Packet Walk(s) In Kubernetes

Don Jayakody, Big Switch Networks

@techjaya

linkedin.com/in/jayakody
About

About Big Switch
• We are in the business of "Abstracting Networks" (As one Big Switch) using Open Networking Hardware

About Me
• Spent 4 years in Engineering building our products. Now in Technical Product Management

Legacy

Next-Gen

Why Not Abstract the Network?

Kubernetes: Abstrated Compute Domain

http://www.youtube.com/watch?v=HlAXp0-M6SY&t=0m43s
Agenda

Intro: K8S Networking

- Namespace/Pods/CNIs?
- What’s that “Pause” Container really do?
- Flannel: Intro / Packet Flows
- Exposing Services

Calico: Networking

- Architecture
- IP-IP Mode (Route formation / Pod-to-pod communication / ARP Resolution/ Packet Flow)
- BGP Mode (Peering requirements/ Packet Flow)

Cilium: Networking

- Architecture
- Overlay Network Mode (Configuration/ Pod-to-pod communication/ Datapath / ARP Resolution/ Packet Flow)
- Direct Routing Mode
K8S Networking: Basics

Namespaces

- Linux kernel has 6 types of namespaces: pid, net, mnt, uts, ipc, user
- Network namespaces provide a brand-new network stack for all the processes within the namespace
- That includes network interfaces, routing tables and iptables rules
Pods

- Lowest common denominator in K8S. Pod is comprised of one or more containers along with a "pause" container
- Pause container act as the “parent” container for other containers inside the pod. One of its primary responsibilities is to bring up the network namespace
- Great for the redundancy: Termination of other containers do not result in termination of the network namespace

```
[root@Master1 ~]# kubectl get pods --namespace web --o wide
NAME                      READY STATUS    AGE          IP            NODE
nginx-deployment-76bf4969df-5xzv7 1/1  Running 7m53s     172.31.155.17 Worker-1
```

```
[root@Worker-1 ~]# docker ps
CONTAINER ID        IMAGE                        COMMAND                  CREATED            STATUS          NAME                      LABELS
9349bffece72b      docker.io/nginx5sha256        "nginx -g "daemon off" 47 seconds ago  Running           k8s_nginx_nginx-deployment
2e012ea5db0         k8s.gcr.io/pause:3.1     "/pause"                 57 seconds ago   k8s_pod_pginx-deployment
```
K8S Networking: Basics

Accessing Pod Namespaces

- Multiple ways to access pod namespaces
- `kubectl exec --it`
- `docker exec --it`
- `nsenter ("namespacenter", let you run commands that are installed on the host but not on the container)`

Both containers belong to the same pod => Same Network Namespace => same ‘ip a’ output
K8S Networking: Basics

Container Networking Interface: CNI

- Interface between container runtime and network implementation
- Network plugin implements the CNI spec. It takes a container runtime and configure (attach/detach) it to the network
- CNI plugin is an executable (in: /opt/cni/bin)
- When invoked it reads in a JSON config & Environment Variables to get all the required parameters to configure the container with the network

Credit: https://www.slideshare.net/weaveworks/introduction-to-the-container-network-interface-cni
Topology

Topology Info

- Node IP belongs to 2 different subnets: 25.25.25.0/24 & 35.35.35.0/24
- Gateway is configured with .254 in the network for each network segment
- Bond0 interface is created by bonding two 10G interfaces
Flannel

**Intro**

- To make networking easier, Kubernetes does away with port-mapping and assigns a unique IP address to each pod.
- If a host cannot get an entire subnet to itself things get pretty complicated.
- Flannel aims to solve this problem by creating an overlay mesh network that provisions a subnet to each server.
Flannel

Default Config
• “Flannel.1” Is the VXLAN interface
• CNI0 is the bridge

Node-121
25.25.25.121

Node-122
35.35.35.122
Flannel

Pod-to-Pod Communication

- Brought up 2 pods

```
$ kubectl get pods -n test -o wide
NAME          READY STATUS    RESTARTS AGE   IP            NODE            
busybox0-nzxgg 1/1  Running 10  10h   10.244.1.2  flannel-node-18121
busybox0-6qrqm 1/1  Running  9  10h   10.244.2.2  flannel-node-18122
```
Flannel

Pod-to-Pod Communication
- VETH Interface with "veth" got created on the root namespace
- Other end is attached to the pod namespace

