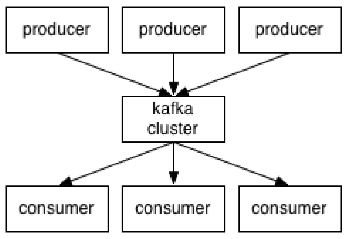
Kafka is a distributed, partitioned, replicated commit log service. It provides the functionality of a messagin system, but with a unique design.

What does all that mean?

First let's review some basic messaging terminology:

- Kafka maintains feeds of messages in categories called *topics*.
- We'll call processes that publish messages to a Kafka topic *producers*.
- We'll call processes that subscribe to topics and process the feed of published messages *consumers*..
- Kafka is run as a cluster comprised of one or more servers each of which is called a *broker*.

So, at a high level, producers send messages over the network to the Kafka cluster which in turn serves then up to consumers like this:

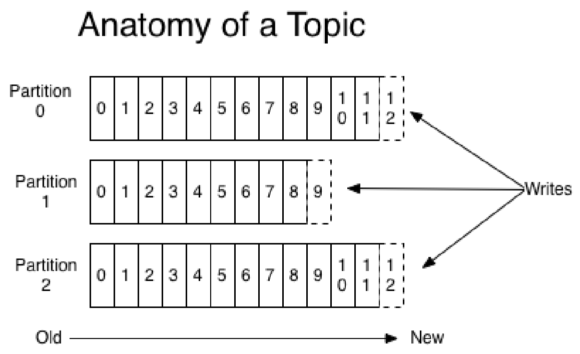


Communication between the clients and the servers is done with a simple, high-performance, language agnostic **TCP protocol**. We provide a Java client for Kafka, but clients are available in **many languages**.

Topics and Logs

Let's first dive into the high-level abstraction Kafka provides—the topic.

A topic is a category or feed name to which messages are published. For each topic, the Kafka cluster maintains a partitioned log that looks like this:



Each partition is an ordered, immutable sequence of messages that is continually appended to—a commit l
The messages in the partitions are each assigned a sequential id number called the *offset* that uniquely identifies each message within the partition.

The Kafka cluster retains all published messages—whether or not they have been consumed—for a configurable period of time. For example if the log retention is set to two days, then for the two days after a message is published it is available for consumption, after which it will be discarded to free up space. Kafka performance is effectively constant with respect to data size so retaining lots of data is not a problem.

In fact the only metadata retained on a per-consumer basis is the position of the consumer in the log, called the "offset". This offset is controlled by the consumer: normally a consumer will advance its offset linearly a reads messages, but in fact the position is controlled by the consumer and it can consume messages in any order it likes. For example a consumer can reset to an older offset to reprocess.

This combination of features means that Kafka consumers are very cheap—they can come and go without much impact on the cluster or on other consumers. For example, you can use our command line tools to "ta the contents of any topic without changing what is consumed by any existing consumers.

The partitions in the log serve several purposes. First, they allow the log to scale beyond a size that will fit o single server. Each individual partition must fit on the servers that host it, but a topic may have many partitions so it can handle an arbitrary amount of data. Second they act as the unit of parallelism—more on that in a bit.

Distribution

The partitions of the log are distributed over the servers in the Kafka cluster with each server handling data and requests for a share of the partitions. Each partition is replicated across a configurable number of serve for fault tolerance.

Each partition has one server which acts as the "leader" and zero or more servers which act as "followers". T leader handles all read and write requests for the partition while the followers passively replicate the leader the leader fails, one of the followers will automatically become the new leader. Each server acts as a leader some of its partitions and a follower for others so load is well balanced within the cluster.

Producers

Producers publish data to the topics of their choice. The producer is responsible for choosing which messag to assign to which partition within the topic. This can be done in a round-robin fashion simply to balance lo or it can be done according to some semantic partition function (say based on some key in the message). M on the use of partitioning in a second.

Consumers

Messaging traditionally has two models: **queuing** and **publish-subscribe**. In a queue, a pool of consumers m
read from a server and each message goes to one of them; in publish-subscribe the message is broadcast to
consumers. Kafka offers a single consumer abstraction that generalizes both of these—the *consumer group*.

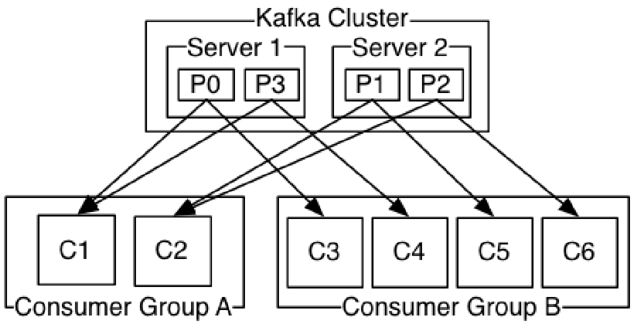
Consumers label themselves with a consumer group name, and each message published to a topic is delive
to one consumer instance within each subscribing consumer group. Consumer instances can be in separate
processes or on separate machines.

If all the consumer instances have the same consumer group, then this works just like a traditional queue
balancing load over the consumers.

