실습과 함께 완성해보는

도커 없이 컨테이너 만들기

4편

Sam.0

시작하기에 앞서 ...

본 컨텐츠는 앞편을 보았다고 가정하고 준비되었습니다.

원활한 이해 및 실습을 위하여 앞편을 먼저 보시기를 추천드립니다

특히 3편은 네트워크 네임스페이스 1부이므로 안보셨다면 꼭 보세요

:-)

1편 링크 클릭 *2편 링크 클릭* *3편 링크 클릭*

실습을 위한 사전 준비 사항

실습 환경 구성하기 클릭

실습은 . . .

- 맥 환경에서 VirtualBox + Vagrant 기반으로 준비되었습니다
  - 맥 이외의 OS 환경도 괜찮습니다만
  - 원활한 실습을 위해서
  - “VirtualBox or VMware + Vagrant”는 권장드립니다.
- 실습환경 구성을 위한 Vagrantfile을 제공합니다.

실습을 위한 사전 준비 사항

Vagrantfile

- 오른쪽의 텍스트를 복사하신 후
- 로컬에 Vagrantfile로 저장하여 사용하면 됩니다.
- vagrant 사용법은 공식문서를 참고해 주세요 https://www.vagrantup.com/docs/index

실습 환경 구성하기 클릭

```
BOX_IMAGE = "bento/ubuntu-18.04"
HOST_NAME = "ubuntu1804"

$pre_install = <<-SCRIPT
  echo ">>>> pre-install <<<<<<"
  sudo apt-get update &&
  sudo apt-get -y install gcc &&
  sudo apt-get -y install make &&
  sudo apt-get -y install pkg-config &&
  sudo apt-get -y install libseccomp-dev &&
  sudo apt-get -y install tree &&
  sudo apt-get -y install jq &&
  sudo apt-get -y install bridge-utils

  echo ">>>> install go <<<<<<"
  curl -O https://storage.googleapis.com/golang/go1.15.7.linux-amd64.tar.gz > /dev/null 2>&1 &&
  tar xf go1.15.7.linux-amd64.tar.gz &&
  sudo mv go /usr/local/ &&
  echo 'PATH=$PATH:/usr/local/go/bin' | tee /home/vagrant/.bash_profile

  echo ">>>>> install docker <<<<<<"
  sudo apt-get -y install apt-transport-https ca-certificates curl gnupg-agent software-properties-common > /dev/null 2>&1 &&
  sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add - &&
  sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" &&
  sudo apt-get update &&
  sudo apt-get -y install docker-ce docker-ce-cli containerd.io > /dev/null 2>&1
SCRIPT

Vagrant.configure("2") do |config|

 config.vm.define HOST_NAME do |subconfig|
   subconfig.vm.box = BOX_IMAGE
   subconfig.vm.hostname = HOST_NAME
   subconfig.vm.network :private_network, ip: "192.168.104.2"
   subconfig.vm.provider "virtualbox" do |v|
     v.memory = 1536
     v.cpus = 2
   end
   subconfig.vm.provision "shell", inline: $pre_install
 end

end
```

실습을 위한 사전 준비 사항

실습 환경 구성하기 클릭

Vagrantfile 제공 환경

vagrant + virtual vm
ubuntu 18.04
docker 20.10.5 ** 도커 이미지 다운로드 및 컨테이너 비교를 위한 용도로 사용합니다.*

기타 설치된 툴
~ tree, jq, brctl, … 등 실습을 위한 툴

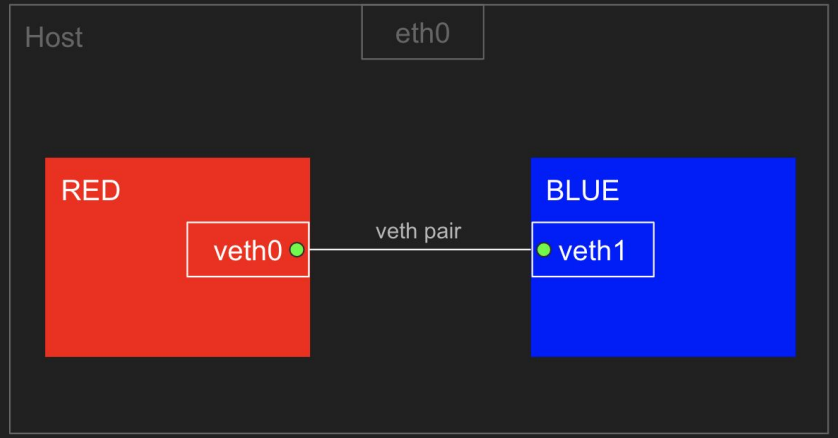
실습 계정 (root)

```
# sudo -Es
```

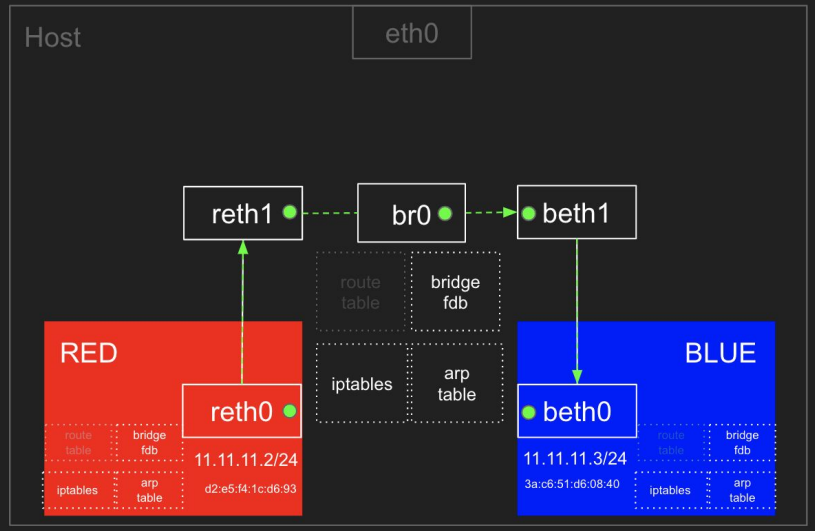
실습 폴더

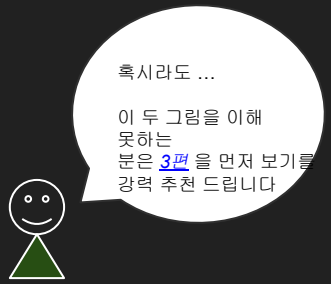
```
# cd /tmp
```

바로 지난 3편에서 우리는 ...



두 개의 그림을 그렸습니다



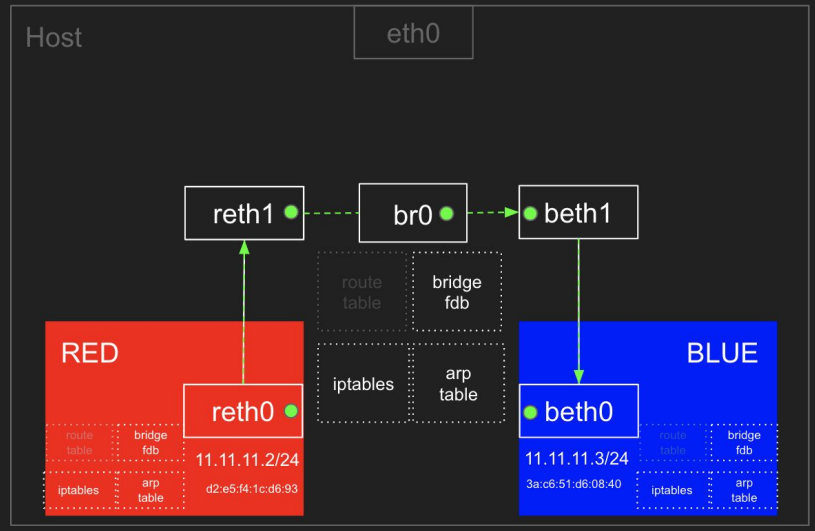


실습 준비를 해보겠습니다.

브릿지가 없으신 분은
스크립트를 참고하셔서
새로 생성해 주세요
코드링크도
첨부드립니다.
bridge.sh 코드 링크

```
# vi bridge.sh

#!/bin/bash

ip netns add RED;
ip link add reth0 netns RED type veth peer name reth1

ip netns add BLUE;
ip link add beth0 netns BLUE type veth peer name beth1

ip link add br0 type bridge
ip link set reth1 master br0
ip link set beth1 master br0

ip netns exec RED ip addr add 11.11.11.2/24 dev reth0
ip netns exec BLUE ip addr add 11.11.11.3/24 dev beth0

ip netns exec RED ip link set reth0 up;
ip link set reth1 up;
ip netns exec BLUE ip link set beth0 up;
ip link set beth1 up;
ip link set br0 up;

iptables -t filter -A FORWARD -s 11.11.11.0/24 -j ACCEPT
```

지난 시간에 만들었던 브릿지를 준비해 주세요



터미널 #1

```
# ./bridge.sh
```

실습 준비를 해보겠습니다.

