Flannel & VXLAN Research

1 Environment Description

If you have followed “01_setup_k8s_cluster_on_qemu/kvm_vms-20240622” and “02_deploy_flannel_on_k8s-20240629”, now, you should have a k8s 1.30 cluster with 1 control-plane (Fedora 40 QEMU/KVM VM) and two nodes (Fedora 40 QEMU/KVM VMs) as follows.

Note: in the context of this document, the Firewalld service is disabled and stopped. If you want to enable Firewalld, please configure the allow the corresponding ports in your Firewalld rules.

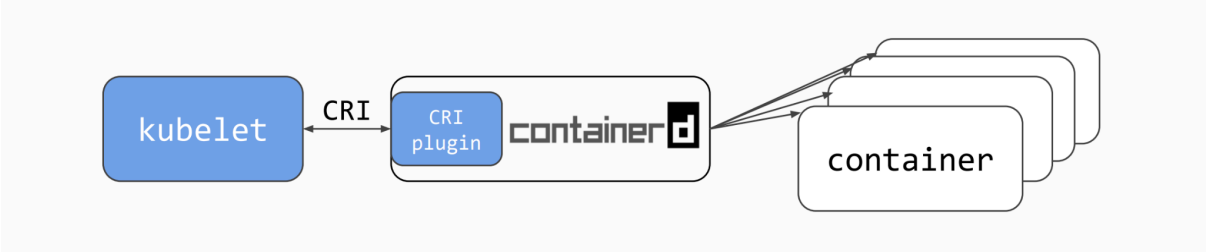
```
[root@controlPlane ~]# kubectl get nodes -o wide
NAME           STATUS   ROLES           AGE   VERSION   INTERNAL-IP       EXTERNAL-IP   OS-IMAGE
controlplane   Ready    control-plane   19h   v1.30.2   192.168.124.249   <none>        Fedora Linux 40 (Cloud Edition)
node1          Ready    <none>          19h   v1.30.2   192.168.124.130   <none>        Fedora Linux 40 (Cloud Edition)
node2          Ready    <none>          19h   v1.30.2   192.168.124.220   <none>        Fedora Linux 40 (Cloud Edition)
[root@controlPlane ~]#
[root@controlPlane ~]# kubectl get pods -A -o wide
NAMESPACE      NAME                                   READY   STATUS    RESTARTS   AGE   IP                NODE
kube-flannel   kube-flannel-ds-jl5s6                  1/1     Running   0          17h   192.168.124.220   node2
kube-flannel   kube-flannel-ds-lbdsd                  1/1     Running   0          17h   192.168.124.249   controlplane
kube-flannel   kube-flannel-ds-mf8p4                  1/1     Running   0          17h   192.168.124.130   node1
kube-system    coredns-7b5944fdcf-6nb7c               1/1     Running   0          19h   10.244.1.3        node1
kube-system    coredns-7b5944fdcf-8vzhb               1/1     Running   0          19h   10.244.1.2        node1
kube-system    etcd-controlplane                      1/1     Running   0          19h   192.168.124.249   controlplane
kube-system    kube-apiserver-controlplane            1/1     Running   0          19h   192.168.124.249   controlplane
kube-system    kube-controller-manager-controlplane   1/1     Running   0          19h   192.168.124.249   controlplane
kube-system    kube-proxy-hckw5                       1/1     Running   0          19h   192.168.124.130   node1
kube-system    kube-proxy-jsvkv                       1/1     Running   0          19h   192.168.124.249   controlplane
kube-system    kube-proxy-qfs2z                       1/1     Running   0          19h   192.168.124.220   node2
kube-system    kube-scheduler-controlplane            1/1     Running   0          19h   192.168.124.249   controlplane
```

2 Pre-knowledge: The architecture of the CRI Plugin for Containerd

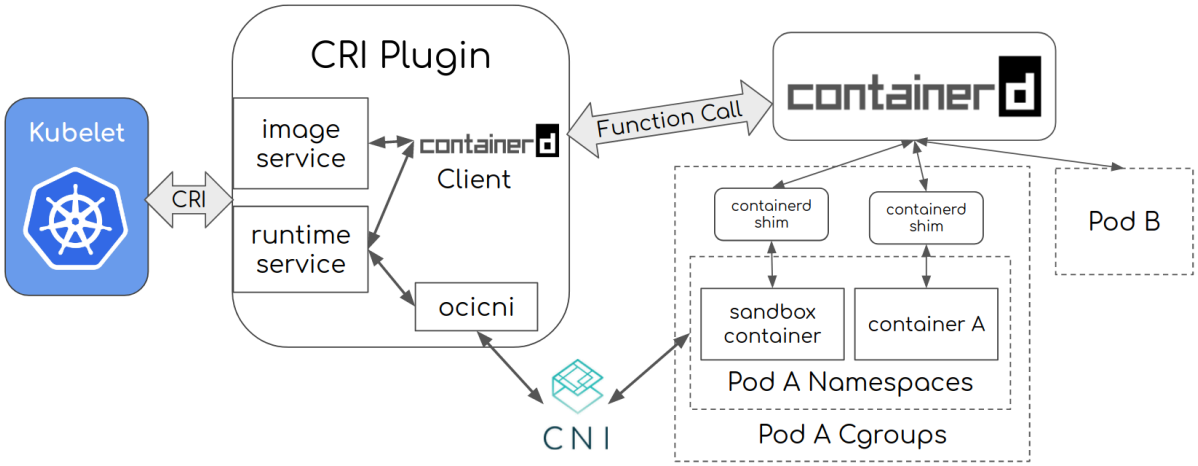
Here, I will briefly introduce the architecture of the CRI plugin for Containerd. More detailed information about Containerd can be seen on the official website[1].

