Troubleshooting Kubernetes Cluster Component Failures

29m 20s left

Open Cloud Environment
17%
Ready to login
Average setup time: 2m 27s

Open Code Environment
100%
Setup completed

Credentials

Account ID
530695445 Copy

Username
student Copy

Password
Ca1_01gM2 Copy

Region
US West 2 Copy

PEM Download
PPK Download

Bridge Connection
Loading...

Introduction

Kubernetes is made up of several core components. Recall that every node in a Kubernetes cluster has the following components:

- **kubelet**: The primary node agent that accepts pod specifications
- **Container runtime**: Software responsible for running containers, for example, containerd
- **kube-proxy**: Implements network rules and connection forwarding to enable the Kubernetes service abstraction

In addition to the components on every node, control-plane nodes also include the following components to provide the Kubernetes control plane:

- **kube-apiserver**: Exposes the Kubernetes API
- **etcd**: Key-value store for backing cluster data
- **kube-scheduler**: Responsible for scheduling pods onto nodes
- **kube-controller-manager**: Responsible for running Kubernetes controllers, for example, the node controller that responds to changes in a node's status

This lab step focuses on how these components are deployed in a cluster, and how to diagnose and resolve issues with them.

Instructions

1. List all of the pods in the `kube-system` namespace:

Copy code

```
kubectl get pods -n kube-system
```

Support

```
coredns-6d4b75cb6d-dn29p                                          1/1     Running
coredns-6d4b75cb6d-xrdqh                                          1/1     Running
ebs-csi-controller-75b7b645cc-vnlls                               6/6     Running
ebs-csi-controller-75b7b645cc-ztjt6                               6/6     Running
ebs-csi-node-v4pxg                                                3/3     Running
ebs-csi-node-z7b5d                                                3/3     Running
etcd-ip-10-0-0-100.us-west-2.compute.internal                     1/1     Running
kube-apiserver-ip-10-0-0-100.us-west-2.compute.internal           1/1     Running
kube-controller-manager-ip-10-0-0-100.us-west-2.compute.internal  1/1     Running
kube-proxy-f8dl7                                                  1/1     Running
kube-proxy-hktwm                                                  1/1     Running
kube-proxy-xxkfp                                                  1/1     Running
kube-scheduler-ip-10-0-0-100.us-west-2.compute.internal           1/1     Running
metrics-server-77b976bb4b-2kmxh                                   0/1     Evicted
metrics-server-77b976bb4b-kwq79                                   1/1     Running
```

Note: It may take another minute for all pods to become ready.

You can see that all of the components besides the kubelet and the container runtime have pods running in the cluster's `kube-system` namespace, which is reserved for resources created by the Kubernetes system. Namely, **etcd-...**, **kube-apiserver-...**, **kube-controller-manager-...**, **kube-proxy-...**, and **kube-scheduler**. This means that troubleshooting issues with these components will mainly involve working within Kubernetes. You will see how each of the components is deployed within the cluster in the following instructions.

It is worth pausing to explain the other pods that are included in the list. The **calico-...** pods are related to the cluster networking implementation. Calico is one of many networking options that implement the container network interface (CNI). **core-dns-...** is one implementation of a cluster DNS for Kubernetes. The **kubernetes-dashboard-** is a graphical interface for managing the Kubernetes cluster. Because these are all deployed using pods in Kubernetes, you could troubleshoot them using the same general troubleshooting techniques that you will use in this and the following lab step

2. List all of the daemonsets in the `kube-system` namespace:

Copy code

```
kubectl get daemonset -n kube-system
```

```
NAME                          DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR                            AGE
aws-cloud-controller-manager  1         1         1       1            1           node-role.kubernetes.io/control-plane=   23d
calico-node                   1         1         1       1            1           kubernetes.io/os=linux                   23d
ebs-csi-node                  1         1         1       1            1           kubernetes.io/os=linux                   23d
kube-proxy                    1         1         1       1            1           kubernetes.io/os=linux                   23d
```

