Kubernetes 네트워크 이해하기 (1)

: 컨테이너 네트워크부터 CNI까지

전효준 (https://hyojun.me)

다룰 내용

1. 컨테이너 네트워크의 동작 원리
2. Pod 네트워크의 동작 원리
3. CNI 살펴보기(flannel)
4. 실습

1. 컨테이너 네트워크의 동작원리

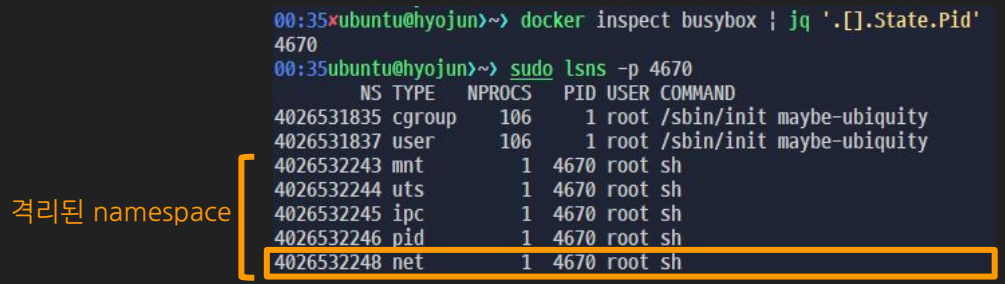
컨테이너란?

- 컨테이너는 “격리된 환경”에서 실행되는 “프로세스”
- 격리된 환경을 구현하는 주요 원리
  - chroot, cgroups
  - Linux namespaces
    - Mount, Process ID, Network, IPC, UTS, User

```
00:35 ubuntu@hyojun ~> docker inspect busybox | jq '.[].State.Pid'
4670
00:35 ubuntu@hyojun ~> sudo lsns -p 4670
        NS TYPE   NPROCS   PID USER COMMAND
4026531835 cgroup    106     1 root /sbin/init maybe-ubiquity
4026531837 user      106     1 root /sbin/init maybe-ubiquity
4026532243 mnt         1  4670 root sh
4026532244 uts         1  4670 root sh
4026532245 ipc         1  4670 root sh
4026532246 pid         1  4670 root sh
4026532248 net         1  4670 root sh
```

컨테이너란?

- 컨테이너는 “격리된 환경”에서 실행되는 “프로세스”
- 격리된 환경을 구현하는 주요 원리
  - chroot, cgroups
  - Linux namespaces
    - Mount, Process ID, Network, IPC, UTS, User

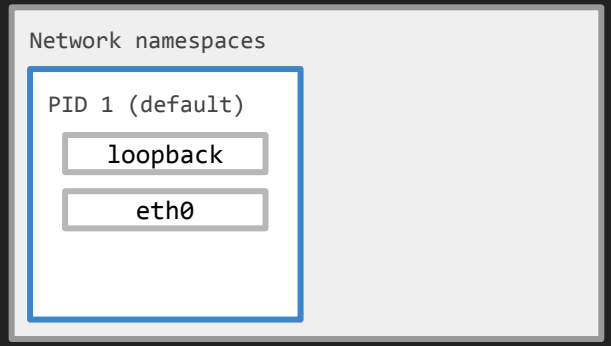


Network namespace

네트워크 인터페이스, 라우팅, 방화벽 규칙들을 격리

```
# “test-ns” 이름의 Network namespace 생성
$ ip netns add test-ns
$ ip netns list
test-ns

# veth pair 생성 (veth1, veth2)
# “veth1”은 test-ns에 생성, “veth2”는 PID 1의 default network namespace에 생성
$ ip link add veth1 netns test-ns type veth peer name veth2 netns 1
```

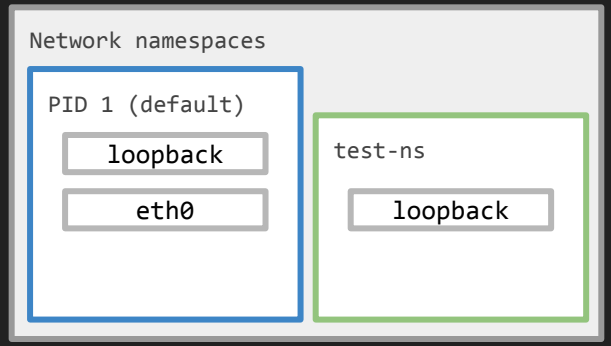


Network namespace

네트워크 인터페이스, 라우팅, 방화벽 규칙들을 격리

```
# “test-ns” 이름의 Network namespace 생성
$ ip netns add test-ns
$ ip netns list
test-ns

# veth pair 생성 (veth1, veth2)
# “veth1”은 test-ns에 생성, “veth2”는 PID 1의 default network namespace에 생성
$ ip link add veth1 netns test-ns type veth peer name veth2 netns 1
```

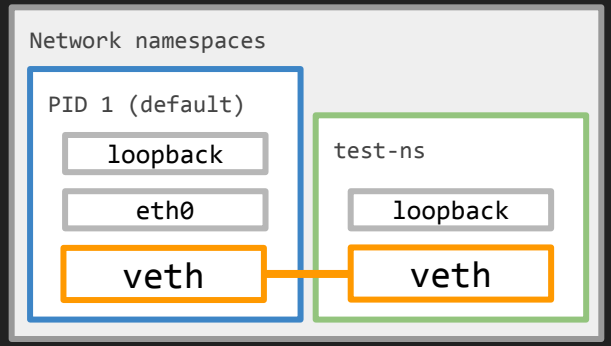


Network namespace

네트워크 인터페이스, 라우팅, 방화벽 규칙들을 격리

```
# “test-ns” 이름의 Network namespace 생성
$ ip netns add test-ns
$ ip netns list
test-ns

# veth pair 생성 (veth1, veth2)
# “veth1”은 test-ns에 생성, “veth2”는 PID 1의 default network namespace에 생성
$ ip link add veth1 netns test-ns type veth peer name veth2 netns 1
```



veth?

- 가상 이더넷 인터페이스(Virtual ethernet interface)
- 항상 쌍(pair)로 생성되어 연결된 상태를 유지
- Network namespace 간의 터널 역할

Network namespace

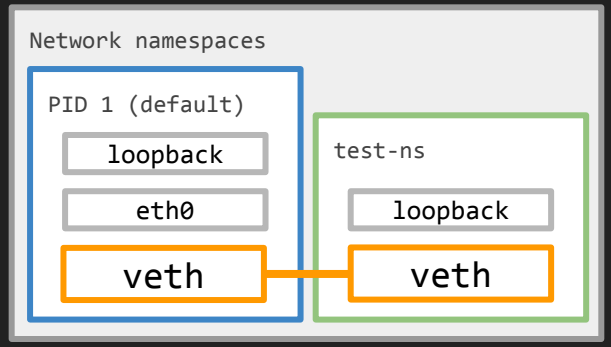
네트워크 인터페이스, 라우팅, 방화벽 규칙들을 격리

```
# 호스트의 default network namespace에서 “ip link list”를 실행 (네트워크 인터페이스 출력)
$ ip link list

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
(... 생략 ...)
7: veth2@if8: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether d2:a1:90:78:3c:4b brd ff:ff:ff:ff:ff:ff link-netnsid 0

# 새로 생성된 “test-ns”에서 “ip link list” 명령어 실행
# test-ns에 할당한 veth1와 loop back 인터페이스만 존재
$ ip netns exec test-ns ip link list

1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
8: veth1@if7: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 2a:aa:60:ee:27:d4 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```



Network namespace

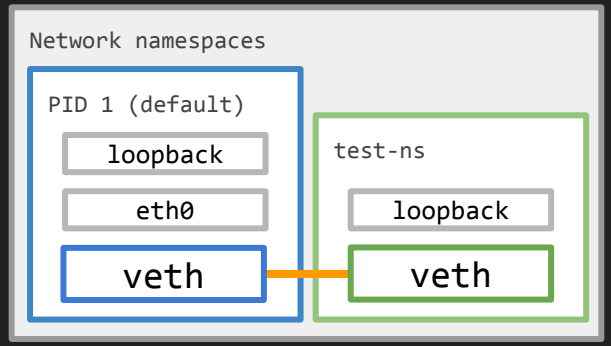
네트워크 인터페이스, 라우팅, 방화벽 규칙들을 격리

```
# 호스트의 default network namespace에서 “ip link list”를 실행 (네트워크 인터페이스 출력)
$ ip link list

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
(... 생략 ...)
7: veth2@if8: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether d2:a1:90:78:3c:4b brd ff:ff:ff:ff:ff:ff link-netnsid 0

# 새로 생성된 “test-ns”에서 “ip link list” 명령어 실행
# test-ns에 할당한 veth1와 loop back 인터페이스만 존재
$ ip netns exec test-ns ip link list

1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
8: veth1@if7: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 2a:aa:60:ee:27:d4 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

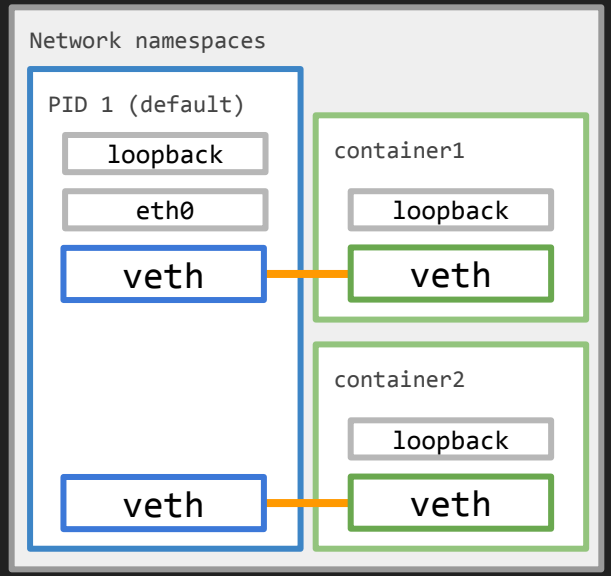


Network namespace

네트워크 인터페이스, 라우팅, 방화벽 규칙들을 격리

컨테이너들은 각각의 veth pair를 통해 호스트와 연결

컨테이너 간 통신은 어떻게?



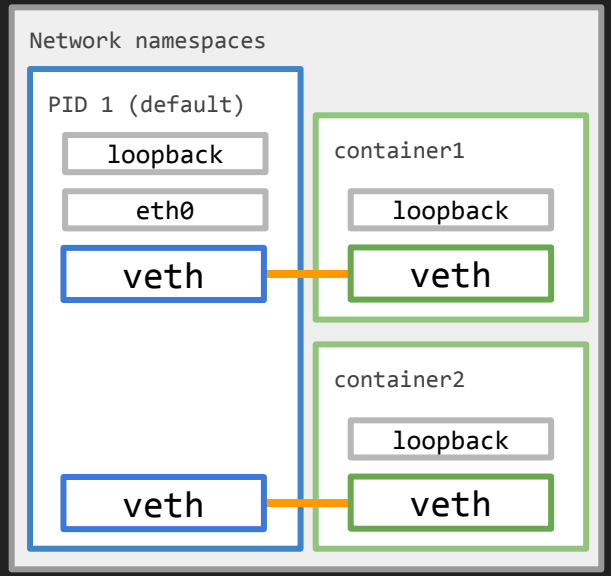
Network namespace

네트워크 인터페이스, 라우팅, 방화벽 규칙들을 격리

컨테이너들은 각각의 veth pair를 통해 호스트와 연결

컨테이너 간 통신은 어떻게?

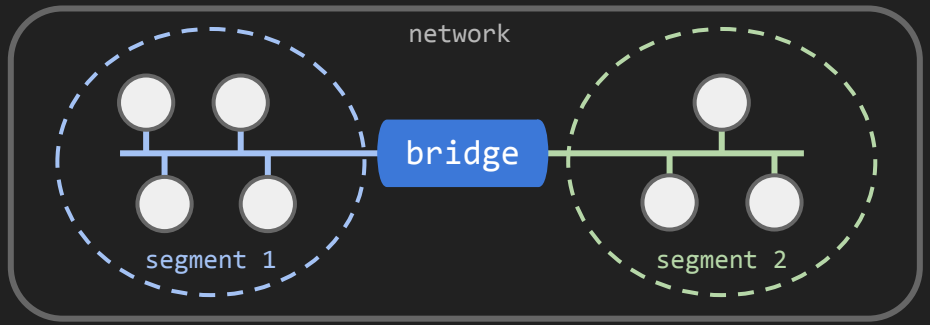
→ **Bridge!**



Bridge란?

