Kubernetes Container Networking

Frank Brockners
Questions?
Use Cisco Spark to communicate with the speaker after the session

How
1. Find this session in the Cisco Live Mobile App
2. Click “Join the Discussion”
3. Install Spark or go directly to the space
4. Enter messages/questions in the space

cs.co/ciscolivebot#BRKDCN-2390
What is Kubernetes (K8s)?

• Container orchestrator
• Runs and manages containers
• Supports multiple cloud and bare-metal environments
• Inspired and informed by Google's experiences and internal systems
• 100% Open source, written in Go
• Manage applications, not machines
• Rich ecosystem of plug-ins for scheduling, storage, networking
Agenda

• Kubernetes Networking
  • Basic Concepts
  • Putting concepts to work – Kubernetes Networking in Examples
  • Container Networking Interface (CNI)

• The Evolution
  • Making container networking fast and scalable: Contiv-VPP plugin
  • Cloud Native Container Networking: Ligato
  • Mesh Networking: Istio
Towards Kubernetes Networking
Basic Concepts
Nodes, Pods, Containers

- **Node:**
  - A server

- **Cluster:**
  - Collection of nodes

- **Pod:**
  - Collection of containers; Nodes can run multiple Pods
Nodes, Pods, Containers, Labels

• Labels:
  • key/value pairs applied to container

• Pod addressing/reachability
  • Each pod gets exactly 1 interface with exactly 1 IP
  • All containers in Pod share a localhost
  • All Pods (subject to network policy) can talk to each other via L3
  • Containers within a Pod share a common networking namespace

• Nodes
  • are assigned unique IP subnets
Example Setup with the Kubernetes default bridge cbr0

"Fabric": logical "router" (provided by CNI)
Example Setup with the Kubernetes: Pod Addressing

“Fabric”: logical “router” (provided by CNI)

Node 1
- eth0
- cbr0
- veth
- host/root ns
- eth0
- 10.100.10.10
- POD 1
- 10.100.10.20
- POD 2

Node 2
- eth0
- cbr0
- veth
- host/root ns
- eth0
- 10.100.20.10
- POD 1
- 10.100.20.20
- POD 2

Node 3
- eth0
- cbr0
- veth
- host/root ns
- eth0
- 10.100.30.10
- POD 1
- 10.100.30.20
- POD 2

Black address managed by operator
Blue address managed by K8s
Namespaces

- K8s namespaces represent Linux ‘namespaces’
- K8s namespaces replace the need for having a ‘tenant’ concept

```bash
$ kubectl get namespaces --show-labels
NAME       STATUS   AGE       LABELS
default    Active   32m       <none>
devlopment Active   29s       name=development
production Active   23s       name=production
```
Selectors

- Selectors are
  - simple
  - powerful
  - can do everything you’re used to do with EPGs

```
$ kubectl get pods -l version=v2

$ kubectl get pods -l version=v2,app=ciscoliveweb
```
Example: Labels and Selectors

```yaml
# Please edit the object below. Lines beginning with a '#' will be ignored, # and an empty file will abort the edit. If an error occurs while saving this file will be # reopened with the relevant failures. #
apiVersion: v1
kind: Service
metadata:
  name: cl-svc-1
  namespace: default
  resourceVersion: "27717"
  selfLink: /api/v1/namespaces/default/services/cl-svc-1
  uid: 52e4cde2-d134-11e7-b632-0025b500014a
spec:
  clusterIP: 10.111.141.65
  ports:
    - port: 80
      protocol: TCP
      targetPort: web-port
  selector:
    app: ciscoliveweb
    version: v2
status:
  loadBalancer: {}
```

v2 is now active

```
[root@controller-0 k8s]# kubectl edit service/cl-svc-1
service "cl-svc-1" edited
[root@controller-0 k8s]# kubectl get services -o wide
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)     AGE       SELECTOR
cl-svc-1     ClusterIP  10.111.141.65  <none>        80/TCP       1h        app=ciscoliveweb,version=v2
kubernetes   ClusterIP  10.96.0.1     <none>        443/TCP      7h        <none>
```
Services Overview

• “Pods can come and go, services stay”

• Define a single IP/Port combination that provides access to a pool of pods

• By default a service connects the client to a Pod in a round-robin fashion

• This solves the dilemma of having to keep up with every transient IP address assigned by Docker

[Diagram showing a service connected to a pool of pods with different addresses]
Services Overview: Example Setup:
Service Addressing: **10.11.12.13:80**

"Fabric": logical "router" (provided by CNI)

Node 1
- **eth0**
- **cbr0**
- **veth**
- **host/root ns**
- **10.100.10.10**
- **10.100.10.20**

POD 1

POD 2

Node 2
- **eth0**
- **cbr0**
- **veth**
- **host/root ns**
- **10.100.20.10**
- **10.100.20.20**

POD 1

POD 2

Node 3
- **eth0**
- **cbr0**
- **veth**
- **host/root ns**
- **10.100.30.10**
- **10.100.30.20**

POD 1

POD 2

External: 10.20.30.1
Internal: 10.100.10.1 / 24

External: 10.20.30.2
Internal: 10.100.20.1 / 24

External: 10.20.30.3
Internal: 10.100.30.1 / 24

Service Address: **10.11.12.13:80**
Service Overview: Example Setup:

Service Addressing:

Service Address: 10.11.12.13:80

“Fabric”: logical “router” (provided by CNI)

Node 1

- eth0
- cbr0
- veth
- host/root ns
- eth0
- 10.100.10.10
- POD 1

Node 2

- eth0
- cbr0
- veth
- host/root ns
- eth0
- 10.100.20.10
- POD 1

Node 3

- eth0
- cbr0
- veth
- host/root ns
- eth0
- 10.100.30.10
- POD 1

External: 10.20.30.1
Internal: 10.100.10.1 / 24

External: 10.20.30.2
Internal: 10.100.20.1 / 24

External: 10.20.30.3
Internal: 10.100.30.1 / 24
Services Overview: Example Setup: Service Addressing:

Service Address: 10.11.12.13:80

“Fabric”: logical “router” (provided by CNI)

- **Node 1**
  - External: 10.20.30.1
  - Internal: 10.100.10.1 / 24
  - Pod 1: 10.100.10.10
  - Pod 2: 10.100.20.20

- **Node 2**
  - External: 10.20.30.2
  - Internal: 10.100.20.1 / 24
  - Pod 1: 10.100.20.10
  - Pod 2: 10.100.20.20

- **Node 3**
  - External: 10.20.30.3
  - Internal: 10.100.30.1 / 24
  - Pod 1: 10.100.30.10
  - Pod 2: 10.100.30.20

- **Service Address:**
  - Src: 10.100.20.10
  - Dst: 10.11.12.13:80

- **Target Address:**
  - Src: 10.100.20.10
  - Dst: 10.100.30.10:8080
Services Overview: Example Setup:
Service Addressing:

Fabric: logical “router” (provided by CNI)

1. **External**: 10.20.30.1
   - **Internal**: 10.100.10.1 / 24

2. **External**: 10.20.30.2
   - **Internal**: 10.100.20.1 / 24

3. **External**: 10.20.30.3
   - **Internal**: 10.100.30.1 / 24

**Node 1**
- eth0
- cbr0
- veth
- host/root ns
- Pod 1: 10.100.10.10
- Pod 2: 10.100.20.10

**Node 2**
- eth0
- cbr0
- veth
- host/root ns
- Pod 1: 10.100.20.10
- Pod 2: 10.100.20.20

**Node 3**
- eth0
- cbr0
- veth
- host/root ns
- Pod 1: 10.100.30.10
- Pod 2: 10.100.30.20

**Service Address**: 10.11.12.13:80

- **Src**: 10.100.20.10
  - **Dst**: 10.100.20.10

- **Src**: 10.100.20.10
  - **Dst**: 10.100.30.10:8080

- **Src**: 10.100.30.10
  - **Dst**: 10.100.20.10
Services Overview: Example Setup:
Service Addressing:

Service Address: 10.11.12.13:80

“Fabric”: logical “router” (provided by CNI)

External: 10.20.30.1
Internal: 10.100.10.1 / 24

External: 10.20.30.2
Internal: 10.100.20.1 / 24

External: 10.20.30.3
Internal: 10.100.30.1 / 24

Src: 10.100.20.10
Dst: 10.100.30.10:8080

Src: 10.100.20.10
Dst: 10.100.20.10

Src: 10.100.30.10
Dst: 10.100.20.10

Src: 10.100.30.10
Dst: 10.100.30.10:8080

Src: 10.100.20.10
Dst: 10.100.20.10

Src: 10.100.30.10
Dst: 10.100.30.10:8080

Src: 10.100.20.10
Dst: 10.100.20.10

Src: 10.100.30.10
Dst: 10.100.30.10:8080

Src: 10.100.20.10
Dst: 10.100.20.10

Src: 10.100.30.10
Dst: 10.100.30.10:8080

Src: 10.100.20.10
Dst: 10.100.20.10

Src: 10.100.30.10
Dst: 10.100.30.10:8080

Src: 10.100.20.10
Dst: 10.100.20.10

Src: 10.100.30.10
Dst: 10.100.30.10:8080

Src: 10.100.20.10
Dst: 10.100.20.10

Src: 10.100.30.10
Dst: 10.100.30.10:8080
Kubernetes: Services – Deployments, EndPoints

Service Definition

```yaml
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  type: ClusterIP
  selector:
    app: nginx
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 9376
```

Deployment

```yaml
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

Endpoints

