Diving into Open Source Messaging: What Is Kafka?

The world of messaging middleware has changed dramatically over the last 30 years. But in truth the world of communication has changed dramatically as well. The options available for application development and application-to-application communication are nearly endless. From open source to proprietary solutions, choosing the right communications paradigm can seem daunting. This paper focuses on one open source option for messaging middleware, Apache Kafka®, some of its strengths and how best to use it in an enterprise architecture.

INTRODUCTION

In a not so distant past, we humans only had a few basic ways of communicating. Sure, many of us spoke different languages, but the means of communication was relatively simple. Up until the late 19th Century there really were only two main forms of communication, verbal (in person) and written. Written could be transmitted around the globe, albeit rather slowly, with the added benefit of persistence. Because the options were limited, the mode was typically a simple choice. The marketplace became the area where information was shared, and libraries became the repository so that future generations could access and retrieve information.

Over the last 150 years, communication has changed rapidly. Today we have Snapchat, Instagram, broadcast media, YouTube, and podcasts. The truth is, every person I know has had that moment when their phone rings and the first thought is "Man, I really don't want to talk to them, I'll just text them later."

Messaging middleware is no different. In the early days, application-to-application communication was limited to sockets, everyone spoke different languages (protocols), and communication media was relatively uniform. Anyone who didn't want to take on the burden of writing their own communications layer looked to messaging middleware, and early on, the options where few and all proprietary protocols developed for a specific purpose. IBM had MQ, TIBCO had Rendezvous®, and web and mobile didn't exist.

Today however the options for application-to-application communications are nearly endless and can be rather confusing when it comes to selecting the right approach. Aside from proprietary middleware solutions provided by TIBCO, IBM, Oracle, etc, many open source options support a wide variety of protocols, like Apache Kafka, Eclipse Mosquito (MQTT), Qpid, or RabbitMQ (AMQP), not to mention other approaches like traditional database communication, distributed caching, and Restful APIs.

Today's middleware offerings are more robust, more functional, and in truth, blur the lines of what is the right choice for the task at hand. Open source offerings today are fully capable of meeting, and in many cases exceeding, the performance and functional requirements available in commercially available offerings. So it's time we start looking at some of these offerings more seriously, dive deeper into the technology, and provide ways to integrate them into robust enterprise architectures.

The rest of this paper offers a glimpse into Apache Kafka highlighting the strengths of Kafka's offerings, providing some architectural guidance about when best to use Kafka for messaging, and offerng some insight into how best to integrate Apache Kafka with existing messaging infrastructure and enterprise architectures.

APACHE KAFKA

Apache Kafka is all the rage right now, and rightfully so. Originally designed by LinkedIn and now under the Apache project, Kafka's real strength comes from its simple architectural approach to solving the problem of message distribution. Kafka's approach to messaging is more focused on log distribution, which means that the server is fairly simple and straight forward.

Kafka's original design was meant to provide a simple message-based log aggregator, think of it as a next generation syslog but with far greater reach and flexibility for log distribution, aggregation, and management. Kafka has followed a common publish/subscribe paradigm that is well understood in messaging solutions allowing for publishers to publish to topics that are then stored in a broker and ultimately delivered (pulled from the broker) to one or more consumers that subscribe to topics for data delivery.

This straight forward design allows for flexibility in the architecture and design of the application space using Kafka as the message streaming platform. Applications simply publish and subscribe to topics, and topic partitions (more on this later), while the brokers handle distribution based on interest. Since Kafka is distributed in nature, servers can be added to provide additional scale. The beauty of the Kafka design lives in the predicable approach to how it stores and delivers messages from the broker to consumers.

