Understanding Apache Kafka®

Overview

Apache Kafka® is a hot technology amongst application developers and architects looking to build the latest generation of real-time and web-scale applications. According the official Apache Kafka® website "Kafka is used for building real-time data pipelines and streaming apps. It is horizontally scalable, fault-tolerant, wicked fast, and runs in production in thousands of companies."

This paper will explore that statement in a bit more detail to help you understand when and why you would use Kafka in your application and some of the key considerations when developing and deploying.

Copyright ©2018, 2021, 2022 Instaclustr, All rights reserved.

Why Use a Queuing or Streaming Engine?

Kafka is part of general family of technologies known as queuing, messaging, or streaming engines. Other examples in this broad technology family include traditional message queue technology such RabbitMQ, IBM MQ, and Microsoft Message Queue. It can be said that Kafka is to traditional queuing technologies as NoSQL technology is to traditional relational databases.

These newer technologies break through scalability and performance limitations of the traditional solutions while meeting similar needs, Apache Kafka can also be compared to proprietary solutions offered by the big cloud providers such as AWS Kinesis, Google Cloud Dataflow, and Azure Stream Analytics.

The wealth of very popular options in this family of technologies is clear evidence of real and widespread need. However, it may not be immediately obvious what role these technologies play in an architecture. Why would I want to stick some other complicated thing in between the source of my events and the consumers that use the events?

- To smooth and increase reliability in the face of temporary spikes in workload.
  That is to deal gracefully with temporary incoming message rates greater than the processing app can deal with by quickly and safely storing the message until the processing system catches up and can clear the backlog. The engineers at Slack have published an excellent blog post explaining how they use Kafka for just this purpose in their architecture: https://slack.engineering/scaling-slacks-job-queue-687222e9d100

  An extension to this buffering case is where the consuming application is completely unavailable. In this case the queuing solution can keep receiving messages from producers and retain them until the consuming application comes back online. An example of this case might be an IoT application—the devices sending readings are not going to stop sending information because your processing system is down or under maintenance. However, the messages can be stored in a queue and processed once the outage is finished.

- To increase flexibility in your application architecture by completely decoupling applications that produce events from the applications that consume them.
  This is particularly important to successfully implementing a microservices architecture, the current state of the art in application architectures. By using a queuing system, applications that are producing events simply publish them to a named queue and applications that are interested in the events consume them off the queue. The publisher and then consumer don't need to know anything about each other except for the name of the queue and the message schema. There can be one or many producers publishing the same kind of message to the queue and one or many consumers reading

the message and neither side will care.

To illustrate this, consider an architecture where you initially have a web front end that captures new customer details and some backend process that stores these details in a database. By putting a queue in the middle and posting "new customer" events to that queue I can, without changing existing code, do things like:

- add an new API application that accepts customer registrations from a new partner and posts them to the queue; or
- add a new consumer application that registers the customer in a CRM system.

Instaclustr's Kongo series of blog posts provides some very detailed examples and considerations when architecting an application this way.

Why Use Kafka?

The objectives we've mentioned above can be achieved with a range of technologies. So why would you use Kafka rather than one of those other technologies for your use case?

- It's highly scalable
- It's highly reliable due to built in replication, supporting true always-on operations
- It's Apache Foundation open source with a strong community
- It has built-in optimizations such as compression and message batching
- It has a strong reputation for being used by leading organizations.

  For example: LinkedIn (orginator), Pinterest, AirBnB, Datadog, Rabobank, Twitter, Netflix (see https://kafka.apache.org/powered-by for more)
- It has a rich ecosystem around it including many connectors

These properties (and others) of Kafka lead it to be suitable for additional architectural functions compared to the broad family of queuing and streaming engines. In particular, Kafka can be used as:

- A distributed log store in a Kappa architecture
  In this model, the messages stored in Kafka are the definitive source of truth for your application. You may use a database, caches, and other mechanism to provide views of the state for performance reasons but these can always be recreated from the message stored. This architecture has significant advances for auditability and recovering from errors.

- A stream processing engine
  Performing calculations on streams (to provide a simple example—calculating an average value over the last 5 messages or 5 minutes) is a complex, specialist problem best supported by an architectural framework that allows you to focus on your business logic. The Kafka Streams library provides this stream processing framework for use with Kafka.

Looking Under the Hood

Let's take a look at how Kafka achieves all this:

We'll start with **PRODUCERS**. Producers are the applications that generate events and publish them to Kafka. Of course, they don't randomly generate events—they create the events based on interactions with people, things, or systems. For example a mobile app could generate an event when someone clicks on a button, an IoT device could generate an event when a reading occurs, or an API application could generate an event when called by another application (in fact, it is likely an API application would sit between a mobile app or IoT device and Kafka). These producer applications use a Kafka producer library (similar in concept to a database driver) to send events to Kafka with libraries available for Java, C/C++, Python, Go, and .NET.

