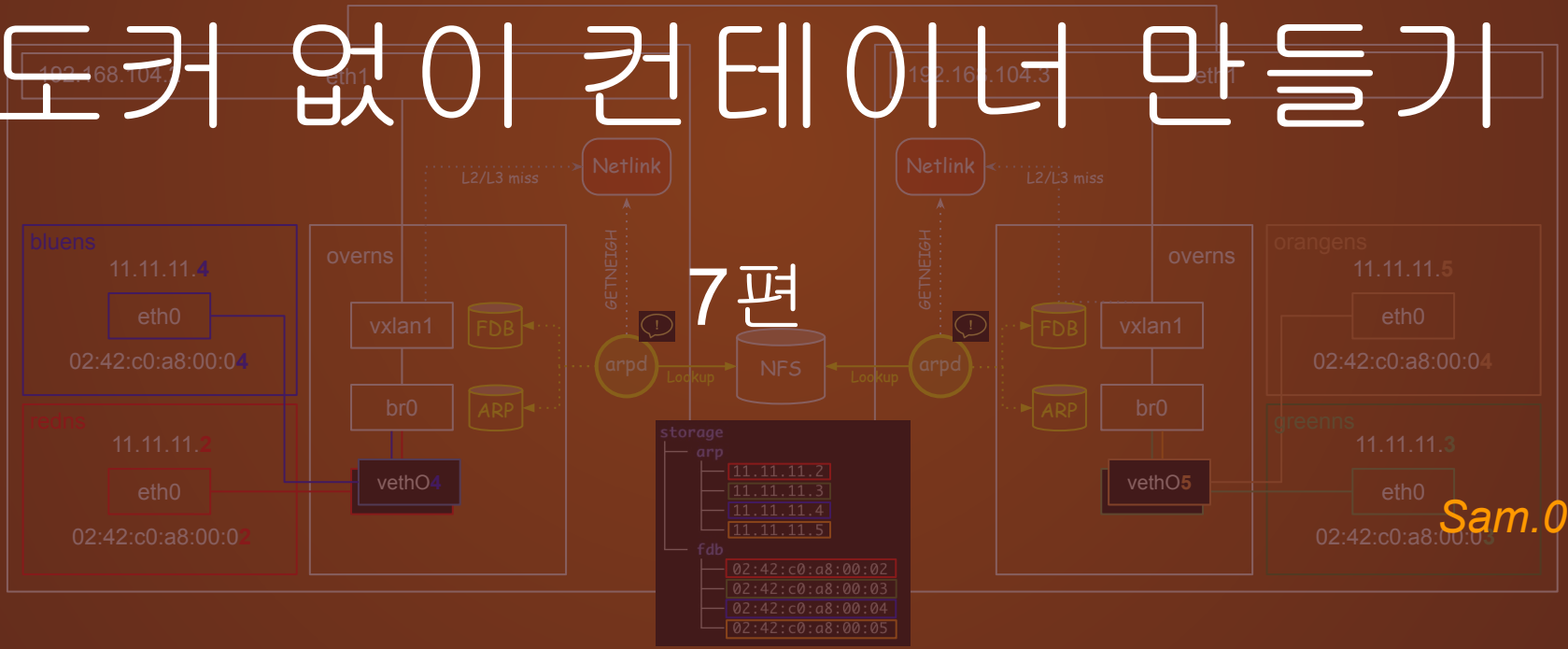
실습과 함께 완성해보는

도커 없이 컨테이너 만들기

7편



Sam.0

시작하기에 앞서 ...

본 컨텐츠는 앞편을 보았다고 가정하고 준비되었습니다.
원활한 이해 및 실습을 위하여 앞편을 먼저 보시기를 추천드립니다
*네트워크 네임스페이스 **3,4편**과 오버레이 네트워크(1) **6편**에 기초하여*
준비되었습니다.

3편 링크 클릭 *4편 링크 클릭* *6편 링크 클릭*

실습환경 구성 링크

실습환경

vagrant + virtual vm
ubuntu 18.04, docker (Vagrantfile)
- ubuntu1804 (기존)
- ubuntu1804-2 (추가)

실습 계정 (root)

```
# sudo -Es
```

실습 폴더

```
# cd /vagrant
```

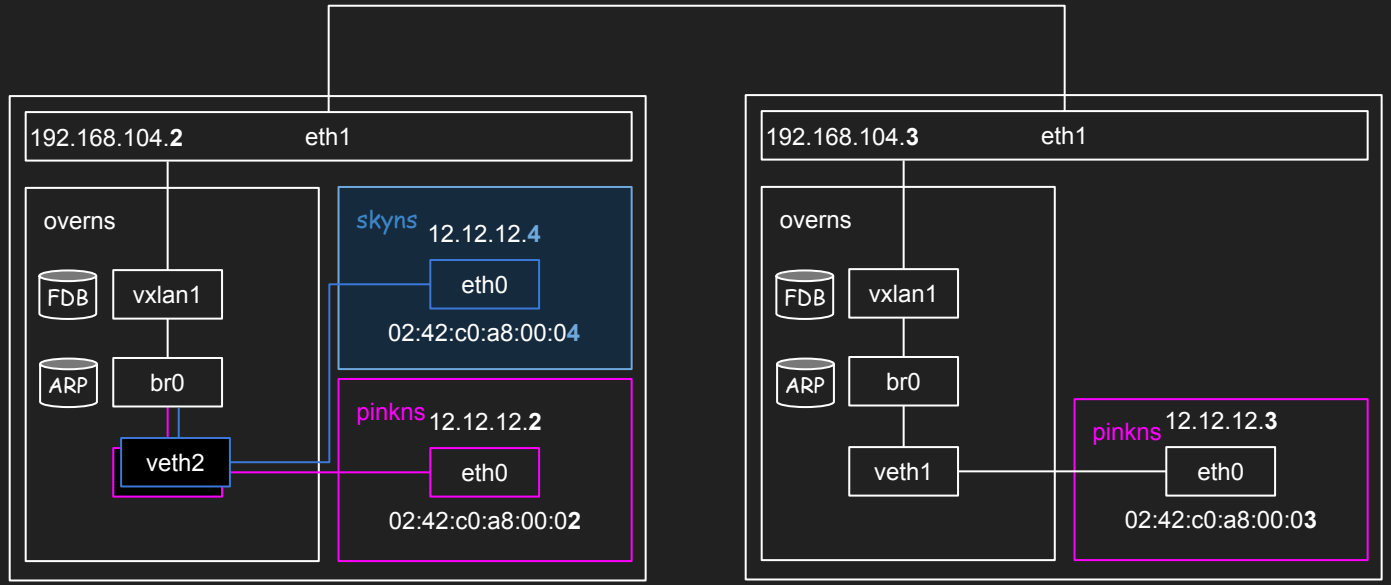
설치 환경

```
# apt-get -y install python-pip > /dev/null 2>&1
# pip install pyroute2
```

지난 시간에 이어서 네트워크 네임스페이스(skyns)를 추가해 볼까요

skyns

컨테이너의 호스트 네트워크는 이렇게 네트워크 네임스페이스로 격리되고
이더넷 페어 쌍으로 브릿지에 연결됩니다

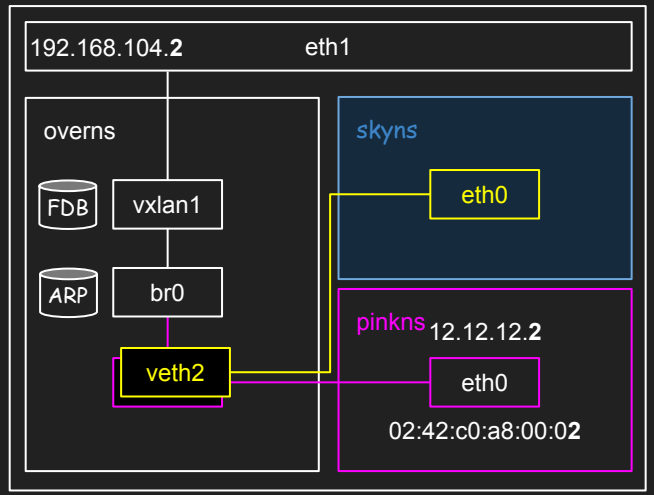


(실습2) skyns 추가

skyns

터미널 #1 (192.168.104.2)

```
# ip netns add skyns
# ip link add dev veth2 mtu 1450 netns overns type veth peer name eth0 mtu 1450 netns skyns
```

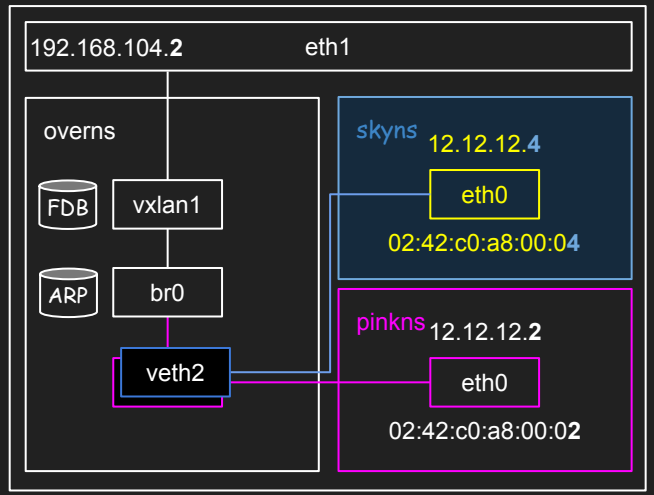


(실습2) skyns 추가

skyns

터미널 #1 (192.168.104.2)

```
# ip netns exec skyns ip link set dev eth0 address 02:42:c0:a8:00:04
# ip netns exec skyns ip addr add dev eth0 12.12.12.4/24
```

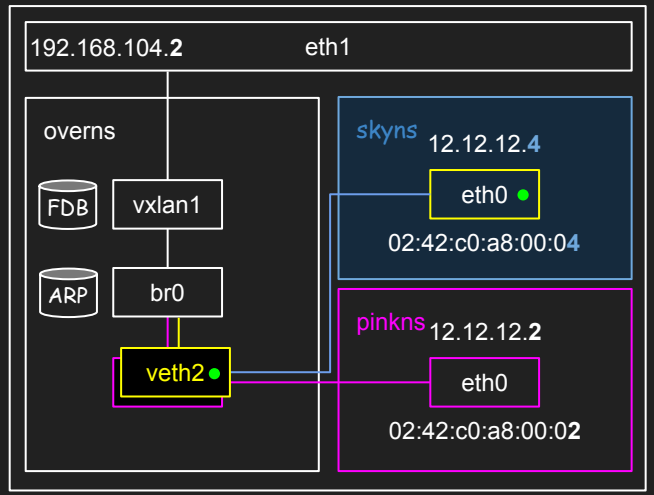


(실습2) skyns 추가

skyns

터미널 #1 (192.168.104.2)

```
# ip netns exec overns ip link set veth2 master br0
# ip netns exec overns ip link set veth2 up
# ip netns exec skyns ip link set eth0 up
```



(실습2) skyns 추가

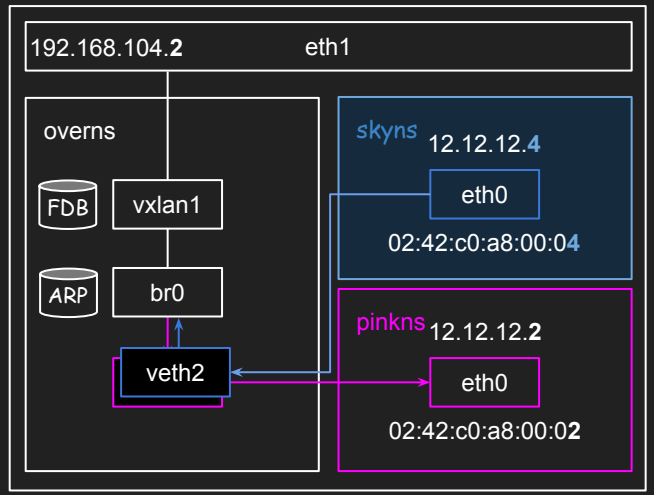
12.12.12.4 ← (ping) → 12.12.12.2

터미널 #1 (192.168.104.2)

```
# ip netns exec skyns ping -c 1 12.12.12.2
64 bytes from 12.12.12.2: icmp_seq=3 ttl=64 time=0.073 ms

# ip netns exec pinkns ping -c 1 12.12.12.4
64 bytes from 12.12.12.4: icmp_seq=35 ttl=64 time=0.051 ms
```

