



Architecting and Operating OpenShift Clusters: OpenShift for Infrastructure and O..



PREV

2. High Availab



Aa



NEXT

4. Storage



© William Caban 2019

W. Caban, *Architecting and Operating OpenShift Clusters*

https://doi.org/10.1007/978-1-4842-4985-7_3

3. Networking

William Caban¹

(1) Columbia, MD, USA

Chapter 2 covers how high availability is achieved for the core components of the platform. The communication for specific control plane



components like the synchronization of the *etcd* database, external connections to the OpenShift Console (in OCP 3.11.x), the communication from the Kubelet to the Kubernetes APIs, and external connections to the cluster's Kubernetes API goes directly to Master's Nodes IPs. Any other intercommunication among components in the cluster uses the OpenShift Networking service.

When considering the *OpenShift Networking* as a whole, there are the *OpenShift SDN plugins* to handle the east-west traffic or the traffic within the cluster and the *OpenShift Router plugins* to handle the north-south traffic, or the inbound traffic destined to *Services* in the cluster.

The default OpenShift software-defined networking (SDN) solution is built on top of Open vSwitch (OVS). With OpenShift, the *cluster admin* is free to choose to deploy with one of the OpenShift native SDN plugins or they can opt to deploy the cluster using a third-party SDN from the supported ecosystem. Should a different SDN is desired, OpenShift supports *Kubernetes CNI-compliant* SDN solutions.

There are multiple Kubernetes CNI-compliant SDN solutions in the market. If considering a third-party SDN, something to keep in mind is the alignment of the release cycle between OpenShift and the third-party SDN solution. The alignment or lack thereof, between the two, will have a direct impact in the supported upgrade cycle for the whole platform.

This chapter provides an overview of the main OpenShift SDN solutions and documents the traffic flow among Pods inside the cluster as well as how these communicate to destination outside the cluster.

East-West Traffic

For the east-west traffic , out of the box, OpenShift provides the following SDN plugins:

- OpenShift ovs-subnet
- OpenShift ovs-multitenant
- OpenShift ovs-networkpolicy



- OpenShift OVN¹ (future)
- Flannel² (limited)

In addition to the native SDN options, at the time of this writing, the following SDN solutions are validated and supported on OpenShift directly by the third-party vendors³:

- Big Switch⁴
- Cisco Contiv
- Cisco ACI CNI⁵
- Juniper Contrail
- Nokia Nuage
- Tigera Calico
- VMware NSX-T
- Kuryr SDN⁶ (or Kuryr-Kubernetes)

OPENSIFT SDN

The native *OpenShift Software-Defined Networking (SDN)* configures an *Open vSwitch (OVS)*-based overlay network to provide communication between *Pods* in the cluster. This overlay network uses the *VXLAN* protocol as the SDN encapsulation protocol.

TIP Standard VLANs provide up to 4094 VLAN IDs to segregate Ethernet traffic, but it requires for every device between two endpoints to be Layer2 devices supporting the IEEE 802.1Q protocol and maintaining the same configuration; hence its support in Cloud and hyperscaled datacenter environments is limited. By default, VLANs cannot work over the Internet, and stretched Layer2 networks are limited. On the other hand, the VXLAN protocol provides 2²⁴ or 16,777,216 VXLAN Network IDs (VNIs or VNIDs) and works over any



Layer2 or Layer3 transport (including the Internet). It only requires IP reachability between the two endpoints. Because of this and other properties, VXLAN has become the preferred transport protocol for SDN solutions.

Independent from the OpenShift SDN plugin in use, there are some default behaviors. For every node registered into the cluster, OpenShift SDN allocates a /23 subnet (see #2 of Figure 3-1) from the cluster network specified by the `osm_cluster_network_cidr` variable in the `inventory` file of the `openshift-ansible` advanced installer. If not specified, the default cluster network is 10.128.0.0/14. The cluster network subnet assigned to each node is used to assign IPs to the *Pods* at the node.

CAUTION When considering the value for `osm_cluster_network_cidr`, keep in mind that once a cluster is deployed, the cluster network cannot be arbitrarily reconfigured.

TIP The `osm_host_subnet_length` variable in the `inventory` file can be used to specify a different subnet length size, in bits, for the subnets to allocate to each registered node. The default subnet length is 9 which is a subnet of size /23. This is why, by default, OpenShift SDN allocates /23 per node, equivalent to two /24, to each node.

