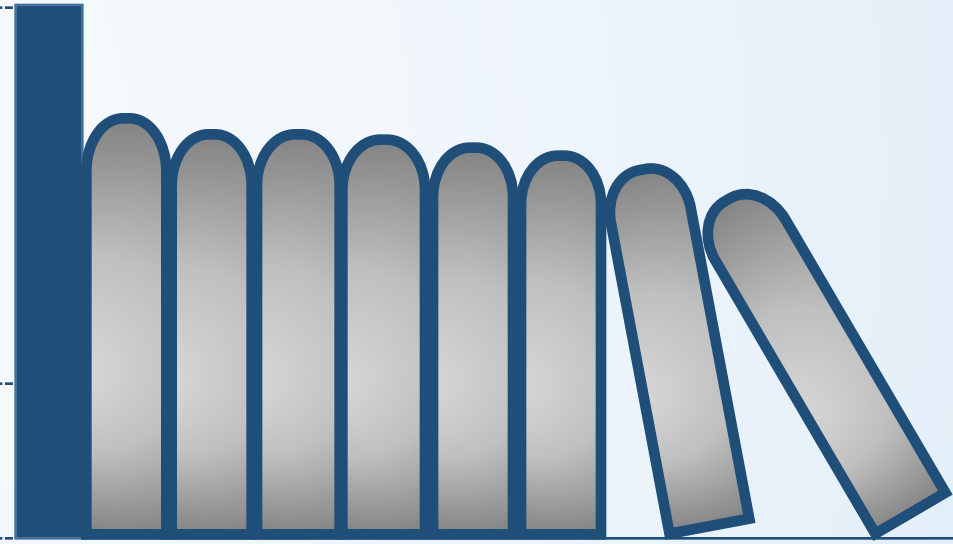
Kubernetes网络

主讲人：宋小金



目录

预期收获

- *了解Kubernetes网络模型*
- 了解Kubernetes网络通信

网络技术术语

- **网络命名空间**：Linux在网络栈中引入网络命名空间，将独立的网络协议栈隔离到不同的命令空间中，彼此间无法通信；Docker利用这一特性，实现不同容器间的网络隔离。
- **Veth设备对**：Veth设备对的引入是为了实现在不同网络命名空间的通信，比如连接Pod和docker0
- **Iptables/Netfilter**：Netfilter负责在内核中执行各种挂接的规则（过滤、修改、丢弃等），运行在内核模式中；Iptables模式是在用户模式下运行的进程，负责协助维护内核中Netfilter的各种规则表；通过二者的配合来实现整个Linux网络协议栈中灵活的数据包处理机制。
- **网桥**：二层网络设备，可以将Linux支持的不同的端口连接起来，并实现类似交换机的通信能力，典型的网桥就是docker0
- **路由**：Linux系统包含一个完整的路由功能，当IP层在处理数据发送或转发的时候，会使用路由表来决定发往哪里。
- **IPAM**：IP地址管理；这个IP地址管理并不是容器所特有的，传统的网络比如说DHCP其实也是一种IPAM，到了容器时代我们谈IPAM，主流的两种方法： 基于CIDR的IP地址段分配地或者精确为每一个容器分配IP
- **Overlay**：在现有二层或三层网络之上再构建的一个独立的网络，通常会有自己独立的IP地址空间、交换或者路由的实现。
- **VxLAN**：解决方案最主要是解决VLAN支持虚拟网络数量（4096）过少的问题。因为在公有云上每一个租户都有不同的VPC，4096明显不够用。就有了Vxlan，它可以支持1600万个虚拟网络，基本上公有云是够用的。
- **BGP协议**：主干网自治网络的路由协议，今天有了互联网，互联网由很多小的自治网络构成的，自治网络之间的三层路由是由BGP实现的

Kubernetes网络模型

Pod是Kurbernetes进行**创建、调度和管理的最小单位**，同一个Pod内的container不会跨宿主机，每个Pod都有一个**独立的IP**（IP-per-Pod），同一个Pod包含的全部Container**共享同一个网络协议栈**（或网络命名空间），集群中的**所有Pod都处在一个可以扁平互通的网络空间**中。

Kubernetes中3种常见IP：

- Pod IP
- Service Cluster IP
- Node IP

Kubernetes网络模型

查看一个Pod的IP:

```
$ kubectl get po frontend-5fjb4 -o wide
NAME             READY     STATUS    RESTARTS   AGE       IP              NODE
frontend-5fjb4   1/1       Running   0          3d        10.244.1.107    bjo-ep-dep-040.dev.fwm
rm.net
```

或者:

```
$ kubectl get --all-namespaces --output json  pods | jq '.items[] | select(.metadata.name=="fro
ntend-5fjb4")' | jq .status.podIP
"10.244.1.107"
```

CNM & CNI

容器网络两大阵营

- Docker的CNM
- Google、CoreOS、Kuberenetes主导的CNI

- 说明：CNM和CNI并不是网络实现，他们是网络规范和网络体系，从研发的角度他们就是一堆接口，你底层是用Flannel也好、用Calico也好，他们并不关心，CNM和CNI关心的是网络管理的问题。

CNM & CNI

CNM是Docker LibnetworkContainer Network Model的简称，Docker Libnetwork优势就是原生，而且和Docker容器生命周期结合紧密；缺点也可以理解是原生，被Docker“绑架”。

- Docker Swarm overlay
- Macvlan & IP networkdrivers
- Calico
- Contiv
- Weave

CNI是Container NetworkInterface的简称，CNI的优势是兼容其他容器技术（e.g. rkt）及上层编排系统（Kubernetes & Mesos），而且社区活跃势头迅猛，Kubernetes加上CoreOS主推；缺点是非Docker原生。

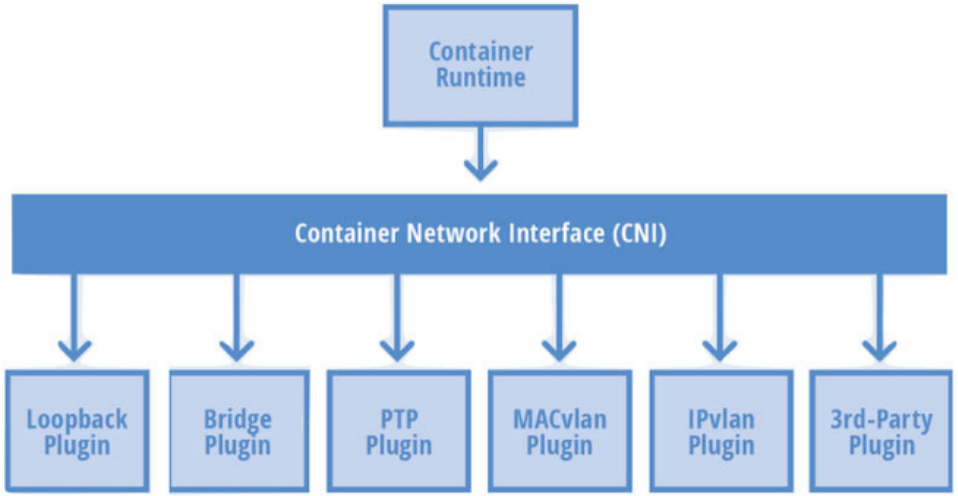
- Kubernetes
- Weave
- Macvlan
- Calico
- Flannel
- Contiv