Easy ~



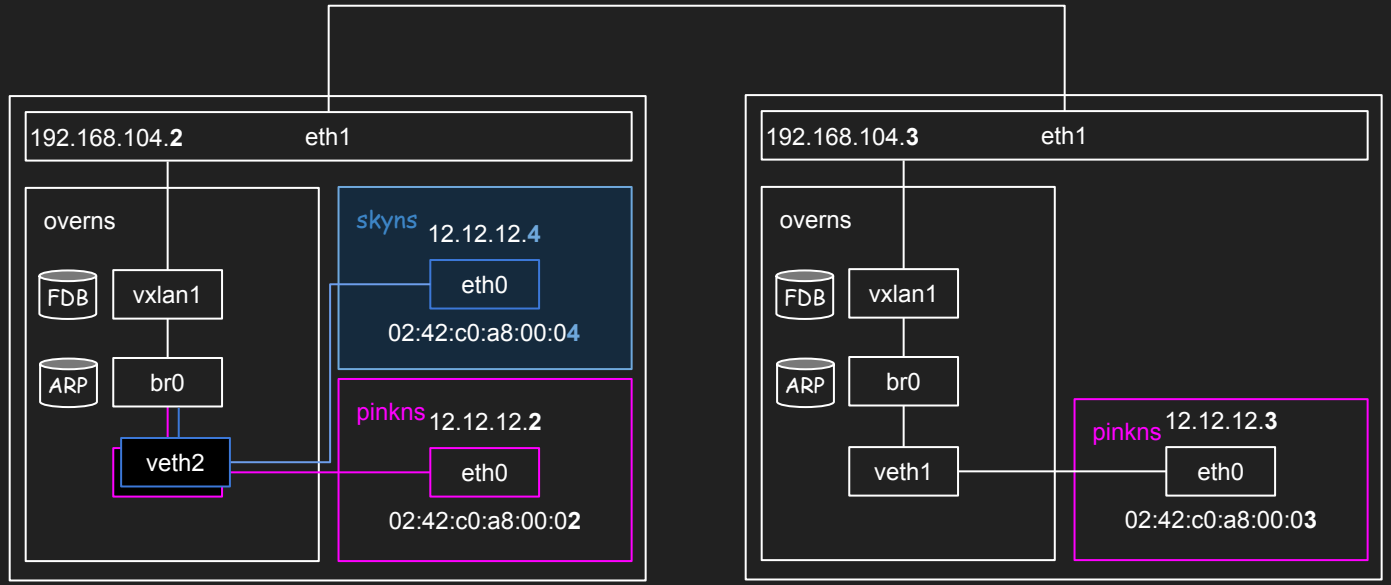
(실습2) skyns 추가

12.12.12.4 → 12.12.12.3

터미널 #1 (192.168.104.2)

```
# ip netns exec skyns ping 12.12.12.3
```

무엇을 더 해주어야 할까요?



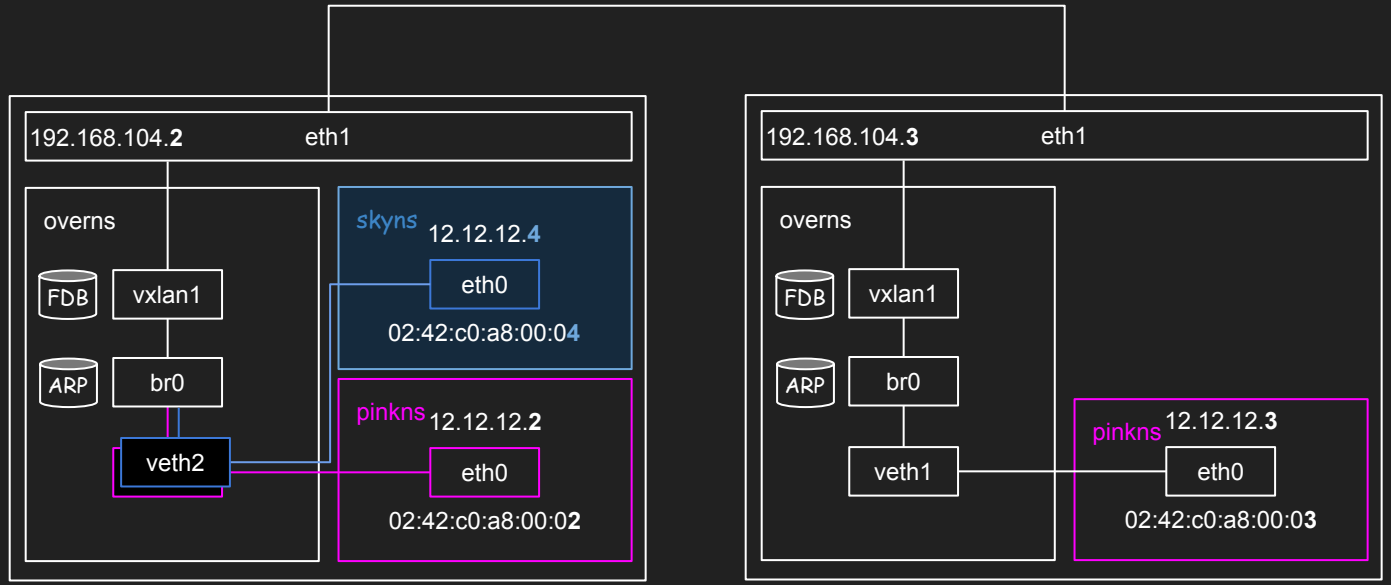
(실습2) skyns 추가

12.12.12.4 → 12.12.12.3

터미널 #1 (192.168.104.2)

```
# ip netns exec skyns ping 12.12.12.3
```

정답: ARP,FDB 등록



(실습2) skyns 추가

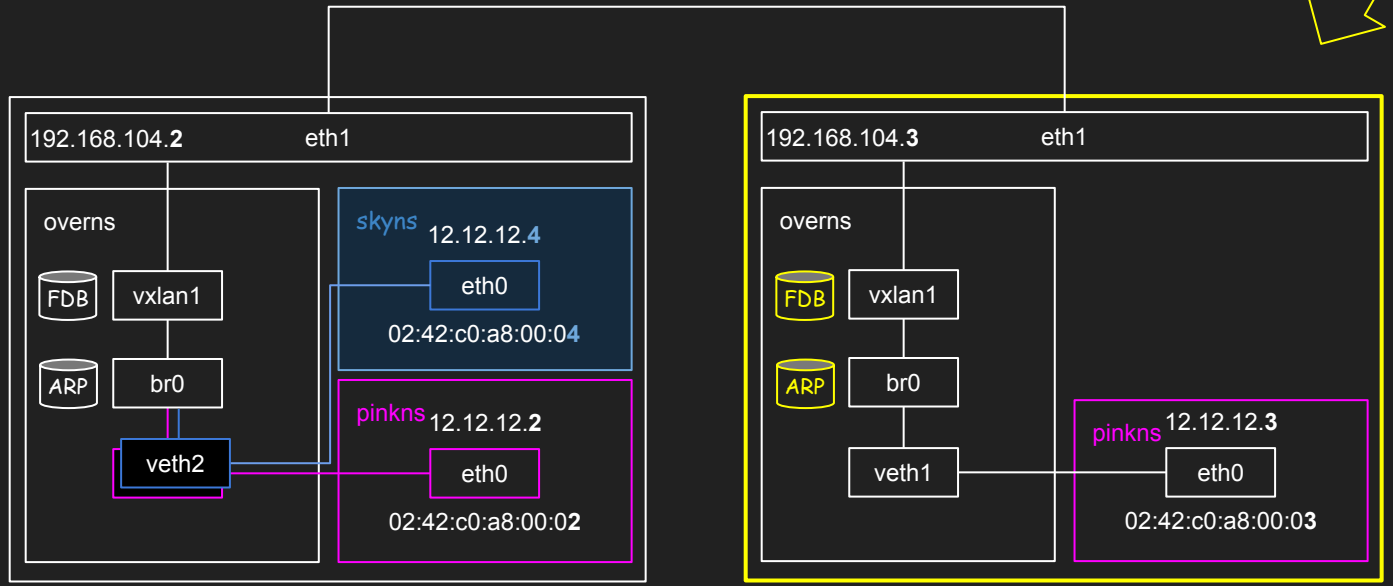
12.12.12.4 → 12.12.12.3

터미널 #1 (192.168.104.2)

```
# ip netns exec skyns ping 12.12.12.3
```

정답: ARP,FDB 등록

어느쪽 ?



(실습2) skyns 추가

엇? ARP 정보만 넣었는데 ping이 가네요?

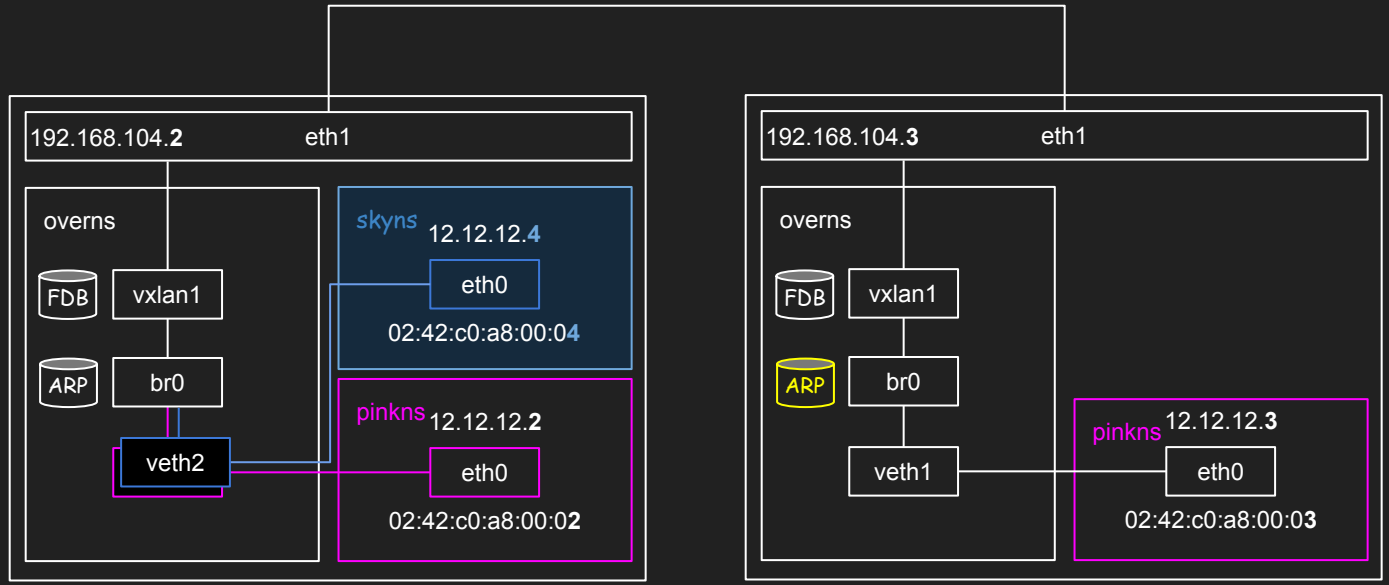
터미널 #1 (192.168.104.2)

```
# ip netns exec skyns ping 12.12.12.3
...
64 bytes from 12.12.12.3: icmp_seq=731 ttl=64 time=0.466 ms
```

터미널 #2 (overns@192.168.104.3)

```
# nsenter --net=/var/run/netns/overns

# ip neigh add 12.12.12.4 lladdr 02:42:c0:a8:00:04 dev vxlan1
```



(실습2) skyns 추가

bridge FDB는 "learning" 되기 때문이죠 :-)

터미널 #1 (192.168.104.2)

