Kubernetes Networking Made Easy with Open vSwitch and OpenFlow

Péter Megyesi
Co-founder @ LeanNet ltd.
Who Am I?

PhD in Telecommunications @ Budapest University of Technology
- Measurement and monitoring in Software Defined Networks
- Participating in 5G-PPP EU projects
- Graduated in the EIT Digital Doctoral School

Co-founder & CTO @ LeanNet Ltd.
- Evangelist of open networking solutions
- Currently focusing on SDN in cloud native environments

megyesi@leannet.eu
twitter.com/M3gy0
linkedin.com/in/M3gy0
What is Open vSwitch?

The de facto production quality, multilayer virtual switch
- Originally developed by Nicira (the inventors of SDN and OpenFlow)
- Now it’s developed under the Linux Foundation
- Designed to be programmable by OVSDB and OpenFlow
- Compatible with standard management interfaces (NetFlow, sFlow, IPFIX, RSPAN, LACP)
- The basis of VMware NSX-T, OpenStack and many other public clouds...
- Able to run in user-space mode via DPDK, thus can provide speed up to ~80 Gbps
What is Kubernetes?

The de facto production quality, container-orchestration framework

- Originally developed by Google (Borg project)
- Now maintained by the Cloud Native Compute Foundation
- Automating deployment, scaling, and management of containerized applications
### Basic Kubernetes Terminology

<table>
<thead>
<tr>
<th>Kubernetes Master</th>
<th>Replication Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>▪ Controller of a Kubernetes cluster</td>
<td>▪ Ensures availability and scalability</td>
</tr>
<tr>
<td>Kubernetes Node (Worker / Minion)</td>
<td>Services</td>
</tr>
<tr>
<td>▪ Hosts (server or VM) that run Kubernetes</td>
<td>▪ Collection of pods exposed as an</td>
</tr>
<tr>
<td>applications</td>
<td>endpoint</td>
</tr>
<tr>
<td>Container</td>
<td>Node Port</td>
</tr>
<tr>
<td>▪ Unit of packaging</td>
<td>▪ Expose services internally</td>
</tr>
<tr>
<td>Pod</td>
<td>Load Balancer</td>
</tr>
<tr>
<td>▪ Unit of deployment</td>
<td>▪ The way for external access</td>
</tr>
</tbody>
</table>

Labels and Selectors

▪ Key-Value pairs for identification
The Docker Model

Docker Host

- eth0
- vethxx
- vethxy

Root namespace

- docker0 172.17.0.1/24

Container 1

www.leanet.eu

Péter Megyesi: Kubernetes Networking Made Easy with Open vSwitch and OpenFlow
The Docker Model

Docker Host

- eth0
- docker0: 172.17.0.1/24
- vethxx
- eth0
- Container 1: 172.17.0.2

Root namespace
The Docker Model

Docker Host

docker0  172.17.0.1/24

vethxx  vethyy

eth0

172.17.0.2

Container 1

eth0

172.16.0.3

Container 2

Root namespace

The Docker Model
Docker Host Ports

Host 1: 10.0.0.10

- 172.17.0.2
- 172.17.0.3

- 80
- 5001

- SNAT

Host 2: 10.0.0.20

- 172.17.0.2
- 9898

- 26432

This is unfeasible in a very large cluster!
Networking in Kubernetes

Pod-to-Pod communication
- Each Pod in a Kubernetes cluster is assigned an IP in a flat shared networking namespace
- All PODs can communicate with all other PODs without NAT
- The IP that a PODs sees itself as is the same IP that others see it as

Pod-to-Service communication
- Requests to the Service IPs are intercepted by a Kube-proxy process running on all hosts
- Kube-proxy is then responsible for routing to the correct POD

External-to-Internal communication
- All nodes can communicate with all PODs (and vice-versa) without NAT
- Node ports are can be assigned to a service on every Kuberentes host
- Public IPs can be implemented by configuring external Load Balancers which target all nodes in the cluster
- Once traffic arrives at a node, it is routed to the correct Service backends by Kube-proxy
The Container Network Interface
CNI in Kubernetes

Script / binary placed on every host
- Kubelet calls it with the right environmental variables and STDIN parameters

