What Is Etcd and How Do You Set Up an Etcd Kubernetes Cluster?

CALIN RUS

PUBLISHED: JANUARY 31, 2019
UPDATED: JANUARY 26, 2021

KUBERNETES ETCD TUTORIAL CONCEPTUAL

Computing Foundation. It is pronounced “et-cee-dee”, making reference to distributing the Unix “/etc” directory, where most global configuration files live, across multiple machines. It serves as the backbone of many distributed systems, providing a reliable way for storing data across a cluster of servers. It works on a variety of operating systems including Linux, BSD and OS X.

Etcd has the following properties:

- **Fully Replicated:** The entire store is available on every node in the cluster
- **Highly Available:** Etcd is designed to avoid single points of failure in case of hardware or network issues
- **Consistent:** Every read returns the most recent write across multiple hosts
- **Simple:** Includes a well-defined, user-facing API (gRPC)
- **Secure:** Implements automatic TLS with optional client certificate authentication
- **Fast:** Benchmarked at 10,000 writes per second
- **Reliable:** The store is properly distributed using the **Raft** algorithm

How Does Etcd Work?

To understand how Etcd works, it is important to define three key concepts: leaders, elections, and terms. In a Raft-based system, the cluster holds an election to choose a leader for a given term.

at any given time.

If a leader dies, or is no longer responsive, the rest of the nodes will begin a new **election** after a predetermined timeout to select a new leader. Each node maintains a randomized election timer that represents the amount of time the node will wait before calling for a new election and selecting itself as a candidate.

If the node does not hear from the leader before a timeout occurs, the node begins a new election by starting a new **term**, marking itself as a candidate, and asking for votes from the other nodes. Each node votes for the first candidate that requests its vote. If a candidate receives a vote from the majority of the nodes in the cluster, it becomes the new leader. Since the election timeout differs on each node, the first candidate often becomes the new leader. However, if multiple candidates exist and receive the same number of votes, the existing election term will end without a leader and a new term will begin with new randomized election timers.

As mentioned above, any changes must be directed to the leader node. Rather than accepting and committing the change immediately, Etcd uses the Raft algorithm to ensure that the majority of nodes all agree on the change. The leader sends the proposed new value to each node in the cluster. The nodes then send a message confirming receipt of the new value. If the majority of nodes confirm receipt, the leader commits the new value and messages each node that the value is committed to the log. This means that each change requires a quorum from the cluster nodes in order to be committed.

Etcd in Kubernetes

Etcd's job within Kubernetes is to safely store critical data for distributed systems. It's best known as Kubernetes' primary datastore used to store its configuration data, state, and metadata. Since Kubernetes usually runs on a cluster of several machines, it is a distributed system that requires a distributed datastore like Etcd.

Etcd makes it easy to store data across a cluster and watch for changes, allowing any node from Kubernetes cluster to read and write data. Etcd's `watch` functionality is used by Kubernetes to **monitor** changes to either the `actual` or the `desired` state of its system. If they are different, Kubernetes makes changes to reconcile the two states. Every read by the `kubectl` command is retrieved from data stored in Etcd, any change made (`kubectl apply`) will create or update entries in Etcd, and every crash will trigger value changes in etcd.

Deployment and Hardware Recommendations

For testing or development purposes, Etcd can run on a laptop or a light cloud setup. However, when running Etcd clusters in production, we should take in consideration the guidelines offered by Etcd's **official documentation**. The page offers a good starting point for a robust production deployment. Things to keep in mind:

- Since Etcd writes data to disk, SSD is highly recommended
- Always use an odd number of cluster members as quorum is needed to agree on updates to the cluster state
- For performance reasons, clusters should usually not have more than seven nodes

RANCHER

Prerequisites

To follow along with this demo, you will need the following:

- **a Google Cloud Platform account:** The free tier should be more than enough. You should be able to use most other cloud providers with little modification.
- **A server to run Rancher**

Starting a Rancher Instance

To begin, start a Rancher instance on your control server. There is a very intuitive getting started guide for this purpose **here**.