- 데이터 링크 계층의 장치
- 네트워크를 세그먼트(Segment) 단위로 분할
  - 네트워크 세그먼트(Network Segment)

    → Bridge, Router 등에 의해 더 작은 단위로 분할된 네트워크
- 네트워크 세그먼트 간의 트래픽을 전달
- 세그먼트 간의 프레임(Frame)을 필터링하여 전송 가능
  - 프레임 → 데이터 링크 계층의 데이터 전송 단위



Docker의 Bridge networks

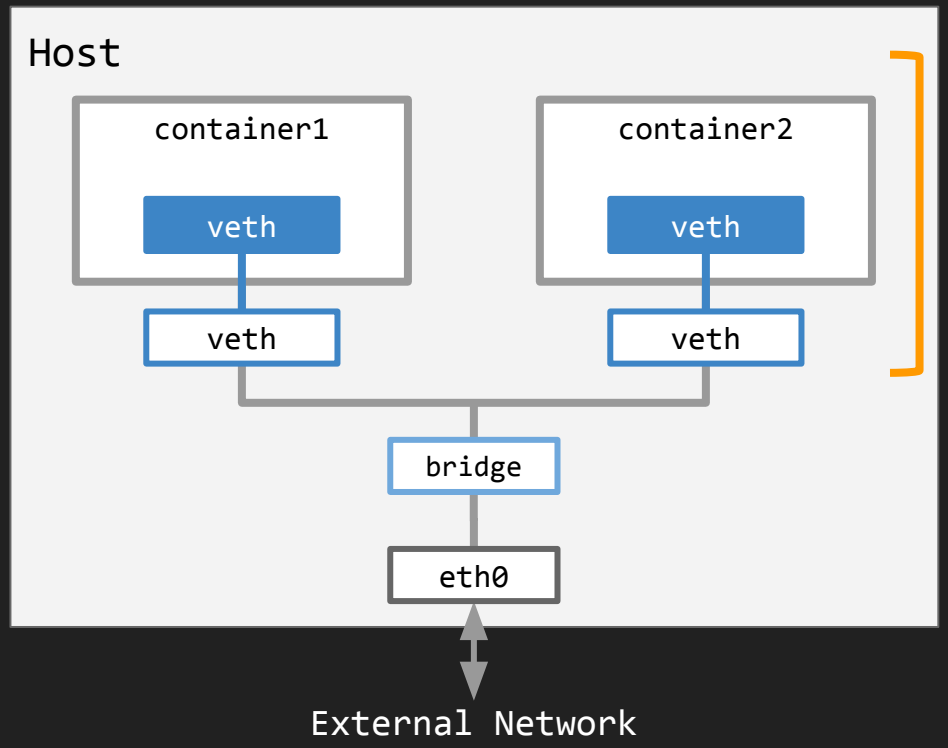
- 컨테이너 네트워크 인터페이스를 Bridge로 연결
  - 연결된 컨테이너들은 Bridge를 통해 서로 통신 가능
  - 같은 Bridge에 연결되지 않은 컨테이너들과의 네트워크 격리
  - Bridge로 연결된 컨테이너들은 하나의 네트워크 세그먼트
- Docker 시작 시 기본적으로 Default bridge network 생성

```
02:50ubuntu@hyojun>~> ifconfig
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        inet6 fe80::42:bbff:fe1b:5545  prefixlen 64  scopeid 0x20<link>
        ether 02:42:bb:1b:55:45  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 5  bytes 446 (446.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.15  netmask 255.255.255.0  broadcast 10.0.2.255
```

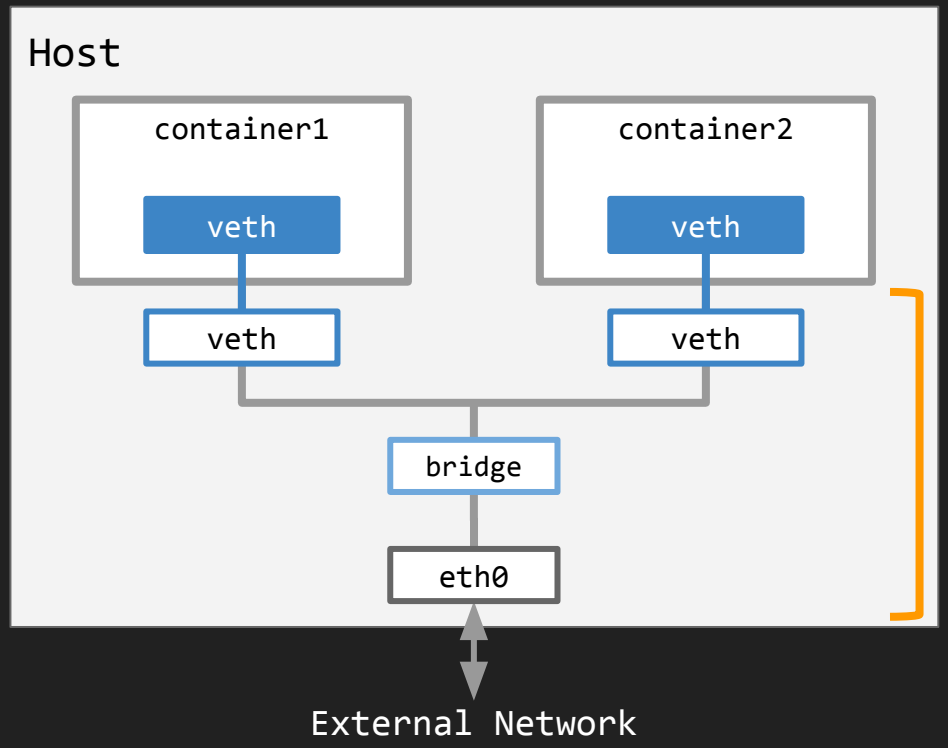
Docker의 default bridge

Docker의 Bridge networks



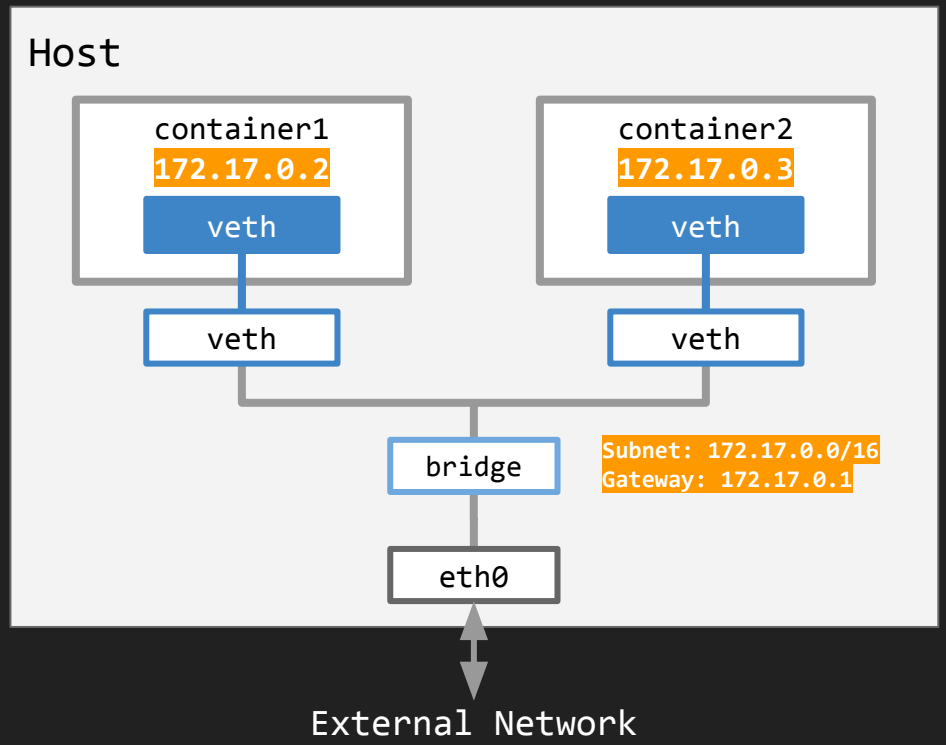
컨테이너는 Host 와 Network namespace 격리
Host - Container 간 veth pair를 생성하여 연결

Docker의 Bridge networks



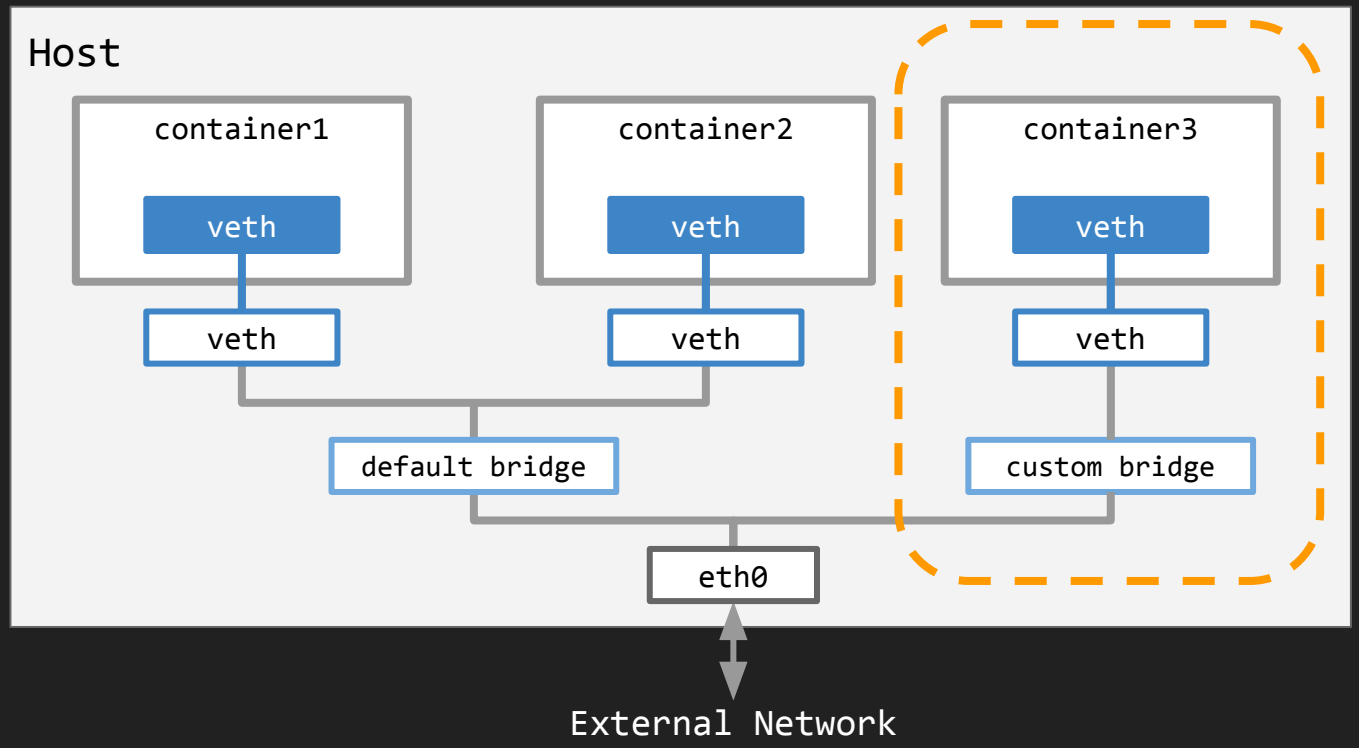
각 컨테이너의 veth들은 Bridge로 연결
→ 같은 Bridge에 연결된 컨테이너 간 통신 가능
→ 컨테이너 네트워크 외부로 통신 시 Bridge를 경유

Docker의 Bridge networks




```
03:38✖ubuntu@hyojun>~> docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "a5372691b4dfd31981e97b6cd8745ce820f10eaefe60bec82
        "Created": "2021-03-28T13:10:52.71880201+09:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "1934b2424446234fce8810a863ce523891f262c213e8b39a0ed
                "Name": "busybox2",
                "EndpointID": "6f7206f2f1af03baf728d03286bfd69b7
                "MacAddress": "02:42:ac:11:00:03",
                "IPv4Address": "172.17.0.3/16",
                "IPv6Address": ""
            },
            "85e67b563ffa20b55479f4910a554ff9f26f2bdc6199965de8e
                "Name": "busybox",
                "EndpointID": "b1661c866903125df96b705937bf22b0f
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            }
        },
```

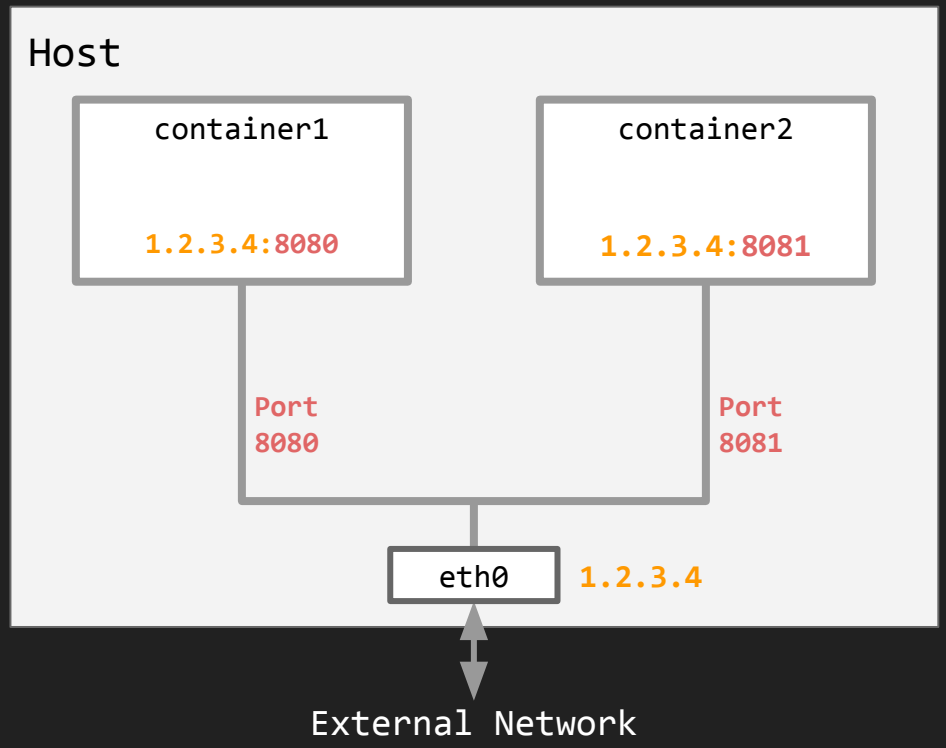
Docker의 Bridge networks



Custom bridge를 통해
별도의 컨테이너 네트워크 생성 가능

$ docker network create <name>
$ docker run --network <name> ...