CRI (Container Runtime Interface) is a plugin interface that allows Kubelet to use different container runtimes. Various container runtimes implement the CRI API and this allows users to use the container runtime of their choice in their k8s installation.

The CRI plugin inside Containerd handles all CRI service requests from the Kubelet and uses Containerd internals to manage containers and container images.



The following diagram will show how the CRI plugin works for the case when Kubelet creates a single-container pod:



- Kubelet calls the CRI plugin, via the CRI runtime service API, to create a pod;
- CRI creates the pod's network namespace, and then configures it using CNI;
- CRI uses Containerd internal to create and start a special pause container (the sandbox container) and put that container inside the pod's Cgroups and namespace (steps omitted for brevity);
- Kubelet subsequently calls the CRI plugin, via the CRI image service API, to pull the application container image;
- CRI further uses Containerd to pull the image if the image is not present on the node;
- Kubelet then calls CRI, via the CRI runtime service API, to create and start the application container inside the pod using the pulled container image;
- CRI finally uses Containerd internal to create the application container, put it inside the pod's Cgroups and namespace, and then start the pod's new application container. After these steps, a pod and its corresponding application container are created and running.

3 Flannel Network Plugin Analysis

3.1 Introduction to Flannel

Flannel is responsible for providing a layer 3 IPv4 network between multiple nodes in a cluster. To achieve it, a single binary agent called "flanneld" is deployed on each node, which will allocate a subnet lease to each node out of a larger, preconfigured address space. Flannel uses either the Kubernetes API or ETCD directly to store the network configuration, the allocated subnets, and any auxiliary data (such as the host's public IP). Packets are forwarded using one of several backend mechanisms including VXLAN and various cloud integrations [2].

3.2 Deployment Process of Flannel

All the YAML related information provided below is obtained from the file below: https://github.com/flannel-io/flannel/blob/master/Documentation/kube-flannel.yml.

3.2.1 Namespace Creation

A Kubernetes Namespace named kube-flannel is created to isolate the Flannel resources.

3.2.2 ServiceAccount Creation

A ServiceAccount named flannel is created within the “kube-flannel” namespace. The Flannel pods use this ServiceAccount to authenticate with the Kubernetes API server.

3.2.3 ClusterRole and ClusterRoleBinding

A ClusterRole named flannel defines the permissions for the Flannel pods to interact with the Kubernetes API server.

A ClusterRoleBinding named flannel binds the flannel ClusterRole to the “system:serviceaccount:kube-flannel:flannel” ServiceAccount, granting the Flannel pods the required permissions.

3.2.4 ConfigMap Creation

A ConfigMap named “kube-flannel-cfg” is created within the kube-flannel namespace. This ConfigMap contains the Flannel configuration, including the CNI configuration (cni-conf.json) and the Flannel network settings (net-conf.json).

```
data:
 cni-conf.json: |
   {
     "name": "cbr0",
     "cniVersion": "0.3.1",
     "plugins": [
       {
         "type": "flannel",
         "delegate": {
           "hairpinMode": true,
           "isDefaultGateway": true
         }
       },
       {
         "type": "portmap",
         "capabilities": {
           "portMappings": true
         }
       }
     ]
   }
 net-conf.json: |
   {
     "Network": "10.244.0.0/16",
     "Backend": {
       "Type": "vxlan"
     }
   }
```

3.2.5 DaemonSet Deployment

3.2.5.1 Pod Creation

For each eligible node, the DaemonSet controller creates a pod based on the pod template defined in the DaemonSet. **This includes the main kube-flannel container and the init containers install-cni-plugin and install-cni.**

3.2.5.2 Init Containers Execution

The init containers run in sequence before the main container starts.

- install-cni-plugin: copy the Flannel CNI plugin binary to the host's “/opt/cni/bin” directory.
- install-cni: copy the Flannel CNI configuration file to the host's “/etc/cni/net.d” directory.

