How the CNI Works

Learn how the CNI works in Kubernetes.

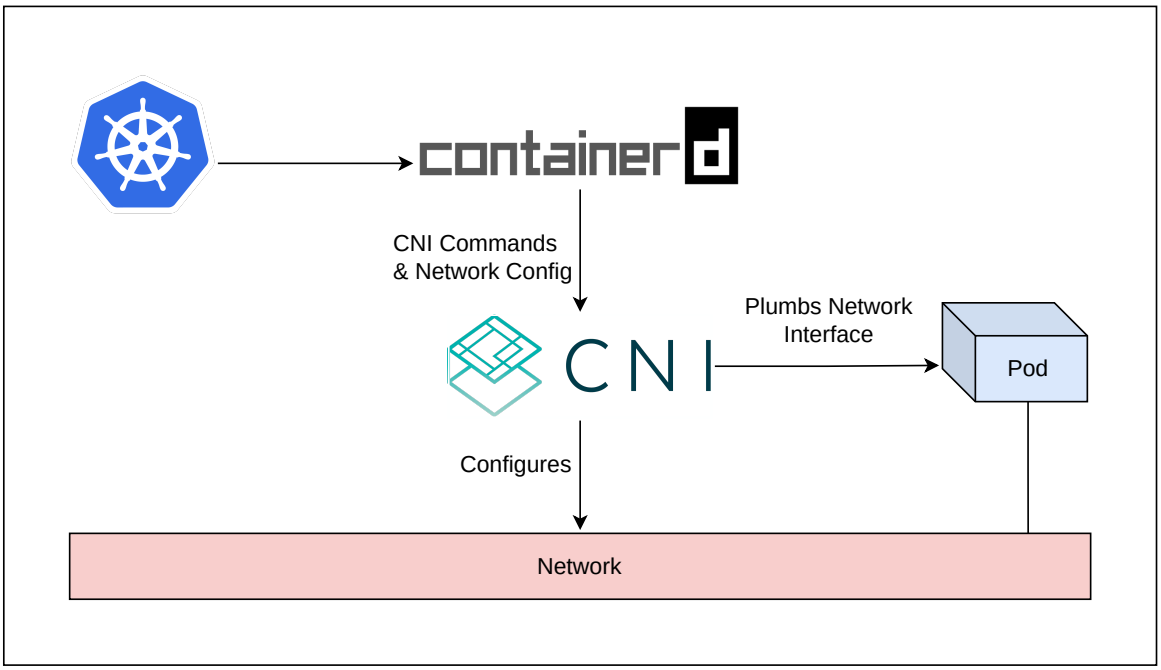
We'll cover the following

Overview

The CNI defines a container networking framework that allows the dynamic configuring of the network through a group of specifications and libraries.

How the CNI fits in

The Kubernetes networking model is implemented by the container runtime on each node, instead of `kubelet`. Just as the graph below shows, the container runtime (e.g. `containerd`) calls the executable CNI plugin to configure the network and add or remove a network interface to or from the container's networking namespace (`netns`). As we mentioned previously, every Pod gets its own unique IP address, and it is the CNI plugin that has the responsibility to allocate the IP address and assign it to a Pod.



How the CNI fits in

The following code snippets show how containerd invokes CNI plugins to set up the network for a Pod:

```
// From github.com/containerd/containerd/pkg/cri/sbserver/sandbox_run.go

// getNetworkPlugin returns the network plugin to be used by the runtime class
// defaults to the global CNI options in the CRI config
func (c *criService) getNetworkPlugin(runtimeClass string) cni.CNI {
    if c.netPlugin == nil {
        return nil
    }
    i, ok := c.netPlugin[runtimeClass]
    if !ok {
        if i, ok = c.netPlugin[defaultNetworkPlugin]; !ok {
            return nil
        }
    }
    return i
}

// setupPodNetwork setups up the network for a pod
func (c *criService) setupPodNetwork(ctx context.Context, sandbox *sandboxstore.Sandbox) error {
    var (
        id        = sandbox.ID
        config    = sandbox.Config
        path      = sandbox.NetNSPath
        netPlugin = c.getNetworkPlugin(sandbox.RuntimeHandler)
    )
    if netPlugin == nil {
        return errors.New("cni config not initialized")
    }

    opts, err := cniNamespaceOpts(id, config)
    if err != nil {
```

How containerd invokes the CNI to set up the network for a Pod

With CNI plugins, we can make the creation and administration of container networking easier and pluggable.

Execution flow of the CNI plugins

Let's dive deep into the execution flow of the CNI plugins.

When the container runtime (e.g. `containerd`) performs network operations on a container, such as creating or deleting a container, it calls the CNI plugin with the desired command. These CNI plugins are executable. Usually, we put them in the folder `/opt/cni/bin` by default. Below is such a sample command. Here, the `CNI_COMMAND` could be either `ADD`, `DEL`, `CHECK`, or `VERSION`.

```
$ CNI_COMMAND=ADD \
  CNI_CONTAINERID=$id \
  CNI_NETNS=/proc/$pid/ns/net \
  CNI_PATH=/opt/cni/bin \
  CNI_IFNAME=eth0 \
  my-plugin < my-config
```

Sample command to call the CNI plugin

Here, the container runtime also needs to provide related network configuration and container-specific data. The configuration file is in JSON format, as shown below. By default, we put the configuration file in the folder `/etc/cni/net.d`. In the configuration file, we specify the `cniVersion` to indicate which version the caller is using. Multiple chaining plugins can be configured here. The container runtime caller will look for matching plugins by the specified plugin type, such as `flannel` and `portmap`, and call each one in turn.

```
{
  "name": "cbr0",
  "cniVersion": "0.3.1",
  "plugins": [
    {
      "type": "flannel",
      "delegate": {
        "hairpinMode": true,
        "isDefaultGateway": true
      }
    },
    {
      "type": "portmap",
      "capabilities": {
        "portMappings": true
      }
    }
  ]
}
```

Sample CNI configuration file

The result will be reported back in JSON format as well, after performing the required operations.

```
{
  "cniVersion": "0.3.1",
  "interfaces": [{
    "name": "eth0",
    "mac": "ce:60:4c:b9:3a:06",
    "sandbox": "/proc/15116/ns/net"
  }],
  "ips": [{
    "version": "4",
    "address": "10.240.0.6/24",
    "gateway": "10.240.0.1",
    "interface": 0
  }]
}
```

Sample CNI result

Conclusion

The objective of CNI specification is to standardize the containers with the underlying network and make it pluggable and portable. There are dozens of network solutions provided by various vendors to address different infrastructures and environments. Some of those plugins only provide basic features like adding and removing network interfaces, while others provide more sophisticated features. Regardless of where you are deploying your Kubernetes cluster, appropriate plugins can be found to meet your needs.

Back

Understanding the Kubernetes Networking Model

Next

Implementing a CNI plugin