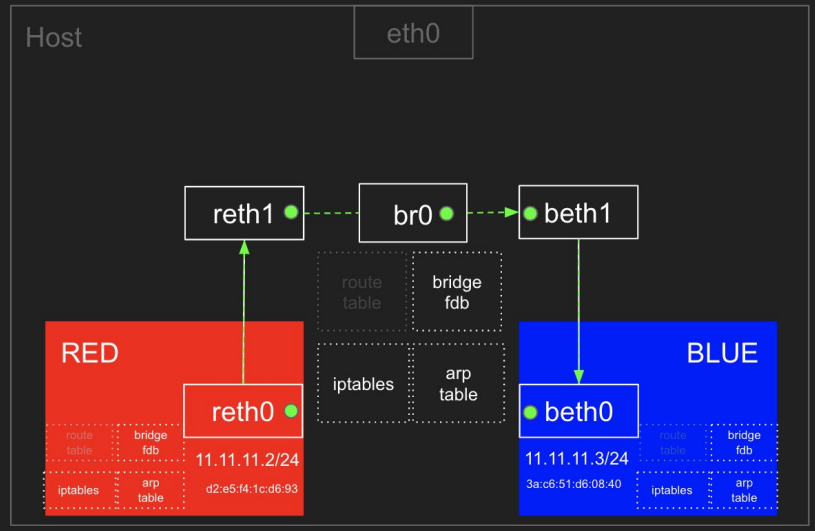
잘 동작하는지 확인해주세요

터미널 #1 (RED, 11.11.11.2)

```
# nsenter --net=/var/run/netns/RED
# ping -c 3 11.11.11.2
```

터미널 #3 (BLUE, 11.11.11.3)

```
# nsenter --net=/var/run/netns/BLUE
# ping -c 3 11.11.11.2
```

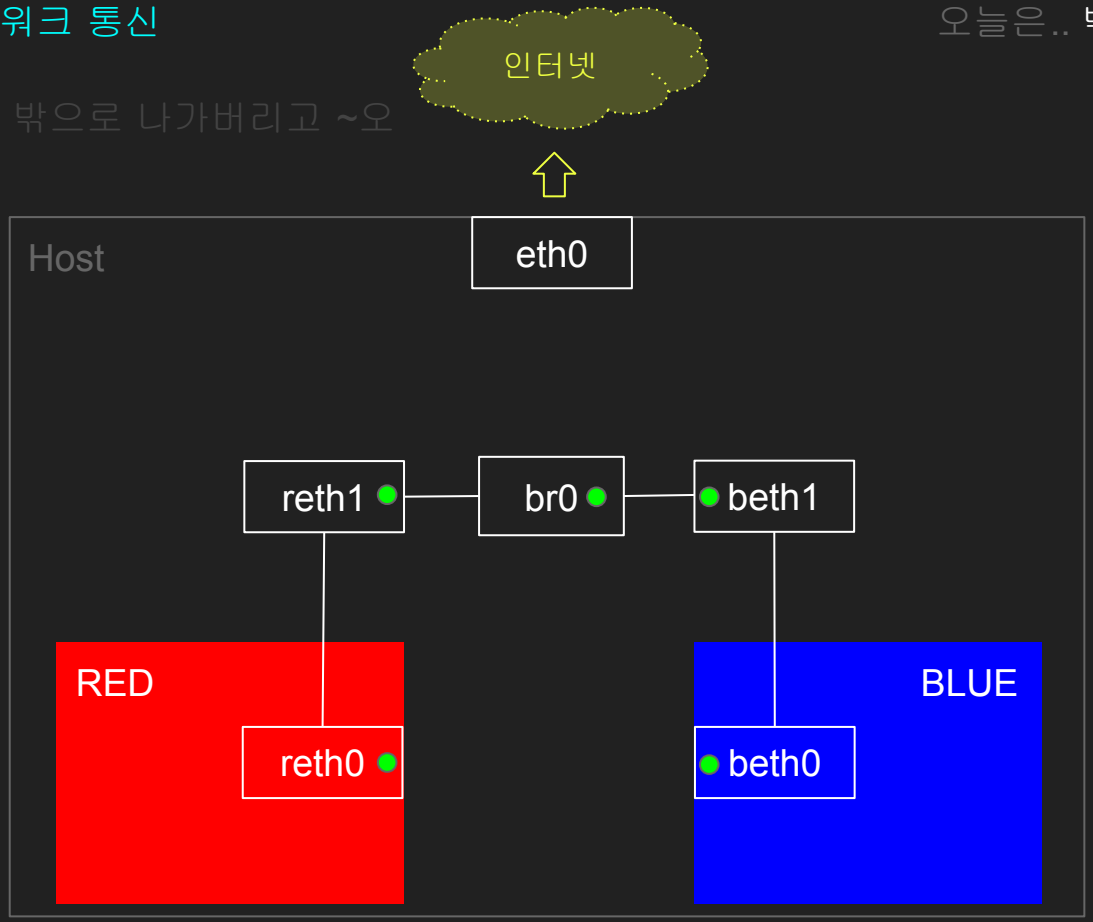


✸혹시 잘 동작안하시면 bridge_reset.sh 를 실행 후 다시 bridge.sh 해주세요

(실습3) 외부 네트워크 통신

오늘은.. 밖으로 나가보겠습니다

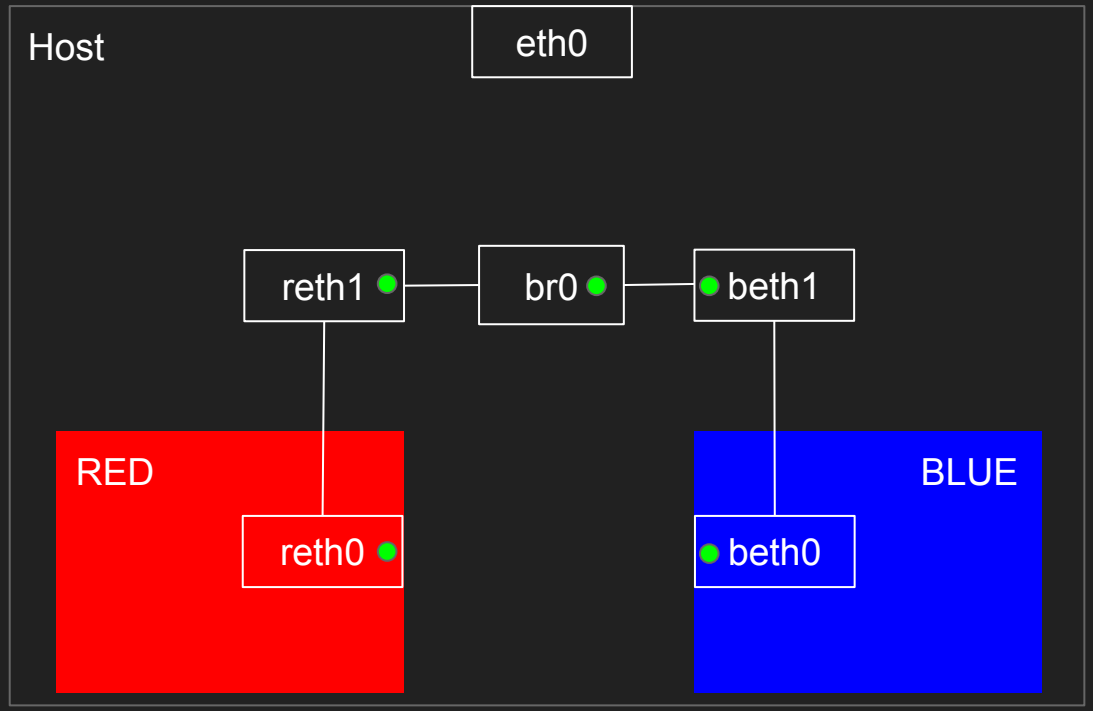
밖으로 나가버리고 ~오



(실습3) 외부 네트워크 통신

자.. 먼저 호스트와 핑을 주고 받아 봅시다

Host → RED



(실습3) 외부 네트워크 통신

PING 호스트 → RED (11.11.11.2)

터미널 #1 (RED, 11.11.11.2)

```
# tcpdump -i any
```

터미널 #2 (호스트)

```
# ping -c 1 11.11.11.2
```

터미널 #3 (호스트)

```
# tcpdump -i br0
```

(실습3) 외부 네트워크 통신

호스트 → RED (11.11.11.2) PING 실패 이유는?

터미널 #1 (RED, 11.11.11.2)

```
# tcpdump -i any
```

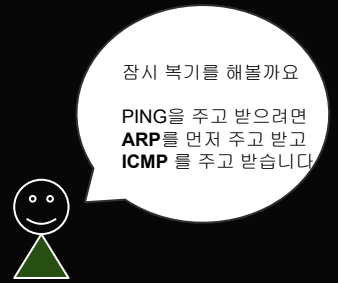
터미널 #2 (호스트)

```
# ping -c 1 11.11.11.2

--- 11.11.11.2 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

터미널 #3 (호스트)