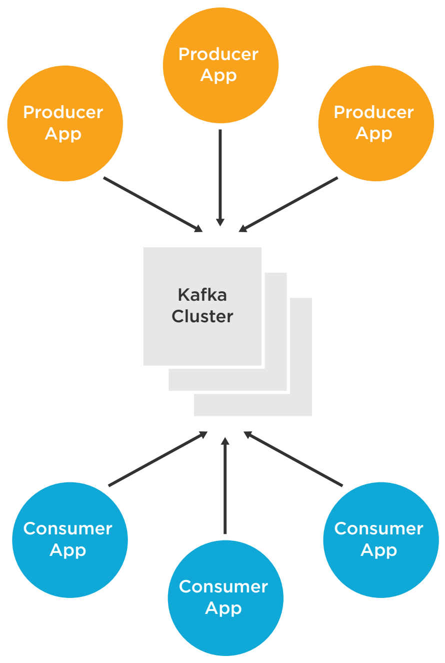

Figure 1: Basic Kafka Architecture

The concepts that Kafka uses for data distribution, publish/subscribe, and topic based distribution have been around for well over 30 years. The challenge that many messaging middleware solutions (including TIBCO's messaging solutions) face, however, is that due to the requirements of the many varied applications using middleware, the simple concepts of publish/subscribe messaging have become bloated and in many ways cumbersome to use. So while many of these proprietary messaging solutions like TIBCO Enterprise Message Service™ and IBM MQ Series all offer significantly more features, functions, and options, like transactional support, for the majority of application use cases, the additional functionality adds complexity that in many cases is not needed or desired.

This is where the true power of Apache Kafka comes into play. Being specifically built for log distribution and monitoring of front-end services at LinkedIn, the solution is built and designed for that purpose. This approach puts Kafka in a unique position in the messaging middleware solutions set as it has avoided, at least at present, falling into the one-size fits all category. Kafka's design and feature set are all built around extending its purpose-built architecture.

Take for example the Kafka storage model. Being designed for log distribution, each topic partition maps to a logical physical log. Architectural implementation of the storage solution is simple and straight forward. Anytime a publisher publishes a message to a given topic partition, the broker appends the message to the topic partitions physical log. This model allows for messages to be indexed by their offset in the message log versus a traditional approach using a message ID to provide indexing and message lookup. This reduces complexity, and more importantly, reduces state management compared to other broker-based messaging systems.

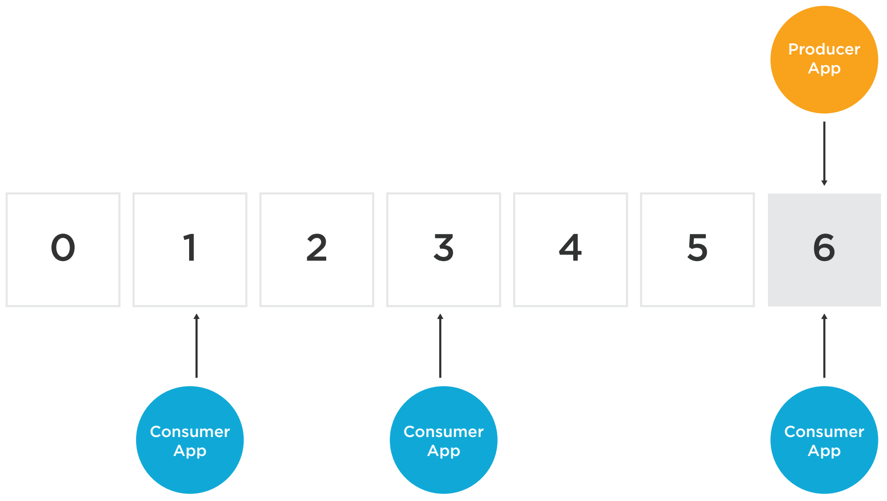

Figure 2: Kafka Message Storage

Consumers are managed in much the same way as producers, such that consumers are designed to consume messages sequentially from a given topic partition. Since sequential consumption is built into the architecture, consumers can acknowledge all messages received by simply acknowledging the last message received in the sequence. In addition, Kafka brokers do not maintain any information about consumer consumption. This allows for brokers to be stateless by design, and messages are purged after a configurable time period. This means new consumers coming online can replay history and/or existing consumers can rewind and re-consume data on demand.

Since Kafka treats the topic stream like a log, the only information retained in the Kafka server is the per-consumer offset. This means that where a consumer is in the process stream is maintained in the Kafka server, but unlike other server based messaging solutions, the rest of the metadata around the consumer is held in the consumer application. This provides a lightweight and fast way to store and retrieve data.

The Kafka server persists message data based on a configurable retention period. This is what allows consumers to replay and or catch up to the data stream based on how long the administrator has configured the stream to persist data. For example, if the application requires that seven days worth of message data be available to a consumer, the administrator of the Kafka system can configure a seven day retention period allowing for the Kafka server to store up to seven days worth of published data on the given topic and be available for consumption up to seven days after publishing. After this retention period, the message data is discarded and the space allocated for this message is reclaimed.