3.2.5.3 Main Container Execution

Once the init containers complete their tasks, the main kube-flannel container starts.
The kube-flannel container runs the “/opt/bin/flanneld” command with the arguments “--ip-masq” and “--kube-subnet-mgr”.
The container mounts several volumes, including “/run/flannel”, “/etc/kube-flannel/”, and “/run/xtables.lock”.

3.2.5.4 Network Setup and Volumes Mount

The pod is configured to use the host network (hostNetwork: true), meaning it shares the network namespace with the host node.
The pod mounts several hostPath volumes and a ConfigMap volume for Flannel configuration.

3.3 The Workflow for Flannel to Assign an IP to a Pod

3.3.1 Flannel fetches pod CIDR for the nodes

Flannel fetches pod CIDR and other cluster network metadata from the k8s APIServer for all the nodes in the k8s cluster, and writes it to the subnet.env.

```
[root@node1 ~]# ps -ef | grep flanneld
root        3577    3280  0 Jul01 ?        00:01:24 /opt/bin/flanneld --ip-masq --kube-subnet-mgr
root      676447  668955  0 06:55 pts/1    00:00:00 grep --color=auto flanneld
[root@node1 ~]#
[root@node1 ~]# ifconfig flannel.1
flannel.1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1450
        inet 10.244.1.0  netmask 255.255.255.255  broadcast 0.0.0.0
        inet6 fe80::58dd:ccff:fe32:b43b  prefixlen 64  scopeid 0x20<link>
        ether 5a:dd:cc:32:b4:3b  txqueuelen 0  (Ethernet)
        RX packets 42  bytes 3071 (2.9 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 42  bytes 6197 (6.0 KiB)
        TX errors 0  dropped 81 overruns 0  carrier 0  collisions 0

[root@node1 ~]# cat /run/flannel/subnet.env
FLANNEL_NETWORK=10.244.0.0/16
FLANNEL_SUBNET=10.244.1.1/24
FLANNEL_MTU=1450
FLANNEL_IPMASQ=true
```

3.3.2 Schedule a pod to “node1”

Schedule a pod to “node1” in the k8s cluster according to the following YAML file. The pod is running on the “node1” as expected eventually.

```
[root@controlPlane ~]# cat node1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: alpine-node1
  labels:
    app: alpine-node1
spec:
  containers:
  - name: alpine-container
    image: alpine:latest
    command: ["sleep", "infinity"]
  nodeSelector:
    kubernetes.io/hostname: node1
[root@controlPlane ~]#
[root@controlPlane ~]# kubectl get pods -o wide
NAME           READY   STATUS    RESTARTS   AGE   IP           NODE
alpine-node1   1/1     Running   0          20h   10.244.1.6   node1
```

3.3.3 The CRI plugin creates a network namespace for the pod

Once Kubelet receives the request for pod creation, it will call container runtime (namely Containerd) through the CRI plugin. The CRI plugin will first create a network namespace for the pod, and call the CNI plugin with the CNI config file.

```
[root@node1 ~]# ls /var/run/netns/cni-1ed6db96-cc73-5046-6369-691a88c0d365
/var/run/netns/cni-1ed6db96-cc73-5046-6369-691a88c0d365
[root@node1 ~]#
[root@node1 ~]# ip netns exec cni-1ed6db96-cc73-5046-6369-691a88c0d365 ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host proto kernel_lo
       valid_lft forever preferred_lft forever
2: eth0@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether ea:99:36:81:27:d5 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.244.1.6/24 brd 10.244.1.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::e899:36ff:fe81:27d5/64 scope link proto kernel_ll
       valid_lft forever preferred_lft forever
```

3.3.4 Flannel configures and calls bridge CNI plugin

Bridge CNI plugin creates the cni0 bridge on the node if it does not exist, and creates a veth device with one end inserted into the container network namespace and the other end connected to the cni0 bridge.

```
[root@node1 ~]# ifconfig cni0
cni0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1450
        inet 10.244.1.1  netmask 255.255.255.0  broadcast 10.244.1.255
        inet6 fe80::8c88:52ff:fe57:fb96  prefixlen 64  scopeid 0x20<link>
        ether 8e:88:52:57:fb:96  txqueuelen 1000  (Ethernet)
        RX packets 356327  bytes 28749730 (27.4 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 369153  bytes 34573416 (32.9 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