Example for configuration

```
- /etc/cni/net.d/01-dunlin.conf

```

```yaml
1 2 3 4 5 6 7 8 9 10 11 12
{ "cniVersion": "0.2.0",
  "name": "dunlin",
  "type": "ovs_cni",
  "bridge": "br0",
  "isGateway": true,
  "ipam": {
    "type": "host-local",
    "subnet": "10.244.1.0/24",
    "gateway": "10.244.1.1"
  }
}
```
CNI in Kubernetes

Script / binary placed on every host
- Kubelet calls it with the right environmental variables and STDIN parameters

Example environment variables
- CNI_command: add or delete
- CNI_netns: /proc/<PID>/ns/net
- CNI_ifname: eth0
- CNI_path: /opt/bin/cni
- CNI_containerid
- K8S_pod_name
- K8S_pod_namespace
Create virtual ethernet port pair

- `ip link add veth0 type veth peer name veth1`
CNI With Open vSwitch

Create virtual ethernet port pair

- `ip link add veth0 type veth peer name veth1`

Add interface to OVS bridge

- `ovs-vsctl add-port br0 veth0`
CNI With Open vSwitch

Create virtual ethernet port pair
  ▪ ip link add *veth0* type veth peer name *veth1*

Add interface to OVS bridge
  ▪ ovs-vsctl add-port *br0 veth0*

Add the other interface to namespace
  ▪ ip set link *veth1* netns *$CNI_netns*
CNI With Open vSwitch

Create virtual ethernet port pair
- `ip link add veth0 type veth peer name veth1`

Add interface to OVS bridge
- `ovs-vsctl add-port br0 veth0`

Add the other interface to namespace
- `ip set link veth1 netns $CNI_netns`

Rename and setup interface
- `ip netns exec $CNI_netns`
  - `ip link set dev veth1 name eth0`
  - `ip link set dev eth0 address 10.244.1.2`
  - `ip link set dev eth0 mtu 1450`
  - `ip route add default via 10.244.1.1`
CNI With Open vSwitch

Create virtual ethernet port pair
- `ip link add veth0 type veth peer name veth1`

Add interface to OVS bridge
- `ovs-vsctl add-port br0 veth0`

Add the other interface to namespace
- `ip set link veth1 netns $CNI_netns`

Rename and setup interface
- `ip netns exec $CNI_netns`
  - `ip link set dev veth1 name eth0`
  - `ip link set dev eth0 address 10.244.1.2`
  - `ip link set dev eth0 mtu 1450`
  - `ip route add default via 10.244.1.1`
The Kubernetes Model – The IP per POD Model

Kubernetes Node 1
10.244.1.0/24

- 10.244.1.2
  - POD

- br0

- 10.244.1.3
  - POD

Kubernetes Node 2
10.244.2.0/24

- 10.244.2.2
  - POD

- Host 2: 10.0.0.20

Kubernetes Node 3
10.244.3.0/24

- 10.244.3.2
  - POD

- Host 3: 10.0.0.30

Host 1: 10.0.0.10

The diagram illustrates the Kubernetes networking model where each Kubernetes node has a unique IP range, and each POD within a node has a dedicated IP address. The hosts linked to the Kubernetes nodes also have their own IP addresses.
Cluster Networking in Kubernetes
Life of a Packet: POD-to-POD, Same Node

L2 src: pod1
L2 dst: pod2

L3 src: pod1
L3 dst: pod2
Life of a Packet: POD-to-POD, Same Node

Linux Bridge
- MAC learning

Open vSwitch
- MAC learning: `action=normal`
- L2 rule: `dl_dst=pod2`, `action=output:2`
- L3 rule: `ip,nw_dst=pod2`, `action=output:2`
Life of a Packet: POD-to-POD, Same Node

Linux Bridge
- MAC learning

Open vSwitch
- MAC learning: action=normal
- L2 rule: dl_dst=pod2, action=output:2
- L3 rule: ip,nw_dst=pod2, action=output:2

L2 src: pod1
L2 dst: pod2

L3 src: pod1
L3 dst: pod2
Life of a Packet: POD-to-POD, Between Nodes
Life of a Packet: POD-to-POD, Between Nodes