```yaml
kind: Endpoints
apiVersion: v1
metadata:
  name: my-service
spec:
  subsets:
    - addresses:
        - ip: 1.2.3.4
        - ip: 4.3.2.1
        - ip: 1.1.1.1
    ports:
      - port: 9376
```

Example: Create a ‘Deployment’ standing up 3 ‘replicas’ (Pods) to provide that service

‘Endpoints’ connect ‘Service’ to ‘location’ via IP/port; K8s creates an endpoint for each Pod

Service Definition (not connected to any running app yet)
Kubernetes: Services – Summary

A Service may LB to multiple Pods

A Pod may provide multiple Services

Logical

Actual

Service1

Endpoint1

Pod1

Selector1: ...

Service2

Endpoint4

Pod4

Selector2: ...

Service3

Endpoint5

Pod5

Selector3: ...

Service4

Endpoint6

Pod6

Selector4: ...

Pod1

Pod2

Pod3
Kubernetes Service, Pods, Replication Controllers, Labels

Front End Service

app=webapp, role=frontend, version=v1

Front End v1 Pod
app=webapp, role=frontend, version=v1

Front End v1 Pod
app=webapp, role=frontend, version=v1

Front End v2 Pod
app=webapp, role=frontend, version=v2

Front End v1 Controller
Desired Count = 2
app=webapp, role=frontend, version=v1

Front End v2 Controller
Desired Count = 1
app=webapp, role=frontend, version=v2
Kubernetes Network Connectivity: Separation of concerns

• Kubernetes clearly separates the different aspects of network connectivity from each other
  1. Cluster IP addressing concept and IP address assignment
  2. Pod to Pod connectivity
  3. Services (within a cluster)
  4. External access to a Pod or service
  5. Forwarding between Pods

• Each of the network connectivity aspects above can be controlled and provisioned independently from each other. E.g.: A Pod can be reached by its native IP address and does not require a service to be provisioned.
Kubernetes Networking

• Highly-coupled container-to-container communications

• Pod-to-Pod communications
  • Kubernetes Network Plugins – base requirements to be implemented:
    • all containers can communicate with all other containers without NAT
    • all nodes can communicate with all containers (and vice-versa) without NAT
    • the IP that a container sees itself as is the same IP that others see it as

• Pod-to-Service communications
  • Kubernetes Service Concept

• External-to-Internal communications
  • Node Port, Load Balancer, External IP

Network plugins determine “how” packets are forwarded. Wide range of plugins available:

• Cilium
• Contiv
• Contrail
• Flannel
• Google Compute Engine
• Nuage VCS
• OVN
• Calico
• Romana
• Weave
• CNI-Genie
• ....
Putting concepts to work...
Our tiny setup
Let’s run a simple POD with 1 container
Creating a Pod with a simple web server image

```
[root@controller-0 k8s]# kubectl run cl-test-1 --labels="test=cl-pod1" --image=brockners/ciscoliveserver-1 --port=80
deployment "cl-test-1" created
```

```
[root@controller-0 k8s]# kubectl get pods -o wide
NAME                         READY     STATUS              RESTARTS   AGE       IP        NODE
cl-test-1-7794869d8b-9cgbw   0/1       ContainerCreating   0          13s       <none>    compute-1.opnfv.org

[root@controller-0 k8s]# kubectl get pods -o wide
NAME                         READY     STATUS    RESTARTS   AGE       IP           NODE
cl-test-1-7794869d8b-9cgbw   1/1       Running   0          1m        10.244.2.5  compute-1.opnfv.org
```
Pod created: We can access our web server now

```
[root@controller-0 k8s]# kubectl get pods -o wide
NAME                         READY     STATUS    RESTARTS   AGE       IP           NODE
cl-test-1-7794869d8b-9cgbw   1/1       Running   0          1m        10.244.2.5 compute-1.opnfv.org

[root@controller-0 k8s]# curl http://10.244.2.5
<!DOCTYPE html>
<html>
    <head>
        <title>CiscoLive</title>
    </head>
    <body>
        <h1>Welcome to CiscoLive - Server 1 - port 80</h1>
    </body>
</html>
```

Let's understand which containers we’re running

<table>
<thead>
<tr>
<th>CONTAINER ID</th>
<th>IMAGE</th>
<th>COMMAND</th>
<th>CREATED</th>
<th>STATUS</th>
<th>PORTS</th>
<th>NAMES</th>
</tr>
</thead>
<tbody>
<tr>
<td>ad013ab8cc2</td>
<td>docker.io/brockners/ciscoliveserver-1@sha256:f2fcf67611ff7502d006d11d95b62cf98d4aa29f910b6b71094888d931dc9b00a</td>
<td>/bin/sh -c 'service '</td>
<td>2 minutes ago</td>
<td>Up 2 minutes</td>
<td>k8s_cl-test-1_cl-test-1-7794869d88-9gcbw_default_7df5e197-d12a-11e7-b632-0025b500014a_0</td>
<td></td>
</tr>
<tr>
<td>52b02b7d78c1</td>
<td>gcr.io/google_containers/pause-amd64:3.0</td>
<td>/pause</td>
<td>3 minutes ago</td>
<td>Up 3 minutes</td>
<td>k8s_POD_cl-test-1-7794869d88-9gcbw_default_7df5e197-d12a-11e7-b632-0025b500014a_0</td>
<td></td>
</tr>
<tr>
<td>538687f67d9c</td>
<td>gcr.io/google_containers/pause-amd64:3.0</td>
<td>/pause</td>
<td>4 hours ago</td>
<td>Up 4 hours</td>
<td>k8s_kube-proxy-kjjtt_kube-system_82174637-d105-11e7-b632-0025b500014a_0</td>
<td></td>
</tr>
<tr>
<td>b6d232b0630a</td>
<td>gcr.io/google_containers/pause-amd64:3.0</td>
<td>/pause</td>
<td>4 hours ago</td>
<td>Up 4 hours</td>
<td>k8s_POD_kube-proxy-kjjtt_kube-system_82174637-d105-11e7-b632-0025b500014a_0</td>
<td></td>
</tr>
<tr>
<td>98900af24b31</td>
<td>gcr.io/google_containers/pause-amd64:3.0</td>
<td>/pause</td>
<td>4 hours ago</td>
<td>Up 4 hours</td>
<td>k8s_POD_kube-proxy-kjjtt_kube-system_82174637-d105-11e7-b632-0025b500014a_0</td>
<td></td>
</tr>
</tbody>
</table>

... there are 2 containers: ciscoliveserver-1 and pause-amd64
Naive approach to linking containers within a Pod

Pod 1:
- Container 1
- Container 2
- Container 3
What if Container 1 would fail?

Pod 1:
- Container 1
- Container 2
- Container 3
The Pause-Container is our gatekeeper and always at home

Pod 1:
- Pause
- Container 1
- Container 2
- Container 3

Pause
10.244.2.5

Container 2

Container 3

Container 2
CiscoLiveServer-1 and pause-amd64 really share the same network namespace...

[root@compute-1 ~]# docker ps
CONTAINER ID        IMAGE
COMMAND                  CREATED             STATUS              PORTS               NAMES
ad013ab8cc22 docker.io/brockners/ciscoliveserver:1@sha256:f2fcf6761ff7502d006d11d95b62cf98d4aa29f910b6b7109488d931dc9b00a
"/bin/sh -c 'service " 2 minutes ago       Up 2 minutes                            k8s_cl-test-1_cl-test-1
9cgbw_default_7df5e197-d12a-1ele7-b632-0025b500014a_0
52b02b7d78c1 gcr.io/google_containers/pause-amd64:3.0
"/pause"                 3 minutes ago       Up 3 minutes                            k8s_POD_cl-test-1
52b02b7d78c1

[root@compute-1 ~]# docker inspect ad013ab8cc22 | grep NetworkMode
"NetworkMode": "container:

[root@compute-1 ~]# docker inspect ad013ab8cc22 | grep Pid
"Pid": 68398,

[root@compute-1 ~]# nsenter -t 68398 -n ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

: eth0@if427: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP mode DEFAULT
    link/ether 0a:58:0a:f4:02:05 brd ff:ff:ff:ff:ff:ff

[root@compute-1 ~]# docker inspect 52b02b7d78c1 | grep Pid
"Pid": 68148,

[root@compute-1 ~]# nsenter -t 68148 -n ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

: eth0@if427: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP mode DEFAULT
    link/ether 0a:58:0a:f4:02:05 brd ff:ff:ff:ff:ff:ff

Ciscoserver!
... and we can verify the assigned IP address

```
[root@controller-0 k8s]# kubectl get pods -o wide
NAME     READY     STATUS      RESTARTS   AGE       IP           NODE
cl-test-1-7794869d8b-9cgbw  1/1     Running   0          12m  10.244.2.5  compute-1.opnfv.org
```

