Apache Kafka Guide

Apache Kafka is a streaming message platform. It is designed to be high performance, highly available, and redundant. Examples of applications that can use such a platform include:

- **Internet of Things.** TVs, refrigerators, washing machines, dryers, thermostats, and personal health monitors can all send telemetry data back to a server through the Internet.
- **Sensor Networks.** Areas (farms, amusement parks, forests) and complex devices (engines) can be designed with an array of sensors to track data or current status.
- **Positional Data.** Delivery trucks or massively multiplayer online games can send location data to a central platform.
- **Other Real-Time Data.** Satellites and medical sensors can send information to a central area for processing.

Ideal Publish-Subscribe System

The ideal publish-subscribe system is straight-forward: Publisher A's messages must make their way to Subscriber A, Publisher B's messages must make their way to Subscriber B, and so on.

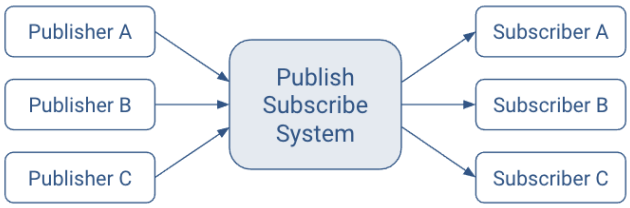

Figure 1: Ideal Publish-Subscribe System

An ideal system has the benefit of:

- **Unlimited Lookback.** A new Subscriber A1 can read Publisher A's stream at any point in time.
- **Message Retention.** No messages are lost.
- **Unlimited Storage.** The publish-subscribe system has unlimited storage of messages.
- **No Downtime.** The publish-subscribe system is never down.
- **Unlimited Scaling.** The publish-subscribe system can handle any number of publishers and/or subscribers with constant message delivery latency.

Now let's see how Kafka's implementation relates to this ideal system.

Kafka Architecture

As is the case with all real-world systems, Kafka's architecture deviates from the ideal publish-subscribe system. Some of the key differences are:

- Messaging is implemented on top of a replicated, distributed commit log.
- The client has more functionality and, therefore, more responsibility.
- Messaging is optimized for batches instead of individual messages.
- Messages are retained even after they are consumed; they can be consumed again.

The results of these design decisions are:

- Extreme horizontal scalability
- Very high throughput
- High availability
- but, different semantics and message delivery guarantees

The next few sections provide an overview of some of the more important parts, while later section describe design specifics and operations in greater detail.

Topics

In the ideal system presented above, messages from one publisher would somehow find their way to each subscriber. Kafka implements the concept of a topic. A topic allows easy matching between publishers and subscribers.

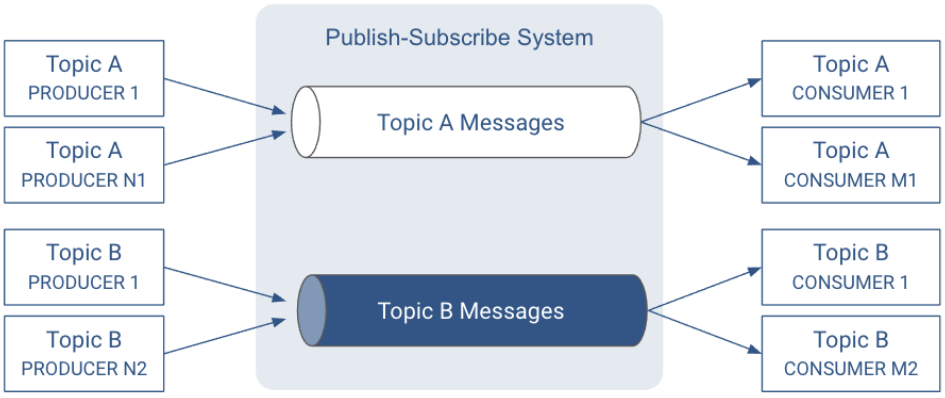

Figure 2: Topics in a Publish-Subscribe System

A topic is a queue of messages written by one or more producers and read by one or more consumers. A topic is identified by its name. This name is part of a global namespace of that Kafka cluster.

Specific to Kafka:

- Publishers are called producers.
- Subscribers are called consumers.

As each producer or consumer connects to the publish-subscribe system, it can read from or write to a specific topic.

Brokers

Kafka is a distributed system that implements the basic features of the ideal publish-subscribe system described above. Each host in the Kafka cluster runs a server called a broker that stores messages sent to the topics and serves consumer requests.