```
[root@node1 ~]# ifconfig vethec07142f
vethec07142f: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1450
        inet6 fe80::5483:dff:fed0:ccf4  prefixlen 64  scopeid 0x20<link>
        ether 56:83:0d:d0:cc:f4  txqueuelen 0  (Ethernet)
        RX packets 63  bytes 4392 (4.2 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 118  bytes 9436 (9.2 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

```
[root@node1 ~]# cat /var/lib/cni/results/cbr0-7ea064cd1b8c4b2cf904215f224cc8764bfa9b
9ee845e277a606646f4a4e76ed-eth0 | jq '{networkName, interfaces: .result.interfaces,
ips: .result.ips}'
{
  "networkName": "cbr0",
  "interfaces": [
    {
      "mac": "8e:88:52:57:fb:96",
      "name": "cni0"
    },
    {
      "mac": "56:83:0d:d0:cc:f4",
      "name": "vethec07142f"
    },
    {
      "mac": "ea:99:36:81:27:d5",
      "name": "eth0",
      "sandbox": "/var/run/netns/cni-1ed6db96-cc73-5046-6369-691a88c0d365"
    }
  ],
```

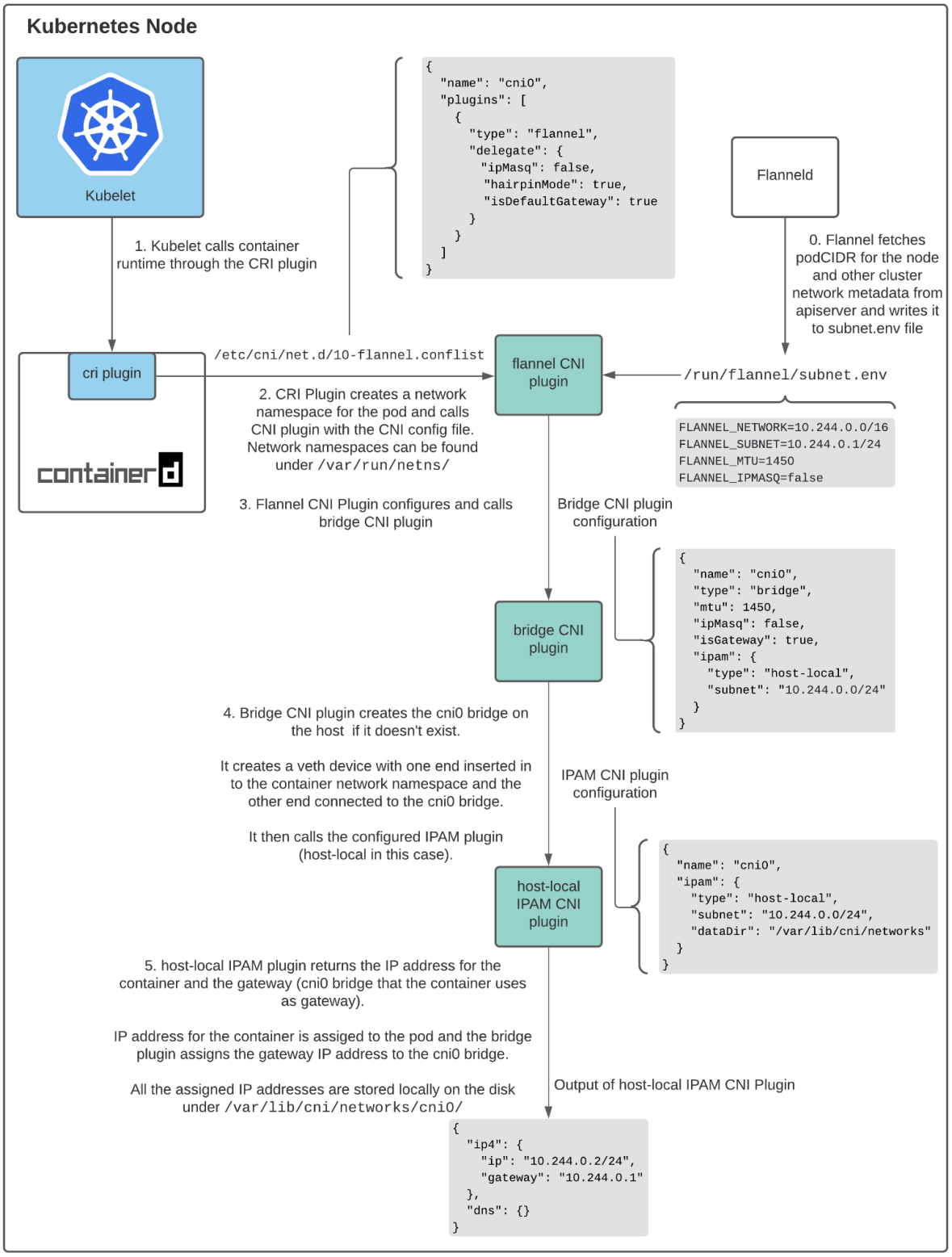
3.3.5 Bridge CNI plugin calls the host-local IPAM plugin