Kubernetes采用CNI

CNI有多简单

一个配置文件
一个可执行文件

读取6个环境变量
接收1个命令行参数
实现2个操作（ADD/DEL）



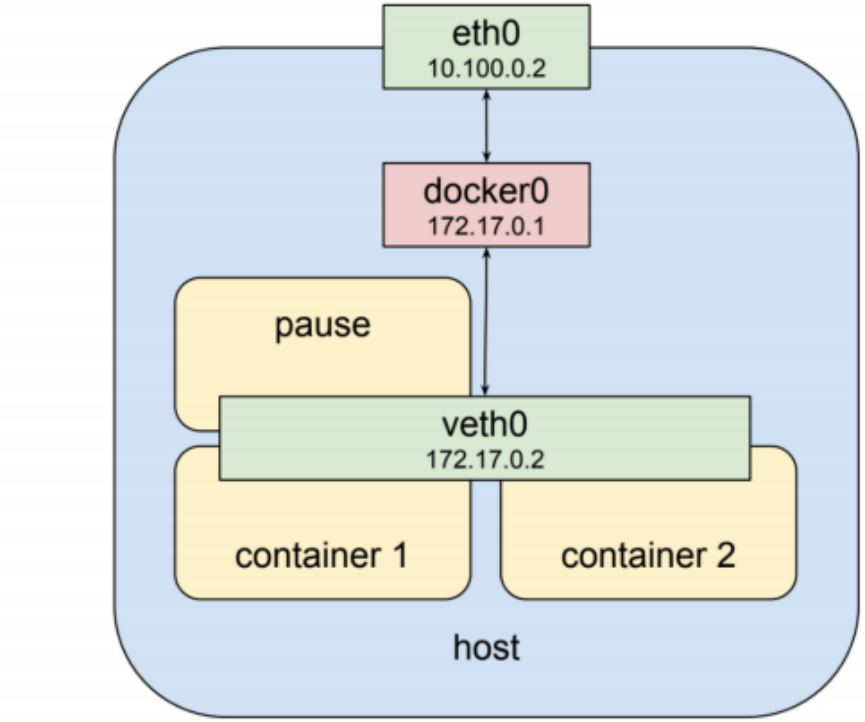
完整的CNI规范内容：https://github.com/containernetworking/cni/blob/master/SPEC.md
将Docker network命令转换为CNI插件的例子：https://github.com/kubernetes/contrib/tree/master/cni-plugins/to_docker

Kubernetes网络实现

根据不同的业务场景需要，Kubernetes网络设计主要考虑了几种通信场景：

- 同一个Pod内容器间互通
- 同一个Node上Pod间互通
- 不同Node上Pod间的互通
- Service与Pod之间的通信
- K8s集群内外组件间通信

同一个Pod内容器间互通



- 同一个Pod内的容器共享一个网络Namespace
- Pod内的容器不会跨宿主机，共享一个网络Namespace
- 容器间使用localhost访问
- Pod内的容器因为在同一网络Namespace，所有必须使用不同的端口通信

同一个Pod内容器间互通

```yaml
apiVersion: extensions/v1beta1 # for
 apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 1 # tells deployment to
plate
  template: # create pods using pod
    metadata:
      # unlike pod-nginx.yaml, the na
 meta data as a unique name is
      # generated from the deployment
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
      - name: busybox
        image: busybox
        command:
          - sleep
          - "3600"
```

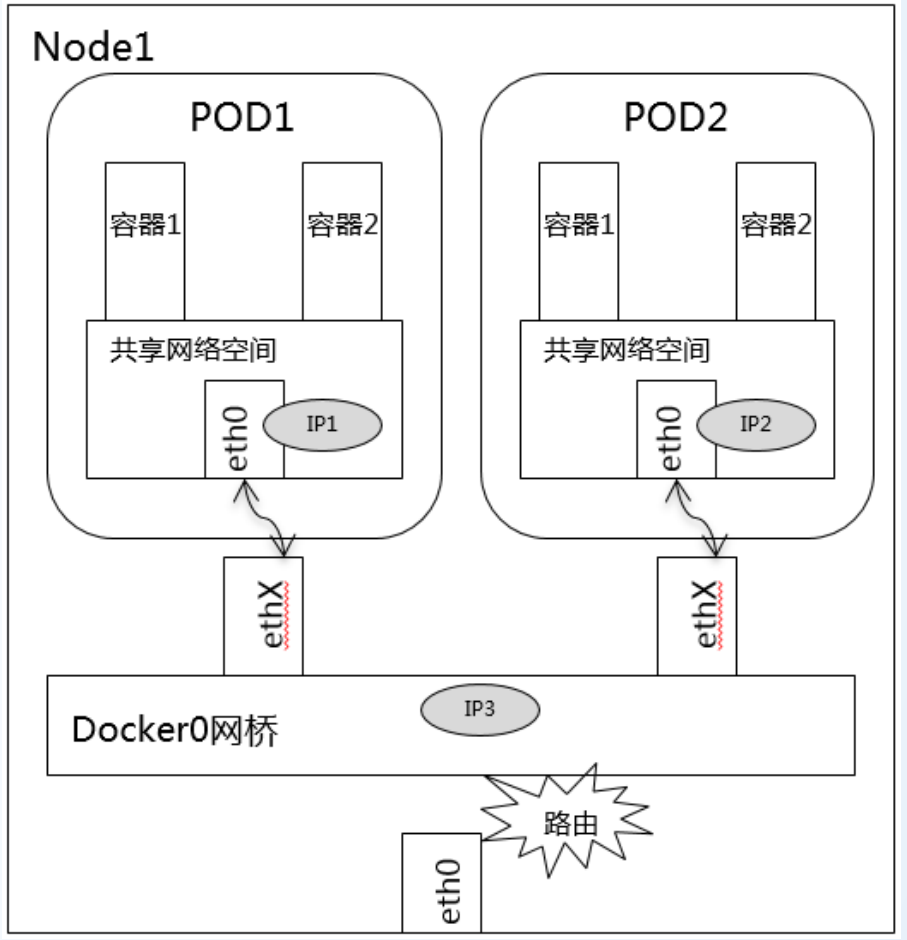
同一个Pod内的容器采用container的网络模式，共享同一个网络栈，可通过 *localhost*或者127.0.0.1 相互访问

```
!2605 $ docker inspect k8s_busybox_nginx-deployment-84553510-6w2jx
_default_682dd4aa-4c17-11e8-97d3-fa163ee6fdff_0 | grep NetworkMode
            "NetworkMode": "container:41c32a9c57b2684503b9a0a5cb6f
8d01ae475bd0cfa8c73ec1727a54888c9208",
$ docker inspect k8s_nginx_nginx-deployment-84553510-6w2jx_default
_682dd4aa-4c17-11e8-97d3-fa163ee6fdff_0 | grep NetworkMode
            "NetworkMode": "container:41c32a9c57b2684503b9a0a5cb6f
8d01ae475bd0cfa8c73ec1727a54888c9208",
```

均为container模式，且指向相同容器

同一个Node上Pod间互通

- Pod1与Pod2都是通过虚拟网络设备Veth，连接到同一个docker0 bridge的，这两个Pod的IP地址也是通过docker0网段动态分配的，与docker0 bridge属于同一个网段。
- Pod的默认路由都是docker0 bridge的地址，所有非本地地址的网络数据，默认都会发送到docker0网桥上，由docker0网桥中转
- Pod与docker0之间是Veth设备对连接的，而docker0 bridge与Node的eth0是路由转发的，Docker0上默认网关就是Node的eth0

