Introduction to the Container Runtime Interface (CRI)

Get introduced to the Container Runtime Interface (CRI) in Kubernetes.

In Kubernetes, Pods are introduced to describe a cohesive unit of Service, where multiple cooperating processes are running as containers with shared storage, network, etc. To better manage these Pods, a component called `kubelet` is running as a daemon on every node. The `kubelet` plays a crucial role in the Kubernetes system. It is the primary implementer of Pods and the driver of the container execution layer. It is `kubelet` that makes Pods into a group of running containers. However, `kubelet` itself doesn't directly manage the lifecycle of containers. A **container engine**, also known as a **container runtime**, is invoked to do low-level container management, such as loading container images from a registry, mounting a volume to a container, assigning network interfaces, and so on.

Some common container runtimes are containerd, CRI-O, and Docker. However, Kubernetes support for Docker via `dockershim` is marked as deprecated since v1.20 and fully removed in v1.24. As part of the effort to support various container runtimes, the community introduced a plugin interface called CRI (Container Runtime Interface) since v1.5, when Docker and `rkt` were built-in integrated into the source code of `kubelet`.

Why Kubernetes needs CRI

Let's look back at the early days of Kubernetes. At first, Docker was the only container runtime used by Kubernetes. Later, `rkt` was introduced as an alternative. Not long after, the community realized that this was challenging.

This kind of deep coupling made it hard to evolve Kubernetes, and container runtimes evolved as well. This brought great trouble for developers and maintainers. For every small fix for container runtimes, we need to upgrade the dependency, retest, and recompile `kubelet`. It was impossible to get a stable version, not to mention backwards compatibility.

What's worse, it was harder to bring in other container runtimes. This would help create a monopoly on the container runtime when using Kubernetes.

Thus, a standard interface is preferred—that standard interface is CRI. With that, it's easy for `kubelet` to talk with a wide variety of container runtimes. The core part of `kubelet` becomes clearer, since it does not need to talk directly to any specific container runtime. Rather, it communicates with them via gRPC requests. With CRI, `kubelet` becomes more extensible.

What is CRI

The **Container Runtime Interface (CRI)** defines the main gRPC protocol that is used for communications between `kubelet` and a container runtime, as shown below:

```
 1  // Runtime service defines the public APIs for remote container runtimes
 2  service RuntimeService {
 3      // Version returns the runtime name, runtime version, and runtime API version.
 4      rpc Version(VersionRequest) returns (VersionResponse) {}
 5
 6      // RunPodSandbox creates and starts a pod-level sandbox. Runtimes must ensure
 7      // the sandbox is in the ready state on success.
 8      rpc RunPodSandbox(RunPodSandboxRequest) returns (RunPodSandboxResponse) {}
 9      // StopPodSandbox stops any running process that is part of the sandbox and
10      // reclaims network resources (e.g., IP addresses) allocated to the sandbox.
11      // If there are any running containers in the sandbox, they must be forcibly
12      // terminated.
13      // This call is idempotent, and must not return an error if all relevant
14      // resources have already been reclaimed. kubelet will call StopPodSandbox
15      // at least once before calling RemovePodSandbox. It will also attempt to
16      // reclaim resources eagerly, as soon as a sandbox is not needed. Therefore,
17      // multiple StopPodSandbox calls are expected.
18      rpc StopPodSandbox(StopPodSandboxRequest) returns (StopPodSandboxResponse) {}
19      // RemovePodSandbox removes the sandbox. If there are any running containers
20      // in the sandbox, they must be forcibly terminated and removed.
21      // This call is idempotent, and must not return an error if the sandbox has
22      // already been removed.
23      rpc RemovePodSandbox(RemovePodSandboxRequest) returns (RemovePodSandboxResponse) {}
24      // PodSandboxStatus returns the status of the PodSandbox. If the PodSandbox is not
25      // present, returns an error.
26      rpc PodSandboxStatus(PodSandboxStatusRequest) returns (PodSandboxStatusResponse) {}
27      // ListPodSandbox returns a list of PodSandboxes.
28      rpc ListPodSandbox(ListPodSandboxRequest) returns (ListPodSandboxResponse) {}
29
30      // CreateContainer creates a new container in specified PodSandbox
```

CRI gRPC interfaces

The Protocol Buffers API above defines two gRPC services—`RuntimeService` (**lines 1–93**) and `ImageService` (**lines 95–111**). The `RuntimeService` provides RPCs to manage the whole lifecycle of Pods and containers, such as creating a sandbox container or deleting a container. Other interactions (`exec/attach/port-forward`) with containers are supported as well. On the other hand, the `ImageService` contains RPCs to manage container images, such as pulling an image from a registry or deleting an image. Any container runtimes that implement the CRI can be used together with `kubelet`.

With the CRI, Kubernetes has the flexibility to run with a variety of container runtimes. Thus, we can choose one that best meets our business needs. Some companies choose to use containerd as the runtime, while some use the Kata runtime to isolate the containers from the host machine due to operational requirements.

Back

Quiz on the Container Network Interface

Next

How the CRI Works