ETCD in HA

In this lecture, we will talk about etcd in a high availability setup. So this is really a prerequisite lecture for the next lecture where we talk about configuring Kubernetes in a highly available mode. Well, one portion of that deals with configuring etcd in an HA mode. So in this lecture, we will discuss etcd in HA mode.

In the beginning of this course, we took a quick look at etcd. We will now recap real quick and more importantly, focus on the cluster configuration on etcd. So let's recap real quick and look at the number of nodes in the cluster, what raft protocol is, etc

So what is etcd?

It's a distributed, reliable key-value store that is simple, secure, and fast.

So let's break it up.

Traditionally, data was organized and stored in tables like this. For example, to store details about a number of individuals.

Name	Age	Location	Salary	Grade
John Doe	45	New York	5000	
Dave Smith	34	New York	4000	
Aryan Kumar	10	New York		A
Lauren Rob	13	Bangalore		C
Lily Oliver	15	Bangalore		B

A key-value store stores information in the form of documents or pages. So each individual gets a document, and all information about that individual is stored within that file.

These files can be in any format or structure, and changes to one file do not affect the others. In this case, the working individuals can have their files with salary fields.

While you could store and retrieve simple key and values, when your data gets complex, you typically end up transacting in data formats like JSON or YAML. So that's what etcd is and how you quickly get started with it.

We also said that etcd is distributed. So what does that mean? And that is what we are going to focus on in this lecture.

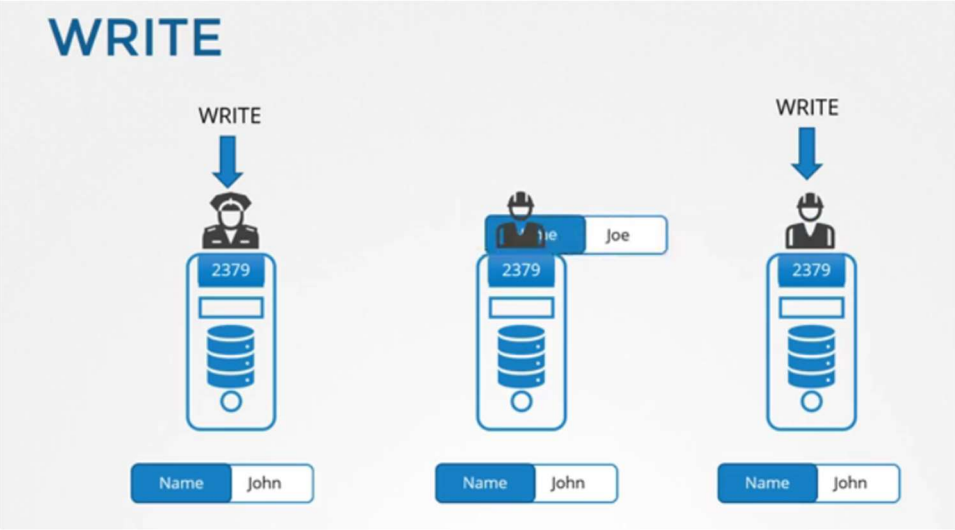
We had etcd on a single server, but it's a database and maybe storing critical data. So it is possible to have your data store across multiple servers. Now, you have three servers all running etcd and all maintaining an identical copy of the database. So if you lose one, you still have two copies of your data. But how does it ensure the data on all the nodes are consistent?

You can write to any instance and read your data from any instance. etcd ensures that the same consistent copy of the data is available on all instances at the same time. So how does it do that?

With reads, it's easy. Since the same data is available across all nodes, you can easily read it from any nodes. But that is not the case with writes. What if two write requests come in on two different instances? Which one goes through?

For example, I have writes coming in for the name set to John on one end, with the name Joe on the other. Of course, we cannot have two different data on two different nodes. When I said etcd can write through any instance, I wasn't 100% right. etcd does not process the writes on each node. Instead, only one of the instances is responsible for processing the writes.