Host-local IPAM plugin returns the IP address for the container and the gateway (cni0 bridge).

```
[root@node1 ~]# cat /var/lib/cni/networks/cbr0/10.244.1.6 ; echo
7ea064cd1b8c4b2cf904215f224cc8764bfa9b9ee845e277a606646f4a4e76ed
eth0
[root@node1 ~]#
[root@node1 ~]# cat /var/lib/cni/flannel/7ea064cd1b8c4b2cf904215f
{
  "ranges": [
    [
      {
        "subnet": "10.244.1.0/24"
      }
    ]
  ],
  "routes": [
    {
      "dst": "10.244.0.0/16"
    }
  ],
  "type": "host-local"
}
```

3.3.6 The whole workflow for Flannel to Assign an IP to a Pod

The diagram above shows how Kubelet, Container Runtime, and CNI Plugins work when scheduling a pod on a node [3].


Kubernetes Node
Kubelet
1. Kubelet calls container runtime through the CRI plugin
{
"name": "cni0",
"plugins": [
{
"type": "flannel",
"delegate": {
"ipMasq": false,
"hairpinMode": true,
"isDefaultGateway": true
}
}
]
}
Flanneld
0. Flannel fetches podCIDR for the node and other cluster network metadata from apiserver and writes it to subnet.env file
cri plugin
containerd
/etc/cni/net.d/10-flannel.conflist
flannel CNI plugin
/run/flannel/subnet.env
FLANNEL_NETWORK=10.244.0.0/16
FLANNEL_SUBNET=10.244.0.1/24
FLANNEL_MTU=1450
FLANNEL_IPMASQ=false
2. CRI Plugin creates a network namespace for the pod and calls CNI plugin with the CNI config file. Network namespaces can be found under /var/run/netns/
3. Flannel CNI Plugin configures and calls bridge CNI plugin
Bridge CNI plugin configuration
{
"name": "cni0",
"type": "bridge",
"mtu": 1450,
"ipMasq": false,
"isGateway": true,
"ipam": {
"type": "host-local",
"subnet": "10.244.0.0/24"
}
}
bridge CNI plugin
4. Bridge CNI plugin creates the cni0 bridge on the host if it doesn't exist.
It creates a veth device with one end inserted in to the container network namespace and the other end connected to the cni0 bridge.
It then calls the configured IPAM plugin (host-local in this case).
IPAM CNI plugin configuration
{
"name": "cni0",
"ipam": {
"type": "host-local",
"subnet": "10.244.0.0/24",
"dataDir": "/var/lib/cni/networks"
}
}
host-local IPAM CNI plugin
5. host-local IPAM plugin returns the IP address for the container and the gateway (cni0 bridge that the container uses as gateway).
IP address for the container is assiged to the pod and the bridge plugin assigns the gateway IP address to the cni0 bridge.
All the assigned IP addresses are stored locally on the disk under /var/lib/cni/networks/cni0/
Output of host-local IPAM CNI Plugin
{
"ip4": {
"ip": "10.244.0.2/24",
"gateway": "10.244.0.1"
},
"dns": {}
}

4 Pod-to-Pod Communication on the Same Node

4.1 Schedule two pods on the same node

Follow the step in “3.3.2”, and schedule the second pod on “node1”.

```
[root@controlPlane ~]# kubectl get pods -o wide
NAME              READY   STATUS    RESTARTS   AGE   IP           NODE
alpine-node1      1/1     Running   0          40h   10.244.1.6   node1
alpine-node1-2nd  1/1     Running   0          7s    10.244.1.8   node1
```

```
[root@node1 ~]# cat /var/lib/cni/results/cbr0-7ea064cd1b8c4b2cf904215f224cc
8764bfa9b9ee845e277a606646f4a4e76ed-eth0 | jq '{networkName, interfaces: .r
esult.interfaces, ips: .result.ips}'
{
  "networkName": "cbr0",
  "interfaces": [
    {
      "mac": "8e:88:52:57:fb:96",
      "name": "cni0"
    },
    {
      "mac": "56:83:0d:d0:cc:f4",
      "name": "vethec07142f"
    },
    {
      "mac": "ea:99:36:81:27:d5",
      "name": "eth0",
      "sandbox": "/var/run/netns/cni-1ed6db96-cc73-5046-6369-691a88c0d365"
    }
  ],
  "ips": [
    {
      "address": "10.244.1.6/24",
```

```
[root@node1 ~]# cat /var/lib/cni/results/cbr0-f11b5f13d65e11cf4d4a475c30544
0ac17d5d06c4dc1afe12dc0e69e669695ab-eth0 | jq '{networkName, interfaces: .r
esult.interfaces, ips: .result.ips}'
{
  "networkName": "cbr0",
  "interfaces": [
    {
      "mac": "8e:88:52:57:fb:96",
      "name": "cni0"
    },
    {
      "mac": "8a:4f:bb:06:16:c0",
      "name": "veth050ae3e6"
    },
    {
      "mac": "36:f9:4e:4a:53:6c",
      "name": "eth0",
      "sandbox": "/var/run/netns/cni-f6b7771a-4a57-4065-d049-df6b29fe739f"
    }
  ],
  "ips": [
    {
      "address": "10.244.1.8/24",
```

