1. Getting to Know Apache Kafka's Architecture

Contents

[1. Apache Kafka as a Messaging System 1](#_Toc2164)

[2. The Apache Kafka Cluster 2](#_Toc26093)

[3. Principles of Distributed Systems 2](#_Toc1188)

[4. Reliable Work Distribution 3](#_Toc6213)

[5. Distributed Consensus with Apache Zookeeper 4](#_Toc3716)

# Apache Kafka as a Messaging System

=>slides: Pg. 1

This module will start by examining the overall architecture of Apache Kafka.

=>slides: Pg. 2

As I highlighted in the first module, Apache Kafka is truly a messaging system. More specifically, it is a publish subscribe messaging system in a pub subsystem, there are publishers of messages and subscribers of messages. A publisher creates some data and sends it to a specific location where an interested and authorized subscriber can retrieve the message and process it. In Kafka, we call these traditional publishers something slightly different ‑ producers, and the subscribers we call ‑ consumers. As we'll see in the upcoming modules, producers and consumers are simply applications that you write or use to implement the producing and consuming APIs.

=>slides: Pg. 3

Now the producer sends its messages to what I said was a specific location. In Kafka, this location is referred to as a topic, which is really a collection or grouping of messages. Topics have a specific name that can be defined up front or on demand as long as producers know the topic name and have permission to send to it, messages can be sent to that specific location. The same goes for consumers. Consumers retrieve messages based on the topic it is interested in. This should be very familiar for those with experience using pub sub messaging. As these concepts were intentionally carried forward into Kafka due to the simplicity of the paradigm.

=>slides: Pg. 4

The messages and their topics need to be kept somewhere, after all, they are physical containers of data. The place where Kafka keeps and maintains topics is called the broker, as it is in other messaging systems. Let's look closer at the Kafka broker.

=>slides: Pg. 5

The Kafka broker is a software process also referred to as an executable or daemon service that runs on a machine, a physical machine or a virtual machine. A synonym for a broker is also a server, but I like to avoid using the term server, since it has a tendency to be overloaded. The broker has access to resources on the machine, such as the file system, which it uses to store messages, which it categorizes as topics. Like any executable, you can run more than one on a machine, but each must have unique settings so that they don't conflict with one another. It is in the Kafka broker, where the differences between other messaging systems become apparent.

# The Apache Kafka Cluster

**=**>slides: Pg. 6

How the Kafka broker handles messages in their topics is what gives Kafka its high throughput capabilities. Achieving high throughput is largely a function of how well a system can distribute its load and efficiently process it on multiple nodes in parallel. This is a hallmark of scalable design. With Apache Kafka, you can see scale out the number of brokers as much as needed to achieve the levels of throughput required. And all of this without affecting existing producer and consuming applications. In fact, LinkedIn as of April of 2016 has publicly stated that they have upwards of 1400 brokers processing over two petabytes per week. You simply can't find another messaging system out there with that capability. Let's continue to dive deeper and understand what about Kafka's architecture enables it to scale so well and furthermore, how it can achieve such high levels of reliability.

**=**>slides: Pg. 7

It's time to introduce another key concept within the Apache Kafka architecture and that is the cluster. A Kafka cluster is a grouping of multiple Kafka brokers. As I said, you can have a single broker or multiple brokers on a single machine or brokers on different machines.

**=**>slides: Pg. 8

For example, if you had only a single broker on a machine and only one machine, it would be said that you have a cluster of one.

**=**>slides: Pg. 9

If you have two brokers running on the same machine, it would be considered a cluster of two.

**=**>slides: Pg. 10

This would be the same if each broker was running on its own machine, you still would have a cluster of two.

**=**>slides: Pg. 11

The important thing to remember here is that a Kafka cluster is just a grouping of brokers. It doesn't matter if they're running on their own machines or not. What matters is how independent brokers are grouped to form a cluster. The grouping mechanism that determines a cluster's membership of brokers is an important part of Kafka's architecture, and what really enables its ability to scale to thousands upon thousands of brokers and be distributed in a fault tolerant way. For the sake of putting down a placeholder, this is where Apache ZooKeeper comes in

# Principles of Distributed Systems

**=**>slides: Pg. 12

At this point, it is important to discuss a little theory about distributed systems. But along the way I will do my best to associate the principles to Apache Kafka. So you'll understand how brokers organize into clusters and how Kafka clusters achieve amazing feats of scalability and reliability. First, I'm going to give you my general definition of a distributed system for the purposes of describing how Kafka, as a distributed system, works without spending too much time discussing philosophy. Let's just say a system is a collection of resources that have instructions to achieve a specific goal or function. A distributed system is one that consists of multiple independent resources, also known as workers or nodes, sometimes even called worker nodes. Obviously, the reason there are multiple nodes is to spread the work around, presumably to get more done than what could otherwise be achieved with less. But in order to do that, there needs to be coordination amongst all of the available working nodes to ensure consistency and optimal progress towards the overall task or goal at hand. Without coordination, it would be difficult, if not outright chaotic, to divide up the work appropriately and track progress as to ensure the most optimal use of resources, for example, proper coordination can avoid duplication of effort or individual worker nodes, undermining each other's work without knowing it. Of course, coordination isn't possible without clear and thorough communication between all components within the system. In Kafka, these worker nodes are the Kafka brokers.

**=**>slides: Pg. 13

Within a distributed system, there are different roles and responsibilities. And like any organization of able workers, there is generally a hierarchy that starts with a controller or supervisor. A controller is really just a worker node like any other. It just happened to be elected from amongst its peers to officiate in the administrative capacity of a controller. In fact, the worker node selected is the controller is commonly the one that's been around the longest. But other factors could be considered as well. Once selected, the controller has some critical responsibilities. First, to maintain an inventory of what workers are available to take on work. Second, to maintain a list of work items that has been committed to and assigned to workers, and third to maintain active status of the staff and their progress on assigned tasks.