$ kubectl exec -it busybox-nxngg -n test /bin/sh
  # ip a
  3: eth0@if12: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP qlen 1000
     link/ether 0c:50:00:01:01:01 brd ff:ff:ff:ff:ff:ff
     inet 10.244.1.2/24 scope global eth0
       valid_lft forever preferred_lft forever
     inet6 fe80::c00:50ff:fe01:0101/64 scope link
       valid_lft forever preferred_lft forever
  / #

$ kubectl exec -it busybox-09240 -n test /bin/sh
  # ip a
  3: eth0@if12: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP qlen 1000
     link/ether 0c:50:00:02:02:02 brd ff:ff:ff:ff:ff:ff
     inet 10.244.2.2/24 scope global eth0
       valid_lft forever preferred_lft forever
     inet6 fe80::c00:50ff:fe02:0202/64 scope link
       valid_lft forever preferred_lft forever
  / #

Notice the index on veth. eg: "eth0@if12" on pod1 ns corresponds to 12th interface on the root ns.
Flannel

ARP Handling

- 'cni0' bridge is replying to ARP requests from the pod

```
$ kubectl get pods -n test -o wide
NAME               READY STATUS    RESTARTS AGE   IP           NODE
busybox0-nzxgg     1/1       Running 10 10h   10.244.1.2  flannel-node-18121
busybox0-6qrqm     1/1       Running  9 10h   10.244.2.2  flannel-node-18122
```

ARP Handling Diagram:

- 'cni0' bridge is replying to ARP requests from the pod
**Flannel**

**Packet Flow**

- Routing table lookup to figure out where to send the packet
Flannel

Packet Flow

- Flannel.1 device does VXLAN encap/decap
- Traffic from pod1 is going through “flannel.1” device before exiting
Calico

Felix
• The primary Calico agent that runs on each machine that hosts endpoints.
• Responsible for programming routes and ACLs, and anything else required on the host

Bird
• BGP Client: responsible of route distribution
• When Felix inserts routes into the Linux kernel FIB, Bird will pick them up and distribute them to the other nodes in the deployment
Calico

Architecture

- Felix's primary responsibility is to program the host's iptables and routes to provide the connectivity to pods on that host.
- Bird is a BGP agent for Linux that is used to exchange routing information between the hosts. The routes that are programmed by Felix are picked up by bird and distributed among the cluster hosts.

*etcd/confd components are not shown for clarity
Calico

Default Configuration

- Node-to-node mesh
- IP-IP encapsulation
Calico

Default Configuration

- Default: Node Mesh with IP-IP tunnel
- Route table has entries to all the other tunl0 interfaces through other node IPs

```
[root@calico-node-18121 ~]# ip a
18: tun0@if54: <UP,PROMISC,NOARP,LINK_UP> mtu 1448 qdisc noqueue state UNKNOWN qlen 1000
link/addr 0.0.0.0 brd 0.0.0.0
inet 192.168.83.64/32 brd 192.168.83.64 scope global tun0
valid_lft forever preferred_lft forever
```

“ipip” tunnel

```
[root@calico-node-18121 ~]# route -n
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
25.25.25.0 0.0.0.0 255.255.255.0 UG 0 0 0 0 tun0
35.35.35.0 0.0.0.0 255.255.255.0 UG 0 0 0 0 tun0
169.254.0.0 0.0.0.0 255.255.255.0 UG 0 0 0 0 tun0
192.168.83.64 0.0.0.0 255.255.255.0 UG 0 0 0 0 tun0
192.168.243.0 0.0.0.0 255.255.255.0 UG 0 0 0 0 tun0
```

Route to the other node’s tunnel
**Default Configuration**

- Default: Node Mesh with IP-IP tunnel
- Route table has entries to all the other tunl0 interfaces through other node IPs

```
[root@calico-node-18122 ~]# ip a
eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP qlen 1000
link/ether 00:00:00:00:00:00 brd 00:00:00:00:00:00
inet 192.168.243.0/32 scope global tunl0
   valid_lft forever preferred_lft forever
```

```
[root@calico-node-18122 ~]# route -n
Kernel IP routing table
Destination     Gateway          Genmask          Flags Metric Ref    Use Iface
192.168.243.0   0.0.0.0          255.255.255.0   U     0     0      0 tunl0
```

**“ipip” tunnel**

```
[root@calico-node-18122 ~]# route -n
Kernel IP routing table
Destination     Gateway          Genmask          Flags Metric Ref    Use Iface
192.168.243.0   0.0.0.0          255.255.255.0   U     0     0      0 tunl0
```