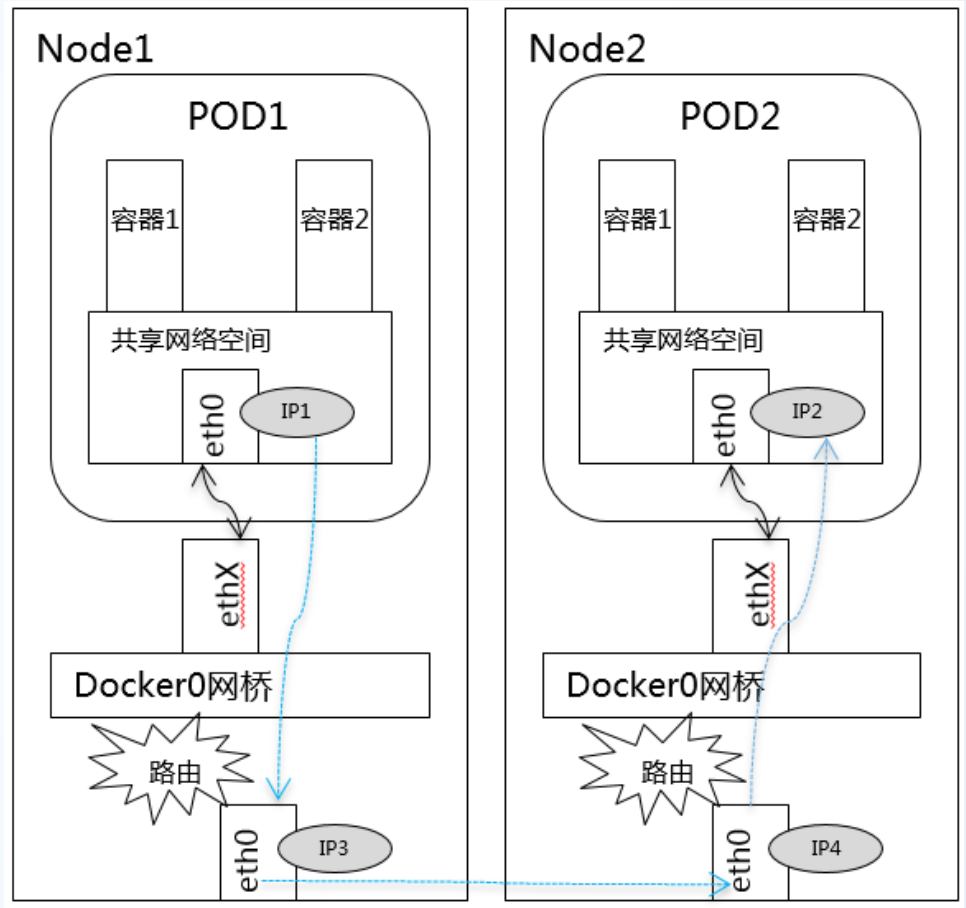


不同Node上Pod间通信

同一个Node 上的Pod 通过Veth 连接在同一个docker0 网桥上，地址段相同，原生能通信。

但是不同Node 之间的Pod 如何通信的，本质是在网路上再架设一层overlay network 使容器的网络运行在这层overlay 网络上。

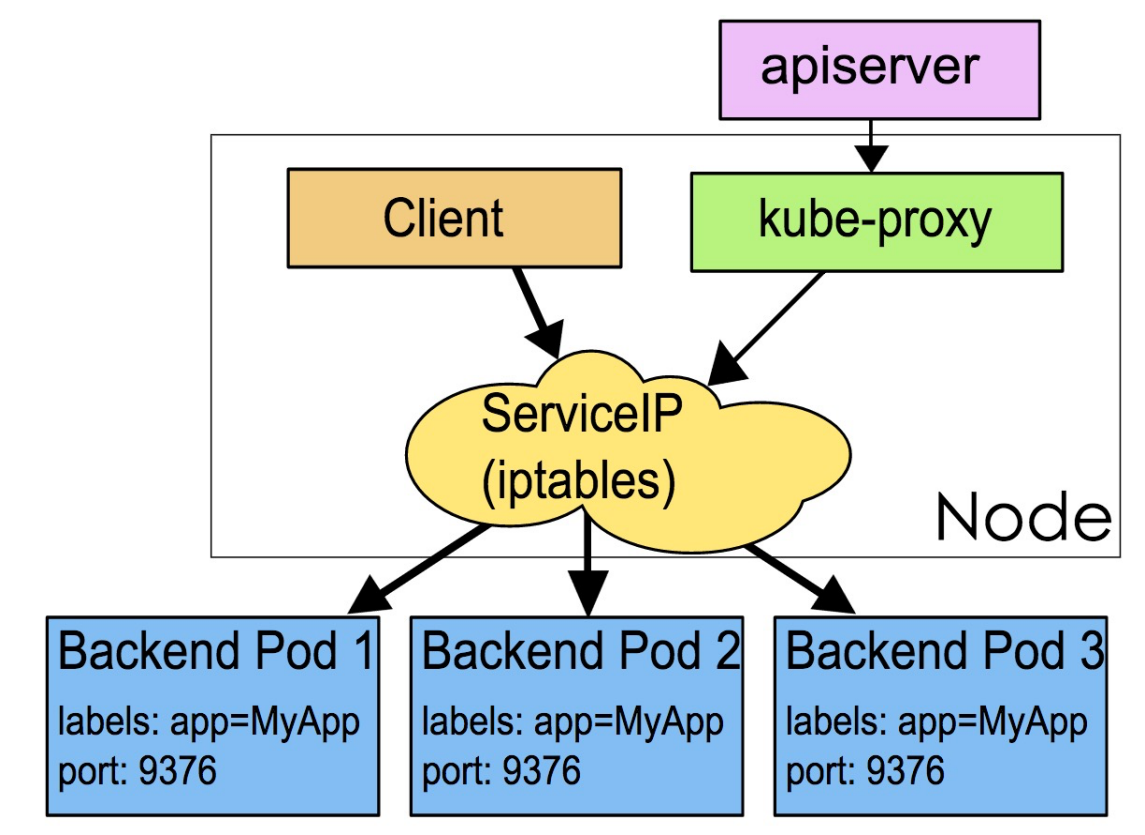
现有方案有Flannel，Calico，Cannal等。



Service与Pod之间的通信

Proxy-mode: iptables

以暴露NodePort的Service为例，NodePort的工作原理与ClusterIP大致相同，发送到某个NodeIP:NodePort的请求，通过**iptables重定向到kube-proxy对应的端口**(Node上的随机端口)上，然后由kube-proxy再将请求发送到其中的一个Pod:TargetPort