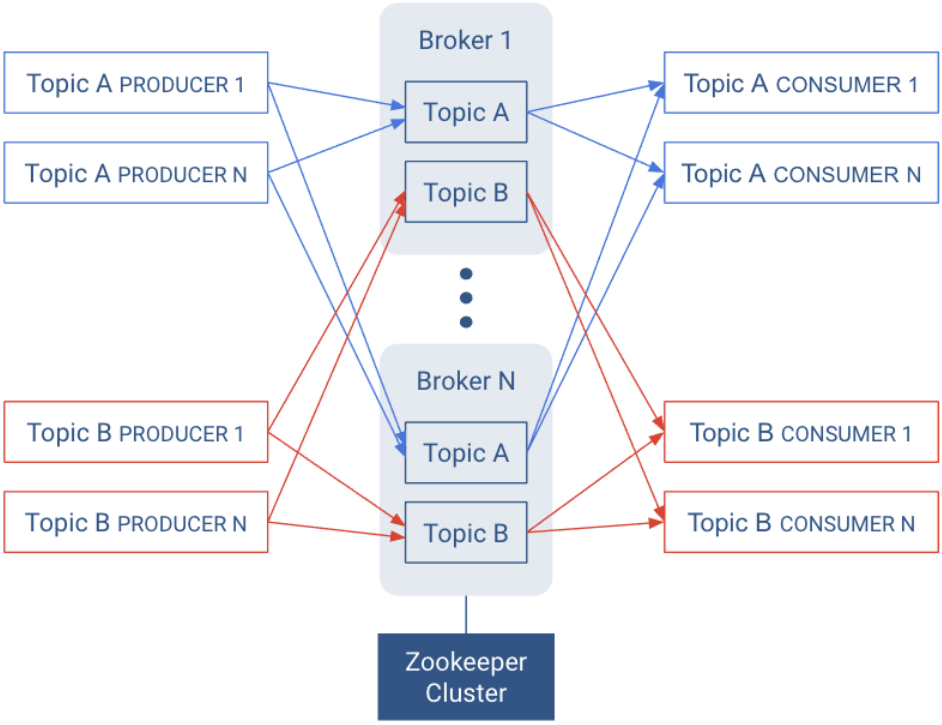

Figure 3: Brokers in a Publish-Subscribe System

Kafka is designed to run on multiple hosts, with one broker per host. If a host goes offline, Kafka does its best to ensure that the other hosts continue running. This solves part of the “No Downtime” and “Unlimited Scaling” goals from the ideal publish-subscribe system.

Kafka brokers all talk to Zookeeper for distributed coordination, additional help for the "Unlimited Scaling" goal from the ideal system.

Topics are replicated across brokers. Replication is an important part of “No Downtime,” “Unlimited Scaling,” and “Message Retention” goals.

There is one broker that is responsible for coordinating the cluster. That broker is called the controller.

As mentioned earlier, an ideal topic behaves as a queue of messages. In reality, having a single queue has scaling issues. Kafka implements partitions for adding robustness to topics.

Records

In Kafka, a publish-subscribe message is called a record. A record consists of a key/value pair and metadata including a timestamp. The key is not required, but can be used to identify messages from the same data source. Kafka stores keys and values as arrays of bytes. It does not otherwise care about the format.

The metadata of each record can include headers. Headers may store application-specific metadata as key-value pairs. In the context of the header, keys are strings and values are byte arrays.

For specific details of the record format, see the Record definition in the Apache Kafka documentation.

Partitions

Instead of all records handled by the system being stored in a single log, Kafka divides records into partitions. Partitions can be thought of as a subset of all the records for a topic. Partitions help with the ideal of “Unlimited Scaling”.

Records in the same partition are stored in order of arrival.

When a topic is created, it is configured with two properties:

partition count

The number of partitions that records for this topic will be spread among.

replication factor

The number of copies of a partition that are maintained to ensure consumers always have access to the queue of records for a given topic.

Each topic has one leader partition. If the replication factor is greater than one, there will be additional follower partitions. (For the replication factor = M, there will be M-1 follower partitions.)

Any Kafka client (a producer or consumer) communicates only with the leader partition for data. All other partitions exist for redundancy and failover. Follower partitions are responsible for copying new records from their leader partitions. Ideally, the follower partitions have an exact copy of the contents of the leader. Such partitions are called in-sync replicas (ISR).

With N brokers and topic replication factor M, then

- If M < N, each broker will have a subset of all the partitions
- If M = N, each broker will have a complete copy of the partitions

In the following illustration, there are N = 2 brokers and M = 2 replication factor. Each producer may generate records that are assigned across multiple partitions.