```
[root@compute-1 ~]# nsenter -t 68398 -n ifconfig eth0
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1450
    pmｶ_capacity chubby
    inet 10.244.2.5  netmask 255.255.255.0  broadcast 0.0.0.0
    inet6 fe80::1883:bff:feb4:f13e  prefixlen 64  scopeid 0x20<link>
    ether 0a:58:0a:f4:02:05  txqueuelen 0  (Ethernet)
    RX packets 14  bytes 1066 (1.0 KiB)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 15  bytes 1470 (1.4 KiB)
    TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0
```
This is what we know so far...
ethtool or ip link show will tell us the peer

[root@compute-1 ~]# nsenter -t 68398 -n ethtool -S eth0
NIC statistics:
   peer_ifindex: 427

[root@compute-1 ~]# nsenter -t 68398 -n ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT qlen 1
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
3: eth0@if427: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP mode DEFAULT qlen 1
   link/ether 0a:58:0a:f4:02:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
Our ip link connects to a bridge: cni0

[root@compute-1 ~]# ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT qlen 1
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp6s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq master br-ctlplane state UP mode DEFAULT qlen 1000
   link/ether 00:25:b5:00:01:4d brd ff:ff:ff:ff:ff:ff
5: enp9s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT qlen 1000
   link/ether 00:25:b5:42:02:0d brd ff:ff:ff:ff:ff:ff
6: enp10s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT qlen 1000
   link/ether 00:25:b5:42:02:0c brd ff:ff:ff:ff:ff:ff
7: enp15s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9000 qdisc mq state UP mode DEFAULT qlen 1000
   link/ether 00:25:b5:42:02:1c brd ff:ff:ff:ff:ff:ff

[...]
363: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT
   link/ether 02:42:61:83:31:40 brd ff:ff:ff:ff:ff:ff
422: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN mode DEFAULT
   link/ether aa:ea:72:a3:0a:70 brd ff:ff:ff:ff:ff:ff
423: cni0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP mode DEFAULT qlen 1000
   link/ether 0a:58:0a:f4:02:01 brd ff:ff:ff:ff:ff:ff
427: veth267de2f5@f3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP mode DEFAULT master cni0
   link/ether 8a:19:56:2d:61:47 brd ff:ff:ff:ff:ff:ff linx-netnsid 0
A closer look at the cni0 bridge

```
[root@compute-1 ~]# ifconfig cni0
  cni0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1450
     inet 10.244.2.1 netmask 255.255.255.0 broadcast 0.0.0.0
     inet6 fe80::80fd:40ff:fea1:9617 prefixlen 64 scopeid 0x20<link>
        ether 0a:58:0a:f4:02:01 txqueuelen 1000 (Ethernet)
        RX packets 59  bytes 4286 (4.1 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 36  bytes 3053 (2.9 KiB)
        TX errors 0  dropped 0 overruns 0 carrier 0 collisions 0

[root@compute-1 ~]# ifconfig flannel.1
  flannel.1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1450
     inet 10.244.2.0 netmask 255.255.255.255 broadcast 0.0.0.0
     inet6 fe80::a8ea:72ff:fea3:a70 prefixlen 64 scopeid 0x20<link>
        ether aa:ea:72:a3:0a:70 txqueuelen 0 (Ethernet)
        RX packets 23  bytes 1957 (1.9 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 21  bytes 2070 (2.0 KiB)
        TX errors 0  dropped 5 overruns 0 carrier 0 collisions 0
```
Bridge cni0 is linked via a VXLAN tunnel (setup by Flannel) to the other nodes

```
[root@compute-1 ~]# ip -d l show flannel.1
422: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN mode DEFAULT
    link/ether aa:ea:72:a3:0a:70 brd ff:ff:ff:ff:ff:ff promiscuity 0
        vxlan id 1 local 10.60.17.132 dev enp10s0 srcport 0 0 dstport 8472 nolearning ageing 300 addrgenmode eui64
```
Our learnings added to our wiring picture

- Bridge: cni0
- Flannel.1: 10.244.2.0
- Container ID 422
- Container ID b3a628114695
- Container ID 1c750f990859
- Eth0
- Veth 10.244.2.5
- Netnsid 0
- Pause
Let’s try a POD with 2 Containers
One Pod with 2 containers, each running a web server

- Pause
- ciscolive-1
- ciscolive-2

1
2
Our deployment file: web-pod-2.yaml

```
apiVersion: extensions/v1beta1
type: Deployment
metadata:
  name: web-pod-2
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: ciscolive-1
          image: brockners/ciscoliveserver-1
          ports:
            - containerPort: 80
              name: web-tcp-80
              protocol: TCP
        - name: ciscolive-2
          image: brockners/ciscoliveserver-2
          ports:
            - containerPort: 80
              name: web-tcp-80
              protocol: TCP
```
Only one of our two containers starts properly...

```
[root@controller-0 k8s]# kubectl get deploy
NAME    DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
cl-test-1 1 1 1 1 45m
[root@controller-0 k8s]# kubectl delete deploy cl-test-1
deployment "cl-test-1" deleted

[root@controller-0 k8s]# kubectl create -f web-pod-2.yaml
deployment "web-pod-2" created
[root@controller-0 k8s]# kubectl get pods -o wide
NAME               READY STATUS             RESTARTS AGE   IP           NODE
web-pod-2-5c8d8bf9cc-z96b7 1/2 CrashLoopBackOff 2 52s    10.244.1.6  compute-0.opnfv.org
[root@controller-0 k8s]# kubectl logs web-pod-2-5c8d8bf9cc-z96b7 -c ciscolive-1
  * Starting nginx nginx
[root@controller-0 k8s]# kubectl logs web-pod-2-5c8d8bf9cc-z96b7 -c ciscolive-2
  * Starting nginx nginx
  ...fail!
```
What went wrong?

• Two containers with web servers – both on port 80?

```yaml
spec:
  containers:
  - name: ciscolive-1
    image: brockners/ciscoliveserver-1
    ports:
      - containerPort: 80
        name: web
        protocol: TCP
  - name: ciscolive-2
    image: brockners/ciscoliveserver-2
    ports:
      - containerPort: 80
        name: web
        protocol: TCP
```
Our modified deployment file: web-pod-3.yaml

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: web-pod-3
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: ciscolive-1
        image: brockners/ciscoliveserver-1
        ports:
        - containerPort: 80
          name: web-tcp-80
          protocol: TCP
      - name: ciscolive-3
        image: brockners/ciscoliveserver-3
        ports:
        - containerPort: 8080
          name: web-tcp-8080
          protocol: TCP
```
This time with success

[root@controller-0 k8s]# kubectl delete deploy web-pod-2
deployment "web-pod-2" deleted

root@controller-0 k8s]# kubectl create -f web-pod-3.yaml
deployment "web-pod-3" created
[root@controller-0 k8s]# kubectl get pods -o wide

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
<th>IP</th>
<th>NODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>web-pod-3-57c99668d9-r2fbj</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>21s</td>
<td>10.244.2.6</td>
<td>compute-1.opnfv.org</td>
</tr>
</tbody>
</table>
Verification: We have 3 containers in our Pod

[root@compute-1 ~]# docker ps
CONTAINER ID        IMAGE                                                                 COMMAND                  CREATED             STATUS              PORTS               NAMES
9fc7f67f770c       docker.io/brockners/ciscoliveserver-3@sha256:288035c90198f6e146c90382895d62bf89f2eaaf96c5b76373f6effa07df0f7f   "/bin/sh -c 'service "   4 minutes ago       Up 4 minutes                            k8s_ciscolive-3_web-pod-3-57c99668d9-
2r2fbj_default_a5034786-d131-11e7-b632-0025b500014a_0                 3e6068230d9c      docker.io/brockners/ciscoliveserver-1@sha256:f2fc67611ff7502d006d11d95b62cf98d4aa29f910b6b7109488d931dc9b00a   "/bin/sh -c 'service "   4 minutes ago       Up 4 minutes                            k8s_ciscolive-1_web-pod-3-57c99668d9-
2r2fbj_default_a5034786-d131-11e7-b632-0025b500014a_0 50661e2c29e6       gcr.io/google_containers/pause-amd64:3.0       "/pause"                 4 minutes ago       Up 4 minutes                            k8s_POD_web-pod-3-57c99668d9-r2fbj_default_a5034786-d131-11e7-b632-0025b500014a_0  
538687f67d9c       quay.io/coreos/flannel@sha256:056cf57fd3bbe7264c0be1a3b34ec2e289b33e51c70f332f4e88a83970ad91    "/opt/bin/flanneld --    5 hours ago         Up 5 hours                            k8s_kube-flannel_kube-flannel-ds-vf5kf_kube-system_821748d0-d105-11e7-b632-0025b500014a_0  
50661e2c29e6       gcr.io/google_containers/kube-proxy-amd64@sha256:8dce98802846d5219093f0313dcb8697a8a5d7cad647c3b71a816cf3a2b2eb2a   "/usr/local/bin/kube-    5 hours ago         Up 5 hours                            k8s kube-proxy_kube-proxy-kjtt_kube-system_82174637-d105-11e7-b632-0025b500014a_0  
9b6d232b0630a      gcr.io/google_containers/pause-amd64:3.0       "/pause"                 5 hours ago         Up 5 hours                            k8s_POD_kube-proxy-kjtt_kube-system_82174637-d105-11e7-b632-0025b500014a_0  
98900af24b31       gcr.io/google_containers/pause-amd64:3.0       "/pause"                 5 hours ago         Up 5 hours                            k8s_POD_kube-flannel-ds-vf5kf_kube-system_821748d0-d105-11e7-b632-0025b500014a_0  
98900af24b31       gcr.io/google_containers/pause-amd64:3.0       "/pause"                 5 hours ago         Up 5 hours                            k8s_POD_kube-flannel-ds-vf5kf_kube-system_821748d0-d105-11e7-b632-0025b500014a_0  

[root@compute-1 ~]# docker inspect 9fc7f67f770c | grep NetworkMode
"NetworkMode": "container:50661e2c29e61512586c7573d151c6d722161aa2b45843a6825eba353c5f23a",
[root@compute-1 ~]# docker inspect 3e6068230d9c | grep NetworkMode
"NetworkMode": "container:50661e2c29e61512586c7573d151c6d722161aa2b45843a6825eba353c5f23a",
[root@compute-1 ~]#
Pause

ciscolive-1  ciscolive-3

1  3

eth0: 10.244.2.6

veth2594a3ca
Kubernetes Services and how they are wired
A Service is your friendly representative for eventually a lot of workers (read PODs)
Services