Observe that there is a **kube-proxy** daemonset. Daemonsets are commonly used to ensure that a copy of a pod is run on every node in a cluster. Because every node in the cluster needs to run a kube-proxy, a daemonset is the natural way to deploy it in Kubernetes. The **DESIRED** count should equal the **READY** count under normal operation. If it does not, you could analyze the output of `kubectl get pods -n kube-system -o wide` to see ...heduled on.

configured:

Copy code

```
kubectl get daemonset kube-proxy -n kube-system -o yaml | more +
```

The `get` command with `-o yaml` option is a useful pattern to use for generating resource specification YAML files from existing resources. The `more +/"^spec:"` searches the output for `spec:` to start displaying just after the `spec:` line. The daemonset specification is quite complex in this case making use of **initContainers**, **configMap**, and **serviceAccount**. Because the specification is generated by the system, it is not likely to need troubleshooting. However, it is useful to understand everything that is required for the kube-proxy to function.

4. View the logs of one of the kube-proxy pods:

Copy code

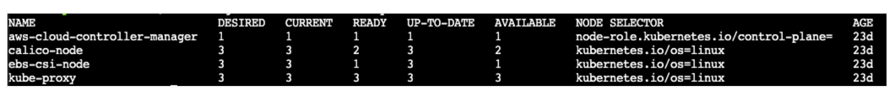
```
# proxy_pod_1 is a variable with the name of the first kube-prox
proxy_pod_1=$(kubectl get pods -n kube-system | grep proxy | cut
kubectl logs -n kube-system $proxy_pod_1
```

The `logs` command is the first place to check when you detect something has gone wrong with one of the kube-system pods. In this case, there are no errors and everything is operating normally.

5. Delete the kube-proxy pod and immediately get the daemonsets after:

Copy code

```
kubectl delete pod $proxy_pod_1 -n kube-system
kubectl get daemonset -n kube-system
```



NAME	DESIRED	CURRENT	READY	UP-TO-DATE	AVAILABLE	NODE SELECTOR	AGE
aws-cloud-controller-manager	1	1	1	1	1	node-role.kubernetes.io/control-plane=	23d
calico-node	3	3	2	3	2	kubernetes.io/os=linux	23d
ebs-csi-node	3	3	1	3	1	kubernetes.io/os=linux	23d
kube-proxy	3	3	3	3	3	kubernetes.io/os=linux	23d

While the pod is gracefully being deleted, the **READY** count drops down to **2**. However, because kube-proxy is deployed using a daemonset, there will be 3 ready pods within 30 seconds of the deletion. This demonstrates the self-healing capabilities of Kubernetes components deployed in Kubernetes.

6. List the system daemonsets to confirm that a new kube-proxy pod is automatically created to replace the deleted one:

Copy code

7. Connect to the control-plane node using SSH:

Copy code

```
ssh control-plane
```

The remaining components are deployed on the control-plane. The way that they are created is different from the way you usually create pods as you will see in the following instructions.

8. Attempt to change the image that the kube-apiserver pod is using:

Copy code

```
# Get the name of the kube-apiserver pod
apiserver_pod=$(kubectl get pods -n kube-system | grep apiserve
# change the pod's image to hello-world using the patch command
kubectl patch pod $apiserver_pod -n kube-system \
  -p '{"spec":{"containers":[{"name":"kube-apiserver","image":"h
```

```
pod/kube-apiserver-ip-10-0-0-100.us-west-2.compute.internal patched
```

The success message leads you to believe the kube-apiserver is now running the hello-world image. If this is true, kubectl will no longer work because it depends on the kube-apiserver container image to be running.