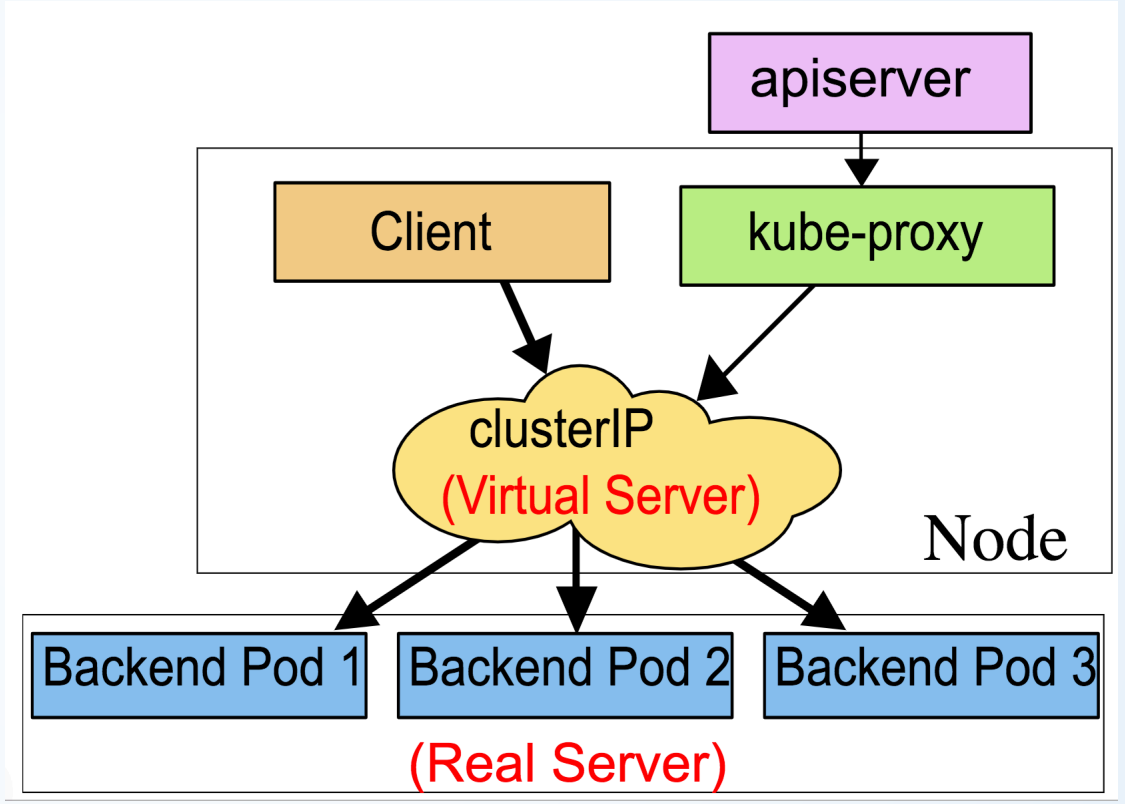


Service与Pod之间的通信

Proxy-mode: ipvs

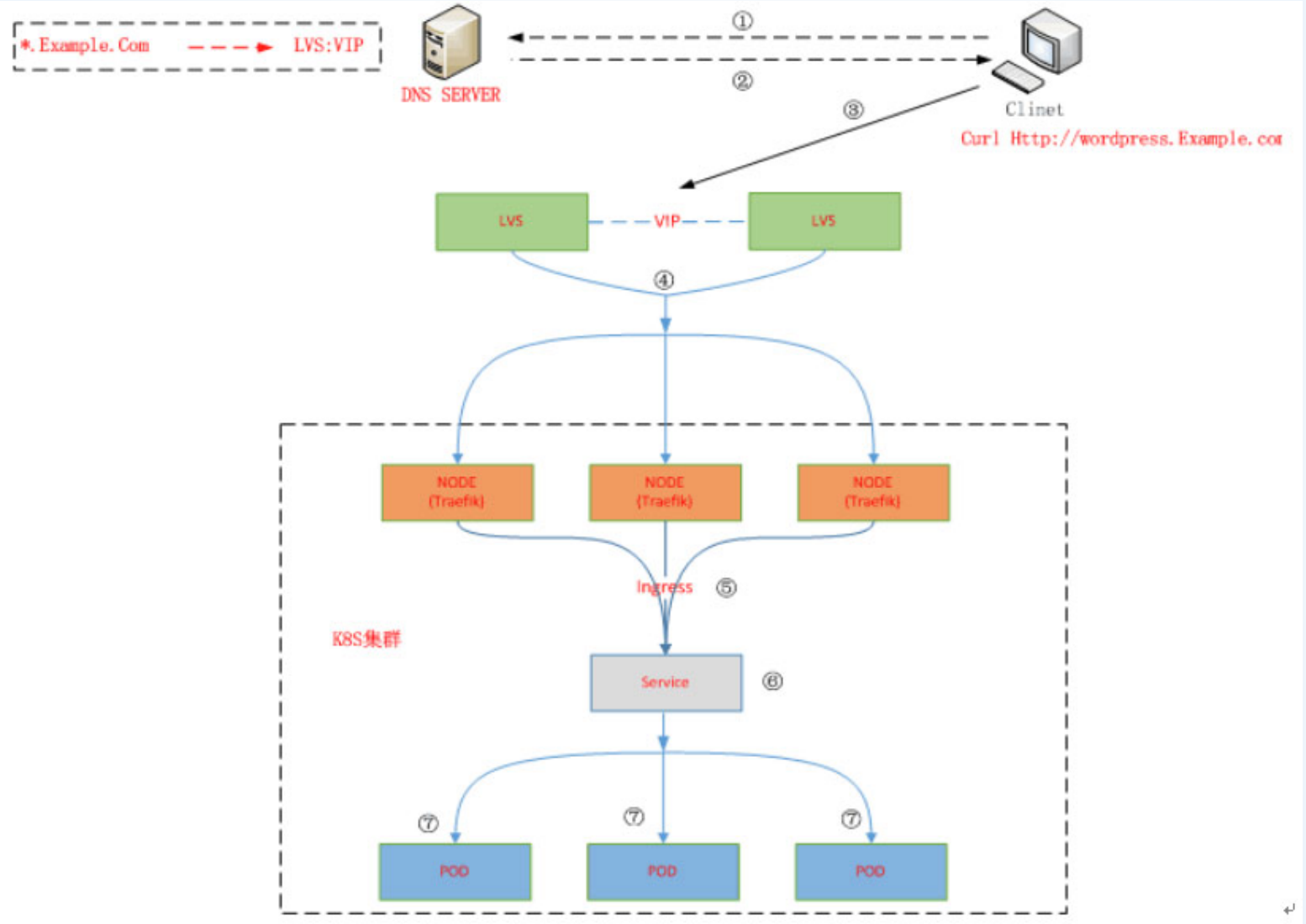
IPVS 是 LVS 项目的一部分，是一款运行在 Linux kernel 当中的 4 层负载均衡器，性能异常优秀。支持如下负载均衡策略：

- rr: round-robin
- lc: least connection
- dh: destination hashing
- sh: source hashing
- sed: shortest expected delay
- nq: never queue



ipvs vs Iptables： https://www.objectif-libre.com/en/blog/2018/03/19/kubernetes-ipvs/