```
# tcpdump -i br0
```

그런데 … ARP request 가 보이지 않습니다



(실습3) 외부 네트워크 통신

호스트 → RED (11.11.11.2) PING 실패 이유는?

ARP request 가 보이지 않는다는 것은 어떤 의미일까요 ?

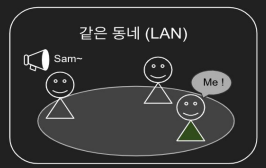
Hint) IP의 MAC정보를 알아내기 위해 ARP 브로드캐스팅을 합니다. (아래 3편 참고)

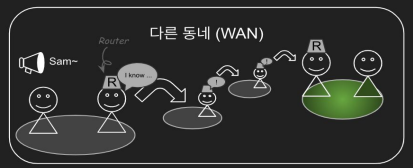
Network ?

3편 클릭

연결

- LAN : Local Area Network (브로드캐스트 도메인)
- WAN : Wide Area Network (라우터로 구분되는 네트워크)





ARP (Address Resolution Protocol)

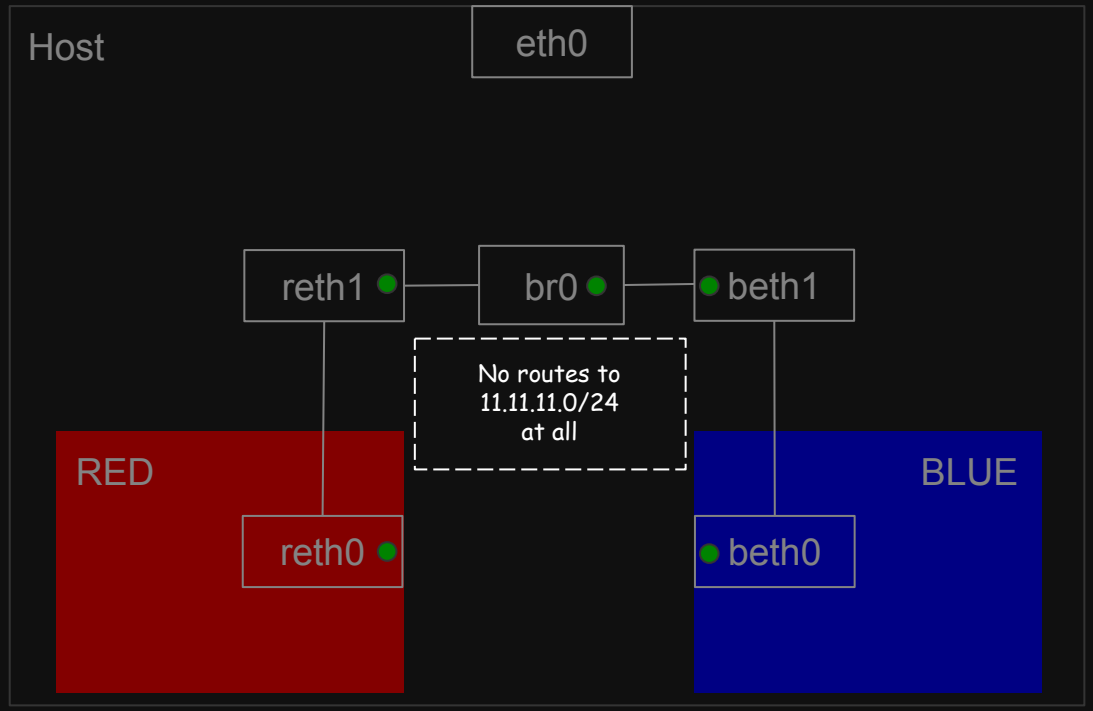
3편 클릭

- 주소(MAC) 를 물어보는 프로토콜 예) 11.11.11.3의 MAC 주소가 뭐에요?
- 브로드캐스트 (FF:FF:FF:FF:FF:FF)
- 라우터는 본인의 MAC주소를 알려줌 "나한테 물어봐 Sam이 어디사는지 알려줄께"

(실습3) 외부 네트워크 통신

호스트 → RED (11.11.11.2) PING 실패 이유는?

Host 네트워크와 RED 네트워크는 isolation 되어 있습니다. (즉, 다른 네트워크입니다)



(실습3) 외부 네트워크 통신

라우팅 테이블 확인

터미널 #1 (RED, 11.11.11.2)

```
#
```

터미널 #3 (BLUE, 11.11.11.3)

```
#
```

터미널 #2 (호스트)

```
# 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
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdown
192.168.104.0/24 dev eth1 proto kernel scope link src 192.168.104.2
```

11.11.11.* 로 가는
라우트 정보는 없음

라우팅 테이블 ?

패킷이 목적지, 목적지까지의 거리와 가는 방법 등을 명시한 테이블
IP address 를 이용하여 네트워크 경로를 파악

(실습3) 외부 네트워크 통신

br0 (bridge)에 연결할 네트워크(RED) 대역의 IP 주소 부여

터미널 #1 (RED, 11.11.11.2)

```
#
```

터미널 #2 (호스트)

```
# ip addr add 11.11.11.1/24 dev br0
```

터미널 #3 (BLUE, 11.11.11.3)

```
#
```

br0가 RED와 물리적인 연결(link)은 되어 있지만 호스트와 RED는 IP 대역이 다릅니다.
즉, HOST에서 11.11.11. 를 찾아가기 위해서는 br0가 인터페이스 역할을 해주어야 하는데요*
그러기 위해서는 호스트에서 연결할 네트워크 대역의 주소를 br0에 부여해야 합니다

(next hop)

(실습3) 외부 네트워크 통신

라우팅 테이블에 br0가 관리하는 네트워크 대역 정보 추가 (kernel)

터미널 #1 (RED, 11.11.11.2)

```
#
```

터미널 #3 (BLUE, 11.11.11.3)

```
#
```

터미널 #2 (호스트)

```
# 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
11.11.11.0/24 dev br0 proto kernel scope link src 11.11.11.1
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdown
192.168.104.0/24 dev eth1 proto kernel scope link src 192.168.104.2
```

br0에 연결할 대역(RED)의 IP주소를 부여하면
kernel은 11.11.11.0/24에 대한 라우트 정보를 새로 추가합니다.

Q) Router와 Bridge는 어떻게 다를까요?
브릿지는 목적지와 물리적인 연결(link)을 기반으로 Gateway 역할을 합니다.
라우터는 최종 목적지가 LAN의 범위를 넘어가는 경우로
패킷 전송 시 프레임의 destination (next hop)을 수정해서 물어물어 찾아가게 됩니다.

참고: 라우터와 게이트웨이의 차이

(실습3) 외부 네트워크 통신

호스트 → RED (11.11.11.2) PING ~ 성공

터미널 #1 (RED, 11.11.11.2)

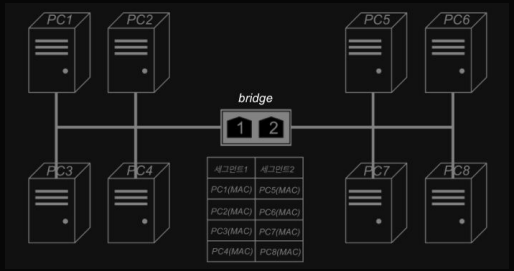
```
#
```

터미널 #3 (BLUE, 11.11.11.3)

```
#
```

터미널 #2 (호스트)

```
# ping -c 1 11.11.11.2

64 bytes from 11.11.11.2: icmp_seq=1 ttl=64 time=0.046 ms

# ping -c 1 11.11.11.3

64 bytes from 11.11.11.3: icmp_seq=1 ttl=64 time=0.046 ms
```



BLUE도 11.11.11.0/24 대역을 사용하고 br0와 연결(link)돼 있기 때문에 HOST → BLUE 통신이 가능합니다.

(실습3) 외부 네트워크 통신

PING RED/BLUE → 호스트(11.11.11.1, br0) ~ 성공

터미널 #1 (RED, 11.11.11.2)

```
# ping -c 1 11.11.11.1
64 bytes from 11.11.11.1: icmp_seq=1 ttl=64 time=0.046 ms
```

터미널 #2 (호스트, br0)

```
#
```

터미널 #3 (BLUE, 11.11.11.3)

```
# ping -c 1 11.11.11.1
64 bytes from 11.11.11.1: icmp_seq=1 ttl=64 time=0.046 ms
```

br0 (11.11.11.1) 는 RED/BLUE와 연결이 돼 있고, 같은 대역 (11.11.11.0)으로 통신이 됩니다

(실습3) 외부 네트워크 통신

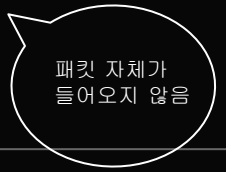
PING RED/BLUE → 호스트(192.168.104.2) ~ 실패

터미널 #1 (RED, 11.11.11.2)

```
# ping -c 192.168.104.2
connect: Network is unreachable
```

터미널 #2 (호스트, 192.168.104.2)

```
# tcpdump -l -i br0
```



터미널 #3 (BLUE, 11.11.11.3)

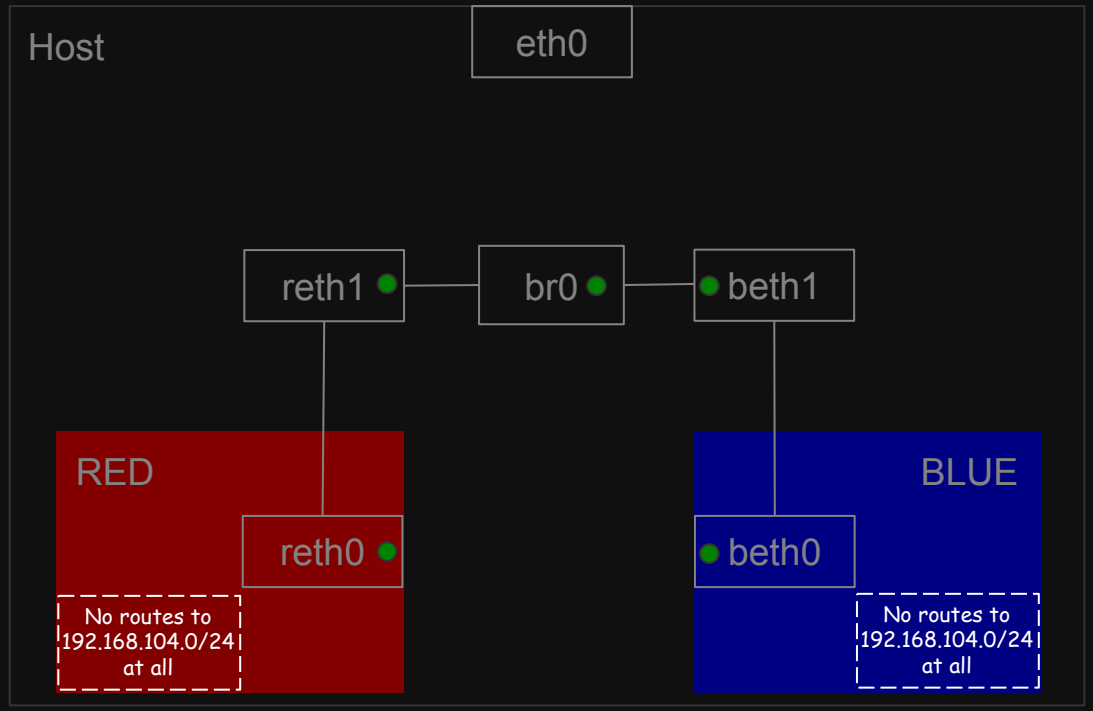
```
# ping -c 192.168.104.2
connect: Network is unreachable
```

호스트의 eth1(192.168.104.2)과의 통신은 왜 안될까요 ?

(실습3) 외부 네트워크 통신

PING RED/BLUE → 호스트(192.168.104.2) ~ 실패

네 … 아마도 다들 예상하셨을 것 같네요. 192.168.104.0 대역에 대한 route 정보가 없기 때문입니다



(실습3) 외부 네트워크 통신

PING RED/BLUE → 호스트(192.168.104.2) ~ 실패

터미널 #1 (RED, 11.11.11.2)