If all the consumer instances have different consumer groups, then this works like publish-subscribe and all
messages are broadcast to all consumers.

More commonly, however, we have found that topics have a small number of consumer groups, one for eac
"logical subscriber". Each group is composed of many consumer instances for scalability and fault toleranc
This is nothing more than publish-subscribe semantics where the subscriber is cluster of consumers instead
a single process.

Kafka has stronger ordering guarantees than a traditional messaging system, too.



A two server Kafka cluster hosting four partitions (P0-P3) with two consumer groups. Consumer group A has two consumer instances and group B has four.

A traditional queue retains messages in-order on the server, and if multiple consumers consume from the queue then the server hands out messages in the order they are stored. However, although the server hands out messages in order, the messages are delivered asynchronously to consumers, so
they may arrive out of order on different consumers. This effectively means the ordering of the messages is
in the presence of parallel consumption. Messaging systems often work around this by having a notion of
"exclusive consumer" that allows only one process to consume from a queue, but of course this means that
there is no parallelism in processing.

Kafka does it better. By having a notion of parallelism—the partition—within the topics, Kafka is able to
provide both ordering guarantees and load balancing over a pool of consumer processes. This is achieved b
assigning the partitions in the topic to the consumers in the consumer group so that each partition is
consumed by exactly one consumer in the group. By doing this we ensure that the consumer is the only rea
of that partition and consumes the data in order. Since there are many partitions this still balances the load
over many consumer instances. Note however that there cannot be more consumer instances in a consume
group than partitions.

Kafka only provides a total order over messages *within* a partition, not between different partitions in a topi

Per-partition ordering combined with the ability to partition data by key is sufficient for most applications. However, if you require a total order over messages this can be achieved with a topic that has only one partition, though this will mean only one consumer process per consumer group.

Guarantees

At a high-level Kafka gives the following guarantees:

- Messages sent by a producer to a particular topic partition will be appended in the order they are sent That is, if a message M1 is sent by the same producer as a message M2, and M1 is sent first, then M1 wil have a lower offset than M2 and appear earlier in the log.
- A consumer instance sees messages in the order they are stored in the log.
- For a topic with replication factor N, we will tolerate up to N-1 server failures without losing any messages committed to the log.

More details on these guarantees are given in the design section of the documentation.

1.2 Use Cases

Here is a description of a few of the popular use cases for Apache Kafka. For an overview of a number of thes areas in action, see **this blog post**.

Messaging

Kafka works well as a replacement for a more traditional message broker. Message brokers are used for a variety of reasons (to decouple processing from data producers, to buffer unprocessed messages, etc). In comparison to most messaging systems Kafka has better throughput, built-in partitioning, replication, and fault-tolerance which makes it a good solution for large scale message processing applications.

In our experience messaging uses are often comparatively low-throughput, but may require low end-to-end latency and often depend on the strong durability guarantees Kafka provides.

In this domain Kafka is comparable to traditional messaging systems such as **ActiveMQ** or **RabbitMQ**.

Website Activity Tracking

The original use case for Kafka was to be able to rebuild a user activity tracking pipeline as a set of real-time publish-subscribe feeds. This means site activity (page views, searches, or other actions users may take) is published to central topics with one topic per activity type. These feeds are available for subscription for a range of use cases including real-time processing, real-time monitoring, and loading into Hadoop or offline data warehousing systems for offline processing and reporting.

Activity tracking is often very high volume as many activity messages are generated for each user page view

Metrics

Kafka is often used for operational monitoring data. This involves aggregating statistics from distributed applications to produce centralized feeds of operational data.

Log Aggregation

Many people use Kafka as a replacement for a log aggregation solution. Log aggregation typically collects physical log files off servers and puts them in a central place (a file server or HDFS perhaps) for processing. Kafka abstracts away the details of files and gives a cleaner abstraction of log or event data as a stream of messages. This allows for lower-latency processing and easier support for multiple data sources and distributed data consumption. In comparison to log-centric systems like Scribe or Flume, Kafka offers equal good performance, stronger durability guarantees due to replication, and much lower end-to-end latency.

Stream Processing

Many users end up doing stage-wise processing of data where data is consumed from topics of raw data anc then aggregated, enriched, or otherwise transformed into new Kafka topics for further consumption. For example a processing flow for article recommendation might crawl article content from RSS feeds and publ it to an "articles" topic; further processing might help normalize or deduplicate this content to a topic of cleaned article content; a final stage might attempt to match this content to users. This creates a graph of re time data flow out of the individual topics. **Storm** and **Samza** are popular frameworks for implementing thes kinds of transformations.

Event Sourcing

Event sourcing is a style of application design where state changes are logged as a time-ordered sequence c records. Kafka's support for very large stored log data makes it an excellent backend for an application built this style.

Commit Log

Kafka can serve as a kind of external commit-log for a distributed system. The log helps replicate data betwe nodes and acts as a re-syncing mechanism for failed nodes to restore their data. The **log compaction** feature Kafka helps support this usage. In this usage Kafka is similar to **Apache BookKeeper** project.