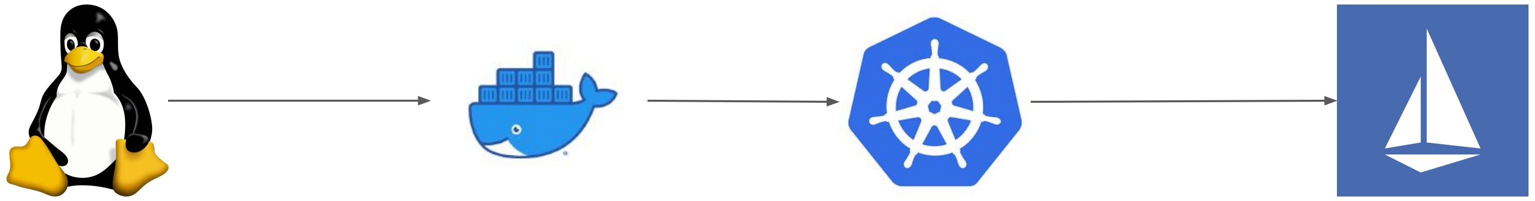
Life of a packet in k8s

from Linux network namespaces to Service Mesh



Objectives

- Learn how packets travel in a kubernetes system
- Provide a high level understanding of networking concepts in linux and kubernetes

Prerequisites

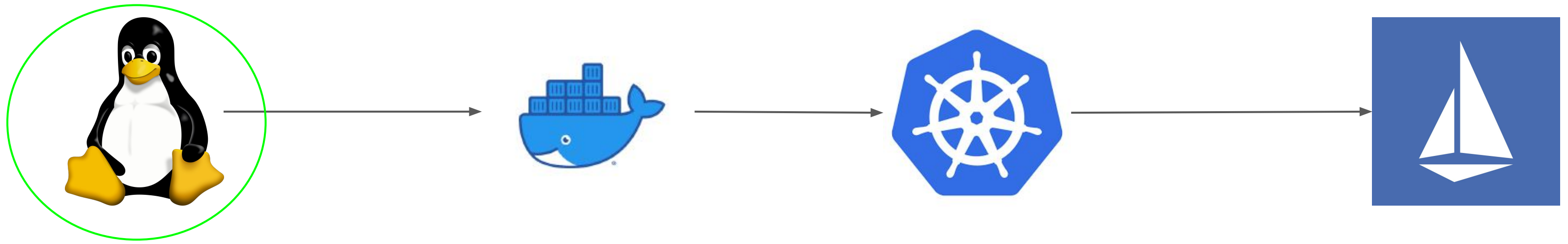
- Basic networking knowledge(ip,ARP,NAT)
- Basic docker knowledge(containers)
- Basic kubernetes knowledge(pods,services)

List of covered topics

- Well learn what are Linux namespaces
- Well learn how docker networking works
- Well learn basic kubernetes networking concepts
- Well learn what is service mesh
- Well learn how they all integrate

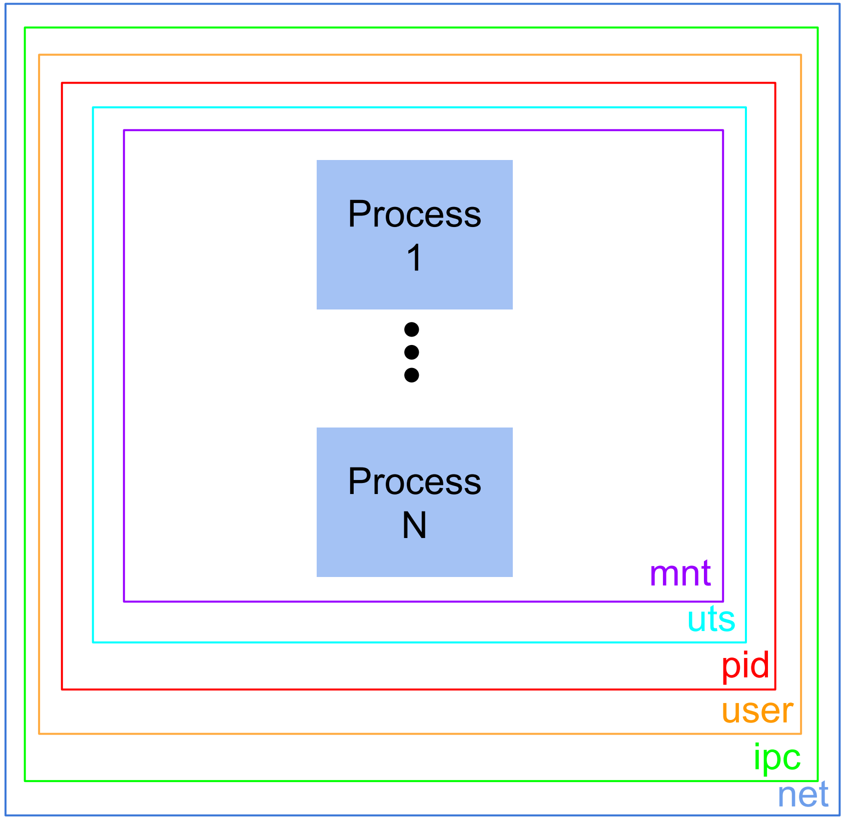
Life of a packet in k8s

from Linux network namespaces to Service Mesh



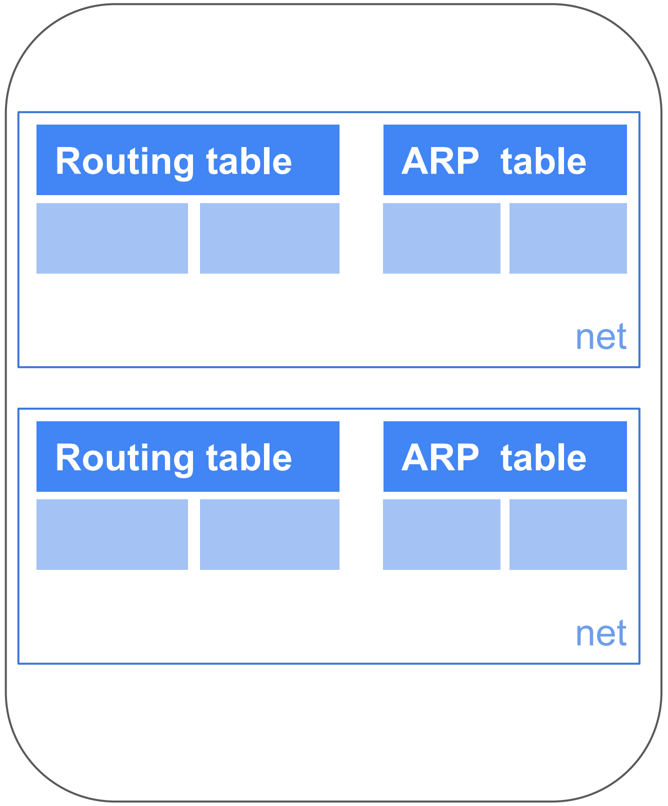
Linux namespaces

- A feature of the linux kernel
- Partitions kernel resources that processes see
- Example: the mount(mnt) namespace controls mount points
- Example: the process ID (pid) controls the process ids counting



The network namespace

- Logical copy of the network stack from the host system
- Each network namespace has a private set of IP addresses, its own routing table, socket listing, connection tracking table, firewall, and other network-related resources
- Like any other namespace network namespaces are isolated from each other

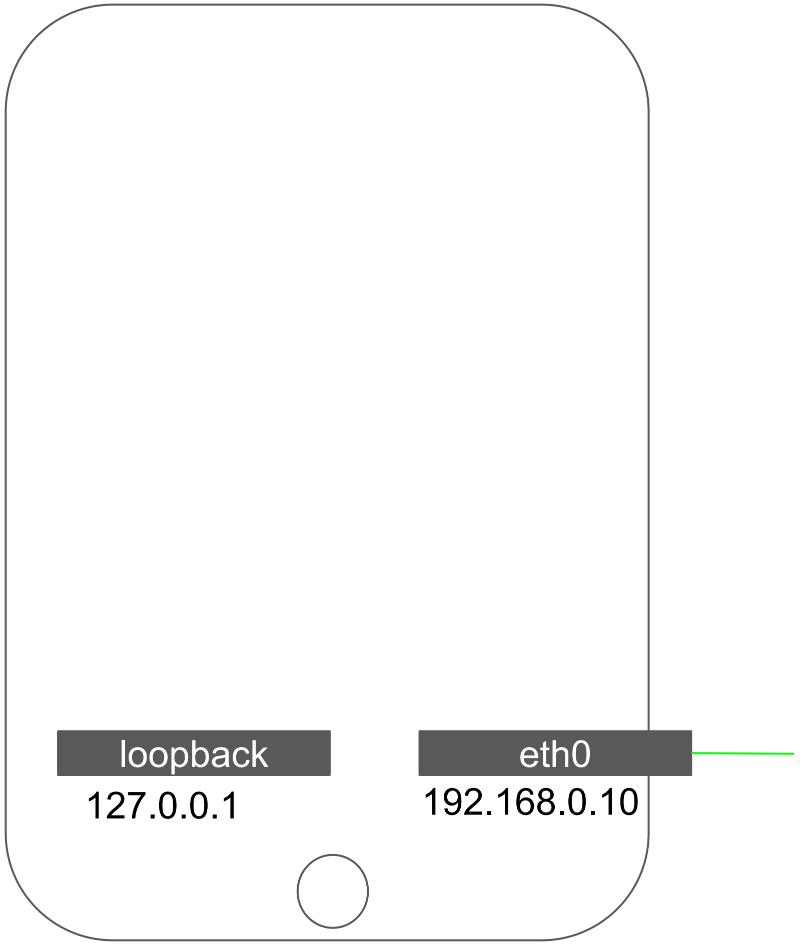


The network namespace

- We have a machine with internet

  connection
- It has 2 network interfaces:

  loopback and etho.
- Suppose we want to run program foo in an isolated environment



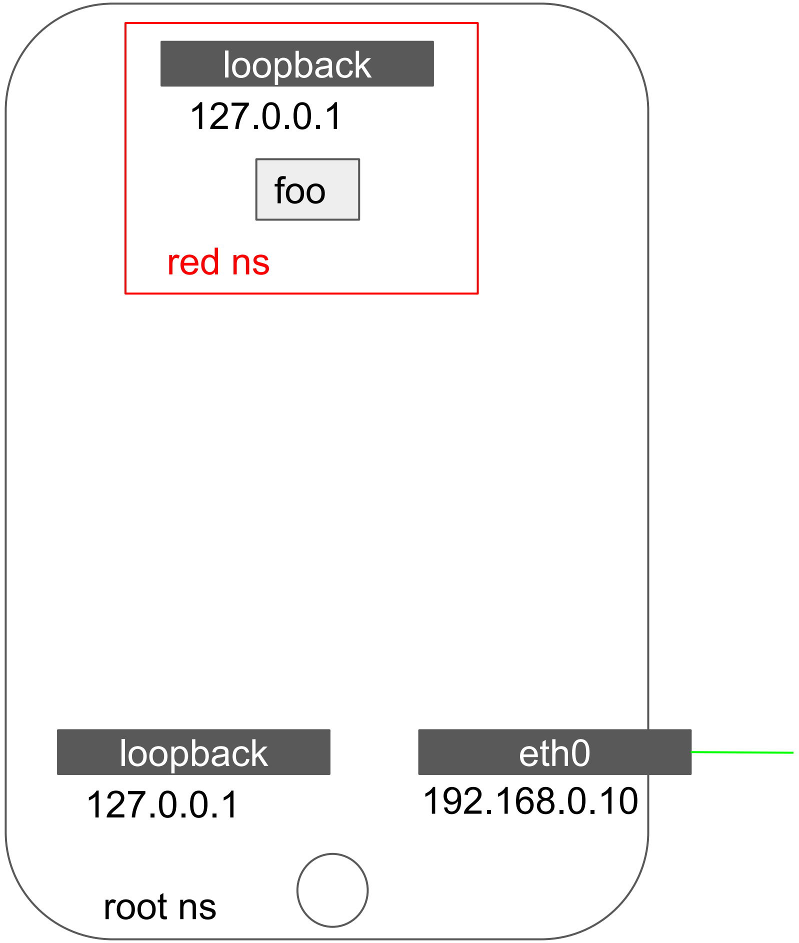
The network namespace

- We now create a new network namespace red for program foo

```
ip netns add red
```

- The created namespace does not have access to the etho interface , has its own set of arp and route tables(empty upon creation)
- Program foo runs in an isolated environment

- Job accomplished



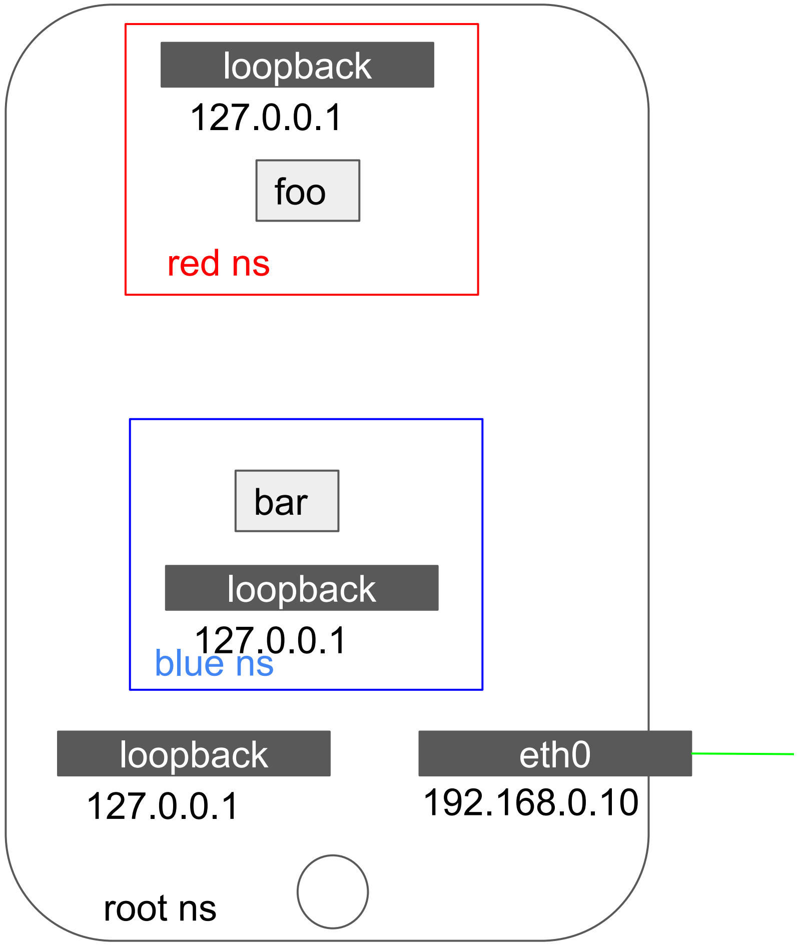
The network namespace

- Suppose we now want to run another program bar in an isolated environment.
- We now create a new network namespace blue:

```
ip netns add blue
```

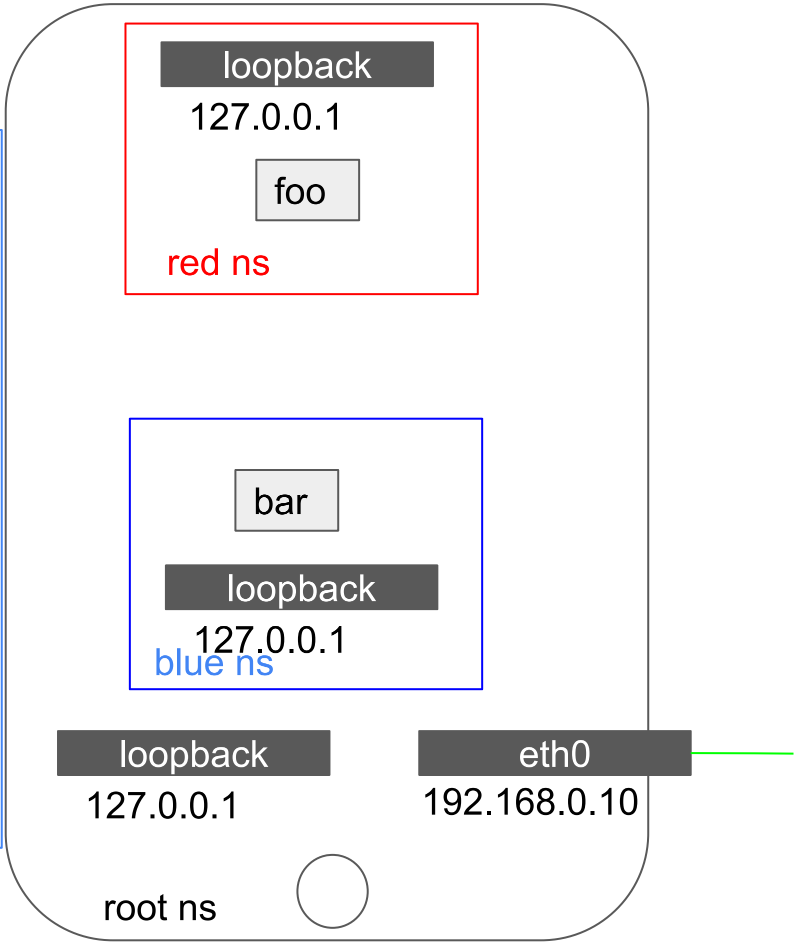
- Bar now run in an isolated environment