L2 src: pod1
L2 dst: br0 (gw)
L3 src: pod1
L3 dst: pod4
Using an overlay network
- An overlay network obscures the underlying network architecture from the pod network through traffic encapsulation (for example VxLAN, GRE)
- Encapsulation reduces performance, though exactly how much depends on your solution

Where to go from here??
Life of a Packet: POD-to-POD, Between Nodes

Without an overlay network
- Configure the underlying network fabric (switches, routers, etc.) to be aware of pod IP addresses
- This does not require the encapsulation provided by an overlay, and so can achieve better performance

Network Fabric

Where to go from here??

L2 src: pod1
L2 dst: br0 (gw)
L3 src: pod1
L3 dst: pod4

Kubernetes Node 1

Kubernetes Node 4
Kubernetes Cluster Networking Plugins

Public clouds which supports Kubernetes program this into the fabric

- E.g. in Google Container Engine: “everything to 10.1.1.0/24, send to this VM”

In other cases we need to use an external plugin

- Flannel
- Calico
- Canal
- Romana
- Weave
- Cisco Contiv
- Huawei CNI-Genie
- Nuage Networks VCS (by Nokia)
- Open Virtual Network
Life of a Packet: POD-to-POD, Between Nodes

L2 src: pod1
L2 dst: br0 (gw)
L3 src: pod1
L3 dst: pod4

Calico defines BGP agents and advertises the POD subnets to the fabric. It uses IP-IP encapsulation.
Life of a Packet: POD-to-POD, Between Nodes

L2 src: pod1
L2 dst: br0 (gw)
L3 src: pod1
L3 dst: pod4

Flannel and Weave creates VxLAN tunnels between nodes using a kernel implementation.
Life of a Packet: POD-to-POD, Between Nodes

L2 src: pod1
L2 dst: br0 (gw)
L3 src: pod1
L3 dst: pod4

Network Fabric

Set-up VxLAN ports to every other node
- ovs-vsctl add-port br0 vxlan4 -- set interface
  vxlan4 type=vxlan option:remote_ip={node4_ip}

Add rule for their subnet
- ip,nw_dst={node4_subnet},
  action=output:vxlan4
**Life of a Packet: POD-to-POD, Between Nodes**

**Network Fabric**

- **L2 src:** pod1
- **L2 dst:** br0 (gw)
- **L3 src:** pod1
- **L3 dst:** pod4

**Set-up VxLAN one port**
- `ovs-vsctl add-port br0 vxlan0` – set interface
  `vxlan0 type=vxlan option:key=flow`  
  `option:remote_ip=flow`

**Add rule including tunnel destination**
- `ip,nw_dest= {node4_subnet},actions=
  set_field:{node4_ip}->tun_dst,output:vxlan0`

---

**Kubernetes Node 1**

- `eth0`
- `br0`
- `vethxx`  
- `vethyy`  
- `eth0`
- `pod 1`
- `eth0`
- `pod 2`

**Kubernetes Node 4**

- `eth0`
- `br0`
- `vethzz`  
- `vethvv`
- `eth0`
- `pod 3`
- `eth0`
- `pod 4`
Life of a Packet: POD-to-POD, Between Nodes

Set-up VxLAN one port
- `ovs-vsctl add-port br0 vxlan0` – set interface
  `vxlan0 type=vxlan option:key=flow
  option:remote_ip=flow`

Add rule including tunnel destination
- `ip,nw_dest= {node4_subnet},actions= set_field:{node4_ip}->tun_dst,output:vxlan0`
Life of a Packet: POD-to-POD, Between Nodes

L2 src: br0 (node4)
L2 dst: pod4
L3 src: pod1
L3 dst: pod4

Set-up VxLAN one port
- `ovs-vsctl add-port br0 vxlan0` – set interface vxlan0 type=vxlan option:key=flow option:remote_ip=flow

Add rule including tunnel destination
- `ip,nw_dest= {node4_subnet},actions= set_field:{node4_ip}->tun_dst,output:vxlan0`
For This, You Will Need a Control Plane