4.2 Capture network packets when two pods communicate

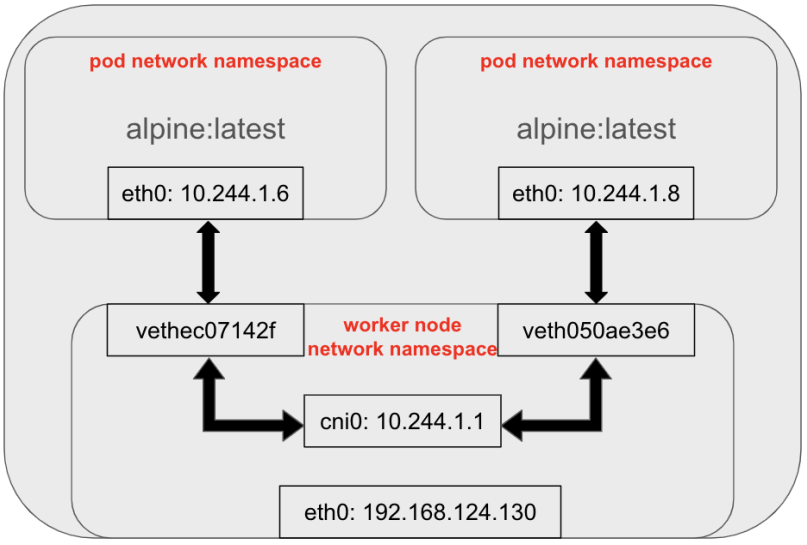
Send ICMP packets from the “alpine-node1” pod to the “alpine-node1-2nd” pod, and capture the network packets from the “cni0” on “node1”.

```
[root@controlPlane ~]# kubectl get pods -o wide
NAME               READY   STATUS    RESTARTS   AGE   IP           NODE    N
alpine-node1       1/1     Running   0          40h   10.244.1.6   node1   <
alpine-node1-2nd   1/1     Running   0          13m   10.244.1.8   node1   <
[root@controlPlane ~]#
[root@controlPlane ~]# kubectl exec -it alpine-node1 -- ping -c1 10.244.1.8
PING 10.244.1.8 (10.244.1.8): 56 data bytes
64 bytes from 10.244.1.8: seq=0 ttl=64 time=0.041 ms

--- 10.244.1.8 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.041/0.041/0.041 ms
```

```
[root@node1 ~]# ifconfig cni0
cni0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1450
        inet 10.244.1.1  netmask 255.255.255.0  broadcast 10.244.1.255
        inet6 fe80::8c88:52ff:fe57:fb96  prefixlen 64  scopeid 0x20<link>
        ether 8e:88:52:57:fb:96  txqueuelen 1000  (Ethernet)
        RX packets 502575  bytes 40546887 (38.6 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 520499  bytes 48740718 (46.4 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

[root@node1 ~]#
[root@node1 ~]# tcpdump -i cni0 icmp -v
dropped privs to tcpdump
tcpdump: listening on cni0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
02:11:54.904047 IP (tos 0x0, ttl 64, id 45682, offset 0, flags [DF], proto ICMP (1), length 84)
    10.244.1.6 > 10.244.1.8: ICMP echo request, id 172, seq 0, length 64
02:11:54.904056 IP (tos 0x0, ttl 64, id 34609, offset 0, flags [none], proto ICMP (1), length 84)
    10.244.1.8 > 10.244.1.6: ICMP echo reply, id 172, seq 0, length 64
^C
2 packets captured
2 packets received by filter
0 packets dropped by kernel
```

4.3 Network flow between two pods on the same node

Two pods on the same host can talk to each other via a Linux bridge “cni0”, to be more specific, a veth device is created for each pod, and one end of this veth device is inserted into the pod, and the other end is connected to the Linux bridge “cni0” on the worker node. All pods on the same node communicate via “cni0”.