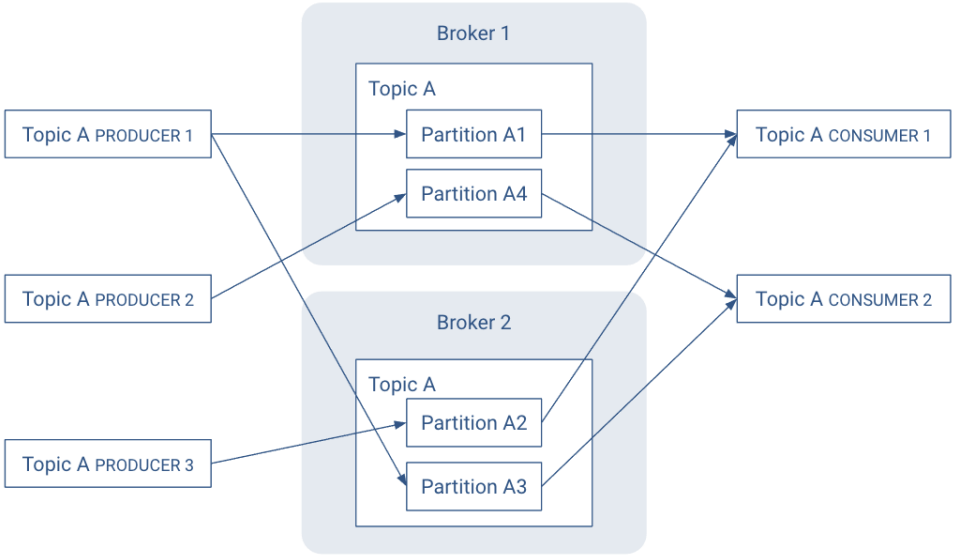

Figure 4: Records in a Topic are Stored in Partitions, Partitions are Replicated across Brokers

Partitions are the key to keeping good record throughput. Choosing the correct number of partitions and partition replications for a topic

- Spreads leader partitions evenly on brokers throughout the cluster
- Makes partitions within the same topic are roughly the same size.
- Balances the load on brokers.

Record Order and Assignment

By default, Kafka assigns records to a partitions round-robin. There is no guarantee that records sent to multiple partitions will retain the order in which they were produced. Within a single consumer, your program will only have record ordering within the records belonging to the same partition. This tends to be sufficient for many use cases, but does add some complexity to the stream processing logic.

Tip: Kafka guarantees that records in the same partition will be in the same order in all replicas of that partition.

If the order of records is important, the producer can ensure that records are sent to the same partition. The producer can include metadata in the record to override the default assignment in one of two ways:

- The record can indicate a specific partition.
- The record can includes an assignment key.

  The hash of the key and the number of partitions in the topic determines which partition the record is assigned to. Including the same key in multiple records ensures all the records are appended to the same partition.

Logs and Log Segments

Within each topic, each partition in Kafka stores records in a log structured format. Conceptually, each record is stored sequentially in this type of "log".

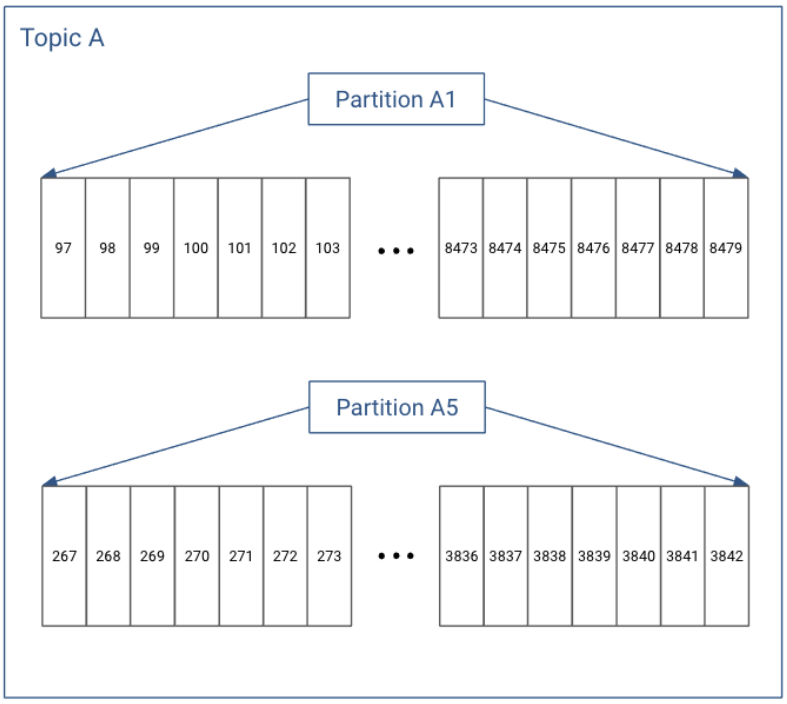

Figure 5: Partitions in Log Structured Format

Note: These references to "log" should not be confused with where the Kafka broker stores their operational logs.

In actuality, each partition does not keep all the records sequentially in a single file. Instead, it breaks each log into log segments. Log segments can be defined using a size limit (for example, 1 GB), as a time limit (for example, 1 day), or both. Administration around Kafka records often occurs at the log segment level.

Each of the partitions is broken into segments, with Segment N containing the most recent records and Segment 1 containing the oldest retained records. This is configurable on a per-topic basis.