- Job accomplished



The network namespace

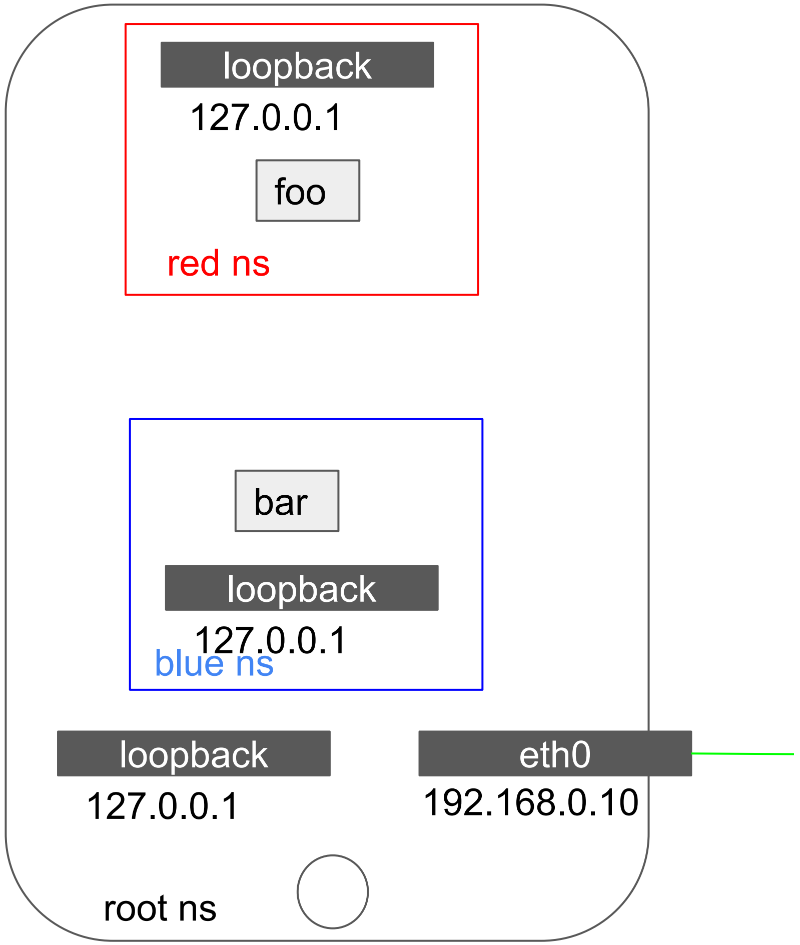
- Suppose now that we program foo needs to interact with function bar
- Since foo and bar run in isolated environments we need to connect them



The network namespace

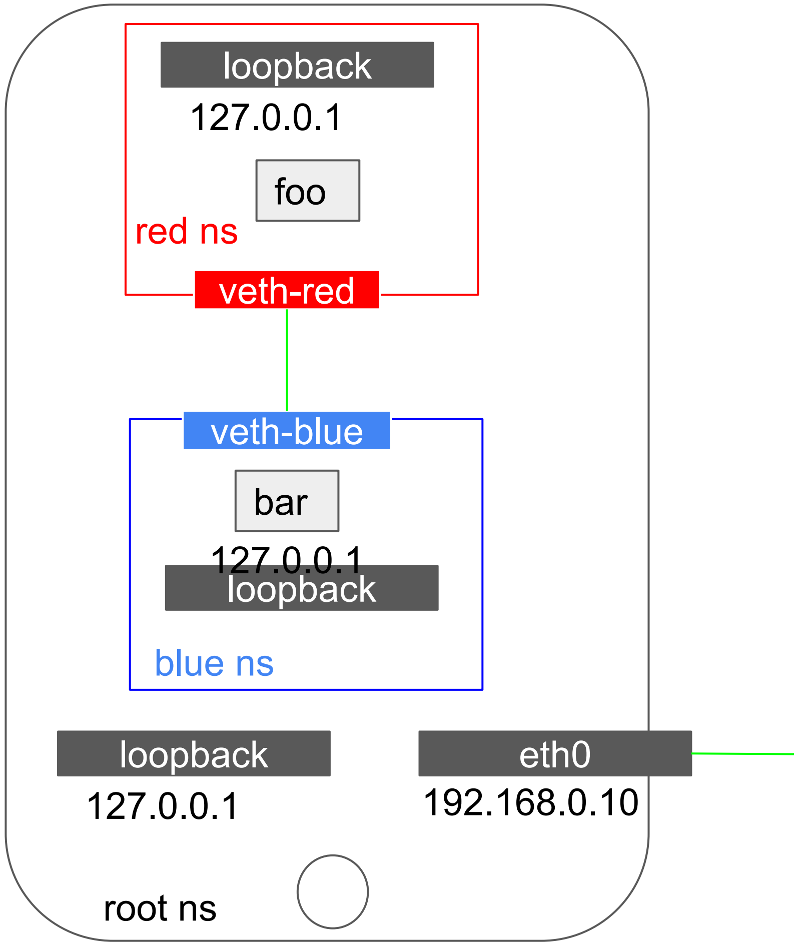
- We will connect the red and blue namespaces using a virtual ethernet pair

- Basically this means we create a virtual cable between the namespaces



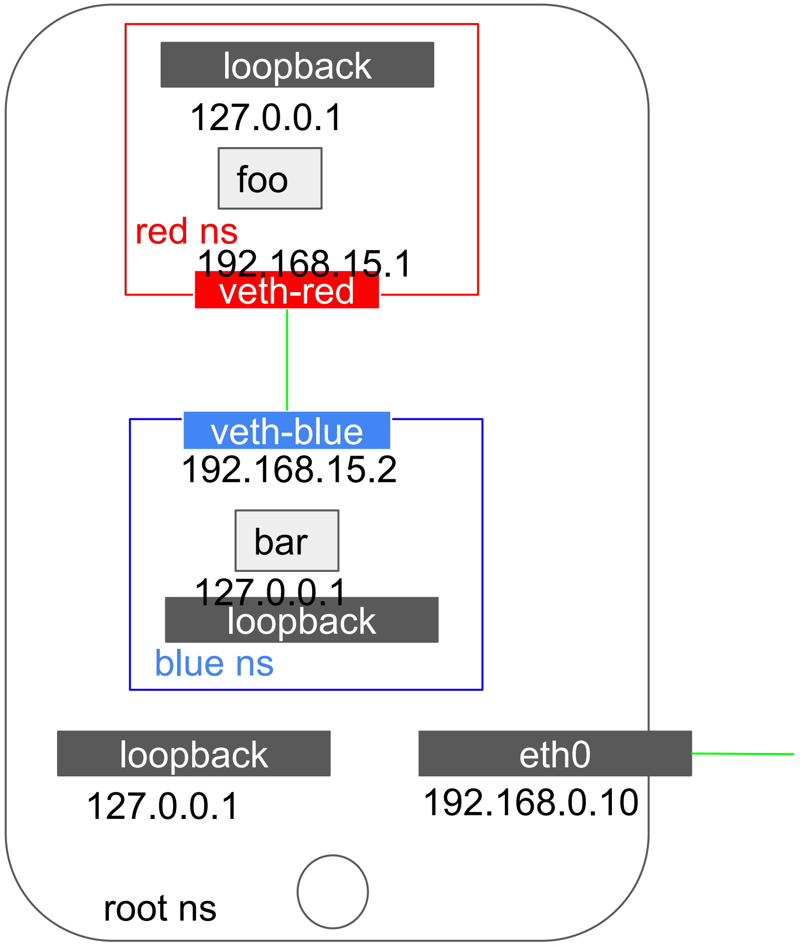
The network namespace

- We now create the virtual pair
- We attach the red interface to the red ns and the blur interface to the blue ns
- We add a veth to each namespace



The network namespace

- We assign ip address to the interfaces
- Now the foo processes can interact with the bar process



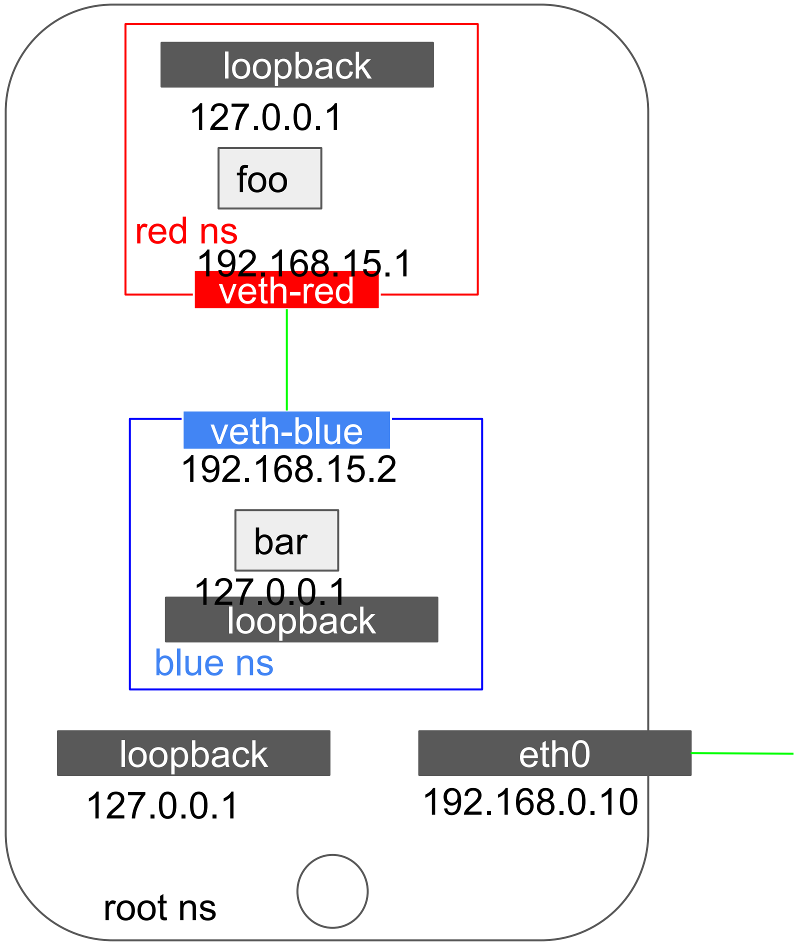
The network namespace

- If we look at the arp table in the red space:

ARP table	
192.168.15.2	ba:b0:6d:68:09:e9

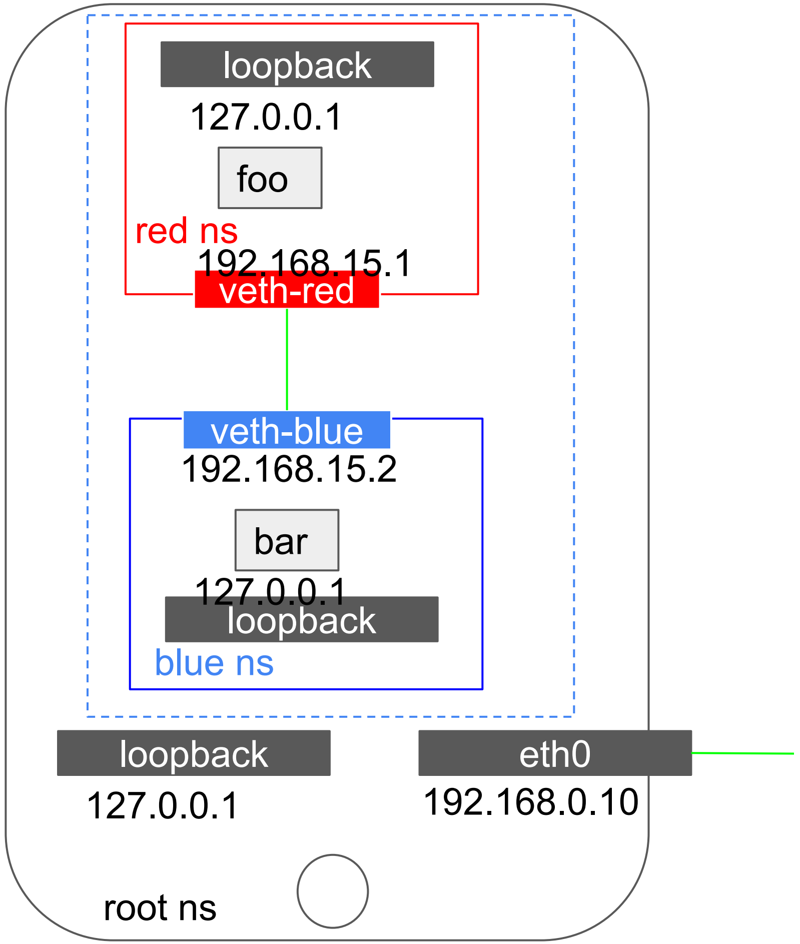
- If we look at the arp table in the blue space:

ARP table	
192.168.15.1	7a:9d:9b:c8:3b:7f



The network namespace

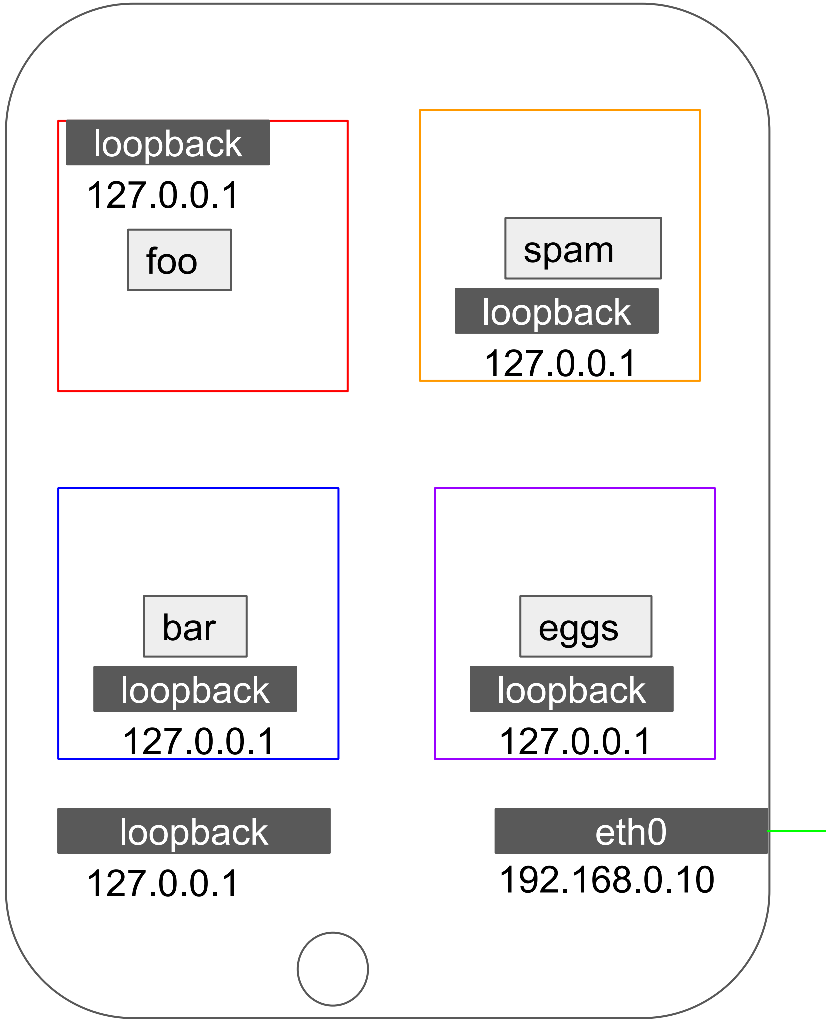
- Basically we created an internal private network