```
# ip route
11.11.11.0/24 dev reth0 proto kernel scope link src 11.11.11.2
```

192.168.104.* 로 가는 정보는 없음

터미널 #2 (호스트, 192.168.104.2)

```
# tcpdump -l -i br0
```

터미널 #3 (BLUE, 11.11.11.3)

```
# ip route
11.11.11.0/24 dev beth0 proto kernel scope link src 11.11.11.3
```

192.168.104.* 로 가는 정보는 없음

default route 설정

터미널 #1 (RED, 11.11.11.2)

```
# ip route add default via 11.11.11.1

# ip route

default via 11.11.11.1 dev reth0
11.11.11.0/24 dev reth0 proto kernel scope link src 11.11.11.2
```

터미널 #2 (호스트, 192.168.104.2)

```
# tcpdump -l -i br0
```

터미널 #3 (BLUE, 11.11.11.3)

```
# ip route add default via 11.11.11.1

# ip route

default via 11.11.11.1 dev beth0
11.11.11.0/24 dev beth0 proto kernel scope link src 11.11.11.3
```

default route

- *목적지 경로(대역)가 라우팅 테이블에 없는 경우 기본으로 선택되는 경로*
- *모든 경로가 장애 시에 선택되는 경로*
- *외부 네트워크로 나가는 통로*

(실습3) 외부 네트워크 통신

PING RED/BLUE → 호스트(192.168.104.2) ~ 성공

터미널 #1 (RED, 11.11.11.2)

```
# ping -c 192.168.104.2

64 bytes from 192.168.104.2: icmp_seq=1 ttl=64 time=0.043 ms
```

터미널 #2 (호스트, 192.168.104.2)

```
# tcpdump -l -i br0

10:52:35.688473 IP 11.11.11.2 > ubuntu1804: ICMP echo request, id 439, seq 1, length 64
10:52:35.688502 IP ubuntu1804 > 11.11.11.2: ICMP echo reply, id 439, seq 1, length 64
10:52:40.850680 ARP, Request who-has 11.11.11.2 tell ubuntu1804, length 28
10:52:40.850710 ARP, Request who-has ubuntu1804 tell 11.11.11.2, length 28
10:52:40.850925 ARP, Reply ubuntu1804 is-at de:d9:26:dc:ab:b5 (oui Unknown), length 28
10:52:40.850916 ARP, Reply 11.11.11.2 is-at 46:e6:04:4d:64:32 (oui Unknown), length 28
10:52:41.880481 IP 11.11.11.3 > ubuntu1804: ICMP echo request, id 440, seq 1, length 64
10:52:41.880511 IP ubuntu1804 > 11.11.11.3: ICMP echo reply, id 440, seq 1, length 64
10:52:46.990797 ARP, Request who-has ubuntu1804 tell 11.11.11.3, length 28
10:52:46.990838 ARP, Reply ubuntu1804 is-at de:d9:26:dc:ab:b5 (oui Unknown), length 28
```

터미널 #3 (BLUE, 11.11.11.3)

```
# ping -c 192.168.104.2

64 bytes from 192.168.104.2: icmp_seq=1 ttl=64 time=0.043 ms
```

RED/BLUE에 default route로 11.11.11.1 (br0)를 설정하였으므로
ip route에 명시되지 않은 대역들은 모두 11.11.11.1 (br0)로 보냅니다

(실습3) 외부 네트워크 통신

RED/BLUE → 인터넷 (8.8.8.8)

터미널 #1 (RED, 11.11.11.2)

```
# ping -c 1 8.8.8.8
```

터미널 #2 (호스트, 192.168.104.2)

```
# tcpdump -l -i br0
```

터미널 #3 (BLUE, 11.11.11.3)

```
# ping -c 1 8.8.8.8
```

(실습3) 외부 네트워크 통신

RED/BLUE → 인터넷 (8.8.8.8) X

터미널 #1 (RED, 11.11.11.2)

```
# ping -c 1 8.8.8.8

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

터미널 #3 (BLUE, 11.11.11.3)

```
# ping -c 1 8.8.8.8

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

터미널 #2 (호스트, 192.168.104.2)

```
# tcpdump -l -i br0

02:36:52.8475 ARP, Request who-has ubuntu1804 tell 11.11.11.2, length 28
02:36:52.8475 ARP, Reply ubuntu1804 is-at aa:83:07:b4:7d:96 (oui Unknown), length 28
02:36:53.7431 IP 11.11.11.2 > dns.google: ICMP echo request, id 1818, seq 45, length 64
02:36:54.7708 IP 11.11.11.2 > dns.google: ICMP echo request, id 1818, seq 46, length 64
02:36:55.7914 IP 11.11.11.2 > dns.google: ICMP echo request, id 1818, seq 47, length 64
```

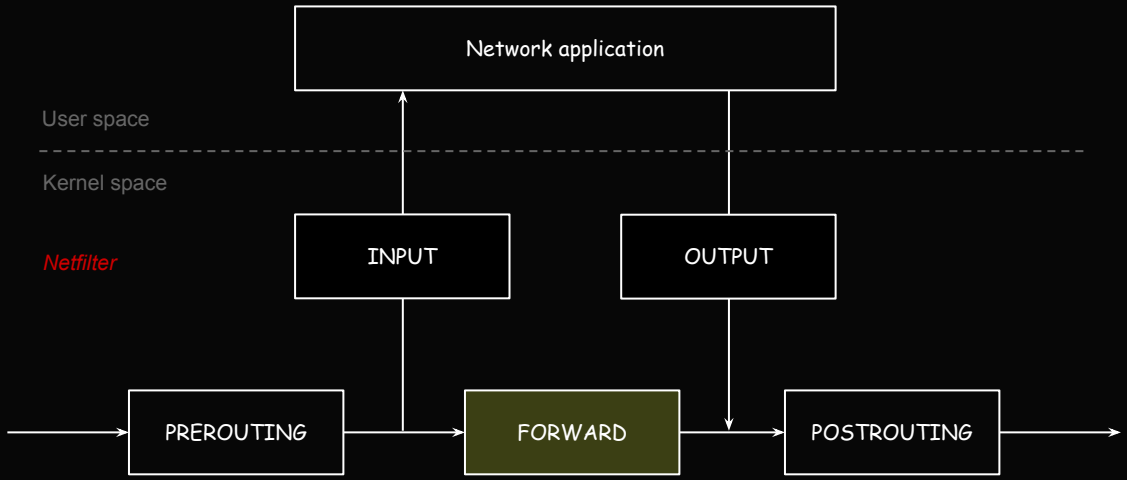
```
# ping -c 1 8.8.8.8