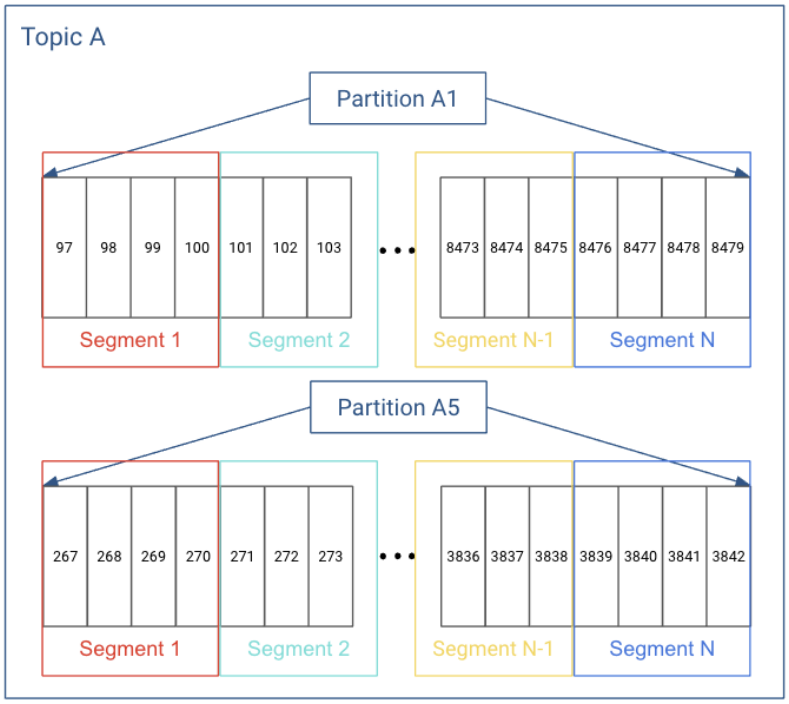

Figure 6: Partition Log Segments

Kafka Brokers and ZooKeeper

The broker, topic, and partition information are maintained in Zookeeper. In particular, the partition information, including partition and replica locations, updates fairly frequently. Because of frequent metadata refreshes, the connection between the brokers and the Zookeeper cluster needs to be reliable. Similarly, if the Zookeeper cluster has other intensive processes running on it, that can add sufficient latency to the broker/Zookeeper interactions to cause issues.

- Kafka Controller maintains leadership via Zookeeper (shown in orange)
- Kafka Brokers also store other relevant metadata in Zookeeper (also in orange)
- Kafka Partitions maintain replica information in Zookeeper (shown in blue)

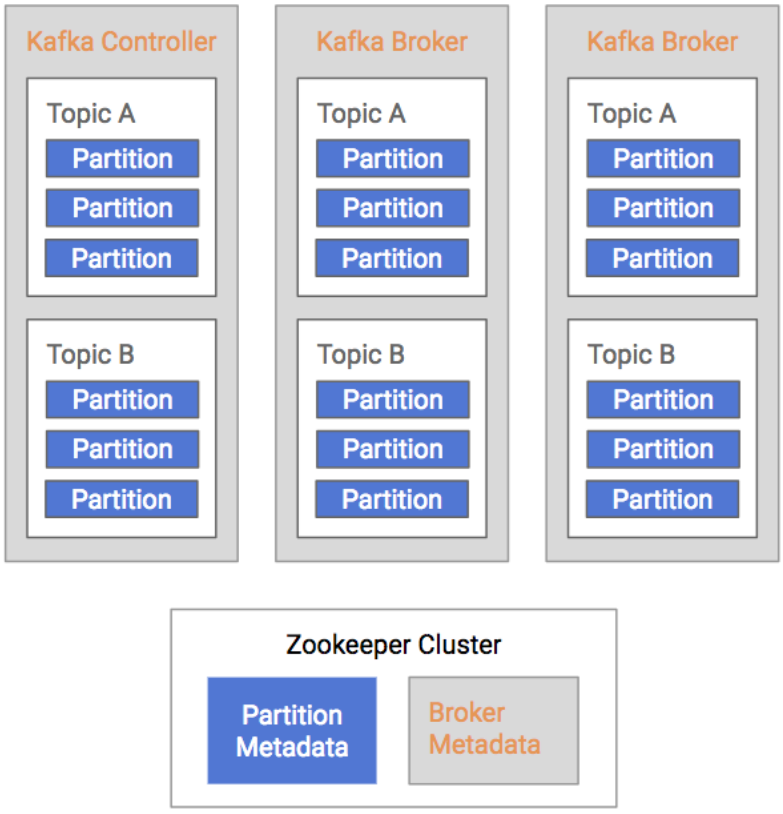

Figure 7: Broker/ZooKeeper Dependencies