Understanding When to use RabbitMQ or Apache Kafka

APRIL 26, 2017 | PIETER HUMPHREY

How do humans make decisions? In everyday life, emotion is often the circuit-breaking factor in pulling the trigger on a complex or overwhelming decision. But for experts making complex decisions that have long term consequences, it can't be pure impulse. High performers typically use the circuit breaker of "instinct", "gut feel" or other emotions only once their expert, unconscious mind has absorbed all the facts required to make a decision.

Today there are dozens of messaging technologies, countless ESBs, and nearly 100 iPaaS vendors in market.   Naturally, this leads to questions about how to choose the right messaging technology for your needs - particularly for those already invested in a particular choice. Do we switch wholesale? Just use the right tool for the right job? Have we correctly framed the job at hand for the business need? Given that, what is the right tool for me? Worse, an exhaustive market analysis might never finish, but due diligence is critical given the average lifespan of integration code.

This post endeavors give the unconscious, expert mind some even handed treatment to consider, starting with the most modern, popular choices today: RabbitMQ and Apache Kafka. Each has it's own origin story, design intent, uses cases where it shines, integration capabilities and developer experience. Origins are revealing about the overall design intent for any piece of software, and make good starting point. However it's important to note that in this article, my aim is to compare the two around the overlapping use case of message broker, less the "event store / event sourcing" use case, where Kafka excels today.

Origins

RabbitMQ is a “traditional” message broker that implements variety of messaging protocols. It was one of the first open source message brokers to achieve a reasonable level of features, client libraries, dev tools, and quality documentation. RabbitMQ was originally developed to implement **AMQP**, an open wire protocol for messaging with powerful routing features. While Java has messaging standards like JMS, it’s not helpful for non-Java applications that need distributed messaging which is severely limiting to any integration scenario, microservice or monolithic. With the advent of AMQP, cross-language flexibility became real for open source message brokers.

Apache Kafka is developed in Scala and **started out at LinkedIn** as a way to connect different internal systems. At the time, LinkedIn was moving to a more distributed architecture and needed to reimagine capabilities like data integration and realtime stream processing, breaking away from previously monolithic approaches to these problems. Kafka is well adopted today within the Apache Software Foundation ecosystem of products and is particularly useful in event-driven architecture.

Architecture and Design

RabbitMQ is designed as a general purpose message broker, employing several variations of point to point, request/reply and pub-sub communication styles patterns. It uses a smart broker / dumb consumer model, focused on consistent delivery of messages to consumers that consume at a roughly similar pace as the broker keeps track of consumer state. It is mature, performs well when configured correctly, is well supported (client libraries Java, .NET, node.js, Ruby, PHP and many more languages) and has dozens of plugins available that extend it to more use cases and integration scenarios.

/

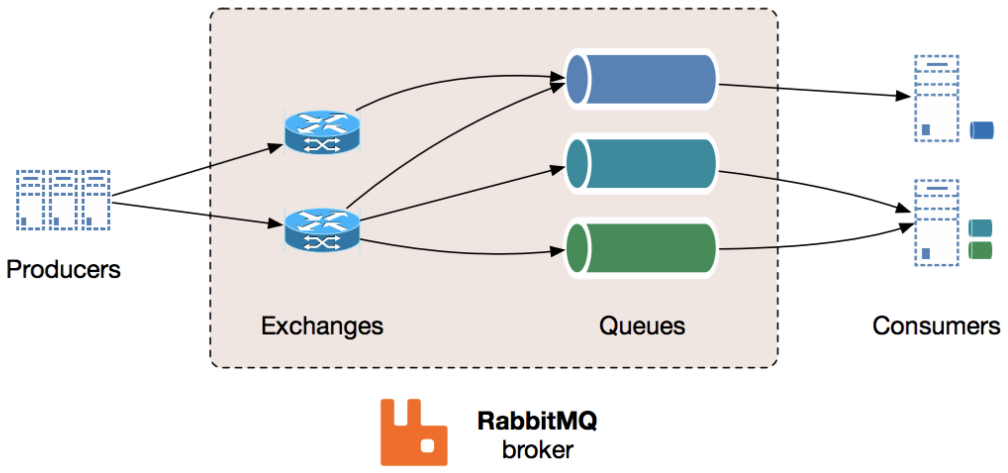

Figure 1 - Simplified overall RabbitMQ architecture. Source: http://kth.diva-portal.org/smash/get/diva2:813137/FULLTEXT01.pdf