The next component to understand is the **CONSUMERS**. Consumers are applications that read the event from Kafka and perform some processing on them. Like producers, they can be written in various languages using the Kafka client libraries.

The core of the system is the Kafka **BROKERS**. When people talk about a Kafka cluster they are typically talking about the cluster of brokers. The brokers receive events from the producer and reliably store them so they can be read by consumers.

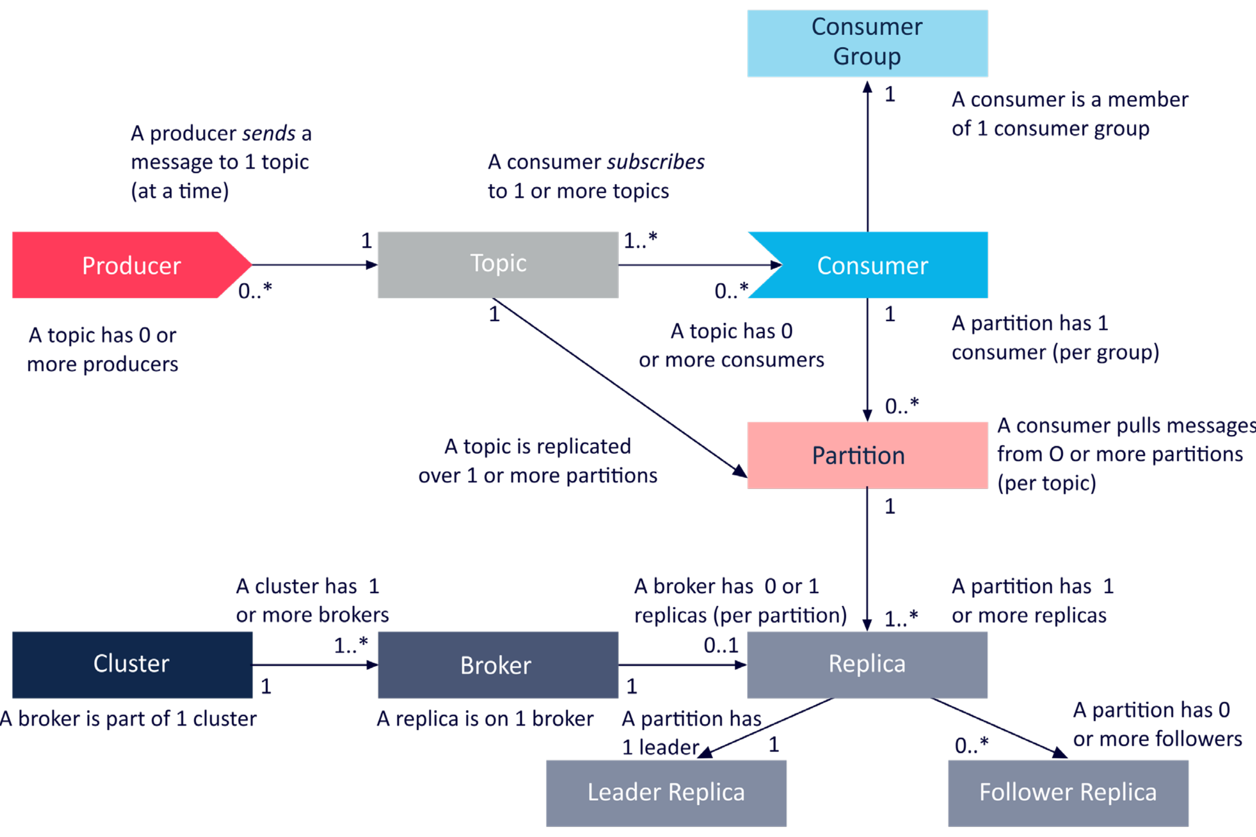
The brokers are configured with **TOPICS**. Topics are a bit like tables in a database, separating different types of data. Each topic is split into **PARTITIONS**. When an event is received, a record is appended to the log file for the topic and partition that the event belongs to (as determined by the metadata provided by the producer). Each of the partitions that make up a topic are allocated to the brokers in the cluster. This allows each broker to share the processing of a topic. When a topic is created, it can be configured to be replicated multiple times across the cluster so that the data is still available for even if a server fails. For each partition, there is a single leader broker at any point in time that serves all reads and writes. The leader is responsible for synchronizing with the replicas. If the leader fails, Kafka will automatically transfer leader responsibility for its partitions to one of the replicas.

