

쿠버네티스(Kubernetes) 네트워크 정리

Kubernetes Network



ShinChul Bang

Jan 8 · 24 min read

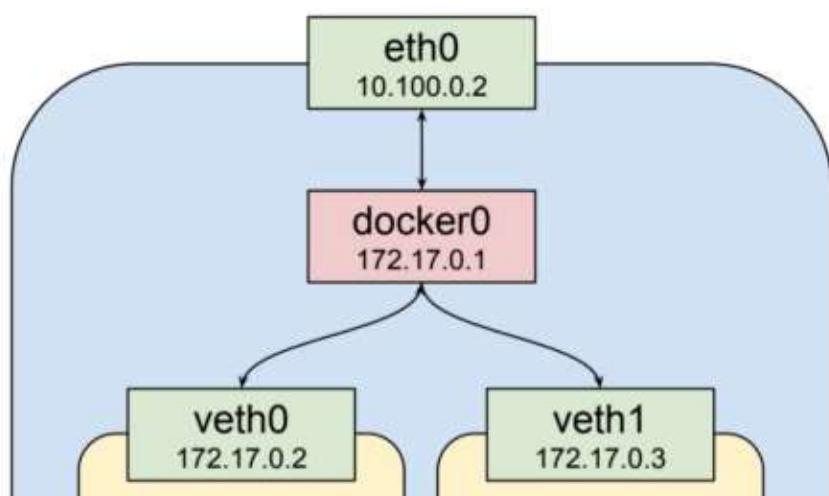
Kubernetes 네트워크는 크게 4가지로 분류된다.

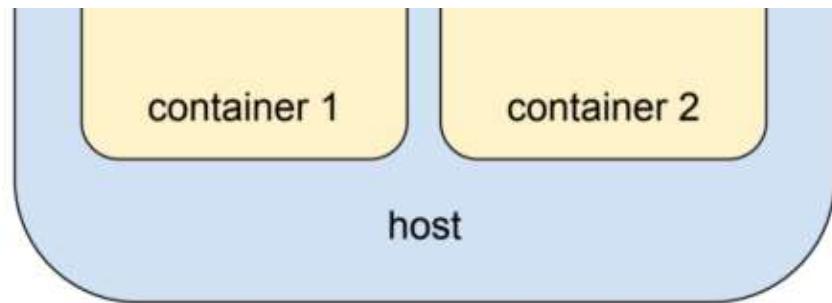
- 서로 결합된 컨테이너와 컨테이너 간 통신
 - Pod와 Pod 간의 통신
 - Pod와 Service간의 통신
 - 외부와 Service간의 통신
- · ·

서로 결합된 컨테이너와 컨테이너 간 통신

Container to Container

먼저 Docker로 생성된 컨테이너의 기본적인 네트워크 동작 구조를 알아보자.

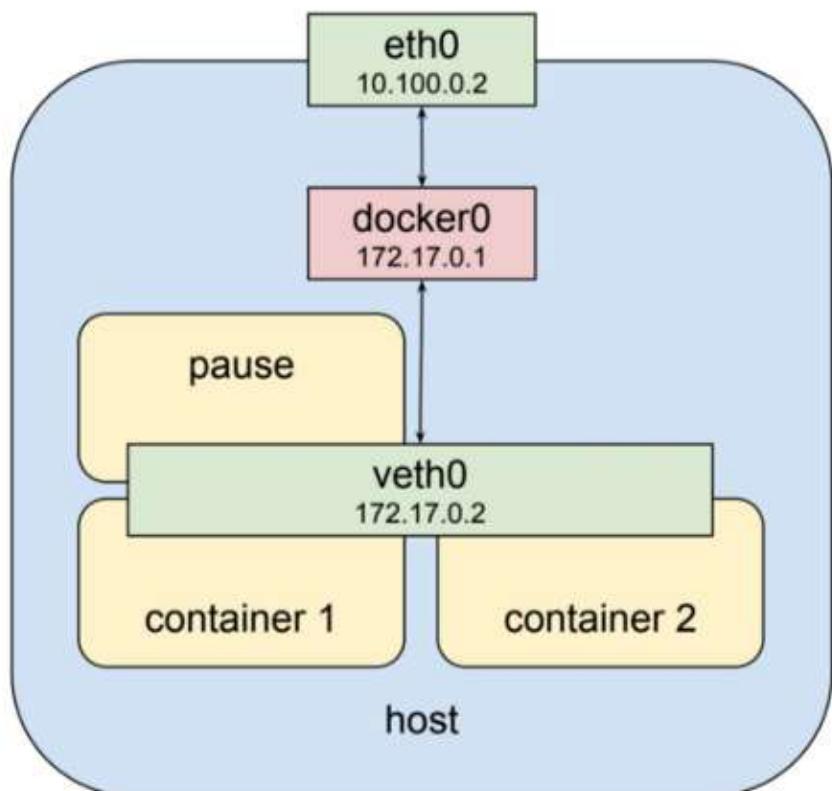




Docker에서는 기본적으로 같은 노드(host) 내의 컨테이너끼리의 통신은 위 그림과 같이 docker0라는 가상 네트워크 인터페이스($172.17.0.0/24$)를 통해 가능하다.

또한 각 컨테이너는 veth라는 가상 네트워크 인터페이스를 고유하게 가지며 따라서 각각의 veth IP 주소 값으로 통신할 수 있다.

아래 그림은 Kubernetes Pod 내의 컨테이너끼리의 통신을 나타낸다.(하지만 Docker에서도 가능하다.)



위 그림에서는 veth0 가상 네트워크 인터페이스에 두 개의 컨테이너가 동시에 할당되어 있다.

즉, 두 개의 컨테이너는 모두 `veth0`라는 동일한 네트워크를 사용하는 것이다.

그렇다면 외부에서는 두 개의 컨테이너가 동일한 IP로 보일 텐데 각 컨테이너를 어떻게 구분할까? 또한 컨테이너끼리는 서로를 어떻게 구분할까?

`veth0` 안에서 각 컨테이너는 고유한 `port` 번호로 서로를 구분한다.**

따라서 Pod 내에서 컨테이너는 각자 고유한 `port` 번호를 사용해야 한다.

이러한 네트워크 인터페이스를 제공해주는 특별한 컨테이너들이 있다.

kubernetes Pod가 실행되고 있는 워커 노드에 들어가서 `docker ps` 명령어를 입력하면 적어도 한 개 이상의 `pause`라는 명령으로 실행된 컨테이너를 볼 수 있다.

이 특별한 컨테이너들은 각 Pod마다 존재하며 다른 컨테이너들에게 네트워크 인터페이스를 제공하는 역할만 담당한다.

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
bf261d157914	/coredns -conf /etc...	"coredns -conf /etc..."	About an hour ago	Up About an hour		k8s_coredns_coredns-5644d7b6d9-wdt7j_kube-system_50edad5e-9176-4927-8c2e-658c6668c53f_2
k8s_acr.io/pause:3.1	"/pause"	"/pause"	About an hour ago	Up About an hour		k8s_POD_coredns-5644d7b6d9-wdt7j_kube-system_50edad5e-9176-4927-8c2e-658c6668c53f_7
bf261d157914	/coredns -conf /etc...	"coredns -conf /etc..."	About an hour ago	Up About an hour		k8s_coredns_coredns-5644d7b6d9-x97vf_kube-system_f2b2cb51-ac48-438c-8dd6-0fb6b5544282_2
k8s_acr.io/pause:3.1	"/pause"	"/pause"	About an hour ago	Up About an hour		k8s_POD_coredns-5644d7b6d9-x97vf_kube-system_f2b2cb51-ac48-438c-8dd6-0fb6b5544282_6
ff281650a721	/opt/bin/flanneld -...	"/opt/bin/flanneld -..."	About an hour ago	Up About an hour		k8s_kube-flannel_kube-flannel-ds-and64-n7n7q_kube-system_708c61de-9ce8-46d5-9fe8-1a74d43f4bba_2
9b65a0f78b09	"/usr/local/bin/kube..."	"/usr/local/bin/kube..."	About an hour ago	Up About an hour		k8s_kube-proxy_kube-proxy-b5hq_kube-system_31cccd30c-bffd-4b4b-af7f-f2fe4f418e85_2
k8s_gcr.io/pause:3.1	"/pause"	"/pause"	About an hour ago	Up About an hour		k8s_POD_kube-flannel-ds-and64-n7n7q_kube-system_708c61de-9ce8-46d5-9fe8-1a74d43f4bba_2
k8s_gcr.io/pause:3.1	"/pause"	"/pause"	About an hour ago	Up About an hour		k8s_POD_kube-proxy-b5hq_kube-system_31cccd30c-bffd-4b04-af7f-f2fe4f418e85_2
bb16442bcdb4	"kube-controller-man..."	"kube-controller-man..."	About an hour ago	Up About an hour		k8s_kube-controller-manager-bsc_kube-master_kube-system_59b476ab714ef48826b7493749d

Pod가 돌아가고 있는 워커 노드에서 `docker ps` 명령어를 입력하면 이런 컨테이너들을 확인할 수 있다.