9. Describe the kube-apiserver pod:

Copy code

```
kubectl describe pod $apiserver_pod -n kube-system | more
```

The first thing you should notice is that the command succeeds. This means that the kube-apiserver container is not using the hello-world image. In the **Containers** section of the output you see:

```
Containers:
  kube-apiserver:
    Container ID:  containerd://3ea9ad3476508ce2825c6cb785f7423e4ac223751ac11bc67f3d4d34a33e1546
    Image:         hello-world
    Image ID:      k8s.gcr.io/kube-apiserver@sha256:a04609b85962da7e6531d32b75f652b4fb9f5fe0b0ee0aa160856faad8ec5d96
```

The **Image** is reporting **hello-world**, but the **Image ID** is still the ID of the correct kube-apiserver image (**k8s.gcr.io/kube-apiserver-amd64**...). There are also no **Events** listed for the container, which you would expect to see if the image was changed. So what exactly happened? The pod returned by `kubectl`, which is what the API server returns, is actually a "mirror pod" of the real pod running on the control-plane node. The changes you make to the

For example, you

Pods that behave in this way are called static pods. Static pods are managed directly by the node's kubelet and not by the API server. Static pods are configured by placing pod specifications in a manifest directory that the kubelet periodically reads to keep the pods in sync with the specifications on disk.

10. Get the status of the kubelet and find the `config`:

Copy code

```
systemctl status kubelet
```

```
● kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/lib/systemd/system/kubelet.service; enabled; vendor preset: enabled)
  Drop-In: /etc/systemd/system/kubelet.service.d
           └─10-hostname.conf, 10-kubeadm.conf
   Active: active (running) since Sun 2020-09-20 00:10:11 UTC; 3h 40min ago
     Docs: https://kubernetes.io/docs/home/
 Main PID: 4622 (kubelet)
    Tasks: 18 (limit: 4618)
   CGroup: /system.slice/kubelet.service
           └─4622 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --kubeconfig=/etc/kubernetes/kubelet.conf --config=/var/lib/kubelet/config.yaml
```

11. Output the config file to view the static pod path:

Copy code

```
sudo cat /var/lib/kubelet/config.yaml
```

```
staticPodPath: /etc/kubernetes/manifests
```

The **staticPodPath** field of the kubelet config sets the directory of the static pod specifications, in this case, **/etc/kubernetes/manifests**.

12. List the contents of the /etc/kubernetes/manifests directory:

Copy code

```
ls /etc/kubernetes/manifests
```

```
etcd.yaml  kube-apiserver.yaml  kube-controller-manager.yaml  kube-scheduler.yaml
```

There is one pod specification for each of the control-plane components. The specifications are the same as a normal pod specification. The difference with static pods is only how the pods are managed directly by the kubelet insead of the API server. The kubelet creates mirror pods of the static pods in the API server so that they are visible using the API server and `kubectl`, but they cannot modify static pods.

Because the file is autogenerated and not likely to be edited, there should not be issues with the specification. However, it is instructive to review because it illustrates a few points about working with etcd. To [backup or restore a cluster]() you need to work with etcd. In the context of troubleshooting, you have the option of restoring the cluster to the state stored in a backup when it is difficult to repair a cluster, but you have a working backup. Focus now on the the **spec** section of the output:

```
spec:
  containers:
  - command:
    - etcd
    - --advertise-client-urls=https://10.0.0.100:2379
    - --cert-file=/etc/kubernetes/pki/etcd/server.crt
    - --client-cert-auth=true
    - --data-dir=/var/lib/etcd
    - --initial-advertise-peer-urls=https://10.0.0.100:2380
    - --initial-cluster=ip-10-0-0-100.us-west-2.compute.internal=https://10.0.0.100:2380
    - --key-file=/etc/kubernetes/pki/etcd/server.key
    - --listen-client-urls=https://127.0.0.1:2379,https://10.0.0.100:2379
    - --listen-metrics-urls=http://127.0.0.1:2381
    - --listen-peer-urls=https://10.0.0.100:2380
    - --name=ip-10-0-0-100.us-west-2.compute.internal
    - --peer-cert-file=/etc/kubernetes/pki/etcd/peer.crt
    - --peer-client-cert-auth=true
    - --peer-key-file=/etc/kubernetes/pki/etcd/peer.key
    - --peer-trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
    - --snapshot-count=10000
    - --trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
    image: k8s.gcr.io/etcd:3.4.13-0
    imagePullPolicy: IfNotPresent
    livenessProbe:
      failureThreshold: 8
      httpGet:
        host: 127.0.0.1
        path: /health
        port: 2381
        scheme: HTTP
```