Now one of the more heated discussions around Apache Kafka is the usage of Zookeeper for server and consumer coordination. As you can see, the Kafka designers made early design decisions to provide system level efficiencies for the data distribution patterns Kafka was intended to address. Kafka brokers were designed to be lightweight, stateless message distributors, and therefore to offload all aspects of system coordination and management. To this end, Kafka uses ZooKeeper to maintain broker node and consumer node awareness, trigger node rebalancing when brokers or consumers are added/removed, and to track and register consumption log information between consumer groups within the system.

While usage of ZooKeeper adds an additional layer of complexity, ZooKeeper like Kafka is specifically designed for this purpose. Its simple file system-like API provides for a path-based broker and consumer registry that is easy to monitor and maintain state information in.

Lastly Apache Kafka, like many messaging solutions, provides message delivery guarantees. Unlike many of the messaging middleware solutions that provide multiple unique delivery models (JMS for example), Kafka provides a simple defined model of at-least-once delivery. This means that applications normally will get a message once and will be guaranteed to get the message at least once, but circumstance can be found where messages can be delivered to an application multiple times. Again Kafka has taken the approach of less is more and has simply defined this delivery model as the sole approach and provided architectural guidelines for managing duplicate detection in the consumer application.

KAFKA CONNECT

Building on the simple approach and design that has made Kafka so attractive, the Kafka Connect toolkit is a flexible and scalable approach to providing integration with other third party systems. Kafka Connect defines a connector as the ingress or egress point of data. It defines a common framework that provides the integration point for third-party systems to interact with the core Kafka messaging system.

Kafka Connect, like Kafka, is designed to provide a simple, scalable approach to integration. Because of this, Kafka Connect is purpose built to act as a data pump into and out of the Kafka core messaging system. This architectural design provides for a purpose built, lightweight integration layer that can be scaled and deployed as integration requirements change.

To keep the integration simple and straight forward, Kafka Connect uses the concept of connectors as the data pump ingress and egress points. For importing data into Kafka, a source-connector is used, and for exporting data out of Kafka, a sink-connector is used.