Docker의 Host networking

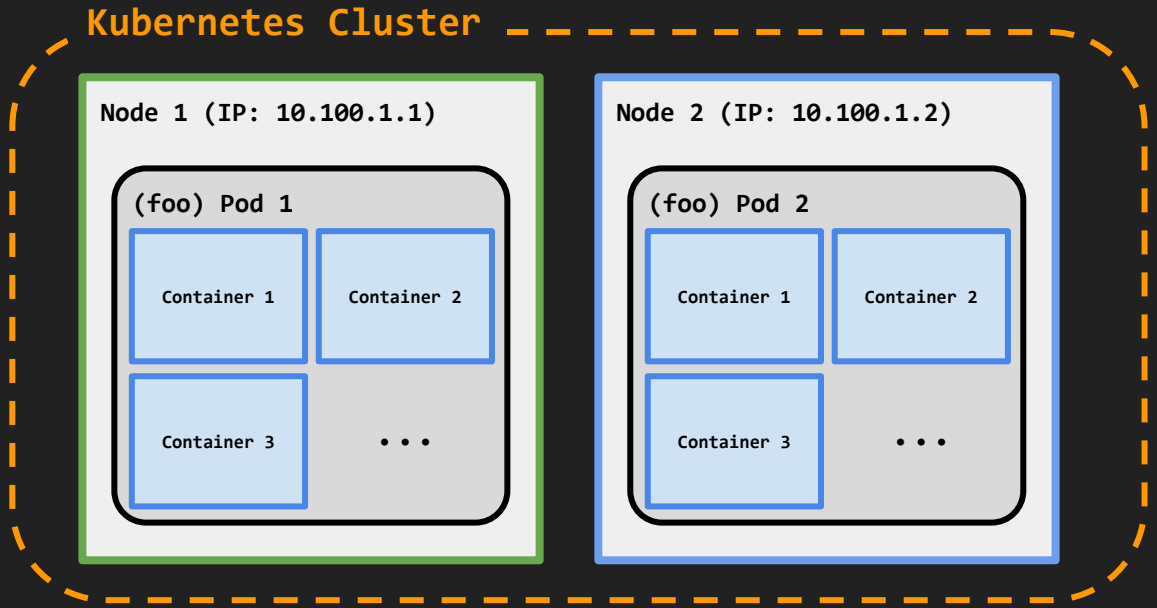


Network namespace를 격리하지 않음
→ 컨테이너에 IP 주소가 할당되지 않음
→ 호스트의 네트워크 환경을 그대로 사용

2. Pod 네트워크의 동작 원리

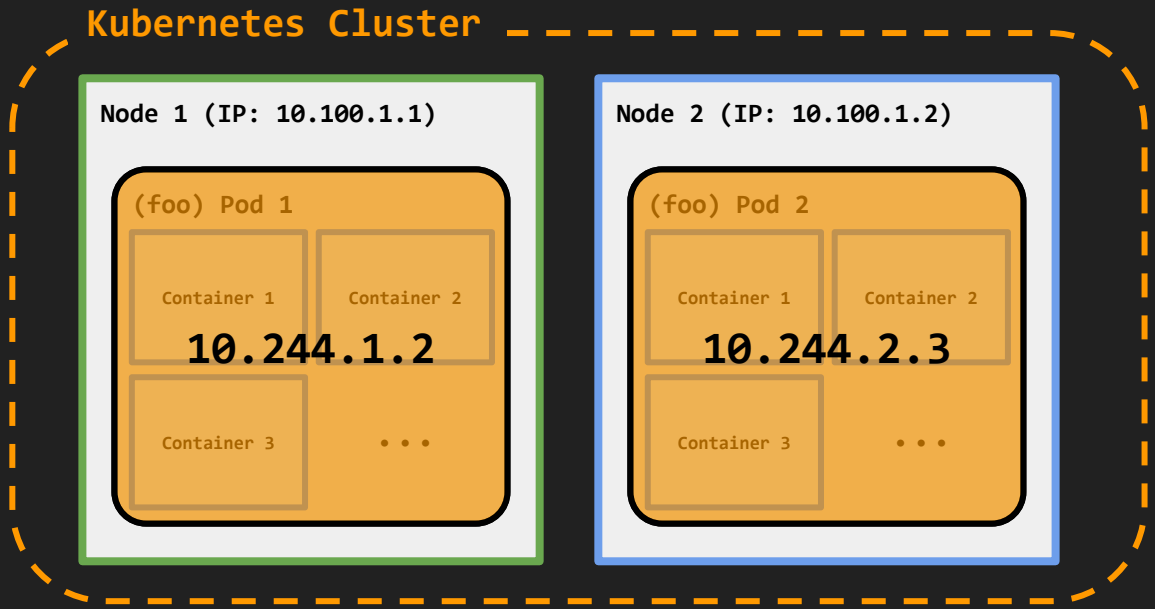
Kubernetes Pod

- Kubernetes에서 배포할 수 있는 최소 객체 단위
- 1개 이상의 컨테이너로 이루어진 그룹



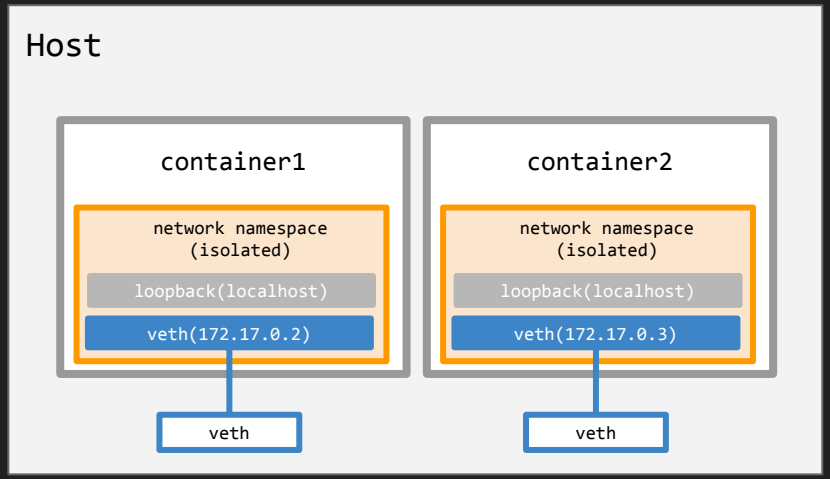
모든 Pod은 고유 IP를 가진다

Pod에 할당된 IP를 통해 클러스터 내의 Pod 간의 통신이 가능

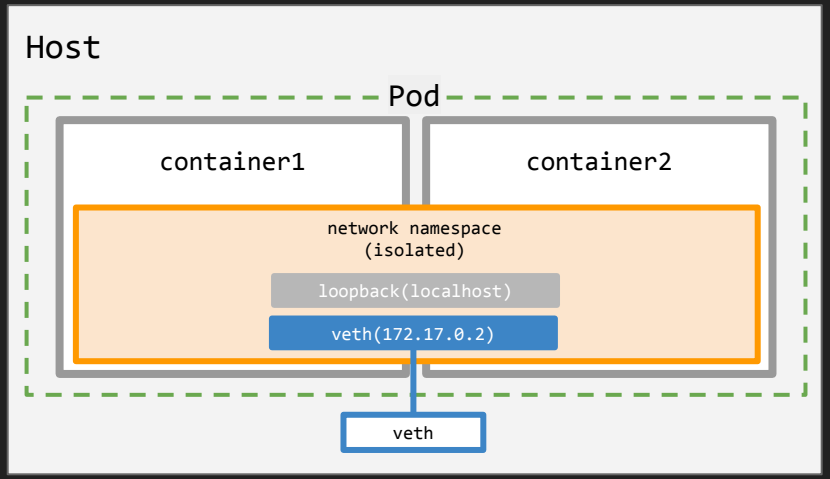


Pod 컨테이너“들”이 같은 IP를 가질 수 있는 이유

- 같은 Pod의 컨테이너들은 Network namespace를 공유
  - loopback 인터페이스를 통해 localhost + port로 통신 가능



기존 컨테이너의 Network namespace 격리

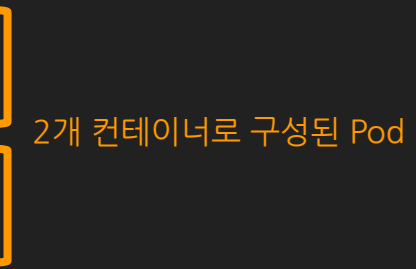


Pod 컨테이너들의 Network namespace 공유

Pod의 Network namespace 공유 확인해보기
(1) 2개 컨테이너가 실행되는 Pod 정의

two-containers-pod.yml

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: two-containers-pod
spec:
  template:
    # This is the pod template
    spec:
      containers:
      - name: hello
        image: busybox
        command: ['sh', '-c', 'echo "first container" && sleep 3600']
      - name: hello2
        image: busybox
        command: ['sh', '-c', 'echo "second container" && sleep 3600']
      restartPolicy: OnFailure
    # The pod template ends here
```



Pod의 Network namespace 공유 확인해보기
(2) Pod 생성 및 Pod이 실행된 호스트 확인

(Kubernetes v1.20.2)

```
vagrant@control-plane:~$ kubectl apply -f two-containers-pod.yml
job.batch/two-containers-pod created
vagrant@control-plane:~$ kubectl get pods -o wide
NAME                       READY   STATUS    RESTARTS   AGE   IP           NODE
two-containers-pod-9pvvt   2/2     Running   0          20s   10.244.2.4   worker-node2
```

Pod의 IP

Pod이 실행 중인 호스트(노드)

Pod의 Network namespace 공유 확인해보기

(3) 컨테이너 프로세스들의 Linux namespace 확인

```
vagrant@worker-node2:~$ ps -ef | grep sleep
root        9318    9295  0 20:01 ?        00:00:00 sleep 3600
root        9382    9356  0 20:01 ?        00:00:00 sleep 3600
vagrant    10647   10383  0 20:04 pts/0    00:00:00 grep --color=auto sleep
vagrant@worker-node2:~$ sudo lsns -p 9318
        NS TYPE   NPROCS   PID USER COMMAND
4026531835 cgroup    121     1 root /sbin/init
4026531837 user      121     1 root /sbin/init
4026532214 ipc         3  9182 root /pause
4026532217 net         3  9182 root /pause
4026532304 mnt         1  9318 root sleep 3600
4026532305 uts         1  9318 root sleep 3600
4026532306 pid         1  9318 root sleep 3600
vagrant@worker-node2:~$ sudo lsns -p 9382
        NS TYPE   NPROCS   PID USER COMMAND
4026531835 cgroup    121     1 root /sbin/init
4026531837 user      121     1 root /sbin/init
4026532214 ipc         3  9182 root /pause
4026532217 net         3  9182 root /pause
4026532307 mnt         1  9382 root sleep 3600
4026532308 uts         1  9382 root sleep 3600
4026532309 pid         1  9382 root sleep 3600
```

Pod이 실행 중인 호스트에 접속하여
sleep 명령어 실행 중인 Pod 컨테이너 PID 확인

Pod의 Network namespace 공유 확인해보기

(3) 컨테이너 프로세스들의 Linux namespace 확인

```
vagrant@worker-node2:~$ ps -ef | grep sleep
root        9318    9295  0 20:01 ?        00:00:00 sleep 3600
root        9382    9356  0 20:01 ?        00:00:00 sleep 3600
vagrant    10647   10383  0 20:04 pts/0    00:00:00 grep --color=auto sleep
vagrant@worker-node2:~$ sudo lsns -p 9318
        NS TYPE   NPROCS   PID USER COMMAND
4026531835 cgroup    121     1 root /sbin/init
4026531837 user      121     1 root /sbin/init
4026532214 ipc         3  9182 root /pause
4026532217 net         3  9182 root /pause
4026532304 mnt         1  9318 root sleep 3600
4026532305 uts         1  9318 root sleep 3600
4026532306 pid         1  9318 root sleep 3600
vagrant@worker-node2:~$ sudo lsns -p 9382
        NS TYPE   NPROCS   PID USER COMMAND
4026531835 cgroup    121     1 root /sbin/init
4026531837 user      121     1 root /sbin/init
4026532214 ipc         3  9182 root /pause
4026532217 net         3  9182 root /pause
4026532307 mnt         1  9382 root sleep 3600
4026532308 uts         1  9382 root sleep 3600
4026532309 pid         1  9382 root sleep 3600
```

$ lsns -p <PID>

특정 프로세스의 namespace 조회

Pod의 Network namespace 공유 확인해보기

(3) 컨테이너 프로세스들의 Linux namespace 확인