As well as reliability, this topic and partition schema has implications for scalability. There

can be as many active brokers receiving and providing events as there are partitions in the topic so, provided sufficient partitions are configured, Kafka clusters can be scaled-out to provide increased processing throughput.

In some instances, guaranteed ordering of message delivery is important so that events are consumed in the same order they are produced. Kafka can support this guarantee at the topic level. To facilitate this, consumer applications are placed in consumer groups and within a **CONSUMER GROUP** a partition is associated with only a single consumer instance per consumer group.

The following diagram illustrates all these Kafka concepts and their relationships:



Operating Kafka

A Kafka cluster is a complex distributed system with many configuration properties and possible interactions between components in the system. Operated well, Kafka can operate at the highest levels of reliability even in relatively unreliable infrastructure environments such as the cloud.

At a high level, the principles for successfully operating Kafka are the same as other distributed server systems:

- choose a hardware and operating system configuration that is appropriate for the characteristics of the system
- have a monitoring system in place, and understand and alert on the key metrics that indicate the health of the system
- have documented and tested procedures (or better yet, automated processes) for dealing with failures, and
- consider, test, and monitor security of your configuration.

Specifically for Kafka you need to consider factors such as appropriate choice of topics and partitions, placement of brokers into racks aligned with failure domains and placement, and configuration of Apache ZooKeeper™. Our white paper on Ten Rules for Managing Kafka provides a great primer on the key considerations. Visit our Resource section to download the same.

Instaclustr Managed Kafka

At Instaclustr, we are specialists in operating distributed systems to provide reliability at scale. We have more than 175 million node hours of experience managing Apache Cassandra and Apache Spark™ and have chosen to extend our offering to include Apache Kafka as a managed service.

We chose to add Kafka to our offering for a number of reasons:

- Kafka, like Cassandra and Spark, is used when you need to build applications that support the highest levels of reliability and scale. The three technologies are often used together in a single application. The applications demand the same mission critical levels of service from a managed service provider.
- Kafka is Apache Foundation open source software with a massive user community—the software is maintained under a robust governance model ensuring it is not overly influenced by commercial interests and that users can freely use the software as they need to. There are no licensing fees and no vendor lock-in.
- Kafka has many architectural similarities to Cassandra and Spark allowing us to leverage our operational experience such as tuning and troubleshooting JVMs, dealing with public cloud environments and their idiosyncrasies and operating according to SOC 2 principles for a secure and robust environment.

> ***“We see Apache Kafka as a core capability for our architectural strategy as we scale our business. Getting set up with Instaclustr’s Kafka service was easy and significantly accelerated our timelines. Instaclustr consulting services were also instrumental in helping us understand how to properly use Kafka in our architecture.”***

Glen McRae, CTO, Lendi

> ***“As very happy users of Instaclustr’s Cassandra and Spark managed services, we’re excited about the new Apache Kafka managed service. Instaclustr quickly got us up and running with Kafka and provided the support we needed throughout the process.”***

Mike Rogers, CTO, SiteMinder

About Instaclustr

Instaclustr helps organizations deliver applications at scale through its managed platform for open source technologies such as **Apache Cassandra®**, **Apache Kafka®**, **Apache Spark™**, **Redis™**, **OpenSearch®**, **PostgreSQL®**, and **Cadence®**.

Instaclustr combines a complete data infrastructure environment with hands-on technology expertise to ensure ongoing performance and optimization. By removing the infrastructure complexity, we enable companies to focus internal development and operational resources on building cutting edge customer-facing applications at lower cost. Instaclustr customers include some of the largest and most innovative Fortune 500 companies.

© 2021 Instaclustr Copyright | Apache®, Apache Cassandra®, Apache Kafka®, Apache Spark™, and Apache ZooKeeper™ are trademarks of The Apache Software Foundation. Elasticsearch™ and Kibana™ are trademarks for Elasticsearch BV. Kubernetes® is a registered trademark of the Linux Foundation. OpenSearch is a registered trademark of Amazon Web Services. Postgres®, PostgreSQL® and the Slonik Logo are trademarks or registered trademarks of the PostgreSQL Community Association of Canada, and used with their permission. Redis™ is a trademark of Redis Labs Ltd. *Any rights therein are reserved to Redis Labs Ltd. Cadence is a trademark of Uber Technologies, Inc. Any use by Instaclustr Pty Limited is for referential purposes only and does not indicate any sponsorship, endorsement or affiliation between Redis and Instaclustr Pty Limited. All product and service names used in this website are for identification purposes only and do not imply endorsement.