**Route to the other node’s tunnel**

```
[root@calico-node-18122 ~]# route -t
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
Pod-to-Pod Communication

- Brought up 2 pods
- "calicoctl get wep" ("workloadendpoints") shows the endpoints in calico end
### Pod-to-Pod Communication

- VETH Interface with “cali-xxx” got created on the root namespace
- Other end is attached to the pod namespace

```bash
$ kubectl exec busybox-41dp -n test -- ip a
1: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1460 qdisc noqueue state UNKNOWN qlen 1000
   link/ether 00:11:22:33:44:55 brd ff:ff:ff:ff:ff:ff
   inet 192.168.83.67/32 scope global eth0
       valid_lft forever preferred_lft forever
   valid_lft forever preferred_lft forever
```

```bash
$ kubectl exec busybox-42dp -n test -- ip a
1: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1460 qdisc noqueue state UNKNOWN qlen 1000
   link/ether 00:11:22:33:44:55 brd ff:ff:ff:ff:ff:ff
   inet 192.168.243.2/32 scope global eth0
       valid_lft forever preferred_lft forever
   valid_lft forever preferred_lft forever
```
ARP Resolution

- How does ARP gets resolved?
- pod1 & pod2 default route is pointing to private IPv4 “169.254.1.1”
Create an ARP request

- ARP request is send to default GW 169.254.1.1
- cali interface replies to ARP request with it's own MAC

Pinging from pod1 to pod2

$ kubectl exec busybox-4ldg -n test -- ping 192.168.243.2 -c 1
64 bytes from 192.168.243.2: seq=0 ttl=62 time=0.381 ms
64 bytes from 192.168.243.2: seq=0 ttl=62 time=0.381 ms
64 bytes from 192.168.243.2: seq=0 ttl=62 time=0.381 ms

$ kubectl exec busybox-4ldg -n test -- arp -n ? (169.254.1.1) at eee:eee:eee:eee:eee [ether] on eth0

ARP Resolution

- Initiate ping from pod1-to pod2
- ARP request is send to default GW 169.254.1.1
- cali interface replies to ARP request with it's own MAC

tcpdump on root veth

[root@calico-node-18121 ~]# tcpdump -i eth0
ARP Request: src=169.254.1.1 dst=192.168.83.67
Length 28
ARP Reply: src=192.168.83.67 dst=169.254.1.1 length 28
ARP Reply: src=192.168.83.67 dst=192.168.83.67 length 28
ARP Request: src=192.168.83.67 dst=169.254.1.1 length 28
ARP Reply: src=192.168.83.67 dst=192.168.83.67 length 28

root veth is replying with its own MAC

[root@calico-node-18121 ~]# ifconfig cali4e7dc5a6ea0
cali4e7dc5a6ea0: flags=1463<UP,BROADCAST,RUNNING,MULTICAST>
to 1440
ether eee:eee:eee:eee:eee:eee

ARP Reply

pod1 eth0
192.168.83.67
Node-121
25.25.25.121

bond0
cali-x
tunl0
192.168.83.64

root
iptables
route table

25.25.25.121
35.35.35.122

Calico

ARP Resolution

• Why “Cali-” VETH Interfaces replies to the ARP request?
• Reason: Proxy-ARP is enabled on the veth interface on root namespace
• Network is only learning the Node IP/MACs, not pod macs

No pod-MAC/IPs will be learned on the network. Only Node-IPs

Proxy ARP is enabled on Cali veth:

```
[root@calico-node-18121 ~]# cat /proc/sys/net/ipv4/conf/cali4e7dc5a6ea0/proxy_arp
[root@calico-node-18121 ~]#
```
After ARP Resolution, the packet gets forwarded according to the routing table entries. Once the ARP is resolved, routing table rules kick in.
Calico

Packet Forwarding

- After ARP Resolution packet gets forwarded according to the routing table entries
- Packets will get encapsulated via IP-IP
Packet Forwarding in the same host

- /32 routes are present in the routing table for all the containers
- Packets get forwarded to the appropriate calico veth interface based on the routing rules
Calico

Non IP-IP
• Disable IP-IP Mode

```bash
$ calicoctl get ipool --wide
NAME       CIDR      NAT  IPIPMODE   DISABLED
default-ipv4-ipool  192.168.0.0/16 true Always  false
```

```yaml
$ cat changeIPPool.yaml
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
  name: default-ipv4-ipool
spec:
  cidr: 192.168.0.0/16
  ipipMode: Never
  natOutgoing: false
```

```bash
$ calicoctl apply -f changeIPPool.yaml
Successfully applied 1 'IPPool' resource(s)
```