`pause` 명령으로 실행된 컨테이너는 kubernetes가 SIGTERM 명령을 내리기 전까지 아무것도 하지 않고 `sleep` 상태로 존재한다.

위 그림에서 볼 수 있는 네트워크를 담당하는 컨테이너들의 리스트는 아래와 같다.

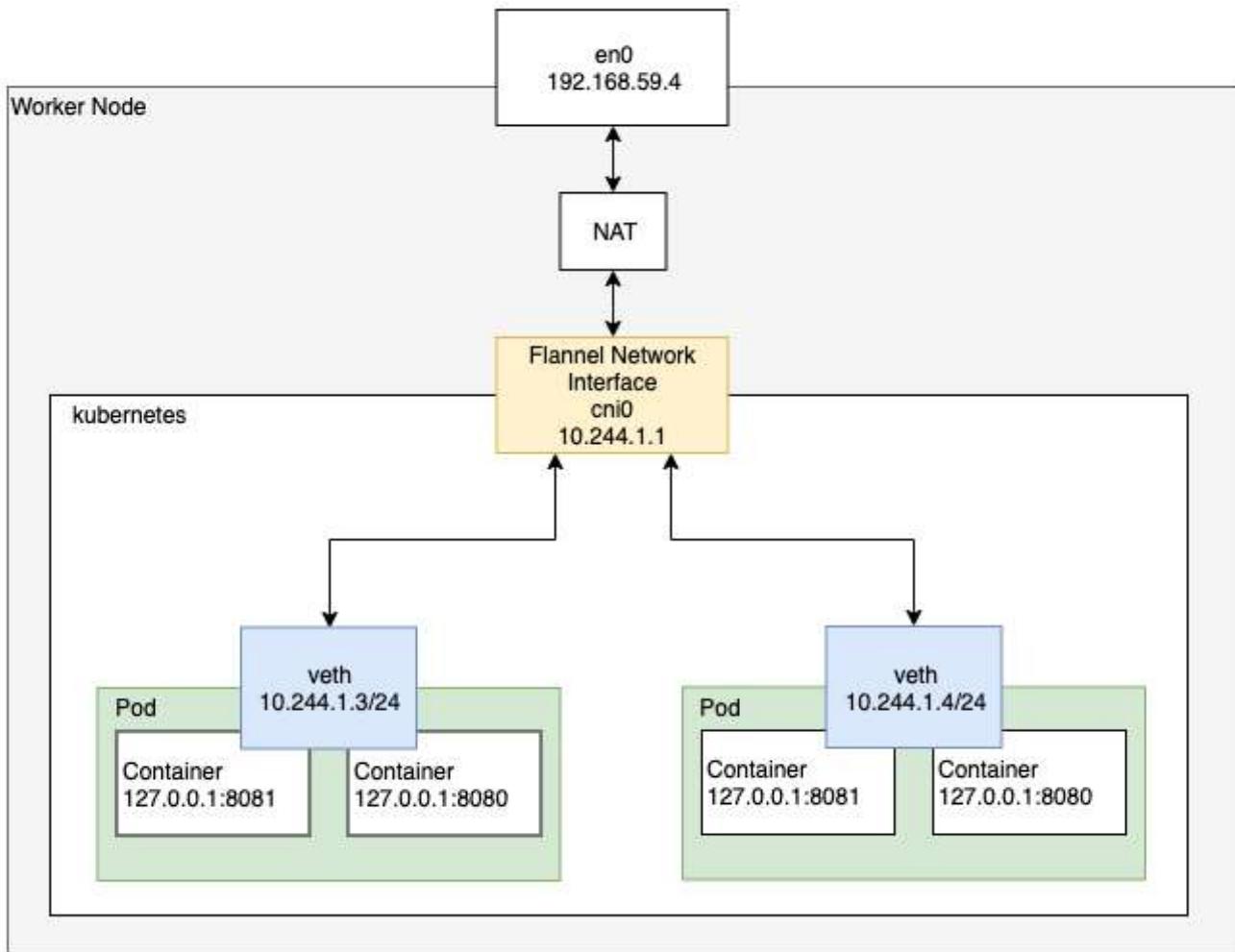
- coredns
- kube-flannel
- kube-proxy

• • •

Pod와 Pod 간의 통신

Pod to Pod

- 싱글 노드 Pod 네트워크



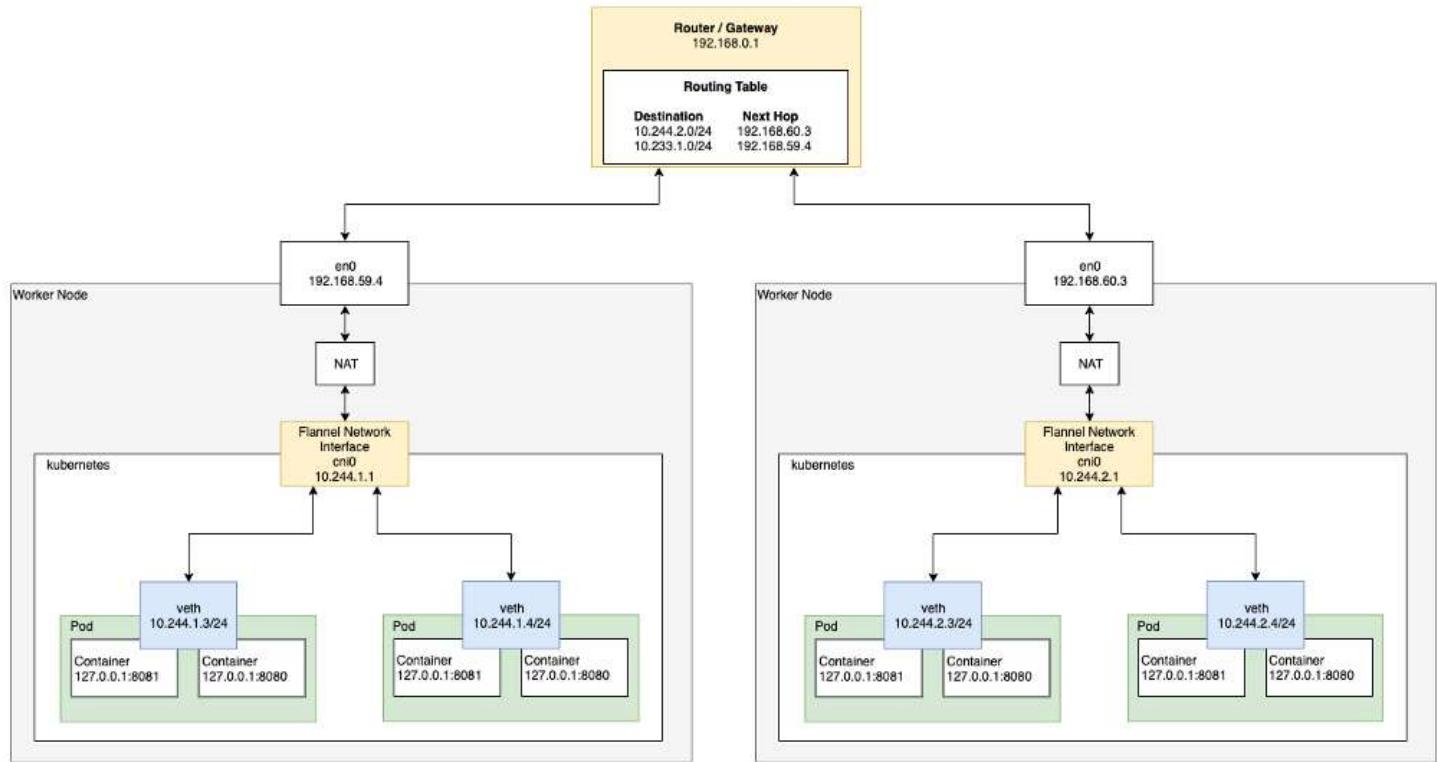
기본적으로 kubernetes는 kubenet이라는 아주 기본적이고 간단한 네트워크 플러그인을 제공 해주지만, 정말로 매우 기본적이고 간단한 기능만 제공해주는 네트워크 플러그인이기 때문에 이 자체로는 크로스 노드 네트워킹이나 네트워크 정책 설정과 같은 고급 기능은 구현되어 있지 않다.

따라서 kubernetes에서는 Pod 네트워킹 인터페이스로 CNI 스펙을 준수하는 다양한 네트워크 플러그인을 사용하는 것을 권장한다. (kubernetes에서 제공해주는 클러스터 빌드 전용 커맨드라인 인터페이스인 `kubeadm`은 기본적으로 CNI 기반 네트워크 플러그인만 사용할 수 있도록 되어 있다.)