Communication in RabbitMQ can be either synchronous or asynchronous as needed. Publishers send messages to exchanges, and consumers retrieve messages from queues. Decoupling producers from queues via exchanges ensures that producers aren't burdened with hardcoded routing decisions. RabbitMQ also offers a number of distributed deployment scenarios (and does require all nodes be able to resolve hostnames). It can be setup for multi-node clusters to cluster federation and does not have dependencies on external services (but some cluster formation plugins can use AWS APIs, DNS, Consul, etcd).

Apache Kafka is designed for high volume publish-subscribe messages and streams, meant to be durable, fast, and scalable. At its essence, Kafka provides a durable message store, similar to a log, run in a server cluster, that stores streams of records in categories called topics.

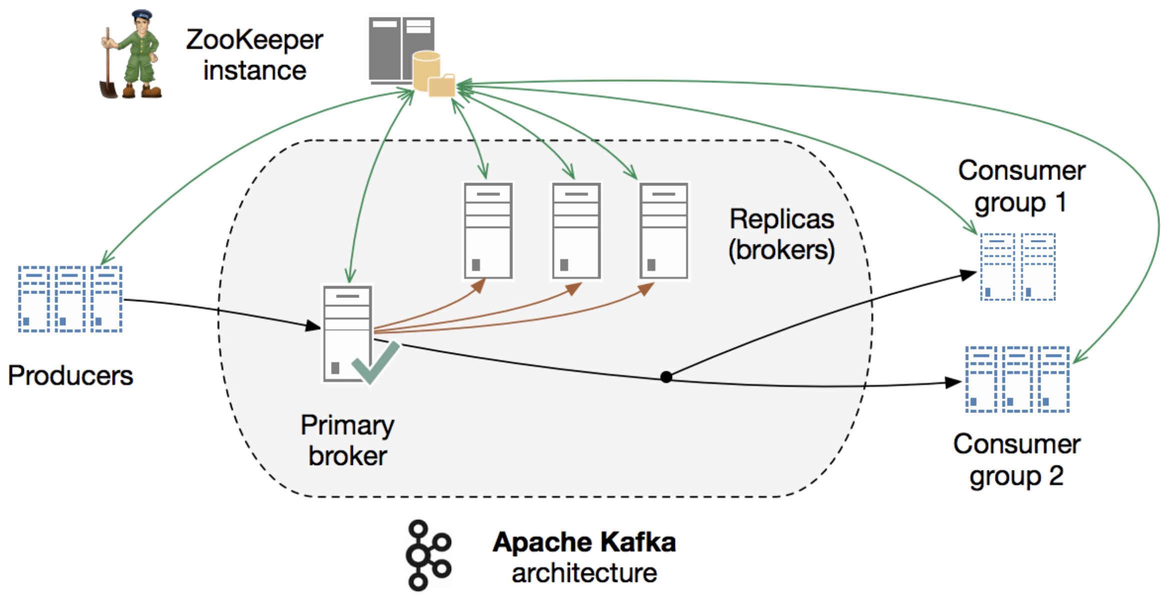

Figure 2 - Global Apache Kafka architecture (with 1 topic, 1 partition, replication factor 4). Source: http://kth.diva-portal.org/smash/get/diva2:813137/FULLTEXT01.pdf

Every message consists of a key, a value, and a timestamp. Nearly the opposite of RabbitMQ, Kafka employs a dumb broker and uses smart consumers to read its buffer. Kafka does not attempt to track which messages were read by each consumer and only retain unread messages; rather, Kafka retains all messages for a set amount of time, and consumers are responsible to track their location in each log (consumer state). Consequently, with the right developer talent creating the consumer code, Kafka can support a large number of consumers and retain large amounts of data with very little overhead. As the diagram above shows, Kafka does require external services to run - in this case Apache Zookeeper, which is often regarded as non-trivial to understand, setup and operate.