```bash
$ calicoctl get ipool --wide
NAME       CIDR      NAT  IPIPMODE   DISABLED
default-ipv4-ipool  192.168.0.0/16 false Never  false
```
Calico

Non IP-IP

- Disable IP-IP Mode. tunl0 interface is not present anymore
- Routes are pointing to the bond0 interface
- Bring up 2 pods as before

No more tunl0. Pod routes are directly pointing to bond0
Calico

Non IP-IP

- Ping from pod1 to pod2 is unsuccessful
- Reason: Routes are not advertised to the Network

Packets won’t go
Reason: No routes to the pod networks
Calico

BGP mode

- Need the Calico nodes to peer with the network fabric
Create a BGP configuration with AS:63400

Create a “global” BGP Peer

Both Nodes are peering with the global BGP Peer
- Configure the Calico nodes as BGP Neighbors on the network
- As a result, the network will get to learn about 192.168.83.64/26 & 192.168.243.0/26 routes

Network is peering with both the Calico Nodes

Network is learning pod network routes via BGP
Calico

BGP Mode

- Packets go across the network without any encapsulations
Cilium

Architecture

- Cilium Agent, Cilium CLI Client, CNI Plugin will be running on every node
- Cilium agent compiles BPF programs and makes the kernel runs these programs at key points in the network stack to have visibility and control over all network traffic in/out of all containers
- Cilium interacts with the Linux kernel to install BPF program which will then perform networking tasks and implement security rules

*etcd/monitor components are not shown for clarity
Cilium: Networking

Overlay Network Mode

- All nodes form a mesh of tunnels using the UDP based encapsulation protocols: VXLAN (default) or Geneve
- Simple: Only requirement is cluster nodes should be able to reach each other using IP/UDP
- Auto-configured: Kubernetes is being run with the "--allocate-node-cidrs" option, Cilium can form an overlay network automatically without any configuration by the user

Direct/Native Routing Mode

- In direct routing mode, Cilium will hand all packets which are not addressed for another local endpoint to the routing subsystem of the Linux kernel
- Packets will be routed as if a local process would have emitted the packet
- Admins can use routing daemon such as Zebra, Bird, BGPD. The routing protocols will announce the node allocation prefix via the node’s IP to all other nodes.
Default Configuration

- Overlay Networking Mode
- VXLAN encapsulation
- Both VETH/IPVLAN is supported (Higher Performance gains with IPVLAN)

```
# cilium bpf tunnel list
TUNNEL   VALUE
192.168.2.0 35.35.35.122
```

Node-121
25.25.25.121
Cilium

Default Configuration

- "cilium_vxlan" interface is in metadata mode. It can send & receive on multiple addresses.
- Cilium addressing model allows to derive the node address from each container address.
- This is also used to derive the VTEP address, so you don’t need to run a control plane protocol to distribute these addresses. All you need to have are routes to make the node addresses routable.

```
# root@node-121 ~> ip a
d: cilium_host:cilium_net: <BROADCAST,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP qlen 1000
   link/ether 00:0c:29:00:00:00 brd ff:ff:ff:ff:ff:ff
   inet 192.168.1.2/32 scope link cilium_host
       valid_lft forever preferred_lft forever
   inet6: fe80::e024:ff:fe00:7096/64 scope link
       valid_lft forever preferred_lft forever

# root@node-122 ~> ip a
d: cilium_host:cilium_net: <BROADCAST,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP qlen 1000
   link/ether 00:0c:29:00:00:00 brd ff:ff:ff:ff:ff:ff
   inet 192.168.2.2/32 scope link cilium_host
       valid_lft forever preferred_lft forever
   inet6: fe80::e024:ff:fe00:7096/64 scope link
       valid_lft forever preferred_lft forever
```

'cilium_vxlan' interface is in Metadata mode. No IP assigned to this interface.
Pod-to-Pod Communication

- VETH Interface with “lxc-xxx” got created on the root namespace
- Other end is attached to the pod namespace
Cilium

Datapath

- Cilium datapath uses eBPF hooks to load BPF programs
- XDP BPF hook is at the earliest point possible in the networking driver and triggers a run of the BPF program upon packet reception
- Traffic Control (TC) Hooks: BPF programs are attached to the TC Ingress hook of host side of the VETH pair for monitoring & enforcement

XDP Hooks

TC Ingress Hooks
**Cilium**

**ARP Resolution**

- Default GW of the container is pointing to the IP of "cilium_host"
- BPF Program is installed to reply to the ARP request
- LXC Interface MAC is used for the ARP reply

