**Introduction to CNI**

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

**We'll cover the following**

* [The Container Network Interface](https://www.educative.io/courses/programming-with-kubernetes/introduction-to-cni" \l "The-Container-Network-Interface)
* [Why we need CNI](https://www.educative.io/courses/programming-with-kubernetes/introduction-to-cni" \l "Why-we-need-CNI)
* [What is CNI?](https://www.educative.io/courses/programming-with-kubernetes/introduction-to-cni" \l "What-is-CNI)
  + [The CNI specification](https://www.educative.io/courses/programming-with-kubernetes/introduction-to-cni" \l "The-CNI-specification)
  + [A set of reference and example plugins](https://www.educative.io/courses/programming-with-kubernetes/introduction-to-cni" \l "A-set-of-reference-and-example-plugins)

**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.

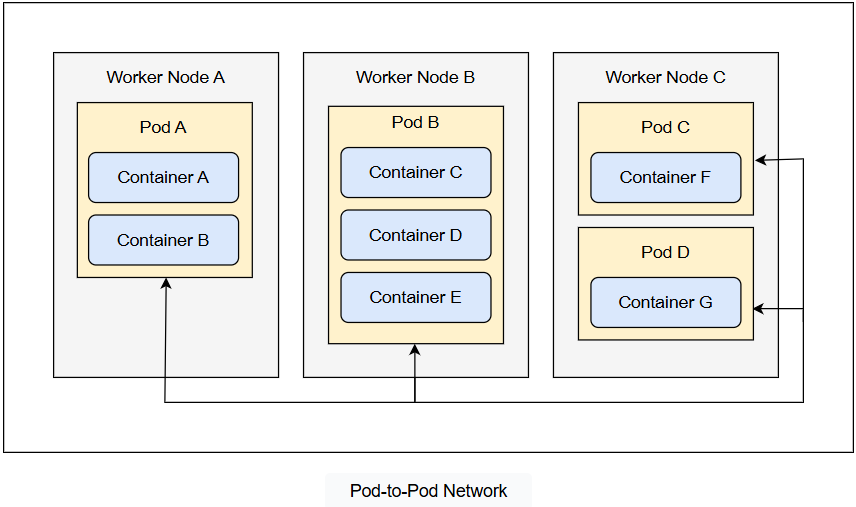
But why do we need CNI?

**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.



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"

}

},

{

"type": "portmap",

"capabilities": {"portMappings": true}

}

]

}

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)

}

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.

# Understanding the Kubernetes Networking Model

Understand the container networking model in Kubernetes.

We'll cover the following

* [Overview](https://www.educative.io/courses/programming-with-kubernetes/understanding-the-kubernetes-networking-model" \l "Overview)
* [The Kubernetes networking model](https://www.educative.io/courses/programming-with-kubernetes/understanding-the-kubernetes-networking-model" \l "The-Kubernetes-networking-model)
* [Networking scenarios in Kubernetes](https://www.educative.io/courses/programming-with-kubernetes/understanding-the-kubernetes-networking-model" \l "Networking-scenarios-in-Kubernetes)
  + [Container to container communication](https://www.educative.io/courses/programming-with-kubernetes/understanding-the-kubernetes-networking-model" \l "Container-to-container-communication)
  + [Intra-node Pod to Pod communication](https://www.educative.io/courses/programming-with-kubernetes/understanding-the-kubernetes-networking-model" \l "Intra-node-Pod-to-Pod-communication)
  + [Inter-node Pod to Pod communication](https://www.educative.io/courses/programming-with-kubernetes/understanding-the-kubernetes-networking-model" \l "Inter-node-Pod-to-Pod-communication)
  + [Service to Pod communication](https://www.educative.io/courses/programming-with-kubernetes/understanding-the-kubernetes-networking-model" \l "Service-to-Pod-communication)
  + [External to service networking](https://www.educative.io/courses/programming-with-kubernetes/understanding-the-kubernetes-networking-model" \l "External-to-service-networking)
* [Conclusion](https://www.educative.io/courses/programming-with-kubernetes/understanding-the-kubernetes-networking-model" \l "Conclusion)

## Overview

In Kubernetes, the CNI is seamlessly integrated with kubelet to allow configurable networks between Pods. The network could be overlay or underlay. The **overlay network** encapsulates the network traffic using a virtual interface like VxLan, while the **underlay network** works at the physical layer composed of switches and routers. This decoupling gives us more freedom to choose appropriate network solutions and keeps the Pods portable to heterogeneous infrastructures.