The network namespace

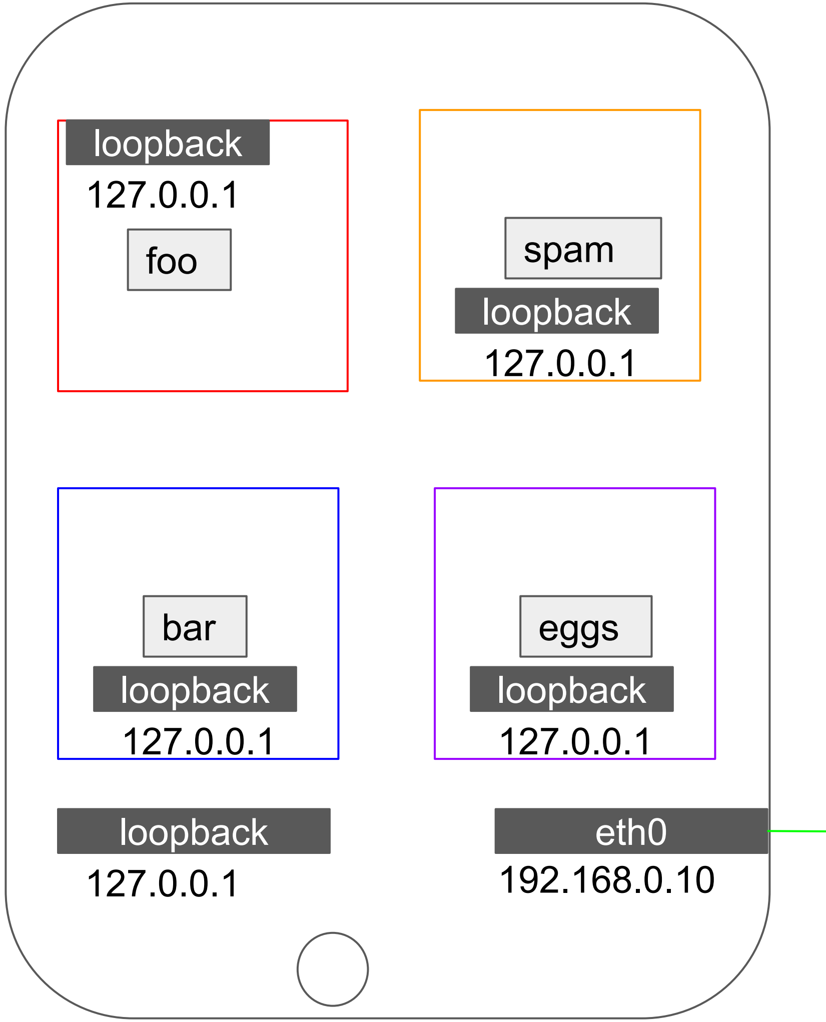
- Now suppose we have 2 more programs

  And we have the same requirement



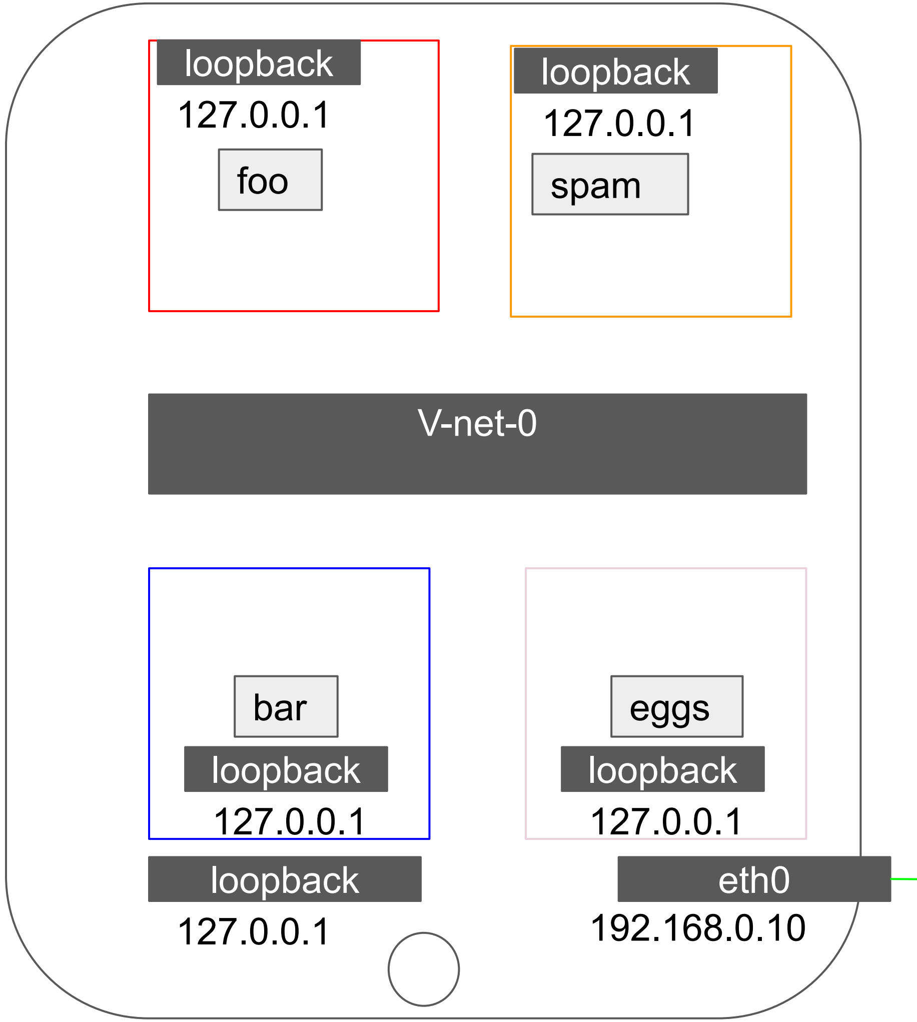
The network namespace

- We could do what we did earlier and attach a veth between each pair
- In total we would need 10 pairs!!
- And if we had N such processes we would need in total N*(N-1)/2 pair!!



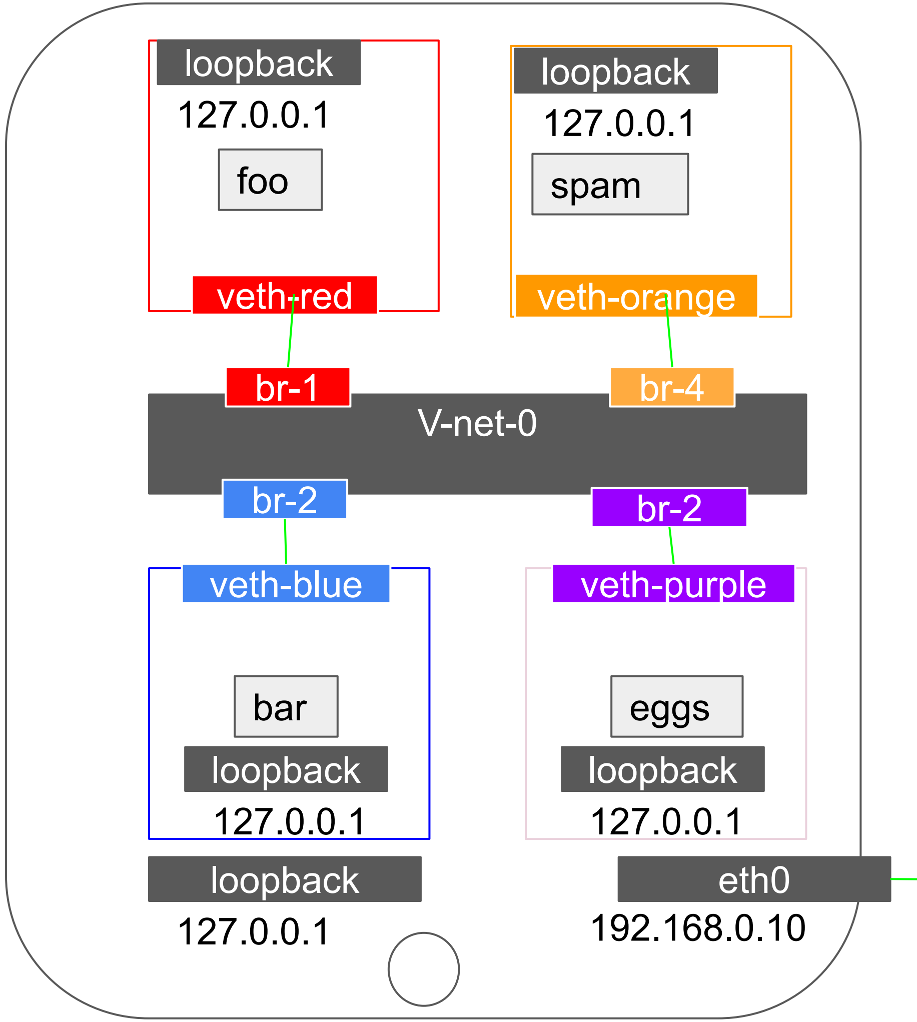
The network namespace

- Or we can create a virtual switch



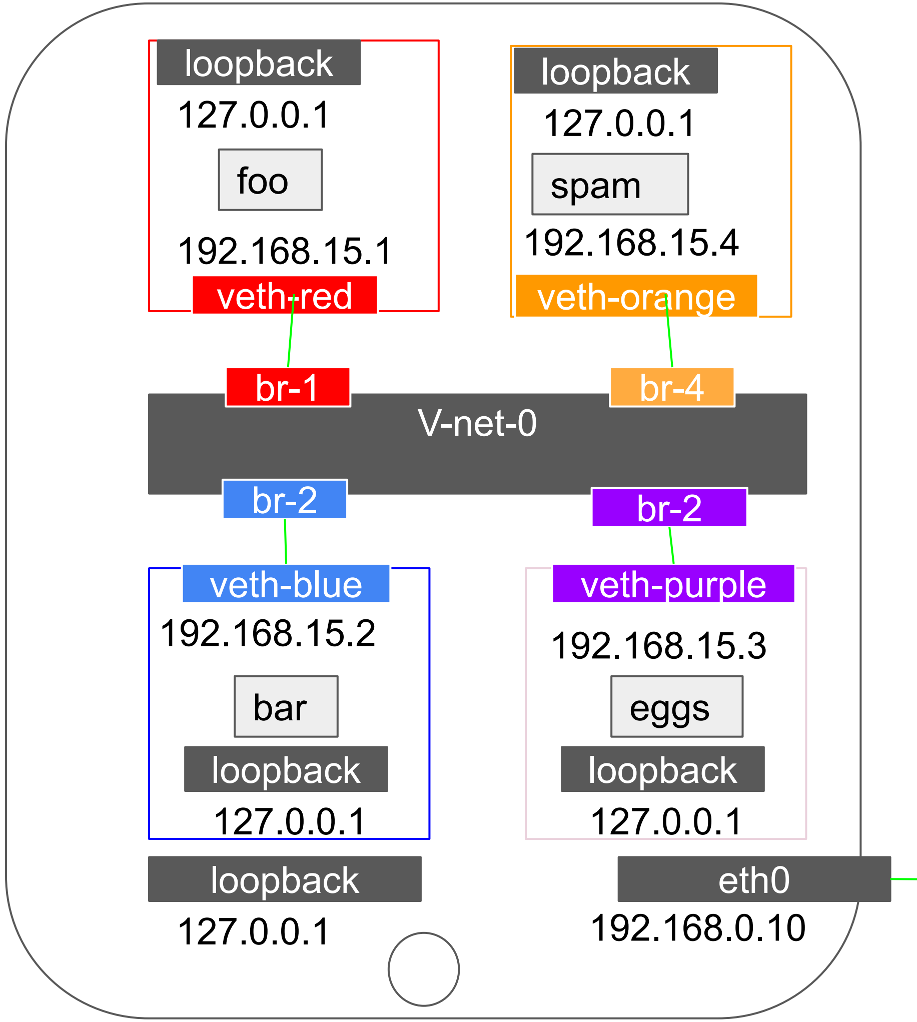
The network namespace

- Attach a veth between the switch and each network namespace
- In total we would have 4 pairs
- If we N process we would need N pairs only!!