Internally, the two nodes elect a leader among them. Of the total instances, one node becomes the leader and the other node becomes the followers. If the writes came in through the leader node, then the leader processes the write. The leader makes sure that the other nodes are sent a copy of the data. If the writes came in through any of the other follower nodes, then they forward the writes to the leader internally, and then the leader processes the writes.



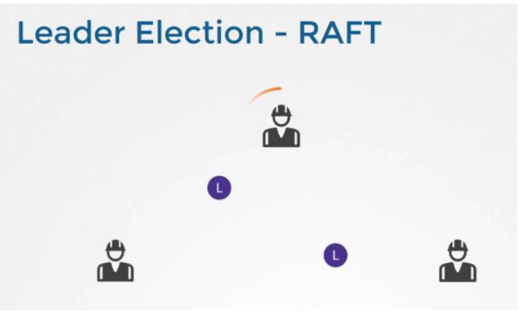
Again, when the writes are processed, the leader ensures that copies of the write are distributed to other instances in the cluster. Thus, a write is only considered complete if the leader gets consent from the other members in the cluster. Thus, a write is only considered complete if the leader gets consent from

the other members in the cluster. Thus, a write is only considered complete if the leader gets consent from the other members in the cluster.

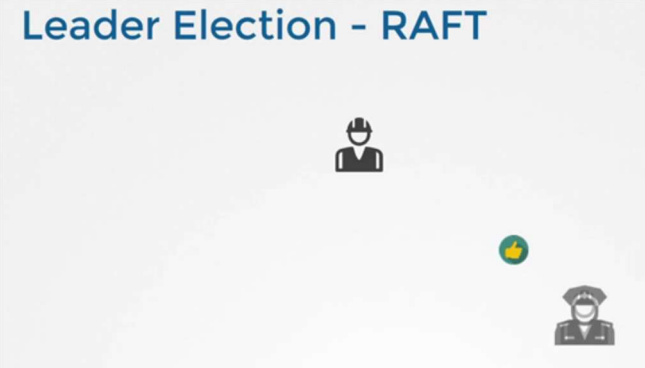
So how do they elect the leader among themselves? And how do they ensure a write is propagated across all instances?

etcd implements distributed consensus using the Raft protocol. Let's see how that works in a three-node cluster.

When the cluster is set up, we have three nodes that do not have a leader elected. The Raft algorithm uses random timers for initiating requests. For example, a random timer is kicked off on the three managers. The first one to finish the timer sends out a request to the other nodes requesting permission to be the leader. The other managers, on receiving the request, respond with their vote, and the node assumes the leader role.