No matter what the underlying infrastructure or network is, the Pods are accessible and can communicate with each other. Before we dive into how the CNI works, it’s worth discussing the Kubernetes networking model. This model is not only a design philosophy, but also has a close relationship with the CNI.

## The Kubernetes networking model

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

These applications share all the resources in the Kubernetes cluster, including the CPU, memory, and network. However, we can’t run applications with the same port on the same node. This is physically impossible. Typically, we need to ensure that two applications on the same node are using non-conflicting ports. Using dynamic port allocation brings a lot of complications, such as how to assign ports dynamically on each node under high concurrency, how to notify other service consumers on the ports, how to maintain the configuration blocks on the changing dynamic port numbers, and so on. To solve these complications, Kubernetes takes an opinionated approach to container networking. In particular, Kubernetes strictly stipulates that any network implementations should follow the guidelines below.

* Every Pod gets its own IP address. The IP address that a Pod sees itself as is exactly the same IP that other Pods see it as. This is the most important and fundamental design philosophy in the Kubernetes networking model. Moreover, this IP is unique in a whole cluster. All the containers of this Pod share this IP. They can run with different ports. To achieve this IP sharing for multiple containers in a Pod, there is a **pause container** (also called a **sandbox container**) for each Pod. This pause container’s purpose is to hold a network namespace (netns), which is shared by all the other containers in this Pod. In this way, the IP address of a Pod doesn’t change even if containers are killed and recreated in place. This is the so called IP-per-Pod model.
* Pods can use Pod IP addresses to communicate with all other Pods on other nodes without network address translation (NAT).
* Agents (such as system daemons, kubelet, etc.) on a node can communicate with all the Pods on it.

## Networking scenarios in Kubernetes

The Kubernetes networking model is concerned with the following four networking scenarios:

* Container to container communications
* Pod to Pod communication
  + Intra-node communication
  + Inter-node communication
* Service to Pod networking
* External to cluster communication

Let’s see how these four scenarios fit in with the Kubernetes networking model.

### Container to container communication

In Linux, each running process communicates within a network namespace, which provides a logic networking stack, including routes, firewall rules, and network devices. All the processes within the namespace share the network stack. With the following command, we can create a new network namespace ns1:

ip netns add ns1

Once the namespace is created, a corresponding mount point for it is created under /var/run/netns, where the namespace is persisted even if we have no processes attached to it.

ls /var/run/netns # or we can run ip netns

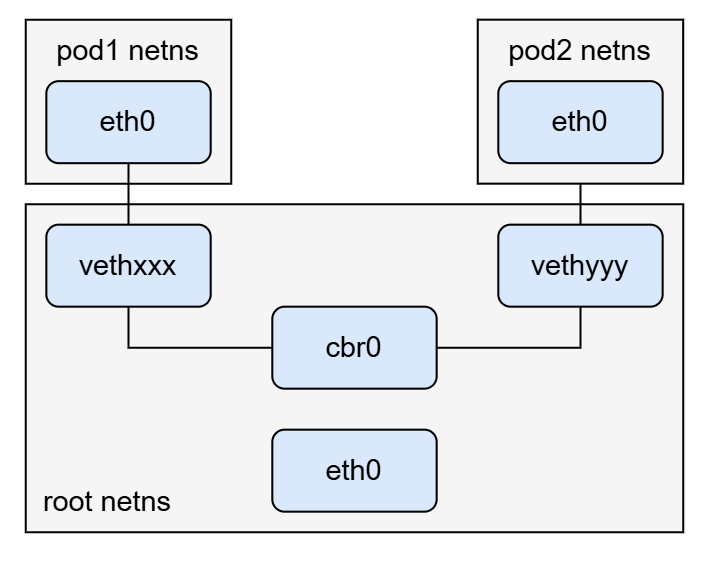
We can list the available network namespaces by running the commands above in the following terminal:

In Linux, every process is attached to the root network namespace by default. The main network interface eth0 is in this root network namespace. Similarly, every Pod has such a self-owned network namespace. Containers within a Pod share the network namespace, such as the IP address, ports, etc. They can communicate with each other via the localhost since they reside in the same network namespace.