64 bytes from 8.8.8.8: icmp_seq=1 ttl=63 time=40.2 ms
```

- *호스트 → 8.8.8.8 OK*
- *RED/BLUE → 8.8.8.8 FAILED Why?*

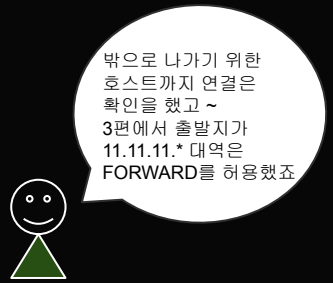
(실습3) 외부 네트워크 통신

RED/BLUE → 인터넷 (8.8.8.8) X *Why?*

iptables 확인

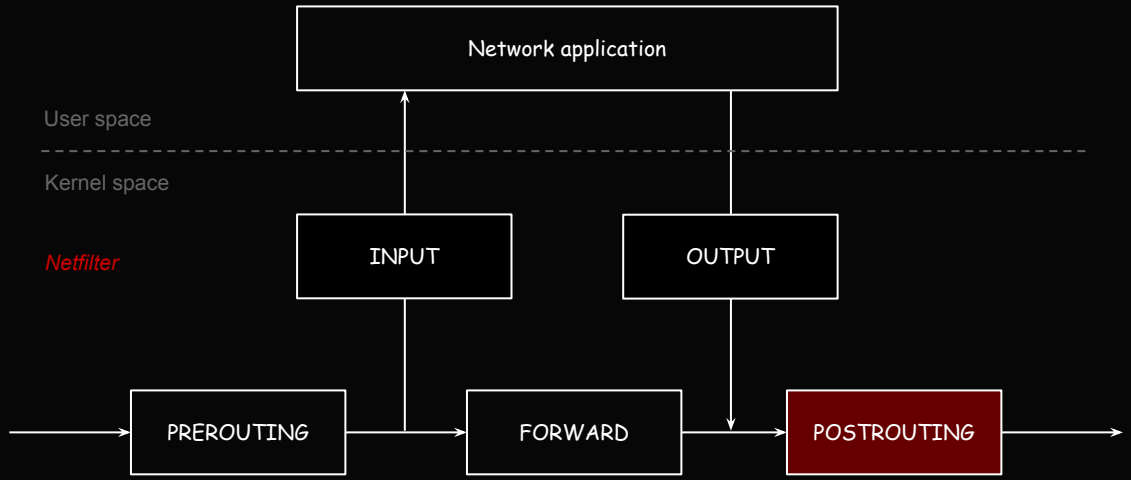


어딜까요 이번엔?



RED/BLUE → 인터넷 (8.8.8.8) X *Why?*

iptables 확인



다음 차례는 . . .

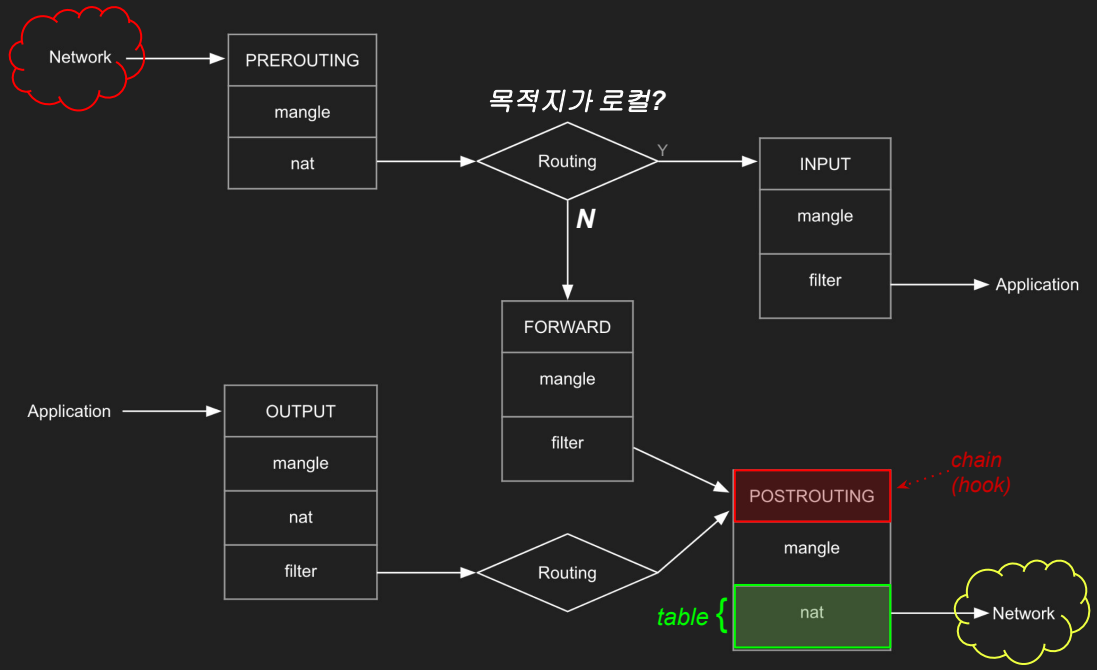
밖으로 나가버리고
~오

POSTROUTING : 라우팅 Outbount or 포워딩 트래픽에 의해 트리거되는 netfilter hook

(실습3) 외부 네트워크 통신

RED/BLUE → 인터넷 (8.8.8.8) X *Why?*

iptables 확인



POSTROUTING

nat (table) : NAT 대상 패킷의 출발지(src) 혹은 목적지(dest.) address 수정 방법을 결정

→ POSTROUTING 에서는 *SNAT (Source NAT)*

* 패킷이 network으로 direct access가 불가능한 경우에 주로 사용

(실습3) 외부 네트워크 통신

RED/BLUE → 인터넷 (8.8.8.8) X *Why?*

터미널 #2 (호스트, 192.168.104.2)

```
# iptables -t nat -A POSTROUTING -s 11.11.11.0/24 -j MASQUERADE
```

table / APPEND chain rule / src. address / jump to MASQUERADE

```
# iptables -t nat -S
```

MASQUERADE (가면)

(실습3) 외부 네트워크 통신

RED/BLUE → 인터넷 (8.8.8.8) X . . . 안돼 Why ?

터미널 #1 (RED, 11.11.11.2)

```
# ping -c 1 8.8.8.8

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

터미널 #3 (BLUE, 11.11.11.3)

```
# ping -c 1 8.8.8.8

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

터미널 #2 (호스트, 192.168.104.2)

```
# tcpdump -l -i br0

02:36:52.8475 ARP, Request who-has ubuntu1804 tell 11.11.11.2, length 28
02:36:52.8475 ARP, Reply ubuntu1804 is-at aa:83:07:b4:7d:96 (oui Unknown), length 28
02:36:53.7431 IP 11.11.11.2 > dns.google: ICMP echo request, id 1818, seq 45, length 64
02:36:54.7708 IP 11.11.11.2 > dns.google: ICMP echo request, id 1818, seq 46, length 64
02:36:55.7914 IP 11.11.11.2 > dns.google: ICMP echo request, id 1818, seq 47, length 64
```

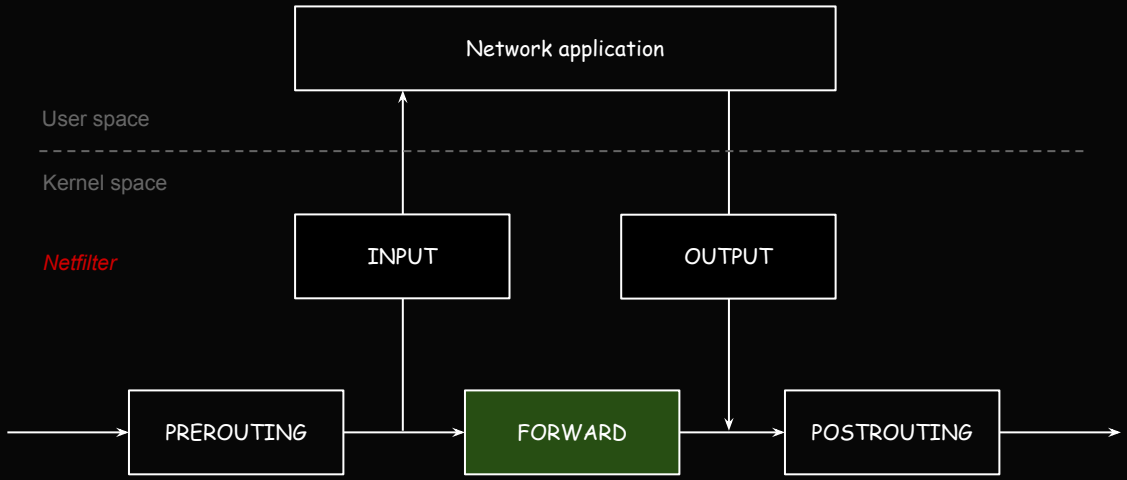
```
# ping -c 1 8.8.8.8