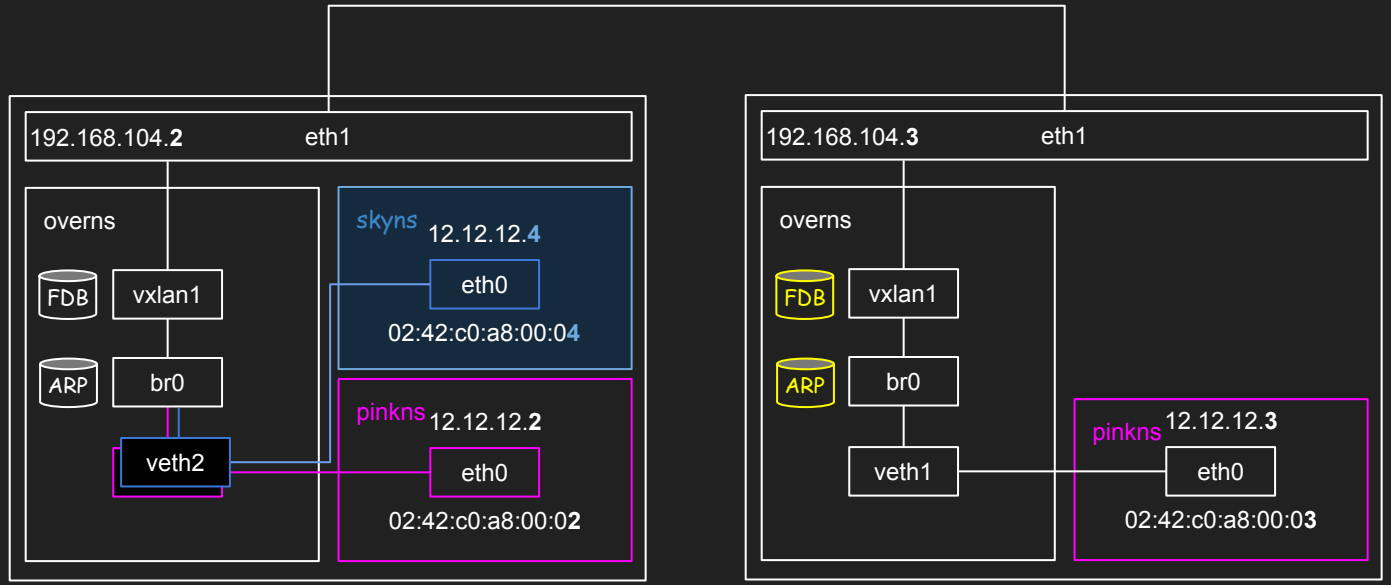
```
# ip netns exec skyns ping 12.12.12.3
...
64 bytes from 12.12.12.3: icmp_seq=731 ttl=64 time=0.466 ms
```

터미널 #2 (overns@192.168.104.3)

```
# bridge fdb replace 02:42:c0:a8:00:04 dev vxlan1
self dst 192.168.104.2 vni 42 port 4789

## learning된 정보가 있으면 add는 실패하므로 replace 사용
```

~ But, aging 때문에 fdb entry가 지워지기 때문에 입력해 줍시다



정리해보면 ...

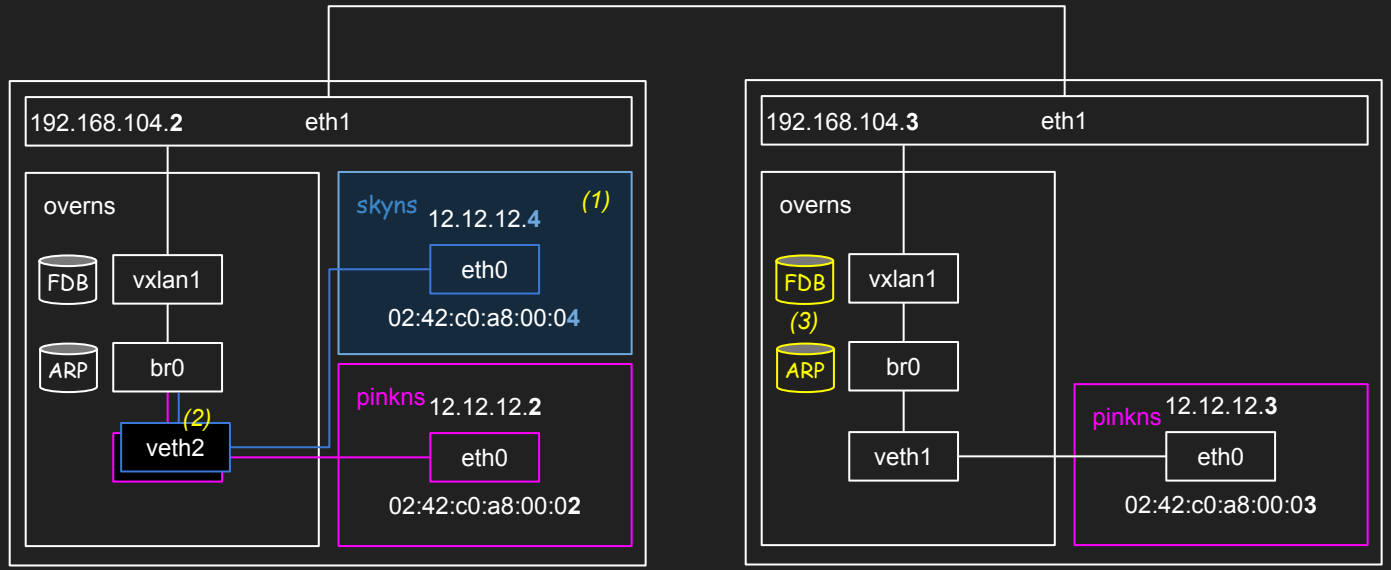
(0) overns 와 vxlan1, br0는 노드당 최초 한번만 생성

컨테이너 추가는 (1) ~ (3) 반복

(1) 컨테이너 생성

(2) veth를 br0에 연결

(3) ARP, FDB에 정보등록



동적으로 컨테이너를 추가해 봅시다

실습 스크립트 링크 클릭

(0) overns 와 vxlan1, br0는 노드당 최초 한번만 생성 → create_overlay.sh

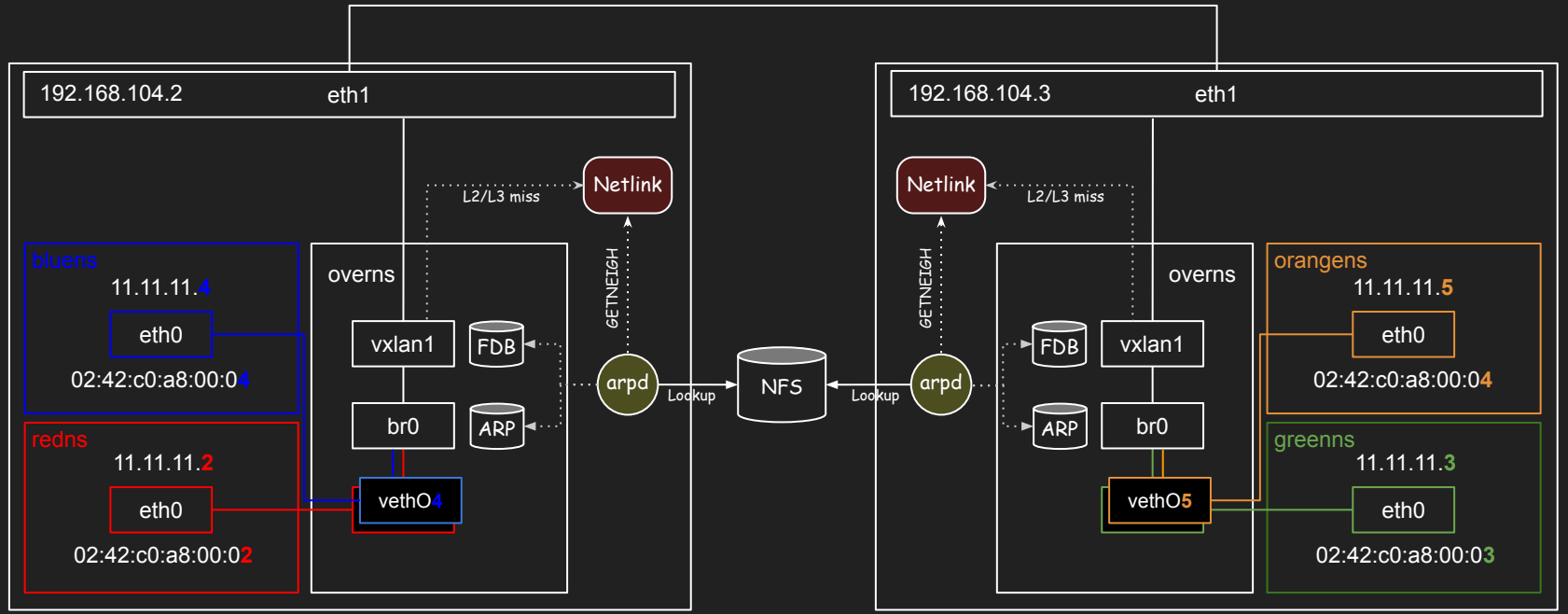
(1) 컨테이너 생성 → attach_ctn.sh

(2) veth를 br0에 연결 → attach_ctn.sh

(3) ARP, FDB에 정보등록 → arpd.py

Dynamic Overlay Network

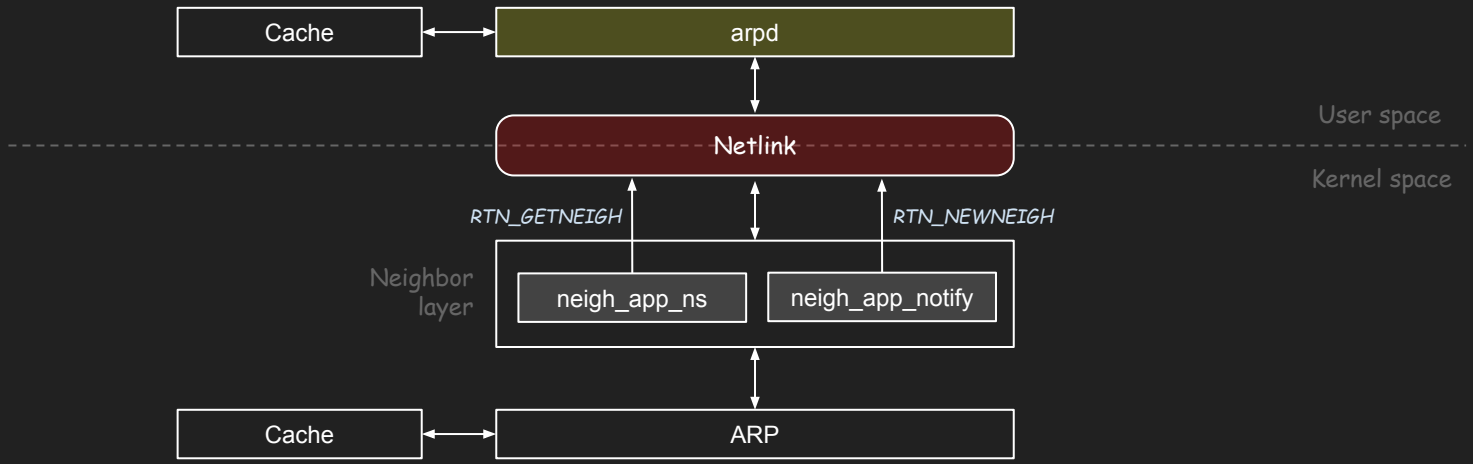
완성할 그림



Dynamic Overlay Network

RTNL (RTNetlink)를 이용한 arp table, bridge fdb 동적 갱신

*~ arpd (*pyroute2)가 **L2 miss, L3miss** 이벤트를 **catch**하여 arp table, fdb table 갱신*



Netlink

Netlink는 커널과 유저스페이스 프로세스 사이에서 정보교환을 하는데 사용됨

- communication between kernel and user space
- socket-based interface
- datagram-oriented service
- netlink_family selects the kernel module or netlink group to communicate with

“User space 와 Kernel space 간의 불편한 ioctl 통신방법에 대한 좀 더 유연한 대안으로 만들어짐”

https://man7.org/linux/man-pages/man7/netlink.7.html

Netlink Families

커널 내 여러 컴포넌트들과의 통신을 위한 다양한 프로토콜들 (families)로 구성

- NETLINK_ROUTE
- NETLINK_USERSOCK
- NETLINK_FIREWALL
- NETLINK_SOCK_DIAG
- NETLINK_INET_DIAG
- NETLINK_NETFILTER
- NETLINK_SELINUX
- NETLINK_W1
- NETLINK_NFLOG
- NETLINK_XFRM
- NETLINK_ISCSI
- NETLINK_AUDIT
- … 그외 다수 ...