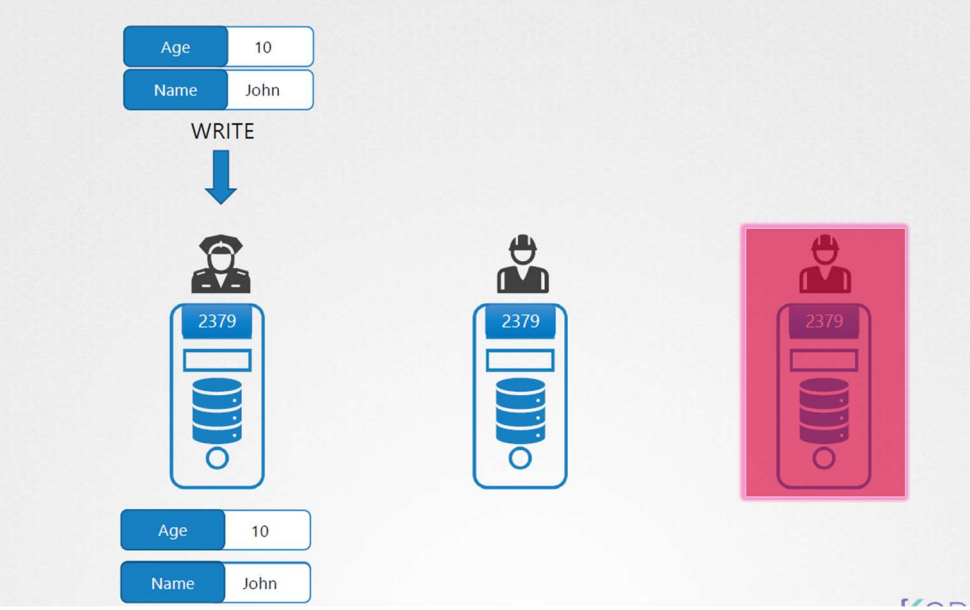


Now that it is elected the leader, it sends out notifications at regular intervals to other masters informing them that it is continuing to assume the role of the leader. In case the other nodes do not receive a notification from the leader at some point in time, which could either be due to the leader going down or losing network connectivity, the nodes initiate a reelection process among themselves, and a new leader is identified.



Going back to our previous example where a write comes in, it is processed by the leader and is replicated to other nodes in the cluster. The write is considered to be complete only once it is replicated to the other instances in the cluster. The write is considered to be complete only once it is replicated to the other instances in the cluster.

We said that the etcd cluster is highly available. So even if we lose a node, it should still function. Say, for example, a new write comes in, but one of the nodes is not responding, and hence, the leader is only able to write to two nodes in the cluster. Is the write considered to be complete? Does it wait for the third node to be up? Or does it fail? A write is considered to be complete if it can be written on the majority of the nodes in the cluster. For example, in this case of three nodes, the majority is two. So if the data can be written on two of the nodes, then the write is considered to be complete. If the third node were to come online, then the data is copied to that as well.



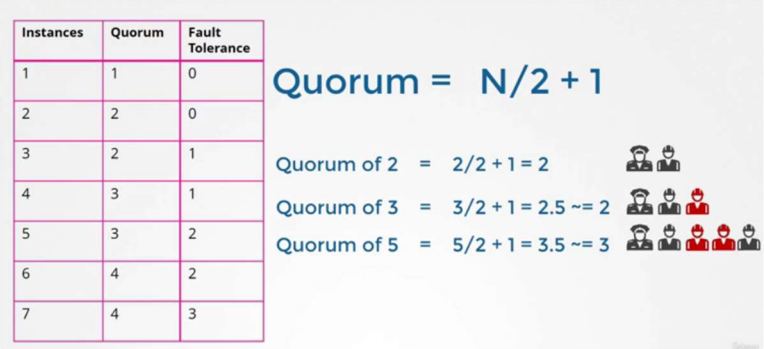
So what is the majority? Well, a more appropriate term to use would be quorum. Quorum is the minimum number of nodes that must be available for the cluster to function properly or make a successful write.

In the case of three, we know it's two. For any given number of nodes, quorum is the total number of nodes divided by 2 plus 1. So quorum of three nodes is 3 by 2, which is 1.5, plus 1 equals 2.5. If there is a .5, consider the whole number only. So that's 2.

Similarly, the quorum of five nodes is 3. So here is a table that shows the quorum of clusters of size one to seven. Quorum of three and five are what we calculated. Quorum of one is one itself. Meaning if you have a single node cluster, none of these really apply. If you lose that node, everything's gone.

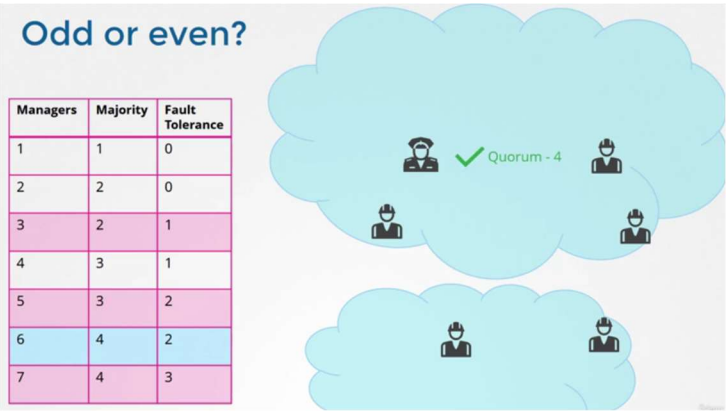
If you look at two and apply the same formula, the quorum is two itself. 2 by 2 is 1, and 1 plus 1 is 2. So even if you have two instances in the cluster, the majority is still two. If one fails, there's no quorum, so writes won't be processed. So having two instances is like having one instance. It doesn't offer you any real value as quorum cannot be met, which is why it is recommended to have a minimum of three instances in an etcd cluster. That way it offers fault tolerance of at least one node. If you lose one, you can still have quorum, and the cluster will continue to function.

