1. Apache Kafka® Abandons the Zoo(Keeper)

"Abandon Ship!": Kafka animals abandon the Zoo(Keeper) on a Kafka (K)Raft

(Source: Shutterstock)

For several years Kafka has been modified to remove the dependency on ZooKeeper for meta-data management, using a new protocol called KRaft. More details on the Kafka Raft protocol are here. "KRaft" stands for "Kafka Raft", and is a combination of Kafka topics and the Raft consensus algorithm. Here's a high-level explanation.

In a Kafka cluster, the Kafka Controller manages broker, topic, and partition meta-data (the "control plane")—it functions as Kafka's "Brain" as follows:

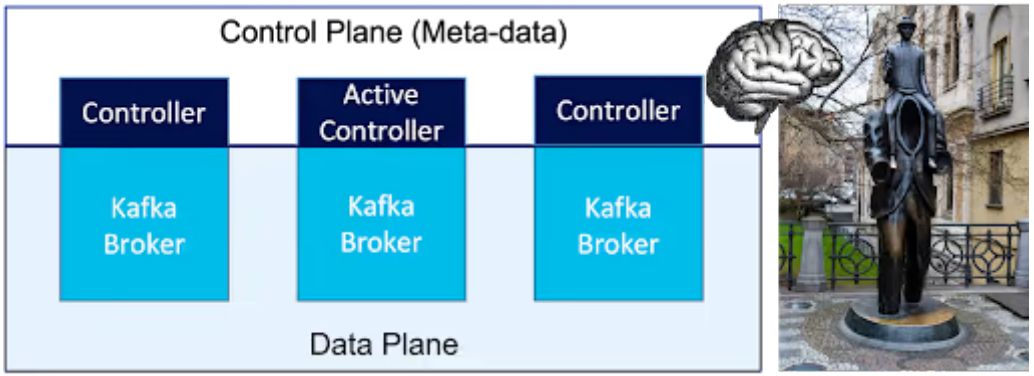


Diagram 1: The Kafka Control Plane

(Source: Shutterstock)

But how does Kafka know which controller should be active, and where the meta-data is stored in case of broker failures? Traditionally a ZooKeeper Ensemble (cluster) has managed the controller elections and meta-data storage as in this diagram:

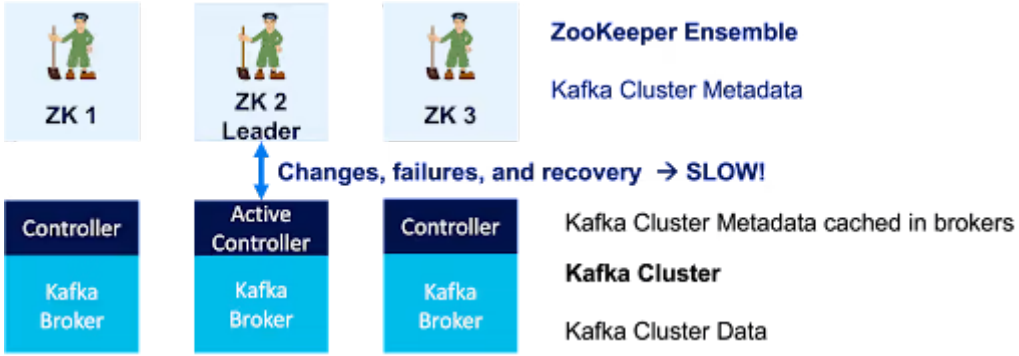


Diagram 2: The Kafka controller working with help from ZooKeeper

But this introduces a bottleneck between the active Kafka controller and the ZooKeeper leader, and meta-data changes (updates) and failovers are slow (as all communication is between one Kafka broker and one ZooKeeper server, and ZooKeeper is not horizontally write scalable). The new Kafka Raft mode (KRaft for short) replaces ZooKeeper with Kafka topics and the Raft consensus algorithm to make Kafka self-managed, as in this diagram:

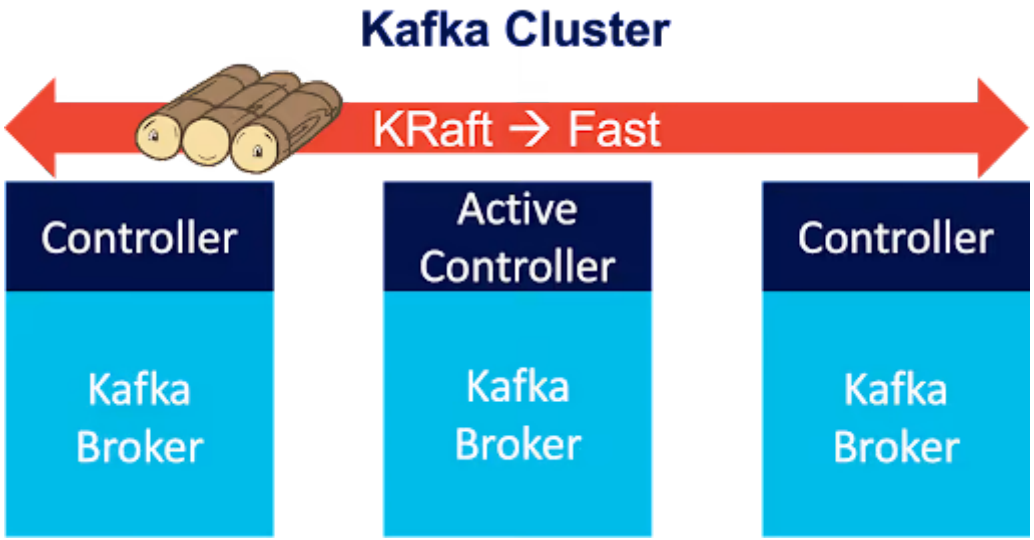


Diagram 3: The self-managed Kafka controller using KRaft

(Source: The Raft Mascot https://raft.github.io/logo/solo.svg)

The Kafka cluster meta-data is now only stored in the Kafka cluster itself, making meta-data update operations faster and more scalable. The meta-data is also replicated to all the brokers, making failover from failure faster too. Finally, the active Kafka controller is now the Quorum Leader, using Raft for leader election.

The motivation for giving Kafka a “brain transplant” (replacing ZooKeeper with KRaft) was to fix scalability and performance issues, enable more topics and partitions, and eliminate the need to run an Apache ZooKeeper cluster alongside every Kafka cluster.

The changes have been happening incrementally, starting from Kafka version 2.8 (released April 2021) which had KRaft in early access. In version 3.0 (September 2021) it was in preview. The maturity of KRaft has improved significantly since version 2.8 (3.0 had 20 KRaft improvements and bug fixes, 3.0.1 had 1, 3.1.0 had 5, and 3.1.1 had only 1 minor fix), and on October 3 2022 version 3.3 was marked as production ready (for new clusters).

On July 18, 2022 we announced the general availability of a new version of our managed Kafka service, Instaclustr for Apache Kafka version 3.1.1. With the general availability of our 3.1.1 managed service, we decided that it was a good time to evaluate the Kafka KRaft mode. However, given that KRaft is not production ready in 3.1.1, our managed service is still leveraging Apache ZooKeeper with a KRaft compatible version expected to be released in the near future.

In order to provide a fair comparison of ZooKeeper and KRaft, we decided to start with the managed service using ZooKeeper, and then modify it for KRaft. This means that everything is identical apart from the meta-data management mode in use. The instructions we used are here. Note that an extra step is required using the Kafa storage tool to format the disk before starting each node. Thanks to my colleague John Del Castillo for assistance with KRaft cluster configuration and testing.

2. Partitions vs. Producer Throughput

The first thing we wanted to understand about the change from ZooKeeper to KRaft was if there would be any impact on production workload message throughput for a cluster—particularly for producer throughput with increasing numbers of partitions. We previously conducted detailed benchmarking for an older version of Kafka in 2020 (2.3.1) which we published in one of our most popular blogs “The Power of Apache Kafka Partitions: How to Get the Most Out of Your Kafka Cluster”.

In this blog we revealed that the replication factor has a significant overhead on Kafka cluster CPU (RF=1 has no overhead, RF=3 has the most overhead), and that producer throughput is optimal for partitions between the number of cores in the cluster and 100, dropping off substantially for more partitions.

The cluster configuration back then was:

> 3 nodes x r5.xlarge (4 cores, 32GB RAM)—12 cores in total

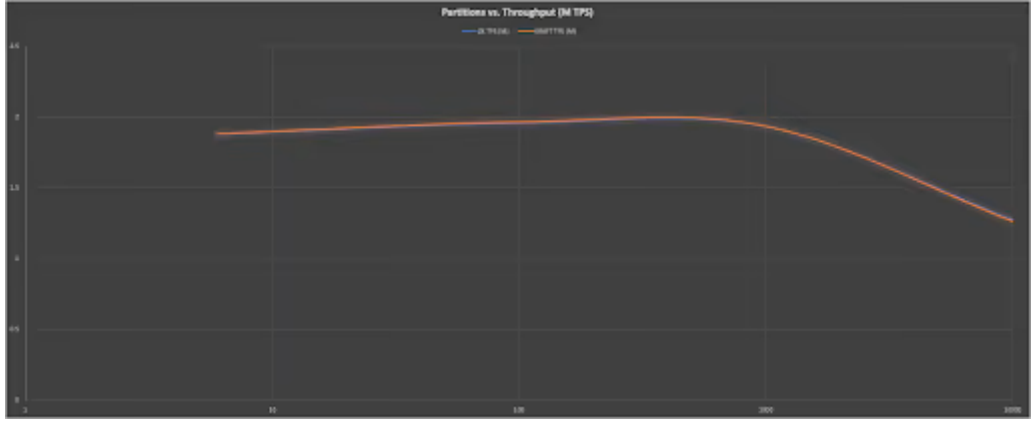
For the latest experiments, Kafka is deployed on the new AWS Graviton2 instance types:

> 3 nodes x r6g.large (2 cores, 16GB RAM)—6 cores in total

For the ZooKeeper mode, we used additional dedicated ZooKeeper nodes (m6g.large x 3). However, using the Instaclustr console monitoring it was obvious that there was no load on the ZooKeeper instances resulting from the producer load on the Kafka cluster, so for all intents and purposes the Kafka cluster resources are identical.

The benchmarking client was a single large EC2 instance running the kafka-producer-perf-test.sh command, with acks=all and 8 bytes per message. The methodology used was to create a single topic with increasing partitions, run the producer test to find the maximum throughput, delete the topic, and repeat up to 10,000 partitions.

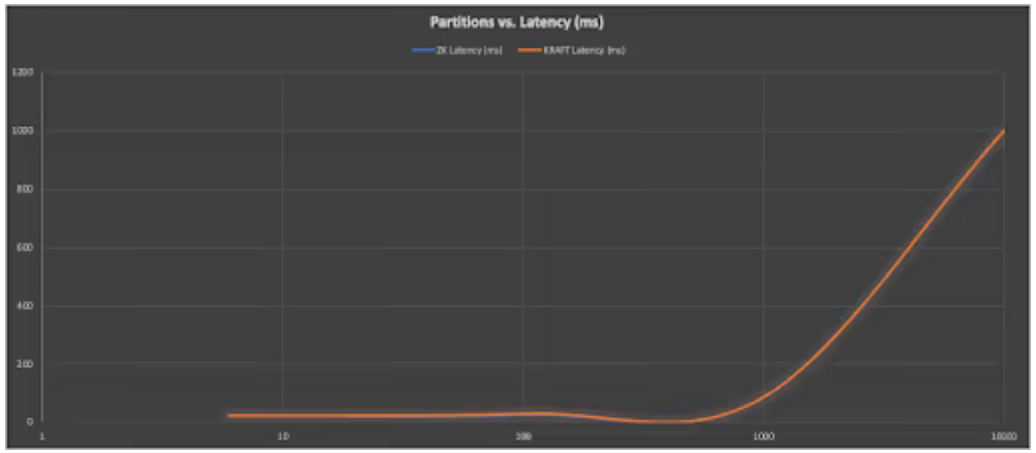
So here are the results, showing partitions (log x-axis) vs. throughput (millions of messages per second):



The maximum producer rate is 2 million messages/s at 100 partitions.

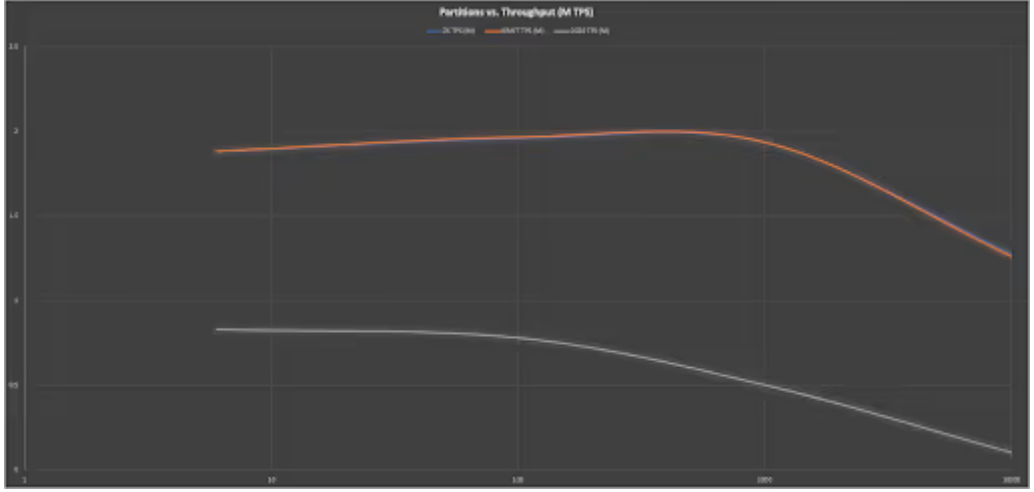
But the first surprise is that there's no difference between ZooKeeper and KRaft throughput (which is why only the orange KRaft line is visible in the graph). This is not unexpected, as Zookeeper and KRaft are only being used to manage the cluster meta-data, including nodes and partitions. They are not involved in the producers writing *data* to partitions, including replication. As expected, the throughput does eventually drop off with increasing partitions, at around 1000 partitions.