https://man7.org/linux/man-pages/man7/netlink.7.html

Netlink Families

커널 내 여러 컴포넌트들과의 통신을 위한 다양한 프로토콜들 (families)로 구성

- NETLINK_ROUTE
- NETLINK_USERSOCK
- NETLINK_FIREWALL
- NETLINK_SOCK_DIAG
- NETLINK_INET_DIAG
- NETLINK_NETFILTER
- NETLINK_SELINUX
- NETLINK_W1
- NETLINK_NFLOG
- NETLINK_XFRM
- NETLINK_ISCSI
- NETLINK_AUDIT
- … 그외 다수 ...

https://man7.org/linux/man-pages/man7/netlink.7.html

RTNETLINK (RTNL)

라우팅과 링크 계층의 설정을 위한 인터페이스

- 커널의 라우팅 테이블을 읽거나 수정
- 커널과 커뮤니케이션하는 창구 역할
- 네트워크 제어를 제공
  - n/w routes, ip addresses, neighbor setups, queueing disciplines, traffic classes, ...
  - NETLINK_ROUTE sockets 을 통해서 제어
- based on netlink messages

https://man7.org/linux/man-pages/man7/rtnetlink.7.html

Register message callback

```
extern rtnl_register(
            int protocol,
            int msgtype,
            rtnl_doit_func,
            rtnl_dumpit_func,
            rtnl_calcit_func
);
```

추가/삭제/수정

정보검색

버퍼 사이즈 계산

RTNETLINK Message Types

LINK : network interface
ADDR : IP address
ROUTE : routing information
NEIGH : neighbor table (or arp table)
RULE : routing rule
QDISC : queueing discipline
TCLASS : traffic class
TFILTER : traffic filter

CRD (create/read/delete) 제공

https://man7.org/linux/man-pages/man7/rtnetlink.7.html

RTNETLINK Message Types

- RTM_NEWLINK, RTM_DELLINK, RTM_GETLINK
- RTM_NEWADDR, RTM_DELADDR, RTM_GETADDR
- RTM_NEWROUTE, RTM_DELROUTE, RTM_GETROUTE
- RTM_NEWNEIGH, RTM_DELNEIGH, RTM_GETNEIGH
- RTM_NEWRULE, RTM_DELRULE, RTM_GETRULE
- RTM_NEWQDISC, RTM_DELQDISC, RTM_GETQDISC
- RTM_NEWTCLASS, RTM_DELTCLASS, RTM_GETTCLASS
- RTM_NEWTFILTER, RTM_DELTFILTER, RTM_GETTFILTER

https://man7.org/linux/man-pages/man7/rtnetlink.7.html

RTNETLINK Message Types

- RTM_NEWLINK, RTM_DELLINK, RTM_GETLINK
- RTM_NEWADDR, RTM_DELADDR, RTM_GETADDR
- RTM_NEWROUTE, RTM_DELROUTE, RTM_GETROUTE
- RTM_NEWNEIGH, RTM_DELNEIGH, RTM_GETNEIGH
- RTM_NEWRULE, RTM_DELRULE, RTM_GETRULE
- RTM_NEWQDISC, RTM_DELQDISC, RTM_GETQDISC
- RTM_NEWTCLASS, RTM_DELTCLASS, RTM_GETTCLASS
- RTM_NEWTFILTER, RTM_DELTFILTER, RTM_GETTFILTER

https://man7.org/linux/man-pages/man7/rtnetlink.7.html

RTM_*NEIGH CRD for neighbor table (or arp table)

ndmsg (neighbor discovery message) 구조체

```
struct ndmsg {
          unsigned char ndm_family;
          int           ndm_ifindex;  /* Interface index */
          __u16         ndm_state;    /* State */
          __u8          ndm_flags;    /* Flags */
          __u8          ndm_type;
      };

struct nda_cacheinfo {
          __u32         ndm_confirmed;
          __u32         ndm_used;
          __u32         ndm_updated;
          __u32         ndm_refcnt;
      };
```

https://man7.org/linux/man-pages/man7/rtnetlink.7.html

ndm_state

ndmsg (neighbor discovery message)

NUD_INCOMPLETE	a currently resolving cache entry
NUD_REACHABLE	a confirmed working cache entry
NUD_STALE	an expired cache entry
NUD_DELAY	an entry waiting for a timer
NUD_PROBE	a cache entry that is currently reprobed
NUD_FAILED	an invalid cache entry
NUD_NOARP	a device with no destination cache
NUD_PERMANENT	a static entry

https://man7.org/linux/man-pages/man7/rtnetlink.7.html

rta_type

NDA_UNSPEC unknown type
NDA_DST a neighbor cache **n/w layer destination** address
NDA_LLADDR a neighbor cache **link layer destination** address
NDA_CACHEINFO cache statistics

Routing attributes

Some rtnetlink messages have optional attributes after the initial header:

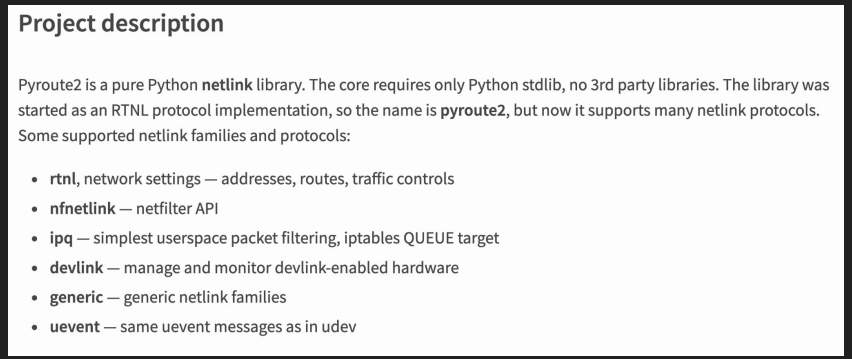
```
struct rtattr {
    unsigned short rta_len;    /* Length of option */
    unsigned short rta_type;   /* Type of option */
    /* Data follows */
};
```

These attributes should be manipulated using only the RTA_* macros or libnetlink, see rtnetlink(3).

https://man7.org/linux/man-pages/man7/rtnetlink.7.html

Pyroute2

Pyroute2 is a pure Python netlink library.
… The library was started as an RTNL protocol implementation, so the name is pyroute2, but now it supports many netlink protocols



Project description

Pyroute2 is a pure Python **netlink** library. The core requires only Python stdlib, no 3rd party libraries. The library was started as an RTNL protocol implementation, so the name is **pyroute2**, but now it supports many netlink protocols. Some supported netlink families and protocols:

- **rtnl**, network settings — addresses, routes, traffic controls
- **nfnetlink** — netfilter API
- **ipq** — simplest userspace packet filtering, iptables QUEUE target
- **devlink** — manage and monitor devlink-enabled hardware
- **generic** — generic netlink families
- **uevent** — same uevent messages as in udev

https://pypi.org/project/pyroute2/

(실습3) Dynamic Overlay Network

실습준비

실습 스크립트 링크 참고

준비물 : overlaynet.tar

```
# cd /vagrant
# tar -xf overlaynet.tar
# tree /vagrant/overlaynet
```

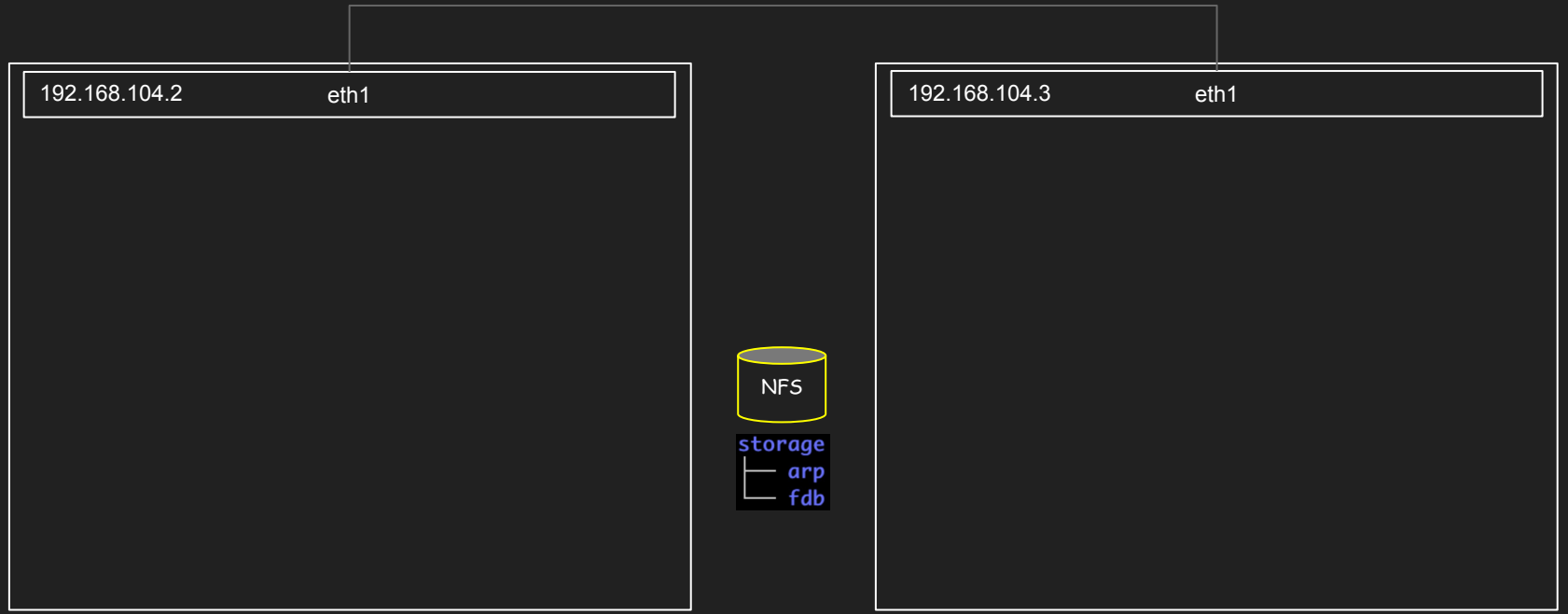
```
/vagrant/overlaynet
├── arpd.py ------------------- netlink event를 catch하여 arp daemon. arp/fdb 정보 갱신
├── attach_ctn.sh ------------- container를 생성하고 br0 (bridge)에 연결, mac/ip 주소 등록
├── check.sh ------------------ app_solicit 활성여부 체크 (사용안함) * vxlan l2miss/l3miss 활성
├── create_overlay.sh -------- overns 및 bridge/vxlan 구성
├── l2l3miss.py --------------- container 생성, bridge 연결, mac/ip 주소 등록 (NFS)
├── reset.sh ------------------ 초기화. 모든 netns 및 storage/{arp,fdb} 삭제
└── storage ------------------- NFS. 클러스터 내 모든 컨테이너의 arp/fdb 공유 저장소
    ├── arp
    └── fdb
```

지금부터 실습 경로는 /vagrant/overlaynet 입니다.