```
vagrant@worker-node2:~$ ps -ef | grep sleep
root        9318  9295  0 20:01 ?        00:00:00 sleep 3600
root        9382  9356  0 20:01 ?        00:00:00 sleep 3600
vagrant    10647 10383  0 20:04 pts/0    00:00:00 grep --color=auto sleep
vagrant@worker-node2:~$ sudo lsns -p 9318
        NS TYPE   NPROCS   PID USER COMMAND
4026531835 cgroup    121     1 root /sbin/init
4026531837 user      121     1 root /sbin/init
4026532214 ipc         3  9182 root /pause
4026532217 net         3  9182 root /pause
4026532304 mnt         1  9318 root sleep 3600
4026532305 uts         1  9318 root sleep 3600
4026532306 pid         1  9318 root sleep 3600
vagrant@worker-node2:~$ sudo lsns -p 9382
        NS TYPE   NPROCS   PID USER COMMAND
4026531835 cgroup    121     1 root /sbin/init
4026531837 user      121     1 root /sbin/init
4026532214 ipc         3  9182 root /pause
4026532217 net         3  9182 root /pause
4026532307 mnt         1  9382 root sleep 3600
4026532308 uts         1  9382 root sleep 3600
4026532309 pid         1  9382 root sleep 3600
```

Network namespace는 pod의 컨테이너 간 공유

→ 컨테이너 간 동일한 IP 주소, 포트를 공유(충돌 주의)

Pause?

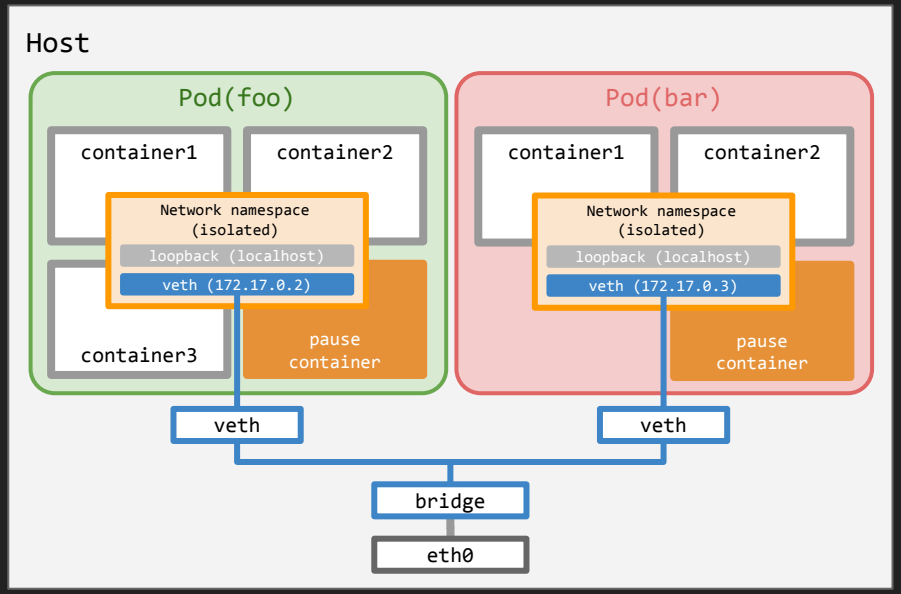
Pause Container?