Kubernetes API ➔ State information ➔ Control Plane Software

Network Fabric

Rule installation via OpenFlow and OVSDB

Kubernetes Node 1

- eth0
- br0
- vethxx
- vethyy
- eth0 (pod 1)
- eth0 (pod 2)

Kubernetes Node 4

- eth0
- br0
- vethzz
- vethvv
- eth0 (pod 3)
- eth0 (pod 4)
Pod to Service Communication in Kubernetes
Definition:
- Service is an abstraction to define a logical set of Pods bound by a policy by to access them
- Defined by labels and selectors
- Supports TCP and UDP
- Interfaces with Kube-Proxy to manipulate IP tables
- Service can be exposed internally by cluster/service IP

**Remember:**

*PODs are Mortal!!!*
Services in Kubernetes

```
kind: Service
apiVersion: v1
metadata:
  name: store-be
  namespace: default
  creationTimestamp: 2016-05-06T19:16:56Z
  resourceVersion: "7"
  selfLink:
    /api/v1/namespaces/default/services/store-be
  uid: 196d5751-13bf-11e6-9353-42010a800fe3
spec:
  type: ClusterIP
  selector:
    app: store
    role: be
  clusterIP: 10.9.3.76
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 80
      sessionAffinity: None
```

- **This will be the DNS name of the service**
- **This will be the IP of the service**
- **This will be the port of the service**
- **This is the POD port**
- **Selector for PODs**
Life of a Packet: POD-to-Service

L2 src: pod1
L2 dst: br0
L3 src: pod1
L3 dst: svc1

Kubernetes Node 1

Root namespace

eth0
br0
vethxx
vethyy
eth0
eth0
pod 1
pod 2
Life of a Packet: POD-to-Service

L2 src: pod1
L2 dst: br0
L3 src: pod1
L3 dst: svc1

Kubernetes Node 1

eth0
br0
vethxx
vethyy
eth0
eth0
pod 1
pod 2
Life of a Packet: POD-to-Service

L2 src: pod1
L2 dst: br0
L3 src: pod1
L3 dst: svc1
L3 src: pod1
L3 dst: svc1
L3 dst: pod88

DNAT, conntrack

Remember:
- Every node should reach every POD in the cluster
- `ip route add {global_pod_cidr} via br0`
  
  *e.g. 10.244.0.0/16*
Example for IPtables Ruleset

Chain KUBE-SERVICES (2 references)
target prot opt source destination
KUBE-MARK-MAPO tcp -- 10.244.0.0/16 10.110.89.105 /* sock-shop/front-end: cluster IP */ tcp dpt:http
KUBE-SVC-LPMD63S3E2A0US8 tcp -- anywhere 10.110.89.105 /* sock-shop/front-end: cluster IP */ tcp dpt:http
KUBE-MARK-MAPO tcp -- 10.244.0.0/16 10.97.201.132 /* sock-shop/orders-db: cluster IP */ tcp dpt:29017
KUBE-SVC-KTVGUVR3S4J4S5G2 tcp -- anywhere 10.97.201.132 /* sock-shop/orders-db: cluster IP */ tcp dpt:29017
KUBE-MARK-MAPO tcp -- 10.244.0.0/16 10.97.121.97 /* sock-shop/rabbitmq: cluster IP */ tcp dpt:amqp
KUBE-SVC-HF7SSIC3BQV7VZI5 tcp -- anywhere 10.97.121.97 /* sock-shop/rabbitmq: cluster IP */ tcp dpt:amqp

Chain KUBE-SVC-LPMD63S3E2A0US8 (2 references)
target prot opt source destination
KUBE-SEP-5SSOKL8CEKVOUEYD all -- anywhere anywhere /* sock-shop/front-end: */ statistic mode random probability 0.025000000000
KUBE-SEP-5VWB3SN3QRO1S95MQ all -- anywhere anywhere /* sock-shop/front-end: */ statistic mode random probability 0.33332999982
KUBE-SEP-KTE54G5F5LP4QY1 all -- anywhere anywhere /* sock-shop/front-end: */ statistic mode random probability 0.50000000000
KUBE-SEP-NXFT3BEH1E5FWGY3W all -- anywhere anywhere /* sock-shop/front-end: */