(실습3) Dynamic Overlay Network

터미널 #1 (192.168.104.2) + 터미널 #2 (192.168.104.3)

```
# cd /vagrant/overlaynet
# ./reset.sh
# ip netns
```

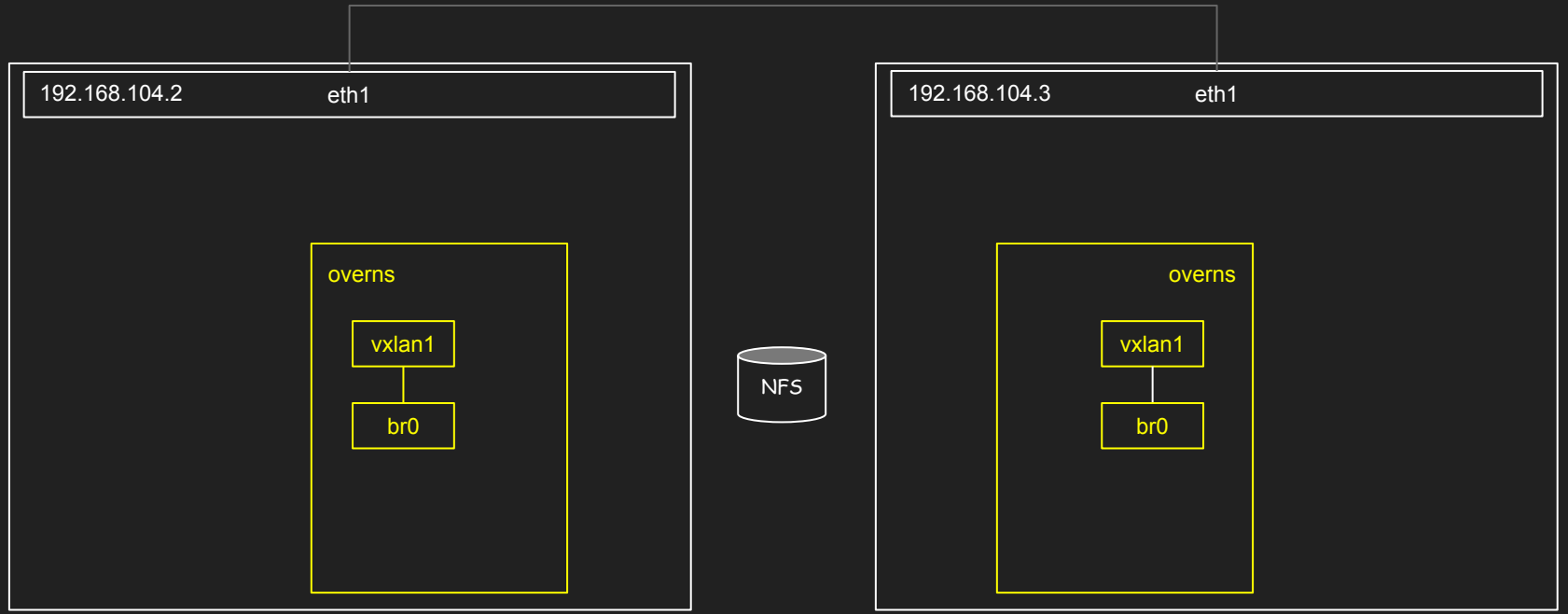


(실습3) Dynamic Overlay Network

overns 생성

터미널 #1 (192.168.104.2) + 터미널 #2 (192.168.104.3)

```
# ./create_overlay.sh
```



(실습3) Dynamic Overlay Network

컨테이너 생성

터미널 #1 (192.168.104.2)

주소 postfix

```
# ./attach_ctn.sh redns 2
```

네임스페이스

터미널 #2 (192.168.104.3)

```
# ./attach_ctn.sh greenns 3
```

(실습3) Dynamic Overlay Network

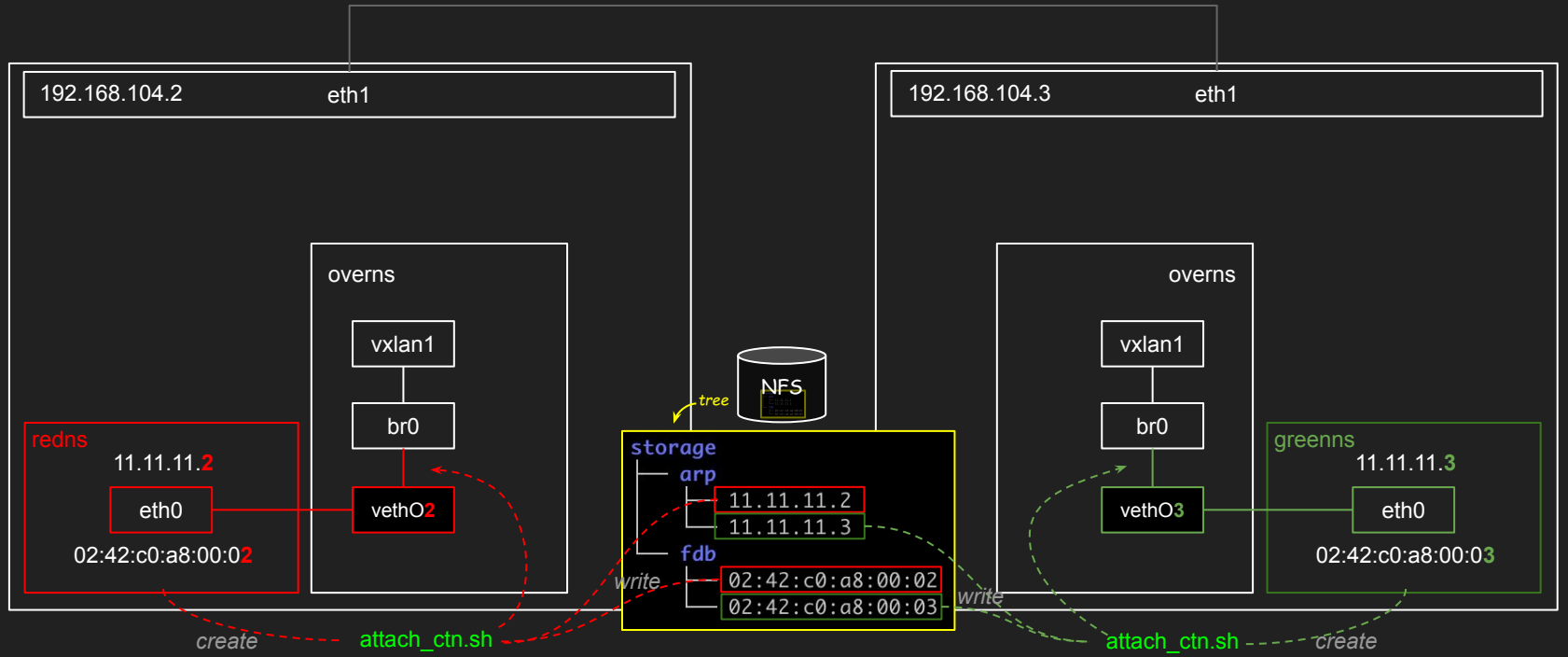
컨테이너 생성

터미널 #1 (192.168.104.2)

```
# ./attach_ctn.sh redns 2
# tree
```

터미널 #2 (192.168.104.3)

```
# ./attach_ctn.sh greenns 3
# tree
```



(실습3) Dynamic Overlay Network

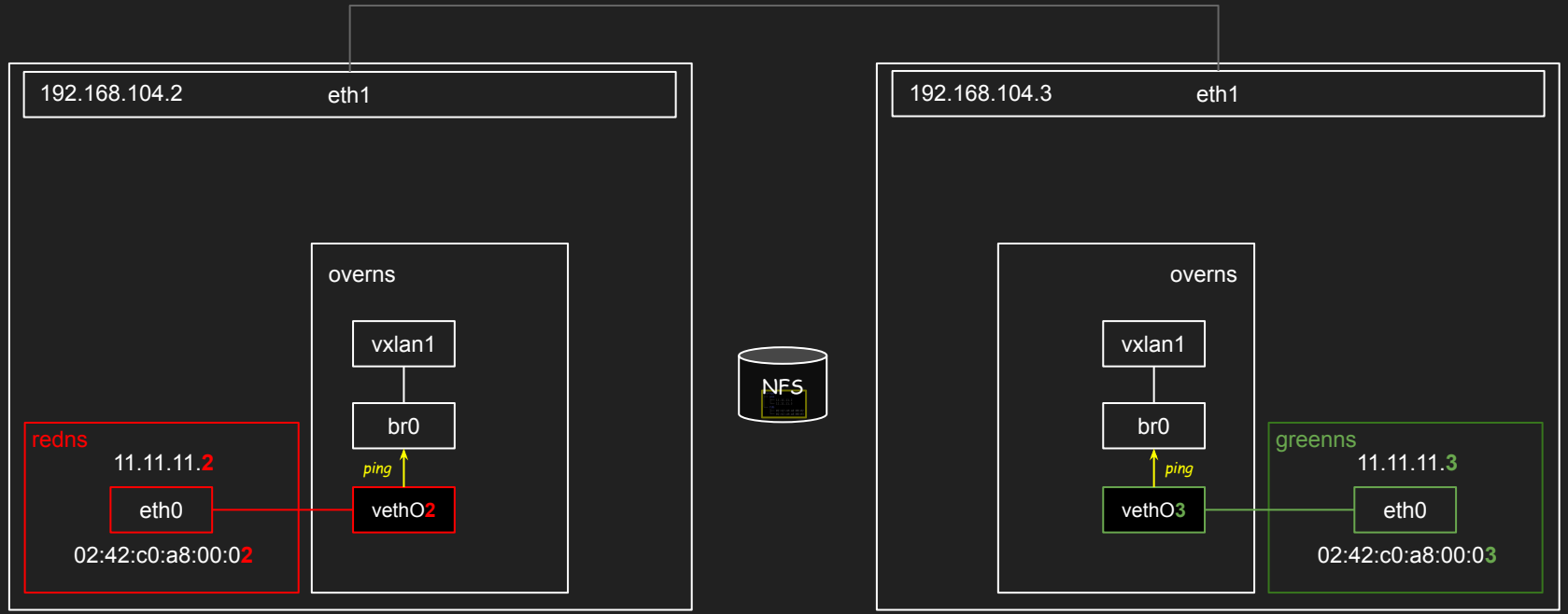
컨테이너 생성

터미널 #1 (192.168.104.2)

```
# ip netns exec redns ping -c 1 11.11.11.1
```

터미널 #2 (192.168.104.3)

```
# ip netns exec greenns ping  -c 1 11.11.11.1
```



(실습3) Dynamic Overlay Network

Check L2L3miss

터미널 #1 (192.168.104.2)

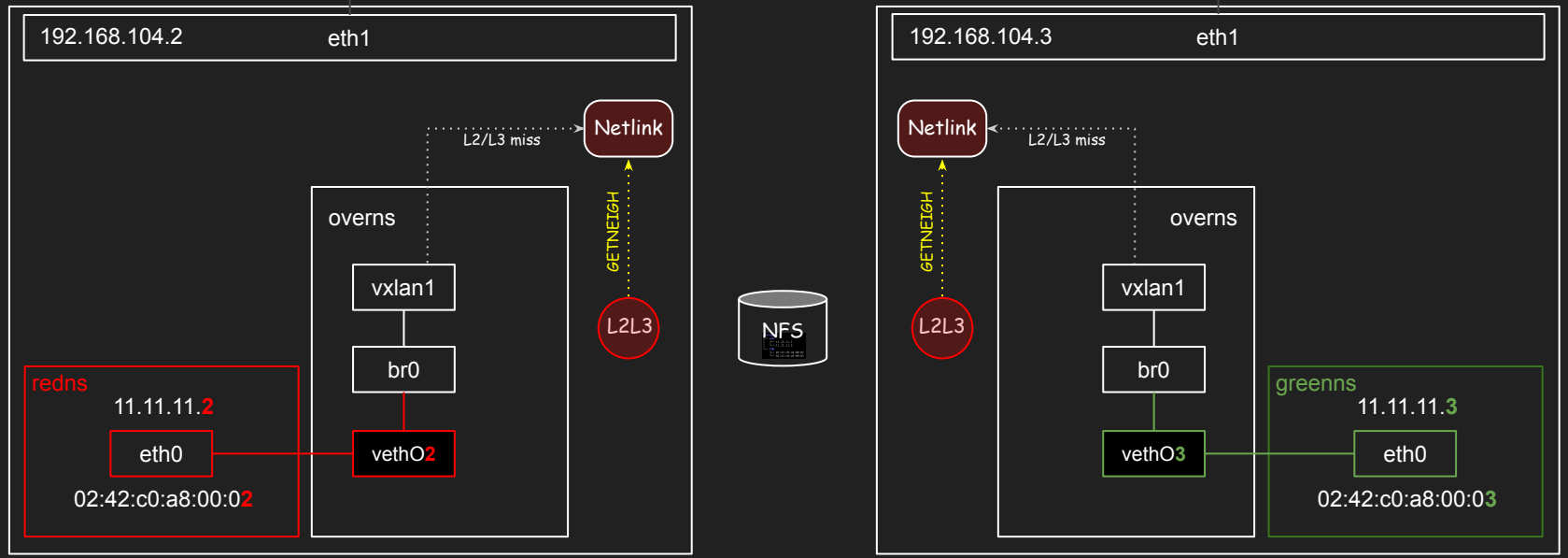
```
# ip netns exec redns ping 11.11.11.3
```