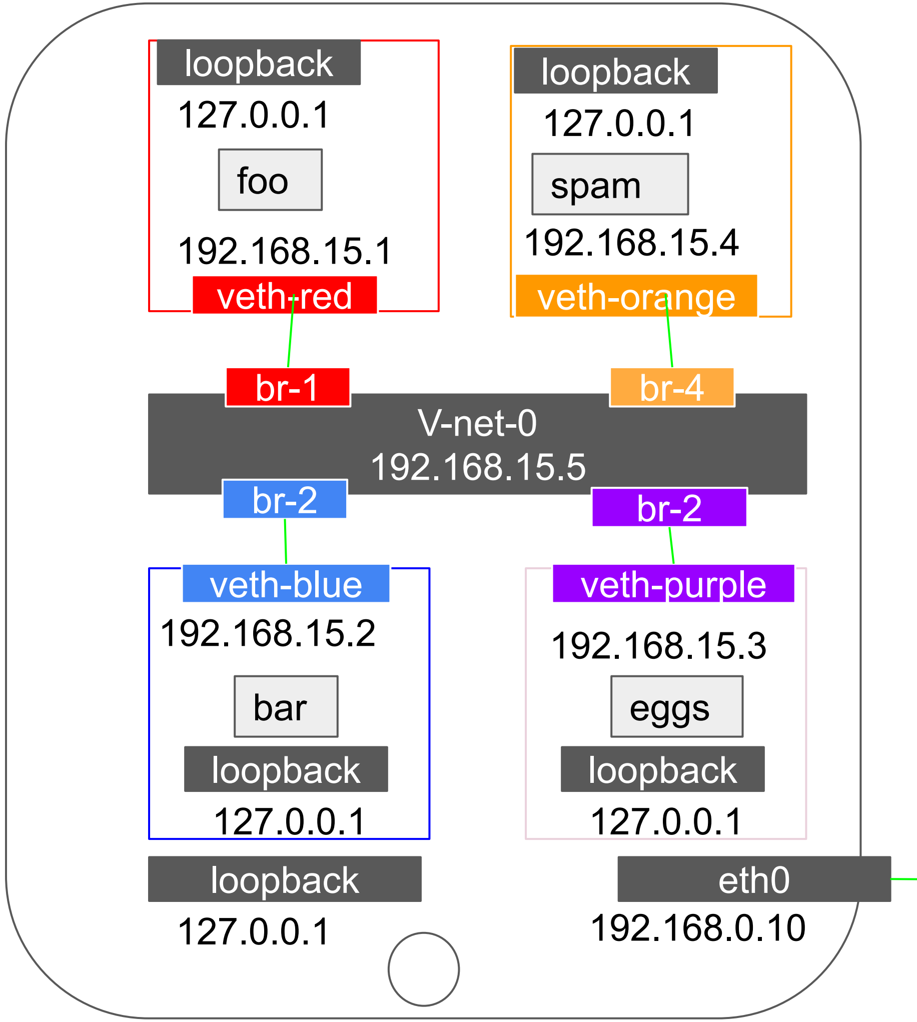
The network namespace

- Assign IP address for each veth pair



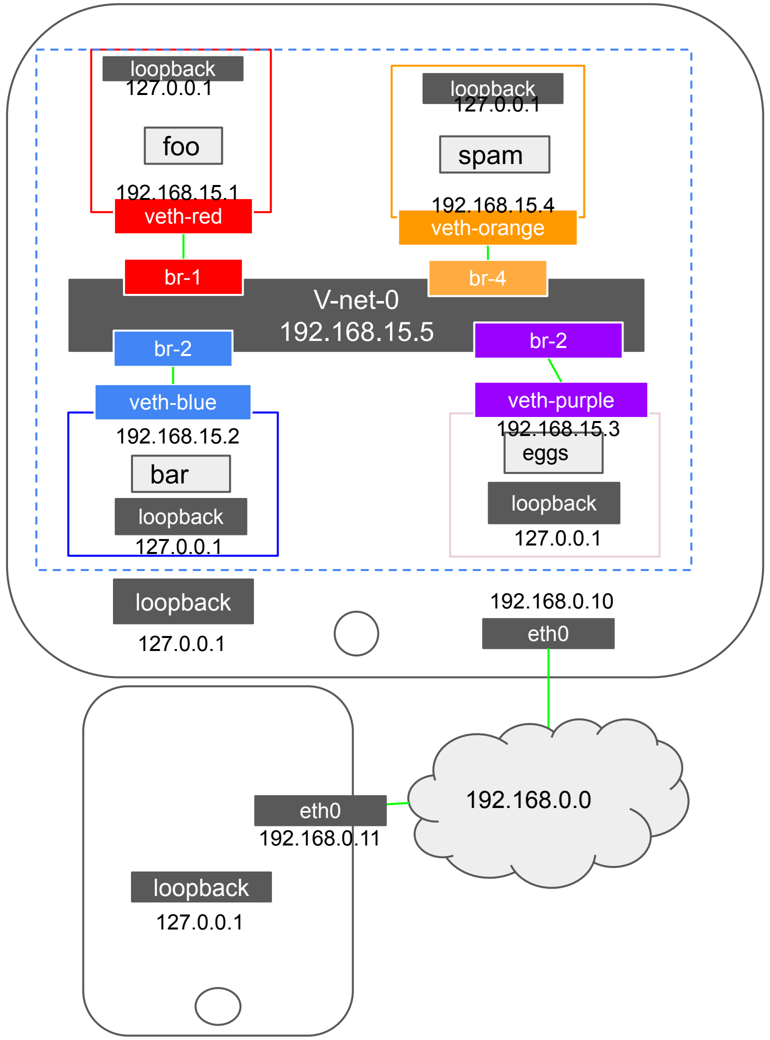
The network namespace

- Assign IP to the switch
- Now each process can communicate



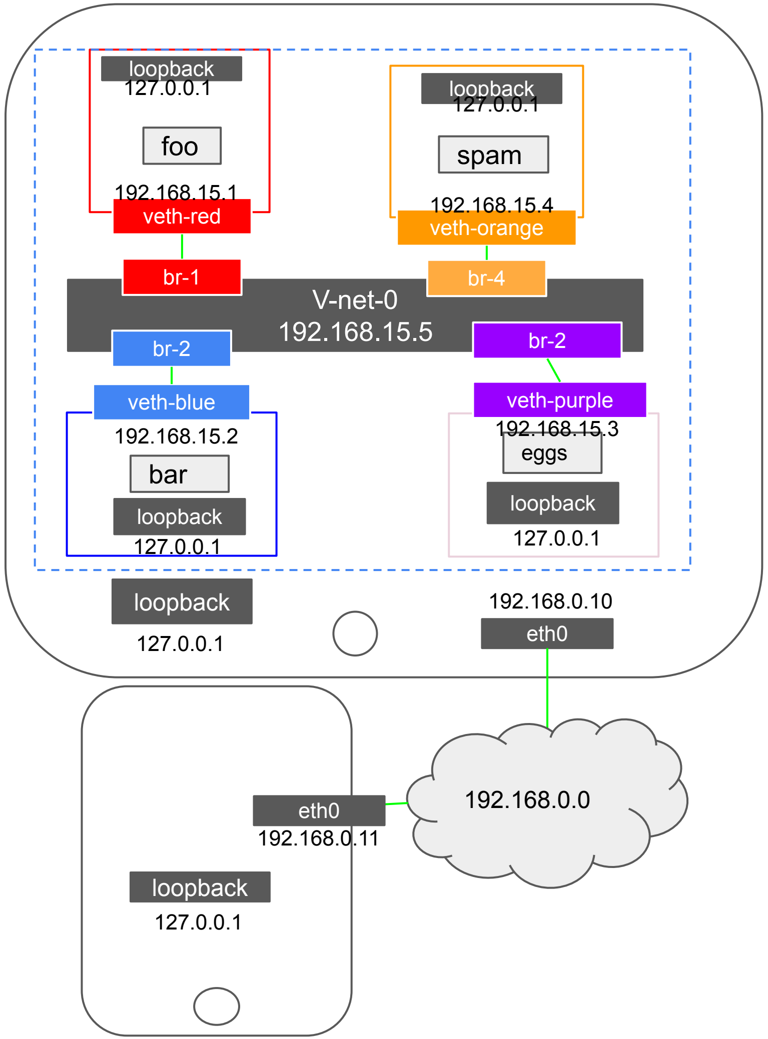
The network namespace

- Now assume that we also want to enable communication between the foo,bar,spam and eggs process and another node



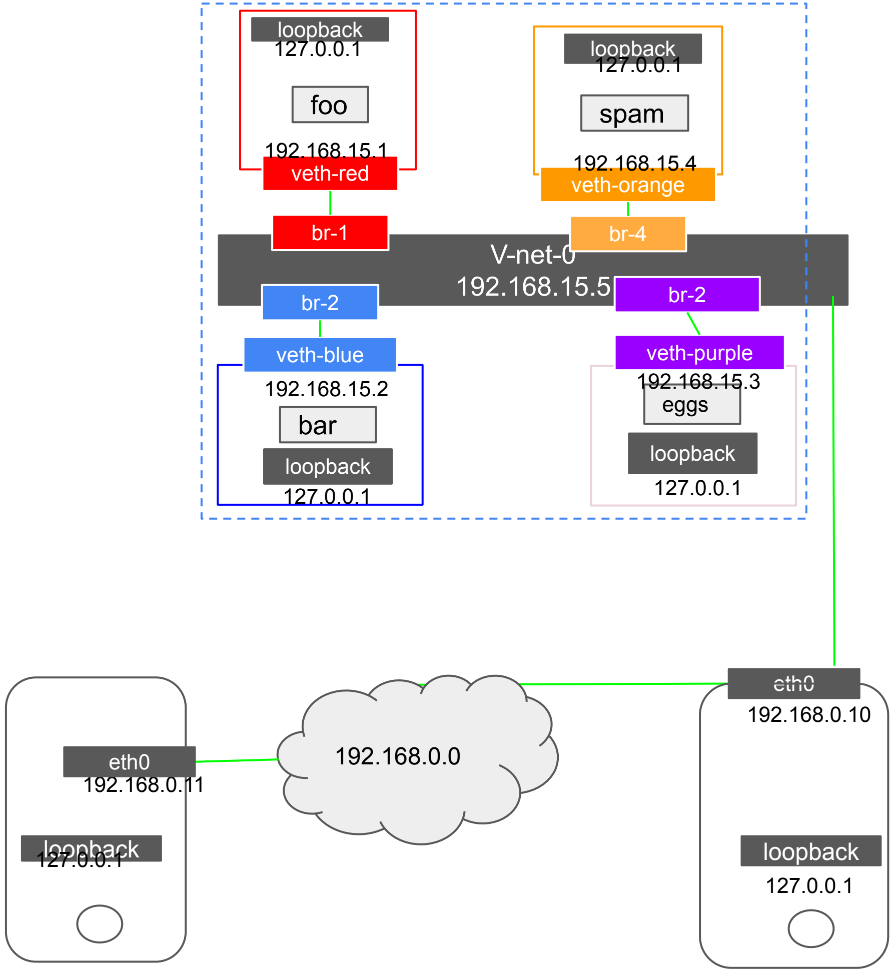
The network namespace

- As of now foo,bar,spam and eggs process run in an isolated environment
- If we tried to send a message from the bar to 192.168.0.11 we would not succeed
- foo,bar,spam and eggs run in a different local network than 192.168.0.10 and 192.168.0.11



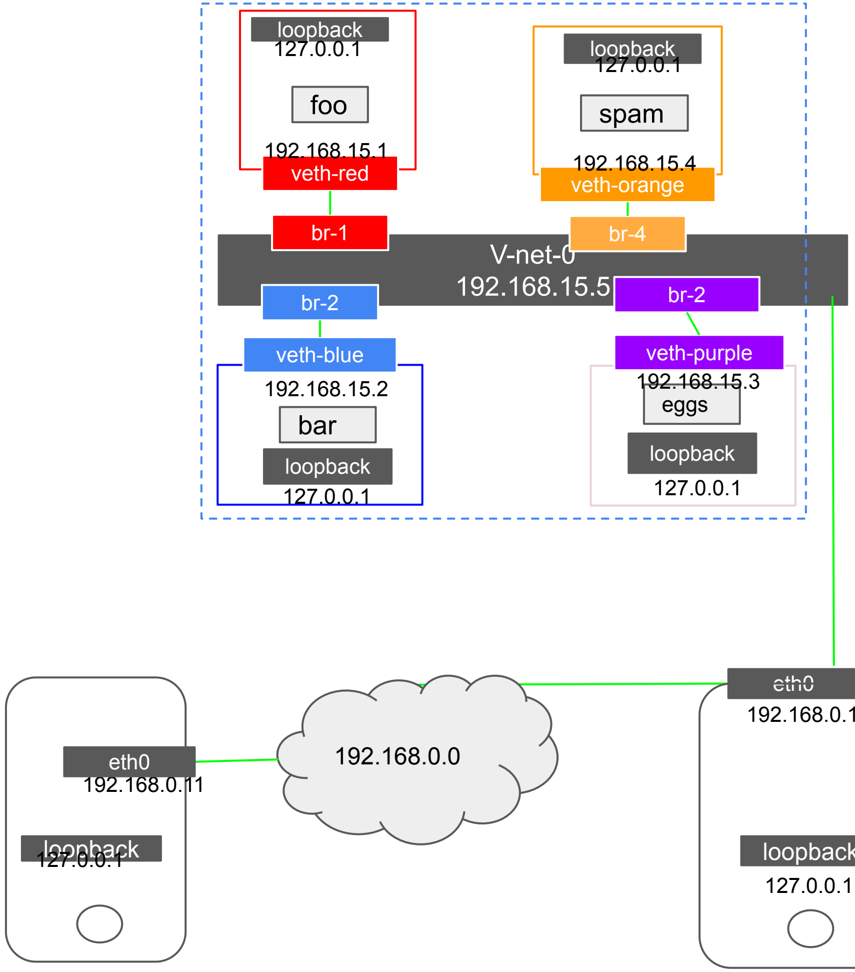
The network namespace

- We can notice that v-net-o connects between both networks
- If we add forwarding rule to the blue namespace to route all traffic to 192.168.0.0 via 192.168.15.5 then we could send messages



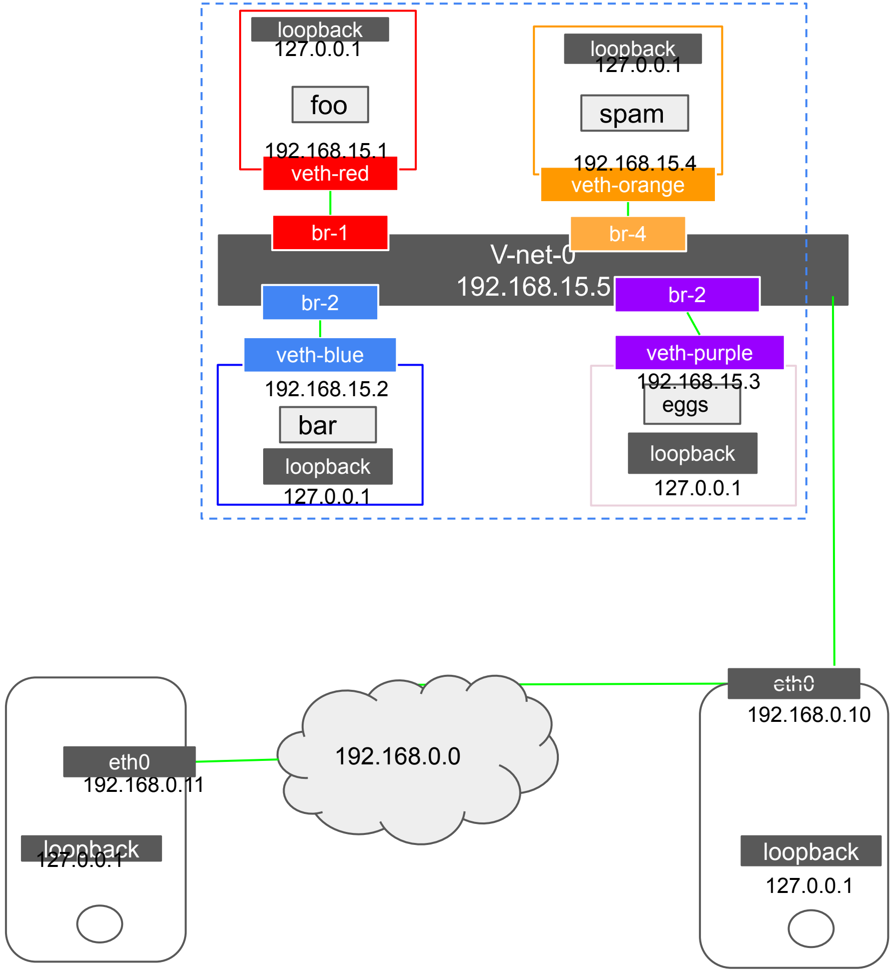
The network namespace

- However we would not receive any response as the device 192.168.0.11 is not aware of the blue network(192.168.15.0/24) which is an internal private network



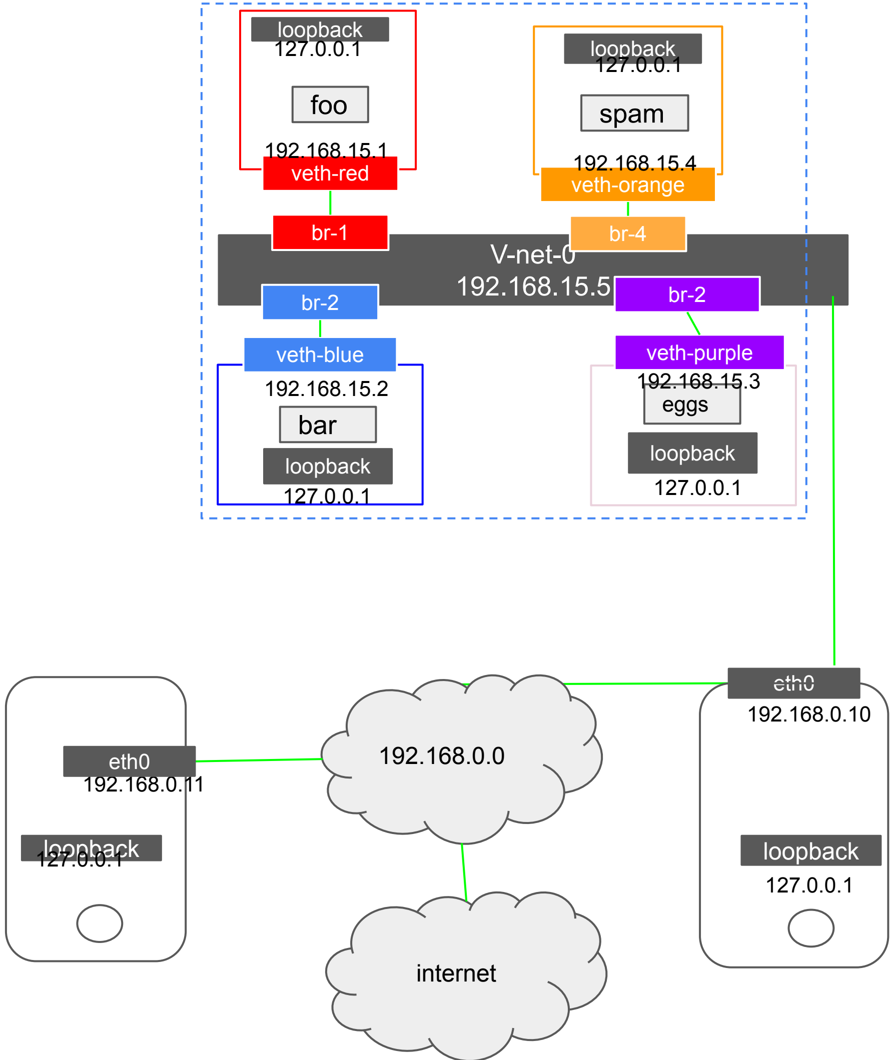
The network namespace

- To solve this problem we could add NAT rules to the ip table of the host namespace to replace all message from the 192.168.15.0/24 network to 192.168.1.2



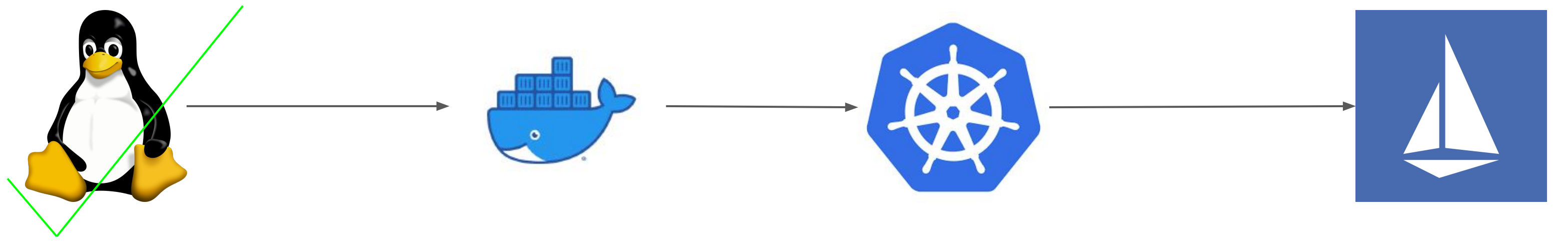
The network namespace

- Now suppose we also want to send messages from bar to the internet
- Similary, we add default ip forwarding rules to forward all traffic via 192.168.15.5