So the first column minus the second column gives you the fault tolerance, the number of nodes that you can afford to lose while keeping the cluster alive. So we have one to seven nodes here. One and two are out of consideration.



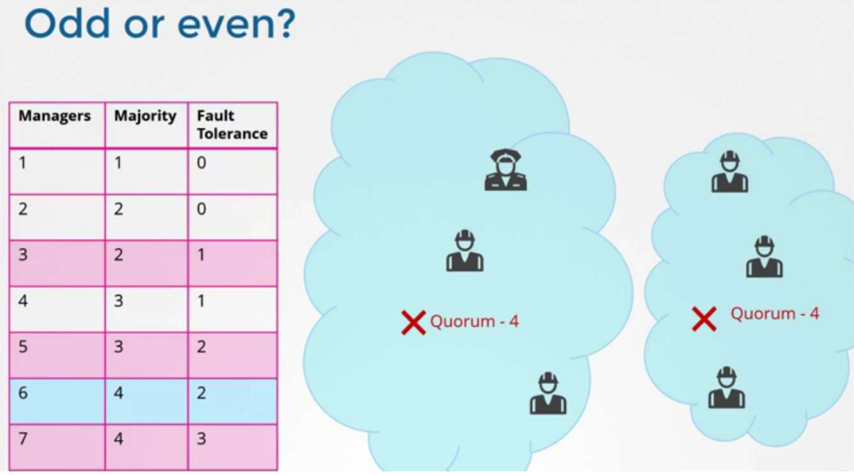
So from three to seven, what do we consider? As you can see, three and four have the same fault tolerance of one. Five and six have the same fault tolerance of two. When deciding on the number of master nodes, it is recommended to select an odd number, as highlighted in the table, three or five or seven.

Say we have a six-node cluster. Say, for example, due to a disruption in the network, it fails and causes the network to partition. We now have four nodes on one and two on the other. In this case, the group with four nodes has quorum and continues to operate normally.



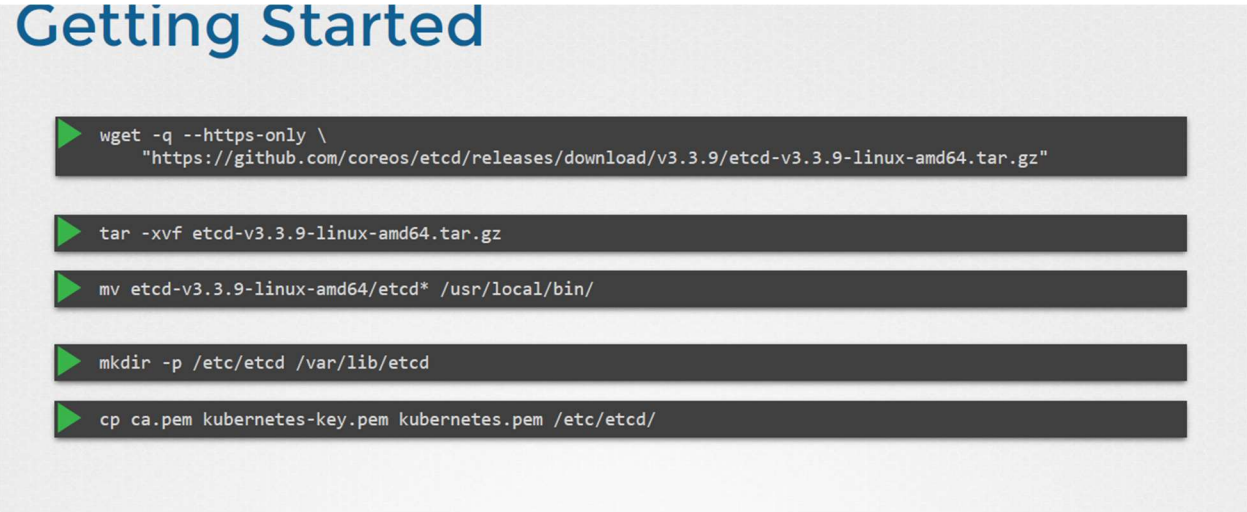
However, if the network got partitioned in a different way, resulting in nodes being distributed equally between the two, each group now has three nodes only. But since we originally had six manager nodes, the quorum for the cluster to stay alive is four. But if you look at the groups here, neither of these groups have four managers to meet the quorum, so it results in a failed cluster.