• “Pods can come and go, services stay”

• Define a single IP/Port combination that provides access to a pool of pods

• By default a service connects the client to a Pod in a round-robin fashion

• This solves the dilemma of having to keep up with every transient IP address assigned by Docker
Let’s choose a new deployment: cl-deploy-1.yaml

```yaml
apiVersion: extensions/v1beta1
class: Deployment
metadata:
  name: cl-deploy-1
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: ciscoliveweb
        version: v1
    spec:
      containers:
        - name: ciscolive-3
          image: brockners/ciscoliveserver-3
          ports:
            - containerPort: 8080
              name: web-port
              protocol: TCP
```

Deploying cl-deploy-1.yaml:
We’ve started yet another web server

```
[root@controller-0 k8s]# kubectl create -f cl-deploy-1.yaml
deployment "cl-deploy-1" created
[root@controller-0 k8s]# kubectl get pods -o wide
NAME                        READY     STATUS    RESTARTS AGE       IP           NODE
cl-deploy-1-cb8b886bb-vj9bs 1/1       Running   0          2m        10.244.1.7 compute-0.opnfv.org
[root@controller-0 k8s]# curl http://10.244.1.7
curl: (7) Failed connect to 10.244.1.7:80; Connection refused
[root@controller-0 k8s]# curl http://10.244.1.7:8080
<!DOCTYPE html>
<html>
<head>
<title>CiscoLive</title>
</head>
<body>
<h1>Welcome to CiscoLive - Server 3 - port 8080</h1>
</body>
</html>
[root@controller-0 k8s]#
```
Defining a service: cl-svc-1.yaml

```yaml
kind: Service
apiVersion: v1
metadata:
  name: cl-svc-1
spec:
  selector:
    app: ciscoliveweb
  ports:
    - protocol: TCP
      port: 80
      targetPort: web-port
```
Accessing our Service

[root@controller-0 k8s]# kubectl create -f cl-svc-1.yaml
service "cl-svc-1" created
[root@controller-0 k8s]# kubectl get services -o wide

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CLUSTER-IP</th>
<th>EXTERNAL-IP</th>
<th>PORT(S)</th>
<th>AGE</th>
<th>SELECTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>cl-svc-1</td>
<td>ClusterIP</td>
<td>10.111.141.65</td>
<td>&lt;none&gt;</td>
<td>80/TCP</td>
<td>6m</td>
<td>app=ciscoliveweb</td>
</tr>
<tr>
<td>kubernetes</td>
<td>ClusterIP</td>
<td>10.96.0.1</td>
<td>&lt;none&gt;</td>
<td>443/TCP</td>
<td>5h</td>
<td>&lt;none&gt;</td>
</tr>
</tbody>
</table>

[root@controller-0 k8s]# curl http://10.111.141.65
<!DOCTYPE html>
<html>
<head>
  <title>CiscoLive</title>
</head>
<body>
  <h1>Welcome to CiscoLive - Server 3 - port 8080</h1>
</body>
</html>
IP-addresses are nice, names are nicer:

```
[root@controller-0 k8s]# curl http://cl-svc-1
curl: (6) Could not resolve host: cl-svc-1; Name or service not known
```
.. but wait: We were not in the right namespace...

Let’s create ourselves an instance within the namespace of the POD
Let's create ourselves a “tools” container

```
tools.yaml

apiVersion: v1
kind: Pod
metadata:
  name: tools
  namespace: default
spec:
  containers:
  - image: brockners/tools:latest
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
    name: tools
    restartPolicy: Always
```
And now we can access the service by name

```
[root@controller-0 k8s]# kubectl create -f tools.yaml
pod "tools" created
[root@controller-0 k8s]# kubectl get pods
NAME          READY STATUS    RESTARTS AGE
cl-deploy-1-cb8b886bb-vj9bs   1/1   Running   0   18m
tools         1/1   Running   0   12s
[root@controller-0 k8s]# kubectl exec -ti tools -- curl http://cl-svc-1
<!DOCTYPE html>
<html>
<head>
  <title>CiscoLive</title>
</head>
<body>
  <h1>Welcome to CiscoLive - Server 3 - port 8080</h1>
</body>
</html>
[root@controller-0 k8s]#
```
kubedns does the magic for us...

```
[root@controller-0 k8s]# kubectl exec -ti tools -- nslookup cl-svc-1  
Server: 10.96.0.10  
Address: 10.96.0.10#53

Name: cl-svc-1.default.svc.cluster.local  
Address: 10.111.141.65
```

- Pods are assigned a DNS A record in the form of “pod-ip-address.my-namespace.pod.cluster.local”.  

```
kubedns

• Kubernetes DNS schedules a DNS Pod and Service on the cluster, and configures the kubelets to tell individual containers to use the DNS Service’s IP to resolve DNS names.

• Every Service defined in the cluster (including the DNS server itself) is assigned a DNS name. By default, a client Pod’s DNS search list will include the Pod’s own namespace and the cluster’s default domain.

• Example:
  • Assume a Service named foo in the Kubernetes namespace bar.
  • A Pod running in namespace bar can look up this service by simply doing a DNS query for foo.
  • A Pod running in namespace quux can look up this service by doing a DNS query for foo.bar.
How does forwarding between our tools pod and the web-pod work? How are things wired together?
Our tiny setup

```
[root@controller-0 k8s]# kubectl get pods -o wide
NAME                     READY     STATUS   RESTARTS  AGE       IP           NODE
cl-deploy-1-cb8b886bb-vj9bs 1/1       Running   0   23m       10.244.1.7   compute-0.opnfv.org
tools                    1/1       Running   0   5m        10.244.2.7   compute-1.opnfv.org
```
2 Pods (1 x Web-server, 1 x Tools)

### cl-deploy-1

<table>
<thead>
<tr>
<th>CONTAINER ID</th>
<th>IMAGE</th>
<th>COMMAND</th>
<th>CREATED</th>
<th>STATUS</th>
<th>PORTS</th>
<th>NAMES</th>
</tr>
</thead>
<tbody>
<tr>
<td>86f78f83b974</td>
<td>docker.io/brockners/ciscoliveserver-3</td>
<td>/bin/sh -c 'service &quot;</td>
<td>43 minutes ago</td>
<td>Up 43 minutes</td>
<td>k8s_ciscolive-3_cl-deploy-1-cb8b886bb-vj9bs_default_96f11a38-d133-11e7-b632-0025b500014a_0</td>
<td></td>
</tr>
<tr>
<td>15db6d859aa7</td>
<td>gcr.io/google_containers/pause-amd64:3.0</td>
<td>&quot;/pause&quot;</td>
<td>44 minutes ago</td>
<td>Up 43 minutes</td>
<td>k8s_POD_cl-deploy-1-cb8b886bb-vj9bs_default_96f11a38-d133-11e7-b632-0025b500014a_0</td>
<td></td>
</tr>
</tbody>
</table>

### tools

<table>
<thead>
<tr>
<th>CONTAINER ID</th>
<th>IMAGE</th>
<th>COMMAND</th>
<th>CREATED</th>
<th>STATUS</th>
<th>PORTS</th>
<th>NAMES</th>
</tr>
</thead>
<tbody>
<tr>
<td>docker.io/brockners/tools@sha256:0009252d684535df121984a38ecdd8b38f19e70ae9e0ec95796ce4dc5d9e5a60f</td>
<td>gcr.io/google_containers/pause-amd64:3.0</td>
<td>&quot;sleep 3600&quot;</td>
<td>26 minutes ago</td>
<td>Up 26 minutes</td>
<td>k8s_tools_tools_default_328d53d6-d136-11e7-b632-0025b500014a_0</td>
<td></td>
</tr>
<tr>
<td>29075c12c761</td>
<td>gcr.io/google_containers/pause-amd64:3.0</td>
<td>&quot;/pause&quot;</td>
<td>26 minutes ago</td>
<td>Up 26 minutes</td>
<td>k8s_POD_tools_default_328d53d6-d136-11e7-b632-0025b500014a_0</td>
<td></td>
</tr>
<tr>
<td>f3bcae70c092</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Understanding the wiring of our containers

<table>
<thead>
<tr>
<th>cl-deploy-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>[root@compute-0 ~]# ip link show</td>
</tr>
<tr>
<td>[..]</td>
</tr>
<tr>
<td>430: vethb2542a3f@if3: &lt;BROADCAST,MULTICAST,UP,LOWER_UP&gt; mtu 1450 qdisc noqueue master cni0 state UP mode DEFAULT</td>
</tr>
<tr>
<td>link/ether 9e:f9:06:f4:cc:39 brd ff:ff:ff:ff:ff:ff link-netnsid 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>[root@compute-1 ~]# ip link show</td>
</tr>
<tr>
<td>[..]</td>
</tr>
<tr>
<td>429: vethdcf61296@if3: &lt;BROADCAST,MULTICAST,UP,LOWER_UP&gt; mtu 1450 qdisc noqueue master cni0 state UP mode DEFAULT</td>
</tr>
<tr>
<td>link/ether 0a:5c:54:d2:d9:22 brd ff:ff:ff:ff:ff:ff link-netnsid 0</td>
</tr>
</tbody>
</table>
Our tiny setup

NODE: Compute-1

POD: tools

Container: 69170738e986
Image: docker.io/brockners/tools

eth0: 10.244.2.7

429 vethdcf61296

cni0

NODE: Compute-0

POD: cl-deploy-1

Container: 86f78f83b974
Image: docker.io/brockners/ciscoliveserver-3

eth0: 10.244.1.7

430 vethb2542a3f

cni0
kubectl exec -ti tools -- curl http://cl-svc-1

