Configure High Availability

We now look at high availability in Kubernetes. So what happens when you lose the master node in your cluster? As long as the workers are up and containers are alive, your applications are still running. Users can access the application until things start to fail.

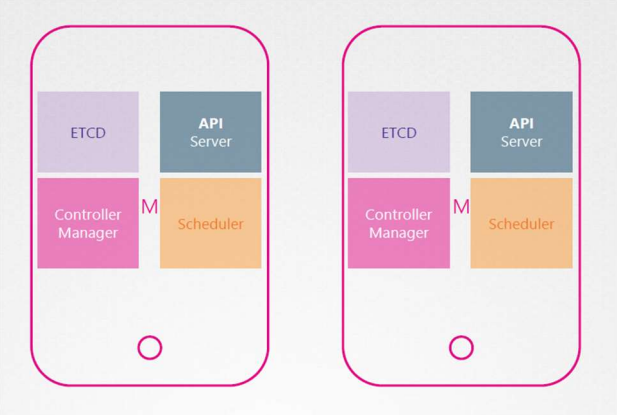
For example, a container or a pod on the worker node crashes. Now if that pod was part of a replica set, then the replication controller on the master needs to instruct the worker to load a new pod, but the master is not available and so are the controllers and schedulers on the master. There is no one to recreate the pod and no one to schedule it on nodes.

Similarly, since the kube-apiserver is not available, you cannot access the cluster externally through the kubectl tool or through API for management purposes, which is why you must consider multiple master nodes in a high availability configuration in your production environment.

A high availability configuration is where you have redundancy across every component in the cluster so as to avoid a single point of failure. The master nodes, the worker nodes, the control plane components, the application of course, which we already have multiple copies in the form of replica sets and services. So our focus in this lecture is going to be on the master and control plane components.

As we learned already, the master node hosts the control plane components including the API, controller manager, scheduler, and etcd server. In a high availability setup with an additional master node, you have the same components running on the new master as well.

So how does that work, running multiple instances of the same components, are they going to do the same thing twice? How do they share the work among themselves? Well, that differs based on what they do.



Well, that differs based on what they do.

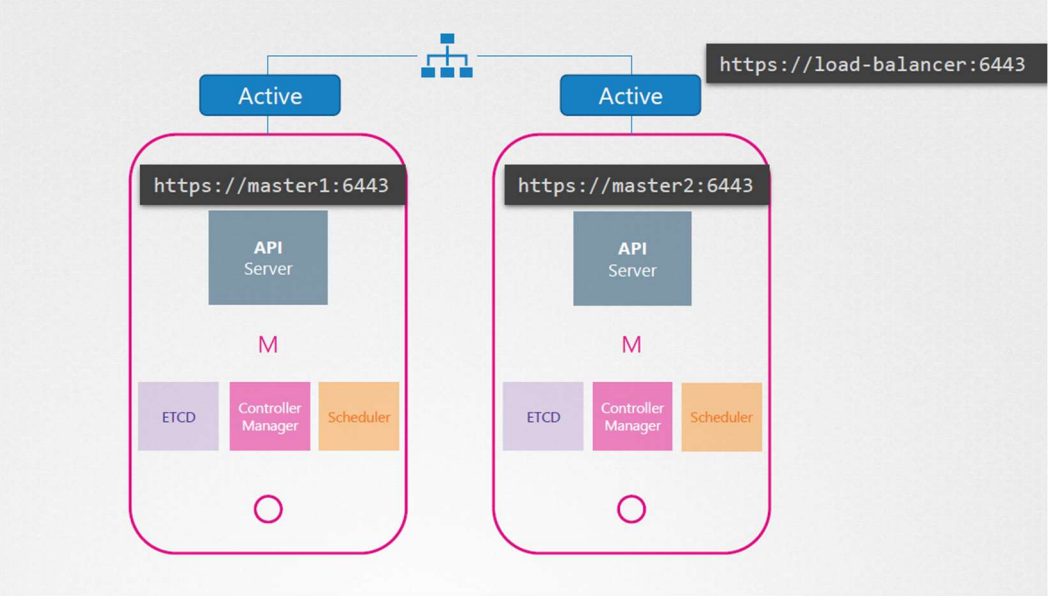
API server

Now we know that the API server is responsible for receiving requests and processing them or providing information about the cluster. They work on one request at a time, so the API servers on all cluster nodes can be alive and running at the same time in an **active-active mode**.