```
vagrant@worker-node2:~$ docker ps | grep two-containers
ae2b5c6f7882        busybox                  "sh -c 'echo \"second…"
e8670003df44        busybox                  "sh -c 'echo \"first …"
19802e369987        k8s.gcr.io/pause:3.2     "/pause"
```

Pause 컨테이너는 격리된 IPC, Network namespace를 생성하고 유지
→ 나머지 컨테이너들은 해당 namespace를 공유하여 사용

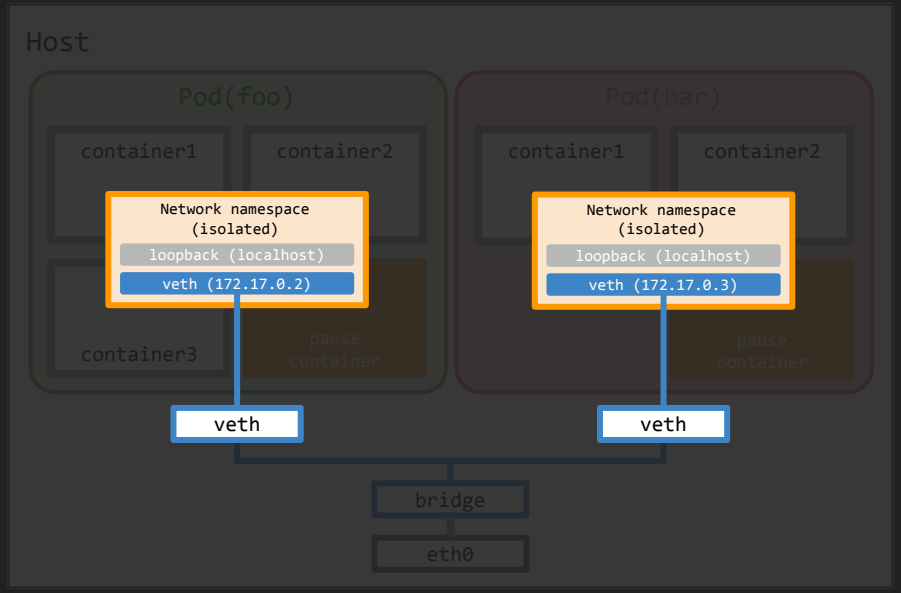
유저가 실행한 특정 컨테이너가 비정상 종료되어,
컨테이너 전체에서 공유되는 namespace에 문제가 발생하는 것을 방지

Container Bridge Network + Pod



다시 살펴봅시다.

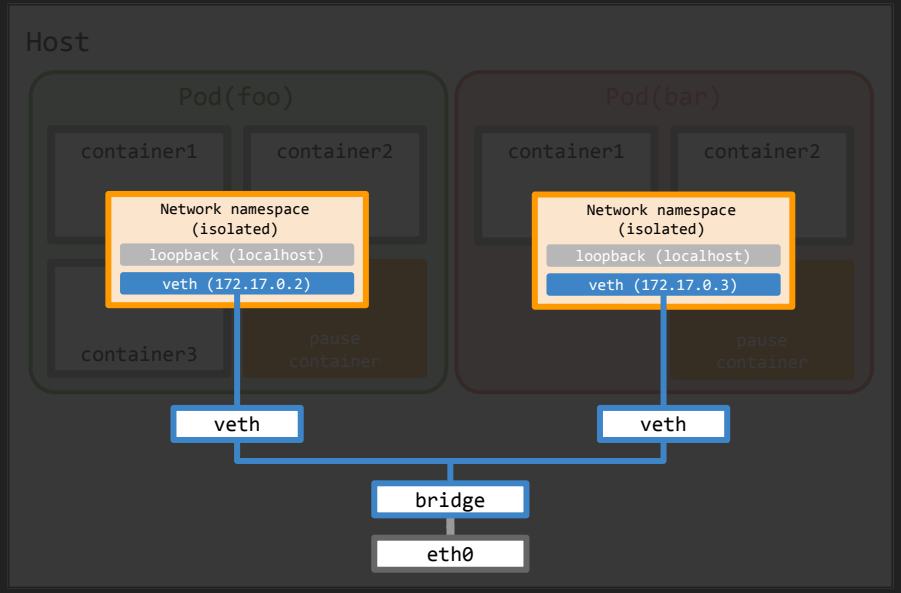
Container Bridge Network + Pod



컨테이너의 Network namespace 격리

- 컨테이너는 Network namespace를 통해 네트워크 인터페이스, 라우팅, 방화벽 규칙들을 격리
- veth(Virtual ethernet interface) pair를 통해 호스트와 연결

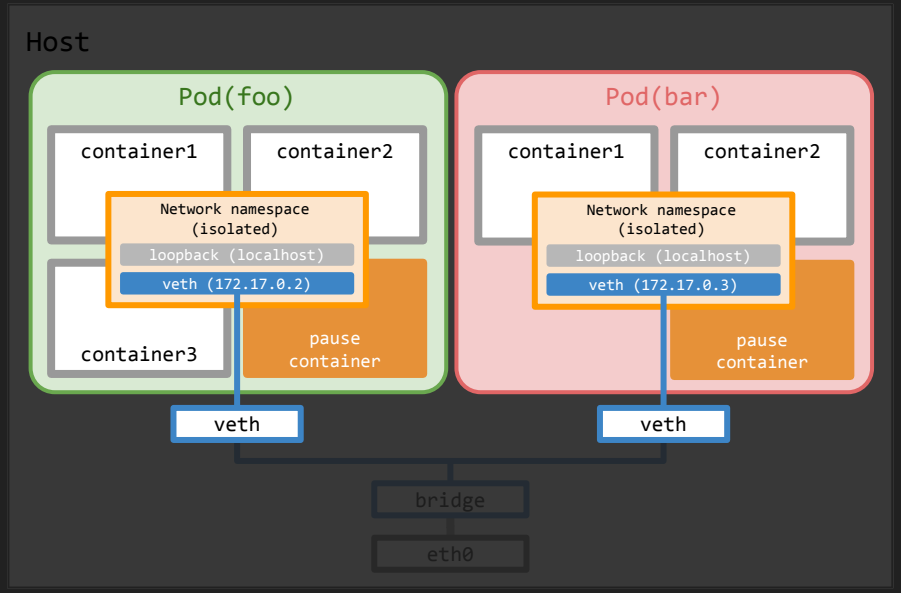
Container Bridge Network + Pod



Bridge networks

- 컨테이너 네트워크 인터페이스들을 Bridge로 연결
- 연결된 컨테이너들은 Bridge를 통해 서로 통신 가능
- 컨테이너 네트워크 외부로 통신 시 Bridge를 경유

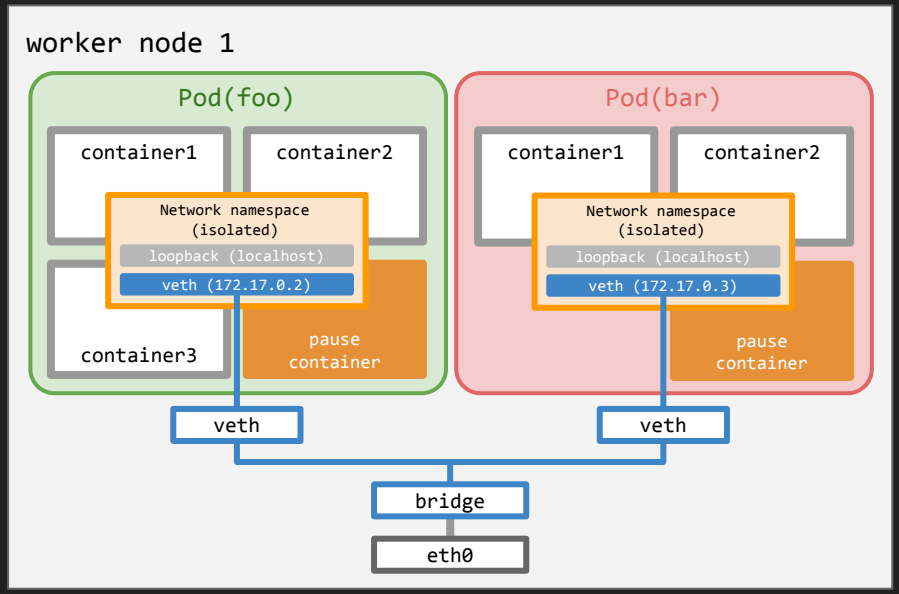
Container Bridge Network + Pod

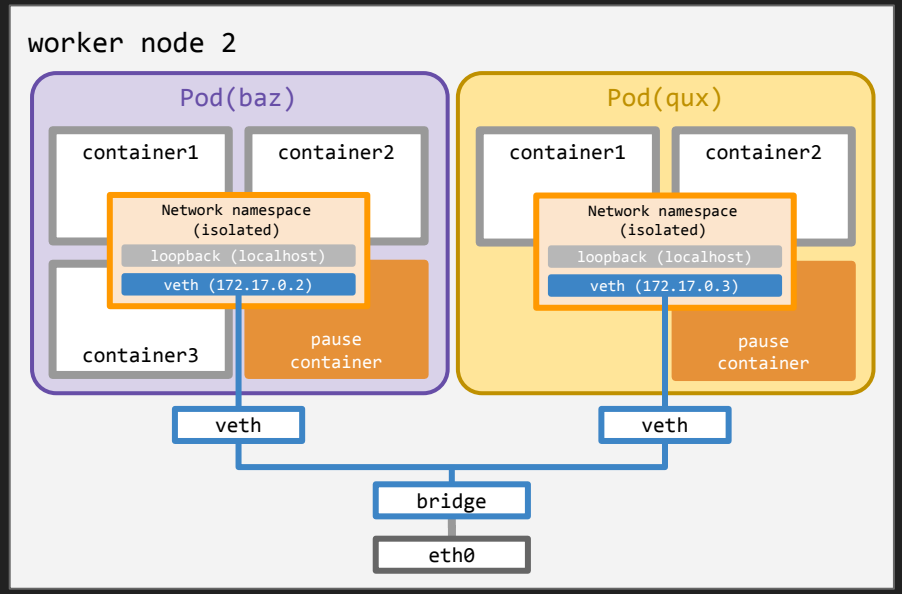


Pod 컨테이너의 Network namespace 공유

- Pod마다 기본적으로 생성되는 Pause 컨테이너는 격리된 Network namespace를 생성 및 유지
- Network namespace는 같은 Pod의 컨테이너들과 공유
  - 같은 Pod의 컨테이너들은 동일한 IP를 가짐
  - 같은 Pod의 컨테이너들은 localhost + port를 통해 접근 가능

여러 개의 노드로 구성된 Kubernetes Cluster에서는?

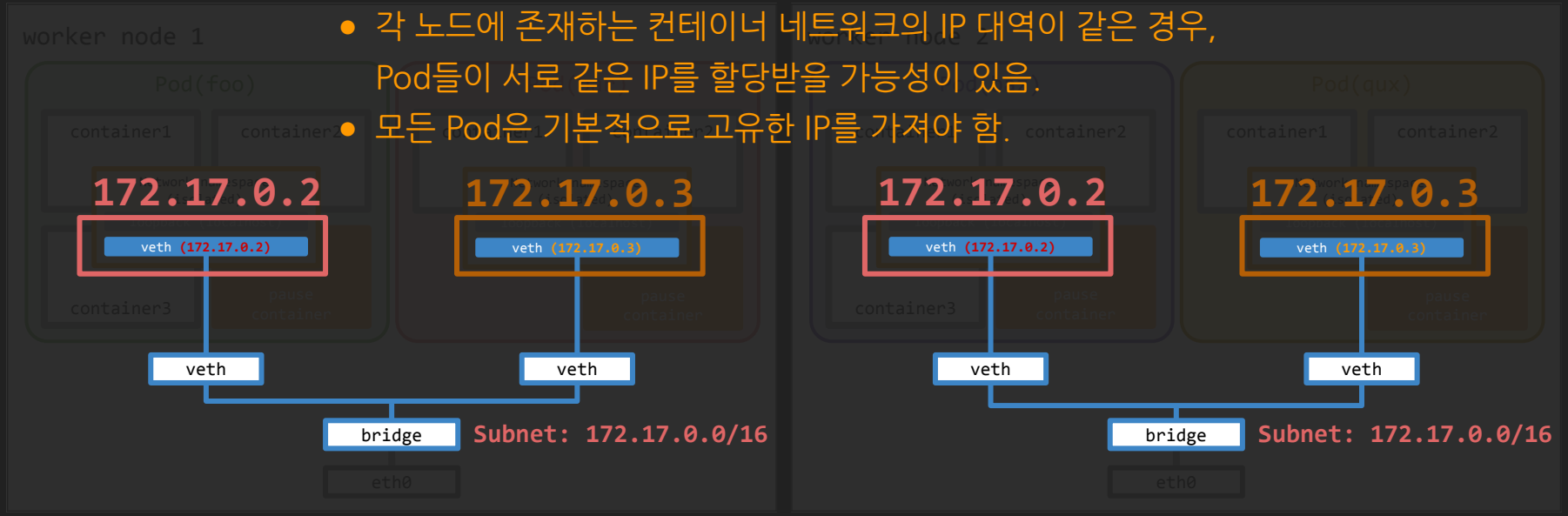




여러 개의 노드로 구성된 Kubernetes Cluster에서는?

문제점 1

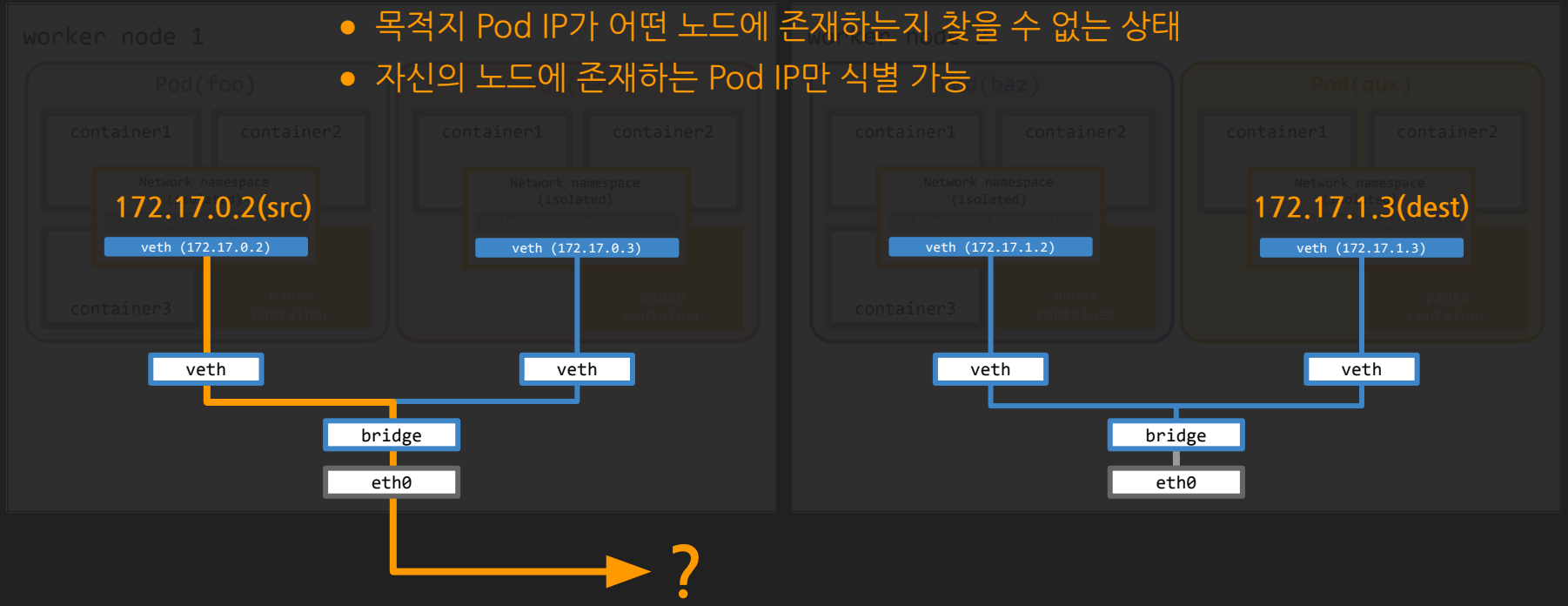
- 각 노드에 존재하는 컨테이너 네트워크의 IP 대역이 같은 경우,
  Pod들이 서로 같은 IP를 할당받을 가능성이 있음.
- 모든 Pod은 기본적으로 고유한 IP를 가져야 함.

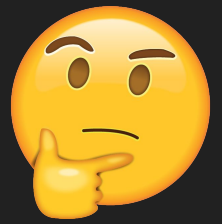


여러 개의 노드로 구성된 Kubernetes Cluster에서는?

문제점 2

- 목적지 Pod IP가 어떤 노드에 존재하는지 찾을 수 없는 상태
- 자신의 노드에 존재하는 Pod IP만 식별 가능





Kubernetes Pod Network를 어떻게 구성할 것인가?

Kubernetes Pod Network를 어떻게 구성할 것인가?

➡ “기본적인 요구 사항과 규격을 정해줄 테니, 알아서 해.”

Kubernetes Pod Network 기본 요구 사항

- Pod들은 각자 고유한 IP를 가진다.
- Cluster 내의 모든 Pod 들은 NAT없이 서로 통신 가능하다.
- 노드의 Agent(e.g. system daemons, kubelet)들은
  해당 노드의 모든 Pod과 통신 가능하다.

CNI (Container Network Interface)

- Pod가 생성, 삭제될 때 호출되는 API의 규격과 인터페이스를 정의
- 여러 가지 형태의 네트워크를 Kubernetes에서 쉽게 연동 가능