```
[root@node-18121 ~]# tcpdump -i lxc97b73386e5f -nn
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on lxc97b73386e5f, link-type EN10MB (Ethernet), capture size 262144 bytes
01:11:02.484685 ARP, Request who-has 192.168.1.1 tell 192.168.1.22, length 28
```

BPF program is responding with to the ARP with VETH's (lxc-x) MAC

```
[root@cilium-node-18121 ~]# docker exec -it ed8ba415daac /bin/sh
/# route
Kernel IP routing table
Destination     Gateway         Genmask Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     0.0.0.0   UG   0      0      0 eth0
192.168.1.1     0.0.0.0         255.255.255.255 U      0      0      0 eth0
/#
```

Default GW of the pod is pointing to 'cilium_host' IP

```
```
Cilium

Packet Flow

- **cilium_vxlan** device does VXLAN encap/decap
- Traffic from pod1 is going through "cilium_vxlan" device before exiting

```
• cilium_vxlan device does VXLAN encap/decap
• Traffic from pod1 is going through “cilium_vxlan” device before exiting
```
Cilium

Direct Routing

- No VXLAN/GENEVE overlays
- As an admin you are able to run your own flavor of routing daemon (Bird/BGPD/Zebra etc) to distribute routes
- You can make the Pod IP’s routable as a result
K8S Networking: Basics

Services

- Pods are mortal
- Need a higher level abstractions: Services
- "Service" in Kubernetes is a conceptual concept. Service is not a process/daemon. Outside networks doesn't learn Service IP addresses
- Implemented through Kube Proxy with IPTables rules

```
$ kubectl get services -n demo -o wide
NAME      TYPE     CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
hostnames ClusterIP 10.96.13.117 <none>     80/TCP      23h
```

```
$ kubectl describe service -n demo
Name:      hostnames
Namespace: demo
Labels:    <none>
Annotations: <none>
Selector:  app=hostnames
Type:      ClusterIP
IP:        10.96.13.117
Port:      default: 80/TCP
TargetPort: 9376/TCP
Session Affinity: None
Events:    <none>
```
K8S Networking: Basics

Exposing Services

- If Services are an abstracted concept without any meaning outside of the K8S cluster how do we access?
- NodePort / LoadBalancer / Ingress etc.

NodePort: Service is accessed via ‘NodeIP:port’

Credit: https://medium.com/google-cloud/kubernetes-nodeport-vs-loadbalancer-vs-ingress-when-should-i-use-what-922f010849e0
Load Balancer: Spins up a load balancer and binds service IPs to Load Balancer VIP

- Very common in public cloud environments
- For baremetal workloads: *MetalLB* (Up & coming project, load-balancer implementation for bare metal K8S clusters, using standard routing protocols)

Exposing Services

Credit: https://medium.com/google-cloud/kubernetes-nodeport-vs-loadbalancer-vs-ingress-when-should-i-use-what-922f010849e0
K8S Networking: Basics

Exposing Services

- **Ingress**: K8S Concept that lets you decide how to let traffic into the cluster
- Sits in front of multiple services and act as a "router"
- Implemented through an ingress controller (NGINX/HA Proxy)

Credit: https://medium.com/google-cloud/kubernetes-nodeport-vs-loadbalancer-vs-ingress-when-should-i-use-what-922f010849e0
K8S Networking: Basics

Exposing Services

- **Ingress**: Network (“Abstracted Network”) can really help you out here
- Ingress controllers are deployed in some of your “public” nodes in your cluster
- Eg: **Big Cloud Fabric** (by Big Switch), can expose a Virtual IP in front of the Ingress Controllers and perform Load Balancing/Health Checks/Analytics

```bash
$ kubectl get pods --all-namespaces --o wide
kube-system      nginx-ingress-controller-5b95b96fb7-r6ltq   1/1  Running  10.18.62.1   kube-1862
kube-system      nginx-ingress-controller-5b95b96fb7-vj86x   1/1  Running  10.18.63.1   kube-1863
```
Thanks!

• Repo for all the command outputs/PDF slides: https://github.com/jayakody/ons-2019

• Credits:
  • Inspired by Life of a Packet- Michael Rubin, Google (https://www.youtube.com/watch?v=0Omvgd7Hq1I)
  • Sarath Kumar, Prashanth Padubidry : Big Switch Engineering
  • Thomas Graf, Dan Wendlandt: Isovalent (Cilium Project)
  • Project Calico Slack Channel: special shout out to: Casey Davenport, Tigera
  • And so many other folks who took time to share knowledge around this emerging space through different mediums (Blogs/YouTube videos etc)