CAUTION The host subnet length is one of the attributes that has a direct impact in the maximum number of Pods that can run per node, and its value cannot be reconfigured after deployment.

To identify the cluster network subnet allocated to each Node, execute the “`oc get hostsubnet`” command with a user with *cluster-admin* privilege. The resulting output will be similar to Figure 3-1. The *Host IP* column (#1 in Figure 3-1) is the *Nodes* physical IP address (i.e., the IP Address of *eth0* in the *Node*) and the *Subnet* column (#2 in Figure 3-1) is the cluster network subnet allocated to the corresponding *Node*.

```
$ oc get hostsubnet
```

NAME	HOST	1 HOST IP	SUBNET	2
infranode1.demo.internal	infranode1.demo.internal	192.168.0.217	10.1.10.0/23	...
infranode2.demo.internal	infranode2.demo.internal	192.168.0.30	10.1.8.0/23	...
master1.demo.internal	master1.demo.internal	192.168.0.176	10.1.4.0/23	...
master2.demo.internal	master2.demo.internal	192.168.0.201	10.1.2.0/23	...
master3.demo.internal	master3.demo.internal	192.168.0.10	10.1.0.0/23	...
node1.demo.internal	node1.demo.internal	192.168.0.5	10.1.6.0/23	...
node2.demo.internal	node2.demo.internal	192.168.0.40	10.1.12.0/23	...
node3.demo.internal	node3.demo.internal	192.168.0.48	10.1.14.0/23	...




Figure 3-1 Sample output showing the cluster network subnet allocation

When removing or deleting a node from the cluster, the OpenShift SDN frees the corresponding cluster network subnet. This subnet becomes available for future allocations to new nodes.

NOTE Unless *Master Nodes* are also configured as *Nodes*, the OpenShift SDN will not configure or allocate a cluster network subnet for the *Master Nodes*. If the *Master Nodes* are not configured as *Nodes*, they do not have access to *Pods* via the SDN.

In every *Node* that is registered as part of a cluster, the *OpenShift SDN* registers the *Node* with the *SDN Master*. The *SDN Master* allocates a cluster network subnet for the new *Node* (see #2 in Figure 3-1). This subnet is stored in the *etcd* database of the cluster (see #2 in Figure 3-2). The *OpenShift SDN* at the *Node* creates the local host *Open vSwitch (OVS)* named *br0* with two interfaces: the *vxlan_sys_4789* in port 1 and *tun0* in port 2 of the OVS *br0* (refer to #4, #5, #8, #9, and #10 in Figure 3-2).

For each Pod in the *Node*, the local OpenShift SDN creates a *vethXX* interface and assigns it to the OVS *br0* (refer to #6 and #8 in Figure 3-2).

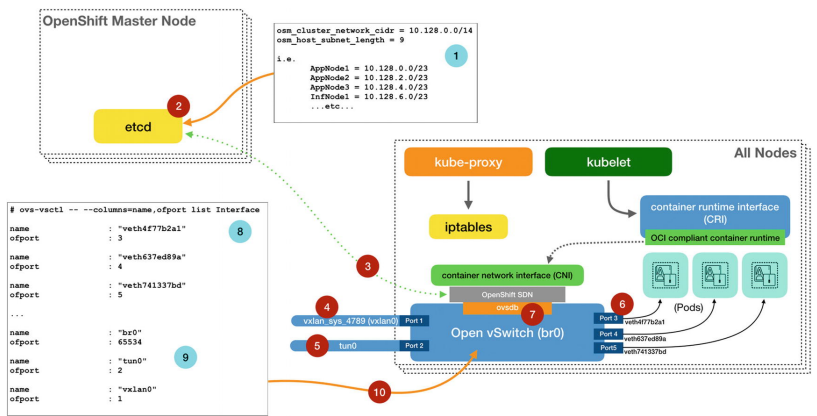


Figure 3-2 Diagram of the OpenShift SDN

During the initialization, the local *OpenShift SDN* instance injects an *OpenFlow* entry for every cluster network subnet that has been allocated by the *SDN Master*. After this, the local *OpenShift SDN* of each *Node* monitors the *SDN Master* for subnet updates. Upon detecting an update (i.e., new subnet allocation or deletion of a subnet), the local *OpenShift SDN* injects or removes a corresponding *OpenFlow* entry in the *ovsdb* in *br0*.