Chain KUBE-SEP-5SSOKL8CEKVOUEYD (1 references)
target prot opt source destination
KUBE-MARK-MAPO all -- 10.244.1.7 anywhere /* sock-shop/front-end: */
DNAT top -- anywhere anywhere /* sock-shop/front-end: */ tcp to:10.244.1.7:8079
Life of a Packet: POD-to-Service

L3 src: pod1
L3 dst: pod88
via tunnel network

Kubernetes Node 1

eth0
br0
vethxx
vethyy
eth0
pod 1
eth0
pod 2

IPtables
Root namespace

Network

Tunnel

L3 src: pod1
L3 dst: pod88
via tunnel network
Life of a Packet: POD-to-Service

L3 src: pod88
L3 dst: pod1

via tunnel
network

Kubernetes Node 1

eth0
IPtables
br0
vethxx
vethyy
eth0
pod 1
pod 2

Root namespace

via tunnel network

Pod-to-Service

Network

Kubernetes Node 1

eth0
IPtables
br0
vethxx
vethyy
eth0
pod 1
pod 2

Root namespace

via tunnel network

Pod-to-Service

Network
Life of a Packet: POD-to-Service

L3 src: pod88
L3 dst: pod1
via tunnel network

https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/
Life of a Packet: POD-to-Service

L3 src: pod88
L3 src: svc1
L3 dst: pod1
un-DNAT

https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/
Péter Megyesi: Kubernetes Networking Made Easy with Open vSwitch and OpenFlow

Life of a Packet: POD-to-Service

L3 src: svc1
L3 dst: pod1

https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/
Unfortunately, you can’t do the same with OVS

https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/
Handling Service Communication with OVS: Option 1

Use `ct*` flow rules:

- It uses the same `conntrack` kernel module as IPtables
- You can specify similar NAT rules than you would in IPtables
- For load balancing between POD backend, you can use group rules

```
table=0, ip, nw_src={pod_cidr}, nw_dst={service_cidr}, ct_state=-trk, action=ct(table=2)
table=0, ip, nw_src={pod_cidr}, nw_dst={pod_cidr}, ct_state=-trk, action=ct(table=4)

```table=2, ip, nw_dst={svc1_ip}, tp_dst={svc1_port}, ct_state=+trk+new, action=group:1
```table=2, ip, nw_dst={svc2_ip}, tp_dst={svc2_port}, ct_state=+trk+new, action=group:2
```table=2, ct_state=+trk-new, action=table:4
```

`table=4` contains the original switching / routing rules

- `group_id=1, type=select,`
  - `bucket=ct(commit, nat(dst={pod1_ip}:{pod_port})), table=4,`
  - `bucket=ct(commit, nat(dst={pod2_ip}:{pod_port})), table=4,`
  - `bucket=ct(commit, nat(dst={pod3_ip}:{pod_port})), table=4`

* `ct` rules are actually not OpenFlow compatible
Handling Service Communication with OVS: Option 2

Use *stateless* NAT rules:

- If we see a Service IP we switch the destination IP to a POD backend
- But at the same time we modify the source IP to a shifted domain (e.g. 10.244.x.y \(\rightarrow\) 172.24.x.y)
- This way we don’t use any kernel specific rules which allows the integration into user-space (e.g. DPDK)

```plaintext

<table>
<thead>
<tr>
<th>Table</th>
<th>Conditions</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>ip, nw_src={pod_cidr}, nw_dst={service_cidr}</code></td>
<td><code>action=table:2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>ip, nw_src={pod_cidr}, nw_dst={shifted_pod_cidr}</code></td>
<td><code>action=table:3</code></td>
</tr>
<tr>
<td>0</td>
<td><code>ip, nw_src={pod_cidr}, nw_dst={pod_cidr}</code></td>
<td><code>action=table:4</code></td>
</tr>
<tr>
<td>2</td>
<td><code>ip, nw_dst={svc1_ip}, tp_dst={svc1_port}</code></td>
<td><code>actions=load:44056-&gt;NXM_OF_IP_SRC[16..31], group:1</code></td>
</tr>
</tbody>
</table>
| 3       | `ip, nw_src={pod1_ip}, tp_src={pod_port}`                                  | `actions=mod_nw_src:{svc1_ip}, mod_tp_src:{svc1_port}
, load:2804->NXM_OF_IP_DST[16..31], resubmit:4` |
| 4       |                                                                             |                     |