64 bytes from 8.8.8.8: icmp_seq=1 ttl=63 time=40.2 ms
```

- *호스트 → 8.8.8.8 OK*
- *RED/BLUE → 8.8.8.8 FAILED Why?*

(실습3) 외부 네트워크 통신

RED/BLUE → 인터넷 (8.8.8.8) X *Why?*

iptables 확인



어딜까요 이번엔?

(실습3) 외부 네트워크 통신

RED/BLUE → 인터넷 (8.8.8.8) X *Why?*

iptables 확인 ~ 외부 패킷의 HOST 통과여부는 FORWARD 체인룰로 관리합니다

(실습2) Bridge 통신

RED → BLUE ping failed why?

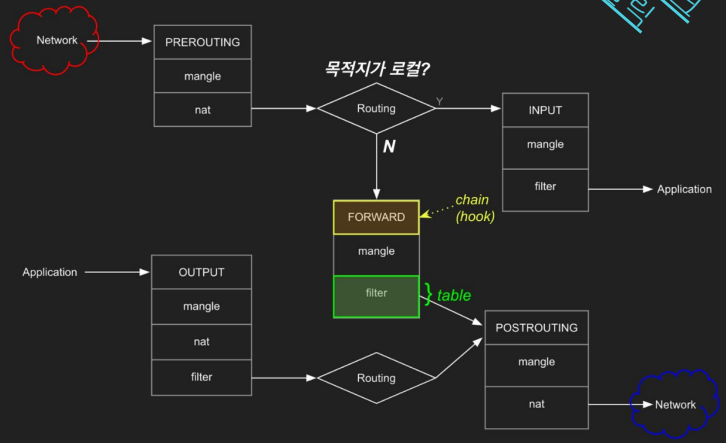
RED → (HOST) → BLUE HOST 입장에서는 …

"외부(src) → 외부(dst)" 패킷이므로

FORWARD 체인의 filter 테이블 룰을

봐야 합니다

RED → BLUE 통신은 Bridge를 경유하는데
Bridge가 호스트 네임스페이스에 있기때문에
패킷은 외부(RED 네임스페이스)에서 왔고
목적지 또한 외부(BLUE 네임스페이스)이므로
(로컬)호스트를 경유, 즉 FORWARD로 처리됩니다

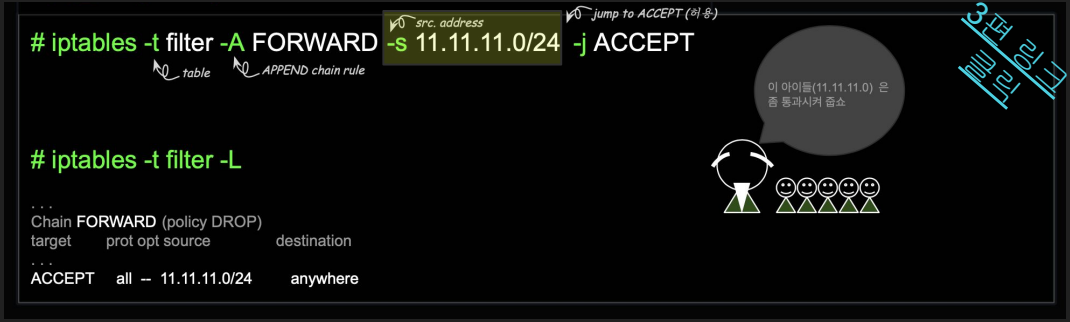


FOWARD : NF_IP_FORWARD (hook)에 등록된 체인
~ NF_IP_FORWARD : incoming 패킷이 다른 호스트로 포워딩되는 경우에 트리거되는 netfilter hook
filter (table) : 패킷을 목적지로 전송 여부를 결정

(실습3) 외부 네트워크 통신

RED/BLUE → 인터넷 (8.8.8.8) X *Why?*

그래서.. 3편에서 FORWARD 체인룰을 등록해 주었는데요 “src. address” 만 허용을 해주었습니다



(실습3) 외부 네트워크 통신

RED/BLUE → 인터넷 (8.8.8.8) X *Why?*

요청을 보낼 때 (request)

RED → 8.8.8.8
출발지(src) : 11.11.11.2 ~ 외부
목적지(dst.) : 8.8.8.8 ~ 외부

PASSED


```
root@ubuntu1804:~# iptables -t filter -S | grep 11.11.11.0
-A FORWARD -s 11.11.11.0/24 -j ACCEPT
```

(실습3) 외부 네트워크 통신

RED/BLUE → 인터넷 (8.8.8.8) X *Why?*

응답을 받을 때 (Response)

8.8.8.8 → RED
출발지(src) : 8.8.8.8 ~ 외부
목적지(dst.) : 11.11.11.2 ~ 외부

DENIED

8.8.8.8은 FORWARD룰에 허용된 대역이 아니기 때문에 거부 당합니다

(실습3) 외부 네트워크 통신

RED/BLUE → 인터넷 (8.8.8.8) X *Why?*

터미널 #2 (호스트)

```
# iptables -t filter -A FORWARD -d 11.11.11.0/24 -j ACCEPT
```

table / *APPEND chain rule* / *dest. address* / *jump to ACCEPT (허용)*

```
# iptables -t filter -L

. . .
Chain FORWARD (policy DROP)
target     prot opt source               destination
. . .
ACCEPT     all  --  11.11.11.0/24        anywhere
ACCEPT     all  --  anywhere             11.11.11.0/24
```

추가된 룰

11.11.11. 을 목적지로 하는 패킷에 대하여도 **FORWARD**를 허락해줍니다. (일종의 입국 심사네요)*

(실습3) 외부 네트워크 통신

RED/BLUE → 인터넷 (8.8.8.8) OK

터미널 #1 (RED, 11.11.11.2)

```
# ping -c 1 8.8.8.8

64 bytes from 8.8.8.8: icmp_seq=1 ttl=61 time=47.7 ms
```

터미널 #3 (BLUE, 11.11.11.3)

```
# ping -c 1 8.8.8.8

64 bytes from 8.8.8.8: icmp_seq=1 ttl=61 time=47.7 ms
```

터미널 #2 (호스트, 192.168.104.2)

```
# tcpdump -l -i br0

08:16:31.1998 IP 11.11.11.2 > dns.google: ICMP echo request, id 2200, seq 1, length 64
08:16:31.2406 IP dns.google > 11.11.11.2: ICMP echo reply, id 2200, seq 1, length 64

# ping -c 1 8.8.8.8

64 bytes from 8.8.8.8: icmp_seq=1 ttl=63 time=40.2 ms
```

혹시... 여전히 안되시는 분?

커널 설정을 확인해주세요 ip_forward 설정이 0 이면 1로 바꿔 주세요

커널의 IP Forwarding (routing) 기능 확인

- 0 - off
- 1 - on

https://www.joinc.co.kr/w/man/12/ipforwarding

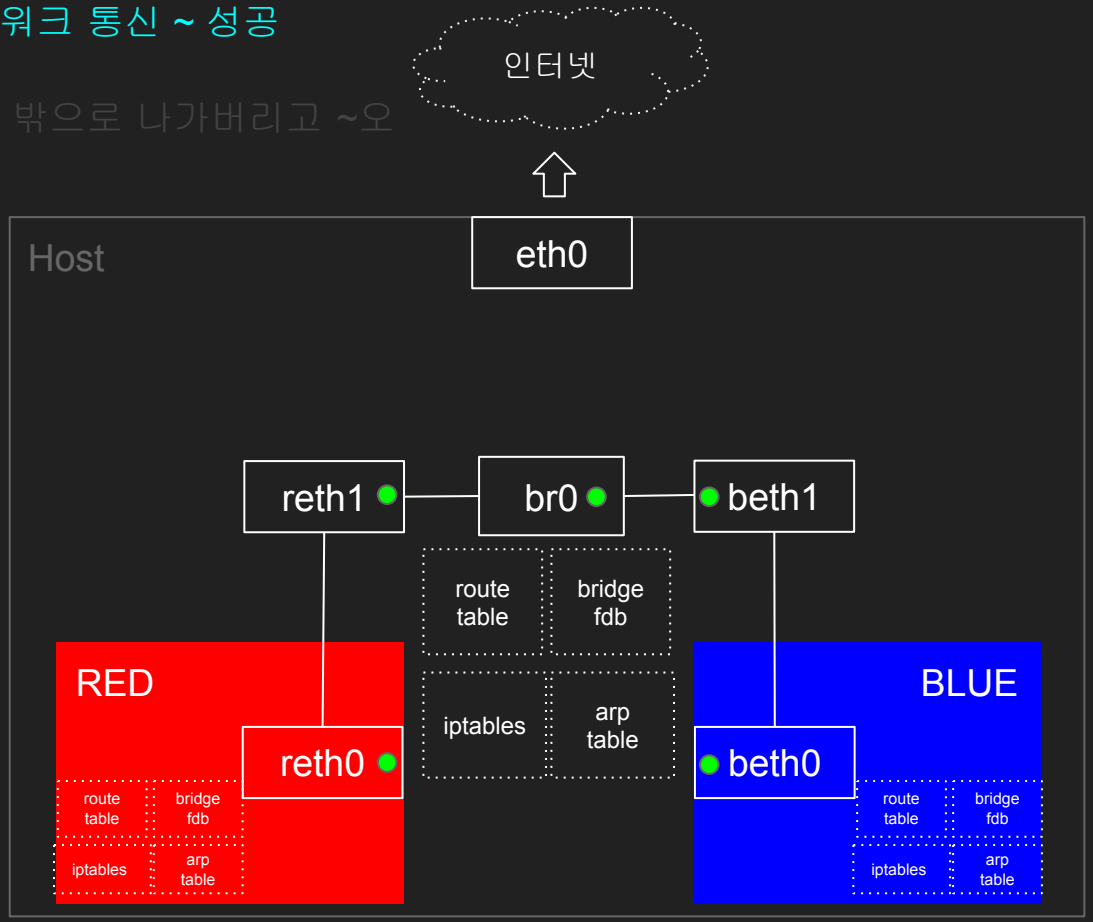
터미널 #2 (호스트)

```
# cat /proc/sys/net/ipv4/ip_forward
1
```

리눅스 IP Forwarding 참고: https://www.joinc.co.kr/w/man/12/ipforwarding

(실습3) 외부 네트워크 통신 ~ 성공

밖으로 나가버리고 ~오

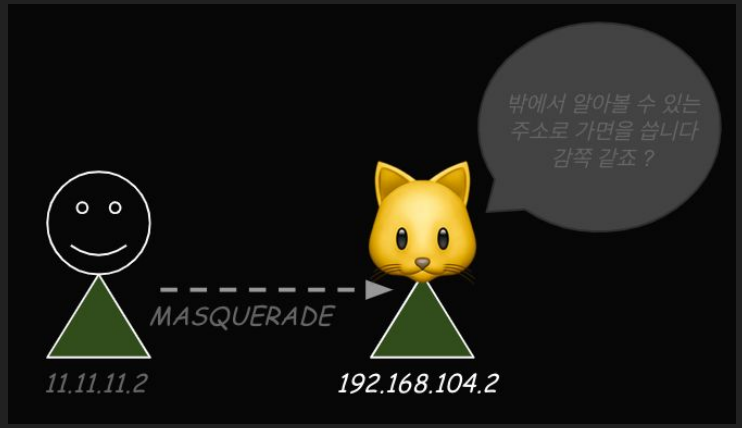


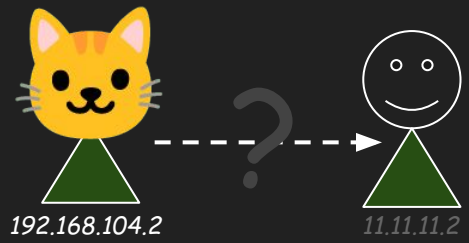
(Homework) 그런데 말입니다 ...

이 장면 기억 나시나요?

밖으로 나가기 위해서 가면(masquerade)을 썼는데 …

8.8.8.8로 부터 온 응답 패킷은 11.11.11.2를 어떻게 찾아 왔을까요?





답은.. 오버레이 네트워크 편에서 공개하겠습니다.

(실습4) 도커 네트워크

지금까지 네트워크 네임스페이스에 대해서 알아 보았습니다.

도커에서는 컨테이너 네트워크가 어떻게 돼 있는지 궁금하시지 않나요?

공부한 내용을 바탕으로 도커 네트워크를 들여다 보겠습니다 ~ ㄱ ㄱ

터미널 #2 (HOST)