### Intra-node Pod to Pod communication

Making sure Pods running on the same node can talk to each other is essential so that we are able to extend the communications across nodes, to the Internet, etc.

Every kubelet node has a designated CIDR range for the Pods running on it. This helps make sure that the Pods never have overlapping IP addresses. Every Pod has its own Ethernet namespace. To connect Pod namespaces, we can use a virtual Ethernet device or a veth pair, as the image below shows. We assign one side of the veth pair to the root namespace and the other side to the Pod network namespace. Such a veth pair pipes two virtual interfaces and allows network traffic to flow between them. To make Pods connective with each other, a virtual network bridge (e.g. cbr0) is used to connect these virtual interfaces. Moreover, Pods can communicate with each other through the root namespace using the Address Resolution Protocol (ARP).



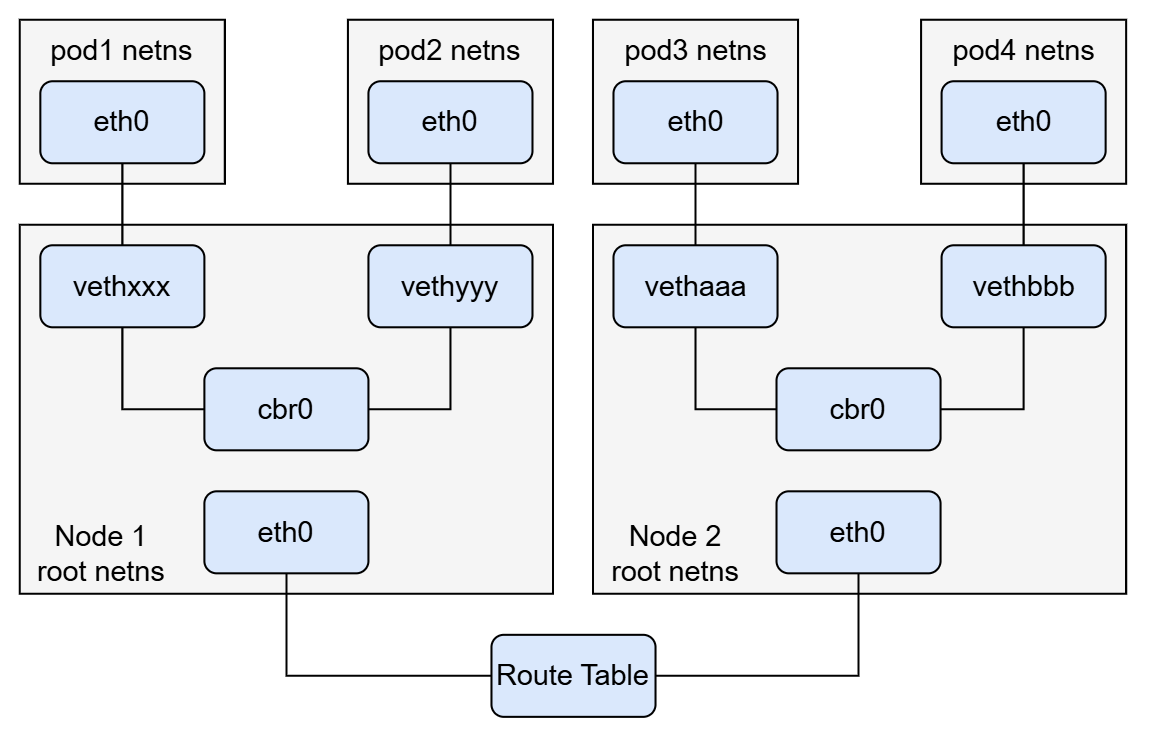
Thus, when we send a packet from pod1 to pod2, the network flow is as follows:

1. The pod1 traffic flows through eth0 to the virtual interface vethxxx in the root network namesapce.
2. The traffic is then passed on to the virtual bridge cbr0, which resolves the destination using an ARP request saying “who has this IP?”
3. The virtual interface vethyyy says it has that IP, so cbr0 knows exactly where to forward the packet.
4. When the packet reaches the virtual interface vethyyy, it is piped to the pod2 network namespace eth0.