- CNI Plugins
  - IPAM (IP Address management) Plugin
    - Pod 생성, 삭제 시 IP 주소를 할당 및 해제
  - Network Plugin
    - Pod 생성, 삭제 시 네트워크 연결을 구현

여러 종류의 CNI들

- Pod Network를 구성하는 여러 가지 방법들
  - Flannel
  - Calico
  - Cilium
  - Weave Net
  - AWS VPC CNI for Kubernetes
  - Azure CNI for Kubernetes
  - … 등

여러 종류의 CNI들

- Pod Network를 구성하는 여러 가지 방법들
  - Flannel Flannel이 어떻게 네트워크를 구성하는지 알아봅시다
  - Calico
  - Cilium
  - Weave Net
  - AWS VPC CNI for Kubernetes
  - Azure CNI for Kubernetes
  - … 등

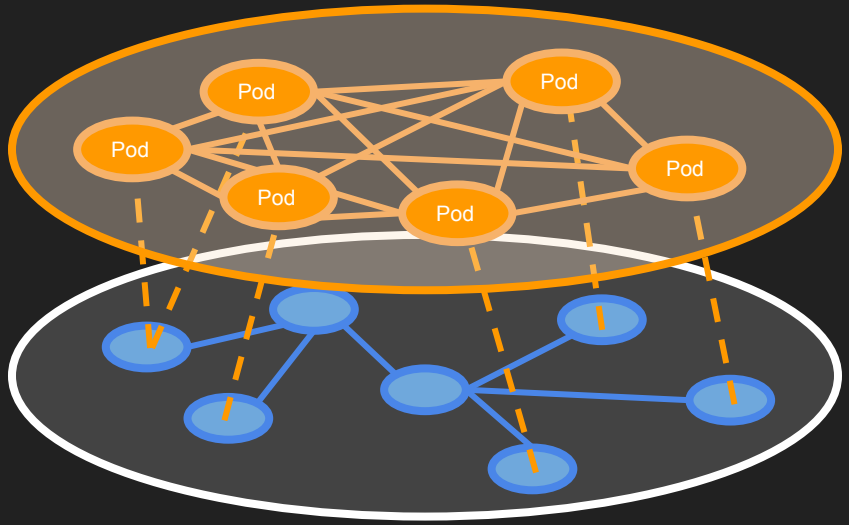
3. CNI 살펴보기(flannel)

Flannel

- Kubernetes에서 Overlay Network를 구성
- 기본적으로 VXLAN 방식을 권장
  - VXLAN 이외의 다른 방식들
    https://github.com/flannel-io/flannel/blob/master/Documentation/backends.md

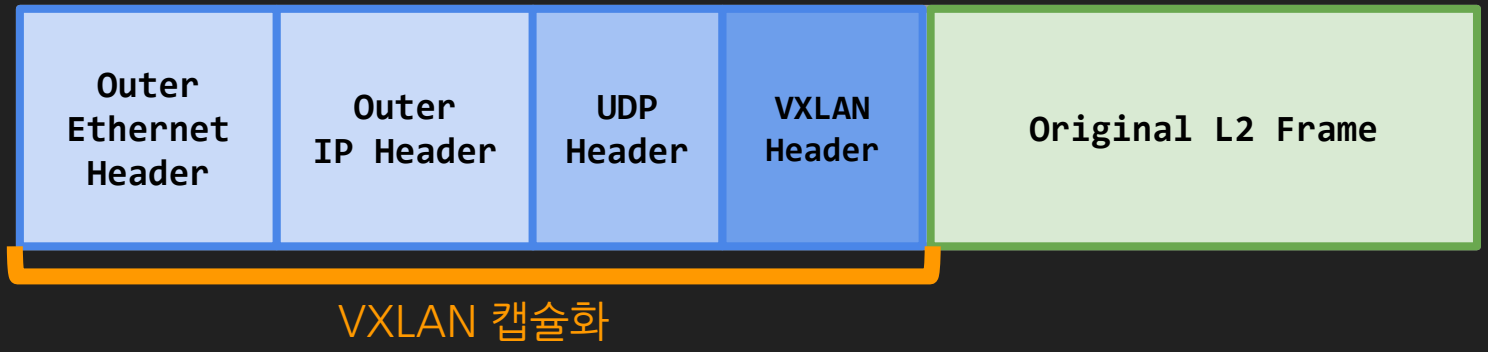
Overlay Network

- 다른 네트워크 위에서 계층화된 네트워크
  - 기존 네트워크의 변경 없이, 그 위에 새로운 네트워크를 구축

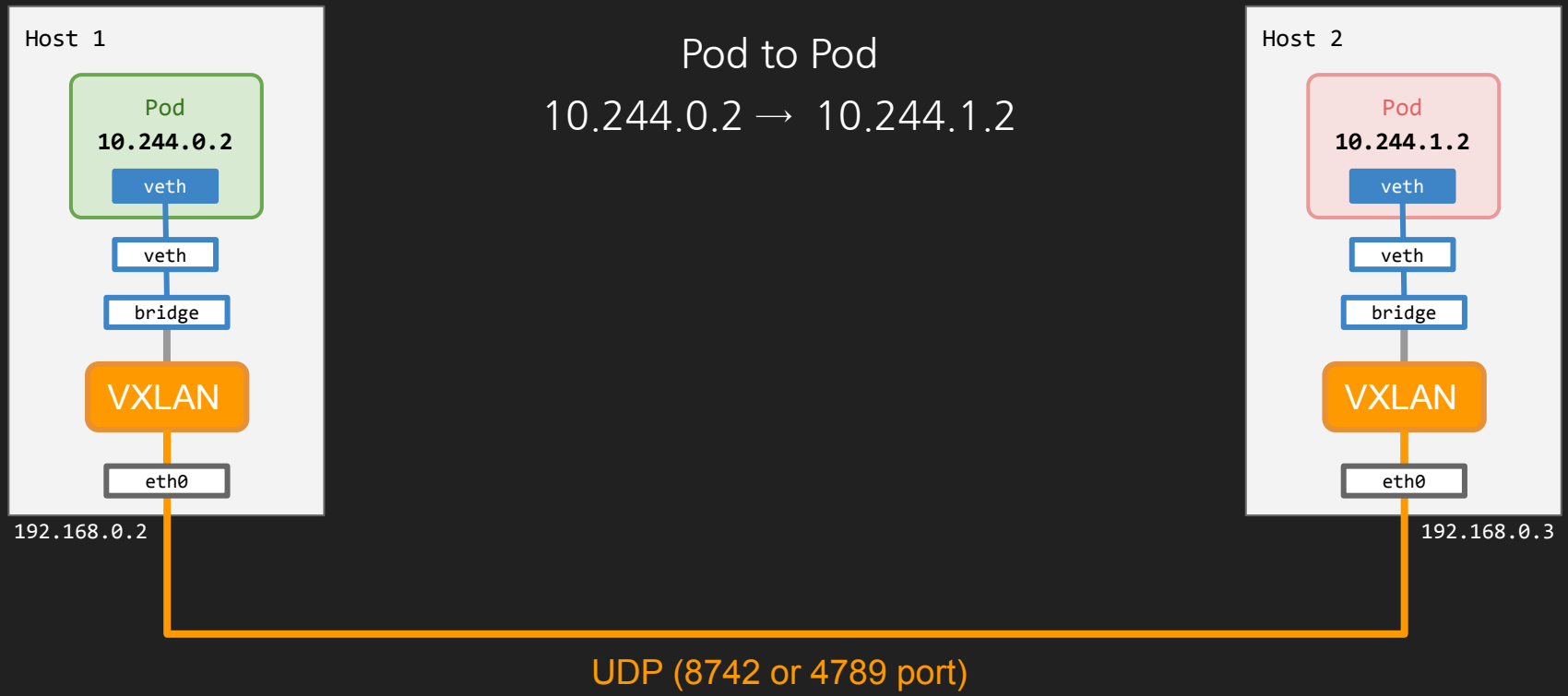


VXLAN

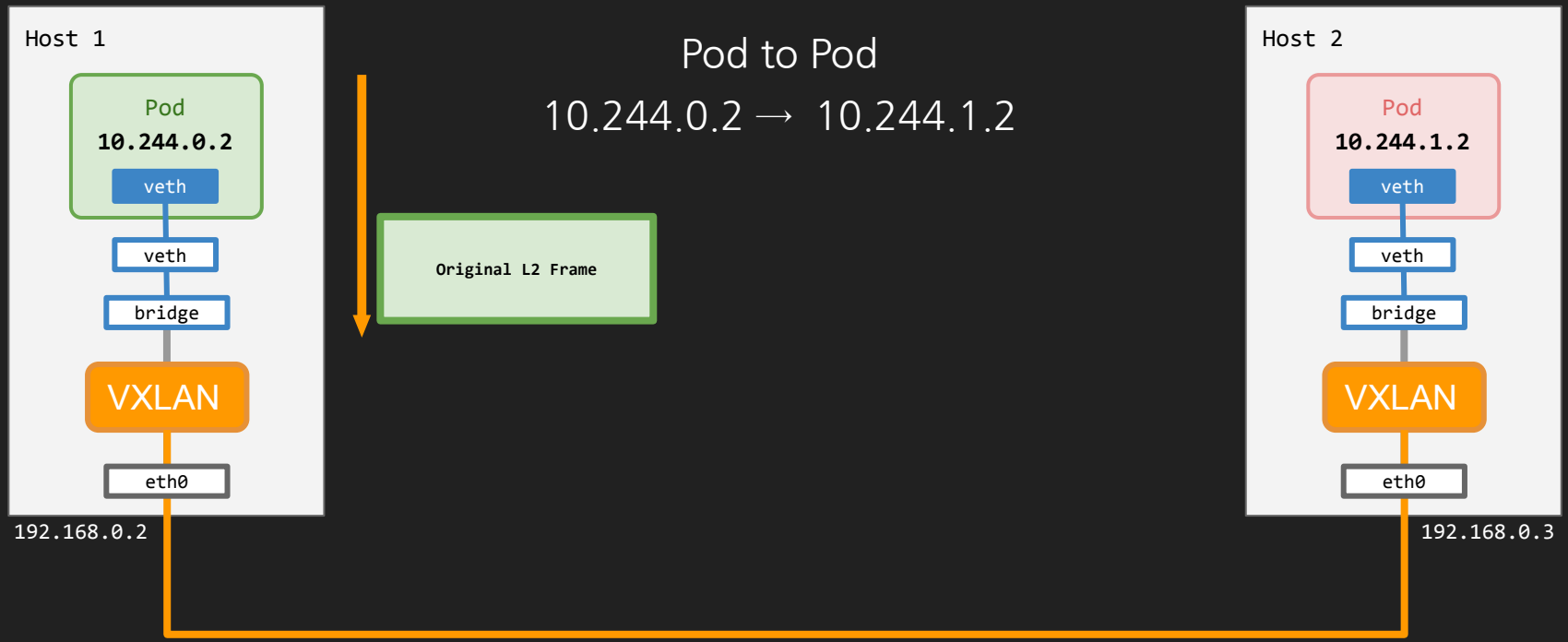
- 표준 Overlay Network 가상화 기술 중 하나
  - 대규모 클라우드 환경 구성에서 가상 네트워크(VLAN)의 확장성 문제를 해결
    - VLAN ID 개수 제한(4096개) = 격리 가능한 네트워크 개수 제한
- UDP 패킷 안에 L2 Frame을 캡슐화하여 터널링 (L2 Over L3)



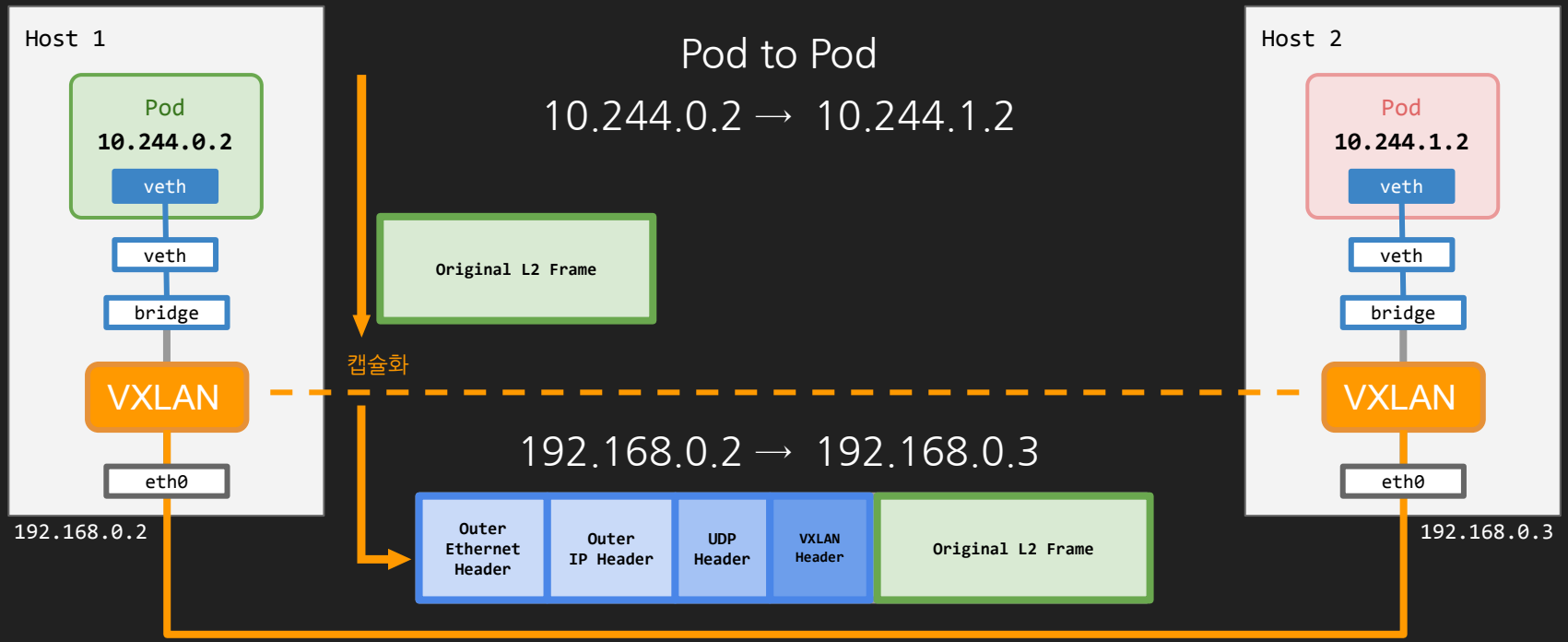
VXLAN - Encapsulation



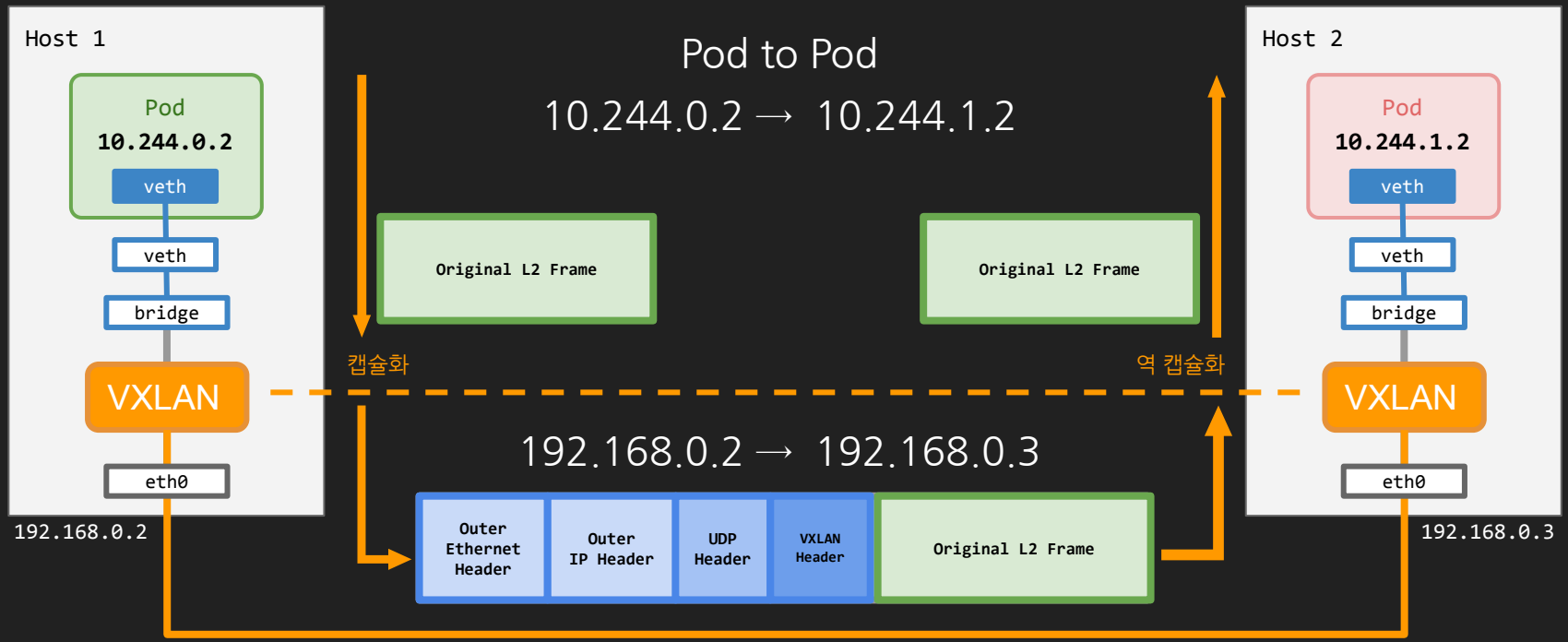
VXLAN - Encapsulation



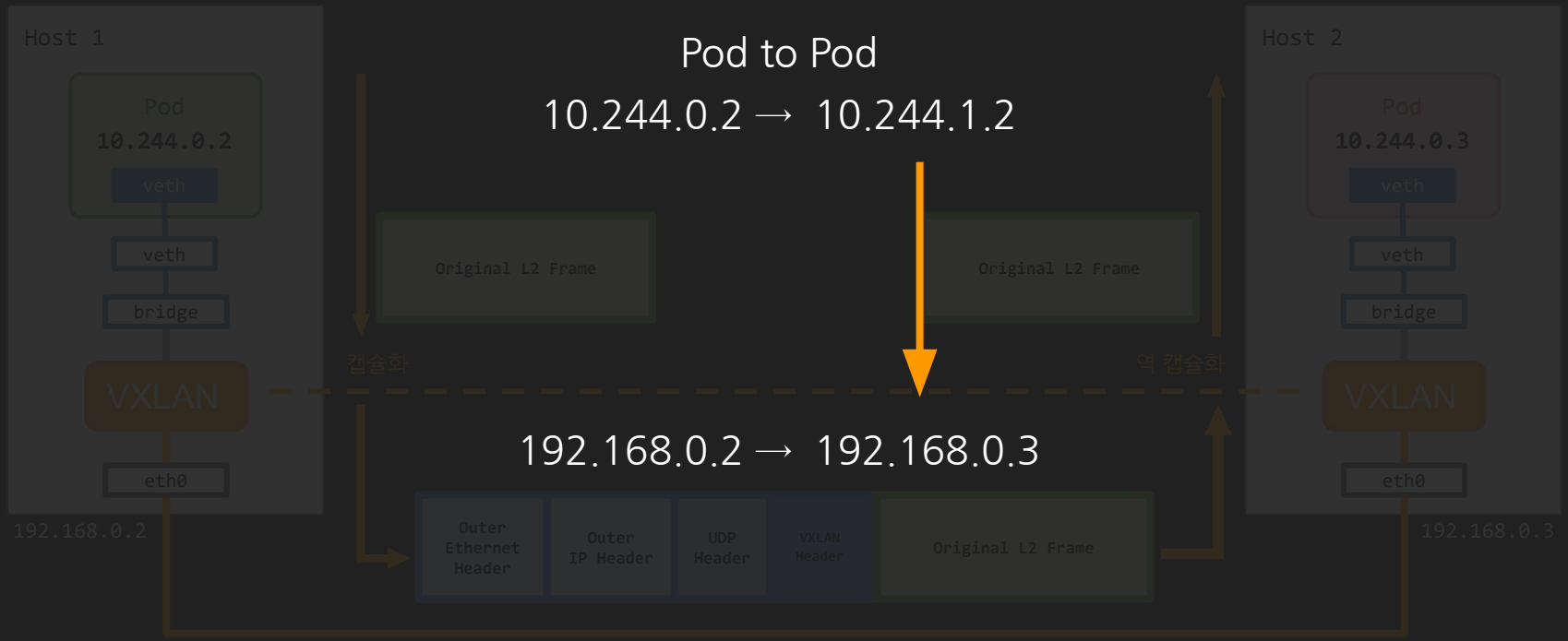
VXLAN - Encapsulation



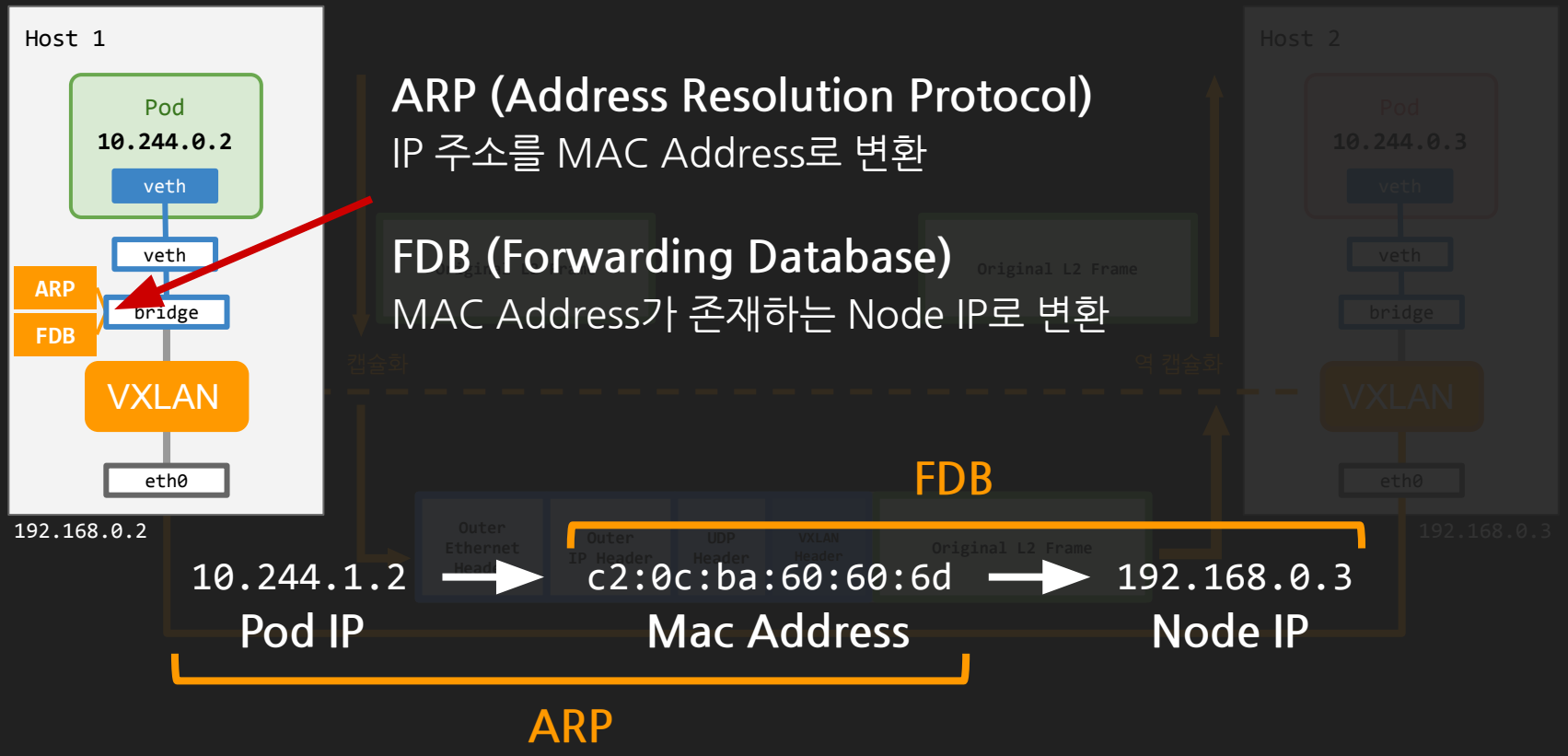
VXLAN - Encapsulation



캡슐화 과정에서 목적지 Pod IP의 Node IP는 어떻게 알 수 있을까?



캡슐화 과정에서 목적지 Pod IP의 Node IP는 어떻게 알 수 있을까?



Flannel CNI의 역할

- 호스트에 VXLAN 생성 및 Overlay network 구성
- 호스트마다 IP Subnet 을 할당
- Subnet 내에서 Pod IP 부여
- Routing table 업데이트
- ARP, FDB 관리
- 등 …

(실습) 컨테이너 네트워크 및 Pod 네트워크 구성을 직접 확인해봅시다 😎

실습 환경 준비

busyboxes.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: busybox1
spec:
 containers:
 - name: busybox
   image: busybox
   command: ['sh', '-c', 'sleep infinity']
---
apiVersion: v1
kind: Pod
metadata:
 name: busybox2
spec:
 containers:
 - name: busybox
   image: busybox
   command: ['sh', '-c', 'sleep infinity']
```

```
# 2개의 Pod 생성
$ kubectl apply -f busyboxes.yaml
```