In the **command** you can see several useful pieces of information:

- The URL that etcd is listening on (**https://127.0.0.1:2379**, **https://10.0.0.100:2379**)
- The variety of SSL/TLS certificates that are used and where they are located, for example the **peer-**... options that all point to files in **/etc/kubernetes/pki/etcd/**

Further down you can see that the container uses the host's network (**hostNetwork: true**). This means you can use etcdctl installed on the host to interact with etcd and not rely on containers within the container's network.

You can also see the **command** from the output of `kubectl describe` when that is more convenient, thanks to the mirror pod.

14. Display the TCP ports that are being listened to on the host network to confirm the etcd endpoint (127.0.0.1:2379) is being listened to:

Copy code

```
ss -tl
```

15. Run the following `etcdctl` (The command-line client for etcd) command in the etcd pod's container by using `kubectl exec`: as an example of retrieving a value (`/registry/clusterrolebindings/cluster-admin`) from etcd:

Copy code

```
etcd_pod=$(kubectl get pods -n kube-system | grep ^etcd | cut -
kubectl exec -n kube-system $etcd_pod -- \
  /bin/sh -ec \
  'ETCDCTL_API=3 \
   etcdctl \
   --endpoints=127.0.0.1:2379 \
   --cacert=/etc/kubernetes/pki/etcd/ca.crt \
   --cert=/etc/kubernetes/pki/etcd/peer.crt \
   --key=/etc/kubernetes/pki/etcd/peer.key \
   get \
   /registry/clusterrolebindings/cluster-admin'
```

The `kubectl exec` command allows you to run commands in containers. The command to run comes after the `--` symbol.

The command simply `gets` the value of a key named `foo`. Notice that the command also specifies options for the etcd endpoint (`--endpoints`), the certificate authority certificate (`--cacert`), and a key (`--key`) and certificate (`--cert`) for client authentication. These options correspond to options in the etcd manifest beginning with `--peer-`. Referencing the etc spec is a conveient way to construct the etcdctl command. The etcd database is using version 3, so the environment variable `ETCDCTL_API` must be set to 3.

The `registry/`clusterrolebindings`/cluster-admin` key is retrieved. You should never have to retrieve individual keys directly from etcd, but the example shows you how to use etcdctl and confirms that Kubernetes state is stored in etcd.

16. Disconnect from the control-plane node:

Copy code

```
exit
```

17. Enter the following command to describe the condition of the control-

```
done
```

```
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
```

This command can quickly identify any issues in the listed components. The API server is not listed since `kubectl` would not be functional if the API server was not healthy. If a component is not healthy, you can use the methods discussed before to narrow in on the cause and attempt to remedy the situation.

18. Display the help for the cluster-info command:

Copy code

```
kubectl cluster-info --help
```

```
Display addresses of the control plane and services with label kubernetes.io/cluster-service=true. To further debug and
diagnose cluster problems, use 'kubectl cluster-info dump'.

Examples:
  # Print the address of the control plane and cluster services
  kubectl cluster-info

Available Commands:
  dump          Dump relevant information for debugging and diagnosis

Usage:
  kubectl cluster-info [flags] [options]

Use "kubectl <command> --help" for more information about a given command.
Use "kubectl options" for a list of global command-line options (applies to all commands).
```

The `cluster-info` command is useful if you need to know the address of the control-plane or other cluster services. You can also use the **dump** command to easily get a concatenation of resource specifications, logs, and other useful diagnostic information. Redirecting the output to a file and searching the contents is a useful way to work with the massive dump.

VALIDATION CHECKS

1 **Checks** Start check

Changed Pod Image Name

Changed apiserver pod image name to hello-world

Kubernetes

Did you like this step?

End Lab

Back

Start check