터미널 #2 (192.168.104.3)

```
# ./l2l3miss.py
```

터미널 #3 (192.168.104.2)

```
# ./l2l3miss.py
```



(실습3) Dynamic Overlay Network

터미널 #1 (192.168.104.2)

```
# ip netns exec redns ping -c 1 11.11.11.3
```

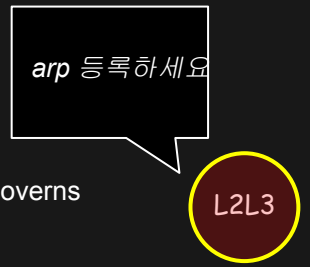
터미널 #2 (192.168.104.3)

```
# ./l2l3miss.py
```

터미널 #3 (192.168.104.2)

```
# ./l2l3miss.py

DEBUG:root:Received an event: RTM_GETNEIGH
DEBUG:root:------------ ndmsg start ------------
DEBUG:root:Family: AF_INET
DEBUG:root:Interface vxlan1 index: 3
DEBUG:root:State: NUD_STALE
DEBUG:root:Flags: 0
DEBUG:root:Type: RTN_UNICAST
DEBUG:root:------------ ndmsg end --------------
INFO:root:L3Miss on vxlan1: Who has IP: 11.11.11.3? Check arp table on overns
```



(실습3) Dynamic Overlay Network

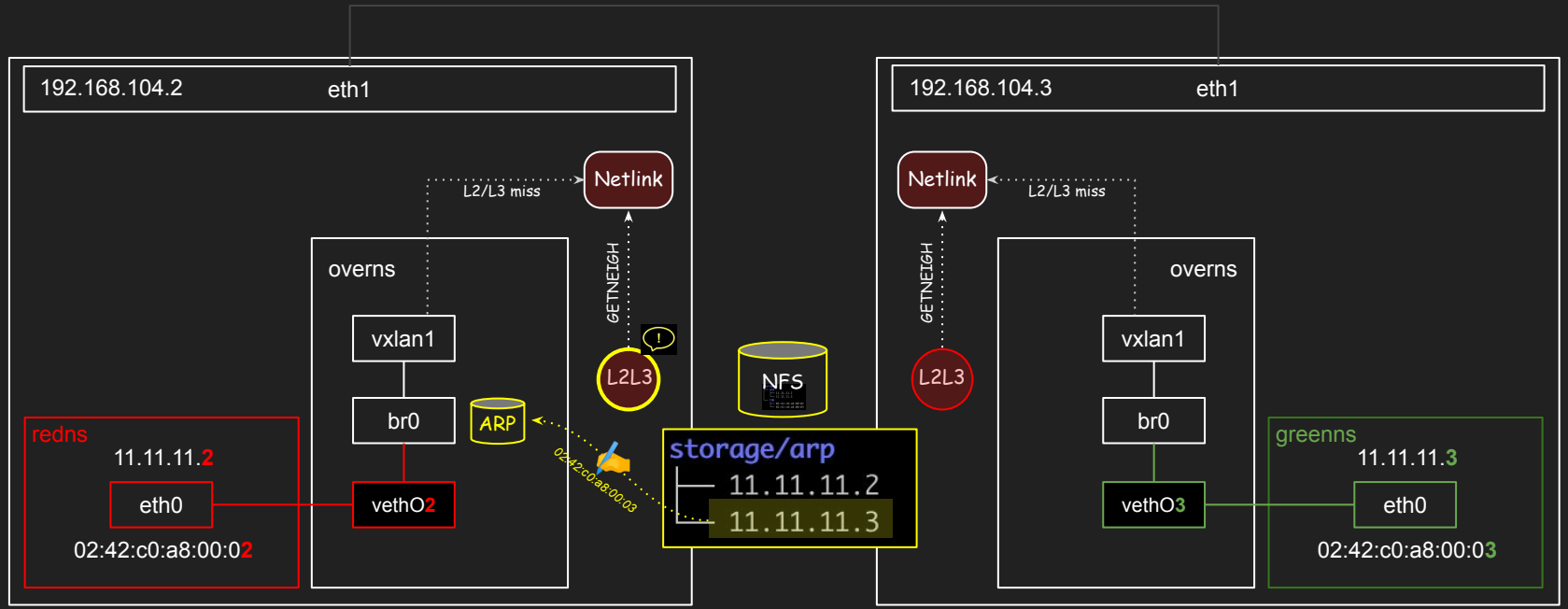
INFO:root:L3Miss on vxlan1: Who has IP: 11.11.11.3? Check arp table on overns

(실습3) Dynamic Overlay Network

L3miss → arp table 등록

터미널 #1 (192.168.104.2)

```
# ip netns exec overns ip neigh add 11.11.11.3 lladdr 02:42:c0:a8:00:03 dev vxlan1
```



(실습3) Dynamic Overlay Network

L3miss OK → L2miss 발생

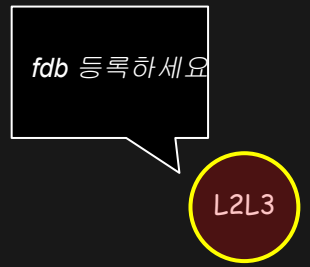
터미널 #1 (192.168.104.2)

```
# ip netns exec redns ping -c 1 11.11.11.3
```

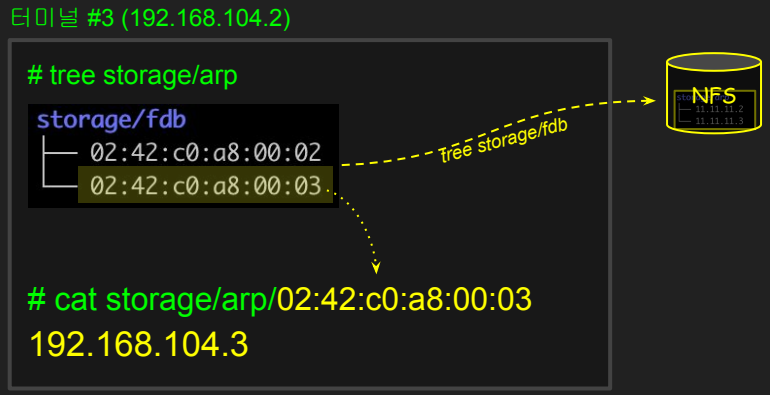
터미널 #3
(192.168.104.2)

```
# ./l2l3miss.py

DEBUG:root:Received an event: RTM_GETNEIGH
DEBUG:root:------------ ndmsg start ------------
DEBUG:root:Family: AF_INET
DEBUG:root:Interface vxlan1 index: 3
DEBUG:root:State: NUD_STALE
DEBUG:root:Flags: 0
DEBUG:root:Type: RTN_UNICAST
DEBUG:root:------------ ndmsg end --------------
INFO:root:L2Miss on vxlan1: Who has MAC: 02:42:c0:a8:00:03? Check bridge fdb on overns
```



INFO:root:L2Miss on vxlan1: Who has MAC: 02:42:c0:a8:00:03? Check bridge fdb on overns

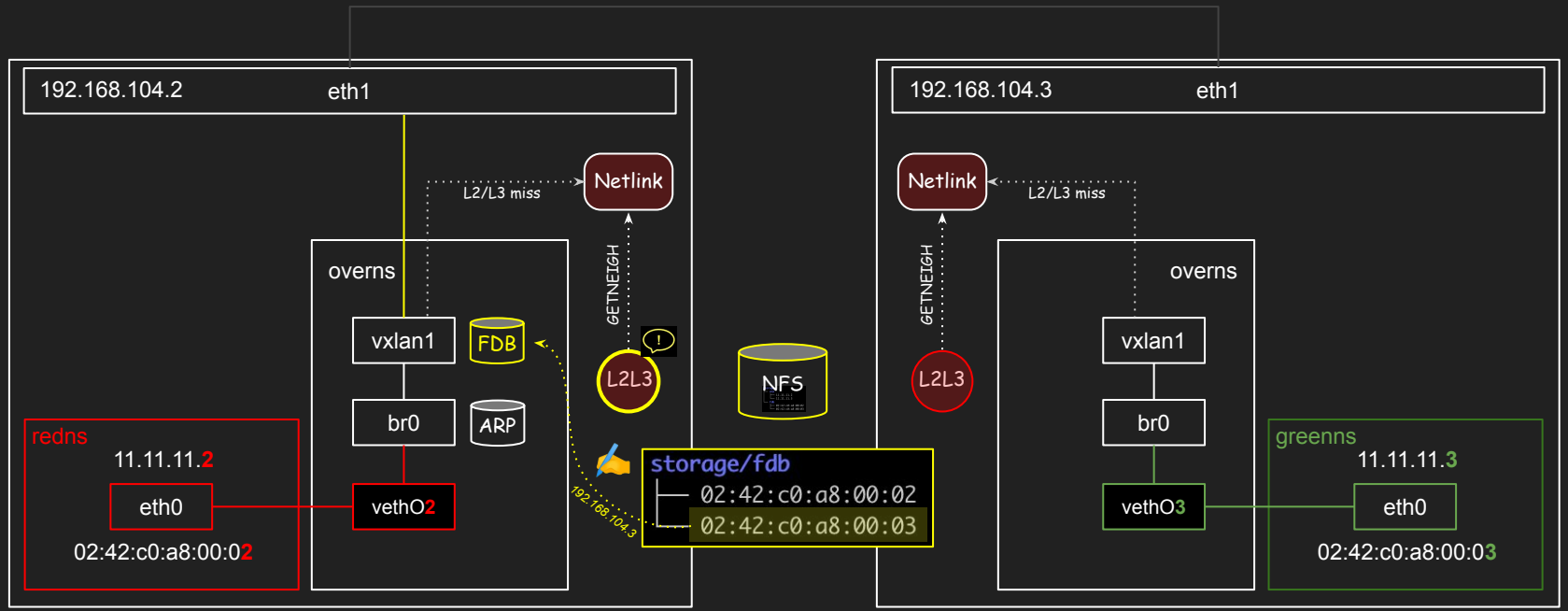


(실습3) Dynamic Overlay Network

L2miss → bridge fdb 등록

터미널 #1 (192.168.104.2)

```
# ip netns exec overns bridge fdb replace 02:42:c0:a8:00:03 dev vxlan1 self dst 192.168.104.3 vni 42 port 4789
```



(실습3) Dynamic Overlay Network

L2miss OK → 수신측 L2miss

터미널 #1 (192.168.104.2)

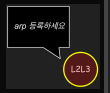
```
# ip netns exec redns ping 11.11.11.3
```

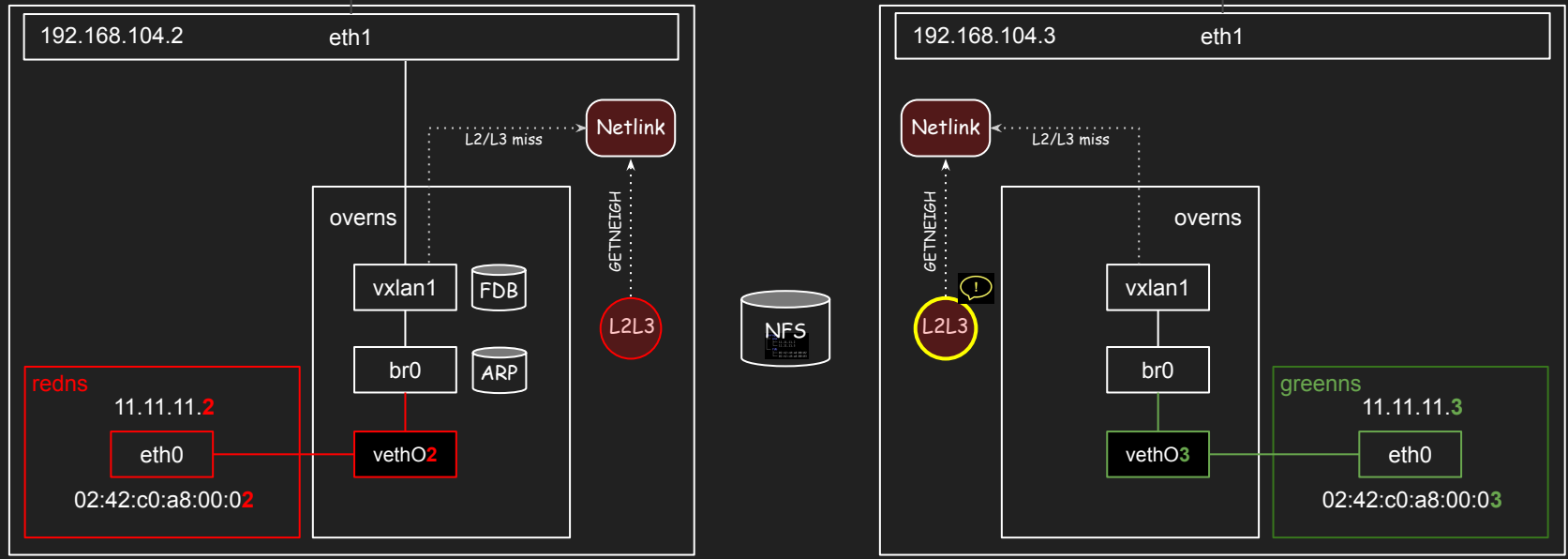
터미널 #3 (192.168.104.2)

```
# ./l2l3miss.py
```