실습 1. Pod의 Network namespace 격리 확인

Pod가 실행 중인 노드를 확인하여 SSH 접속

```
vagrant@control-plane:~$ kubectl get pods -o wide
NAME          READY   STATUS    RESTARTS   AGE    IP           NODE
busybox1      1/1     Running   0          2m6s   10.244.2.6   worker-node2
busybox2      1/1     Running   0          2m6s   10.244.1.6   worker-node1
```

실습 1. Pod의 Network namespace 격리 확인

```
vagrant@worker-node2:~$ docker ps | grep busybox
12753cb1a39b        busybox                  "sh -c 'sleep infini…"   23 minutes ago      Up 23 minutes
eb20d4578bd5        k8s.gcr.io/pause:3.2     "/pause"                 23 minutes ago      Up 23 minutes
```

busybox Pod의 Container ID 확인

실습 1. Pod의 Network namespace 격리 확인

```
vagrant@worker-node2:~$ docker ps | grep busybox
12753cb1a39b        busybox                 "sh -c 'sleep infini…"   23 minutes ago      Up 23 minutes
eb20d4578bd5        k8s.gcr.io/pause:3.2    "/pause"                 23 minutes ago      Up 23 minutes
vagrant@worker-node2:~$ docker inspect 12753cb1a39b | grep -i networkmode
            "NetworkMode": "container:eb20d4578bd5c5a9638c619976667183516b628bac08c6861a1aaa12c5d74213",
```

busybox 컨테이너의 Network mode 확인

실습 1. Pod의 Network namespace 격리 확인

```
vagrant@worker-node2:~$ docker ps | grep busybox
12753cb1a39b        busybox                 "sh -c 'sleep infini…"   23 minutes ago      Up 23 minutes
eb20d4578bd5        k8s.gcr.io/pause:3.2    "/pause"                 23 minutes ago      Up 23 minutes
vagrant@worker-node2:~$ docker inspect 12753cb1a39b | grep -i networkmode
            "NetworkMode": "container:eb20d4578bd5c5a9638c619976667183516b628bac08c6861a1aaa12c5d74213",
```

busybox 컨테이너의 Network mode 확인

→ NetworkMode의 값이 busybox Pod의 Pause Container ID와 일치하는 것을 알 수 있음
(busybox 컨테이너는 Pause container와 Network namespace를 공유)

실습 1. Pod의 Network namespace 격리 확인

```
vagrant@worker-node2:~$ docker ps | grep busybox
12753cb1a39b        busybox                 "sh -c 'sleep infini…"   23 minutes ago      Up 23 minutes
eb20d4578bd5        k8s.gcr.io/pause:3.2    "/pause"                 23 minutes ago      Up 23 minutes
vagrant@worker-node2:~$ docker inspect 12753cb1a39b | grep -i networkmode
            "NetworkMode": "container:eb20d4578bd5c5a9638c619976667183516b628bac08c6861a1aaa12c5d74213",
vagrant@worker-node2:~$ docker inspect 12753cb1a39b | grep -i pid
            "Pid": 24334,
            "PidMode": "",
            "PidsLimit": null,
```

busybox 컨테이너의 PID 확인

실습 1. Pod의 Network namespace 격리 확인

```
vagrant@worker-node2:~$ docker ps | grep busybox
12753cb1a39b        busybox                 "sh -c 'sleep infini…"   23 minutes ago      Up 23 minutes
eb20d4578bd5        k8s.gcr.io/pause:3.2    "/pause"                 23 minutes ago      Up 23 minutes
vagrant@worker-node2:~$ docker inspect 12753cb1a39b | grep -i networkmode
            "NetworkMode": "container:eb20d4578bd5c5a9638c619976667183516b628bac08c6861a1aaa12c5d74213",
vagrant@worker-node2:~$ docker inspect 12753cb1a39b | grep -i pid
            "Pid": 24334,
            "PidMode": "",
            "PidsLimit": null,
vagrant@worker-node2:~$ sudo lsns -p 24334
        NS TYPE   NPROCS   PID USER COMMAND
4026531835 cgroup    122     1 root /sbin/init
4026531837 user      122     1 root /sbin/init
4026532396 ipc         2 24169 root /pause
4026532399 net         2 24169 root /pause
4026532474 mnt         1 24334 root sleep infinity
4026532475 uts         1 24334 root sleep infinity
4026532476 pid         1 24334 root sleep infinity
```

buxybox 컨테이너의 Namespace 확인

→ Pause 컨테이너와 네트워크 네임스페이스를 공유

실습 2. Pod의 veth peer 확인

```
vagrant@control-plane:~$ kubectl exec busybox1 -- ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue 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
3: eth0@if10: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1450 qdisc noqueue
    link/ether ae:ac:9d:3d:10:68 brd ff:ff:ff:ff:ff:ff
    inet 10.244.2.6/24 brd 10.244.2.255 scope global eth0
       valid_lft forever preferred_lft forever
```

buxybox1 Pod의 “veth” 확인

실습 2. Pod의 veth peer 확인

```
vagrant@control-plane:~$ kubectl exec busybox1 -- ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue 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
3: eth0@if10  <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1450 qdisc noqueue
    link/ether ae:ac:9d:3d:10:68 brd ff:ff:ff:ff:ff:ff
    inet 10.244.2.6/24 brd 10.244.2.255 scope global eth0
       valid_lft forever preferred_lft forever
```

```
vagrant@worker-node2:~$ ip addr show type veth
7: veth9247e89a@if3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue
group default
    link/ether f6:dd:84:ed:cc:9d brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

"eth@ifXX"에서 if 뒤의 숫자는 한 쌍으로 연결된 peer veth의 interface index를 뜻한다.
(= busybox1이 실행 중인 노드의 default network namespace에서 10번 interface와 한 쌍)

```
    inet6 fe80::6d:7dff:fe84:ca2d/64 scope link
       valid_lft forever preferred_lft forever
10: veth10e030c1@if3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue
group default
    link/ether e2:24:75:50:8e:bd brd ff:ff:ff:ff:ff:ff link-netnsid 2
    inet6 fe80::e024:75ff:fe50:8ebd/64 scope link
       valid_lft forever preferred_lft forever
```

실습 3. Routing table 확인

```
vagrant@worker-node2:~$ ip route
default via 10.0.2.2 dev eth0 proto dhcp src 10.0.2.15 metric 100
10.0.2.0/24 dev eth0 proto kernel scope link src 10.0.2.15
10.0.2.2 dev eth0 proto dhcp scope link src 10.0.2.15 metric 100
10.244.0.0/24 via 10.244.0.0 dev flannel.1 onlink
10.244.1.0/24 via 10.244.1.0 dev flannel.1 onlink
10.244.2.0/24 dev cni0 proto kernel scope link src 10.244.2.1
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdown
192.168.102.0/24 dev eth1 proto kernel scope link src 192.168.102.4
```

Pod IP 대역(10.244.0.0/16)

실습 3. Routing table 확인

```
vagrant@worker-node2:~$ ip route
default via 10.0.2.2 dev eth0 proto dhcp src 10.0.2.15 metric 100
10.0.2.0/24 dev eth0 proto kernel scope link src 10.0.2.15
10.0.2.2 dev eth0 proto dhcp scope link src 10.0.2.15 metric 100
10.244.0.0/24 via 10.244.0.0 dev flannel.1 onlink
10.244.1.0/24 via 10.244.1.0 dev flannel.1 onlink
10.244.2.0/24 dev cni0 proto kernel scope link src 10.244.2.1
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdown
192.168.102.0/24 dev eth1 proto kernel scope link src 192.168.102.4
```

다른 노드들에 할당된 Pod IP 대역(10.244.2.0/24)은
“flannel.1” 인터페이스로 라우팅

실습 3. Routing table 확인