K8s集群内外组件间通信



Kubernetes网络问题

主要解决的问题

- 容器IP分配冲突，不是全网唯一（比如配置中心化）
- 跨Node容器不可达（Overlay方案，路由方案）
- 网络策略控制

有问题，就会有人来解决，出现了很多网络方案及相关组件（Flannel，Calico等）

常用开源网络组件 -- Flannel

官网描述：Flannel是CoreOS公司专为kubernetes定制的三层网络解决方案，主要用于解决容器的跨主机通信问题。

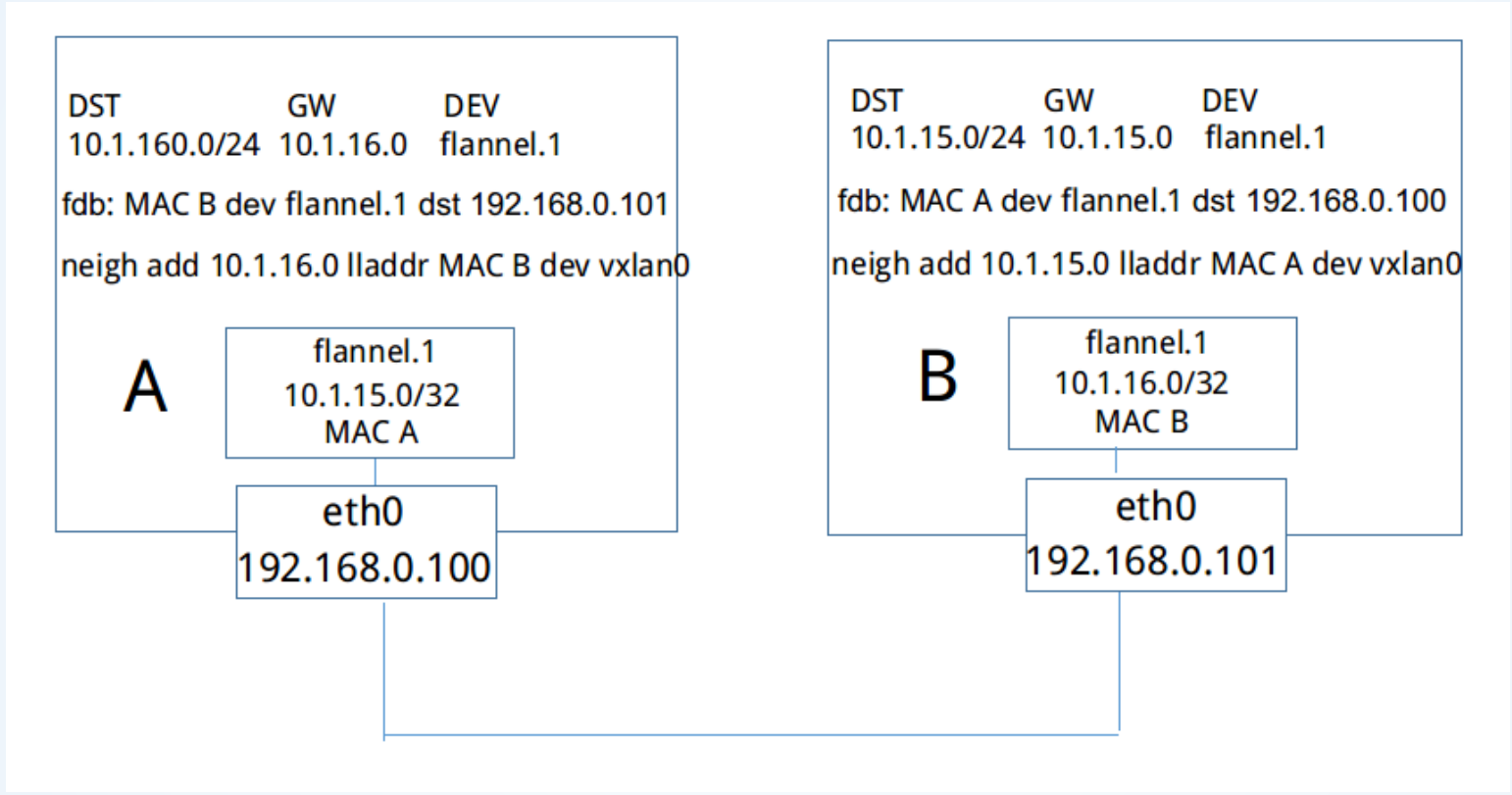
工作原理：

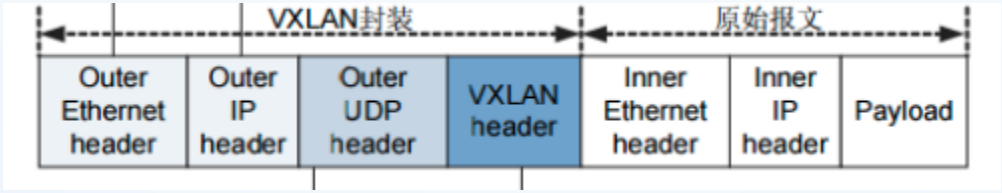
通过给每台宿主机分配一个子网的方式为容器提供虚拟网络，使用UDP/Vxlan封装IP包来创建overlay网络，并借助etcd维护网络的分配情况。控制平面上host本地的flanneld负责从远端的Etcd集群同步本地和其它host上的subnet信息，并为POD分配IP地址。数据平面flannel通过Backend（比如UDP/Vxan封装）来实现L3 Overlay

支持多种backend机制

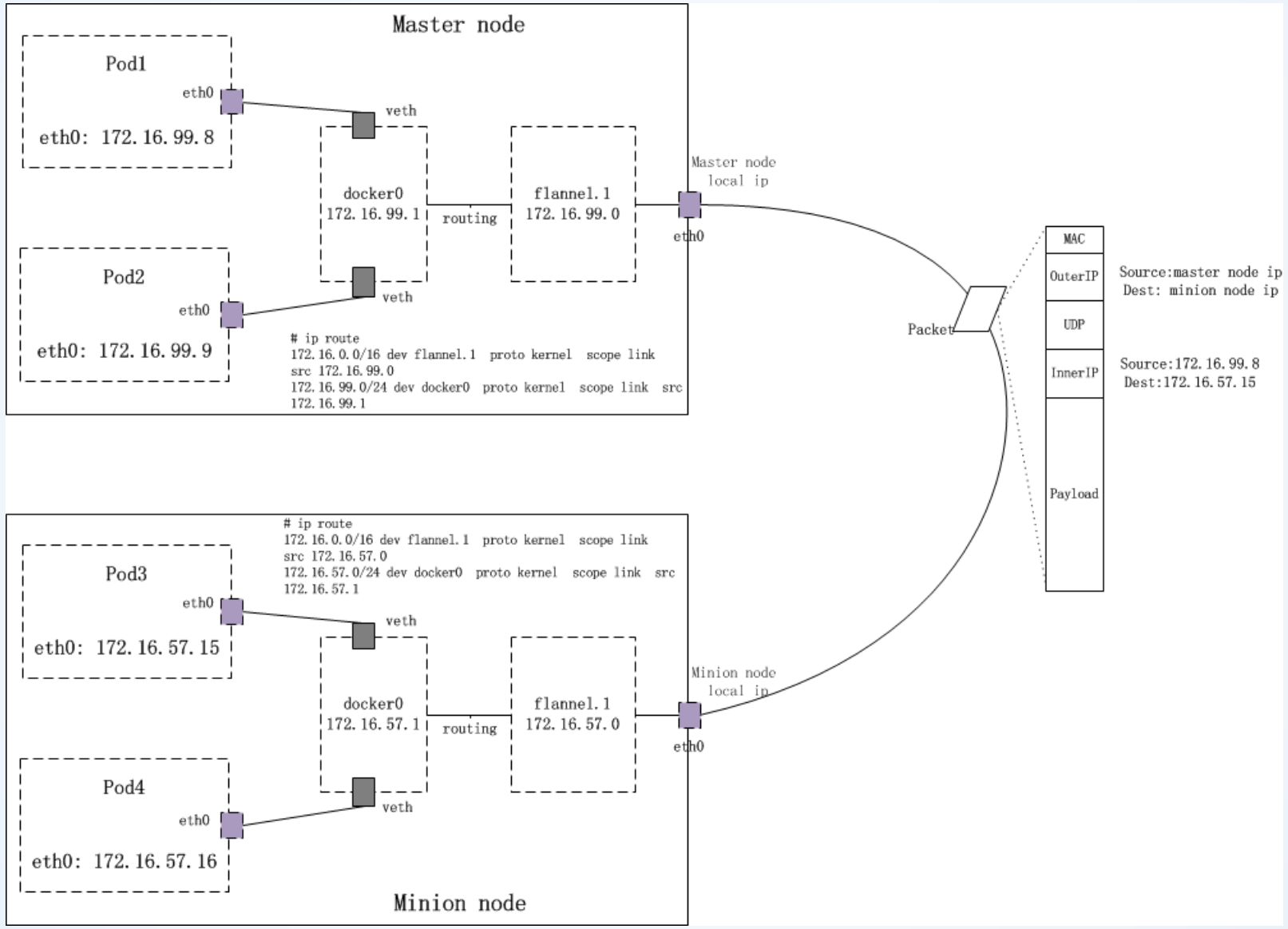
- UDP
- VxLAN
- Host-GW
- AliVPC【以下试验性，不推荐使用】
- AWS VPC
- GCE

常用开源网络组件 -- Flannel





常用开源网络组件 -- Flannel



Flannel -- Vxlan原理

```
# docker exec ba75f81455c7 ip route
default via 172.16.99.1 dev eth0
172.16.99.0/24 dev eth0  proto kernel  scope link  src 172.16.99.8
```

目的地址172.16.57.15并不在直连网络中，因此数据包通过default路由出去。default路由的路由器地址是172.16.99.1，也就是上面的docker0 bridge的IP地址。相当于docker0 bridge以“三层的工作模式”直接接收到来自容器的数据包(而并非从bridge的二层端口接收)。

b) docker0与flannel.1之间的包转发

数据包到达docker0后，docker0的内核栈处理程序发现这个数据包的目的地址是172.16.57.15，并不是真的要送给自己，于是开始为该数据包找下一hop。根据master node上的路由表：

```
master node:

# ip route
... ...
172.16.0.0/16 dev flannel.1  proto kernel  scope link  src 172.16.99.0
172.16.99.0/24 dev docker0  proto kernel  scope link  src 172.16.99.1
... ...
```

我们匹配到“172.16.0.0/16”这条路由！这是一条直连路由，数据包被直接送到flannel.1设备上。

Flannel -- Vxlan原理

flannel.1将包转发出去，因为毕竟包不是给自己的（包目的ip是172.16.57.15, vxlan设备ip是172.16.99.0），但不会走寻常套路去转发包，flannel.1收到数据包后，由于自己不是目的地，也要尝试将数据包重新发送出去。数据包沿着网络协议栈向下流动，在二层时需要封二层以太包，填写目的mac地址，这时一般应该发出arp：”who is 172.16.57.15”。但vxlan设备的特殊性就在于它并没有真正在二层发出这个arp包，因为下面的这个内核参数设置：

而是由linux kernel引发一个”L3 MISS”事件并将arp请求发到用户空间的flanned程序。

flanned程序收到”L3 MISS”内核事件以及arp请求(who is 172.16.57.15)后，并不会向外网发送arp request，而是尝试从etcd查找该地址匹配的子网的vtep信息