Pod의 특징 중 하나로 각 Pod는 고유한 IP 주소를 가진다. (이것이 위에서 설명한 가상 네트워크 인터페이스 `veth`이다.)

따라서 각 Pod는 kubenet 혹은 CNI로 구성된 네트워크 인터페이스를 통하여 고유한 IP 주소로 서로 통신할 수 있다.

- 멀티 노드 Pod 네트워크



여러 개의 워커 노드 사이에 각각 다른 노드에 존재하는 Pod가 서로 통신하려면 라우터를 거쳐서 통신하게 된다.

• • •

Pod와 Service간의 통신

Pod to Service

Pod는 기본적으로 쉽게 대체될 수 있는 존재이기 때문에 Pod to Pod Network만으로는 Kubernetes 시스템을 내구성있게 구축할 수 없다.

어떠한 말이냐면, Pod IP를 어떤 서비스의 엔드포인트로 설정하는 것은 가능하지만, 해당 Pod가 계속 존재하고 있을 것이라는 보장도 없고 새로운 Pod가 생성되었을 때 그 IP 주소가 엔드포인트와 동일할 것이라고 보장할 수 없다는 것이다.

이를 해결하기 위해서는 서비스 앞단에 reverse-proxy(혹은 Load Balancer)를 위치시키는 방법이 있다.

클라이언트에서 proxy로 연결을 하면 proxy의 역할은 서버들 목록을 관리하며 현재 살아있는 서버에게 트래픽을 전달하는 것이다.

이는 몇 가지 요구사항을 만족해야 한다.

- proxy 서버 스스로 내구성이 있어야 하며 장애에 대응할 수 있어야 한다.
- 트래픽을 전달할 서버 리스트를 가지고 있어야 한다.
- 서버 리스트 내 서버들이 정상적인지 확인할 수 있는 방법을 알아야 한다.

Kubernetes 설계자들은 이 문제를 굉장히 우아한 방법으로 해결하였다.

그들은 기존의 시스템을 잘 활용하여 위 3가지 요구사항을 만족하는 것을 만들었고 그 것을 `service` 리소스 타입이라고 정의하였다.

`service` 란 kubernetes 리소스 타입 중 하나로 각 Pod로 트래픽을 포워딩 해주는 프록시 역할을 한다.

이 때 `selector` 라는 것을 이용하여 트래픽을 전달받을 Pod들을 결정한다.

Pod 네트워크와 동일하게 `service` 네트워크 또한 가상 IP 주소이다.

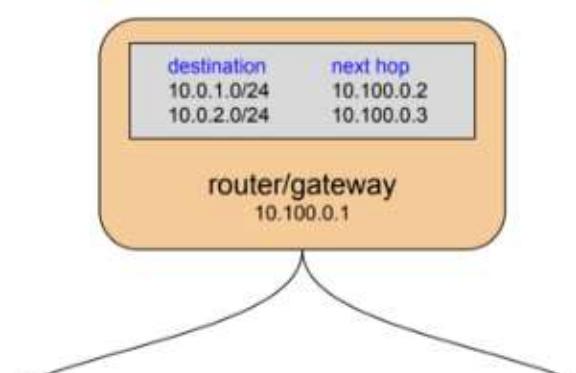
하지만 Pod 네트워크와는 조금 다르게 동작한다.

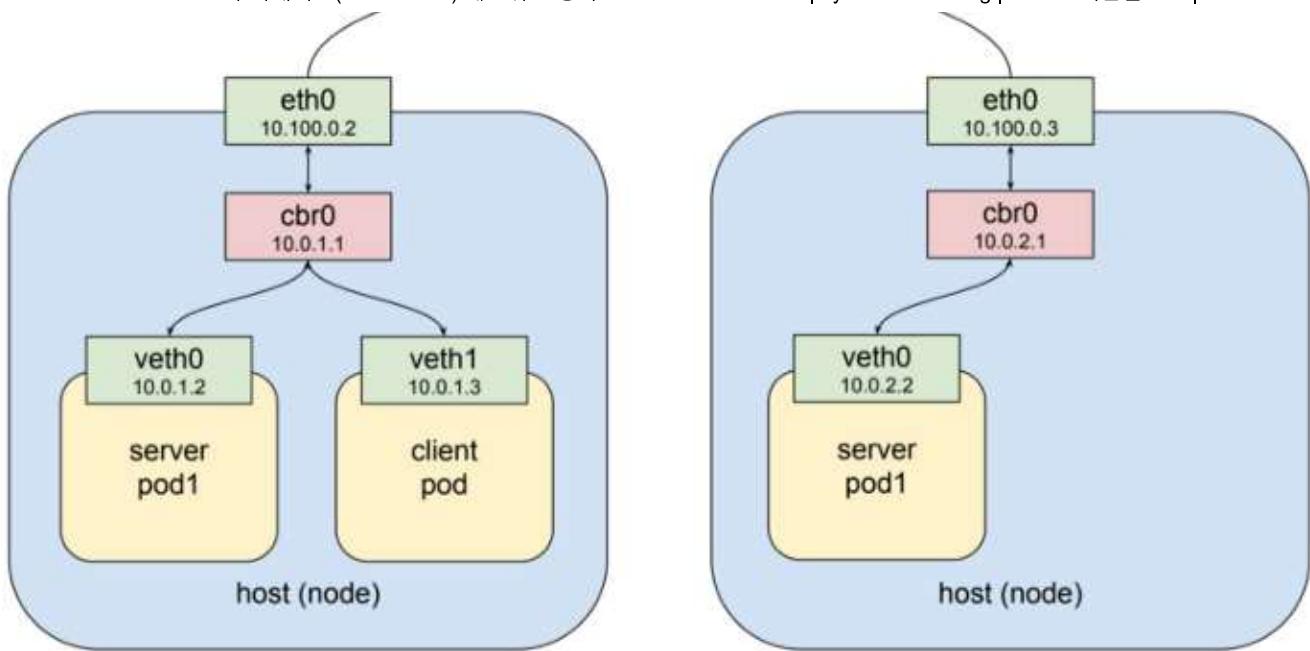
Pod 네트워크는 실질적으로 가상 이더넷 네트워크 인터페이스(veth)가 세팅되어져 `ifconfig`에서 조회할 수 있지만, `service` 네트워크는 `ifconfig`로 조회할 수 없다.

또한 routing 테이블에서도 `service` 네트워크에 대한 경로를 찾아볼 수 없다.

이러한 이유는 `service` 네트워크의 구조와 동작 방식을 통해 확인할 수 있다.

예를 들어 아래와 같은 Pod 네트워크가 구성되어 있다고 가정하자.





위를 보면 두 개의 워커 노드가 있고 하나의 게이트웨이를 통해 서로 연결되어 있다.

