How the CRI Works

Learn how the CRI works in Kubernetes.

We'll cover the following

Overview

In Kubernetes, the smallest deployable and schedulable unit is a Pod. One or more containers are grouped into one Pod. Such a Pod design can well support multiple cooperating processes for a cohesive unit of Service. Each process runs in a separate container. Multiple containers serve as a standalone Service or application.

But how does `kubelet` turn a Pod to multiple running containers?

The role of `kubelet`

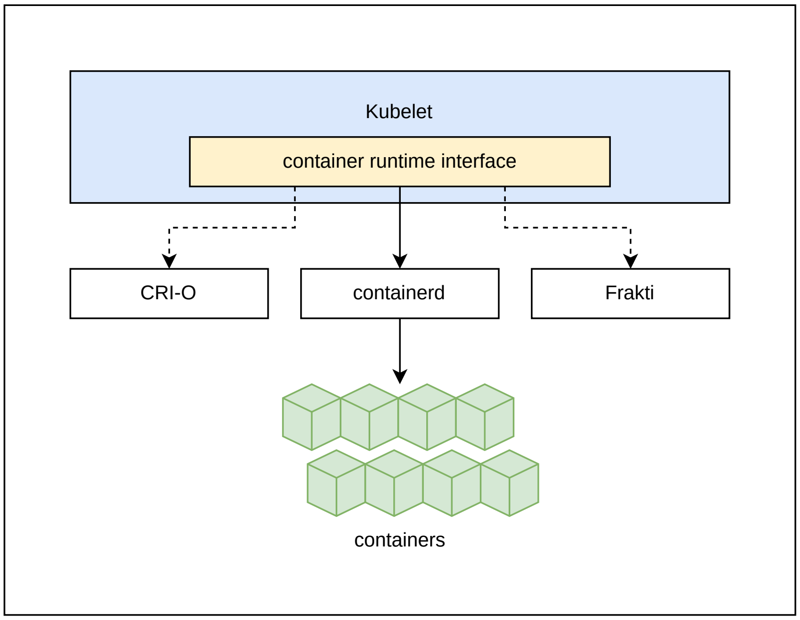
The `kubelet` agent watches the `kube-apiserver` as new Pods get scheduled to the current node. It processes the specification of a Pod and ensures all the specified containers for the Pod are up and running as declared.

To create a container, `kubelet` needs a container runtime, such as Docker. The `kubelet` itself doesn't do low-level container management. All the containers are created by a container runtime. For a long time, Docker was used as the default container runtime and was integrated as the in-tree runtime. However, as Docker itself continues to evolve, having it integrated occasionally broke Kubernetes. It was hard to release a stable Kubernetes. Moreover, it was impossible to provide backward compatibility. Such tight coupling also made it hard to integrate with other container runtimes.

Therefore, it is essential to make `kubelet` and container runtimes loosely coupled. That's why the Container Runtime Interface (CRI) was proposed.

How does the CRI fit in?

The CRI defines a group of protobuf API, specifications, and libraries that let `kubelet` interact with various container runtimes to create and run containers. The image below shows how `kubelet` works with the CRI to manage the whole lifecycle of containers, including creating, starting, deleting, and so on.



How the CRI fits in

With the CRI, `kubelet` can talk to any container runtimes that are compliant with the Open Container Initiative (OCI). The OCI currently defines three specifications—OCI image format, runtime specification, and distribution specification. A seed low-level container runtime called runc is also provided. Most container runtimes, such as containerd and CRI-O, build on top of it. For container runtimes other than runc, it can also be used for a compatibility measure. At a high level, when a container runtime receives the gRPC request, it will download an OCI image and unpack into an OCI Runtime filesystem bundle. This bundle will then be run by an OCI runtime.

In the OCI specifications, the Kubernetes concept Pod, where we can create and start a Pod-level sandbox, is introduced as well. Container runtimes are responsible for managing the whole lifecycle of these sandbox containers, such as creating, reclaiming network resources (such as IP addresses), and so on. When creating an application container, the corresponding sandbox ID is needed. Below is a code snippet illustrating how `kubelet` calls a `CreateContainer` gRPC request to the container runtime. By specifying such a `podSandBoxID`, the container runtimes will make sure that all the containers in a Pod share the same network namespace.

```
// From pkg/kubelet/cri/remote/remote_runtime.go

// CreateContainer creates a new container in the specified PodSandbox.
func (r *remoteRuntimeService) CreateContainer(podSandBoxID string, config *runtimeapi.ContainerConfig,
    klog.V(10).InfoS("[RemoteRuntimeService] CreateContainer", "podSandboxID", podSandBoxID, "timeout",
    ctx, cancel := getContextWithTimeout(r.timeout)
    defer cancel()

    if r.useV1API() {
        return r.createContainerV1(ctx, podSandBoxID, config, sandboxConfig)
    }

    return r.createContainerV1alpha2(ctx, podSandBoxID, config, sandboxConfig)
```

```
    return r.createContainerV1alpha2(ctx, podSandBoxID, config, sandboxConfig)
}

func (r *remoteRuntimeService) createContainerV1alpha2(ctx context.Context, podSandBoxID string, config
    resp, err := r.runtimeClientV1alpha2.CreateContainer(ctx, &runtimeapiV1alpha2.CreateContainerReques
        PodSandboxId:  podSandBoxID,
        Config:        v1alpha2ContainerConfig(config),
        SandboxConfig: v1alpha2PodSandboxConfig(sandboxConfig),
    })
    if err != nil {
        klog.ErrorS(err, "CreateContainer in sandbox from runtime service failed", "podSandboxID", podSa
        return "", err
    }

    klog.V(10).InfoS("[RemoteRuntimeService] CreateContainer", "podSandboxID", podSandBoxID, "container
    if resp.ContainerId == "" {
        errorMessage := fmt.Sprintf("ContainerId is not set for container %q", config.Metadata)
        err := errors.New(errorMessage)
        klog.ErrorS(err, "CreateContainer failed")
```

How kubelet creates a container via the CRI

Conclusion

The OCI standardizes the use of container image formats, container runtimes, and distribution specification. In Kubernetes, `kubelet` only works with container runtimes that are compliant with the OCI standard.

The CRI lets `kubelet` work with a variety of container runtimes; it's quite flexible. We can run `kubelet` on different infrastructures only if we've got a compatible container runtime. There are a variety of mature container runtimes that support the CRI, such as containerd, CRI-O, Kata, gVisor, and so on.

Back
Introduction to the Container Runtime Interface (CRI)

Next
Run Multiple Container Runtimes