instaclustr.com

RabbitMQ vs. Apache Kafka®: Key Differences and Use Cases

Instaclustr

12-15 minutes

Introduction

Whether you're integrating multiple microservices, looking to improve app reliability, or building a new streaming app, you might need a message queue (MQ) or message broker platform. These types of software pass messages from producing apps or services to consuming apps or services.

Two of the most popular platforms for handling messages are Apache Kafka and RabbitMQ. At a high level, they have similar functions, though there are important differences between them. Understanding those differences can help you choose one or the other for your particular use case.

In this blog, we will explore how Kafka and RabbitMQ work, some of their differences, and the best use cases for each.

Asynchronous Messaging Patterns

Both Kafka and RabbitMQ use asynchronous messaging to pass information from producing apps to consuming apps. The messaging is considered asynchronous because producing and consuming apps do not need to be active at the same time. The producer can deliver a message, and if the consumer is not

currently available or able to receive it, the message is stored until the consumer is ready. This approach to messaging is similar to asynchronous email or texting rather than synchronous phone calls or video conferencing: with Kafka and RabbitMQ, messaging does not have to occur in real time.

There are two primary asynchronous messaging patterns: message queues and publish/subscribe patterns.

Message Queues

With the message queue pattern, a producing app delivers messages to a queue. When the consuming app is ready to receive messages, it connects to the queue and retrieves the messages, removing them from the queue. You might have multiple consuming apps, but each message is consumed by only one consumer.

Publish/Subscribe

With the publish/subscribe (pub/sub) messaging pattern, producers publish messages, and multiple consumers are able to consume each message. When consuming apps are interested in a particular producer's messages, they subscribe to a channel where that producer will send its messages.

This pattern is typically used when you need a message or event to trigger multiple actions. Unlike the message queue pattern, pub/sub messaging ensures that consuming apps receive messages in the same order in which they were received by the messaging system.

RabbitMQ is an open source distributed message broker. It is often labeled as a "mature" platform (it was first released in 2007) and grouped with "traditional" messaging middleware platforms, such as IBM MQ and Microsoft Message Queue.

Developers often choose RabbitMQ for its flexibility. It can handle complex routing scenarios, and it supports multiple messaging

protocols, including AMQP, MQTT, and STOMP. It can be deployed in distributed configurations for scaling and delivering high availability.

RabbitMQ has a large community. Developers can easily find clients, plug-ins, and guides, and they can opt for commercial support through Pivotal (which was acquired by VMware). RabbitMQ also has a large number of high-profile enterprise users, including Reddit, Robinhood, T-Mobile, trivago, Accenture, Alibaba Travel, and more.

Architecture

The RabbitMQ architecture includes producers, exchanges, queues, and consumers. A producer pushes messages to an exchange, which then routes messages to queues (or other exchanges). A consumer then continues to read messages from the queue, often up to a predetermined limit of messages.

Queues

A RabbitMQ queue is a sequential data structure. Producers add data to the tail of the queue; consumers receive data from the head of the queue. The queues are “first in, first out” with RabbitMQ: the first message in the queue is consumed first. Queues have some mandatory properties (such as a name) and some optional properties (such as arguments used by plug-ins).

Message Exchanges

RabbitMQ message exchanges—which determine how messages are routed—provide a great deal of flexibility. With RabbitMQ, producers send messages to one of four exchange types:

- **Direct exchanges** route messages according to the routing key that the message carries. The routing key is a string of words,

separated by periods, that has some relevance to the message.

- **Fanout exchanges** route messages to all available queues. In this broadcasting type of exchange, the routing key is ignored.
- **Topic exchanges** route messages to one or more queues according to a complete or partial match with the routing key.
- **Header exchanges** route messages based on the message headers, which can contain more attributes than a routing key.

  These exchange types enable RabbitMQ to handle complex routing scenarios with multiple consuming apps or services.

Apache Kafka

Apache Kafka is an open source distributed event-streaming platform. Originally developed by LinkedIn to track website activity, Kafka today is generally employed for building real-time data pipelines and streaming apps. Often considered the leading streaming and queuing technology for large-scale, always-on, and event-driven apps, Kafka is regularly among the top five most active projects of the Apache Software Foundation.

Developers choose Kafka for several reasons:

- **Scalability:** Kafka's distributed architecture enables significant horizontal scalability.
- **Performance:** Kafka is fast! It can process millions of messages per second with relatively modest resources.
- **Flexibility:** Designed to interface with a variety of systems, Kafka has useful, intuitive APIs.
- **Availability:** Kafka delivers high availability through load balancing and data replication.
- **Community:** As part of the Apache Software Foundation, Kafka

has a rich ecosystem and community.

- **Strong reputation:** Kafka is used by leading, high-profile organizations, including not only LinkedIn but also Netflix, Twitter, Spotify, Pinterest, Airbnb, Uber, and many others.

Architecture

The Kafka architecture comprises producers, consumers, clusters, brokers, topics, and partitions. Producers send records to clusters, which store those records and then pass them to consumers. Each server node in the cluster is a "broker," which stores the data provided by the producer until it is read by the consumer.