Requirements and Use Cases

Many developers begin exploring messaging when they realize they have to connect lots of things together, and other integration patterns such as shared databases are not feasible or too dangerous.

Apache Kafka includes the broker itself, which is actually the best known and the most popular part of it, and has been designed and prominently marketed towards stream processing scenarios. In addition to that, Apache Kafka has recently added Kafka Streams which positions itself as an alternative to streaming platforms such as Apache Spark, Apache Flink, Apache Beam/Google Cloud Data Flow and Spring Cloud Data Flow. The documentation does a good job of discussing popular **use cases** like Website Activity Tracking, Metrics, Log Aggregation, Stream Processing, Event Sourcing and Commit logs. One of those use cases it describes is messaging, which can generate some confusion. So let's unpack that a bit and get some clarity on which messaging scenarios are best for Kafka for, like:

- Stream from A to B without complex routing, with maximal throughput (100k/sec+), delivered in partitioned order at least once.
- When your application needs access to stream history, delivered in partitioned order at least once. Kafka is a durable message store and clients can get a "replay" of the event stream on demand, as opposed to more traditional message brokers where once a message has been delivered, it is removed from the queue.

- Stream Processing
- Event Sourcing

RabbitMQ is a general purpose messaging solution, often used to allow web servers to respond to requests quickly instead of being forced to perform resource-heavy procedures while the user waits for the result. It's also good for distributing a message to multiple recipients for consumption or for balancing loads between workers under high load (20k+/sec). When your requirements extend beyond throughput, RabbitMQ has a lot to offer: **features for reliable delivery**, routing, federation, HA, security, management tools and **other features**. Let's examine some scenarios best for RabbitMQ, like:

- Your application needs to work with any combination of existing protocols like AMQP 0-9-1, STOMP, MQTT, AMQP 1.0.
- You need a finer-grained consistency control/guarantees on a per-message basis (dead letter queues, etc.) However, Kafka has recently added better support for **transactions**.
- Your application needs variety in point to point, request / reply, and publish/subscribe messaging
- Complex routing to consumers, integrate multiple services/apps with non-trivial routing logic

RabbitMQ can also effectively address several of Kafka's strong uses cases above, but with the help of additional software. RabbitMQ is often used with Apache Cassandra when application needs access to stream history, or with the LevelDB plugin for applications that need an "infinite" queue, but neither feature ships with RabbitMQ itself.

For a deeper dive on microservice - specific use cases with Kafka and RabbitMQ, head over to the Pivotal blog and **read this short post by Fred Melo**.

Developer Experience

RabbitMQ officially supports Java, Spring, .NET, PHP, Python, Ruby, JavaScript, Go, Elixir, Objective-C, Swift - with many other **clients and devtools** via community plugins. The RabbitMQ client libraries are mature and well documented.

Apache Kafka has made strides in this area, and while it only ships a Java client, there is a growing catalog of community **open source clients**, **ecosystem projects**, and well as an adapter SDK allowing you to build your own system integration. Much of the configuration is done via .properties files or programmatically.

The popularity of these two options has a strong influence on many other software providers who make sure that RabbitMQ and Kafka work well with or on their technology.

As for developer experience…it's worth mentioning the support that we provide in **Spring Kafka**, **Spring Cloud Stream**, etc.

Security and Operations

Both are strengths of RabbitMQ. RabbitMQ management plugin provides an HTTP API, a browser-based UI for management and monitoring, plus CLI tools for operators. External tools like CollectD, Datadog, or New Relic are required for longer term monitoring data storage. RabbitMQ also provides API and tools for monitoring, audit and application troubleshooting. Besides support for TLS, RabbitMQ ships with RBAC backed by a built-in data store, LDAP or external HTTPS-based providers and supports authentication using x509 certificate instead of username/password pairs. Additional authentication methods can be fairly straightforwardly developed with plugins.

