Kubernetes Worker Nodes

Learn what worker nodes do in Kubernetes.

We'll cover the following

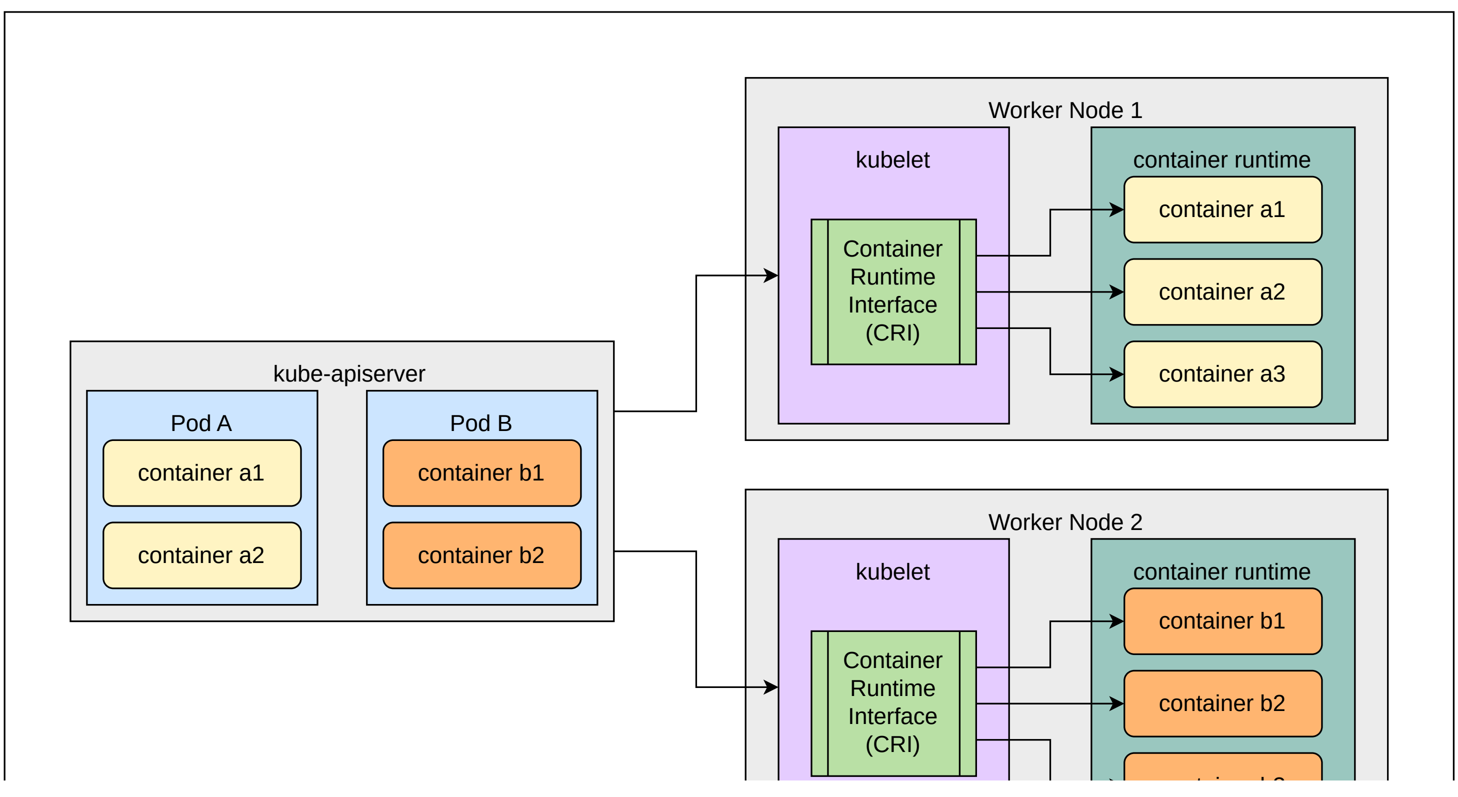
Components of worker nodes

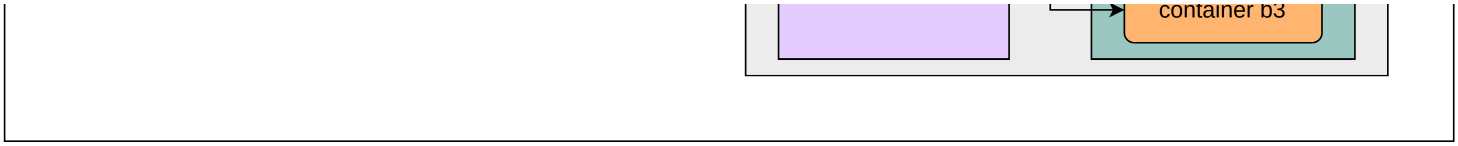
On worker nodes, two Kubernetes components are running—`kubelet` and `kube-proxy`.