```
# docker network list
NETWORK ID     NAME      DRIVER    SCOPE
35e64e36902e   bridge    bridge    local   (기본값)
e970e333e3ac   host      host      local   호스트 네트워크 사용
e741d1f667b5   none      null      local   모든 네트워크 비활성
```

ethernet bridge (like br0)

docker에서 built-in으로 지원하는 network 목록입니다

https://docs.docker.com/network/

도커 브릿지 : docker0

터미널 #2 (HOST)

```
# docker inspect bridge
```

docker 의 bridge 네트워크 정보를 조회합니다

bridge name : docker0 (default bridge)
Subnet 대역 : 172.17.0.0/16
. . .


```
[
    {
        "Name": "bridge",
        "Id": "35e64e36902e74b156812db5f926252386246fbc62b8dea14b7155a30f3d456e",
        "Created": "2021-04-30T01:45:24.188664171Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
    }
]
```

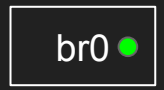
도커 브릿지 : docker0

터미널 #2 (HOST)

```
# ip addr show
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue
state DOWN group default
    link/ether 02:42:e7:83:d2:54 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
```

앞에서 만든 br0 와 비교해 보세요 → docker0가 DOWN 상태인 것 말고는 비슷합니다

```
14: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default qlen 1000
    link/ether 9e:54:88:22:36:09 brd ff:ff:ff:ff:ff:ff
    inet 11.11.11.1/24 scope global br0
       valid_lft forever preferred_lft forever
```



도커 브릿지 : docker0

터미널 #2 (HOST)

```
# brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.0242e783d254       no
```

아래 br0와 달리 docker0 는 아직 연결된 인터페이스가 없군요

※ brctl - ethernet bridge administration

** 지금은 deprecated 되어 별도 설치해야 합니다.*

(실습4) 도커 네트워크

도커 컨테이너(busybox)를 2개 띄워봅시다

터미널 #1 (PINK)

```
# docker run -it --name=PINK --rm busybox
```

터미널 #3 (ORANGE)

```
# docker run -it --name=ORANGE --rm busybox
```

docker run (== docker container run)

-it : interactive + tty *(터미널로 바로 연결)*

--name : *컨테이너 이름*

--rm : *종료 시 컨테이너 삭제*

(실습4) 도커 네트워크

docker ps 명령으로 확인해 보세요

터미널 #2 (HOST)

```
# docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED          STATUS          PORTS     NAMES
357248f27702   busybox   "sh"      14 seconds ago   Up 13 seconds             ORANGE
7f377447d03d   busybox   "sh"      2 minutes ago    Up 2 minutes              PINK
```

docker ps (== docker container list)

컨테이너 리스트 출력

(실습4) 도커 네트워크

호스트의 ip 정보에 새로운 인터페이스가 추가되었네요

터미널 #2 (HOST)

```
# ip a

4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
noqueue state UP group default
    link/ether 02:42:86:2d:c3:29 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
...
112: veth28d0aef@if111: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu
noqueue master docker0 ...
    link/ether 82:26:a0:d1:35:90 brd ff:ff:ff:ff:ff:ff link-netnsid 0
...
114: veth636a01f@if113: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu
noqueue master docker0 ...
    link/ether 6a:42:66:ca:6b:ef brd ff:ff:ff:ff:ff:ff link-netnsid 1
...
```

*어딘가로 연결되는 **veth peer**가 추가됐네요*

3편에서 다룬 veth pair를 떠올려 보세요

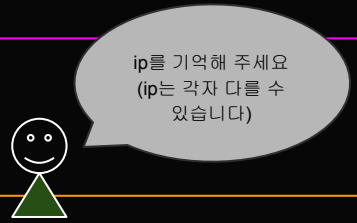
(실습4) 도커 네트워크

호스트의 ip 정보에 새로운 인터페이스가 추가되었네요

veth peer 는 각각 PINK와 ORANGE의 eth0와 연결됩니다

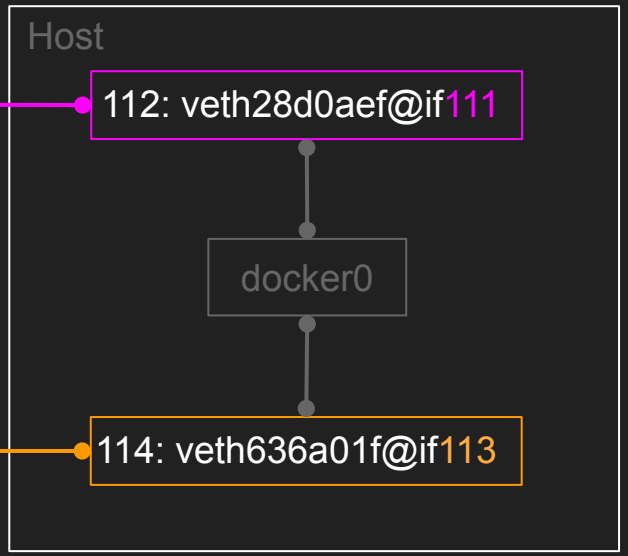
터미널 #1 (PINK)

```
# ip a
111: eth0@if112: <BROADCAST,MULTICAST,UP,LOWER_UP  ...
link/ether 02:42:ac:11:00:03 …
inet 172.17.0.2/16 ...
```



터미널 #3 (ORANGE)

```
# ip a
113: eth0@if114: <BROADCAST,MULTICAST,UP,LOWER_UP  ...
link/ether 02:42:ac:11:00:02 …
inet 172.17.0.3/16 ...
```



3편에서 다룬 veth pair를 떠올려 보세요

docker0 bridge의 연결정보를 확인해 봅시다

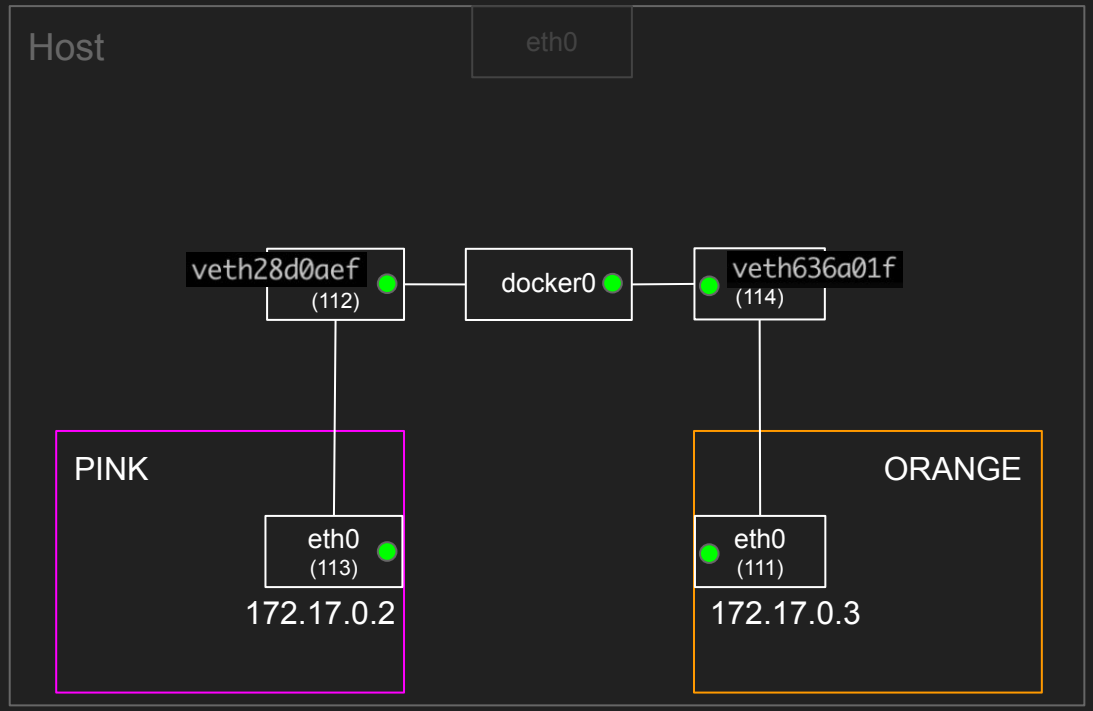
터미널 #2 (HOST)

```
# brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.0242862dc329       no              veth28d0aef
                                                        veth636a01f
```

호스트의 ip 정보