These domains pose a challenge for Apache Kafka. On the security front, the recent Kafka 0.9 release added TLS, JAAS role based access control and kerberos/plain/scram auth, using a CLI to manage security policy. This made a substantial improvement on earlier versions where you could only lock down access at the network level, which didn’t work well for sharing or multi-tenancy.

Kafka uses a management CLI comprised of shell scripts, property files and specifically formatted JSON files. Kafka Brokers, Producers and Consumers emit metrics via Yammer/JMX but do not maintain any history, which pragmatically means using a 3rd party monitoring system. Using these tools, operations is able manage partitions and topics, check consumer offset position, and use the HA and FT capabilities that Apache Zookeeper provides for Kafka. While many view the requirement for Zookeeper with a high degree of skepticism, it does confer clustering benefits for Kafka users.

For example, a 3-node Kafka cluster the system is functional even after 2 failures. However if you want to support as many failures in Zookeeper you need an additional 5 Zookeeper nodes as Zookeeper is a quorum based system and can only tolerate N/2+1 failures. These obviously should not be co-located with the Kafka nodes - so to stand up a 3 node Kafka system you need ~ 8 servers. Operators must take the properties of the ZK cluster into account when reasoning about the availability of any Kafka system, both in terms of resource consumption and design.

Performance

Kafka shines here by design: 100k/sec performance is often a key driver for people choosing Apache Kafka.

Of course, message per second rates are tricky to state and quantify since they depend on so much including your environment and hardware, the nature of your workload, which delivery guarantees are used (e.g. persistent is costly, mirroring even more so), etc.

20K messages per second is easy to push through a single Rabbit queue, indeed rather more than that isn't hard, with not much demanded in the way of guarantees. The queue is backed by a single Erlang lightweight thread that gets cooperatively scheduled on a pool of native OS threads - so it becomes a natural choke point or bottleneck as a single queue is never going to do more work than it can get CPU cycles to work in.

Increasing the messages per second often comes down to properly exploiting the parallelism available in one's environment by doing such things as breaking traffic across multiple queues via clever routing (so that different queues can be running concurrently). When RabbitMQ achieved **1 million message per second** , this use case basically came down entirely to doing that judiciously - but was achieved using lot of resources, around 30 RabbitMQ nodes. Most RabbitMQ users enjoy excellent performance with clusters made up of anywhere from three to seven RabbitMQ nodes.

Making the call

Absorb some research on a few of the other top options on the market. If you want to go deeper with the most popular options, a **master's thesis from Nicolas Nannoni** inspired this article and it features a side-by-side comparison table in Section 4.4 (page 39) but is a bit dated at this point. If you feel like plunking down $15.00 USD, this ACM **report** is also excellent and more recent.

While researching, loop back with the stakeholders and the business as often as possible. Understanding the business use case is the single largest factor in making the right choice for your situation. Then, **if you are pop psychology fan**, your best bet is sleep on it, let it percolate, and let your instincts take over. You got this.

Learn More

- Webinar: **Boosting Microservice Performance with Kafka, RabbitMQ, and Spring**
- Blog: **Operationalizing Apache Kafka on Kubernetes: Pivotal and Confluent Team Up**
- Webinar: **10 Things Every Developer Using RabbitMQ Should Know**
- Watch: **Implementing Raft in RabbitMQ**

About the Author

Pieter Humphrey is a Product Marketing Manager responsible for Java Developer Marketing at Pivotal Software, Inc. Pieter comes from BEA/Oracle with long history of developer tools, Java EE, SOA, EAI, application server and other Java middleware as both a marketing guy and sales engineer since 1998. Find me on Twitter at **https://www.twitter.com/pieterhumphrey**.