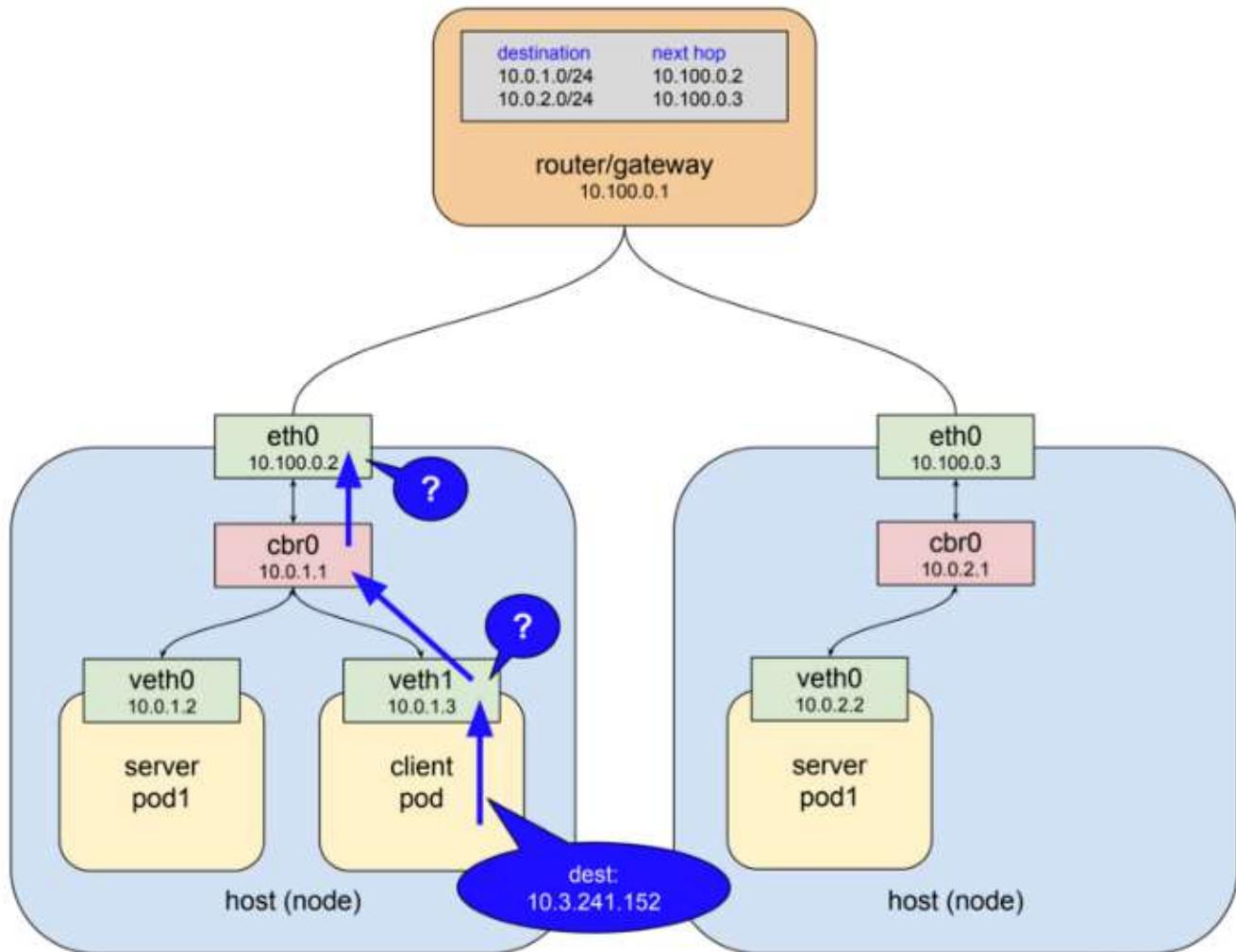
또한 마스터 노드에서(이미지에는 보이지 않지만) 아래의 명세서를 통해 service 네트워크를 생성했다고 가정한다.

```
apiVersion: v1
kind: Service
metadata:
  name: service-test # service의 이름
spec:
  selector:
    app: server_pod1 # 10.0.1.2와 10.0.2.2에서 돌아가고 있는 서버 컨테이너의 pod 라벨
  ports:
    - protocol: TCP
      port: 80 # service에서 서버 컨테이너 어플리케이션과 매핑시킬 포트 번호
      targetPort: 8080 # 서버 컨테이너에서 구동되고 있는 서버 어플리케이션 포트 번호
```

client pod가 service 네트워크를 통해 server pod1으로 http request를 요청하는 과정은 아래와 같다.

1. client pod가 http request를 service-test라는 DNS 이름으로 요청한다.
2. 클러스터 DNS 서버(coredns)가 해당 이름을 service IP(예시로 10.3.241.152이라고 한다)로 매핑시켜준다.
3. http 클라이언트는 DNS로부터 IP를 이용하여 최종적으로 요청을 보내게 된다.

이 과정에 대해 상세하게 알아보자.



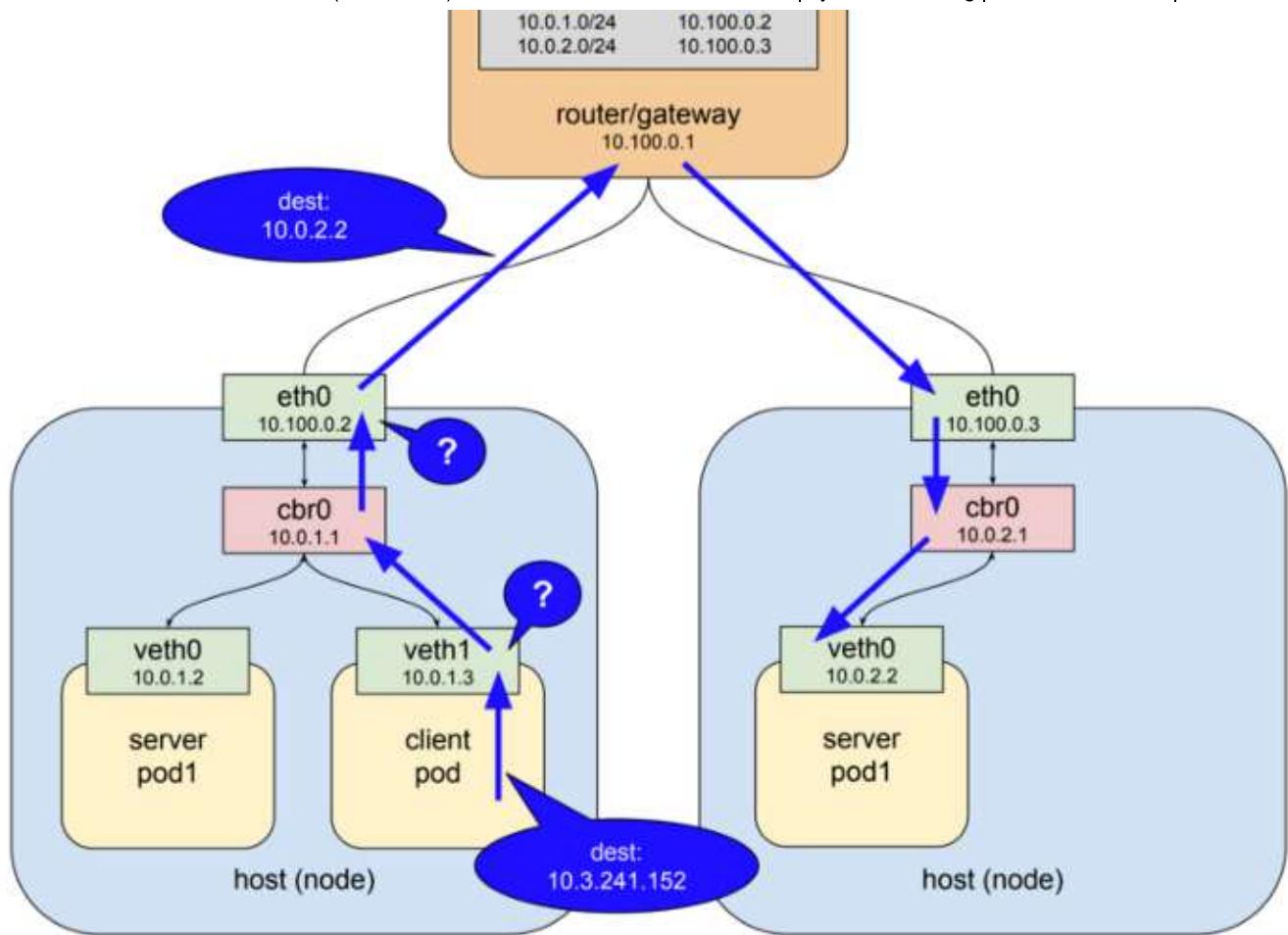
IP 네트워크(Layer 3)는 기본적으로 자신의 host에서 목적지를 찾지 못하면 상위 게이트웨이로 패킷을 전달하도록 동작한다.

예시에서 보자면 client pod 안에 들어 있는 첫번째 가상 이더넷 인터페이스(veth1)에서 목적지 IP를 보게 되고 10.3.241.152라는 주소에 대해 전혀 알지 못하기 때문에 다음 게이트웨이(cbr0)로 패킷을 넘기게 된다.

cbr0는 bridge이기 때문에 단순히 다음 게이트웨이(eth0)로 패킷을 전달한다.

여기서도 마찬가지로 eth0라는 이더넷 인터페이스가 10.3.241.152라는 IP 주소에 대해서 모르기 때문에 보통이라면 최상위에 존재하는 게이트웨이로 전달될 것이다.

하지만 예상과는 달리 아래 그림처럼 특별하게 갑자기 패킷의 목적지 주소가 변경되어 server pod1 중 하나로 패킷이 전달되게 된다.



어떻게 이렇게 동작하게 되는 걸까?

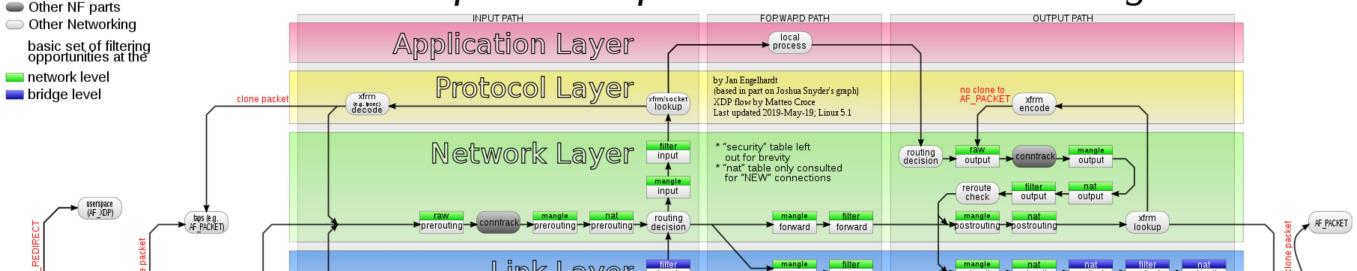
패킷의 흐름이 이렇게 될 수 있는 이유는 kube-proxy라는 컴포넌트 때문이다.