Table 4 contains the original switching / routing rules

<table>
<thead>
<tr>
<th>Group</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>type=select</code>, <code>bucket=mod_nw_dst:{pod1_ip}, mod_tp_dst:{pod_port}, resubmit=4</code>, <code>bucket=mod_nw_dst:{pod2_ip}, mod_tp_dst:{pod_port}, resubmit=4</code>, <code>bucket=mod_nw_dst:{pod3_ip}, mod_tp_dst:{pod_port}, resubmit=4</code></td>
</tr>
</tbody>
</table>
```

*NXM* stands for Nicira eXtended Match rules which are also not OpenFlow compatible
Finally, it’s demo time 😊
Performance Comparison: Google Cloud

Intranode Performance @ Google Cloud

<table>
<thead>
<tr>
<th></th>
<th>Throughput</th>
<th>90 Percentile Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>40.3</td>
<td>30</td>
</tr>
<tr>
<td>Flannel</td>
<td>26.7</td>
<td>35</td>
</tr>
<tr>
<td>Calico</td>
<td>26.5</td>
<td>37</td>
</tr>
<tr>
<td>Weave</td>
<td>29.4</td>
<td>33.2</td>
</tr>
</tbody>
</table>

Internode Performance @ Google Cloud

<table>
<thead>
<tr>
<th></th>
<th>Throughput</th>
<th>90 Percentile Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>7.7</td>
<td>132.8</td>
</tr>
<tr>
<td>Flannel</td>
<td>5.1</td>
<td>166.5</td>
</tr>
<tr>
<td>Calico</td>
<td>4.9</td>
<td>175.3</td>
</tr>
<tr>
<td>Weave</td>
<td>2.7</td>
<td>184</td>
</tr>
<tr>
<td>OVS</td>
<td>3</td>
<td>159.8</td>
</tr>
</tbody>
</table>
Performance Comparison: Amazon Cloud

Intranode Performance @ Amazon Cloud

<table>
<thead>
<tr>
<th></th>
<th>Throughput</th>
<th>90 Percentile Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>43.2</td>
<td>29.7</td>
</tr>
<tr>
<td>Flannel</td>
<td>31.7</td>
<td>34</td>
</tr>
<tr>
<td>Calico</td>
<td>35.3</td>
<td>32.6</td>
</tr>
<tr>
<td>Weave</td>
<td>34.5</td>
<td>36.3</td>
</tr>
<tr>
<td>OVS</td>
<td>32.5</td>
<td>29.9</td>
</tr>
</tbody>
</table>

Internode Performance @ Amazon Cloud

<table>
<thead>
<tr>
<th></th>
<th>Throughput</th>
<th>90 Percentile Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>9.6</td>
<td>75.3</td>
</tr>
<tr>
<td>Flannel</td>
<td>7.9</td>
<td>98.3</td>
</tr>
<tr>
<td>Calico</td>
<td>89.5</td>
<td>91.3</td>
</tr>
<tr>
<td>Weave</td>
<td>6.1</td>
<td>3.6</td>
</tr>
<tr>
<td>OVS</td>
<td>93.0</td>
<td>7.9</td>
</tr>
</tbody>
</table>
Performance Comparison: Packet

Intranode Performance @ Packet.net

<table>
<thead>
<tr>
<th></th>
<th>Throughput</th>
<th>90 Percentil Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>32.5</td>
<td>13.8</td>
</tr>
<tr>
<td>Flannel</td>
<td>19.5</td>
<td>18.9</td>
</tr>
<tr>
<td>Calico</td>
<td>20.5</td>
<td>19</td>
</tr>
<tr>
<td>Weave</td>
<td>21.6</td>
<td>21.2</td>
</tr>
<tr>
<td>OVS</td>
<td>18.8</td>
<td>16</td>
</tr>
</tbody>
</table>

Internode Performance @ Packet.net

<table>
<thead>
<tr>
<th></th>
<th>Throughput</th>
<th>90 Percentil Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>8.2</td>
<td>30.8</td>
</tr>
<tr>
<td>Flannel</td>
<td>47.8</td>
<td>4.6</td>
</tr>
<tr>
<td>Calico</td>
<td>52.5</td>
<td>3.9</td>
</tr>
<tr>
<td>Weave</td>
<td>50.8</td>
<td>3.8</td>
</tr>
<tr>
<td>OVS</td>
<td>42.8</td>
<td>4.2</td>
</tr>
</tbody>
</table>
Kubernetes Networking with Open vSwitch

Pure OVS solution

- CNI binary attaching PODs to and OVS bridge
- POD-to-POD and POD-to-Service communication with OpenFlow rules
- Enhanced monitoring using Prometheus and OVS-exporter
- Speed and latency is comparable with leading plugins (Flannel, Calico, Weave)
- DPDK integration possibility
- 100% open source: https://github.com/dunlinplugin

![Dunlin](dunlin.io)
Backup Slides
Mean Latency

contrib/for-tests/netperf-tester --number=1000
Pod to External Communication in Kubernetes
Life of a packet: pod-to-external

src: pod1
dst: 8.8.8.8
Life of a packet: pod-to-external

- src: pod1
- dst: 8.8.8.8

POD IP address is private
- Needs NAT to communicate with external
Life of a packet: pod-to-external

- **src:** pod1
- **src:** NodeIP
- **dst:** 8.8.8.8

**MASQUERADE**

POD IP address is private
- Needs NAT to communicate with external

Node IPs are usually also private
Life of a packet: pod-to-external

**POD IP address is private**
- Needs NAT to communicate with external

**Node IPs are usually also private**
- Needs second NAT by the fabric

MASQUERADE
Life of a packet: pod-to-external

POD IP address is private
- Needs NAT to communicate with external

Node IPs are usually also private
- Needs second NAT by the fabric

src: PublicIP
dst: 8.8.8.8
The Hairpin Problem

src: pod1
dst: svc1
dst: pod1

DNAT, conntrack
The Hairpin Problem

src: pod1
dst: pod1

Kubernetes Node 1

eth0

vethxx

vethyy

eth0

pod 1

pod 2

IPtables

Root namespace

cbr0

src: pod1
dst: pod1

The Hairpin Problem

src: pod1
dst: pod1

The reply for this packet would not leave this POD at all!

Only SNAT at the in IPtables can solve this problem
The Hairpin Problem