```
[root@controller-0 k8s]# kubectl exec -ti tools -- curl http://cl-svc-1
<!DOCTYPE html>
<html>
<head>
  <title>CiscoLive</title>
</head>
<body>
  <h1>Welcome to CiscoLive - Server 3 - port 8080</h1>
</body>
</html>
```

.... How are the packets forwarded?
17:38:12.243782 IP 10.244.2.7.36372 > 10.111.141.65.80: [..]
17:38:12.243853 IP 10.111.141.65.80 > 10.244.2.7.36372: [..]
Noting what we learned about compute-1 in our diagram...

**NODE: Compute-1**

POD: tools

- Container: 69170738e986
- Image: docker.io/brockners/tools

**NODE: Compute-0**

POD: cl-deploy-1

- Container: 86f78f83b974
- Image: docker.io/brockners/ciscoliveserver-3

- eth0: 10.244.1.7

- vethb2542a3f

- cni0

- 430

- 10.244.2.7 > 10.111.141.65.80

- 429

- vethdcf61296

- cni0

- eth0 10.244.2.7
Compute-0 (cl-deploy-1)
vethb2542a3f

```
[roo@compute-0 ~]# tcpdump -i vethb2542a3f -nn
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on vethb2542a3f, link-type EN10MB (Ethernet), capture size 262144 bytes
17:38:12.024665 IP 10.244.2.7.36372 > 10.244.1.7.8080: Flags [S], seq 2523706886, win 28200, options [mss 1410,sackOK,TS val 3911279006,ecr 0,nop,wscale 7], length 0
17:38:12.024496 IP 10.244.1.7.8080 > 10.244.2.7.36372: Flags [S], seq 1376633030, ack 2523706887, win 27960, options [mss 1410,sackOK,TS val 3918634415,ecr 3911279006,wscale 7], length 0
17:38:12.024613 IP 10.244.2.7.36372 > 10.244.1.7.8080: Flags [..], ack 1, win 221, options [nop,ns,wscale 7], length 0
17:38:12.024634 IP 10.244.2.7.36372 > 10.244.1.7.8080: Flags [..], seq 1:73, ack 1, win 221, options [nop,ns,ts val 3911279006,ecr 3918634415,wscale 7], length 72: HTTP: GET / HTTP/1.1
17:38:12.024652 IP 10.244.1.7.8080 > 10.244.2.7.36372: Flags [..], seq 73, ack 73, win 219, options [nop,ns,ts val 3918634415,ecr 3911279006,wscale 7], length 0
17:38:12.024684 IP 10.244.1.7.8080 > 10.244.2.7.36372: Flags [..], seq 247, ack 247, win 219, options [nop,ns,ts val 3918634415,ecr 3911279006,wscale 7], length 156: HTTP: HTTP/1.1 200 OK
17:38:12.024782 IP 10.244.2.7.36372 > 10.244.1.7.8080: Flags [..], seq 403, ack 403, win 238, options [nop,ns,ts val 3918634415,ecr 3911279007,wscale 7], length 0
17:38:12.024803 IP 10.244.1.7.8080 > 10.244.2.7.36372: Flags [..], seq 73, ack 74, win 238, options [nop,ns,ts val 3918634415,ecr 3911279007,wscale 7], length 0
17:38:12.024910 IP 10.244.2.7.36372 > 10.244.1.7.8080: Flags [..], seq 404, win 238, options [nop,ns,ts val 3918634415,ecr 3911279007,wscale 7], length 0
17:38:12.024964 IP 10.244.2.7.36372 > 10.244.1.7.8080: Flags [..], seq 404, win 238, options [nop,ns,ts val 3918634415,ecr 3911279007,wscale 7], length 0
```

17:38:12.024634 IP 10.244.2.7.36372 > 10.244.1.7.8080: [..]: HTTP: GET / HTTP/1.1
17:38:12.024652 IP 10.244.1.7.8080 > 10.244.2.7.36372: [..]
Noting what we learned about compute-0 in our diagram

NODE: Compute-1
POD: tools
   Container: 69170738e986
   Image: docker.io/brockners/tools
   eth0: 10.244.2.7
   10.244.2.7.36372 > 10.111.141.65.80
   429 vethdcf61296
cni0

NODE: Compute-0
POD: cl-deploy-1
   Container: 86f78f83b974
   Image: docker.io/brockners/ciscoliveserver-3
   eth0: 10.244.1.7
   10.244.2.7.36372 > 10.244.1.7.8080
   430 vethb2542a3f
cni0
Two snippets of the same flow...

...iptables/netfilter rules connect the dots
```
-A KUBE-SEP-OV6T5PWVZVLT7G -p tcp -m comment --comment "default/clsvc-1:" -m tcp -j DNAT --to-destination 10.244.1.7:8080

-A KUBE-SEP-7B15503Q94Z5M84 -p tcp -m comment --comment "default/kuberneteshttps" -m recent --sec --name KUBE-SEP-7B15503Q94Z5M84 -mask 255.255.255.255 --rsource -m tcp -j DNAT --to-destination 10.17.186.201

-A KUBE-SEP-1ZST1525F0Y478F703F -p tcp -m comment --comment "default/kube-system" -m tcp -j KUBE-MARK-MASQ

-A KUBE-SEP-2ZT2B6720TX478F703F -p tcp -m comment --comment "default/kube-system" -m tcp -j KUBE-MARK-MASQ

-A KUBE-SEP-2ZT2B6720TX478F703F -p tcp -m comment --comment "default/kube-system" -m tcp -j KUBE-MARK-MASQ

-A KUBE-SEP-7B15503Q94Z5M84 -p tcp -m comment --comment "default/kuberneteshttps" -m recent --sec --name KUBE-SEP-7B15503Q94Z5M84 -mask 255.255.255.255 --rsource -m tcp -j DNAT --to-destination 10.17.186.201

-A KUBE-SEP-7B15503Q94Z5M84 -p tcp -m comment --comment "default/kuberneteshttps" -m recent --sec --name KUBE-SEP-7B15503Q94Z5M84 -mask 255.255.255.255 --rsource -m tcp -j DNAT --to-destination 10.17.186.201

-A KUBE-SEP-2ZT2B6720TX478F703F -p tcp -m comment --comment "default/kube-system" -m tcp -j KUBE-MARK-MASQ

-A KUBE-SEP-2ZT2B6720TX478F703F -p tcp -m comment --comment "default/kube-system" -m tcp -j KUBE-MARK-MASQ

-A KUBE-SEP-7B15503Q94Z5M84 -p tcp -m comment --comment "default/kuberneteshttps" -m recent --sec --name KUBE-SEP-7B15503Q94Z5M84 -mask 255.255.255.255 --rsource -m tcp -j DNAT --to-destination 10.17.186.201

-A KUBE-SEP-7B15503Q94Z5M84 -p tcp -m comment --comment "default/kuberneteshttps" -m recent --sec --name KUBE-SEP-7B15503Q94Z5M84 -mask 255.255.255.255 --rsource -m tcp -j DNAT --to-destination 10.17.186.201

-A KUBE-SEP-7B15503Q94Z5M84 -p tcp -m comment --comment "default/kuberneteshttps" -m recent --sec --name KUBE-SEP-7B15503Q94Z5M84 -mask 255.255.255.255 --rsource -m tcp -j DNAT --to-destination 10.17.186.201
```

Destination NAT

NODE: Compute-1
POD: tools
Container: 69170738e986
Image: docker.io/brockners/tools
eth0: 10.244.2.7
10.244.2.7.36372 > 10.111.141.65.80
429 vethdf61296
cni0

DNAT: 10.111.141.65:80 to 10.244.1.7:8080

NODE: Compute-0
POD: cl-deploy-1
Container: 86f78f83b974
Image: docker.io/brockners/ciscoliveserver-3
eth0: 10.244.1.7
10.244.2.7.36372 > 10.244.1.7.8080
430 vethb2542a3f
cni0

Note: vethb2542a3f and vethdcf61296 are logical interfaces created by CNI (Container Network Interface) to manage network connections between pods and hosts.
Scaling the deployment

```
[root@controller-0 k8s]# kubectl scale --replicas=2 deployment/cl-deploy-1
deployment "cl-deploy-1" scaled
[root@controller-0 k8s]# kubectl get pods -o wide