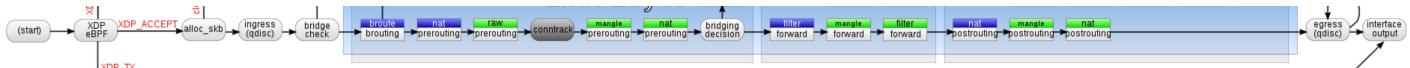
쿠버네티스는 리눅스 커널 기능 중 하나인 netfilter와 user space에 존재하는 인터페이스인 iptables라는 소프트웨어를 이용하여 패킷 흐름을 제어한다.

netfilter란 Rule-based 패킷 처리 엔진이며, kernel space에 위치하여 모든 오고 가는 패킷의 생명주기를 관찰한다. 그리고 규칙에 매칭되는 패킷을 발견하면 미리 정의된 action을 수행한다.

iptables는 netfilter를 이용하여 chain rule이라는 규칙을 지정하여 패킷을 포워딩 하도록 네트워크를 설정한다.

Packet flow in Netfilter and General Networking

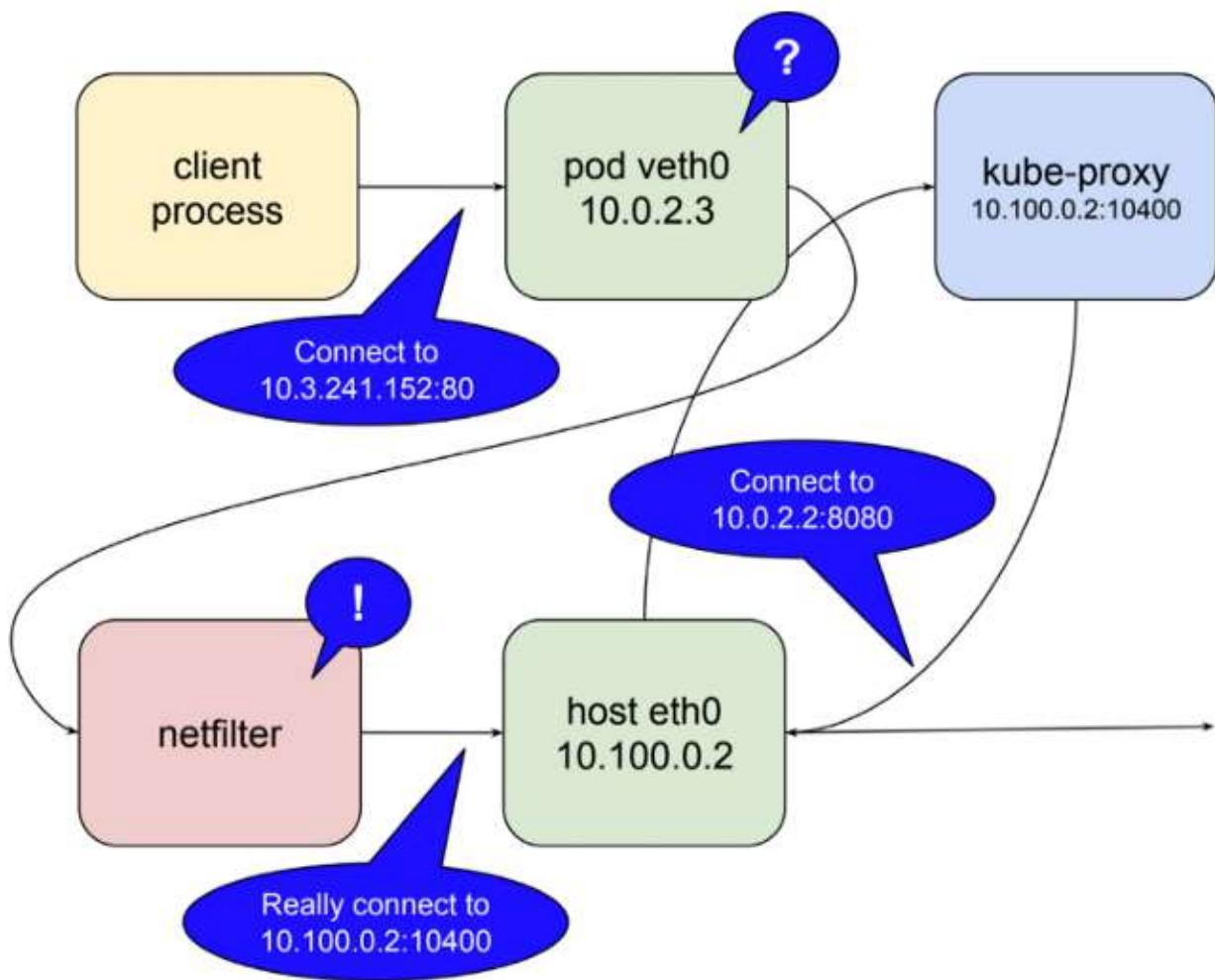




netfilter에서의 패킷 흐름 다이어그램

많은 action들 중에 특히 목적지의 주소를 변경할 수 있는 action도 있다.(Destination NAT)

쿠버네티스는 이 netfilter를 kernel space에서 proxy(Destination proxy) 형태로 사용한다.



kube-proxy가 user space 모드로 동작할 때 과정은 아래와 같다.

1. kube-proxy가 localhost 인터페이스에서 service의 요청을 받아내기 위해 10400 포트(임의)를 연다.
2. kube-proxy가 netfilter로 하여금 service IP(10.3.241.152:80)로 들어오는 패킷을 kube-proxy 자신에게 라우팅 되도록 설정을 한다.

3. kube-proxy로 들어온 요청을 실제 server pod의 IP:Port(예제에서는 10.0.2.2:8080)로 요청을 전달한다.

이러한 방법을 통해 service IP(10.3.241.152:80)로 들어온 요청을 마법처럼 실제 server pod가 위치한 10.0.2.2:8080으로 전달할 수 있다.

netfilter의 능력을 보자면, 이 모든 것을 하기 위해서는 단지 kube-proxy가 자신의 포트를 열고 마스터 API Server로부터 전달 받은 service 정보를 netfilter에 알맞는 규칙으로 입력하는 것 외엔 다른 것이 없다.

하지만 위와 같이 user space에서 proxying 하는 것은 모든 패킷을 user space에서 kernel space로 변환을 해야하기 때문에 그만큼 비용이 든다.

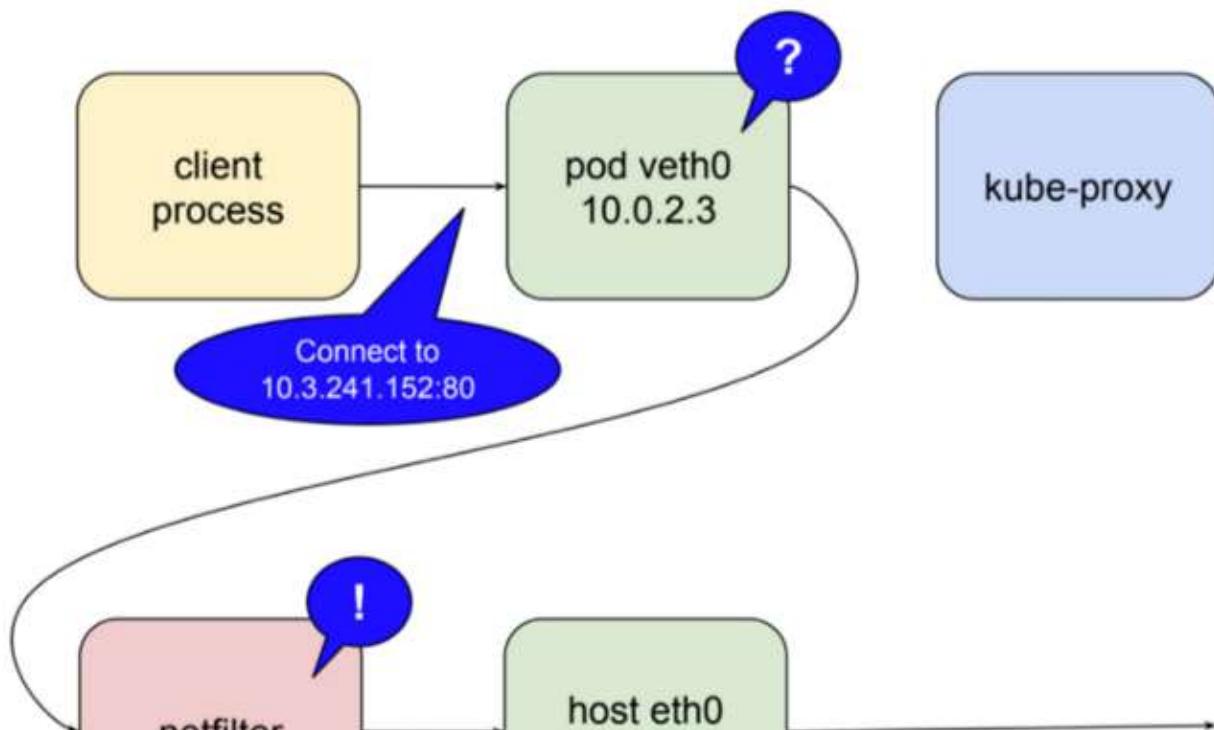
그래서 kubernetes 1.2 버전 이상의 kube-proxy에서는 이 문제를 해결하기 위해 iptables mode가 생겼다.