How `kubelet` works

The `kube-apiserver`, `kube-scheduler`, and `kube-controller` are running on the control plane nodes, while `kubelet` runs on worker nodes where actual containers run. The `kubelet` nodes are the workhorses of a Kubernetes cluster. They expose computational, networking, and storage resources to containers. We can run the `kubelet` on bare-metal servers, virtual machines (VMs), etc.

In a nutshell, the `kubelet` talks to the `kube-apiserver` and manages the containers running on it.





How the kubelet works

The `kubelet`, on start-up, registers itself to the `kube-apiserver` by creating a dedicated `Node` resource. Then, the `kube-scheduler` can see this `Node` and assign new Pods running on it.

We can view all the nodes with the commands given below:

```
# list all the nodes
kubectl get node

# view nodes with extra infomation
kubectl get node -o wide
```

List/view the nodes

Now, let’s run the commands above in the terminal below:

Terminal 1

Terminal

Click to Connect...

The output will be as follows:

```
NAME                            STATUS     ROLES           AGE   VERSION
educative-demo-control-plane    Ready      control-plane   53s   v1.24.0
educative-demo-worker           NotReady   <none>          16s   v1.24.0
educative-demo-worker2          NotReady   <none>          15s   v1.24.0
```

Nodes overview

Here’s a detailed view of the nodes:

```
NAME                          STATUS     ROLES           AGE   VERSION   INTERNAL-IP   EXTERNAL-IP   OS
educative-demo-control-plane  Ready      control-plane   58s   v1.24.0   172.19.0.3    <none>        Ub
educative-demo-worker         NotReady   <none>          21s   v1.24.0   172.19.0.2    <none>        Ub
educative-demo-worker2        NotReady   <none>          20s   v1.24.0   172.19.0.4    <none>        Ub
```

Detailed view of the nodes

Its heartbeats are reported periodically by the `kubelet`. The status of the nodes can be determined by observing their `STATUS` column in the terminal. When a node crashes or stops reporting heartbeats for a while, it will be tainted and marked as `NotReady` by the `NodeLifecycleController` in the `kube-controller-manager`. This helps the `kube-scheduler` avoid scheduling Pods to unhealthy `kubelet` nodes.

We can modify this monitor period of the `kube-controller-manager` with the flag `--node-monitor-grace-period`.

> **Note:** `--node-monitor-grace-period` duration
>
> This flag specifies the amount of duration marking an unresponsive node as unhealthy.

Steps to create a Pod in the `kubelet`

1. The `kubelet` continuously watches the `kube-apiserver` for Pods (including creating, updating, and deleting) that have been successfully scheduled to itself.
2. The `kubelet` regularly monitors all the managed containers, and ensures those Pods and their containers are healthy and running in the desired state. This is done by calling the `CRI` (Container Runtime Interface). There are different implementations, such as containerd, runc, Kata Containers, gVisor, etc.
3. The healthy states, events, and resource consumption of these containers are reported back to the `kube-apiserver`. The `kubelet` also supports running liveness and readiness probes against the containers. When containers fail, the `kubelet` will restart them to bring them back to life. Once the Pod is deleted from the `kube-apiserver`, it will terminate all the containers that belong to this Pod.

Self-provisioned Kubernetes

All three control plane components, the `kube-apiserver`, `kube-scheduler`, `kube-controller-manager`, and `kube-proxy`, can be deployed as Pods that are managed by `kublet`. This isn't surprising, because the `kubelet` can manage all the Pods. These kinds of Pods are called **static Pods** in Kubernetes. Just as the name suggests, these Pods are static and bound to the current node. The `kubelet` manages the whole lifecycle of these nodes. When these Pods fail, the `kubelet` restarts them.

To run these system-wide components as Pods, we need to deploy the `kubelet` on the control plane nodes. We can add the field `staticPodPath: <my manifest directory>` in the `kubelet` configuration file. The `kubelet` will continuously watch all the file contents in this directory. That also means we can only delete a static Pod by removing the `manifest` file.

For control plane components, we need to specify `hostNetwork: true` in the `podSpec`, because the cluster network isn't available before the Kubernetes control plane is ready.

The `kubelet` will automatically create a corresponding mirror Pod for every static Pod. We can view the Pods in the `kube-apiserver`, but we can't modify them there. These static Pods are fully controlled by the `kubelet` that's running on a specific node. A particular annotation `kubernetes.io/config.mirror` is added on every mirror Pod. An example of a `kube-apiserver` mirror Pod is given below:

```yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubeadm.kubernetes.io/kube-apiserver.advertise-address.endpoint: 10.0.0.10:6443
    kubernetes.io/config.hash: 399b2ed792584c1473b4bd0472bbe7af
    kubernetes.io/config.mirror: 399b2ed792584c1473b4bd0472bbe7af
    kubernetes.io/config.seen: "2022-05-23T04:37:57.986927794Z"
    kubernetes.io/config.source: file
  creationTimestamp: "2022-05-23T04:39:34Z"
  labels:
    component: kube-apiserver
    tier: control-plane
  name: kube-apiserver-master
  namespace: kube-system
  ownerReferences:
  - apiVersion: v1
    controller: true
    kind: Node
    name: master
    uid: fdce9347-67ad-4e88-ae45-39c9aaa738d8
  resourceVersion: "55017343"
  selfLink: /api/v1/namespaces/kube-system/pods/kube-apiserver-master
  uid: 2b4d6d1a-68fc-4c80-b1c6-e57e81d129be
spec:
  containers:
  - command:
    - kube-apiserver
    - ...
    image: k8s.gcr.io/kube-apiserver:v1.22.2
```