Flannel -- Vxlan原理

etcd中有各节点的Flannel network子网的配置信息：

```
master node:

# etcdctl --endpoints http://127.0.0.1:{etcd listen port} ls  /coreos.com/network/subnets
/coreos.com/network/subnets/172.16.99.0-24
/coreos.com/network/subnets/172.16.57.0-24

# curl -L http://127.0.0.1:{etcd listen port}/v2/keys/coreos.com/network/subnets/172.16.57.0-24
{"action":"get","node":{"key":"/coreos.com/network/subnets/172.16.57.0-24","value":"{\"PublicIP\":\"{minion
node local ip}\",\"BackendType\":\"vxlan\",\"BackendData\":
{\"VtepMAC\":\"d6:51:2e:80:5c:69\"}}","expiration":"2017-01-
17T09:46:20.607339725Z","ttl":21496,"modifiedIndex":2275460,"createdIndex":2275460}}
```

flanneld从etcd中找到了答案：

```
subnet: 172.16.57.0/24
public ip: {minion node local ip}
VtepMAC: d6:51:2e:80:5c:69
```

我们查看minion node上的信息，发现minion node上的flannel.1 设备mac就是d6:51:2e:80:5c:69:

Flannel -- Vxlan原理

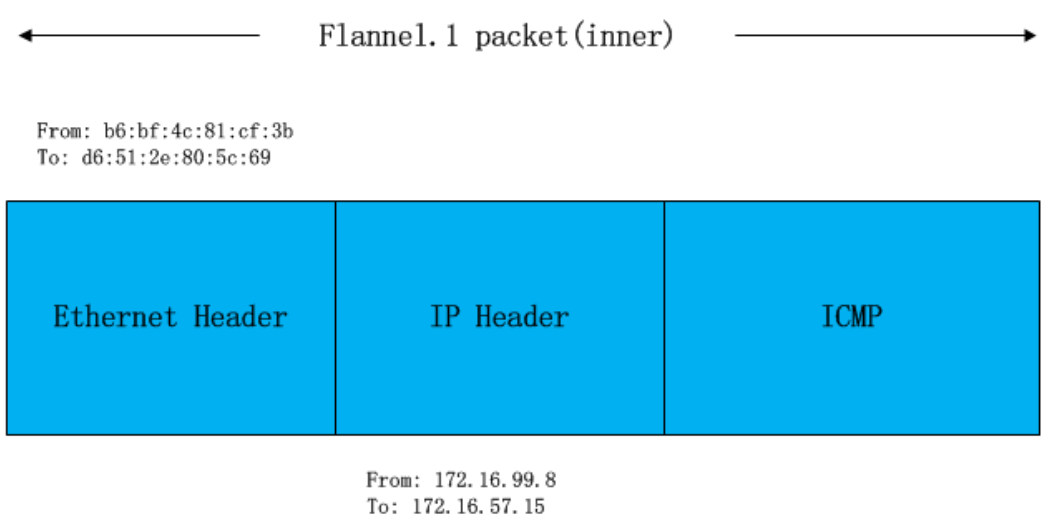
接下来，flanned将查询到的信息放入master node host的arp cache表中：

```
master node:

#ip n |grep 172.16.57.15
172.16.57.15 dev flannel.1 lladdr d6:51:2e:80:5c:69 REACHABLE
```

flanneld完成这项工作后，linux kernel就可以在arp table中找到 172.16.57.15对应的mac地址并封装二层以太包了。

到目前为止，已经呈现在大家眼前的封包如下图：



不过这个封包还不能在物理网络上传输，因为它实际上只是vxlan tunnel上的packet。

Flannel -- Vxlan原理

d) kernel的vxlan封包

我们需要将上述的packet从master node传输到minion node，需要将上述packet再次封包。这个任务在backend为vxlan的flannel network中由linux kernel来完成。

flannel.1为vxlan设备，linux kernel可以自动识别，并将上面的packet进行vxlan封包处理。在这个封包过程中，kernel需要知道该数据包究竟发到哪个node上去。kernel需要查看node上的fdb(forwarding database)以获得上面对端vtep设备（已经从arp table中查到其mac地址：d6:51:2e:80:5c:69）所在的node地址。如果fdb中没有这个信息，那么kernel会向用户空间的flanned程序发起"L2 MISS"事件。flanneld收到该事件后，会查询etcd，获取该vtep设备对应的node的"Public IP"，并将信息注册到fdb中。

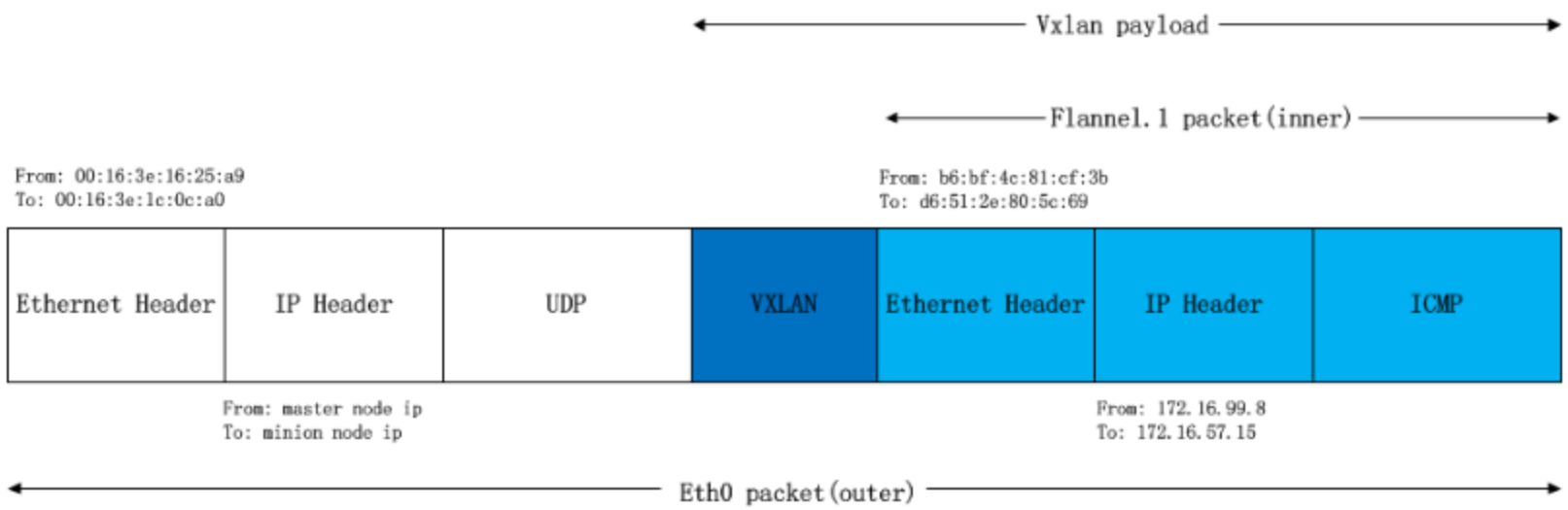
这样Kernel就可以顺利查询到该信息并封包了：

```
master node:

# bridge fdb show dev flannel.1|grep d6:51:2e:80:5c:69
d6:51:2e:80:5c:69 dst {minion node local ip} self permanent
```