So with an even number of nodes, there is a possibility of the cluster failing during a network segmentation.



In case we had an odd number of managers originally, say seven, then after the network segmentation, we have four on one segmented network and three on the other. And so our cluster still lives on the group with four managers, as it meets the quorum of four. No matter how the network segments, there are better chances for your cluster to stay alive in case of network segmentation with an odd number of nodes. So an odd number of nodes is preferred over an even number. Having five is preferred over six, and having more than five nodes is really not necessary, as five gives you enough fault tolerance.

To install etcd on a server, download the latest supported binary, extract it, create the required directory structure, and copy over the certificate files generated for etcd. We discussed how to generate these certificates in detail in the TLS certificate section. Then, configure the etcd service.



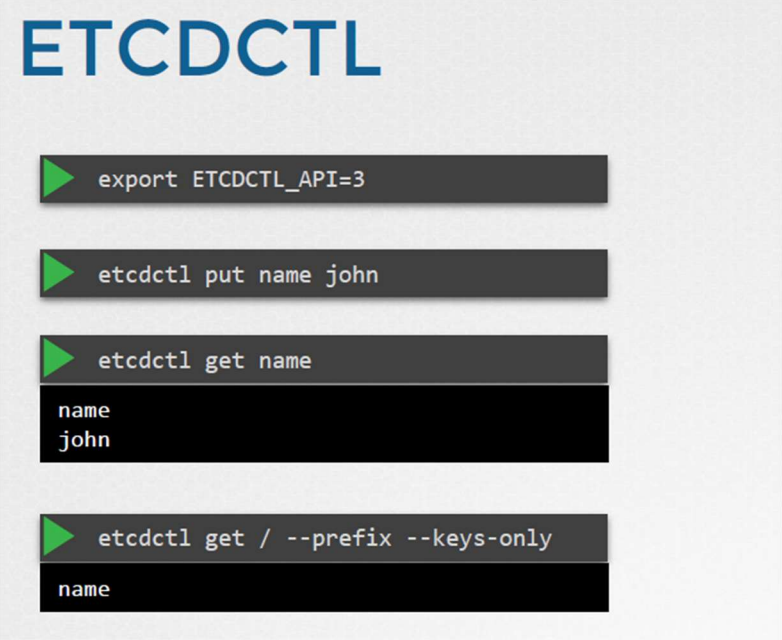
What's important here is to note that the initial cluster option, where we pass the peers' information, that's how each etcd service knows that it is part of a cluster and where its peers are. Once installed and configured, use the etcd cuddle utility to store and retrieve data.

