Introduction to CNI

Get introduced to the Container Network Interface (CNI) in Kubernetes.

We'll cover the following

The Container Network Interface

Kubernetes is a highly modular open-source project. It provides a lot of flexibility, which makes it easier for us to opt-in our customizations. In the Kubernetes ecosystem, many projects and frameworks have come together to help manage containers easier, more flexible and adaptive.

The **Container Network Interface (CNI)**, which helps simplify the container networking in Kubernetes, is one of those projects. CNI is a standard for the common network interface, describing the simplest possible interface between container runtime and network implementation. It is one of the CNCF projects, consisting of the CNI specification documents, a set of references, and example plugins.

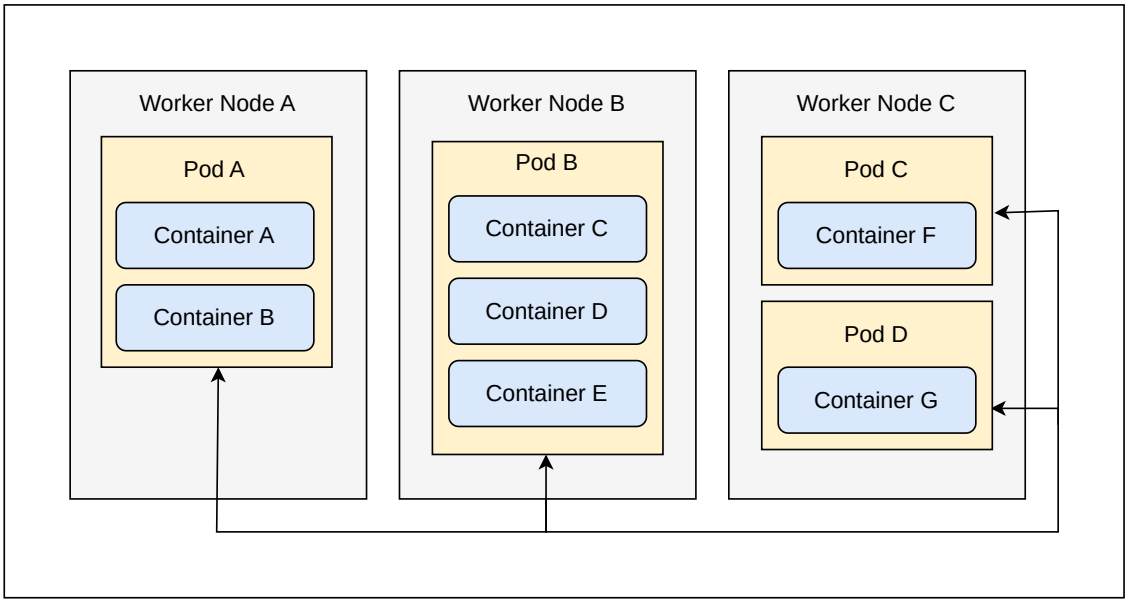
But why do we need CNI?

Why we need CNI

Managing networking is quite difficult, as has been proven time and time again in traditional application architectures. In many companies, developers create the applications, while operators are responsible for running them and making sure they are run well. This creates a segregation of duties. As time goes on, the applications evolve and get more complicated, the needs for the networking infrastructure change, and gaps begin to form. It is also difficult to mitigate these gaps when we want to port our applications to other platforms or infrastructures.

This is also true for containers. One of the biggest benefits of containers is that we can build once and run everywhere with consistent behaviors. Networking in containers matters a lot. Our programs become useless if our containers get coupled with the underlying network. Thus, many container runtimes and orchestrators need to solve this problem of making the network layer modular and pluggable.

Before we talk about CNI, let's see how Pods in Kubernetes communicate with each other. These Pods could be placed on the same node or different nodes. The nodes could also be in the same subnet, different subnets, and even different datacenters. As the image below shows, regardless of the kind of network, Kubernetes aims to connect these Pods in a seamless, agnostic, and routable way.



Pod-to-Pod Network

To make it open standard and avoid building self-owned wheels, the Kubernetes community initiates the open source CNI project to make container networking solutions integratable with various orchestrators and runtimes (such as containerd, CRI-O, gVisor, etc.). CNI defines a generic plugin-based networking solution, where a set of plugins and libraries are provided as well. It makes an abstract layer between the container execution and networking.