Flannel -- Vxlan原理

由于目标ip是minion node，查找路由表，包应该从master node的eth0发出，这样src ip和src mac地址也就确定了。封好的包示意图如下：



e) kernel的vxlan拆包

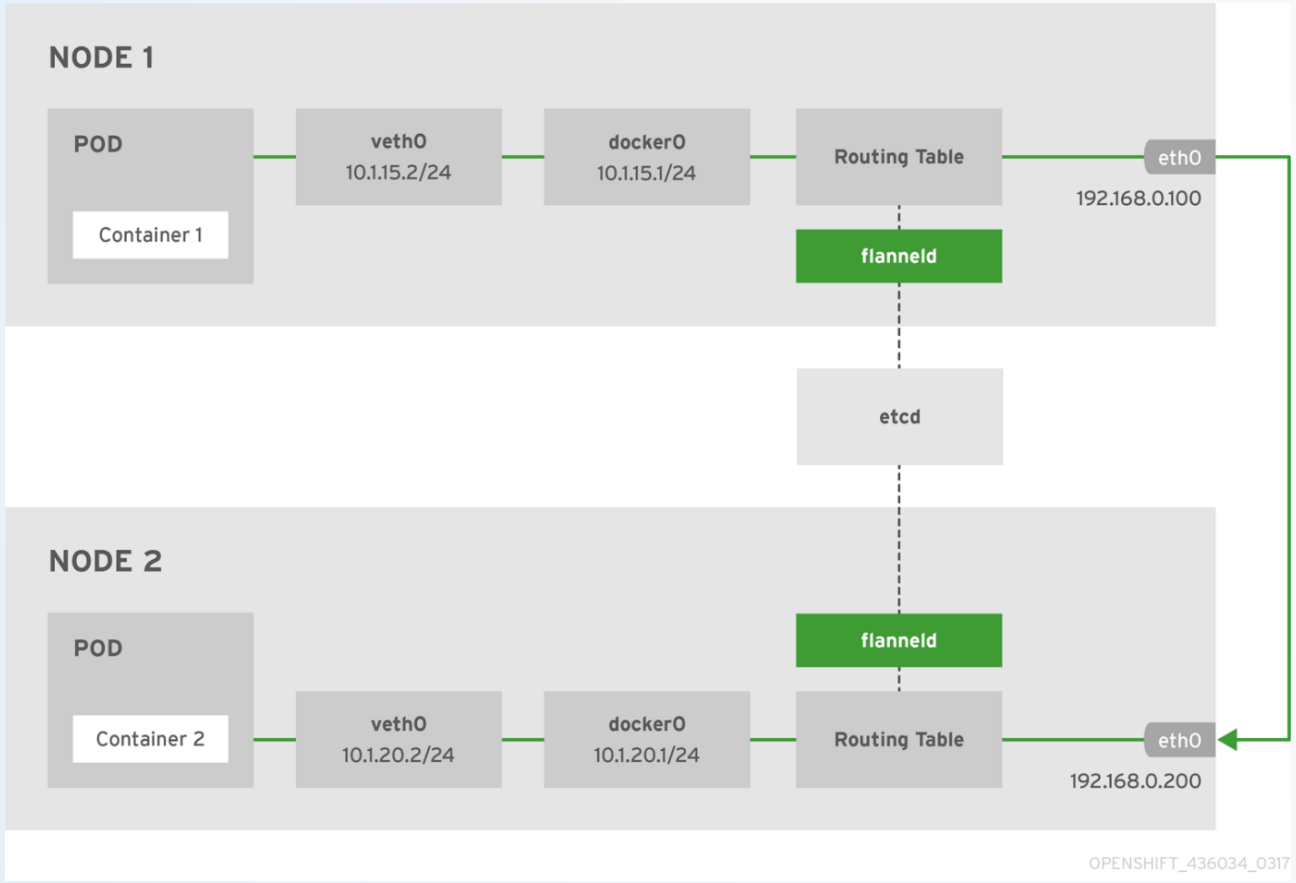
minion node上的eth0接收到上述vxlan包，kernel将识别出这是一个vxlan包，于是拆包后将flannel.1 packet转给minion node上的vtep（flannel.1）。minion node上的flannel.1再将这个数据包转到minion node上的docker0，继而由docker0传输到Pod3的某个容器里。

Flannel -- Vxlan原理

- VxLAN封包采用的是内置在Linux内核里的标准协议，因此虽然它的封包结构比UDP模式复杂，但由于所有数据装、解包过程均在内核中完成，实际的传输速度要比UDP模式快许多
- 大概2014年，主流的Linux系统还是Ubuntu 14.04或者CentOS 6.x，这些发行版的内核比较低（内核版本2.6），没有包含VxLAN的内核模块，CentOS7的基本上是3.10.X
- UDP在前几年还是有点市场的，旧服务器，老环境适配，新版Linux稳定性不明
- 主要问题
  - Payload有损耗，由于加了额外的头部层
  - 虽然在内核执行拆包解包的，但还是有损耗，速度还是不算快

思考：能不能不用Overlay网络？

Flannel -- HostGW原理




```
root@host1:~#
root@host1:~# ip route
default via 10.0.2.1 dev enp0s3
10.0.2.0/24 dev enp0s3  proto kernel  scope link  src 10.0.2.5
10.2.0.0/16 dev flannel.1
10.2.17.0/24 via 192.168.56.105 dev enp0s8
10.2.40.0/24 dev docker0  proto kernel  scope link  src 10.2.40.1
192.168.56.0/24 dev enp0s8  proto kernel  scope link  src 192.168.56.104
192.168.122.0/24 dev virbr0  proto kernel  scope link  src 192.168.122.1 linkdown
root@host1:~#
```

```
root@host2:~#
root@host2:~# ip r
default via 10.0.2.1 dev enp0s3
10.0.2.0/24 dev enp0s3  proto kernel  scope link  src 10.0.2.6
10.2.0.0/16 dev flannel.1
10.2.17.0/24 dev docker0  proto kernel  scope link  src 10.2.17.1
10.2.40.0/24 via 192.168.56.104 dev enp0s8
192.168.56.0/24 dev enp0s8  proto kernel  scope link  src 192.168.56.105
192.168.122.0/24 dev virbr0  proto kernel  scope link  src 192.168.122.1 linkdown
root@host2:~#
```

Flannel -- HostGW原理

- 与VxLAN不同，Host-GW不会封装数据包，而是在主机的路由表中创建到其他主机subnet的路由条目，从而实现容器跨主机通信，由flanneld守护进程监听etcd的信息，自动配置各结点的路由信息
- 主要特点及问题
  - 直接路由，没有拆名解包，Payload的额外损耗
  - 路由条数多，路由表大，由于在同一网络中，Pod多时会产生网络风暴
  - 要求各结点在同一网络，结点不能跨路由，不支持大集群规模

思考：怎样解决路由表太大，所有Pod都在三层可能产生网络风暴的问题？

思考：大规模集群，本质是解决路由器上路由自动配置问题，怎么办？

Flannel总结

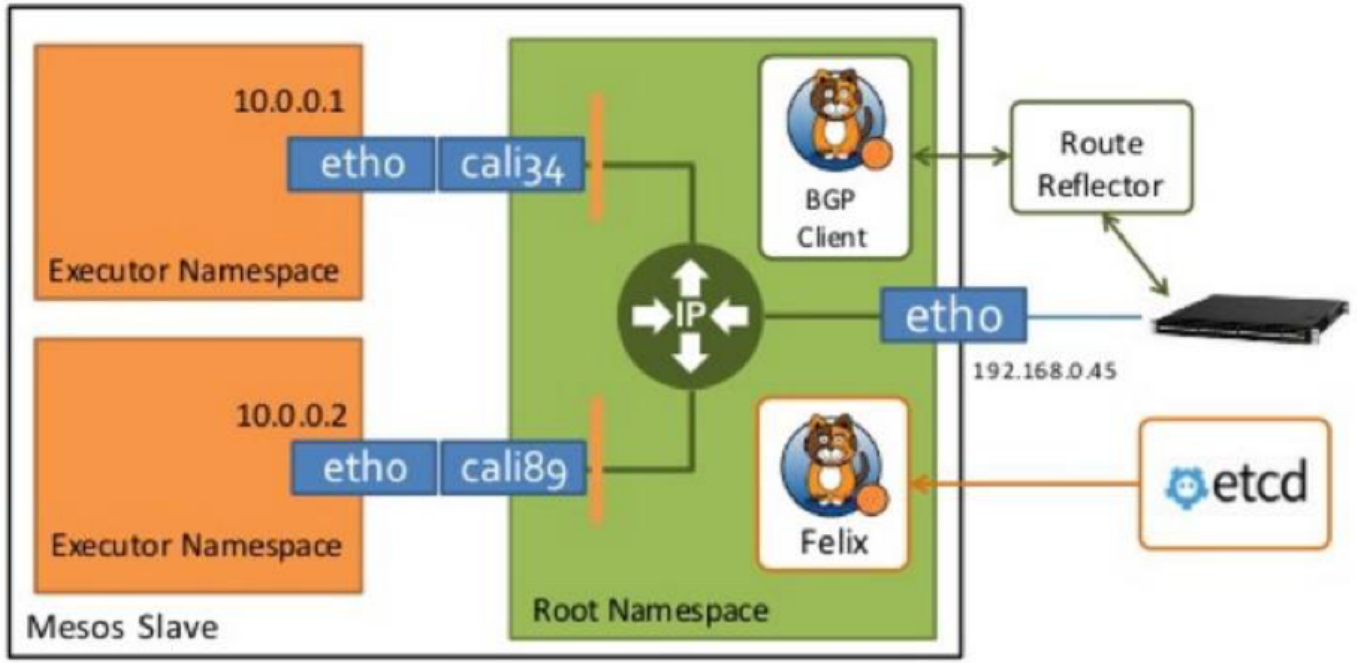
维度	UDP	Vxlan	Host-GW
实现方式	Overlay	Overlay	直接路由
拆解包	有	有	无
拆解方式	用户空间	内核空间	----
Payload	有损失	有损失	无损失
网络性能	非常慢	良好	非常好
网络风暴	无	无	有
集群规模	大，结点通即可	大，结点通即可	小，结点须直连