### Inter-node Pod to Pod communication

In the Kubernetes networking model, Pods running on different nodes should be able to communicate with each other as well. As long as the traffic can reach the desired Pod running on the other node, Kubernetes doesn’t care how we make it. Only if we can make sure each Pod has a unique IP that won’t conflict with Pods on another node, we have multiple ways to do this, such as ARP across nodes (Layer 2), IP routing across nodes (Layer 3), overlay networks, etc.

Here, we’ve got two nodes, as the image below shows. A packet is going from pod1 running on node1 to pod4 on node2.



1. The packet leaves from pod1 at eth0 and directly enters the root netns at vethxxx.
2. It is then passed on to the virtual bridge cbr0, which sends an ARP request to find who owns the destination IP address.
3. On failure, the virtual bridge cbr0 sends the packet out the default eth0 device in the root network namespace. At this point, the packet enters the network.
4. Assuming that the network can route the packet to the target node based on the CIDR blocks to each node, it routes the packet to the node whose CIDR block has the pod4 IP address.
5. Now, the packet arrives at node2 at the main network interface eth0 in the root network namespace. But how do we let the packet be forwarded to the virtual network bridge cbr0, since the interface eth0 doesn’t have the IP address of pod4? We need to configure the node with enabling IP forwarding, so that the packet will be forwarded to any routes matching the pod4 IP. For now, it can find the virtual bridge cbr0.
6. The virtual bridge receives the packet and makes an ARP request that has the IP address of pod4. Finally, it finds out this IP belongs to vethbbb.
7. The packet flows across the pipe-pair and enters the network namespace of pod4.

### Service to Pod communication

In Kubernetes, Pods are ephemeral, in that they can be scaled up or down on demand. They may also be deleted at any time. This makes it a challenge to get an unchanged endpoint for applications.

Kubernetes introduces a concept service to describe a group of Pods. With service, we can do the following.

1. We can assign a static virtual IP address that can be used for in-cluster accessing. This virtual IP address associates with multiple back-end Pods’ IPs. The **Endpoints controller** (or the **EndpointSlice controller** for Kubernetes with a higher version) takes care of the IP changes of matching Pods.
2. The traffic addressed to this virtual IP can be load balanced to the set of back-end Pods.

Normally, in-cluster load balancing occurs in two ways: iptables and ipvs. This part is handled by the kube-proxy. A series of routing rules will be created to interpret this virtual IP address to a matching back-end Pod.

### External to service networking

In Kubernetes, we have multiple ways to expose our services for external accessing.

We can create a service with type=LoadBalancer, which is backed by external cloud providers. The cloud provider is responsible for provisioning a load balancer for this service. One of the most widely used methods for mapping this load balancer to back-end Pods is NodePort.

Using Ingress is another popular and recommended way to expose our services to the outside of the cluster. The Ingress controller acts as a reverse proxy and load balancer for HTTP and HTTPS. It intercepts all incoming requests to our cluster and routes to different services, based on the rules of the requesting path.

## Conclusion

This networking model brings huge benefits. The network is simple to understand. We don’t need to worry about port conflictions for different applications, and it’s easier to migrate applications from one node to another. Pods can be treated as cattle for real.

There are also other aspects to Kubernetes networking, such as service-based load balancing, exposing internal services to outside, and so on. However, the role of the CNI is merely to facilitate the communications between Pods, regardless of which host they land on. This focus makes CNI specifications simple and widely adaptive to various container runtimes and infrastructures.