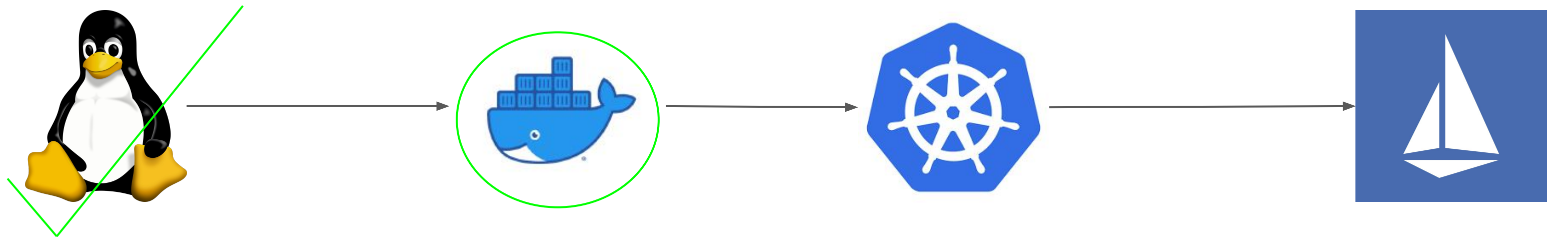
Life of a packet in k8s

from Linux network namespaces to Service Mesh



Life of a packet in k8s

from Linux network namespaces to Service Mesh

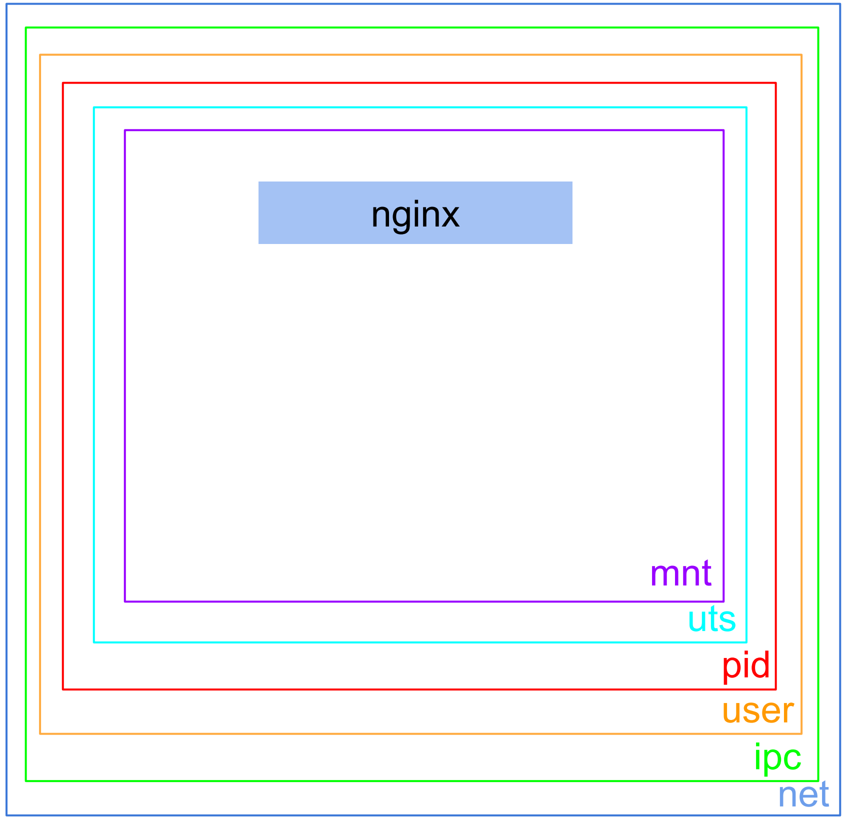


cgroups

- Short for control groups
- A feature of the linux kernel that limits accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.

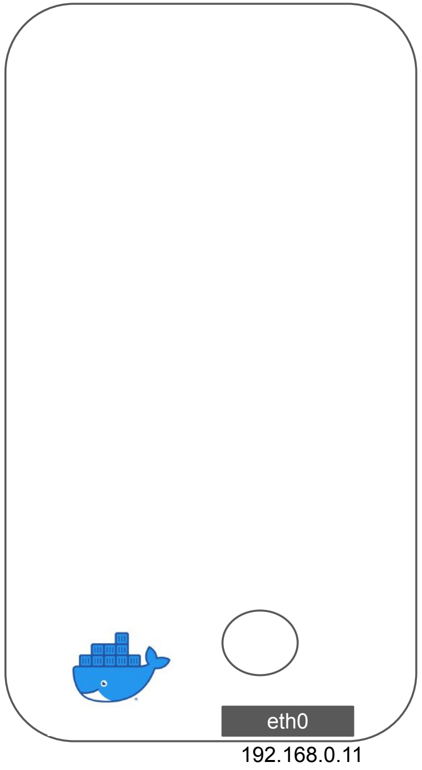
“Containers” what are they

- Not a “real” object in the Linux OS
- A group of user space process that run in 6 namespaces isolated form the host namespaces with different cgroups



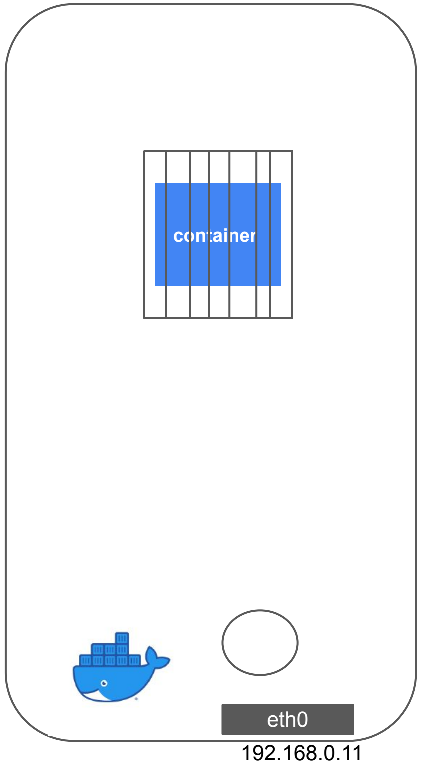
Docker networking

- Docker supports couple of different networking options



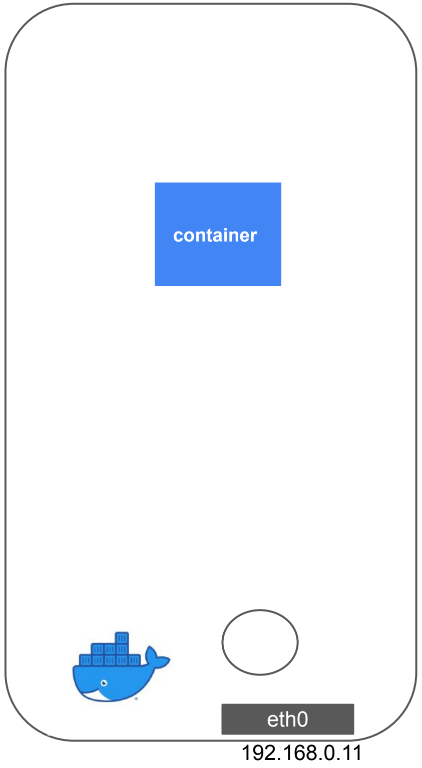
Docker networking-option 1

- In the “None” option the docker container is not attached to any network



Docker networking-option 2

- In the “host” network option the container share the host’s network

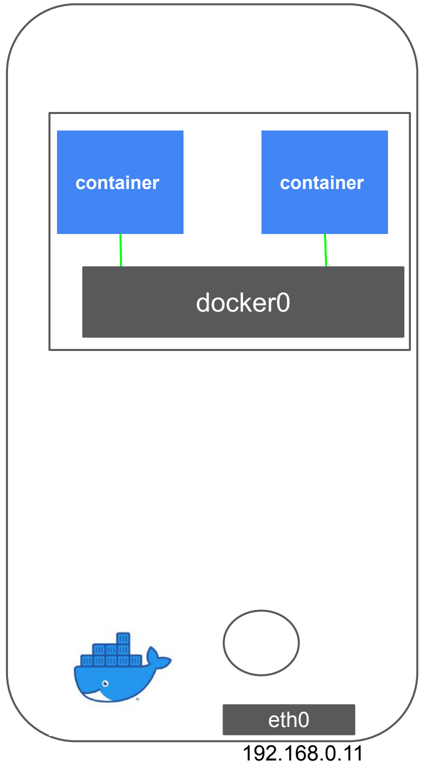


Docker networking-option 3

- In the “bridge” networking option docker create bridge
- Docker creates a bridge called docker0

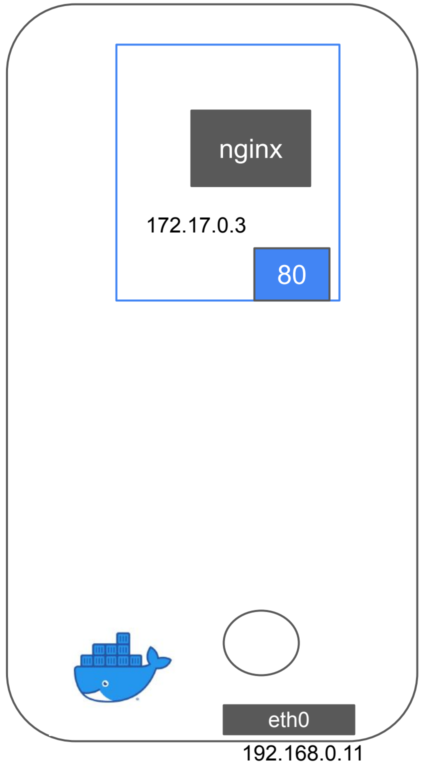
  Which all containers connect to

- Basically in the mode the setup is the same as in the previous section



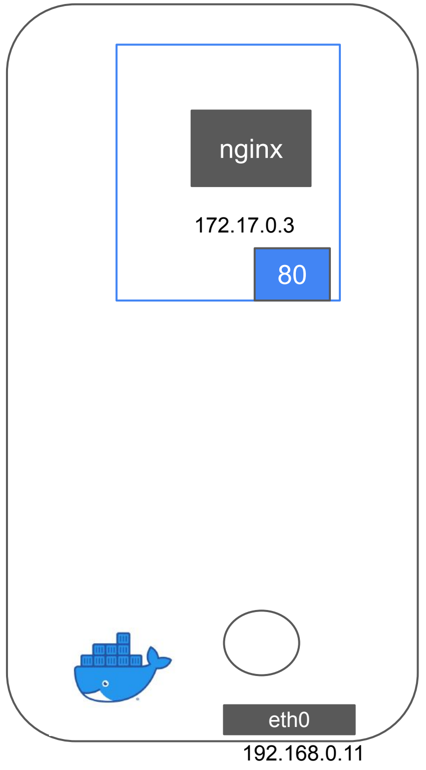
Docker port mapping

- Docker allows mapping ports inside the container to ports on the docker host.
- For example let's look at the app nginx
- Nginx runs on port 80.
- The ip of the container is 172.17.0.3
- When we try to connect to it from the outside we cannot as it runs on an internal private network



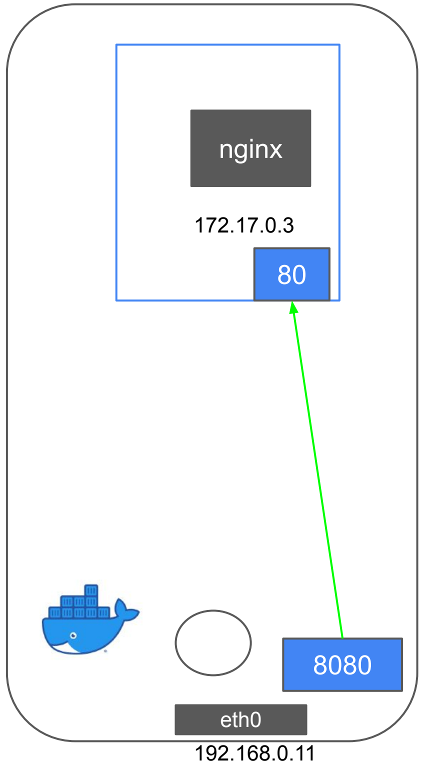
Docker port mapping

- Let's assume that we want to publish port 80 inside the container to port 8080 outside



Docker port mapping

- We add forwarding rules to the iptables of the host such that requests to port 8080 on the hosts ip 192.168.0.11 are forwarded to port 80 on ip 172.17.0.3



Container network interface(CNI)

- As we saw earlier the bridge network mode in docker works similarly to the linux bridge internal private network that we setup in previous section

Container network interface(CNI)

- Similar modes exists in different containerization programs such as rkt,mesos and kubernetes

Container network interface(CNI)

- They all create the bridge network using the same general steps
- Therefore a set of standards was developed called the CNI that all those mentioned solution implement(except docker)
- In the case of docker it implements cmm
- Kubernetes uses the none setup on docker and then implements the cni standards on docker form the none configuration on its own