etcd.service

```
ExecStart=/usr/local/bin/etcd \\
  --name ${ETCD_NAME} \\
  --cert-file=/etc/etcd/kubernetes.pem \\
  --key-file=/etc/etcd/kubernetes-key.pem \\
  --peer-cert-file=/etc/etcd/kubernetes.pem \\
  --peer-key-file=/etc/etcd/kubernetes-key.pem \\
  --trusted-ca-file=/etc/etcd/ca.pem \\
  --peer-trusted-ca-file=/etc/etcd/ca.pem \\
  --peer-client-cert-auth \\
  --client-cert-auth \\
  --initial-advertise-peer-urls https://${INTERNAL_IP}:2380 \\
  --listen-peer-urls https://${INTERNAL_IP}:2380 \\
  --listen-client-urls https://${INTERNAL_IP}:2379,https://127.0.0.1:2379 \\
  --advertise-client-urls https://${INTERNAL_IP}:2379 \\
  --initial-cluster-token etcd-cluster-0 \\
  --initial-cluster peer-1=https://${PEER1_IP}:2380,peer-2=https://${PEER2_IP}:2380 \\
  --initial-cluster-state new \\
  --data-dir=/var/lib/etcd
```

The etcd cuddle utility has two API versions, V2 and V3. So the commands work differently in each version. Version 2 is the default. However, we will use version 3. So set an environment variable, etcdcuddle_api, to three. Otherwise, the below commands won't work.

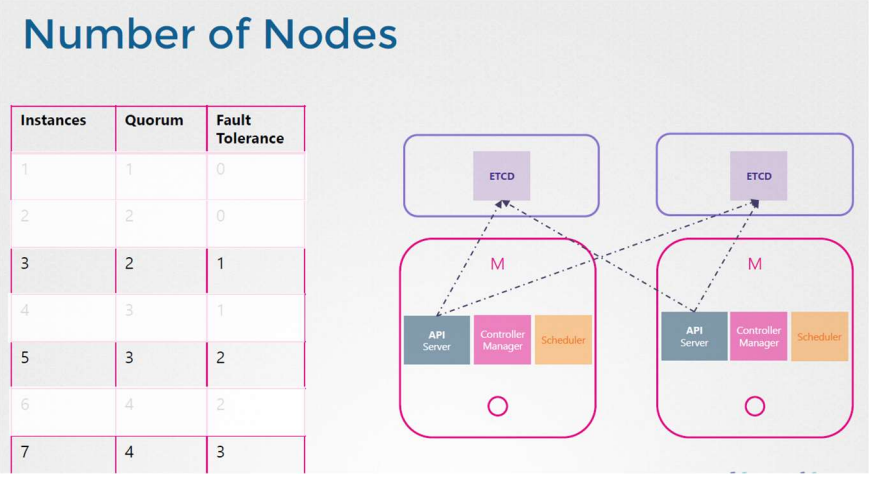
Run the etcd cuddle put command and specify the key as name and the value as John. To retrieve data, run the etcd cuddle get command with the key name, and it returns the value John. To get all keys, run the etcd cuddle get keys only command.



Going back to our design, how many nodes should our cluster have? In an HA environment, as you can see, having one or two instances doesn't really make any sense, as losing one node in either case will leave

you without quorum and thus, render the cluster non-functional. Hence, the minimum required nodes in an HA setup are three.

We also discussed why we prefer an odd number of instances over an even number. Having an even number of instances can leave the cluster without quorum in certain network partition scenarios. So, all the even numbers of nodes are out of scope. Therefore, we are left with three, five, and seven, or any odd number above that. Three is a good start, but if you prefer a higher level of fault tolerance, then five is better. But anything beyond that is just unnecessary. So, considering your environment, the fault tolerance requirements, and the cost that you can bear, you should be able to choose one number from this list. In our case, we go with three.



So how does our design look now? With HA, the minimum required number of nodes for fault tolerance is three. Now, while it would be great to have three master nodes, we are limited by the capacity of our laptop, so we will just go with two. But if you're deploying the setup in another environment and have sufficient capacity, feel free to go with three. We also chose to go with the stacked topology where we will have the etcd servers on the master nodes itself.