```
vagrant@worker-node2:~$ ip -d link show flannel.1
5: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN mode DE
    link/ether c6:e9:1f:07:8a:a9 brd ff:ff:ff:ff:ff:ff promiscuity 0
    vxlan id 1 local 192.168.102.4 dev eth1 srcport 0 0 dstport 8472 nolearning ttl inherit
_size 65536 gso_max_segs 65535
```

“flannel.1” 인터페이스는 VXLAN

실습 3. Routing table 확인

```
vagrant@worker-node2:~$ ip route
default via 10.0.2.2 dev eth0 proto dhcp src 10.0.2.15 metric 100
10.0.2.0/24 dev eth0 proto kernel scope link src 10.0.2.15
10.0.2.2 dev eth0 proto dhcp scope link src 10.0.2.15 metric 100
10.244.0.0/24 via 10.244.0.0 dev flannel.1 onlink
10.244.1.0/24 via 10.244.1.0 dev flannel.1 onlink
10.244.2.0/24 dev cni0 proto kernel scope link src 10.244.2.1
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdown
192.168.102.0/24 dev eth1 proto kernel scope link src 192.168.102.4
```

현재 노드(worker-node2)에 할당된 Pod IP 대역(10.244.2.0/24)은
“cni0” 인터페이스로 라우팅

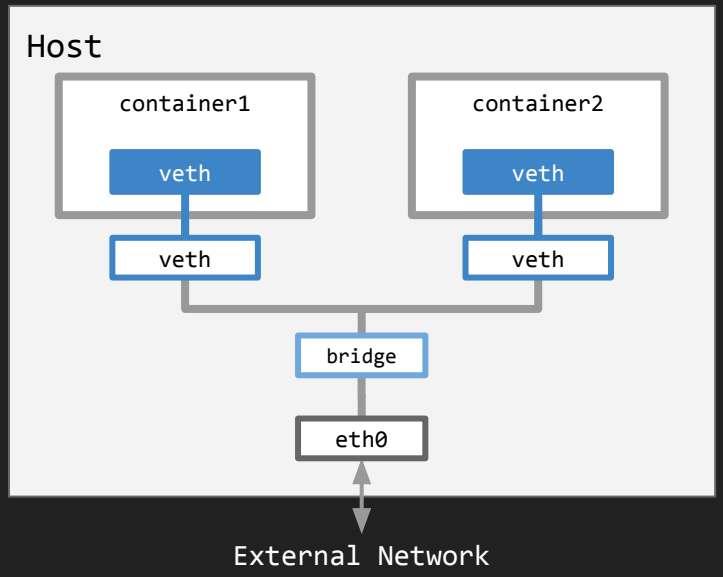
실습 3. Routing table 확인

```
vagrant@worker-node2:~$ ip -d link show cni0
6: cni0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP mode DE
    link/ether 42:9d:ed:e5:e7:49 brd ff:ff:ff:ff:ff:ff promiscuity 0
    bridge forward_delay 1500 hello_time 200 max_age 2000 ageing_time 30000 stp_st
```

“cni0” 인터페이스는 Bridge

(Kubernetes에서는 Docker의 기본 Bridge(docker0)를 사용하지 않는다)

실습 4. “cni0” bridge에 연결된 interface 확인




```
# bridge-utils 설치
$ sudo apt install bridge-utils
```

```
vagrant@worker-node2:~$ brctl show cni0
bridge name     bridge id               STP enabled     interfaces
cni0            8000.429dede5e749       no              veth10e030c1
                                                        veth9247e89a
                                                        vethc5577950
```

cni0에는 3개의 veth가 연결되어 있는 것을 알 수 있다.

실습 5. ARP, FDB 확인

(ARP) 목적지 Pod IP의 MAC address 매핑 정보 (flannel.1, cni0)

```
vagrant@worker-node2:~$ ip neigh show
10.0.2.2 dev eth0 lladdr 52:54:00:12:35:02 DELAY
10.0.2.3 dev eth0 lladdr 52:54:00:12:35:03 STALE
10.244.2.6 dev cni0 lladdr ae:ac:9d:3d:10:68 REACHABLE
10.244.2.4 dev cni0 lladdr fe:01:65:c3:eb:94 STALE
10.244.2.5 dev cni0 lladdr 3a:f0:72:2b:88:e9 STALE
10.244.2.3 dev cni0 lladdr 52:9d:19:25:06:14 STALE
10.244.1.0 dev flannel.1 lladdr 8a:88:e4:a1:fd:34 PERMANENT
192.168.102.1 dev eth1 lladdr 0a:00:27:00:00:06 STALE
192.168.102.2 dev eth1 lladdr 08:00:27:c9:6e:5e REACHABLE
192.168.102.3 dev eth1 lladdr 08:00:27:f2:e7:1b REACHABLE
10.244.0.0 dev flannel.1 lladdr c6:d3:4a:9e:96:1c PERMANENT
```

실습 5. ARP, FDB 확인

(ARP) 목적지 Pod IP의 MAC address 매핑 정보 (flannel.1, cni0)

```
vagrant@worker-node2:~$ ip neigh show
10.0.2.2 dev eth0 lladdr 52:54:00:12:35:02 DELAY
10.0.2.3 dev eth0 lladdr 52:54:00:12:35:03 STALE
10.244.2.6 dev cni0 lladdr ae:ac:9d:3d:10:68 REACHABLE
10.244.2.4 dev cni0 lladdr fe:01:65:c3:eb:94 STALE
10.244.2.5 dev cni0 lladdr 3a:f0:72:2b:88:e9 STALE
10.244.2.3 dev cni0 lladdr 52:9d:19:25:06:14 STALE
10.244.1.0 dev flannel.1 lladdr 8a:88:e4:a1:fd:34 PERMANENT
192.168.102.1 dev eth1 lladdr 0a:00:27:00:00:06 STALE
192.168.102.2 dev eth1 lladdr 08:00:27:c9:6e:5e REACHABLE
192.168.102.3 dev eth1 lladdr 08:00:27:f2:e7:1b REACHABLE
10.244.0.0 dev flannel.1 lladdr c6:d3:4a:9e:96:1c PERMANENT
```

(FDB) 해당 MAC Address의 Flannel VXLAN의 가 존재하는 노드 IP를 알 수 있다.

```
vagrant@worker-node2:~$ bridge fdb show | grep "8a:88:e4:a1:fd:34"
8a:88:e4:a1:fd:34 dev flannel.1 dst 192.168.102.3 self permanent
```

실습 5. ARP, FDB 확인

(ARP) 목적지 Pod IP의 MAC address 매핑 정보 (flannel.1, cni0)

```
vagrant@worker-node2:~$ ip neigh show
10.0.2.2 dev eth0 lladdr 52:54:00:12:35:02 DELAY
10.0.2.3 dev eth0 lladdr 52:54:00:12:35:03 STALE
10.244.2.6 dev cni0 lladdr ae:ac:9d:3d:10:68 REACHABLE
10.244.2.4 dev cni0 lladdr fe:01:65:c3:eb:94 STALE
10.244.2.5 dev cni0 lladdr 3a:f0:72:2b:88:e9 STALE
10.244.2.3 dev cni0 lladdr 52:9d:19:25:06:14 STALE
10.244.1.0 dev flannel.1 lladdr 8a:88:e4:a1:fd:34 PERMANENT
192.168.102.1 dev eth1 lladdr 0a:00:27:00:00:06 STALE
192.168.102.2 dev eth1 lladdr 08:00:27:c9:6e:5e REACHABLE
192.168.102.3 dev eth1 lladdr 08:00:27:f2:e7:1b REACHABLE
10.244.0.0 dev flannel.1 lladdr c6:d3:4a:9e:96:1c PERMANENT
```

(FDB) 해당 MAC Address의 Flannel VXLAN의 가 존재하는 노드 IP를 알 수 있다.

```
vagrant@worker-node2:~$ bridge fdb show | grep "8a:88:e4:a1:fd:34"
8a:88:e4:a1:fd:34 dev flannel.1 dst 192.168.102.3 self permanent
```

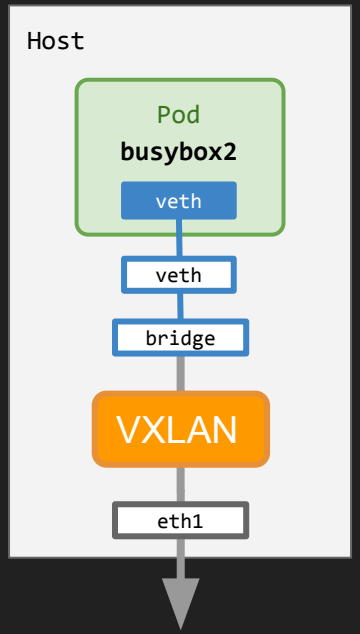
192.168.102.3 노드에 접속해서 flannel.1 VXLAN을 확인해보면,
MAC Address가 일치한다.

```
vagrant@worker-node1:~$ ip link show flannel.1
5: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 (
    link/ether 8a:88:e4:a1:fd:34 brd ff:ff:ff:ff:ff:ff
```

실습 6. VXLAN Encapsulation

```
$ kubectl exec busybox2 -- ping -c 1 <busybox1 IP>
```

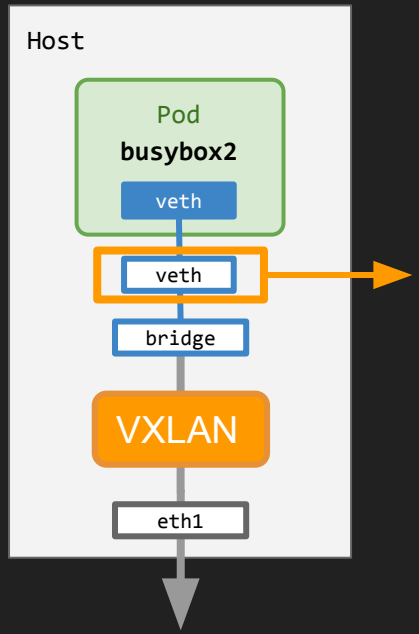
(busybox2 Pod ping → busybox1 Pod)



실습 6. VXLAN Encapsulation

```
$ kubectl exec busybox2 -- ping -c 1 <busybox1 IP>
```

(busybox2 Pod “ping” → busybox1 Pod(다른 노드에서 실행)



ping/pong Packet 헤더에 Pod IP들이 보인다.

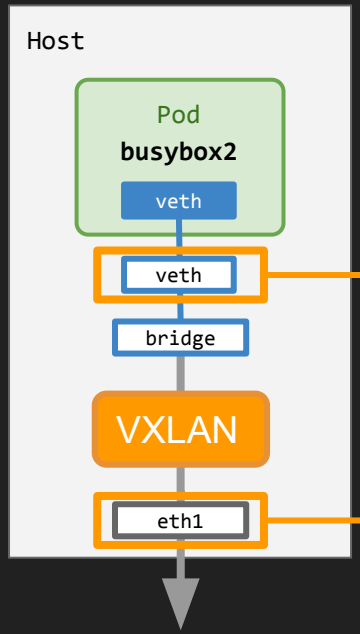
10.244.2.7(busybox2) → 10.244.1.7(busybox1)

```
vagrant@worker-node2:~$ sudo tcpdump -vv -ni vethec3c1e66 icmp
tcpdump: listening on vethec3c1e66, link-type EN10MB (Ethernet), capture size 262144 bytes
19:03:03.182632 IP (tos 0x0, ttl 64, id 28247, offset 0, flags [DF], proto ICMP (1), length 84)
    10.244.2.7 > 10.244.1.7: ICMP echo request, id 4096, seq 0, length 64
```

실습 6. VXLAN Encapsulation

```
$ kubectl exec busybox2 -- ping -c 1 <busybox1 IP>
```

(busybox2 Pod “ping” → busybox1 Pod(다른 노드에서 실행)



ping/pong Packet 헤더에 Pod IP들이 보인다.

10.244.2.7(busybox2) → 10.244.1.7(busybox1)

```
vagrant@worker-node2:~$ sudo tcpdump -vv -ni vethec3c1e66 icmp
tcpdump: listening on vethec3c1e66, link-type EN10MB (Ethernet), capture size 262144 bytes
19:03:03.182632 IP (tos 0x0, ttl 64, id 28247, offset 0, flags [DF], proto ICMP (1), length 84)
    10.244.2.7 > 10.244.1.7: ICMP echo request, id 4096, seq 0, length 64
```

```
vagrant@worker-node2:~$ sudo tcpdump -vv -ni eth1 udp -T vxlan
tcpdump: listening on eth1, link-type EN10MB (Ethernet), capture size 262144 bytes
19:03:03.182681 IP (tos 0x0, ttl 64, id 60220, offset 0, flags [none], proto UDP (17), length 134)
    192.168.102.4.47851 > 192.168.102.3.8472: VXLAN, flags [I] (0x08), vni 1   캡슐화 과정에서 붙은 헤더
IP (tos 0x0, ttl 63, id 28247, offset 0, flags [DF], proto ICMP (1), length 84)
    10.244.2.7 > 10.244.1.7: ICMP echo request, id 4096, seq 0, length 64
```

VXLAN을 거치며 캡슐화된 패킷을 확인할 수 있다.

감사합니다.