터미널 #2 (192.168.104.3)

```
# ./l2l3miss.py
. . .
INFO:root:L3Miss on vxlan1: Who has IP: 11.11.11.2?
Check arp table on overns
```





(실습3) Dynamic Overlay Network

수신측 L2miss ~ arp 등록

터미널 #1 (192.168.104.2)

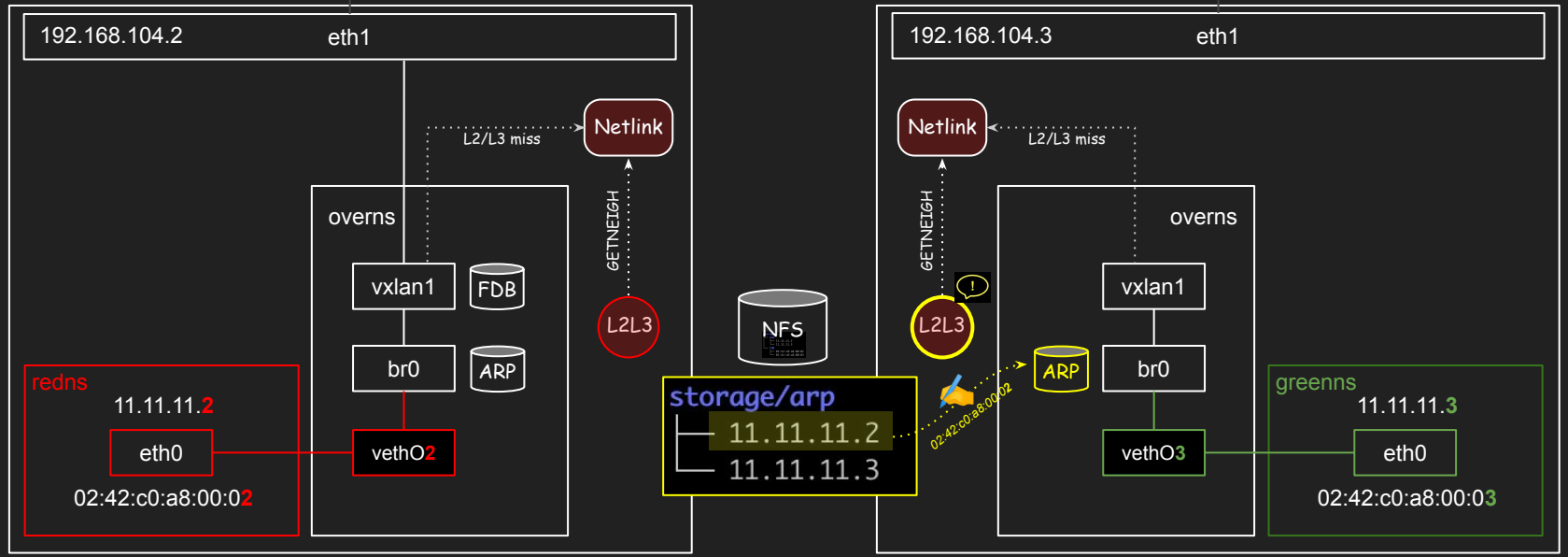
```
# ip netns exec redns ping -c 1 11.11.11.3
```

터미널 #2 (192.168.104.3)

```
# ip netns exec overns ip neigh add 11.11.11.2 lladdr
02:42:c0:a8:00:02 dev vxlan1
```

터미널 #3 (192.168.104.2)

```
# ./l2l3miss.py
```



(실습3) Dynamic Overlay Network

수신측 L2miss ~ OK

터미널 #1 (192.168.104.2)

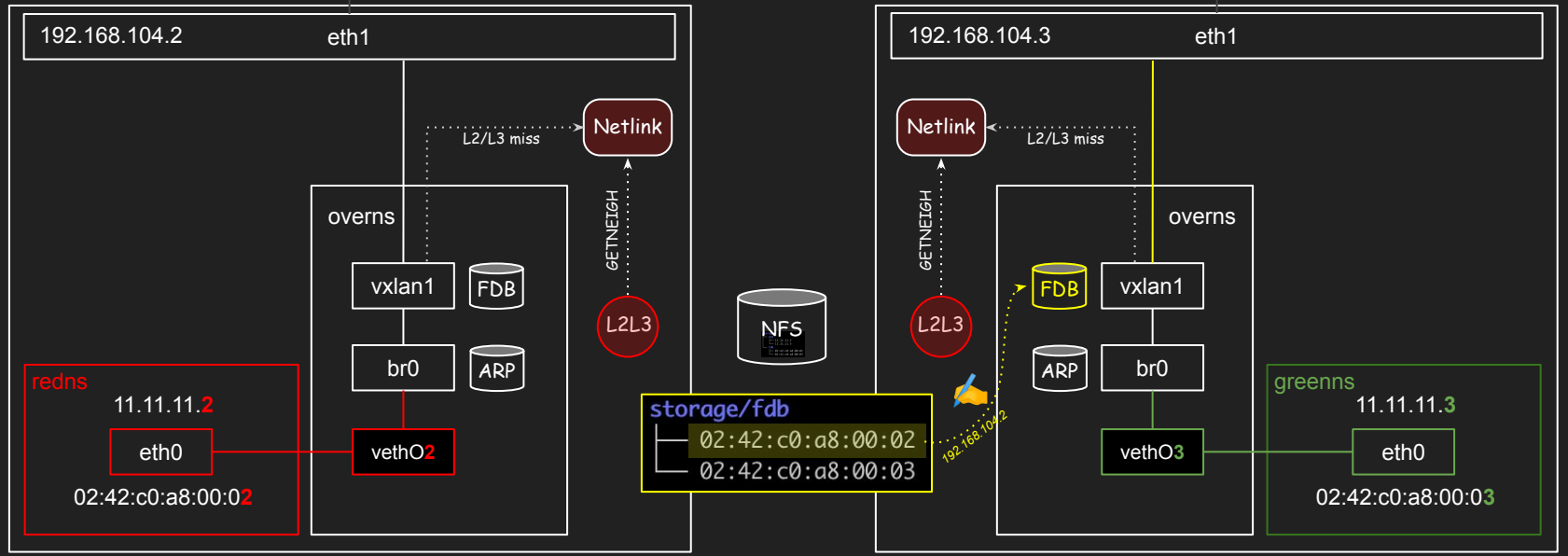
```
# ip netns exec redns ping -c 1 11.11.11.3
```

터미널 #3 (192.168.104.2)

```
# ./l2l3miss.py
```

터미널 #2 (192.168.104.3)

```
# ip netns exec overns bridge fdb replace
02:42:c0:a8:00:02 dev vxlan1 self dst 192.168.104.2 vni
42 port 4789
```



(실습3) Dynamic Overlay Network

red ← ping → green OK

터미널 #1 (192.168.104.2)

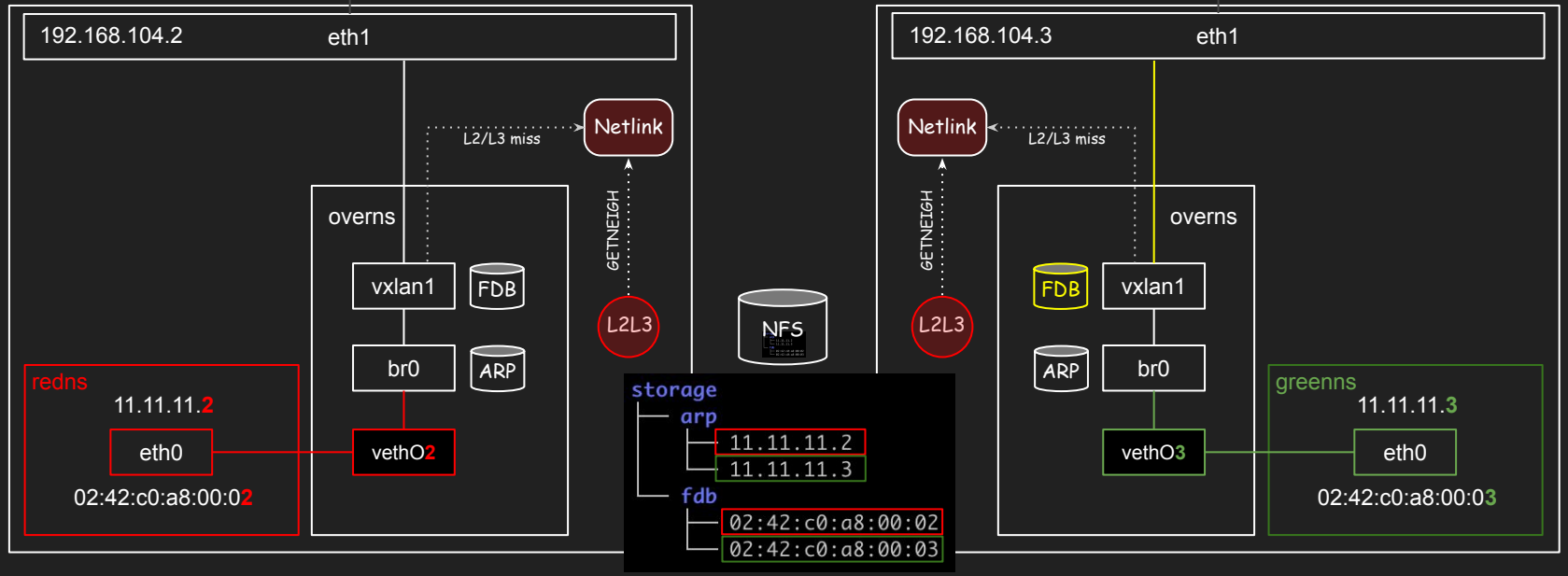
```
# ip netns exec redns ping -c 1 11.11.11.3
```

터미널 #3 (192.168.104.2)

```
# ./l2l3miss.py
```

터미널 #2 (192.168.104.3)

```
# ip netns exec overns bridge fdb replace
02:42:c0:a8:00:02 dev vxlan1 self dst 192.168.104.2 vni
42 port 4789
```



(실습3) Dynamic Overlay Network

arpd를 이용하여 동적으로 arp,fdb를 등록해봅시다

터미널 #1 (192.168.104.2)

```
#
```

터미널 #2 (192.168.104.3)