So far in this course, we know that the kubectl utility talks to the API server to get things done, and we point the kubectl utility to reach the master node at port 6443. That's where the API server listens, and this is configured in the kubeconfig file.

Well, now with two masters, where do we point the kubectl to? We can send the request to either one of them, but we shouldn't be sending the same request to both of them. So it is better to have a load balancer of some kind configured in front of the master nodes that split traffic between the API servers.

So, we then point the kubectl utility to that load balancer. You may use Nginx or HAProxy or any other load balancer for this purpose.



Scheduler and Controller manager

What about the scheduler and the controller manager?

These are controllers that watch the state of the cluster and take actions. For example, the controller manager consists of controllers like the replication controller that is constantly watching the state of pods and taking necessary actions like creating a new pod when one fails, etc.

If multiple instances of these run in parallel, then they might duplicate actions, resulting in more pods than actually needed. The same is true with the scheduler. As such, they must not run in parallel. They run in an **active standby** mode.

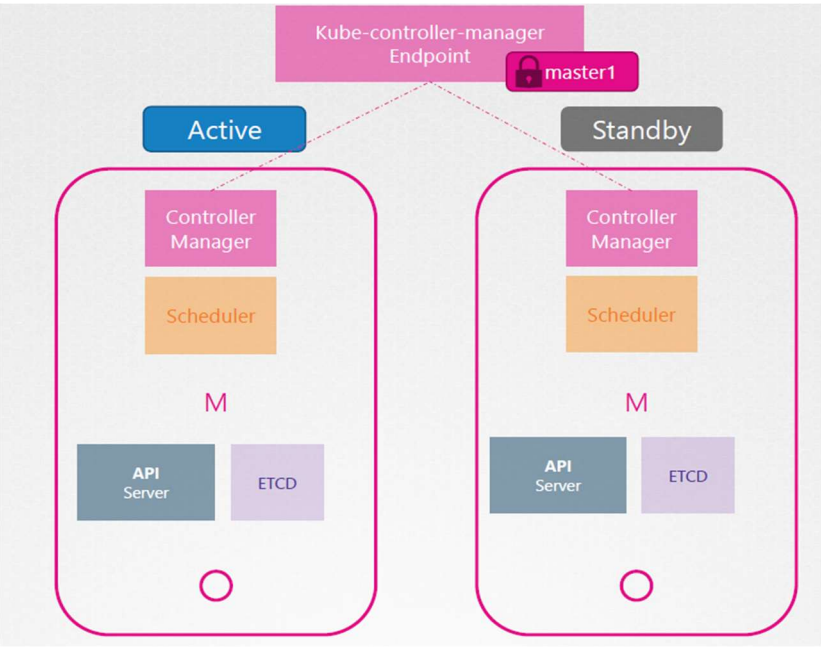
So then who decides which among the two is active and which is passive? This is achieved through a leader election process.

So how does that work? Let's look at the controller manager, for instance.

When a controller manager process is configured, you may specify the leader elect option, which is by default set to true. With this option, when the controller manager process starts, it tries to gain a lease or a lock on an endpoint object in Kubernetes named as **kube-controller-manager-endpoint**. Whichever

process first updates the endpoint with its information gains the lease and becomes the active of the two. The other becomes passive. It holds the lock for the lease duration specified using the leader elect lease duration option, which is by default set to 15 seconds.

The active process then renews the lease every 10 seconds, which is the default value for the option leader elect renew deadline. Both the processes try to become the leader every two seconds set by the leader elect retry period option. That way, if one process fails, maybe because the first master crashes, then the second process can acquire the lock and become the leader. The scheduler follows a similar approach and has the same command line options.