5 Pod-to-Pod Communication Across Nodes

5.1 Schedule two pods on the different nodes

Follow the step in “3.3.2”, and schedule another pod on “node2”.

```
[root@controlPlane ~]# kubectl get pods -o wide
NAME           READY   STATUS    RESTARTS   AGE   IP           NODE
alpine-node1   1/1     Running   0          41h   10.244.1.6   node1
alpine-node2   1/1     Running   0          21s   10.244.2.5   node2
```

```
[root@node1 ~]# cat /var/lib/cni/results/cbr0-7ea064cd1b8c4b2cf904215f224cc8764bfa9
b9ee845e277a606646f4a4e76ed-eth0 | jq '{networkName, interfaces: .result.interfaces
, ips: .result.ips}'
{
  "networkName": "cbr0",
  "interfaces": [
    {
      "mac": "8e:88:52:57:fb:96",
      "name": "cni0"
    },
    {
      "mac": "56:83:0d:d0:cc:f4",
      "name": "vethec07142f"
    },
    {
      "mac": "ea:99:36:81:27:d5",
      "name": "eth0",
      "sandbox": "/var/run/netns/cni-1ed6db96-cc73-5046-6369-691a88c0d365"
    }
  ],
  "ips": [
    {
      "address": "10.244.1.6/24",
```

```
[root@node2 ~]# cat /var/lib/cni/results/cbr0-affc9cef2d22cb5a5dfa3cd195bdc8e806c8c
fee62de2923cfaca828f37ba0fd-eth0 | jq '{networkName, interfaces: .result.interfaces
, ips: .result.ips}'
{
  "networkName": "cbr0",
  "interfaces": [
    {
      "mac": "2a:89:72:b1:65:55",
      "name": "cni0"
    },
    {
      "mac": "7a:32:bc:4c:83:60",
      "name": "vethe8416b83"
    },
    {
      "mac": "a2:ff:a3:78:a9:1b",
      "name": "eth0",
      "sandbox": "/var/run/netns/cni-1dd5be6a-becc-8277-fcfd-dc123f05396a"
    }
  ],
  "ips": [
    {
      "address": "10.244.2.5/24",
```

5.2 Capture network packets when two pods communicate

Send ICMP packets from the “alpine-node1” pod to the “alpine-node2” pod, and capture the network packets from the “node1” and “node2”’s interfaces.

```
[root@controlPlane ~]# kubectl get pods -o wide
NAME           READY   STATUS    RESTARTS   AGE     IP           NODE    NOM
alpine-node1   1/1     Running   0          44h     10.244.1.6   node1   <no
alpine-node2   1/1     Running   0          3h35m   10.244.2.5   node2   <no
[root@controlPlane ~]#
[root@controlPlane ~]# kubectl exec -it alpine-node1 -- ping -c1 10.244.2.5
PING 10.244.2.5 (10.244.2.5): 56 data bytes
64 bytes from 10.244.2.5: seq=0 ttl=62 time=0.442 ms

--- 10.244.2.5 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.442/0.442/0.442 ms
```

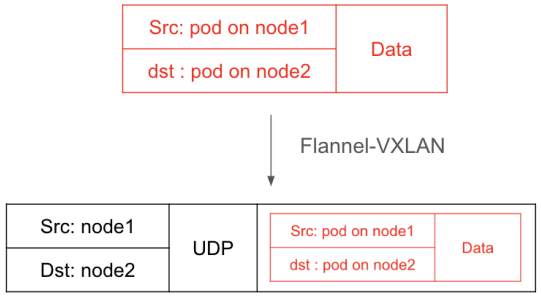
```
[root@node1 ~]# tcpdump -i cni0 icmp -vvv
dropped privs to tcpdump
tcpdump: listening on cni0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
06:47:48.121152 IP (tos 0x0, ttl 64, id 64770, offset 0, flags [DF], proto ICMP (1), length 84)
    10.244.1.6 > 10.244.2.5: ICMP echo request, id 272, seq 0, length 64
06:47:48.121463 IP (tos 0x0, ttl 62, id 56097, offset 0, flags [none], proto ICMP (1), length 84)
    10.244.2.5 > 10.244.1.6: ICMP echo reply, id 272, seq 0, length 64
```