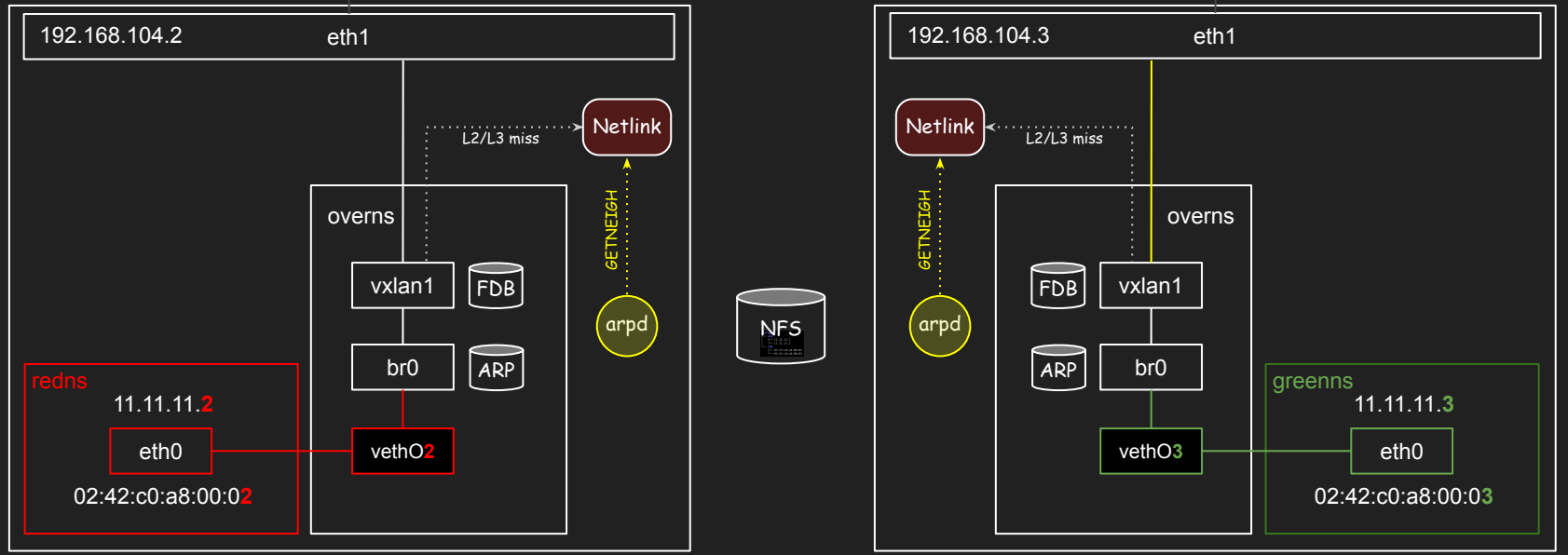
```
#
```

터미널 #3 (192.168.104.2)

```
# ./arpd.py
```

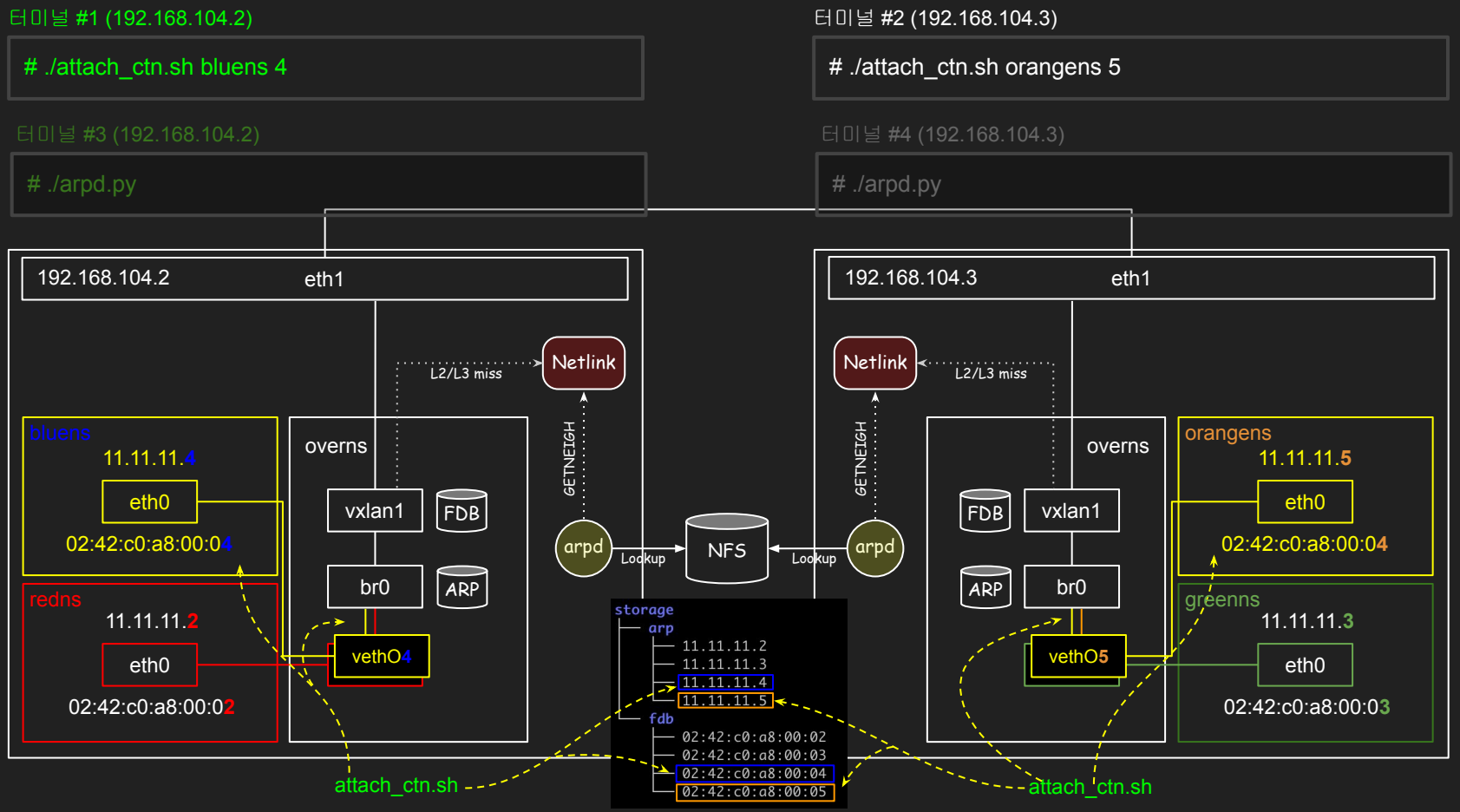
터미널 #4 (192.168.104.3)

```
# ./arpd.py
```



(실습3) Dynamic Overlay Network

arpd를 이용하여 동적으로 arp,fdb를 등록해봅시다



(실습3) Dynamic Overlay Network

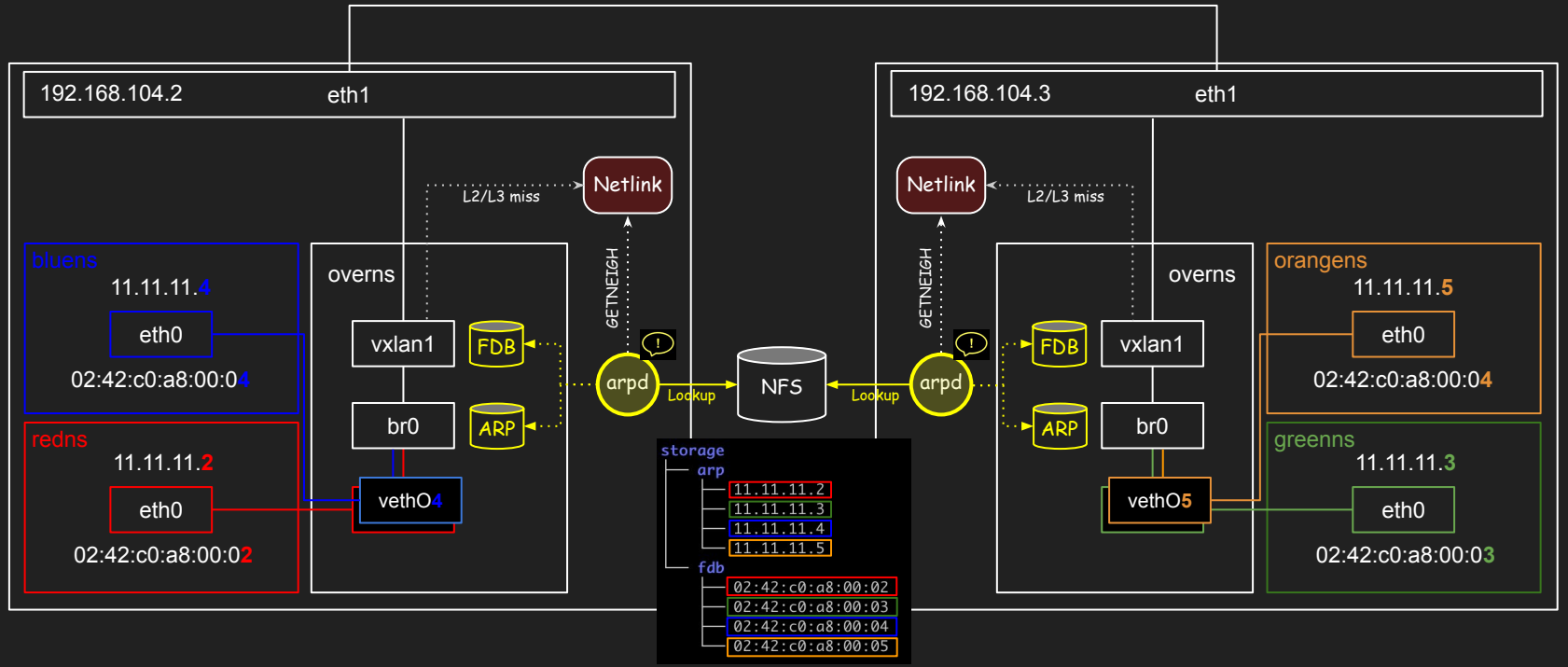
Good Luck ~ :-)

터미널 #1 (192.168.104.2)

```
# ip netns exec bluens ping -c 3 11.11.11.5
# ip netns exec bluens ping -c 3 11.11.11.3
```

터미널 #2 (192.168.104.3)

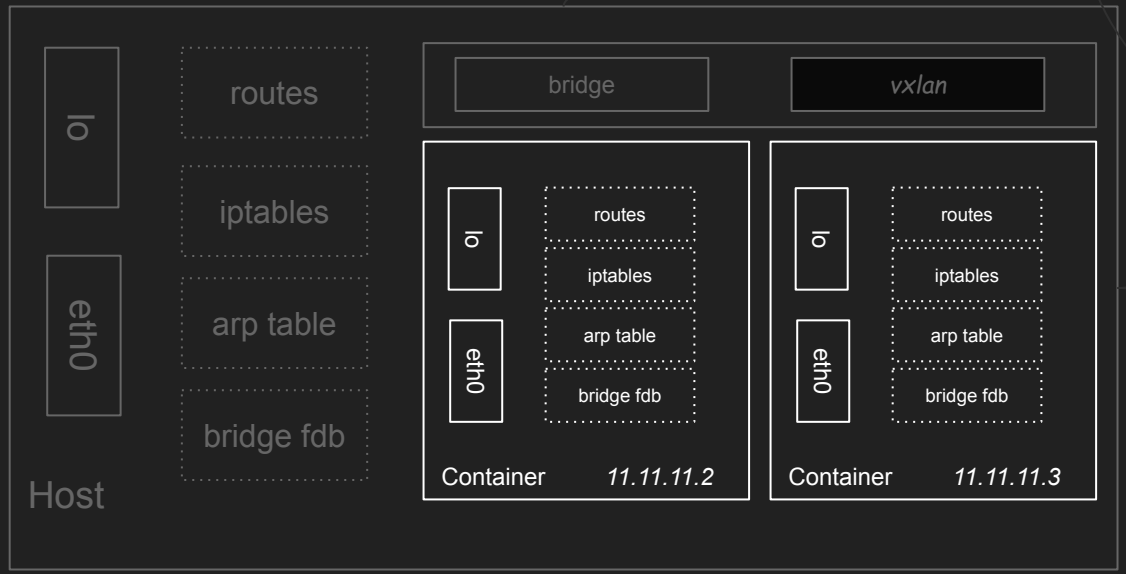
```
# ip netns exec orangens ping -c 3 11.11.11.4
# ip netns exec orangens ping -c 3 11.11.11.2
```

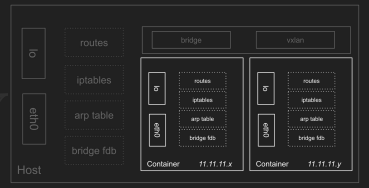


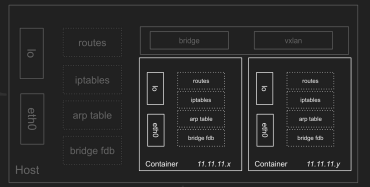
(Homework)

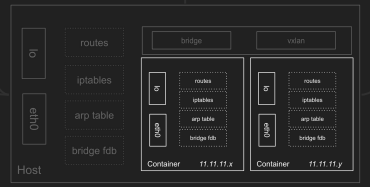
노드 (192.168.104.4)를 추가하고 해당 노드에 pinkns, skyns 를 생성하여 다른 노드의 컨테이너와 통신해보세요

이제 여러분은 분산 환경에서 가상 네트워크를 구축하실 수 있[illegible]









*오늘 배운 **vxlan**을 이용한 오버레이 네트워크 구축 방식은*
*쿠버네티스 네트워크 플러그인 중 하나인 **Flannel** 에서 사용하는 방법입니다.*

목차 보기

END