```
src: pod1
dst: svc1

DNAT, conntrack
```

---

**weaveworks**

---

Kubernetes Node 1

---

**Root namespace**

---

**IPtables**

---

**eth0**

---

**cbr0**

---

**vethxx**

---

**vethyy**

---

**eth0**

---

**pod 1**

---

**pod 2**
External to Internal Communication in Kubernetes
External-to-Internal Traffic

Serveices can be exposed to the outside by

- Node port
- Load Balancer

Example: frontend

- pod 11
- pod 31
- pod 32
Serveices can be exposed to the outside by

- Node port
- Load Balancer

Example: frontend

- pod 11
- pod 31
- pod 32
External-to-Internal Traffic

Node port
- One port on every node gets rerouted to a certain service
- Typically port number > 30000
- ∀NodeIP:30001 → 10.9.8.15:8080
- Node IPs are usually not public!
Node port

- One port on every node gets rerouted to a certain service
- Typically port number > 30000
- ∀NodeIP:30001 → 10.9.8.15:8080
- Node IPs are usually not public!

Translates to one of the pod IP randomly:
External-to-Internal Traffic

Node port
- One port on every node gets rerouted to a certain service
- Typically port number > 30000
- ∀ NodeIP:30001 → 10.9.8.15:8080
- Node IPs are usually not public!

Network Fabric

1/3

Kubernetes Node 1
- eth0
- cbr0
- vethxx
- pod 11
- IPtables

Root ns

Kubernetes Node 2
- eth0
- cbr0
- IPtables

Root ns

Kubernetes Node 3
- eth0
- cbr0
- vethoo
- vethpp
- pod 31
- pod 32

1/3

src: xxx:yyy
dst: Node2:30001
src: Node2:xxxx
dst: pod11:8080

1/3

One port on every node gets rerouted to a certain service
- Typically port number > 30000
- ∀ NodeIP:30001 → 10.9.8.15:8080
- Node IPs are usually not public!
Node port

- One port on every node gets rerouted to a certain service
- Typically port number > 30000
- ∀NodeIP:30001 → 10.9.8.15:8080
- Node IPs are usually not public!

External-to-Internal Traffic

Network Fabric

src: xxx:yyy

dst: Node3:30001

Translates to one of the pod IP randomly

1/3 1/3 1/3

Node port

One port on every node gets rerouted to a certain service
Typically port number > 30000
∀NodeIP:30001 → 10.9.8.15:8080
Node IPs are usually not public!
Node port
- One port on every node gets rerouted to a certain service
- Typically port number > 30000
- ∀NodeIP:30001 → 10.9.8.15:8080
- Node IPs are usually not public!
External-to-Internal Traffic

Load Balancer
- One public IP that maps to a certain service
- Fabric has to manage it!
  - GCE
  - AWS
  - OpenStack

Kubernetes Node 1
- eth0
- Root ns
- cbr0
- vethxx
- pod 11

Kubernetes Node 2
- eth0
- Root ns
- cbr0

Kubernetes Node 3
- eth0
- Root ns
- cbr0
- vethoo
- vethpp
- pod 31
- pod 32

Load Balancer

§ One public IP that maps to a certain service
§ Fabric has to manage it!
- GCE
- AWS
- OpenStack
External-to-Internal Traffic

Load Balancer
- One public IP that maps to a certain service
- Fabric has to manage it!
  - GCE
  - AWS
  - OpenStack

Packet arrives to Load Balancer’s public IP