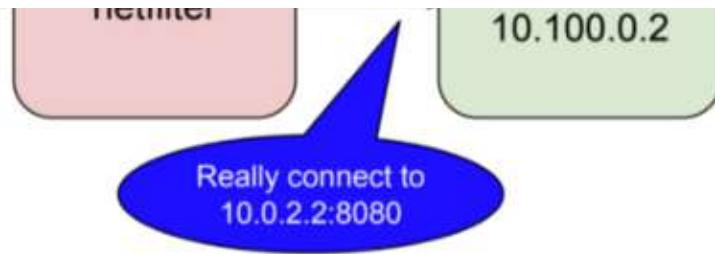
이 모드에서는 kube-proxy가 직접 proxy의 역할을 수행하지 않고 그 역할을 전부 netfilter에게 맡긴다.

이를 통해 service IP를 발견하고 그것을 실제 Pod로 전달하는 것은 모두 netfilter가 담당하게 되었고

kube-proxy는 단순히 netfilter의 규칙을 알맞게 수정하는 것을 담당할 뿐이다.

iptables mode의 동작 예시는 아래 그림과 같다.





아래는 간단한 hello world를 출력하는 웹 서버 Pod를 4개 띄워서 Service에 대한 NAT table을 netfilter에서 어떻게 구성하는지 볼 수 있는 예제이다.

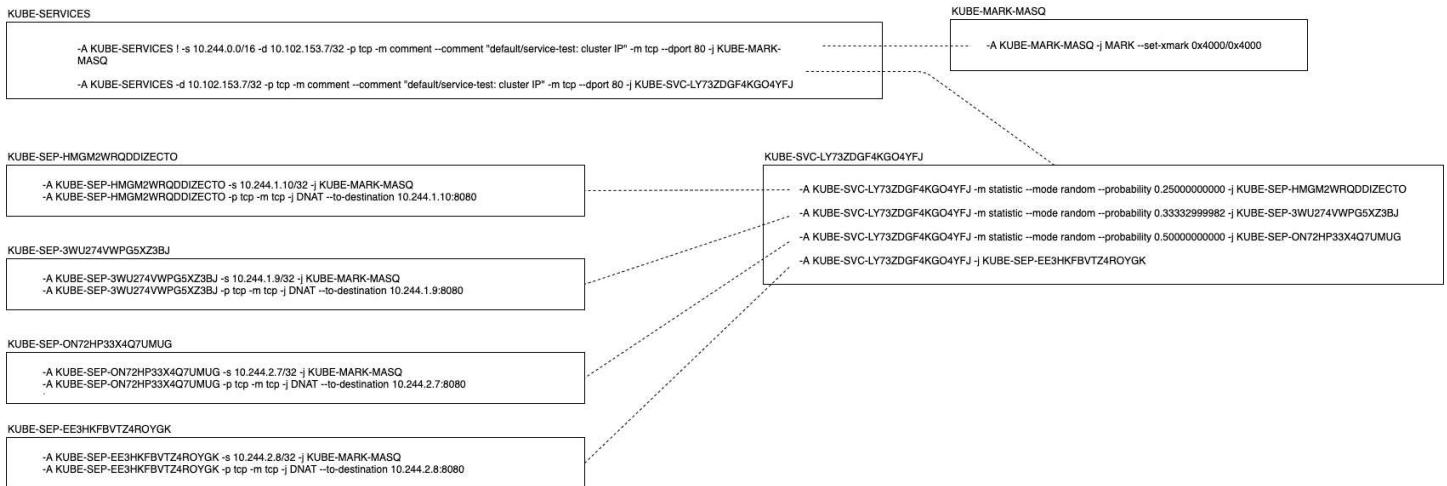
아래의 deployment와 service 패니페스트를 사용하여 테스트한다.

```
# Hello world Server Pod
kind: Deployment
apiVersion: apps/v1
metadata:
  name: service-test
spec:
  replicas: 4
  selector:
    matchLabels:
      app: service_test_pod
  template:
    metadata:
      labels:
        app: service_test_pod
    spec:
      containers:
        - name: simple-http
          image: python:2.7
          imagePullPolicy: IfNotPresent
          command: ["/bin/bash"]
          args: ["-c", "echo \\\"<p>Hello from $(hostname)</p>\\\" > index.html; python -m SimpleHTTPServer 8080"]
      ports:
        - name: http
          containerPort: 8080
---
# Hello world Server Service
apiVersion: v1
kind: Service
metadata:
  name: service-test
spec:
  selector:
    app: service_test_pod
  ports:
    - protocol: TCP
      port: 80
      targetPort: http
```

위의 매니페스트로 Pod와 Service를 생성하면 sudo iptables -S -t nat 명령어를 통해 아래와 같이 해당 노드에 설정되어 있는 NAT 테이블을 조회할 수 있다.

KUBE-SERVICES, KUBE-MARK-MASQ, KUBE-SVC-LY73ZDGF4KGO4YFJ, KUBE-SEP-어쩌구저쩌구 등 많은 netfilter의 체인 룰을 확인할 수 있는데 이 체인 룰은 kube-proxy가 클러스터에 정의된 Service를 감지하여 알맞은 환경으로 생성하거나 혹은 삭제하거나 하면서 Service 네트워크를 관리하는 것이다.

```
test@test-kube-master:~$ kubectl get pods -o wide
NAME           READY   STATUS    RESTARTS   AGE     IP          NODE      NOMINATED-NODE   READINESS   GATES
service-test-65cbbb5968-5pwlt  1/1    Running   0          5s     10.244.2.8   test-kube-worker-2 <none>        <none>
service-test-65cbbb5968-918rm  1/1    Running   0          5s     10.244.1.10  test-kube-worker-1 <none>        <none>
service-test-65cbbb5968-b92bl  1/1    Running   0          5s     10.244.2.7   test-kube-worker-2 <none>        <none>
service-test-65cbbb5968-qn9lj  1/1    Running   0          5s     10.244.1.9   test-kube-worker-1 <none>        <none>
test@test-kube-master:~$ kubectl get service
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP  10.96.0.1    <none>        443/TCP   7d23h
service-test  ClusterIP  10.102.153.7  <none>        80/TCP    18s
test@test-kube-master:~$ kubectl get nodes -o wide
NAME            STATUS   ROLES    AGE     VERSION   INTERNAL-IP       EXTERNAL-IP   OS-IMAGE      KERNEL-VERSION   CONTAINER-RUNTIME
test-kube-master  Ready    master   7d23h   v1.16.3   192.168.57.4   <none>        Ubuntu 18.04.3 LTS  4.15.0-70-generic  docker://18.6.2
test-kube-worker-1  Ready    <none>   7d23h   v1.16.3   192.168.99.103  <none>        Ubuntu 18.04.3 LTS  4.15.0-70-generic  docker://18.6.2
test-kube-worker-2  Ready    <none>   17h     v1.16.3   192.168.59.6   <none>        Ubuntu 18.04.3 LTS  4.15.0-70-generic  docker://18.6.2
```



ClusterIP 탑입의 Service 네트워크를 구성하는 netfilter NAT table 정리. iptables -S -t nat 명령어로 조회할 수 있다.

kube-proxy의 내구성

kube-proxy는 기본적으로 systemd unit으로 동작하거나 daemon set으로 설치가 된다.

따라서 프로세스가 죽어도 새로운 컨테이너를 다시 띄워 다시 살아날 수 있다.

kube-proxy가 user space 모드로 동작할 때는 단일 지점 장애점이 될 수 있다.

하지만 iptables 모드로 동작할 때는 꽤나 안정적으로 동작할 수 있다.(kernel space)

왜냐하면 이는 netfilter를 통해 동작하고 호스트 서버가 살아있는 한 netfilter도 동작하는 것을 보장받을 수 있기 때문이다.

service proxy가 healthy server pod를 감지할 수 있을까?

kube-proxy는 마스터 노드의 API Server에서 정보를 수신하기 때문에 클러스터의 변화를 감지한다.

이를 통해 지속적으로 iptables를 업데이트하여 netfilter 규칙을 최신화 한다.