1. Create network namespace

2. Create Bridge Network/Interface

3. Create VETh Pairs(Pipe,virtual Cable)

4. Attach vEth to Namespace

5. Attach other vEth to bridge

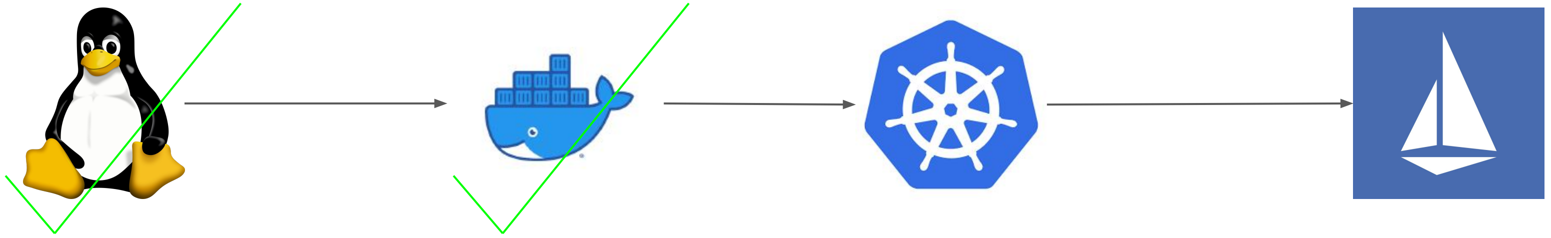
6.Assign IP Addresses

7. Bring the interfaces up

8. Enable NAT- IP Masquerade

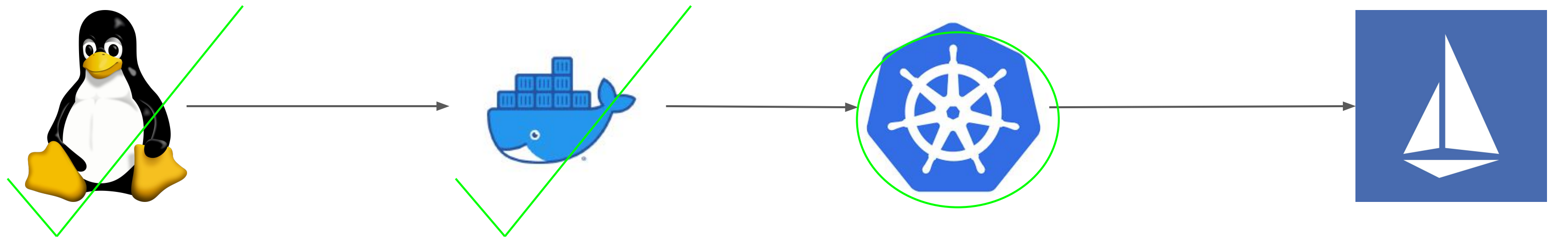
Life of a packet in k8s

from Linux network namespaces to Service Mesh



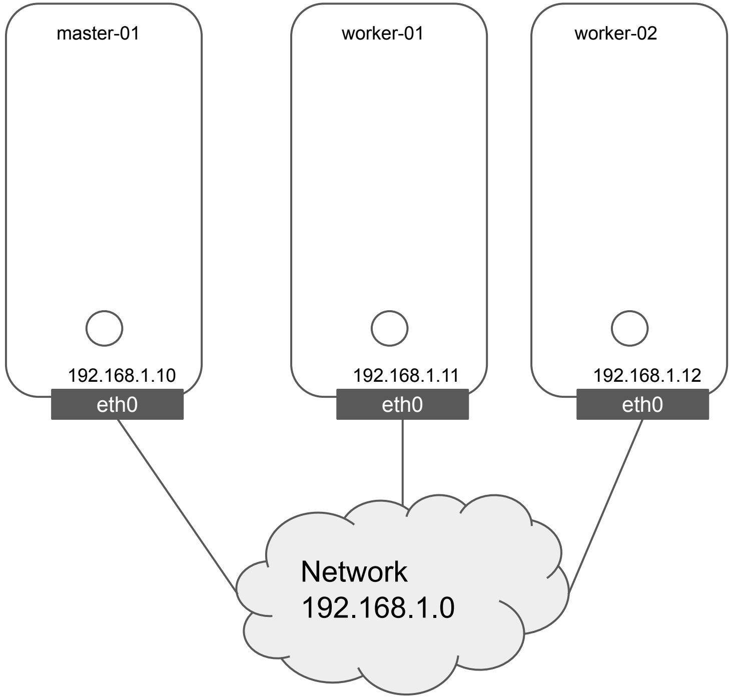
Life of a packet in k8s

from Linux network namespaces to Service Mesh



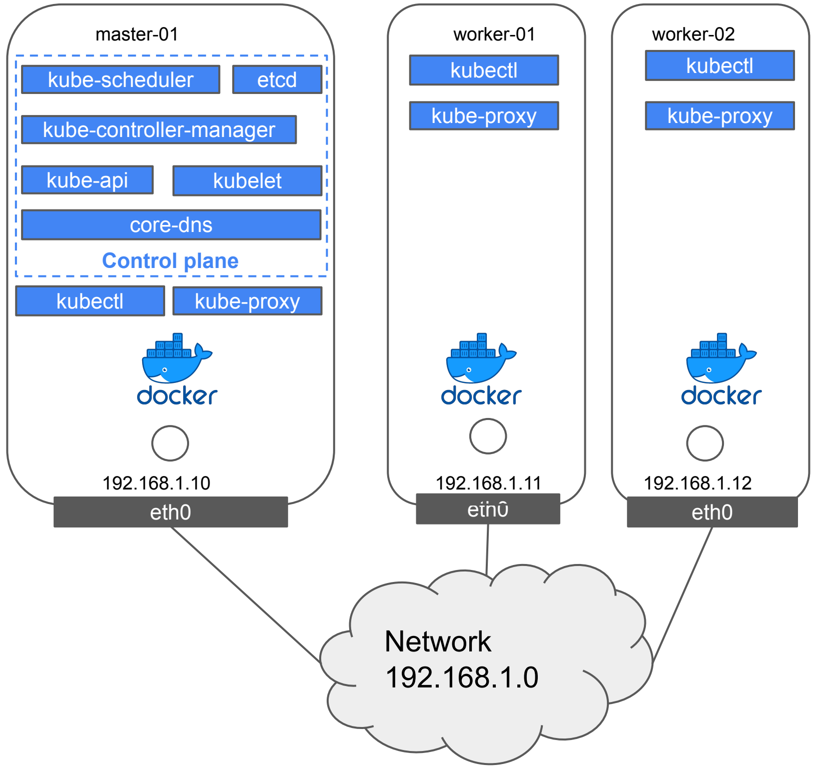
Kubernetes cluster networking

- Kubernetes cluster is made of a master node and worker nodes
- Each node must container an interface connected to a local network
- Each node must have a unique hostname,ip and mac address



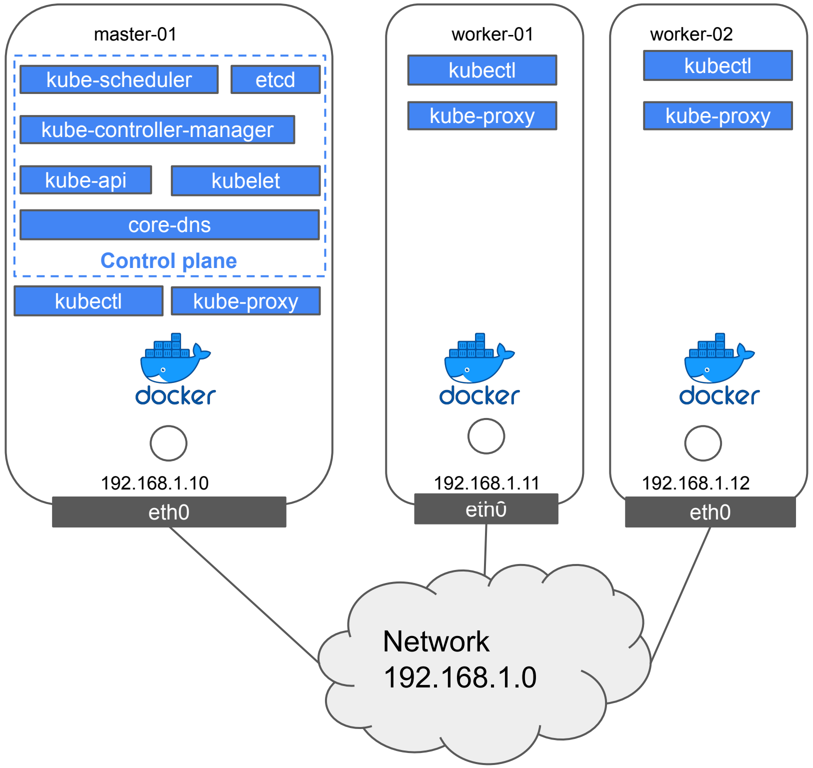
Kubernetes cluster networking

- Each node contains the kube-proxy and kubectl components
- The master node also contains the control plane which is composed of the kube-api,kubelet,etcd,kube-scheduler and the kube-controller-manager



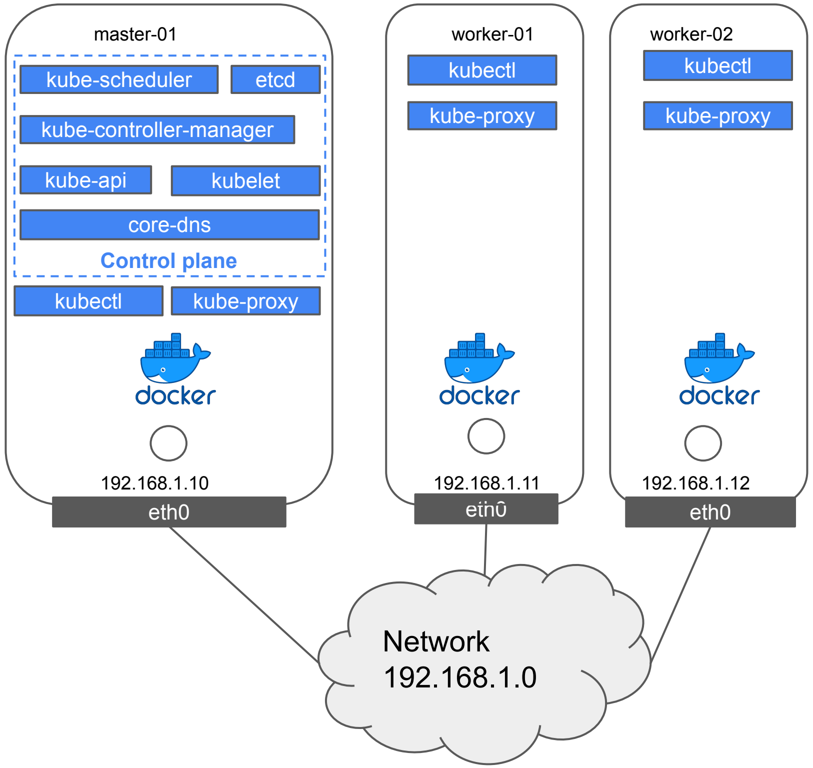
Pod networking

- Assume that we 3 linux machine
- Assume that we setup docker on each machine
- Assume that we install the kubernetes components on each machine



Pod networking

- Now all that remains is to setup the pod networking solution

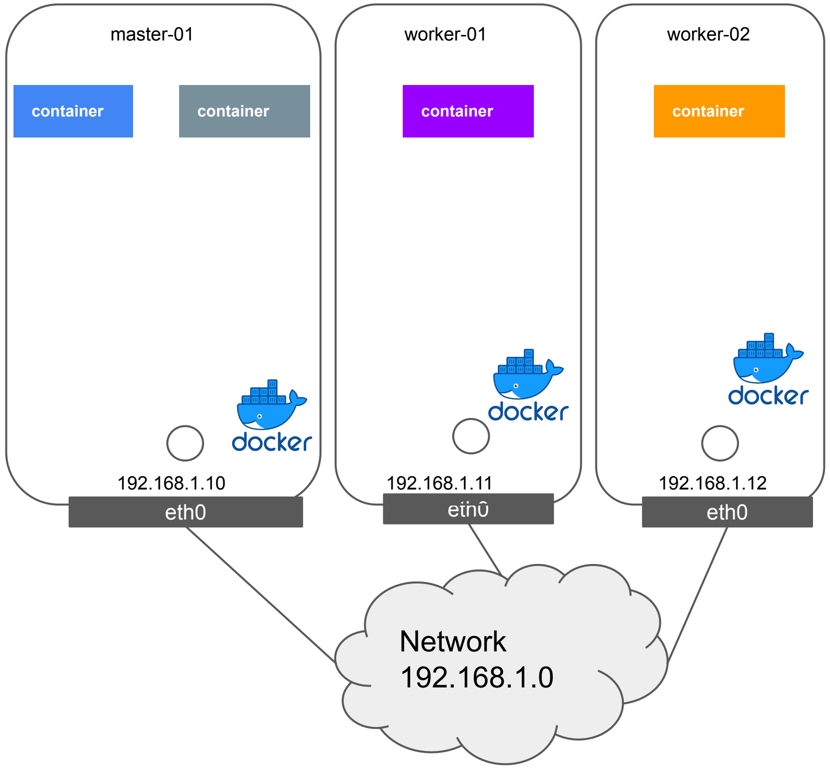


Kubernetes networking model

- Kubernetes does not have a built in component for pod networking and has to use an external solution
- In each solution the following rules are fulfilled :
  - Every pod has an IP address
  - Every pod is able to communicate with every other pod in the same node
  - Every pod is able to communicate with every other pod on other nodes without NAT

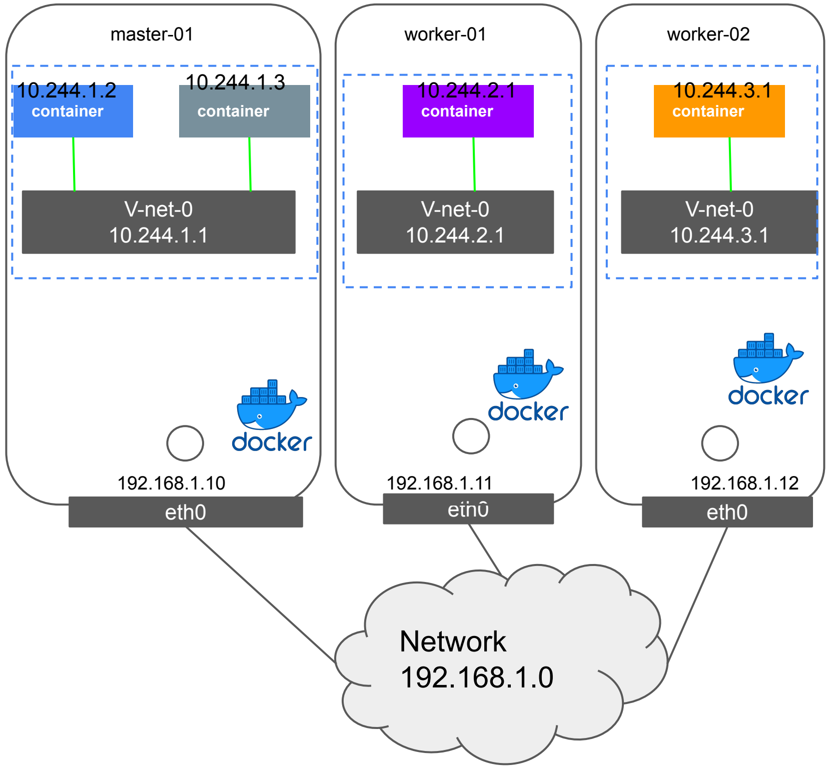
Pod networking solution from scratch

- Suppose we want to create a networking solution from scratch
- We have a 3 node cluster
- On each node we have docker containers



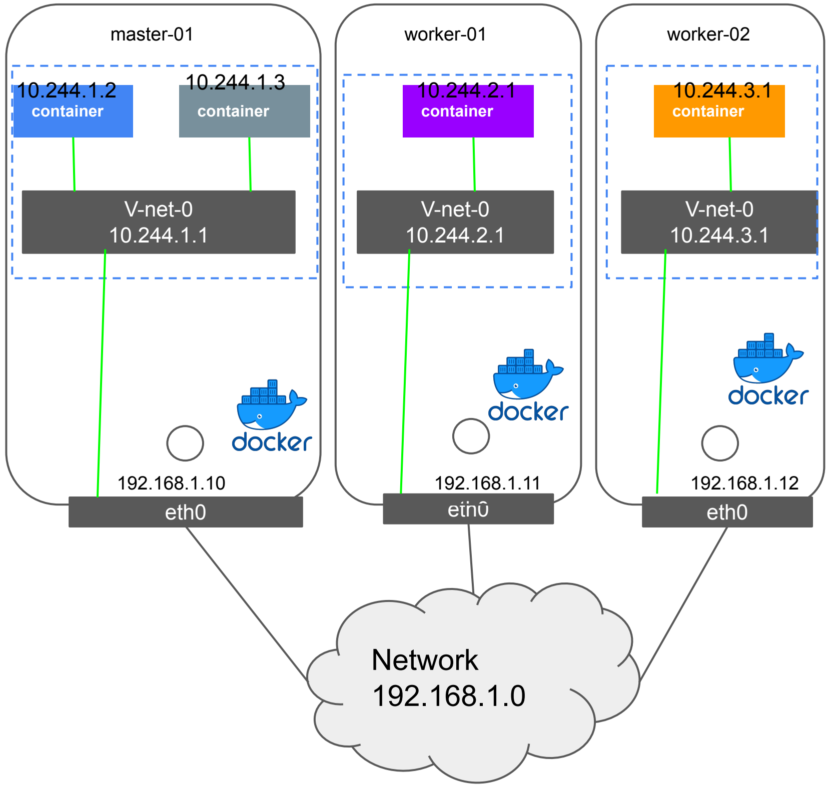
Pod networking solution from scratch

- As in the previous sections we create a linux bridge network and assign ip to each container
- Now each pod has its own ip address
- Each pod is able to communicate with every other pod in the same node



Pod networking solution from scratch

- Now we want to allow every pod to communicate with every other pod
- (First option)Similar to the previous section we add routing rules to the host routing table to forward packets to the designated network
- (Second option) we setup for each node its etho ip address as the default gateway. For example for network 10.244.1.0/24 the gateway is 192.168.1.11.And then configure a router to forward packets
- Now each pod is able to communicate with other pod in the each node