```
...
112: veth28d0aef@if111: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu noqueue master docker0 ...
    link/ether 82:26:a0:d1:35:90 brd ff:ff:ff:ff:ff:ff link-netnsid 0
...
114: veth636a01f@if113: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu noqueue master docker0 ...
    link/ether 6a:42:66:ca:6b:ef brd ff:ff:ff:ff:ff:ff link-netnsid 1
```

지금까지 파악된 내용을 그려봅시다



arp table을 확인해 볼까요

터미널 #1 (PINK)

```
# ip neigh show
```

아무 것도 출력되지 않습니다

터미널 #3 (ORANGE)

```
# ip neigh show
```

(실습4) 도커 네트워크

PINK → ORANGE PING을 보내보겠습니다

터미널 #1 (PINK, 172.17.0.2)

```
# ping -c 1 172.17.0.3
```

터미널 #2 (HOST)

```
# tcpdump -i docker0
```

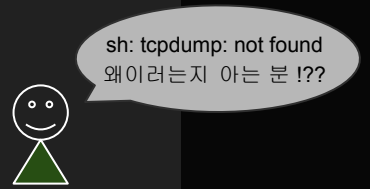
터미널 #3 (ORANGE, 172.17.0.3)

```
# tcpdump -i any
sh: tcpdump: not found
```

Hint) 1,2편을 복기해 보세요

정리해보면 ...

> "Containers are processes" ,
born from tarballs,
anchored to namespaces,
controlled by cgroups"



출처: https://twitter.com/jpetazzo/status/1047179436959956992

(실습4) 도커 네트워크

PINK → ORANGE PING을 보내보겠습니다

터미널 #1 (PINK, 172.17.0.2)

```
# ping -c 1 172.17.0.3
PING 172.17.0.3 (172.17.0.3): 56 data bytes
64 bytes from 172.17.0.3: seq=0 ttl=64 time=0.081 ms
64 bytes from 172.17.0.3: seq=1 ttl=64 time=0.057 ms
64 bytes from 172.17.0.3: seq=2 ttl=64 time=0.120 ms

--- 172.17.0.3 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.057/0.086/0.120 ms
```

PING이 잘 전달되네요

터미널 #2 (HOST)

```
# tcpdump -i docker0
04:42:41.793466 ARP, Request who-has 172.17.0.3 tell 172.17.0.2, length 28
04:42:41.793492 ARP, Reply 172.17.0.3 is-at 02:42:ac:11:00:03 (oui Unknown), length 28
04:42:41.793498 IP 172.17.0.2 > 172.17.0.3: ICMP echo request, id 2048, seq 0, length
64
04:42:41.793520 IP 172.17.0.3 > 172.17.0.2: ICMP echo reply, id 2048, seq 0, length 64
04:42:46.933050 ARP, Request who-has 172.17.0.2 tell 172.17.0.3, length 28
04:42:46.933136 ARP, Reply 172.17.0.2 is-at 02:42:ac:11:00:02 (oui Unknown), length 28
```

※ docker0 (bridge)를 경유하여 ARP/ICMP 성공

arp table을 다시 확인해 볼까요

터미널 #1 (PINK)

```
# ip neigh show
172.17.0.3 dev eth0 lladdr 02:42:ac:11:00:03 ref 1 used 0/0/0 probes 4
REACHABLE
```

arp table에 이웃(neighbour)의 정보가 캐싱 되었네요

터미널 #3 (ORANGE)

```
# ip neigh show

172.17.0.2 dev eth0 lladdr 02:42:ac:11:00:02 ref 1 used 0/0/0 probes 1
REACHABLE
```

route 테이블을 확인해 봅시다

터미널 #1 (PINK)

```
# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         172.17.0.1      0.0.0.0         UG    0      0        0 eth0
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 eth0
```

터미널 #3 (ORANGE)

```
# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         172.17.0.1      0.0.0.0         UG    0      0        0 eth0
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 eth0
```

Default route 가 이미 등록이 돼 있네요

172.17.0.1 (docker0)이 default gateway로

eth0 Iface를 통해서 연결됩니다

즉, PINK/ORANGE → 호스트 통신이

가능합니다

route 테이블을 확인해 봅시다

호스트 route 테이블을 봅시다

172.17.0.0 대역이 등록이 돼 있고

docker0가 인터페이스로 잡혀 있습니다

즉, HOST → PINK/ORANGE 통신이 가능합니다

터미널 #2 (HOST)

```
# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.0.2.2        0.0.0.0         UG    100    0        0 eth0
10.0.2.0        0.0.0.0         255.255.255.0   U     0      0        0 eth0
10.0.2.2        0.0.0.0         255.255.255.255 UH    100    0        0 eth0
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 docker0
192.168.104.0   0.0.0.0         255.255.255.0   U     0      0        0 eth1
```

iptables를 확인해 봅시다

터미널 #2 (HOST)

```
# iptables -t filter -S

-P INPUT ACCEPT
-P FORWARD DROP
-P OUTPUT ACCEPT
. . .
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
. . .
```

-P policy 기본정책

** 앞서 살펴봤듯이 FORWARD룰은 기본이 DROP 입니다*

-i : in-interface
-o: out-interface
docker0를 통과하는 FORWARD를 모두 허용합니다

When the "!" argument is used before the interface name, the sense is inverted

도커 컨테이너(네임스페이스) 간, 외부 → 컨테이너
통신이 가능합니다

iptables를 확인해 봅시다

터미널 #2 (HOST)

```
# iptables -t nat -S
-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
. . .
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE
. . .
```

-P policy 기본정책 (모두 ACCEPT)

-i : in-interface

-o: out-interface

172.17.0.0 대역이고 out-interface가 docker0 아닌 경우 POSTROUTING을 허용하고 masquerade 처리합니다 (SNAT)

도커 컨테이너(네임스페이스) → 외부 (8.8.8.8) 통신이 가능합니다

iptables를 확인해 봅시다

터미널 #1 (PINK)

```
# iptables -t filter -S

# iptables -t nat -S
```

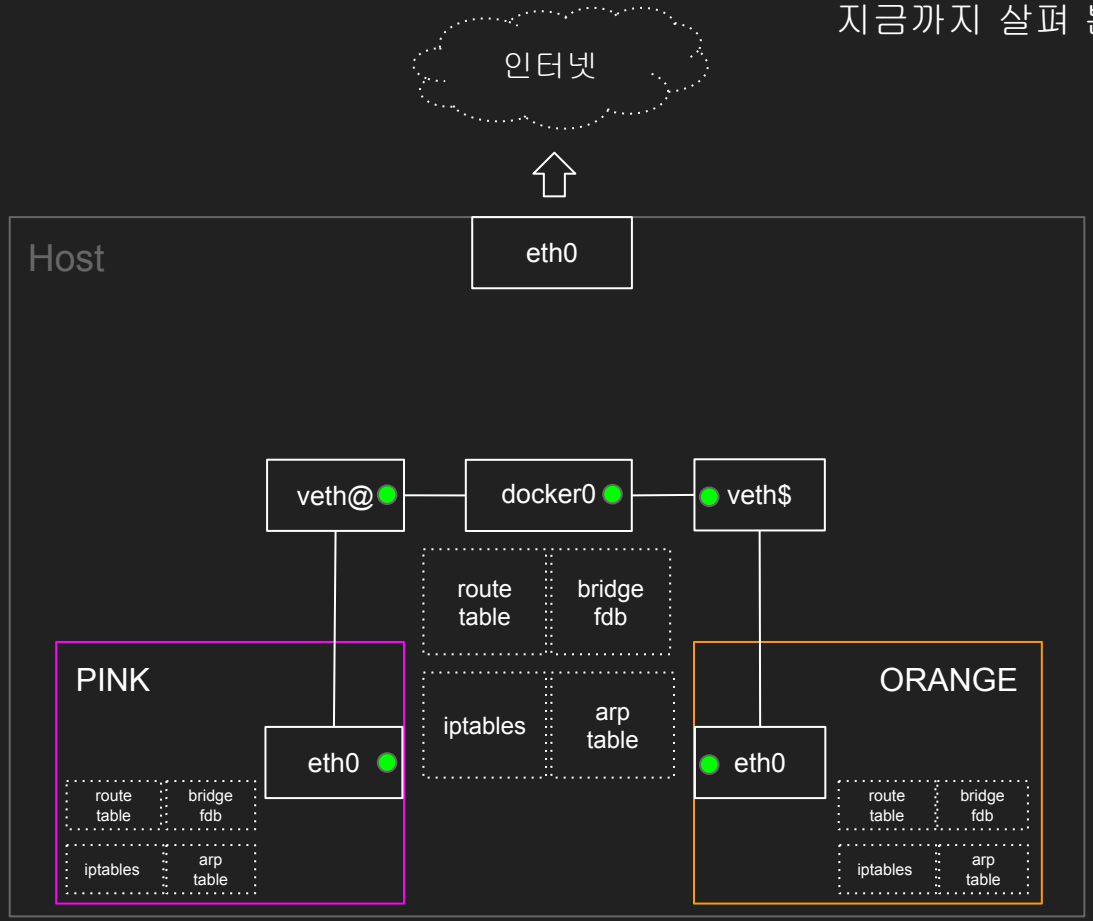
컨테이너의 iptables도 확인해 보세요

터미널 #3 (ORANGE)

```
# iptables -t filter -S

# iptables -t nat -S
```

도커 네트워크

지금까지 살펴 본 도커의 bridge 네트워크 구성입니다



3, 4편에서 다룬 네트워크 네임스페이스는
7,8편에서 다룰 오버레이 네트워크에서 이어가 보겠습니다

목차 보기

5편에서는 mount namespace에 대하여 다루도록 하겠습니다