The next graph shows partitions (log x-axis) vs. producer latency (y-axis, ms):



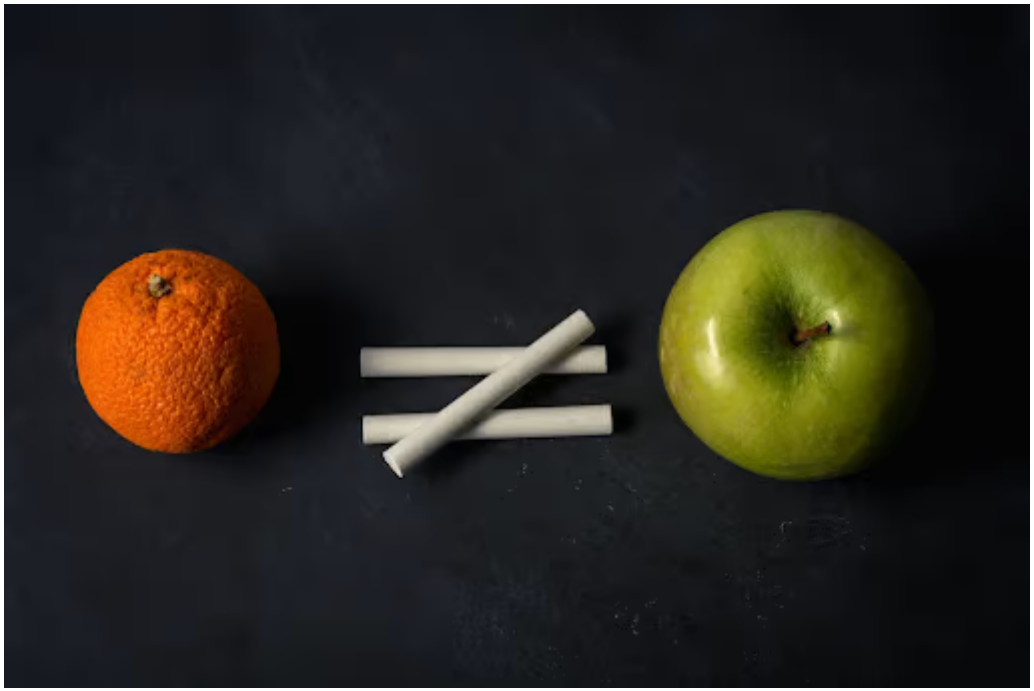
Given the drop-off in throughput obvious from the first graph, the rapid increase in latency around 1000 partitions is no surprise (and the results are again identical so only the KRaft line is visible).

But let's compare the partitions vs. throughput results with the results we obtained in 2020:



What do we notice? First, the absolute throughput is higher—2M messages/s now compared to 0.83M messages/s then—an improvement of 235%. Second, the scalability with increasing partitions is better—the throughput doesn't start to seriously drop until almost 1000 partitions, and the reduction in throughput from 100 to 10000 partitions is only 153%, compared to 780% previously.

What do we conclude from this? First of all, Kafka 3.1.1, whether using ZooKeeper or KRaft, has better throughput and scalability with increasing partitions compared to the previous version.



(Source: Shutterstock)

However, the comparison isn't really apples-to-apples as (1) the instances are different, and (2) the message size was different (8 bytes compared to 80 previously). We may revisit these tests in future blogs in this series, to better understand what factors may have resulted in these differences.

3. Conclusions So Far

These preliminary experiments and initial results give us confidence that Kafka 3.1.1 and above will support higher throughput and better scalability with increasing partitions, and more efficiently than previous versions and instance types—potentially supporting 10s of thousands of partitions in production, not just 100s of partitions. Also, Kafka with KRaft so far appears to be reliable and usable so this gives us more confidence to continue testing.

One of our goals is to attempt to create (and use!) a Kafka cluster with KRaft with 1 million or more partitions. However, we note that at least one challenge may be that a relatively large cluster will be needed. This is because, with these test clusters with 10,000 partitions, the background CPU was 50% with RF=3 and no load on the system (and no data in the partitions). This overhead is purely the result of the consumer's polling from the follower partitions to the leaders. For 1M partitions, we may need a cluster that is 100 times this size, i.e. 6 x 100 cores = 600 cores!

In the remainder of this Kafka ZooKeeper vs. KRaft blog series, we will investigate the performance of ZooKeeper vs. KRaft, meta-data management for partition creation and movement, and discover if there is any limit to how many partitions KRaft will support, by attempting to create a cluster with 1M or more partitions!