Using Rancher to Deploy a GKE Cluster

Use Rancher to set up and configure a Kubernetes cluster in your GCP account using **this guide**.

Install the **Google Cloud SDK** and `kubelet` command on the same server hosting our Rancher instance. Install the SDK by following the link provided above, and install `kubelet` through the Rancher UI.

RANCHER

As soon as cluster is deployed, check basic `kubectl` functionality by typing:

```
kubectl get nodes
```

```
NAME                                   STATUS   ROLES    AGE   VERSION
gke-c-ggchf-default-pool-df0bc935-31mv  Ready    <none>   48s   v1.11.6-gke.2
gke-c-ggchf-default-pool-df0bc935-ddl5  Ready    <none>   48s   v1.11.6-gke.2
gke-c-ggchf-default-pool-df0bc935-qqhx  Ready    <none>   48s   v1.11.6-gke.2
```

Before deploying the Etcd cluster (through `kubectl` or by importing YAML files in Rancher's UI), we need to configure a few items. In GCE, the default persistent disk is `pd-standard`. We will configure `pd-ssd` for our Etcd deployment. This is not mandatory, but as per Etcd recommendations, SSD is very good option. Please check this **page** to learn about other cloud providers' storage classes.

Let's check the available storage class that GCE offers. As expected, we see the default one, called `standard`:

```
kubectl get storageclass
```

RANCHER

Let's apply this YAML file, updating the value of `zone` to match your preferences, so we can benefit of SSD storage:

```
# storage-class.yaml
---
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: ssd
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
  zone: europe-west4-c # Change this value
```

If we check again, we can see that, along the default standard class, `ssd` is now available:

```
kubectl apply -f storage-class.yaml
kubectl get storageclass
```

RANCHER

```
NAME                PROVISIONER            AGE
ssd                 kubernetes.io/gce-pd   7s
standard (default)  kubernetes.io/gce-pd   4m
```

We can now proceed with deploying the Etcd cluster. We will create a StatefulSet with three replicas, each of which have a dedicated volume with the `ssd` `storageClass`. We will also need to deploy two services, one for internal cluster communication and the other to access the cluster externally via the API.

When forming the cluster, we need to pass a few parameters to the Etcd binary to the datastore. The `listen-client-urls` and `listen-peer-urls` options specify the local addresses the Etcd server uses to accepting incoming connections. Specifying `0.0.0.0` as the IP address means that Etcd will listen for connections on all available interfaces. The `advertise-client-urls` and `initial-advertise-peer-urls` parameters specify the addresses Etcd clients or other etcd members should use to contact the etcd server.

The following YAML file defines our two services and the Etcd StatefulSet:

```
# etcd-sts.yaml
---
apiVersion: v1
kind: Service
```

```
spec:
  type: LoadBalancer
  ports:
    - name: etcd-client
      port: 2379
      protocol: TCP
      targetPort: 2379
  selector:
    app: etcd
```

Apply the YAML file by typing:

```
kubectl apply -f etcd-sts.yaml
```

```
service/etcd-client created
service/etcd created
statefulset.apps/etcd created
```

After applying the YAML file, we can the resources it defines within the different tabs Rancher offers:

Fig. 1: Etcd StatefulSet as seen in the Rancher Workloads tab

Fig. 2: Etcd Service as seen in the Rancher Service Discovery tab

Fig. 2: Etcd Service as seen in the Rancher Service Discovery tab

Fig. 3: Etcd volume as seen in the Rancher Volumes tab

Fig. 3: Etcd volume as seen in the Rancher Volumes tab

Interacting with Etcd

There are two primary ways to interact with Etcd: either using `etcdctl` command or directly through the RESTful API. We will cover both of these briefly, but you can find more in depth information and additional examples by visiting the full documentation **here** and **here**.

`etcdctl` is a command-line interface for interacting with an Etcd server. It can be used to perform a variety of actions such as setting, updating, or removing keys, verifying the cluster health, adding or removing Etcd nodes, and generating database snapshots. By default, `etcdctl` talks to the Etcd server with the `v2` API for backward compatibility. If you want `etcdctl` to speak to Etcd using the `v3` API, you must set the version to "3" via the `ETCDCTL_API` environment variable.