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
<th>IP</th>
<th>NODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>cl-deploy-1-cb8b886bb-vj9bs</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>1h</td>
<td>10.244.1.7</td>
<td>compute-0.opnfv.org</td>
</tr>
<tr>
<td>cl-deploy-1-cb8b886bb-vrsxx</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>13s</td>
<td>10.244.2.8</td>
<td>compute-1.opnfv.org</td>
</tr>
<tr>
<td>tools</td>
<td>1/1</td>
<td>Running</td>
<td>1</td>
<td>1h</td>
<td>10.244.2.7</td>
<td>compute-1.opnfv.org</td>
</tr>
</tbody>
</table>
```
-A KUBE-SEP-GJ7P3DFCFPJ2TWHE -p tcp -m comment --comment "default/cl-svc-1:" -m tcp -j DNAT --to-destination 10.244.2.8:8080

-A KUBE-SEP-OVW6T5PVWZVLXT7G -p tcp -m comment --comment "default/cl-svc-1:" -m tcp -j DNAT --to-destination 10.244.1.7:8080

-A POSTROUTING -s 10.244.0.0/16 !d 10.244.0.0/16 -j RETURN
-A POSTROUTING -s 10.244.0.0/16 !d 224.0.0.0/4 -j MASQUERADE
-A POSTROUTING ! -s 10.244.0.0/16 !d 10.244.0.254 -j RETURN
-A POSTROUTING ! -s 10.244.0.0/16 !d 10.244.0.0/16 -j MASQUERADE
-A DOCKER -i docker0 -j RETURN
-A KUBE-MARK-DRP --j MARK --set-mark 0x8000/0x8000
-A KUBE-MARK-MASQ --j MARK --set-mark 0x9000/0x9000
-A KUBE-POSTROUTING -m comment --comment "kubernetes service traffic requiring SNAT" -m mark --mark 0x4000/0x4000 -j MASQUERADE
-A KUBE-SEP-TKBT5SSDU7JHOBN4 -p udp -m comment --comment "default/kubernetes:https" -j KUBE-MARK-MASQ
-A KUBE-SEP-TKBT5SSDU7JHOBN4 -p tcp -m comment --comment "default/kubernetes:https" -m mark --mark 255.255.255.255 -j DNAT --to-destination 10.60.15.21:4843
-A KUBE-SEP-GJ7P3DFCFPJ2TWHE -p tcp -m comment --comment "default/cl-svc-1:" -m tcp -j DNAT --to-destination 10.244.2.8:8080
-A KUBE-SEP-GJ7P3DFCFPJ2TWHE -p tcp -m comment --comment "kube-system/kube-dns:tcp" -j KUBE-MARK-MASQ
-A KUBE-SEP-GJ7P3DFCFPJ2TWHE -p tcp -m comment --comment "kube-system/kube-dns:udp" -j KUBE-MARK-MASQ
-A KUBE-SEP-GJ7P3DFCFPJ2TWHE -p tcp -m comment --comment "default/cl-svc-1:" -m tcp -j DNAT --to-destination 10.244.1.7:8080
-A KUBE-SEP-GJ7P3DFCFPJ2TWHE -p tcp -m comment --comment "kube-system/kube-dns:tcp" -j KUBE-MARK-MASQ
-A KUBE-SEP-GJ7P3DFCFPJ2TWHE -p tcp -m comment --comment "kube-system/kube-dns:udp" -j KUBE-MARK-MASQ
-A KUBE-SERVICES -s 10.244.0.0/16 !d 10.96.0.0/16 -p tcp -m comment --comment "default/kubernetes:https" -m tcp -j DNAT --to-destination 10.244.21:443
-A KUBE-SERVICES -s 10.244.0.0/16 !d 10.96.0.0/16 -p udp -m comment --comment "default/kubernetes:https" -m udp -j DNAT --to-destination 10.244.21:443
-A KUBE-SERVICES -s 10.244.0.0/16 !d 10.96.0.0/16 -p tcp -m comment --comment "kube-system/kube-dns:cluster IP" -m tcp --dport 53 -j KUBE-MARK-MASQ
-A KUBE-SERVICES -s 10.244.0.0/16 !d 10.96.0.0/16 -p udp -m comment --comment "kube-system/kube-dns:cluster IP" -m udp --dport 53 -j KUBE-MARK-MASQ

[<root@compute-1 ~]# iptables-save
[...]
*nat
:PREROUTING ACCEPT [28:1204]
:INPUT ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]
:DOCKER - [0:0]
:kUBE-MARK-DRP - [0:0]
:kUBE-MARK-MASQ - [0:0]
:kUBE-POSTROUTING - [0:0]
:kUBE-SEP-TKBT5SSDU7JHOBN4 - [0:0]
:kUBE-SEP-TI2ZBSY6T4EKFDTO - [0:0]
:kUBE-SEP-OVW6T5PVWZVLXT7G - [0:0]
Our tiny setup

cl-deploy-1
10.224.1.7

Compute-0
10.60.17.254
10.60.17.140
10.60.17.154

tools
10.224.2.7

cl-deploy-1
10.224.2.8

Compute-1
10.60.17.132

Controller-0
10.60.17.140
Simple external access to our service: ExternallIP/ClusterIP
Edit the service to add an external IP address:

```
kubectl edit service cl-svc-1
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: cl-svc-1
  namespace: default
spec:
  clusterIP: 10.111.141.65
  externalIPs:
  - 10.60.17.100
  ports:
  - port: 80
    protocol: TCP
    targetPort: web-port
  selector:
    app: ciscoliveweb
    version: v2
  sessionAffinity: None
```
```
kubectl get service -o wide

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CLUSTER-IP</th>
<th>EXTERNAL-IP</th>
<th>PORT(S)</th>
<th>AGE</th>
<th>SELECTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>cl-svc-1</td>
<td>ClusterIP</td>
<td>10.111.141.65</td>
<td>10.60.17.100</td>
<td>80/TCP</td>
<td>1d</td>
<td>app=ciscoliveweb,version=v2</td>
</tr>
<tr>
<td>kubernetes</td>
<td>ClusterIP</td>
<td>10.96.0.1</td>
<td>&lt;none&gt;</td>
<td>443/TCP</td>
<td>2d</td>
<td>&lt;none&gt;</td>
</tr>
</tbody>
</table>
```

Access the service via the external IP address

root@jump host  k8s]# curl http://10.60.17.100
<!DOCTYPE html>
<html>
<head>
  <title>CiscoLive</title>
</head>
<body>
  <h1>Welcome to CiscoLive - Server 1 - port 80</h1>
</body>
</html>

root@jump host  k8s]#
iptables diff

```
[root@compute-1 k8s]# diff iptables-svc-interal-only iptables-svc-external-ip | grep 10.60.17.100
> -A KUBE-SERVICES -d 10.60.17.100/32 -p tcp -m comment --comment "default/cl-svc-1: external IP" -m tcp --dport 80 -j KUBE-MARK-MASQ
> -A KUBE-SERVICES -d 10.60.17.100/32 -p tcp -m comment --comment "default/cl-svc-1: external IP" -m tcp --dport 80 -m physdev !--physdev-is-in -m addrtype !--src-type LOCAL -j KUBE-SVC-FAQJORC6GMWDY3EK
> -A KUBE-SERVICES -d 10.60.17.100/32 -p tcp -m comment --comment "default/cl-svc-1: external IP" -m tcp --dport 80 -m addrtype --dst-type LOCAL -j KUBE-SVC-FAQJORC6GMWDY3EK
```
Our tiny setup

- cl-deploy-1: 10.224.1.7
- tools: 10.224.2.7
- cl-deploy-1: 10.224.2.8
- Compute-0
- Compute-1
- Controller-0

Network Addresses:
- 10.60.17.254
- 10.60.17.140
- 10.60.17.132
- 10.60.17.154
- 10.224.1.7
- 10.224.2.7
- 10.224.2.8
- 10.55.117.9

IP Addresses:
- 10.60.17.254
- 10.60.17.140
- 10.60.17.132
- 10.60.17.154
- 10.55.117.9
Simple external access to our service: NodePort

- NodePort exposes the service on each Node’s IP at a static port (the NodePort).
- A ClusterIP service, to which the NodePort service will route, is automatically created.
Edit the service to switch to NodePort:

```
kubectl edit service cl-svc-1
```

```
# Please edit the object below. Lines beginning with a:'#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this file will be
# reopened with the relevant failures.
#
apiVersion: v1
kind: Service
metadata:
  name: cl-svc-1
  namespace: default
  resourceVersion: "242623"
  selfLink: /api/v1/namespaces/default/services/cl-svc-1
  uid: 52e4cde2-d134-11e7-b632-0025b500014a
spec:
  clusterIP: 10.111.141.65
  externalTrafficPolicy: Cluster
  ports:
    - nodePort: 32358
      port: 80
      protocol: TCP
      targetPort: web-port
  selector:
    app: ciscoliveweb
    version: v2
    sessionAffinity: None
  type: NodePort
status:
  loadBalancer: {}
```
kubectl get service -o wide

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CLUSTER-IP</th>
<th>EXTERNAL-IP</th>
<th>PORT(S)</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>cl-svc-1</td>
<td>NodePort</td>
<td>10.111.141.65</td>
<td>&lt;none&gt;</td>
<td>80:32358/TCP</td>
<td>1d</td>
</tr>
<tr>
<td>kubernetes</td>
<td>ClusterIP</td>
<td>10.96.0.1</td>
<td>&lt;none&gt;</td>
<td>443/TCP</td>
<td>2d</td>
</tr>
</tbody>
</table>
Service can be reached at NodePort on any node

```
root@jumphost k8s]# curl http://compute-0:32358
<!DOCTYPE html>
<html>
<head>
  <title>CiscoLive</title>
</head>
<body>
  <h1>Welcome to CiscoLive - Server 1 - port 80</h1>
</body>
</html>
```

```
root@jumphost k8s]# curl http://compute-1:32358
<!DOCTYPE html>
<html>
<head>
  <title>CiscoLive</title>
</head>
<body>
  <h1>Welcome to CiscoLive - Server 1 - port 80</h1>
</body>
</html>
```

```
root@jumphost k8s]# curl http://controller-0:32358
<!DOCTYPE html>
<html>
<head>
  <title>CiscoLive</title>
</head>
<body>
  <h1>Welcome to CiscoLive - Server 1 - port 80</h1>
</body>
</html>
```
Edit the service to switch to a specific NodePort:

```
kubectl edit service svc-test-1
```

---

```
---
apiVersion: v1
kind: Service
metadata:
  name: cl-svc-1
  namespace: default
  resourceVersion: "243657"
  selfLink: /api/v1/namespaces/default/services/cl-svc-1
  uid: 52e4cde2-d134-11e7-b632-0025b500014a
spec:
  clusterIP: 10.111.141.65
  externalTrafficPolicy: Cluster
  ports:
  - name: svc-test-1
    nodePort: 30000
    port: 80
    protocol: TCP
    targetPort: web-port
  selector:
    app: ciscolivesweb
    version: v2
  sessionAffinity: None
  type: NodePort
status:
  loadBalancer: {}
```
```bash
[kroot@controller-0 k8s]# kubectl get services -o wide

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CLUSTER-IP</th>
<th>EXTERNAL-IP</th>
<th>PORT(S)</th>
<th>AGE</th>
<th>SELECTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>cl-svc-1</td>
<td>NodePort</td>
<td>10.111.141.65</td>
<td>&lt;none&gt;</td>
<td>80:30000/TCP</td>
<td>1d</td>
<td>app=ciscoliveweb,version=v2</td>
</tr>
<tr>
<td>kubernetes</td>
<td>ClusterIP</td>
<td>10.96.0.1</td>
<td>&lt;none&gt;</td>
<td>443/TCP</td>
<td>2d</td>
<td>&lt;none&gt;</td>
</tr>
</tbody>
</table>
```
Service can be reached at NodePort 30000

```
root@jumphost k8s]#curl http://compute-1:30000
<!DOCTYPE html>
<html>
  <head>
    <title>CiscoLive</title>
  </head>
  <body>
    <h1>Welcome to CiscoLive - Server 1 - port 80</h1>
  </body>
</html>
```
Container Network Interface (CNI)
CNI in the CNCF Reference Architecture

• Resource Management
  • Image Management
  • Container Management
  • Compute Resources

• Cloud Native – Network
  • Network Segmentation and Policy
  • SDN & APIs (eg CNI, libnetwork)

• Cloud Native- Storage
  • Volume Drivers/Plugins
  • Local Storage Management
  • Remote Storage Access
What is CNI?

- The simplest possible interface between container runtime and network implementation
- Originated at CoreOS as part of Rkt
- Now a CNCF project
- Gives driver freedom to manipulate network namespace
- Network described by JSON config
- Plugins support two commands:
  - Add Container to Network
  - Remove Container from Network
What is CNI?

- Provides Container Create/Delete events
- Need to provide information of network namespace, container interface name to the driver to plumb networking
- Networking and IPAM (both executables) run using the network configuration file
- Used by Kubernetes, Cloud Foundry, Weave, Calico, Contiv, Flannel,…
# What does a CNI call look like?

- Set some environment variables and go!

```bash
CNI_COMMAND=ADD \ 
CNI_CONTAINERID=$id \ 
CNI_NETNS=/proc/$pid/ns/net \ 
CNI_PATH=/opt/cni/bin \ 
CNI_IFNAME=eth0 \ 
  my-plugin < my-config
```

Can be either **ADD**, **DEL**, or **VERSION**

A JSON document defining the network
What does a CNI config look like?

```json
{
    "cniVersion": "0.2.0",
    "name": "ciscolivenet",
    "type": "bridge",
    "bridge": "cni1",
    "isGateway": true,
    "ipMasq": true,
    "ipam": {
        "type": "host-local",
        "subnet": "10.22.0.0/16",
        "routes": [
            { "dst": "0.0.0.0/0" }
        ]
    }
}
```

Tells the plugin what version the caller is using

The caller should look for a plugin with this name

First plugin will call a second plugin with this name
Network plugin calls IPAM plugin
CNI project repo
https://github.com/containernetworking/cni

• CNI Specification: the API between runtimes and network plugins

• Conventions: extensions to the API that are not required for all plugins

• Library: a Go implementation of the CNI specification that plugins and runtimes can use
CNI plugins repo
https://github.com/containernetworking/plugins
Using CNI – Try the example on https://github.com/containernetworking/cni

How do I use CNI?

Requirements
The CNI spec is language agnostic. To use the Go language libraries in this repository, you’ll need a recent version of Go. Our automated tests cover Go versions 1.7 and 1.8.

Reference Plugins
The CNI project maintains a set of reference plugins that implement the CNI specification. NOTE: the reference plugins used to live in this repository but have been split out into a separate repository as of May 2017.

Running the plugins
After building and installing the reference plugins, you can use the `priv-net-run.sh` and `docker-run.sh` scripts in the `scripts/` directory to exercise the plugins.

*note - priv-net-run.sh depends on jq*

Start out by creating a netconf file to describe a network:

```
$ mkdir -p /etc/cni/net.d
$ cat /etc/cni/net.d/10-my-net.conf <<EOF
{
  "cniVersion": "0.2.0",
```
Example
/etc/cni/10-ciscolivenet.conf

```
{
    "cniVersion": "0.2.0",
    "name": "ciscolivenet",
    "type": "bridge",
    "bridge": "cni1",
    "isGateway": true,
    "ipMasq": true,
    "ipam": {
        "type": "host-local",
        "subnet": "10.22.0.0/16",
        "routes": [
            { "dst": "0.0.0.0/0" }
        ]
    }
}
```
Running ifconfig..

```
[root@controller-0 scripts]# sudo CNI_PATH=$CNI_PATH ./priv-net-run.sh ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.22.0.3 netmask 255.255.0.0 broadcast 0.0.0.0
    inet6 fe80::6ca4:7fff:fe45:1cf0 prefixlen 64 scopeid 0x20<link>
    ether 6e:a4:7f:45:1c:f0 txqueuelen 0 (Ethernet)
    RX packets 9 bytes 786 (786.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 7 bytes 550 (550.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1450
    inet 10.244.0.3 netmask 255.255.255.0 broadcast 0.0.0.0
    inet6 fe80::c418:6fff:fe3c:c020 prefixlen 64 scopeid 0x20<link>
    ether c6:18:6f:3c:c0:20 txqueuelen 0 (Ethernet)
    RX packets 5 bytes 438 (438.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 6 bytes 460 (460.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1 (Local Loopback)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```
ifconfig cni1

[root@controller-0 scripts]# ifconfig cni1

    cni1: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
            inet 10.22.0.1  netmask 255.255.0.0  broadcast 0.0.0.0
            inet6 fe80::5c09:98ff:fef6:601d  prefixlen 64  scopeid 0x20<link>
            ether 5e:09:98:f6:60:1d  txqueuelen 1000  (Ethernet)
        RX packets 7  bytes 452 (452.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 5  bytes 438 (438.0 B)
        TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0
# Kubernetes Networking – Key Communication patterns

4 distinct communication patterns:

<table>
<thead>
<tr>
<th>Communication Pattern</th>
<th>Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highly-coupled container-to-container communications</td>
<td>Pod concept and localhost communications</td>
</tr>
<tr>
<td>Pod-to-Pod communications</td>
<td>Network Plugins</td>
</tr>
<tr>
<td>Pod-to-Service communications</td>
<td>Kubernetes services concept</td>
</tr>
<tr>
<td>External-to-Service communications</td>
<td>Kubernetes services concept</td>
</tr>
</tbody>
</table>

Diagram:

```
Kubernetes, Rocket…
Container Network Interface
Plugins
```
Kubernetes Network Plugins

- Kubernetes fundamental requirements for any networking implementation
  - all containers can communicate with all other containers without NAT
  - all nodes can communicate with all containers (and vice-versa) without NAT
  - the IP that a container sees itself as is the same IP that others see it as

- Network Plugins provide various levels of sophistication – from simple to highly feature rich and performant

Examples
- Cilium
- Contiv
- Contrail
- Flannel
- Google Compute Engine (GCE)
- Kube-router
- L2 networks and linux bridging
- Multus
- NSX-T
- Nuage Networks VCS
- OpenVSwitch
- OVN (Open Virtual Networking)
- Project Calico
- Romana
- Weave Net from Weaveworks
- CNI-Genie from Huawei
How do you make Kubernetes Networking fast, flexible, and scalable?
Container Networking Control: Contiv-VPP

- Contiv is a networking plugin for Kubernetes that:
  - Allocates IP addresses to Pods (IPAM)
  - Programs the underlying infrastructure it uses (Linux TCP/IP stack, OVS, VPP, …) to connect the Pod’s to other Pods in the cluster and/or to the external world.
  - Implements K8s network policies that define which pods can talk to each other.
  - Implements K8s services; a service exposes one or more (physical) service instances implemented as K8s pods to the other pods in the cluster and/or to external clients as a virtual instance (e.g. as a virtual “service” IP address).

- Contiv is a user-space based, high-performance, high-density networking plugin for Kubernetes. Contiv-VPP leverages FD.io/VPP as the industry’s highest performance data plane
Container Network Plugin **Contiv-VPP**

https://github.com/contiv/vpp
The Universal Dataplane - FD.io Vector Packet Processor (VPP)

- VPP is a rapid packet processing development platform for highly performing network applications
  - 18+ MPPS, single core
  - Multimillion entry FIBs
  - 480Gbps bi-dir on 24 cores
- Runs on commodity CPUs and leverages DPDK
- Creates a vector of packet indices and processes them using a directed graph of nodes – resulting in a highly performant solution.
- Runs as a Linux user-space application
- Ships as part of both embedded & server products, in volume; Active development since 2002
- FD.IO (The Fast Data Project)
VPP Universal Fast Dataplane: Performance at Scale
Per CPU core throughput with linear multi-thread(-core) scaling

IPv4 Routing

IPv6 Routing

Toplogy: Phy-VS-Phy

Hardware:
Cisco UCS C240 M4
Intel® C610 series chipset
2 x Intel® Xeon® Processor E5-2698 v3 (16 cores, 2.3GHz, 40MB Cache)
2133 MHz; 256 GB Total
6 x 2p40GE Intel XL710=12x40GE

Software:
Linux: Ubuntu 16.04.1 LTS
Kernel: 4.4.0-45-generic
FD.io VPP: VPP v17.01.5-gen234726 (DPDK 16.11)

Resources:
1 physical CPU core per 40GE port
Other CPU cores available for other services and other work
20 physical CPU cores available in 12x40GE setup
Lots of Headroom for much more throughput and features
Let’s evolve towards a full solution stack for all types of workloads, including NFV…
Cloud Native Container Networking
What Container Network Stacks Provide Today:

- Lifecycle management for application Containers
- Overlay connectivity for application Containers
  - NAT communication with external world
  - Policy controlled overlay, may extend policy control to DC fabric in some cases
  - Network policy addresses security/connectivity
  - Designed for Data Center applications / use cases

Good start, but not sufficient for NFV use cases (Firewalls, Customs VNFs)
What Container Networks Stacks Lack for NFV Use Cases:

- **NVF-specific policy APIs** (e.g. QoS, placement considering network resources, etc.)