**=**>slides: Pg. 14

Once a controller is established and the workers are assigned and available, you could say a team is formed and work can now be distributed and executed. In Kafka, this team formation is the cluster, and its members are brokers that have assigned themselves to a designated controller within the cluster. **=**>slides: Pg. 15

When a task comes in as an example from a Kafka producer, the controller has to make a decision which workers should take it. There are a lot of factors at play here, many of which are beyond the scope of this module, but let's cover the most important ones.

# Reliable Work Distribution

**=**>slides: Pg. 16

First, the controller needs to know who is available and in good health. And very importantly, the controller needs to know what risk policy should govern its assignment decisions. A great example for risk policy is the redundancy level, the thing that determines what level of replication to employ in case an assigned worker fails.

**=**>slides: Pg. 17

In a distributed system that offers redundancy options, it has to ensure that if work is assigned to a worker and that worker becomes unavailable, the work assigned or the work already done is not lost. That means each task given to a worker must also be given to at least one of the worker's peers in the event of an unexpected catastrophe. But amongst a group of worker nodes, which one will get assigned? This is how it works. For an assignment, if the controller determines redundancy is required, it will promote a worker into a leader which will take direct ownership of the task assigned. It will be the leader's job to recruit two of its peers to take part in the replication. In Kafka, the risk policy to protect against loss is known as its replication factor, and we will cover this in more detail within the next module.

**=**>slides: Pg. 18

Once peers have committed to the leader, a quorum is formed, and these committed peers now take on a new role in relation to a leader, a follower. If, for whatever reason a leader cannot get a quorum, the controller will reassign tasks to leaders that can.

**=**>slides: Pg. 19

Up until this point in covering the principles of distributed systems and how Apache Kafka applies them, we have been using the term work somewhat generally. In Apache Kafka, the work that the cluster of brokers performs is receiving messages, categorizing them into topics and reliably persisting them for eventual retrieval. As we have already discussed, the components that drive this work are producing applications called producers that send the messages to the cluster and consuming applications, called consumers that retrieve and process messages. Comparatively speaking, the effort required to handle messages from the producers is substantially less than the effort required by consumers. We will talk about this more in modules four and five. I just wanted to bring this up to highlight that both add load to the cluster, but in very different ways

# Distributed Consensus with Apache Zookeeper

**=**>slides: Pg. 20

Virtually every component within a distributed system has to keep some form of communication going between themselves. In distributed computing terms, this communication is referred to as a consensus or gossip protocol, and without it distributed system simply cannot operate. Besides the obvious data payloads being transferred as messages There are other types of network communications happening that keep the cluster operating normally. For example, events related to brokers becoming available and requesting cluster membership or broker name lookups. Retrieving Bootstrap configuration settings and being notified of new settings consistently and in a timely fashion. Events related to controller and leader election and health status updates like heartbeat events.

**=**>slides: Pg. 21

It is time to talk about Apache ZooKeeper. I have to give a big disclaimer, though Apache ZooKeeper could use its own course, so we can't spend a lot of time on it here. It is used in a variety of distributed systems. For all the reasons we have been discussing in the context of Apache Kafka, specifically, ZooKeeper serves as a centralized service for metadata about vast clusters of distributed nodes needing Bootstrap and runtime configuration information, health and synchronization status, and cluster and quorum group membership, including the roles of elected nodes. Some rather large distributed systems depend on ZooKeeper, like Apache Hadoop HBase, Mesos, Solr, as well as Redis and Neo4j database is they all use it to enable their scale out fault tolerant capabilities. ZooKeeper itself is a distributed system, and for it to run reliably has to have multiple nodes, which form what is called a ZooKeeper ensemble. An ensemble is like saying a cluster for ZooKeeper. In the case of Kafka, because of the type of work a ZooKeeper ensemble performs, it is generally not needed to have more than one ensemble to power one or many Kafka clusters.

**=**>slides: Pg. 22

Let's bring it all together in a logical view. At the heart of Apache Kafka, you have a cluster which, as we discussed, consists of possibly hundreds of independent brokers closely associated with the Kafka cluster. You have a ZooKeeper environment which provides the brokers within a cluster, the metadata it needs to operate at scale and reliably. As this metadata is constantly changing, connectivity and chatter between the cluster members and ZooKeeper is required. Of course, the cluster doesn't do much unless if you put it to work and that's where Kafka producers and consumer applications come in. Each of these components can scale out to take on more demand and increase levels of reliability and availability. I have given some incredible statistics about LinkedIn's Kafka environment, but Netflix is even more impressive. Over 4000 brokers across 36 clusters, processing over 700 billion messages per day. I hope with this foundation established, you will start to understand how. But we're not finished. We still have to dive deeper into the messaging internals, which happen inside the broker. And with that, learn more about Kafka topics and partitions.

**=**>slides: Pg. 23

In this module, we covered a lot of ground. We started by discussing how Kafka is a publish subscribe messaging system with common concepts as other messaging systems. But we started to differentiate Kafka with slightly different terminology, especially around producers and consumers that publish and receive messages respectively, and independent Kafka brokers and how they are grouped to form a cluster. We also introduced some fundamental characteristics of distributed systems, and we started to describe the worker node roles of controllers, leaders, peers and followers. Also, we discussed how distributed systems enforce and provide flexibility and reliability through replication policies. We rounded it off by highlighting the importance of consensus or gossip‑oriented communication within a distributed system and the critical role that ZooKeeper plays in that, and in enabling Kafka and other distributed systems to function scalably and reliably.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*