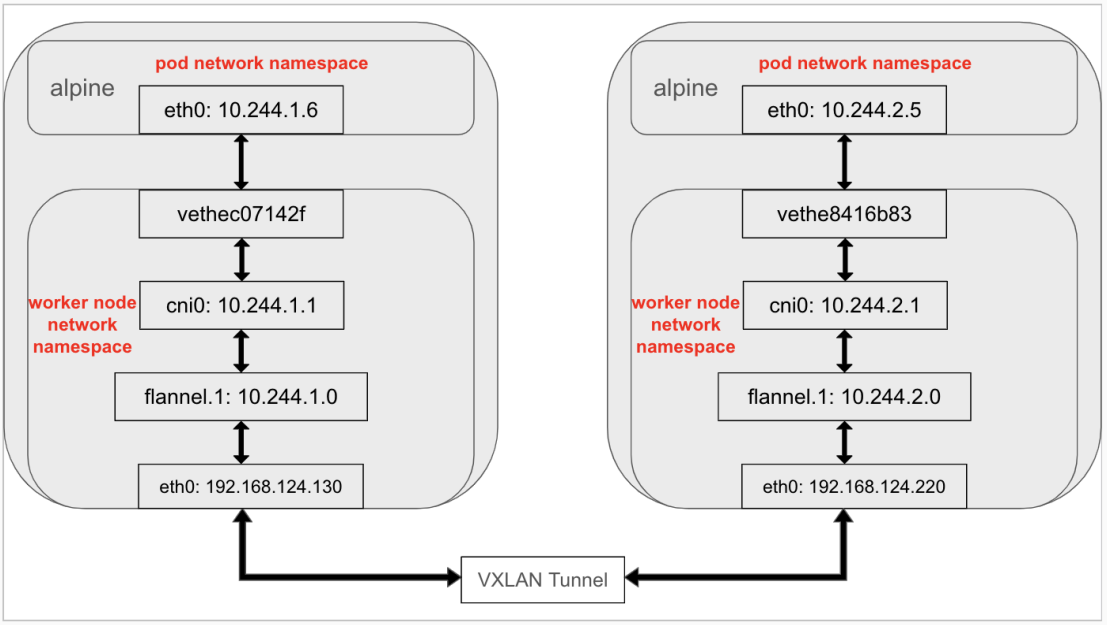
```
[root@node1 ~]# tcpdump -i flannel.1 icmp -vvv
dropped privs to tcpdump
tcpdump: listening on flannel.1, link-type EN10MB (Ethernet), snapshot length 262144 bytes
06:48:05.767068 IP (tos 0x0, ttl 63, id 10996, offset 0, flags [DF], proto ICMP (1), length 84)
    10.244.1.6 > 10.244.2.5: ICMP echo request, id 278, seq 0, length 64
06:48:05.767268 IP (tos 0x0, ttl 63, id 3041, offset 0, flags [none], proto ICMP (1), length 84)
    10.244.2.5 > 10.244.1.6: ICMP echo reply, id 278, seq 0, length 64
```

```
[root@node1 ~]# tcpdump -i eth0 port 8472 -vvv
dropped privs to tcpdump
tcpdump: listening on eth0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
07:16:17.341459 IP (tos 0x0, ttl 64, id 44997, offset 0, flags [none], proto UDP (17), length 134)
    node1.example.org.50095 > node2.example.org.otv: [bad udp cksum 0x7b33 -> 0x3587!] OTV, flags [I] (0x08), overlay 0, instance 1
IP (tos 0x0, ttl 63, id 10840, offset 0, flags [DF], proto ICMP (1), length 84)
    10.244.1.6 > 10.244.2.5: ICMP echo request, id 284, seq 0, length 64
07:16:17.341740 IP (tos 0x0, ttl 64, id 58968, offset 0, flags [none], proto UDP (17), length 134)
    node2.example.org.36635 > node1.example.org.otv: [bad udp cksum 0x7b33 -> 0x6a1b!] OTV, flags [I] (0x08), overlay 0, instance 1
IP (tos 0x0, ttl 63, id 11616, offset 0, flags [none], proto ICMP (1), length 84)
    10.244.2.5 > 10.244.1.6: ICMP echo reply, id 284, seq 0, length 64
07:16:17.422667 IP (tos 0x0, ttl 64, id 45036, offset 0, flags [none], proto UDP (17), length 114)
    node1.example.org.42462 > node2.example.org.otv: [udp sum ok] OTV, flags [I] (0x08), overlay 0, instance 1
IP (tos 0x0, ttl 1, id 32625, offset 0, flags [DF], proto TCP (6), length 64)
    node1.45342 > 10.244.2.5.llmnr: Flags [S], cksum 0xc8f3 (correct), seq 1604337889, win 32430, options [mss 1410,sackOK,TS val 4
cale 7,tfo  cookiereq,nop,nop], length 0
07:16:17.422841 IP (tos 0x0, ttl 64, id 59000, offset 0, flags [none], proto UDP (17), length 142)
    node2.example.org.59779 > node1.example.org.otv: [bad udp cksum 0x7b3b -> 0x0fa3!] OTV, flags [I] (0x08), overlay 0, instance 1
IP (tos 0xc0, ttl 64, id 17923, offset 0, flags [none], proto ICMP (1), length 92)
    10.244.2.0 > node1: ICMP time exceeded in-transit, length 72
        IP (tos 0x0, ttl 1, id 32625, offset 0, flags [DF], proto TCP (6), length 64)
```

5.3 Network flow between two pods across nodes

Using packet encapsulation, two pods running on different hosts can talk to each other via their IP addresses. Flannel supports this through VXLAN which wraps the original packet inside a UDP packet and sends it to the destination.





References

[1] https://github.com/containerd/containerd/blob/main/docs/cri/architecture.md
[2] https://github.com/flannel-io/flannel
[3] https://ronaknathani.com/blog/2020/08/how-a-kubernetes-pod-gets-an-ip-address/