Learn how to launch an Apache Kafka with the Apache Kafka Raft (KRaft) consensus protocol, removing Kafka's dependency on Apache Zookeeper for metadata management. Before jumping into the technical part, let me give you some context as to why and how KRaft came into existence.

Why Kafka Without ZooKeeper?

Apache ZooKeeper has been an integral part of Kafka for distributed coordination and leadership election. While it served its purpose well, managing a ZooKeeper ensemble alongside Kafka added complexity to the deployment. Since the introduction of the KIP-500, it's now possible to run Kafka without ZooKeeper, simplifying the setup and maintenance. KRaft mode uses a new quorum controller service in Kafka, which replaces the previous controller, and it's an event-based variant of the Raft consensus protocol.

Zookeeper x Kraft - from https://developer.confluent.io/learn/kraft/

Prerequisites

An understanding of Apache Kafka, Kubernetes, and Minikube.

The following steps were initially taken on a MacBook Pro with 32GB memory running MacOS Ventura v13.4.

Make sure to have the following applications installed:

- Docker v23.0.5
- Minikube v1.29.0 (running K8s v1.26.1 internally)

It's possible the steps below will work with different versions of the above tools, but if you run into unexpected issues, you'll want to ensure you have identical versions. Minikube was chosen for this exercise due to its focus on local development.

Deployment Components

The deployment we will create will have the following components:

- Namespace: ***kafka***
  This is the namespace within which all components will be scoped.
- Service Account: ***kafka***
  Service accounts are used to control permissions and access to resources within the cluster.
- Headless Service: ***kafka-headless***
  It exposes ports 9092 (for Kafka clients) and 29093 (for Kafka Controller).
- StatefulSet: ***kafka***
  It manages Kafka pods and ensures they have stable hostnames and storage.

The source code for this deployment can be found in this GitHub repository.

Creating the deployment

Clone the repo:

```
git clone https://github.com/rafaelmnatali/kafka-k8s.git
```

deploy Kafka using the following commands:

```
kubectl apply -f 00-namespace.yaml
kubectl apply -f 01-kafka-local.yaml
```

Verify communication across brokers

There should now be three Kafka brokers each running on separate pods within your cluster. Name resolution for the headless service and the three pods within the StatefulSet is automatically configured by Kubernetes as they are created, allowing for communication across brokers. See the related documentation for more details on this feature.

You can check the first pod's logs with the following command:

```
kubectl logs kafka-0
```

The name resolution of the three pods can take more time to work than it takes the pods to start, so you may see *UnknownHostException* warnings in the pod logs initially:

```
WARN [RaftManager nodeId=2] Error connecting to node kafka-
1.kafka-headless.kafka.svc.cluster.local:29093 (id: 1 rack:
null) (org.apache.kafka.clients.NetworkClient)
java.net.UnknownHostException: kafka-1.kafka-
headless.kafka.svc.cluster.local         ...
```

But eventually each pod will successfully resolve pod hostnames and end with a message stating the broker has been *unfenced*:

```
INFO [Controller 0] Unfenced broker: UnfenceBrokerRecord(id=1,
epoch=176) (org.apache.kafka.controller.ClusterControlManager)
```

Create a topic and recovery

The Kafka StatefulSet should now be up and running successfully. Now we can create a topic, verify the replication of this topic, and then see how the system recovers when a pod is deleted.

Open terminal on pod ***kafka-0:***

```
kubectl exec -it kafka-0 -- bash
```

Create a topic named ***test*** with three partitions and a replication factor of 3.

```
kafka-topics --create --topic test --partitions 3 --replication-
factor 3 --bootstrap-server kafka-0.kafka-
headless.kafka.svc.cluster.local:9092
```

Verify the topic partitions are replicated across all three brokers:

```
kafka-topics --describe --topic test --bootstrap-server kafka-
0.kafka-headless.kafka.svc.cluster.local:9092
```

The output of the above command will be similar to the following:

```
Topic: test     TopicId: WmMXgsr2RcyZU9ohfoTUWQ PartitionCount:
3       ReplicationFactor: 3    Configs:
        Topic: test     Partition: 0    Leader: 0
Replicas: 0,1,2 Isr: 0,1,2
        Topic: test     Partition: 1    Leader: 1
Replicas: 1,2,0 Isr: 1,2,0
        Topic: test     Partition: 2    Leader: 2
Replicas: 2,0,1 Isr: 2,0,1
```

The output above shows there are 3 in-sync replicas.

Now we will simulate a loss of one of the brokers by deleting the associated pod. Open a new local terminal for the following command:

```
kubectl scale sts kafka --replicas 2
```

In the remote ***kafka-0*** terminal, check topic replication to see that only 2 replicas exist:

```
kafka-topics --describe --topic test --bootstrap-server kafka-
0.kafka-headless.kafka.svc.cluster.local:9092
Topic: test     TopicId: WmMXgsr2RcyZU9ohfoTUWQ PartitionCount:
3 ReplicationFactor: 3     Configs:
        Topic: test     Partition: 0    Leader: 0
Replicas: 0,1,2    Isr: 0,1
        Topic: test     Partition: 1    Leader: 1
Replicas: 1,2,0    Isr: 0,1
        Topic: test     Partition: 2    Leader: 0
Replicas: 2,0,1    Isr: 0,1
```

Notice that there are only two in-sync replicas for each partition (brokers 0 and 1).

Summary and next steps

This tutorial showed you how to get Kafka running in KRaft mode on a Kubernetes cluster. Both Kafka and Kubernetes are popular technologies, and this tutorial hopefully gives some insight into how it is becoming even easier to use these two tools together.

In the next post, I'll be showing how to configure an SSL listener in Kafka to securely connect to it!