새로운 service가 생성되면 kube-proxy는 알림을 받게 되고 그에 맞는 규칙을 생성한다.

반대로 service가 삭제되면 이와 비슷한 방법으로 규칙을 삭제한다.

서버의 health check는 kubelet을 통해 수행한다.(Pod 내부의 Container health check)

kubelet은 서버에 설치되는 또 다른 쿠버네티스의 컴포넌트 중 하나이다.

이 kubelet이 서버의 health check를 수행하여 문제를 발견 시 마스터 노드의 API Server를 통해 kube-proxy에게 알려 unhealthy Pod의 엔드포인트를 제거한다.

이렇게 Pod to Service 네트워크에 대해서 알아보았다.

하지만 이도 만능은 아니다. 단점이 존재한다.

이런 방식은 클러스터 내부의 Pod에서 요청한 request에 한해서만 위와 같은 방식으로 동작한다.

즉, 외부에서 들어온 요청에 대해서는 다르게 처리해야 한다는 것이다.

· · ·

외부와 Service 간의 통신

External to Service

앞에서 Service 와 Service 네트워크에 대해서 확인해 보았다.

다시 용어를 정리해보자.

service 란, Pod로 액세스 할 수 있는 정책을 정의하는 추상화된 개념이다.

[service] (<<https://kubernetes.io/docs/concepts/services-networking/service/>>) 는 kubernetes 리소스 타입 중 하나로 각 Pod로 트래픽을 포워딩 해주는 프록시 역할을 한다.

이 때 selector 라는 것을 이용하여 트래픽을 전달받을 Pod들을 결정한다.

Service 네트워크는 Service가 할당받는 네트워크 인터페이스이다.

모든 Service는 기본적으로 Cluster-IP라는 IP 주소를 부여받으며, 클러스터 내부적으로 이 IP 주소를 통해 자신이 포워딩 해야 할 Pod들에게 트래픽을 전달한다.

즉, 기본적으로 Service는 클러스터 내부적으로만 통신할 수 있게끔 설계되어 있다.

하지만 Pod는 외부로도 통신이 되어야 한다.

따라서 Service는 여러가지 타입을 통해 외부 통신을 가능하게끔 기능을 제공한다.

아래는 Service에서 외부 통신을 가능하게 해주는 Service의 타입이다.

- NodePort
- Load Balancer

NodePort

NodePort 타입의 서비스는 기본적으로 ClusterIP 타입(default)과 동일하지만 몇 가지 기능들을 더 가지고 있다.

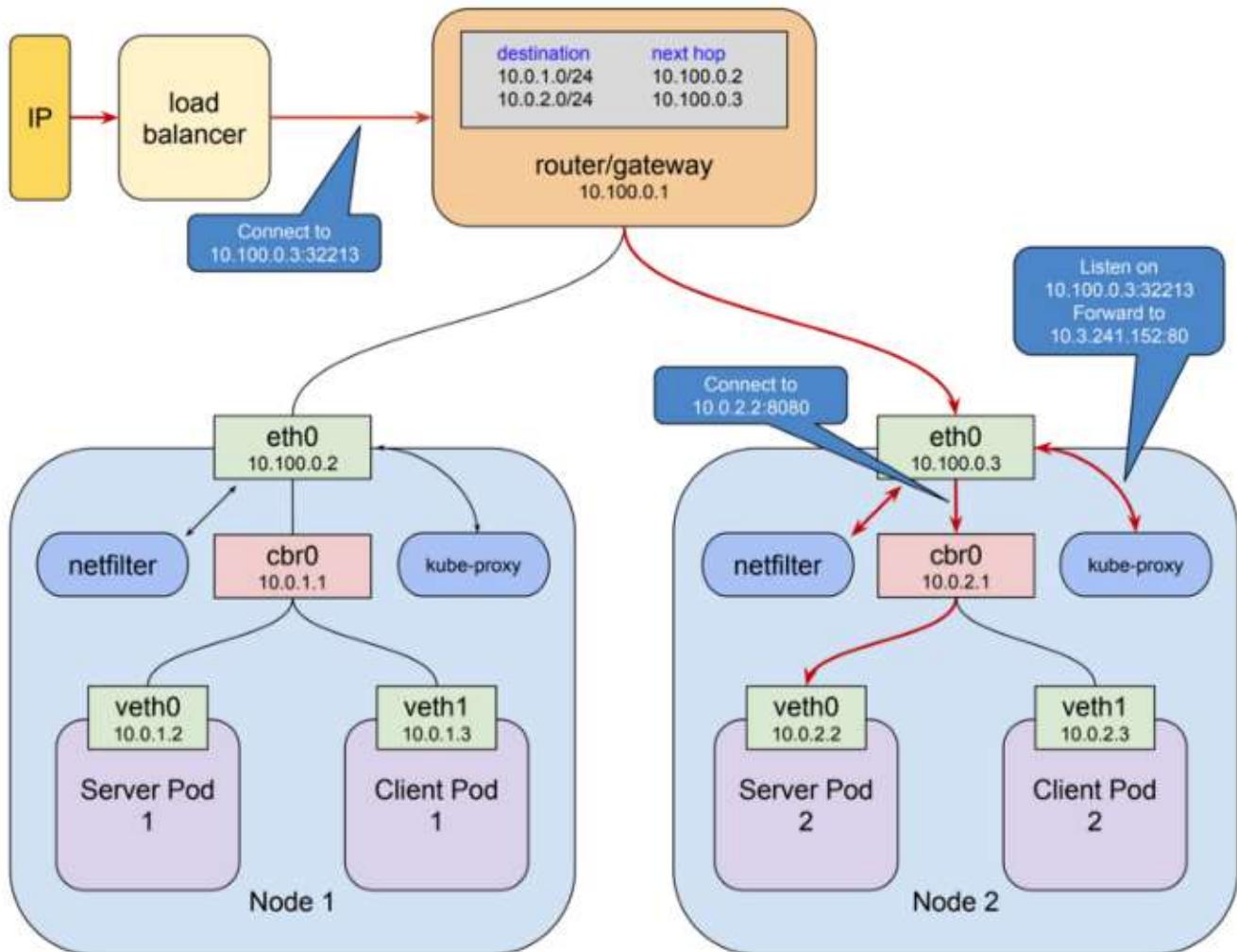
NodePort 타입 서비스는 노드 네트워크의 IP를 통하여 접근 할 수 있을 뿐만 아니라 ClusterIP로도 접근이 가능하다.

이것이 가능한 이유는 간단하다.

쿠버네티스가 NodePort 타입의 서비스를 생성하면 kube-proxy가 각 노드의 eth0 네트워크 interface에 30000–32767 포트 사이의 임의의 포트를 할당한다. (그렇기 때문에 이름이 NodePort이다.)

그리고 할당된 포트로 요청이 오게 되면 이것을 매팅된 ClusterIP로 전달한다. (실제로 따져보자면 ClusterIP로 전달하는 것이 아니라 ClusterIP를 통해 포워딩되는 netfilter 체인 룰로 NAT가 적용되는 것이다.)

아래 그림과 같이 동작한다고 볼 수 있다.



아래는 간단한 hello world를 출력하는 웹 서버 Pod를 4개 띄워서 NodePort 타입의 Service에 대한 NAT table을 어떻게 구성하는지 볼 수 있는 예제이다.

아래의 deployment와 service 매니페스트를 사용하여 테스트 한다.

```
# Hello world Server Pod
kind: Deployment
apiVersion: apps/v1
metadata:
  name: service-test
spec:
  replicas: 4
  selector:
    matchLabels:
      app: service_test_pod
  template:
    metadata:
      labels:
        app: service_test_pod
  spec:
```

```

containers:
  - name: simple-http
    image: python:2.7
    imagePullPolicy: IfNotPresent
    command: ["/bin/bash"]
    args: ["-c", "echo \\\"<p>Hello from $(hostname)</p>\\\" >
index.html; python -m SimpleHTTPServer 8080"]
    ports:
      - name: http
        containerPort: 8080
    ---
# Hello world Server Service
apiVersion: v1
kind: Service
metadata:
  name: service-test
spec:
  type: NodePort
  selector:
    app: service_test_pod
  ports:
    - protocol: TCP
      port: 80
      targetPort: http
      nodePort: 30500
  ---

```

위의 매니페스트로 Pod와 Service를 생성하면 sudo iptables -S -t nat 명령어를 통해 아래와 같이 해당 노드에 설정되어 있는 NAT 테이블을 조회할 수 있다.

여기에서는 이전에 ClusterIP 타입의 서비스를 생성해서 확인해봤던 것과는 조금 다른 것을 확인할 수 있다.

NodePort 타입의 Service에서는 추가적으로 KUBE-NODEPORTS라는 체인 룰이 추가되었다.