Topics

Instead of "queues," Kafka uses "topics." A topic is a stream of data comprising individual records—which, as the introduction to Kafka suggests, is like a folder in a filesystem. Each topic is split into partitions, which are unchangeable sequences of records where messages are appended. Each record has a sequential ID called an "offset," which sets its place in line. A producer appends records to a topic partition, and a consumer subscribes to changes.

Kafka can spread messages across partitions. You might decide to place those partitions on multiple brokers so that multiple consumers can read from a topic in parallel while also enabling a topic to hold more data than could fit on any one machine. Alternatively, producers can create logical message streams, which can help ensure the delivery of messages in the right order for consumers.

Kafka and RabbitMQ Messaging Patterns

While RabbitMQ uses exchanges to route messages to queues, Kafka uses more of a pub/sub approach. A producer sends its

messages to a specific topic. A single consumer or multiple consumers—a “consumer group”—can consume those messages. Consumers and consumer groups can then subscribe to one or more topics so they are alerted when there are new messages or events.

With Kafka, consumers are able to reread previous, stored messages, for as long as you decide to keep messages in partitions. This ability to retain messages opens key use cases such as event sourcing and log aggregation (described later in this post).

Both Kafka and RabbitMQ allow you to push and pull messages, and to buffer messages when the consumer is busy or unavailable. Both also provide a way to get more than one message at a time. With RabbitMQ, this is known as “pre-fetching” and with Kafka, it is known as processing messages in “batch size.”

Security and Operations

Both Kafka and RabbitMQ provide built-in tools and capabilities for managing security and operations. In addition, both platform ecosystems offer third-party tools that augment monitoring and management capabilities.

Kafka offers security features such as Transport Layer Security (TLS) encryption, Simple Authentication and Security Layer (SASL) authentication, and role-based access control (RBAC). If you decide to manage your own cluster, you can control Kafka security policies through a command-line interface (CLI). Third-party monitoring tools can help you track metrics from brokers, topics, partitions, producers, and consumers.

Like Kafka, RabbitMQ supports TLS encryption, SASL authentication, and RBAC. In addition to CLI tools, RabbitMQ offers a browser-based API for management and monitoring of users and

queues. You can use open source or commercial tools to monitor metrics from nodes, clusters, queues, and more.

What Are the Best Use Cases?

At a high level, Kafka and RabbitMQ have some common use cases. For example, both can be used as part of a microservices architecture that connects producing and consuming apps. Both can also be used as a message buffer, providing a location to temporarily store messages when consuming apps are unavailable or smoothing out spikes in messages generated by producers.

Both can also handle very large amounts of messages. But because they handle those messages in different ways, each is best suited for subtly different use cases.

Apache Kafka Use Cases

Some of the best Kafka use cases make use of the platform's high throughput and stream processing capabilities.

High-throughput activity tracking: Kafka can be used for a variety of high-volume, high-throughput activity-tracking applications. For example, you can use Kafka to track website activity (its original use case), ingest data from IoT sensors, monitor patients in hospital settings, or keep tabs on shipments.

Stream processing: Kafka enables you to implement application logic based on streams of events. You might keep a running count of types of events or calculate an average value over the course of an event that lasts several minutes. For example, if you have an IoT application that incorporates automated thermometers, you could keep track of the average temperature over time and trigger alerts if readings deviate from a target temperature range.

Event sourcing: Kafka can be used to support event sourcing, in which changes to an app state are stored as a sequence of events.

So, for example, you might use Kafka with a banking app. If the account balance is somehow corrupted, you can recalculate the balance based on the stored history of transactions.

Log aggregation: Similar to event sourcing, you can use Kafka to collect log files and store them in a centralized place. These stored log files can then provide a single source of truth for your app.

RabbitMQ Use Cases

Some of the best RabbitMQ use cases make use of its flexibility—both for routing messages within microservices architectures and among legacy apps.

Complex routing: RabbitMQ can be the best fit when you need to route messages among multiple consuming apps, such as in a microservices architecture. RabbitMQ consistent hash exchange can be used to balance load processing across a distributed monitoring service, for example. Alternate exchanges can also be used to route a portion of events to specific services for A/B testing.

Legacy applications: Using available plug-ins (or developing your own), you can deploy RabbitMQ as a way to connect consumer apps with legacy apps. For example, you can use a Java Message Service (JMS) plug-in and JMS client library to communicate with JMS apps.

Streamline Kafka Deployment and Management with Instaclustr

At Instaclustr we are passionate about distributed, fault-tolerant, scalable, open source technologies. That's why we provide Kafka as a fully hosted and managed service, as well as support and consulting for Kafka.

Whether you choose a cloud deployment or decide to manage a Kafka cluster in your own data center, we can provide 24×7 expert

technical support with strict SLAs. In addition, our consulting services can help you address any challenges you might encounter throughout design, implementation, and operation. You can tap into our deep expertise drawn from more than 100 million node hours under management. We can help you build a robust application that fully capitalizes on this fast, scalable, distributed streaming platform.

Learn more about the Instaclustr fully managed, hosted service for Kafka, Kafka support, and Kafka consulting or see Instaclustr Pricing Here.