As for the API, every request sent to an Etcd server is a gRPC remote procedure call. This gRPC gateway serves a RESTful proxy that translates HTTP/JSON requests into gRPC messages.

```
kubectl get svc
```

```
NAME          TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S)             AGE
etcd          ClusterIP      None            <none>           2379/TCP,2380/TCP   1m
etcd-client   LoadBalancer   10.15.247.17    35.204.136.231   2379:30525/TCP      1m
kubernetes    ClusterIP      10.15.240.1     <none>           443/TCP             3m
```

We should also find the names of our three Pods so that we can use the `etcdctl` command:

```
kubectl get pods
```

```
NAME     READY   STATUS    RESTARTS   AGE
etcd-0   1/1     Running   0          6m
etcd-1   1/1     Running   0          6m
etcd-2   1/1     Running   0          6m
```

Let's check the Etcd version. For this, we can use the API or CLI (both `v2` and `v3`). The output will be slightly different depending on your chosen method.

RANCHER

```
curl http://35.204.136.231:2379/version
```

```
{"etcdserver":"3.3.8","etcdcluster":"3.3.0"}
```

To check for the version with `v2` of the `etcdctl` client, type:

```
kubectl exec -it etcd-0 -- etcdctl --version
```

```
etcdctl version: 3.3.8
API version: 2
```

To use the `etcdctl` with `v3` of the API, type:

```
kubectl exec -it etcd-0 -- /bin/sh -c 'ETCDCTL_API=3 etcdctl version'
```

```
etcdctl version: 3.3.8
API version: 3.3
```

We can query the API with:

```
curl 35.204.136.231:2379/v2/members
```


```
{"members":[{"id":"2e80f96756a54ca9","name":"etcd-0","peerURLs":["http://etcd-0.etcd:2380"],"c
```


With `etcdctl` using `v2` of the API:

```
kubectl exec -it etcd-0 -- etcdctl member list
```

```
2e80f96756a54ca9: name=etcd-0 peerURLs=http://etcd-0.etcd:2380 clientURLs=http://etcd-0.etcd:2
7fd61f3f79d97779: name=etcd-1 peerURLs=http://etcd-1.etcd:2380 clientURLs=http://etcd-1.etcd:2
b429c86e3cd4e077: name=etcd-2 peerURLs=http://etcd-2.etcd:2380 clientURLs=http://etcd-2.etcd:2
```

With `etcdctl` using `v3` of the API:

RANCHER

```
+------------------+---------+--------+-------------------------+------------------------+
|        ID        | STATUS  |  NAME  |       PEER ADDRS        |      CLIENT ADDRS      |
+------------------+---------+--------+-------------------------+------------------------+
| 2e80f96756a54ca9 | started | etcd-0 | http://etcd-0.etcd:2380 | http://etcd-0.etcd:2379 |
| 7fd61f3f79d97779 | started | etcd-1 | http://etcd-1.etcd:2380 | http://etcd-1.etcd:2379 |
| b429c86e3cd4e077 | started | etcd-2 | http://etcd-2.etcd:2380 | http://etcd-2.etcd:2379 |
+------------------+---------+--------+-------------------------+------------------------+
```

Setting and Retrieving Values in Etcd

The last example we will cover is creating a key and checking it's value on all the 3 Pods in the Etcd cluster. Then we will kill the leader, `etcd-0` in our scenario, and see how a new leader is elected. Finally, once the cluster has recovered, we will verify the value of our previously created key on all members. We will see that there is no data loss, and cluster simply goes on with a different leader.