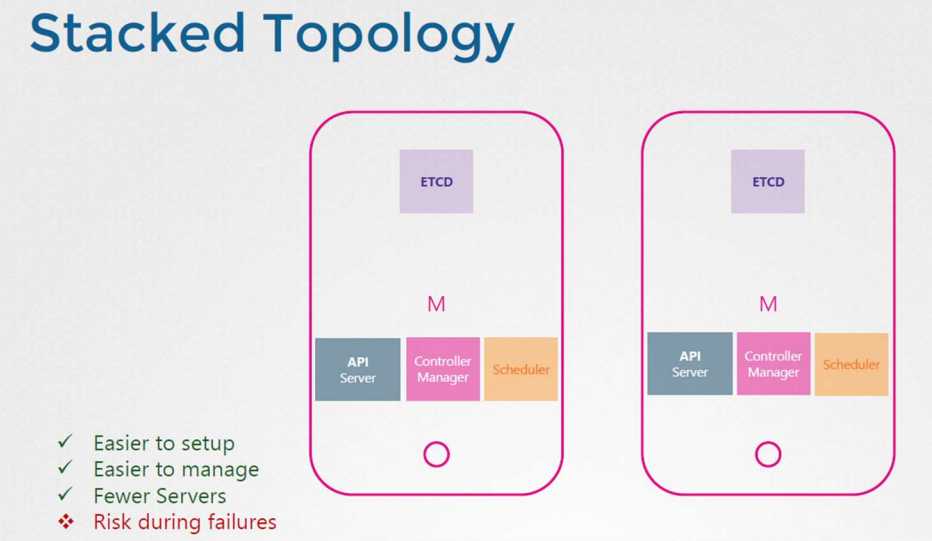



```
kube-controller-manager --leader-elect true  [other options]
                        --leader-elect-lease-duration 15s
                        --leader-elect-renew-deadline 10s
                        --leader-elect-retry-period 2s
```

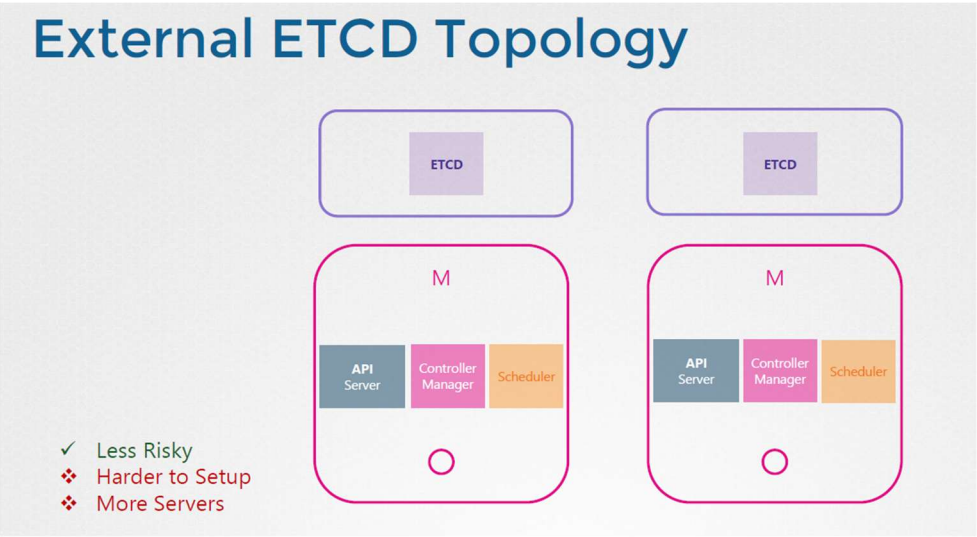
ETCD

Next step is etcd. We discussed etcd earlier in this course. It's a good idea to go through that again. Now, just to quickly refresh your memory, as we're going to discuss some more related topics to how etcd works in this lecture.