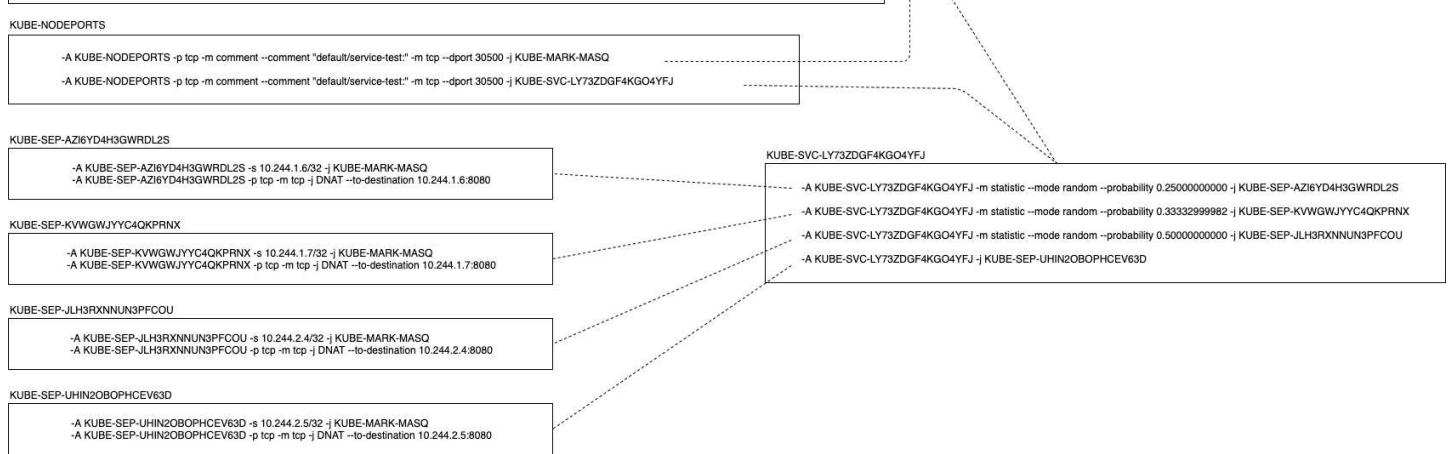
외부에서 들어오는 트래픽이 이 체인 룰을 타고 NAT가 적용되어 Service가 포워딩하는 Pod들로 트래픽을 전달하도록 되어 있다.

```

test@test-kube-master:/usr/sbin$ kubectl get pods -o wide
NAME          READY   STATUS    RESTARTS   AGE     IP           NODE       NOMINATED-NODE   READINESS   GATES
service-test-65ccb5968-bhh47  1/1    Running   1          16h    10.244.2.4  test-kube-worker-2  <none>        <none>
service-test-65ccb5968-fcmp6  1/1    Running   1          16h    10.244.2.5  test-kube-worker-2  <none>        <none>
service-test-65ccb5968-jj5t9  1/1    Running   1          16h    10.244.1.7  test-kube-worker-1  <none>        <none>
service-test-65ccb5968-lwmdh  1/1    Running   1          16h    10.244.1.6  test-kube-worker-1  <none>        <none>
test@test-kube-master:/usr/sbin$ kubectl get service
NAME         TYPE      CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP  10.96.0.1    <none>        443/TCP   7d22h
service-test  NodePort   10.108.205.90  <none>        80:30500/TCP 16h
test@test-kube-master:/usr/sbin$ kubectl get nodes -o wide
NAME          STATUS   ROLES   AGE     VERSION   INTERNAL-IP   EXTERNAL-IP   OS-IMAGE   KERNEL-VERSION   CONTAINER-RUNTIME
test-kube-master  Ready    master   7d22h   v1.16.3   192.168.57.4  <none>      Ubuntu 18.04.3 LTS  4.15.0-70-generic  docker://18.6.2
test-kube-worker-1 Ready    <none>   7d22h   v1.16.3   192.168.99.103 <none>      Ubuntu 18.04.3 LTS  4.15.0-70-generic  docker://18.6.2
test-kube-worker-2 Ready    <none>   16h     v1.16.3   192.168.59.6  <none>      Ubuntu 18.04.3 LTS  4.15.0-70-generic  docker://18.6.2

```





Nodeport 탑입의 Service 네트워크를 구성하는 netfilter NAT table 정리. iptables -S -t nat 명령어로 조회할 수 있다.

Load Balancer

AWS나 Azure같은 외부 클라우드 서비스를 사용하여 로드밸런서를 프로비저닝 할 수 있는 경우에 사용할 수 있는 Service 탑입이다.

로드밸런서의 실제 생성은 비동기적으로 수행되며 프로비저닝 될 로드밸런서에 대한 정보는 Service의 .status.loadBalancer 필드에 작성된다. 예를 들면 아래와 같다.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
  clusterIP: 10.0.171.239
  type: LoadBalancer
status:
  loadBalancer: # 여기에 로드밸런서에 대한 내용 추가
  ingress:
    - ip: 192.0.2.127
```

외부 로드밸런서의 트래픽은 클러스터 내의 Pod로 전달된다.

클라우드 서비스를 제공하는 업체(AWS, GCP 등)는 로드밸런서의 동작을 결정한다.

일부 클라우드 서비스 제공 업체에서는 loadBalancerIP를 지정할 수 있다.

이 경우에는 지정된 loadBalancerIP로 로드밸런서가 생성된다.

loadBalancerIP가 지정되지 않은 로드밸런서는 임시 IP 주소로 설정된다.

만약 클라우드 서비스 제공 업체가 loadBalancerIP를 지원하지 않는다면, loadBalancerIP를 설정한 필드는 무시된다.

Ingress

Ingress란 리버스 프록시를 통해 클러스터 내부 Service로 어떻게 패킷을 포워딩 시킬 것인지 명시한 쿠버네티스 리소스이다.

Ingress는 Ingress Controller와 짹지어진다.

모든 Ingress Controller는 참조 사양에 맞아야 한다.

Ingress Controller는 다양하게 있으며, 대중적으로 많이 사용하는 Ingress Controller는 nginx-ingress 이다.

In the case of #ingress-nginx , the LoadBalancer sends requests to a Service in front of the #ingress-nginx controller pods on port 80/443, where an nginxd is listening. That nginxd has a giant nginx.conf full of reverse proxy rules that the #ingress-nginx controller has created from Ingress resources. The ingress-nginx controller is a go application that subscribes to Ingress resources and writes nginx.conf, plus an nginx daemon that reads nginx.conf and does reverse proxying of requests to other pods.

nginx ingress controller는 ingress 리소스를 읽어서 그에 맞는 리버스 프록시를 구성한다.

eks ingress는 ingress 리소스를 읽어서 그에 맞는 리버스 프록시를 구성하기 위해 Application Load Balancer 및 필수 지원 AWS 리소스가 만들어지도록 트리거하는 컨트롤러이다.

nginx-ingress Installation guide

Set up Nginx Ingress in bare-metal

```
# Hello world Server Pod
kind: Deployment
apiVersion: apps/v1
metadata:
  name: service-test
```

```

spec:
  replicas: 4
  selector:
    matchLabels:
      app: service_test_pod
  template:
    metadata:
      labels:
        app: service_test_pod
  spec:
    containers:
      - name: simple-http
        image: python:2.7
        imagePullPolicy: IfNotPresent
        command: ["/bin/bash"]
        args: ["-c", "echo \\\"<p>Hello from $(hostname)</p>\\\" > index.html; python -m SimpleHTTPServer 8080"]
      ports:
        - name: http
          containerPort: 8080
---
# Hello world Server Service
apiVersion: v1
kind: Service
metadata:
  name: service-test
spec:
  selector:
    app: service_test_pod
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
---
# Hello world Server Ingress
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
spec:
  rules:
    - host: nginx.example.com
      http:
        paths:
          - backend:
              serviceName: service-test
              servicePort: 80
---
# Hello world Server Ingress Service
apiVersion: v1
kind: Service
metadata:
  name: nginx-ingress
  namespace: nginx-ingress
spec:
  type: NodePort
  ports:
    - port: 80

```

```

targetPort: 80
protocol: TCP
name: http
- port: 443
  targetPort: 443
  protocol: TCP
  name: https
  selector:
    app: nginx-ingress
---  

      . . .

```

참고

Cluster Networking

쿠버네티스 네트워킹 : 포드 네트워킹(kubernetes pod networking)

[번역] 쿠버네티스 네트워킹 이해하기 #1: Pods

[번역] 쿠버네티스 네트워킹 이해하기 #2: Services

[번역] 쿠버네티스 네트워킹 이해하기 #3: Ingress

Kubernetes Ingress with Nginx Example

A Guide to the Kubernetes Networking Model

Kubernetes Service Proxy

<https://www.youtube.com/watch?v=chwofyGr80c>

Kubernetes Docker Networking

About Help Legal

Get the Medium app