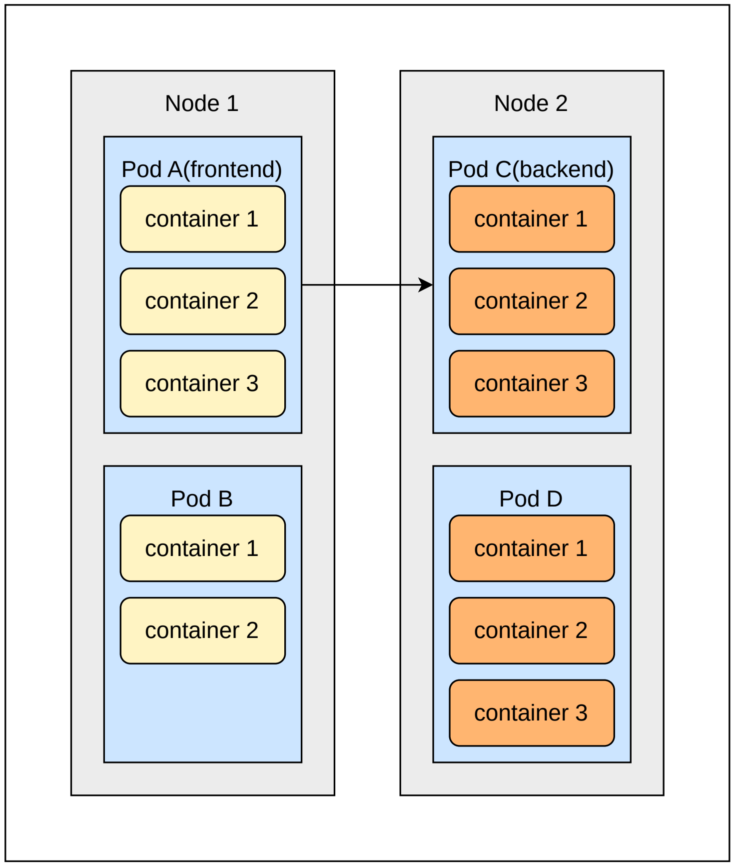
Mirror Pod

How the `kube-proxy` works

The `kube-proxy` is a critical component in a Kubernetes cluster that runs on each node. It provides load balancing between Pods running on different nodes.

In a Kubernetes cluster, every Pod gets its IP and can communicate with each other as they run across nodes. This is implemented by the CNI (Container Network Interface).

Let's suppose we're running an application with two microservices, `frontend` and `backend`. The Pods are running on separate nodes. The `frontend` Pod can reach the `backend` Pod by using its IP address. However, Pods are ephemeral. There is no guarantee that the `backend` Pod always has this same IP address. They may be created and terminated anytime to match the desired state. Moreover, there may be multiple `backend` Pods running on different nodes for load balancing. We can not notify the `frontend` of all these IP changes.



Service discovery across nodes

This problem can be solved with a registry center. To enable communication between Pods in a Kubernetes cluster, the program running inside the Pod needs to include a client that can interact with the cluster's service registry or DNS resolver.

Kubernetes solved this by introducing `Service`. The `Service` abstraction defines a policy used to access a group of Pods. This is usually specified by label selectors. In this way, the `frontend` Pods do not need to be aware of or keep track of all the `backend` Pods themselves. The `Service` abstraction gives a stable, virtual IP address (also called the cluster IP) and a single DNS name for all the matching Pods. This IP address has the same lifecycle as the `Service`, regardless of the changes to the Pods it routes to. The `Service` abstraction is a perfect decoupling that allows to scale out or replace back-end services when necessary. It provides discoverability and dramatically simplifies the design of microservices.

As long as we need to visit another `Service` in a Pod, we only need to configure it with the desired `Service`. In the case above, we only need to provide a back-end `Service` for the `frontend` Pod.

Now, the `kube-proxy` comes into play. Its job is to keep watching `Service/Endpoints/EndpointSlice` and create appropriate rules on each node that can route requests to the `backend` Pod matching those services. The `kube-proxy` itself doesn't route these packets.

Currently, there are four varieties of the `ProxyMode` in the `kube-proxy`.

- `userspace`
- `iptables`
- `ipvs`
- `kernelspace`

It can be configured with a different mode on start-up.

Back

The Manufactory of Kubernetes: The kube-controller-...

Next

Quiz on Kubernetes Architecture