- **Networking:**
  - NAT is a bottleneck, not suitable for NFV use cases
  - VNFs often require more than 1 interface / IP address
  - No support for high-speed wiring of NFVs:
    - To the outside world; To application containers;
    - Between NFV containers; Creation of Service Function Chains (mixed physical & virtual)

- **Management/Control:**
  - Containerized NVFs not really in the data plane (except for the vswitch)
  - No support for cloud-native, high-performance NVFs
Cloud-Native Network Function Summary

• Container-based
• Container stack lifecycle (same as application containers)
• 12-factor app design for management/control
• High-performance forwarding
• High-performance networking
• Seamless virtual/physical world interworking
• Common policy control with application containers (as much as possible)
• Must solve 3 key problems:
  • Lifecycle management
  • High-Performance Networking (wiring/config, operations)
  • Easy installation, operation
Ligato – Accelerate Cloud-Native Development
Platform and code samples for development of cloud native VNFs

Components:

• Cloud-Native Infrastructure – a Golang platform for building cloud-native microservices

• VPP-agent - Golang implementation of a control/management plane for FD.io/VPP based cloud-native Virtual Network Functions (VNFs)

• SFC-Controller - Service Function Chain (SFC) Controller for stitching virtual and physical networks

• BGP-Agent - BGP Agent is a BGP information provider

https://ligato.github.io/
Putting it all together
Enabling Production-Grade Native Cloud Network Services at Scale

Production-Grade Container Orchestration

Performance-Centric Container Networking

Cloud-native NF Orchestration
Cloud-native NF Agent platform

Containerized Fast Data Input/ Output
Cloud-native Network Micro-Services
Putting It All Together Now – The System Design

Functional Layered Diagram

System Components

Control and Management Plane
- Applications
- Contiv-VPP Netmaster
- Tools (e.g. agentctl)

Inter-Process Communication
- Data Store
- Message Bus

Containerized Network Data Plane
- Container Networking
- Networking Plugin
- Kubelet

Production-Grade Container Orchestration
- Kubernetes
- API Proxies

Network Function and Network Topology Orchestration
- SFC Controller
- Container Networking
- Contiv-VPP Netmaster

Containerized Switch
- Containerized Network Functions
- Container Networking

Containers Lifecycle Orchestration
- Kubernetes
- CNI
- CRI

Network Function and Network Topology Orchestration
- Service Policy
- Service Topology
- Lifecycle

Putting It All Together Now – The System Design
Network Micro-Service Use Case: Service Function Chaining with Cloud-Native NFs
A glimpse at performance: Example topology

Topology: Containerized Switch with one or more Containerized Network Functions
Example Benchmark: memif 10x faster than veth
Mesh Networking: Istio
Connect, manage, secure microservices
Micro-Service Architecture Side-Effects

• Replace a reliable in-process call with an unreliable RPC.
• Trivial single stepping replaced by … ?
• Secure in-process communication is replaced by a potentially insecure network.
• Access control within process was formerly a NOOP
• Increased latency for inter-process communication
• Tracing? System-level gdb?
Towards the Service Mesh: Additions to the basic Micro-Service Architecture

- Add retry logic to the application code.
- Add entry-exit traces.
- Secure inter service connections with strong authentication.
- Now that we are adding code … choose the RPC endpoint intelligently
  - Endpoints with low latency.
  - Endpoints with warm caches.

Create this logic once for all applications
(rather than re-write things in N different languages)
Service Mesh:
What to add to the mix to make microservices successful

• Visibility
• Resiliency & Efficiency
• Traffic Control
• Security
• Policy Enforcement
Vanilla Kubernetes
Application

Traffic Control
Policy Enforcement
Visibility
Security
Resiliency & Efficiency
Introducing a sidecar – Istio/Envoy
Proxy (sidecar) details

- Deployed as a sidecar per Pod
- Proxy acts as a client and server

Envoy Edge Proxy

HTTP/1.1, HTTP/2, TLS

Service A Cluster
- Service A
- Envoy

Service B Cluster
- Service B
- Envoy

External Network
Service Mesh Overview

- **Outbound features**
  - Service authentication
  - Load balancing
  - Retry and circuit breaker
  - Fine-grained routing
  - Telemetry
  - Request Tracing
  - Fault Injection

- **Inbound features:**
  - Service authentication
  - Authorization
  - Rate limits
  - Load shedding
  - Telemetry
  - Request Tracing
  - Fault Injection
Architecture

- **Pilot**: Configures Istio deployments and propagate configuration to the other components of the system. Routing and resiliency rules go here.

- **Mixer**: Responsible for policy decisions and aggregating telemetry data from the other components in the system using a flexible plugin architecture.

- **Proxy**: Based on Envoy, mediates inbound and outbound traffic for all Istio-managed services. It enforces access control and usage policies, and provides rich routing, load balancing, and protocol conversion.
Using Istio/Envoy
Example: Identity based request routing

The Comedy of Errors

Summary: Wikipedia Summary: The Comedy of Errors is one of William Shakespeare’s early plays. It is his shortest and one of his most farcical comedies, with a major part of the humour coming from slapstick and mistaken identity, in addition to puns and word play.

<table>
<thead>
<tr>
<th>Book Details</th>
<th>Book Reviews</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type: paperback</td>
<td></td>
</tr>
<tr>
<td>Pages: 200</td>
<td></td>
</tr>
<tr>
<td>Publisher: PublishersA</td>
<td></td>
</tr>
<tr>
<td>Language: English</td>
<td></td>
</tr>
<tr>
<td>ISBN-10: 1234567890</td>
<td></td>
</tr>
</tbody>
</table>

An extremely entertaining play by Shakespeare. The slapstick humour is refreshing!
— Reviewer1

Absolutely fun and entertaining. The play lacks thematic depth when compared to other plays by Shakespeare.
— Reviewer2
Example: Identity based request routing (2/3)
Example: Identity based request routing (3/3)

The Comedy of Errors

Summary: Wikipedia Summary: The Comedy of Errors is one of William Shakespeare's early plays. It is his shortest and one of his most farcical comedies, with a major part of the humour coming from slapstick and mistaken identity, in addition to puns and word play.

<table>
<thead>
<tr>
<th>Book Details</th>
<th>Book Reviews</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:</td>
<td>An extremely entertaining play by Shakespeare. The slapstick humour is refreshing!</td>
</tr>
<tr>
<td>paperback</td>
<td>— Reviewer1</td>
</tr>
<tr>
<td>Pages: 200</td>
<td>★★★★★</td>
</tr>
<tr>
<td>Publisher: PublishersA</td>
<td></td>
</tr>
<tr>
<td>Language:</td>
<td>Absolutely fun and entertaining. The play lacks thematic depth when compared to other plays by Shakespeare.</td>
</tr>
<tr>
<td>English</td>
<td>— Reviewer2</td>
</tr>
<tr>
<td>ISBN-10:</td>
<td>★★★★☆</td>
</tr>
<tr>
<td>1234567890</td>
<td></td>
</tr>
<tr>
<td>ISBN-13:</td>
<td></td>
</tr>
<tr>
<td>123-1234567890</td>
<td></td>
</tr>
</tbody>
</table>
Summary
Summary

Production-Grade Container Workload Scheduling and Orchestration

Layer-7 overlay network with granular control using a “sidecar” (L7 proxy)

Performance-Centric Container Networking

Cloud-native NF Orchestration

Cloud-native NF Agent platform

Data-Plane: Containerized Fast Data Input/ Output
Many thanks to

- Jon Langemak
- Animesh Singh
- Daneyon Hansen
- Jan Medved
- Maciek Konstantynowicz
- Mandar Jog
- Ken Owens
Cisco Spark

Questions?
Use Cisco Spark to communicate with the speaker after the session

How
1. Find this session in the Cisco Live Mobile App
2. Click “Join the Discussion”
3. Install Spark or go directly to the space
4. Enter messages/questions in the space

cs.co/ciscolivebot#BRKDCN-2390
• Please complete your Online Session Evaluations after each session

• Complete 4 Session Evaluations & the Overall Conference Evaluation (available from Thursday) to receive your Cisco Live T-shirt

• All surveys can be completed via the Cisco Live Mobile App or the Communication Stations

Don’t forget: Cisco Live sessions will be available for viewing on-demand after the event at www.ciscolive.com/global/on-demand-library/. Complete Your Online Session Evaluation
Continue Your Education

• Demos in the Cisco campus
• Walk-in Self-Paced Labs
• Tech Circle
• Meet the Engineer 1:1 meetings
• Related sessions
Thank you