CNI plugins

- Software solution plugin to kubernetes networking needs
- Some plugins operate in layer 2 and some in layer3
- Some plugin create a network overlay

cilium

weaveworks

flannel

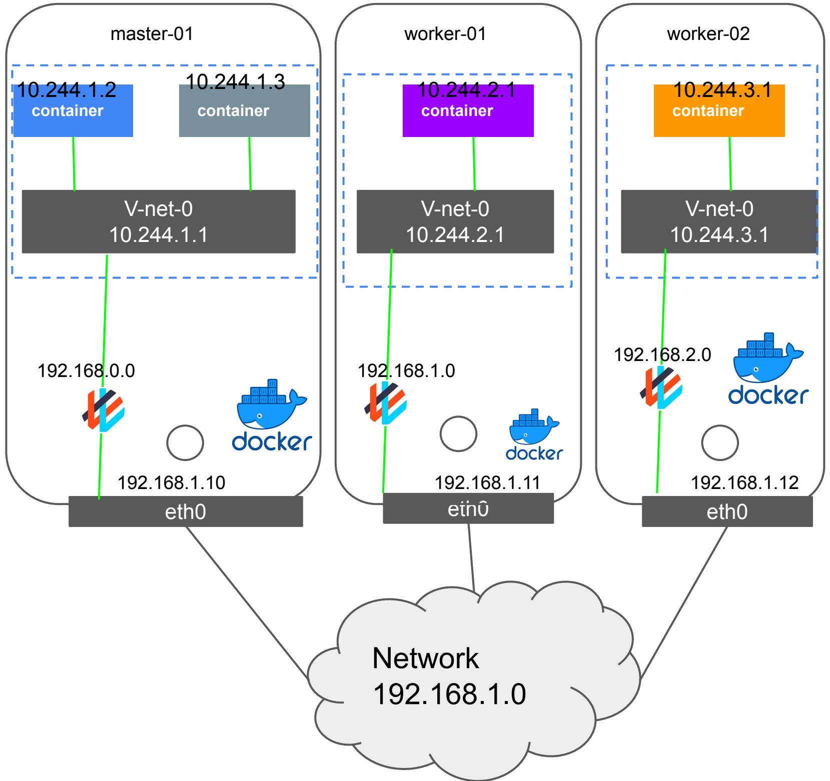
vmware NSX-T

Case study: weavenet by weaveworks

- In this section will look intro wavenet
- Wavenet is a CNI kubernetes plugin
- It is open source
- It creates an overlay network

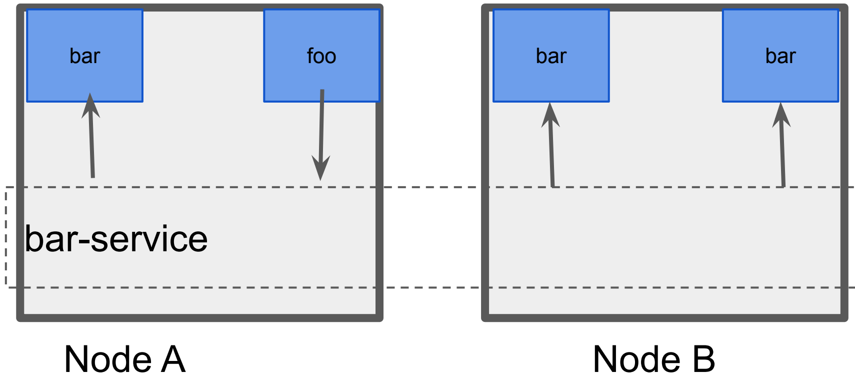
Case study: weavenet by weaveworks

- Weavenet deploys an agent on each node
- Agents communicate with each other
- Each agent stores a topology of the entire network
- Each agent intercepts packets sent from a pod to pods on another agent
- It encapsulated the packet, with new source and destination
- The packet is sent as a udp packet to the destination node
- The sent packet is then received by the agent in the destination node,decapsulated it and forwards it to the correct pod



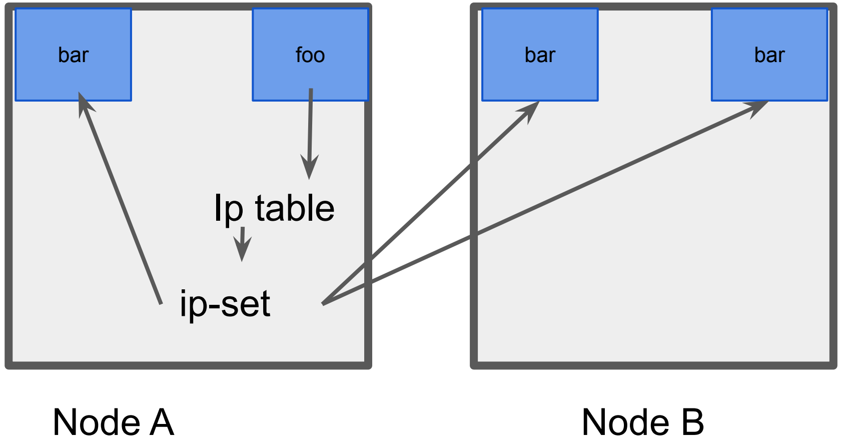
Kubernetes Services

- An abstract way to expose applications running on a set of pods as a network service
- Provides persistent endpoint to the client through virtual IP
- Provides dns address as an endpoint
- Load balances the the backend pods
- There are several types of services:
  - ClusterIP
  - Load Balancer
  - NodePort
- Services are created using the kube-proxy component



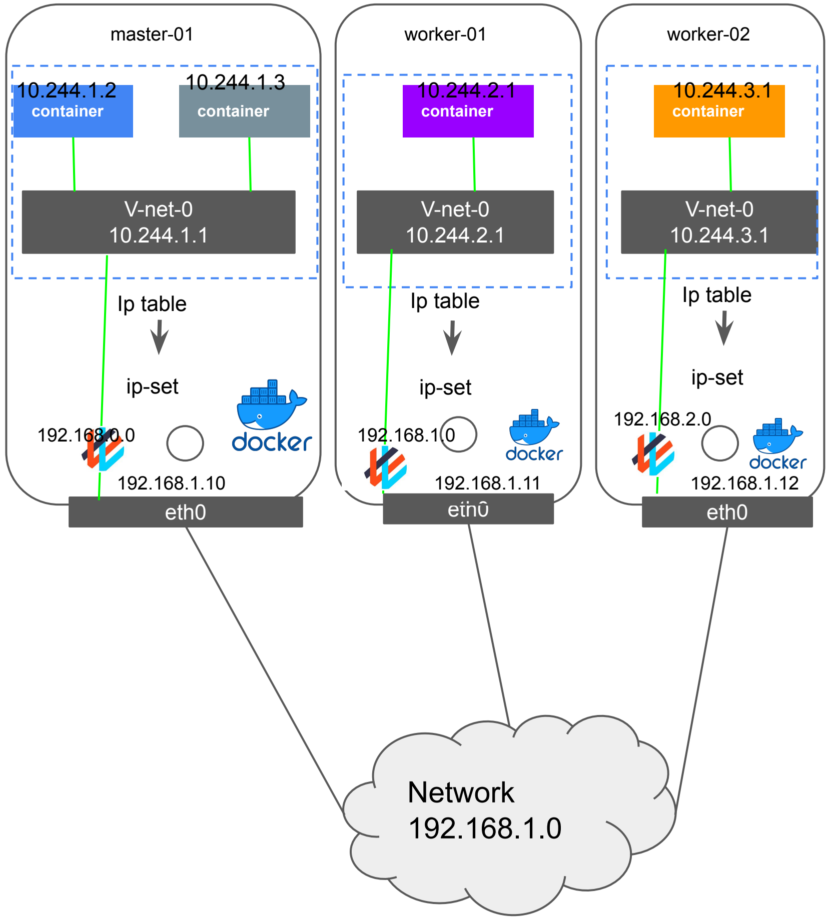
Kubernetes Services

- Selecting pods for a specific service in fact contains two steps:
  - IP resolution from the kube-DNS server
  - IP forwarding using the hosts ip tables
- The IP table is connected to an ipset which randomly translated the service ip to and ip of a pod in the service
- Then it forwards the packet to correct pod



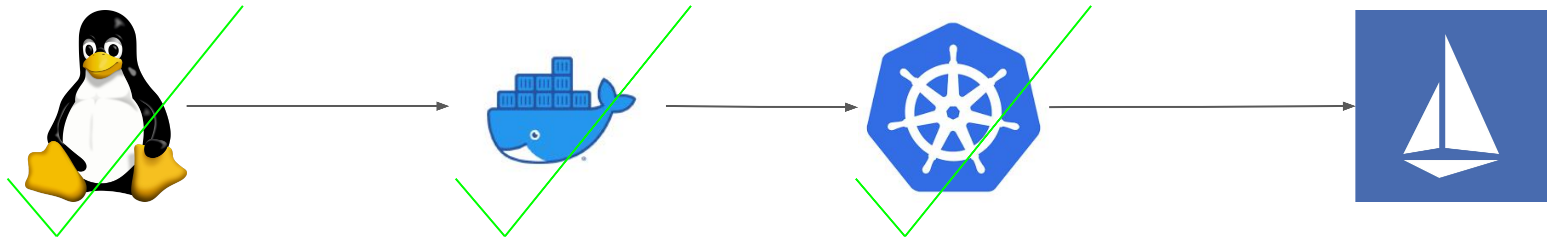
Services in kubernetes

- Service are in fact contain two steps:
    - IP resolution from the kube-DNS server
    - IP forwarding using the hosts ip tables
- The IP table is connected to an ip set which randomly translated the service ip to and ip of a pod in the service
- Then it forwards the packet to correct pod



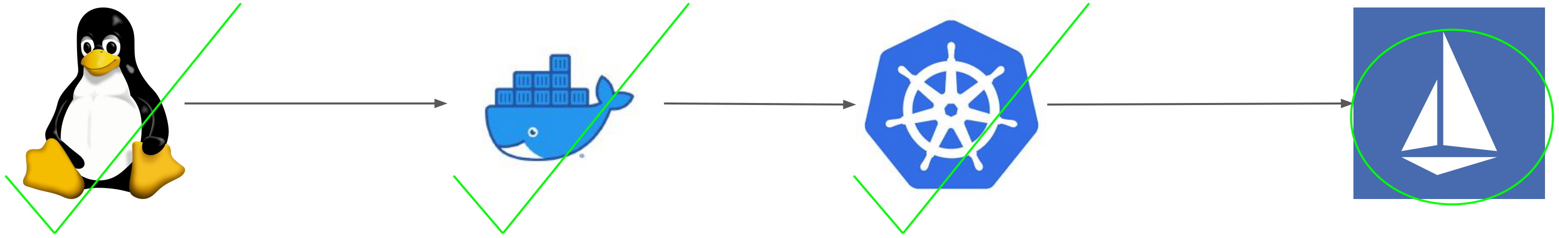
Life of a packet in k8s

from Linux network namespaces to Service Mesh



Life of a packet in k8s

from Linux network namespaces to Service Mesh

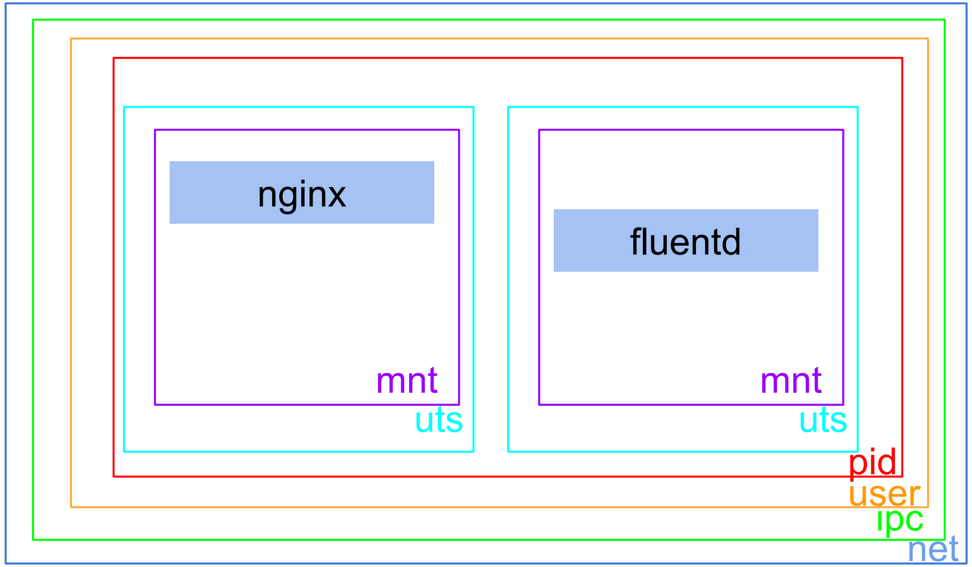


Service mesh

- Service mesh is a dedicated infrastructure layer that can be added to an application
- Service mesh allows to add capabilities such as observability, traffic management and security without adding them to the app code
- The term “service mesh” describe the type of software that you use to implement this pattern, and the security or network domain that is created when you use that software
- Service mesh has 2 components:
  - The data plane
  - The control plane
- Service mesh uses a proxy to intercept all the network traffic .

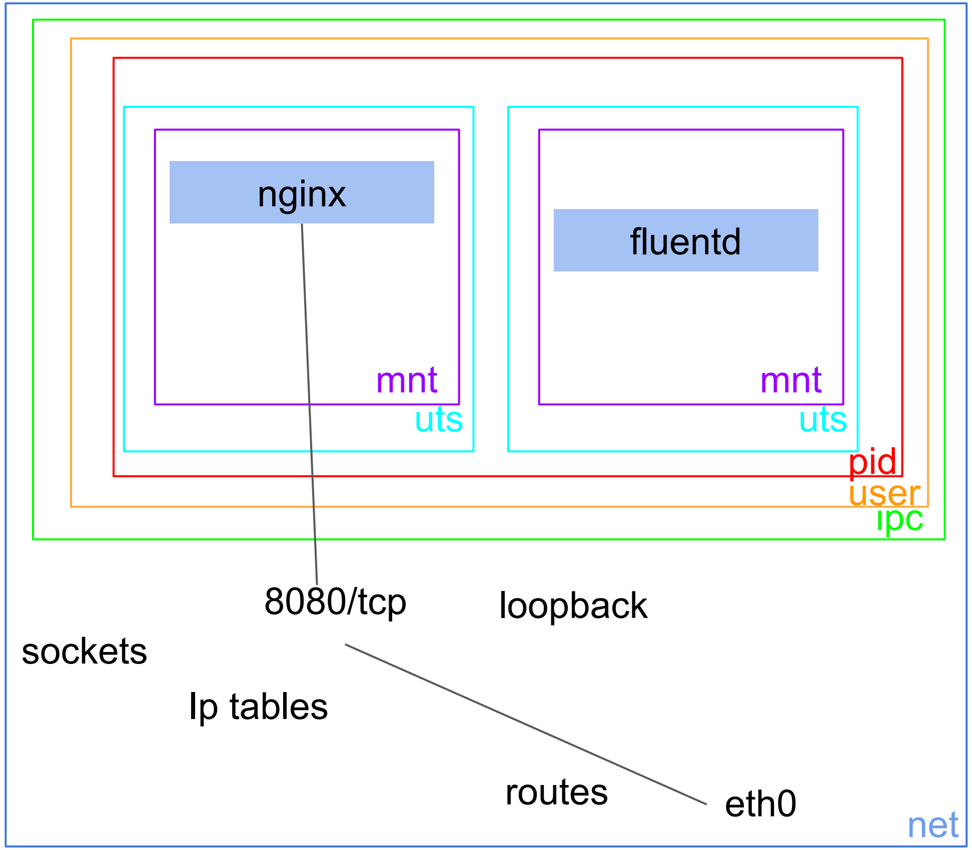
Kubernetes pods what are they

- A group of user space process that run in 6 namespaces isolated form the host namespaces
- All they process in a pod share 4 namespace:pid,user,pic,net
- Process are grouped into containers where each container runs in different mnt,uts namespaces



Kubernetes pods what are they

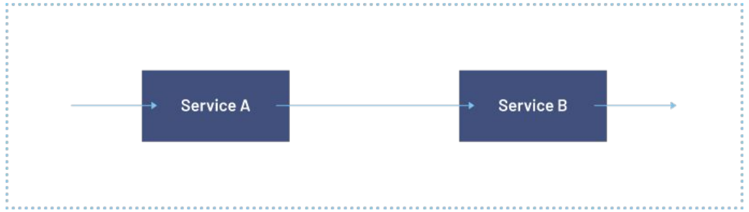
- So multi container pods share the same ip address
- They can communicate using the loopback interface



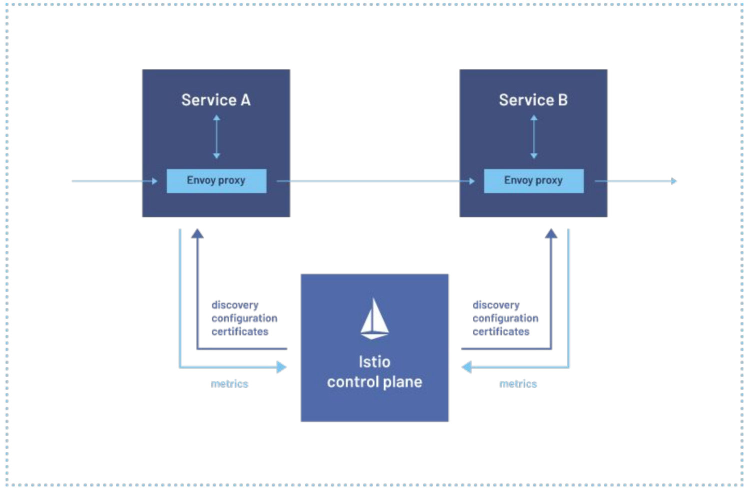
Case study: istio

- Istio is an open source service mesh that layers transparency onto existing applications with few or no service code changes providing important features, including:
    - Secure service to service communication in a cluster with tlsh encryption,strong identity based authentication and authorization
    - Automatic load balancing for HTTP,gRPC,Websocket and TCP traffic
    - Fine-grained control of traffic behaviour with rich routing rules,retries,failovers, and fault injection
    - Automatic metrics,logs, and traces for all traffic within a cluster,including ingress and egress