```
src: xxx:yyy
dst: 95.67.12.3
```

Kubernetes Node 1
- eth0
- Root ns
- cbr0
- vethxx
- eth0
- pod 11

Kubernetes Node 2
- eth0
- Root ns
- cbr0

Kubernetes Node 3
- eth0
- Root ns
- cbr0
- vethoo
- vethpp
- eth0
- pod 31
- pod 32
External-to-Internal Traffic

Load Balancer
- One public IP that maps to a certain service
- Fabric has to manage it!
  - GCE
  - AWS
  - OpenStack

Load Balancer

src: xxx:yyy
dst: 95.67.12.3
dst: Node Port
Packet has to be forwarded to one of the nodes

Kubernetes Node 1
- `eth0`
- `Root ns`
- `cbr0`
- `vethxx`
- `eth0`
- `pod 11`

Kubernetes Node 2
- `eth0`
- `Root ns`
- `cbr0`
- `IPtables`

Kubernetes Node 3
- `eth0`
- `Root ns`
- `cbr0`
- `veth00`
- `vethpp`
- `eth0`
- `pod 31`
- `eth0`
- `pod 32`
External-to-Internal Traffic

Load Balancer

- One public IP that maps to a certain service
- Fabric has to manage it!

- GCE
- AWS
- OpenStack

Load Balancer

src: xxx:yyy
dst: 95.67.12.3
dst: Node Port

If the LB is smart it will only forward to nodes with pod
**Load Balancer**
- One public IP that maps to a certain service
- Fabric has to manage it!
  - GCE
  - AWS
  - OpenStack

---

**External-to-Internal Traffic**

- **Load Balancer**
  - src: xxx:yyy
  - dst: 95.67.12.3
  - dst: Node1
  - If IPtables is smart, it won’t reroute to other node.

---

**Kubernetes Node 1**
- eth0
- Root ns
- cbr0
- vethx
- eth0
- pod 11

**Kubernetes Node 2**
- eth0
- Root ns
- cbr0
- IPtables

**Kubernetes Node 3**
- eth0
- Root ns
- cbr0
- IPtables
- vethoo
- vethpp
- eth0
- pod 31
- pod 32
External-to-Internal Traffic

Load Balancer
- One public IP that maps to a certain service
- Fabric has to manage it!
  - GCE
  - AWS
  - OpenStack

Load Balancer
src: xxx:yyy
dst: 95.67.12.3
Even then load balance might not be perfect!

Even then load balance might not be perfect!