With etcd, there are two topologies that you can configure in Kubernetes. One is as it looks here, the same architecture that we have been following throughout this course where etcd is part of the Kubernetes master nodes. It's called a stacked control plane nodes topology. This is easier to set up and easier to manage and requires fewer nodes. But if one node goes down, both an etcd member and the control plane instances are lost, and redundancy is compromised.



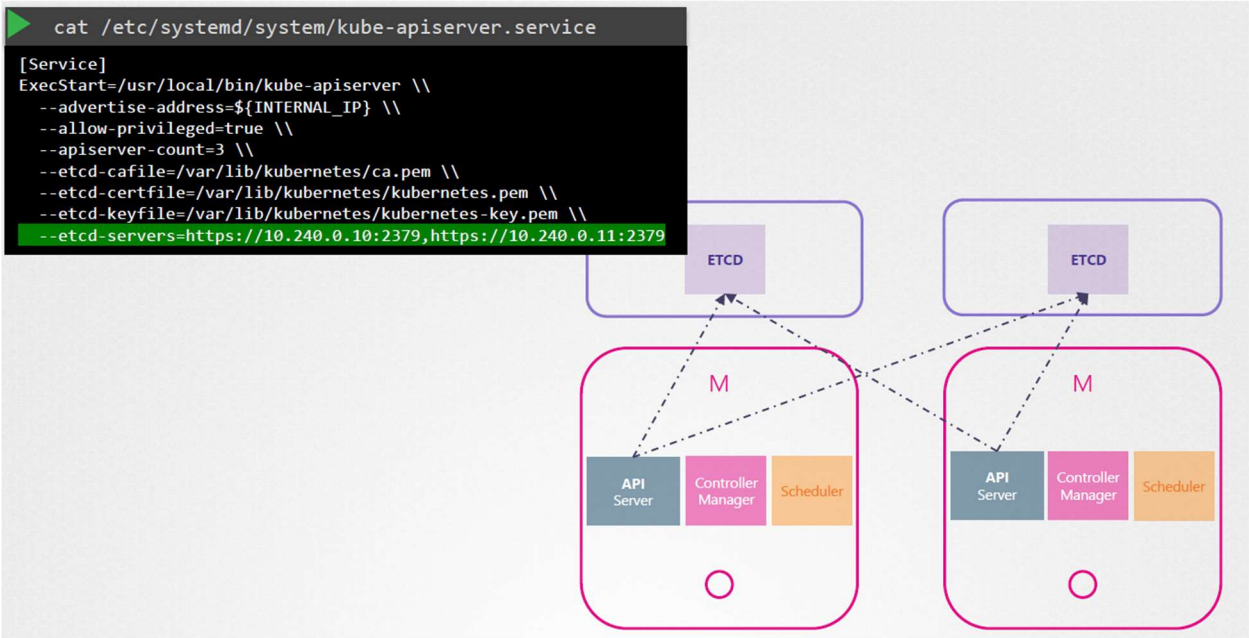
The other topology is where etcd is separated from the control plane nodes and runs on its own set of servers. This is a topology with external etcd servers. Compared to the previous topology, this is less risky, as a failed control plane node does not impact the etcd cluster and the data it stores. However, it is harder to set up and requires twice the number of servers for the external etcd nodes.



So, remember the API server is the only component that talks to the etcd server, and if you look into the API service configuration options, we have a set of options specifying where the etcd server is. Regardless of the topology we use and wherever we configure etcd servers, whether on the same server or on a separate server, ultimately, we need to make sure that the API server is pointing to the right address of the etcd servers.

Now remember etcd is a distributed system, so the API server or any other component that wishes to talk to it can reach the etcd server at any of its instances. You can read and write data through any of the available etcd server instances. This is why we specify a list of etcd servers in the kube-apiserver configuration.

In the next lecture, we discuss more about how etcd servers work in a cluster setup and the best practices around the number of recommended nodes in a cluster.



So back to our design, we had originally planned for a single master node in our cluster. Now with HA, we decided to configure multiple masters. We also mentioned about a load balancer for the API server, so we will have that as well. So we now have a total of five nodes in our cluster.