In this module, we're going to talk about the Kafka's control plane. In the previous module, we have talked about a data plane and how the client requests and the replications handled through the data plane. And in this module, we are going to talk about control plane, which manages the metadata of the cluster.

The old way of managing the control plane is through ZooKeeper. there is like one broker is picked as a special controller, and the controller is responsible for managing and changing the metadata of the whole cluster. The metadata is persisted in the consensus service, which is the external cap that is called ZooKeeper.

And the controller is also responsible for propagating the metadata changes to the rest of the broker. The new world of the control plane is implemented through a new module called KRAFT. Since we completely eliminated the dependency on Zookeeper, instead we have a built-in consensus service based on RAFT within the Kafka cluster.

In this case, you can see a few of the brokers will be selected to build this internal consensus service to manage and store the metadata. And this is a relatively new feature, so it's still just in preview, but since this is the future in this module, we are just only going to talk about the control plane based on KRAFT. Now, why do we build a new control plane using KRAFT? Well, these are some of the main reasons. The number one reason is this makes the operation of Kafka a lot easier by removing the number of moving parts.

Earlier, you had two separate systems, which is Kafka and Zookeeper, you have to manage. Right now, with KRAFT, you just have one type of system that you have to deal with. This makes things like deployment, configuration, monitoring, security much easier and simpler than before. The second reason is the KRAFT model in general is much more efficient because we have this customized build consensus survey just for Kafka.

We can store and manage the metadata in a much more efficient way. So here in this figure, we have shown that in general with KRAFT model, we can achieve at least 10x scalability improvements in terms of the amount of metadata it can handle within a cluster.

KRAFT model also allows the metadata to be propagated from the controller to the brokers in a much more efficient way, as we'll see later on. When you configure a KRAFT-based controller, there are two ways you can configure it. The top one shows you that you can configure it in a non-overlapping way. You can pick some of the nodes as brokers and some other nodes dedicated as the controller.

If you have a smaller cluster, you can also choose to run in a shared mode. In this case, we can say some of the nodes will act both as the broker as well as the controller. Both setups are possible. So once we have selected those controllers in the cluster, those controllers need a way to communicate among themselves, and other brokers also need to communicate with the controller in order to propagate the metadata. So this is done through this configuration.

It provides a list of all the controllers and will include its endpoints, including the hostname and the port. The active controller, as well as the other controllers, they each maintain an in-memory metadata cache. So this is actually pretty useful because if the active controller fails, the rest of the controller can take over as the new controller much quicker because it doesn't need to refresh its metadata because it has an up-to-date in-memory copy of all the metadata.

This is actually one of the ways why KRAFT is much more efficient than the old Zucchella-based control plane. Once the active controller decides to change a particular metadata, we need a way to persist that metadata. And this is achieved through an internal built-in topic called Cluster Metadata.

This is a very special topic. It only has a single partition and is used to persist all the metadata within this cluster. So for example, here, if the controller wants to make a change to change the leader and in sync replica set of a particular partition, the first thing you need to do is write a metadata record into this metadata log.

Then this data will be replicated to other controllers and all other brokers will also be replicating this metadata log to its local log. By keeping a local metadata log, this allows the broker to be able to keep up with the changes in a much more incremental way. For example, if the broker is restarted, It doesn't have to refresh all the metadata, you just have to catch up from what is missing since it's done. This is actually much more efficient than the ZooKeeper way.

Now, how is the metadata replicated in the k-raft mode? The replication of the metadata is very similar to how the data is replicated. We have a similar concept as the leader and the followers. The data will be flowing first into the leader and then to the follower. And there's a similar concept of leader epoch. and all the records will be tagged with the leader epoch as they are appended to the log.

The leader of this metadata log is also the active controller and is responsible for writing the data into this replicated log. But there are some key differences of the metadata replication from the data replication. One of the key differences is the leader election and offset committing is completely different in the KRAFT mode because there's no concept of ISR or NSYNC replica set because we have no other consensus service that we can rely upon to persist this metadata.

So instead, in KRAFT mode, the leader election and offset committing is all based on a quorum-based system, which we'll talk about a bit later. The second difference is in In the kref mode, all the metadata record has to be persisted in the log. It has to be flushed to disk before they can be considered committed. Now let's look at how the leader election works in the kref mode. Let's say in this case the old leader on controller 1 failed and then we need to elect a new leader.

In this case, since there's no concept ISR, the remaining replicas, Controller 2 and 3, need to coordinate among themselves to select a new leader. So the way this works is each of these followers will be first bumping up its epoch and then mark itself as a candidate. It will vote for itself for this particular epoch and will send a vote request to all other replicas to request a vote from them.

To prevent the case where all the followers try to elect themselves at the same time, there's a little back-off logic. So each of the candidates will back off to a random number and then make themselves as a candidate.

Let's say in this case replica 3 is the one first gets selected as a candidate. And then it will send this voter requests to other followers, including its candidate epoch, as well as a little metadata about its log, which is what's the end offset of its log and the latest epoch it has. Once the other follower receives this voter request, it will first check if it has seen any of the epoch higher than this epoch.

If so, it will reject this request. Then it will check if it has already responded to this particular epoch. If so, it will just send the same response it has made before. In this case, it hasn't really made a response before.Then what does Follower do is to compare the candidate's log with its own log to see which one is longer. In this case, you will notice that the candidate's log has the same epoch but has a longer offset than its local log. So in this case, you notice that the candidate's log is at least as long as as its own local log, it will actually vote a yes to this vote request.

So once the candidate has collected enough votes, it will consider itself as the new leader. In this case, it has accumulated two out of three votes so it can become the new leader. Once the new leader is selected, it will inform other replicas through another request to tell them that it is the new leader and they should be following them.

Once the new leader is elected, we need a very similar log reconciliation process to make sure all the replicas are consistent. As you can see in this case, initially, the follower's data is a little bit inconsistent with the new leader because some of the records in this log are never committed.

So in this case, what a follower will do is to go through a similar process as the data replication reconciliation logic by sending its epoch and offset, and eventually the follows log will be truncated and made consistent as the leader. We need a way to prevent the metadata log from going forever. We can't simply just truncate old data in the metadata log because it may still consist of some of the latest value for some of the resources the cluster is managing.

So instead, what we do is to use a concept called snapshot. So periodically each of those controllers as well as the broker will take the latest records in the metadata cache and write that as a snapshot. Once the snapshot is written, which is corresponding to a particular end offset, we know all the records before that particular end offset in the metadata log is redundant.

They are no longer needed. So at that point, once we generate a snapshot, we can start truncating some of the old data in this metadata log. the snapshot, together with the remaining of the metadata log, will still give us the latest metadata for the whole cluster.

Now, how is the snapshot being used? Well, the snapshot is used in a couple cases. Every time when the broker is restarted, it will need to rebuild this in-memory metadata cache. So it does that by first scanning through the latest snapshot and loading that into memory, followed by continuing fetching the data from the last offset associated with the snapshot from this metadata log. The second use case of the snapshot is when the controller or the broker is fetching the metadata from the leader's metadata log.

So sometimes when a fetch request is issued, the offset in a fetch request may no longer exist in the leader's metadatalog, because the leader may have truncated it after generating some of the latest snapshot. So in this case, the leader will be sending a response to the controller to follow to indicate that its offset is missing.

You need to first catch up on the snapshot. So after receiving this response, the broker of the controller will first scan through the, will issue a request to scan through all the snapshot data from the leader. And after that, we'll be switching to consuming the data from the metadata log after the end offset associated with the snapshot. So this concludes this particular module. Thanks for listening. Thank you.