常用开源网络组件 -- Calico

- CNI plugin, CNM plugin, Neutron plugin
- Calico是一个三层的数据中心网络方案，而且方便集成OpenStack这种IaaS云架构，能够提供高效可控的VM、容器、裸机之间的通信。
- Calico是一个多主机路由软件，还包含一个分布式防火墙。前期用Python开发，后用Go重写。
- Calico为每个容器或者虚拟机分配一个独立的IP地址，然后在每台物理主机上定义包含这些 IP 地址的 iptables 规则，实现了防火墙功能。 Calico在每个物理节点上跑一个高效的vRouter, 由它对外广播本机各容器的路由信息。它基于BGP协议，不仅适用于小规模部署，在route reflector的帮助下，更能应用于大型DataCenter。包的转发用是的Linux内核的转发功能，高效而简单。只要编排框架支持为每个服务分配一个 IP 地址，就可以集成使用 Calico 。
- Calico把容器们的访问信息包装在路由信息里，把L3作为容器访问的隔离方式，并使用 Linux 内核转发，很好地作出隔离与性能间的权衡。

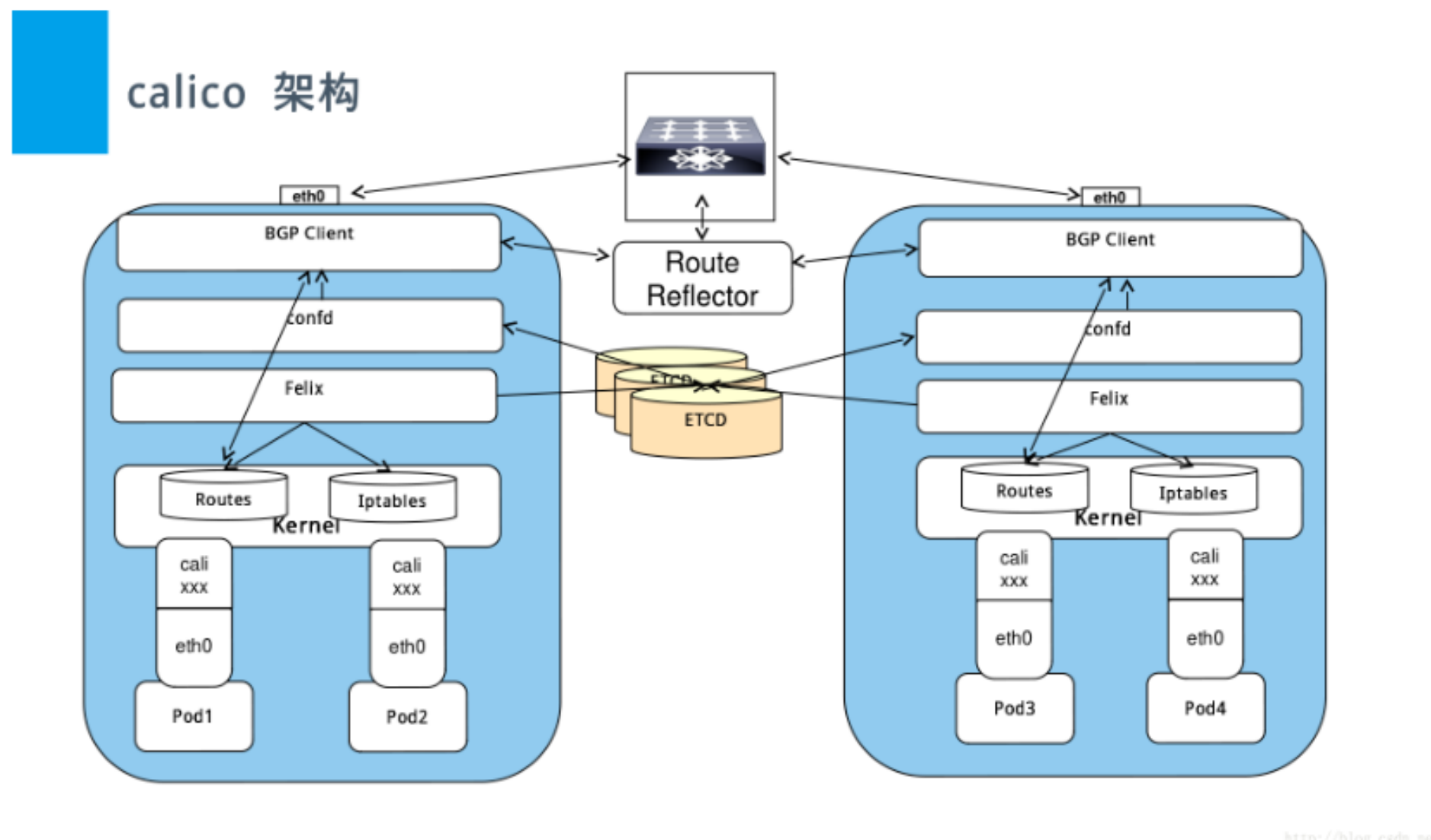
常用开源网络组件 -- Calico

Calico的核心组件



- **Felix，Calico Agent**，跑在每台需要运行Workload的节点上，主要负责配置路由及ACLs等信息来确保Endpoint的连通状态；
- **etcd**，分布式键值存储，主要负责网络元数据一致性，确保Calico网络状态的准确性；
- **BGP Client（BIRD）**，主要负责把Felix写入Kernel的路由信息分发到当前Calico网络，确保Workload间的通信的有效性；
- **BGP Route Reflector（BIRD）**，大规模部署时使用，摒弃所有节点互联的 mesh 模式，通过一个或者多个BGP Route Reflector来完成集中式的路由分发。

常用开源网络组件 -- Calico



常用开源网络组件 -- Calico

补充说明：

- 三层直连，可跨路由，支持大规模集群部署
- 使用BGP协议，要求路由器开应该功能，对底层网络基础设施有侵入
- 为兼容网络基础设施，不开启BGP协议，提供IPIP方式，本质上与Flannel-Vxlan类似，仅封装的隧道格式为IPIP
- 支持较完整的网络策略，网络功能与策略控制是解耦的，可与Flannel结合部署，本质上Canel项目源自于些，Flannel的网络功能+Calico的网络策略
- 构架复杂，学习成本高，部署也较麻烦

按原理速度分类

网络方案	快	良好	慢
Flannel-UDP			Y
Flannel-Vxlan		Y	
Flannel-HostGW	Y+		
Calico-BGP	Y		
Calico-IPIP		Y	

课程回顾

已学知识要点

了解Kubernetes的网络模型与网络通信