We can verify that the cluster is initially healthy by typing:

```
kubectl exec -it etcd-0 -- etcdctl cluster-health
```

RANCHER

```
member b429c86e3cd4e077 is healthy: got healthy result from http://etcd-2.etcd:2379
cluster is healthy
```

Next, verify the current leader by typing the following. The last field indicates that `etcd-0` is the leader in our cluster:

```
kubectl exec -it etcd-0 -- etcdctl member list
```

```
2e80f96756a54ca9: name=etcd-0 peerURLs=http://etcd-0.etcd:2380 clientURLs=http://etcd-0.etcd:2
7fd61f3f79d97779: name=etcd-1 peerURLs=http://etcd-1.etcd:2380 clientURLs=http://etcd-1.etcd:2
b429c86e3cd4e077: name=etcd-2 peerURLs=http://etcd-2.etcd:2380 clientURLs=http://etcd-2.etcd:2
```

Using the API, we will create a key called `message` and assign it a value. Remember to substitute the IP address you retrieved for your cluster in the command below:

```
curl http://35.204.136.231:2379/v2/keys/message -XPUT -d value="Hello world"
```

```
{"action":"set","node":{"key":"/message","value":"Hello world","modifiedIndex":9,"createdIndex
```

```
kubectl exec -it etcd-0 -- etcdctl get message
kubectl exec -it etcd-1 -- etcdctl get message
kubectl exec -it etcd-2 -- etcdctl get message
```

```
Hello world
Hello world
Hello world
```

Demonstrating High Availability and Recovery

Next, we can kill the Etcd cluster leader. This will let us see how a new leader is elected and how the cluster recovers from it's `degraded` state. Delete the pod associated with the Etcd leader you discovered above:

```
kubectl delete pod etcd-0
```

```
pod "etcd-0" deleted
```

```
kubectl exec -it etcd-2 -- etcdctl cluster-health
```

```
failed to check the health of member 2e80f96756a54ca9 on http://etcd-0.etcd:2379: Get http://e
member 2e80f96756a54ca9 is unreachable: [http://etcd-0.etcd:2379] are all unreachable
member 7fd61f3f79d97779 is healthy: got healthy result from http://etcd-1.etcd:2379
member b429c86e3cd4e077 is healthy: got healthy result from http://etcd-2.etcd:2379
cluster is degraded
command terminated with exit code 5
```

The above message indicates that the cluster is in a `degraded` state due to the loss of its leader node.

Once Kubernetes responds to the deleted pod by spinning up a new instance, the Etcd cluster should recover:

```
kubectl exec -it etcd-2 -- etcdctl cluster-health
```

```
member 2e80f96756a54ca9 is healthy: got healthy result from http://etcd-0.etcd:2379
member 7fd61f3f79d97779 is healthy: got healthy result from http://etcd-1.etcd:2379
```

We can see that a new leader has been elected by typing:

```
kubectl exec -it etcd-2 -- etcdctl member list
```

```
2e80f96756a54ca9: name=etcd-0 peerURLs=http://etcd-0.etcd:2380 clientURLs=http://etcd-0.etcd:2
7fd61f3f79d97779: name=etcd-1 peerURLs=http://etcd-1.etcd:2380 clientURLs=http://etcd-1.etcd:2
b429c86e3cd4e077: name=etcd-2 peerURLs=http://etcd-2.etcd:2380 clientURLs=http://etcd-2.etcd:2
```

In our case, the `etcd-1` node was elected as leader.

If we will check the value for the `message` key again, we can verify that there was no data loss:

```
kubectl exec -it etcd-0 -- etcdctl get message
kubectl exec -it etcd-1 -- etcdctl get message
kubectl exec -it etcd-2 -- etcdctl get message
```

```
Hello world
```

Conclusion

Etcd is a very powerful, highly available, and reliable distributed key-value store designed for specific use cases. Common examples are storing database connection details, cache settings, feature flags, and more. It was designed to be sequentially consistent, so that every event is stored in the same order throughout the cluster.

We saw how to get an Etcd Kubernetes cluster up and running with the help of **Rancher**. Afterwards, we were able to play with few basic Etcd commands. In order to learn more about the project, how keys can be organized, how to set TTLs for keys, or how to back up all the data, the **official Etcd repo** is a great starting point.

RANCHER

CALIN RUS

Calin Rus is an infrastructure automation engineer, with a general interest in cloud computing, DevOps principles and the Kubernetes ecosystem.

Privacy Policy Agreements Terms of Service