With CNI, administrators reduce the overhead to generate and maintain network configurations manually. This also helps provide a consistent and reliable network across all Pods, no matter where we run our applications.

Like other distributed systems, Kubernetes relies on the container network to provide connectivity between Pods and services, as well as for exposing endpoints for external access. As long as Kubernetes vendors and developers follow the CNI specification, the `IP-per-Pod` network model can be assured within Kubernetes clusters. We can access every Pod, assign IP addresses, set routing rules, visit in-cluster services, and so on.

What is CNI?

The CNI project comes with two major parts.

The CNI specification

One major part of the CNI project is the CNI specification. The specification outlines the interface between the container runtimes and CNI plugins. Below is an example networking configuration file containing directives for both the container runtime and the CNI plugins to consume:

```
{
  "cniVersion": "1.0.0",
  "name": "dbnet",
  "plugins": [
    {
      "type": "bridge",
      // plugin specific parameters
      "bridge": "cni0",
      "keyA": ["some more", "plugin specific", "configuration"],

      "ipam": {
        "type": "host-local",
        // ipam specific
        "subnet": "10.1.0.0/16",
        "gateway": "10.1.0.1",
        "routes": [
            {"dst": "0.0.0.0/0"}
        ]
      },
      "dns": {
        "nameservers": [ "10.1.0.1" ]
      }
    },
    {
      "type": "tuning",
      "capabilities": {
        "mac": true
      },
      "sysctl": {
        "net.core.somaxconn": "500"
```

CNI network configuration example

To facilitate the container runtime interacting with CNI plugins, a CNI runtime implementation `libcni` (which is written with Go) is provided as well. So, all the container runtimes written with Go (such as containerd) can directly import this module and interact with CNI plugins. The implementation is as follows:

```
type CNI interface {
    AddNetworkList(ctx context.Context, net *NetworkConfigList, rt *RuntimeConf) (types.Result, error)
    CheckNetworkList(ctx context.Context, net *NetworkConfigList, rt *RuntimeConf) error
    DelNetworkList(ctx context.Context, net *NetworkConfigList, rt *RuntimeConf) error
    GetNetworkListCachedResult(net *NetworkConfigList, rt *RuntimeConf) (types.Result, error)
    GetNetworkListCachedConfig(net *NetworkConfigList, rt *RuntimeConf) ([]byte, *RuntimeConf, error)

    AddNetwork(ctx context.Context, net *NetworkConfig, rt *RuntimeConf) (types.Result, error)
    CheckNetwork(ctx context.Context, net *NetworkConfig, rt *RuntimeConf) error
    DelNetwork(ctx context.Context, net *NetworkConfig, rt *RuntimeConf) error
    GetNetworkCachedResult(net *NetworkConfig, rt *RuntimeConf) (types.Result, error)
    GetNetworkCachedConfig(net *NetworkConfig, rt *RuntimeConf) ([]byte, *RuntimeConf, error)

    ValidateNetworkList(ctx context.Context, net *NetworkConfigList) ([]string, error)
    ValidateNetwork(ctx context.Context, net *NetworkConfig) ([]string, error)
}
```

The CNI interface

Likewise, a reference plugin scaffold/library `skel` is provided as well. By building on top of this scaffold, writing CNI plugins becomes easier.

A set of reference and example plugins

The other major part of the CNI project is a set of reference and example plugins, which are provided.

A group of interface creating plugins are built-in, such as `bridge`, `macvlan`, `vlan`, `ptp`, and so on. With these plugins, we can easily create bridges, network devices/taps, and VLANs. In addition, IPAM plugins (`dhcp`, `host-local`, and `static`) ensure a clear picture of the IP addresses in use. Some chained plugins, such as `portmap`, `bandwidth`, `tuning`, etc., are shipped together as well.

These plugins help avoid work duplications when handling basic network functionalities. We can easily build our own CNI plugins on top of these plugins.

Back

Quiz on Aggregated APIServer

Next

Understanding the Kubernetes Networking Model