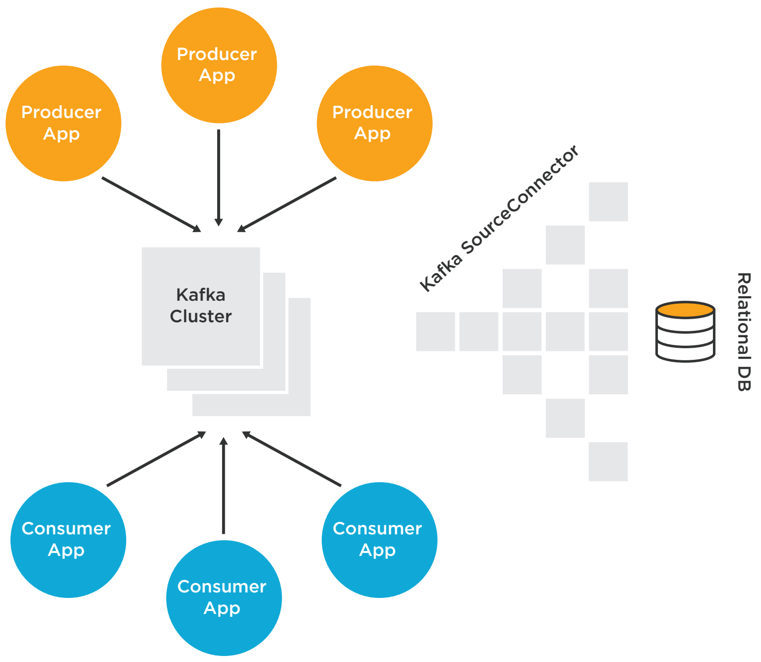

Figure 3: Kafka Connect Source Connector

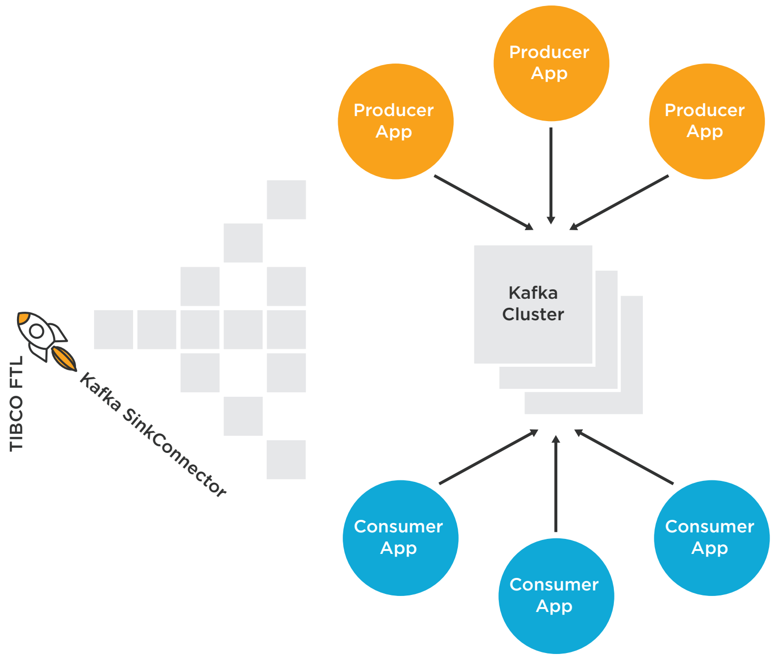

Figure 4: Kafka Connect Sink Connector

By separating import and export operations into two channels, the Kafka Connect interface is simplified as the Kafka core becomes agnostic to the usage of Kafka Connect. In addition, the Kafka core does not need to handle how data is marshaled into or out of Kafka. The logic and processing of this happens within Kafka Connect, and more importantly, is handled in the purpose built Connector for the given integration point. This architecture provides for a simple and flexible approach to integrating between third-party systems. Kafka Connect Source and Sink connectors become plugins for integration points, and those connectors look and behave just like Kafka publishers and subscribers.

APACHE KAFKA CONCLUSIONS

Apache Kafka is an extremely powerful, lightweight, distributed messaging platform. It allows for flexible application development, and its core strength lies in the simple and defined approach it takes to messaging middleware. Is Kafka right for every situation? Probably not, but if the application developer is willing to take on a little more responsibility in the application space, Kafka becomes a very versatile tool in the messaging toolkit.

In addition, with lightweight, flexible integration using Kafka Connect, Kafka becomes a very powerful tool in integration between disparate components within the infrastructure. Using Kafka and Kafka Connect, a enterprise architect can provide simple, flexible, data distribution that can standalone or directly integrate with other third-party systems.

This is what is driving the power and the allure of Kafka. Earlier we discussed where messaging middleware came from, and even in the early days the nirvana state for messaging was to provide a common communications infrastructure that allowed application developers the ability to connect and exchange information seamlessly. Over the years there have been numerous attempts at coming to this state of Nirvana, and let's be honest, all have fallen short. From JMS to AMQP, no one solution for communication is going to fit all the differing demands for data distribution. Kafka is a great solution for data distribution, and for many use cases, it fits very well. Does it fit all uses cases? No, and it wasn't designed to. That is what makes Kafka such a great tool: it knows what it was designed to do, it does it well, and it lets developers tie in other solutions where appropriate.

Here at TIBCO, we have a new mantra about being Better Together. Better Together means that our priorities have a different focus. It means that it is not about the products we build, it is about the problems we solve, and to solve the communications challenges that are facing developers today, we at TIBCO Messaging know that we can and are better together.

Apache Kafka is now a fully supported and maintained solution in TIBCO's Messaging portfolio. So if your messaging needs are primarily data propagation, like logging, monitoring, management information, or signaling information, Kafka is a simple, straight forward approach that allows for relatively high performance message delivery in a simple, easy-to-use package—and TIBCO is now supporting all the open source aspects of Apache Kafka Core. In addition, TIBCO provides native bridging functionality into existing and future TIBCO environments.

Better together means using the best tool for the job, and TIBCO supporting Apache Kafka, MQTT, and other open source solutions, means TIBCO is ready, willing, and able to provide the best tool for the job and help you use those tools in the best way possible.

For more information on TIBCO's Messaging solutions and how TIBCO can support your usage of Apache Kafka or other open source projects, please visit: https://www.tibco.com/products/tibco-messaging.

Global Headquarters
3307 Hillview Avenue
Palo Alto, CA 94304
+1 650-846-1000 TEL
+1 800-420-8450
+1 650-846-1005 FAX
www.tibco.com