Before utilizing Istio

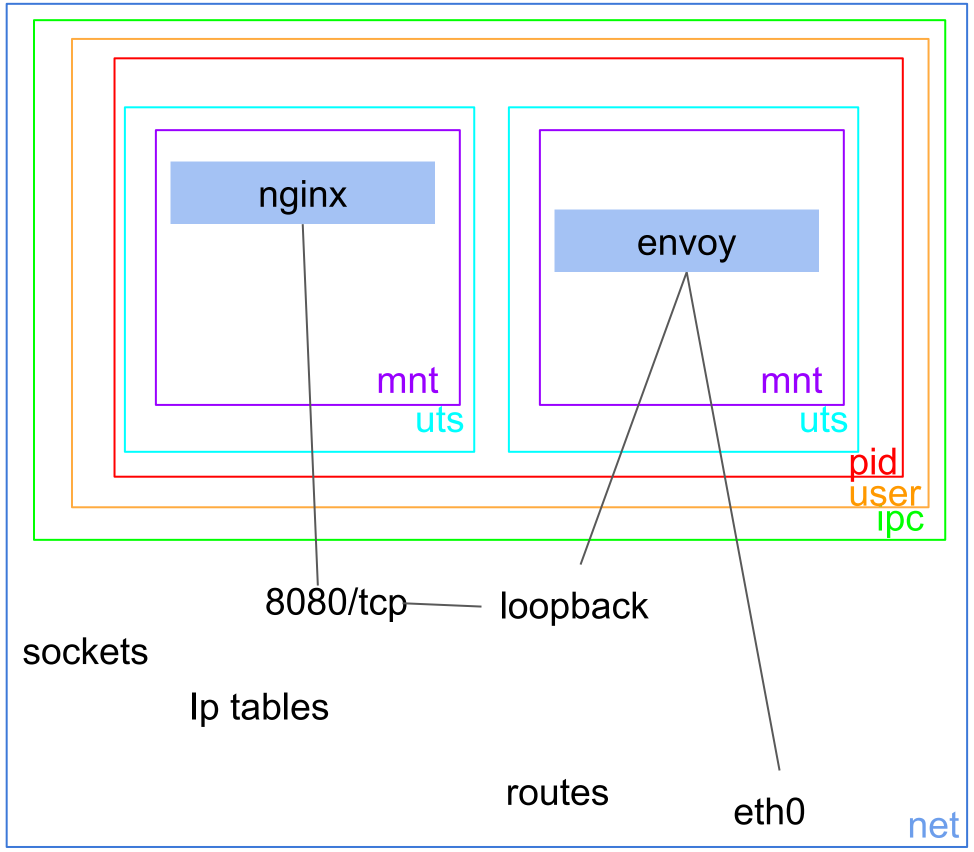


After utilizing Istio

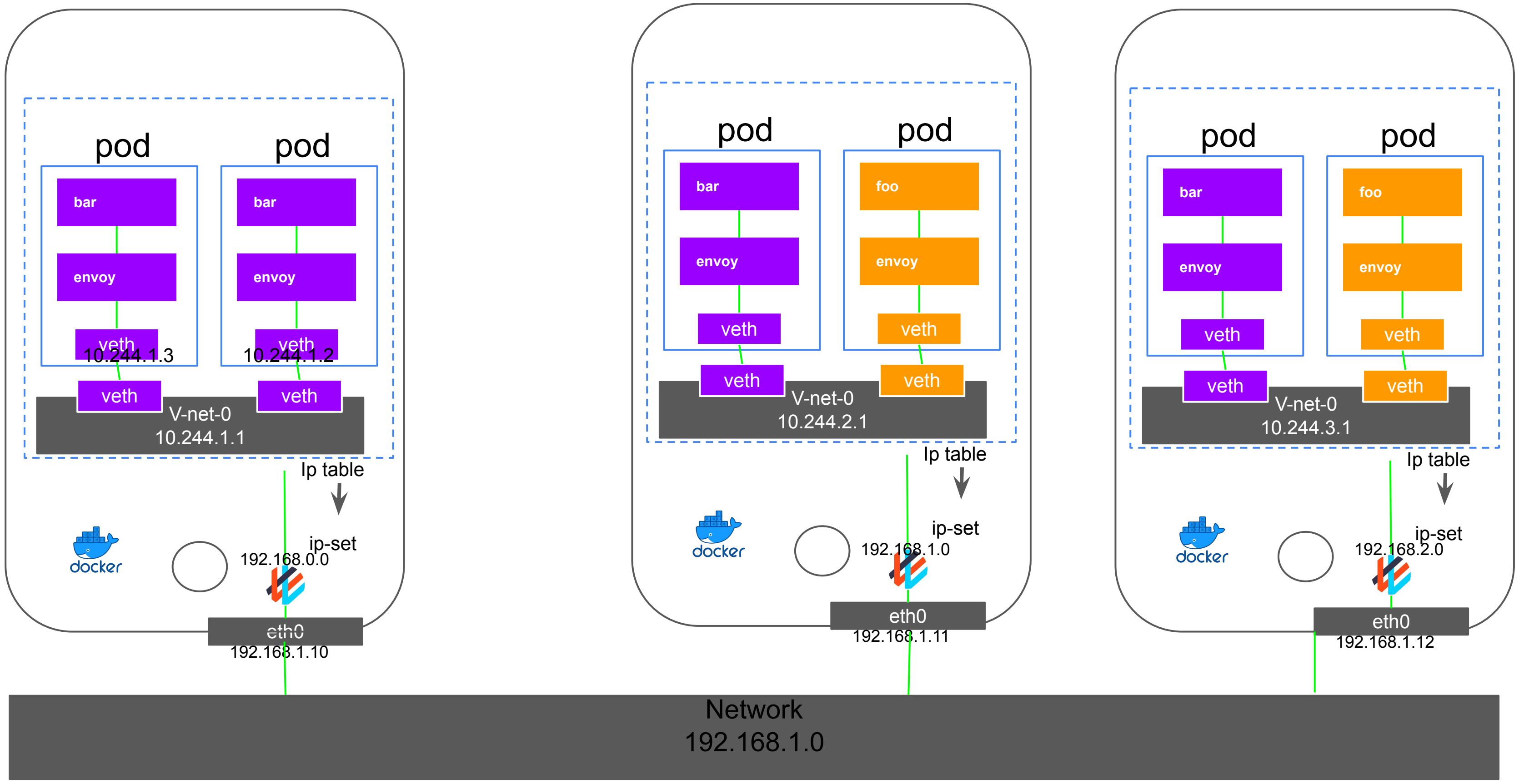


Kubernetes pods what are they

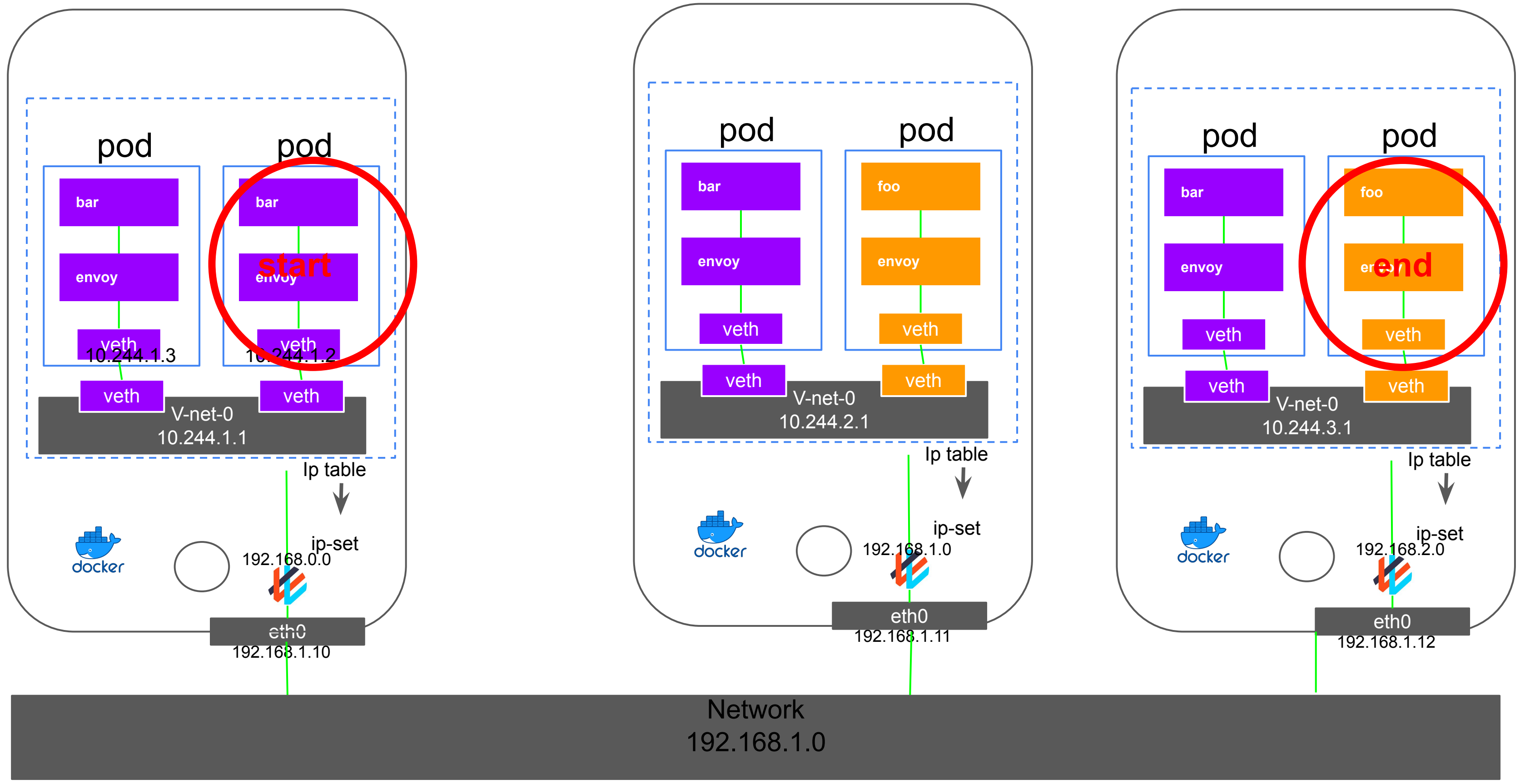
- A group of user space process that run in 6 namespaces isolated form the host namespaces
- All they process in a pod share 4 namespace:pid,user,pic,net
- Process are grouped into containers where each container runs in different mnt,uts namespaces



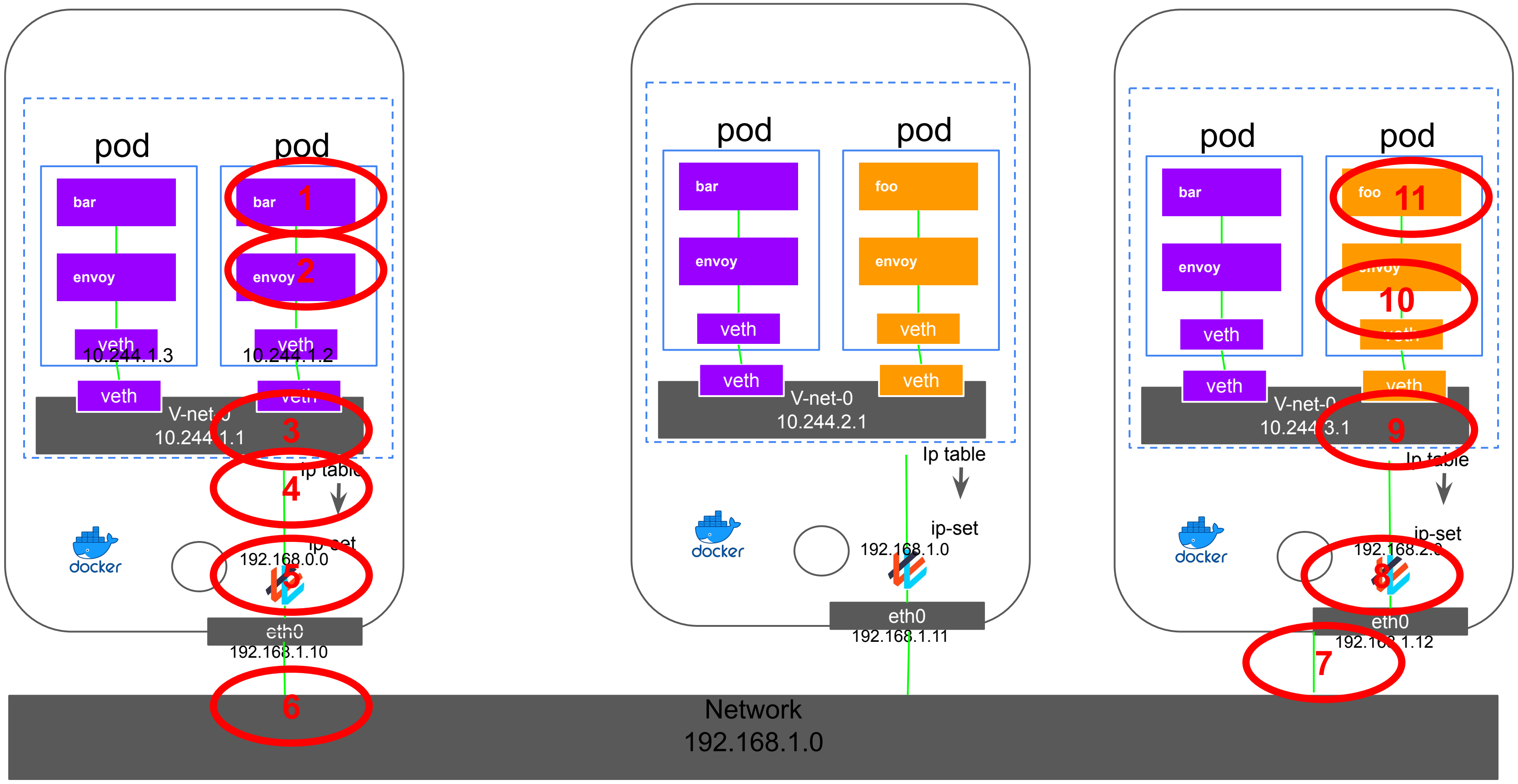
Final architecture



Final architecture

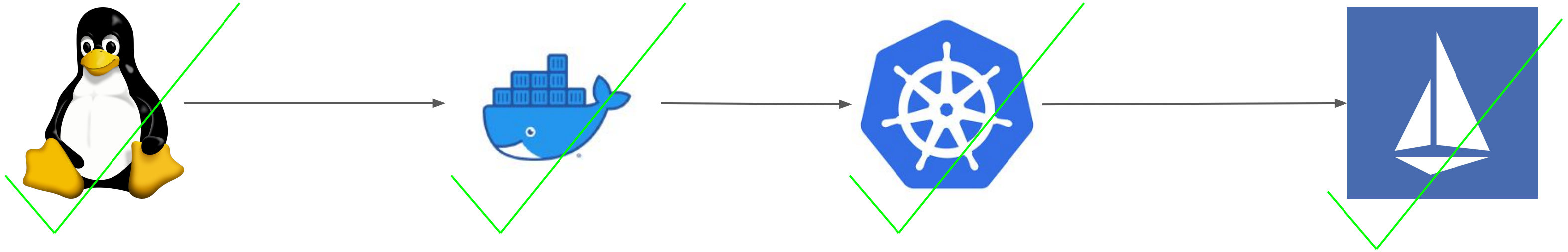


Final architecture



Life of a packet in k8s

from Linux network namespaces to Service Mesh



Other related lecture idea(if there is a demand)

- Deep dive into istio(part 2)
- Deep dive into overlay networks(including case studies)
- End to end overview of how scheduling works in kubernetes
- Building container from scratch in Go
- Kubernetes design patterns(sidecar and more)