The `vxlan_sys_4789` of `br0` is the interface that defines the **VXLAN tunnels**, or the overlay network, that enables the communication between local *Pods* with *Pods* in remote *Nodes* (refer to #1 of Figure 3-3). This interface is known as `vxlan0` interface inside the OVS and that is the name used in the *OpenFlow* entries.

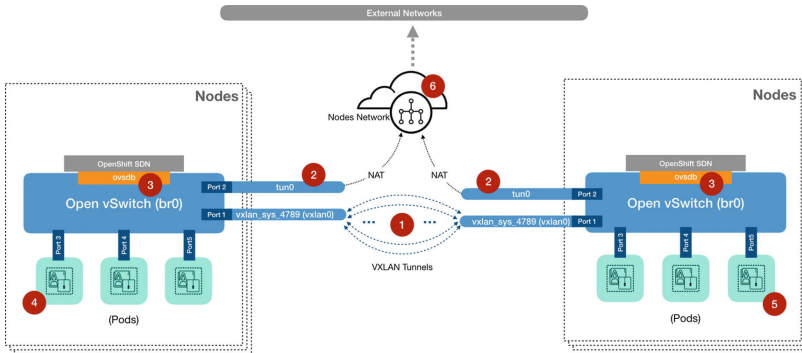


Figure 3-3 Details of the `vxlan0` and `tun0` interfaces of OpenShift SDN

The `tun0` interface gets the local cluster network subnet gateway address (see #4 of Figure 3-4). This is the interface (see #2 of Figure 3-3) that provides *NAT* access from the cluster network subnet to the external network (see #2 of Figure 3-4).

In addition to the local cluster network subnet gateway address, on each *Node* the *Kubernetes Service* objects network is also pointed to the `tun0` interface (see #1 of Figure 3-4).

```
[root@node1 ~]# ip route
default via 192.168.0.1 dev eth0 proto dhcp metric 100
10.1.0.0/16 dev tun0 scope link
...
172.30.0.0/16 dev tun0 1
192.168.0.0/24 dev eth0 proto kernel scope link src 192.168.0.5 metric 100

[root@node1 ~]# $ iptables -t nat -L OPENSHIFT-MASQUERADE
Chain OPENSHIFT-MASQUERADE (1 references)
target prot opt source destination
MASQUERADE all -- ip-10-1-0-0.demo.internal/16 anywhere 2 /* masquerade pod-to-service and pod-to-external traffic */

$ ip addr
...
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc mq state UP group default qlen 1000 3
   link/ether 8e:b9:4d:1c:85:8a brd ff:ff:ff:ff:ff:ff
   inet 192.168.0.5/24 brd 192.168.0.255 scope global noprefixroute dynamic eth0
       valid_lft 3428sec preferred_lft 3428sec
   inet6 fe80::1430:3cfe:fe26:3094/64 scope link
       valid_lft forever preferred_lft forever
...
11: tun0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 8951 qdisc noqueue state UNKNOWN group default qlen 1000 4
   link/ether 8e:b9:4d:1c:85:8a brd ff:ff:ff:ff:ff:ff
   inet 10.1.6.1/23 brd 10.1.7.255 scope global tun0
       valid_lft forever preferred_lft forever
   inet6 fe80::8cb9:4dff:felc:858a/64 scope link
       valid_lft forever preferred_lft forever
...
```

Figure 3-4 Details of routes and NAT for `tun0`

In OpenShift, the *Service* network configuration is set by the `openshift_portal_net` variable in the *inventory* file. If this variable is not defined, the default *Service* network is 172.30.0.0/16.



TIP After the initial installation of the cluster, the service network can be expanded as long as the existing network is at the beginning of the new network range. ⁷

As new *Pods* are created on a host, the local *OpenShift SDN* allocates and assigns an *IP Address* from the cluster network subnet assigned to the *Node* and connects the *vethXX* interface to a port in the *br0* switch. At the same time, the *OpenShift SDN* injects new *OpenFlow* entries into the *ovsdb* of *br0* to route traffic addressed to the newly allocated *IP Address* to the correct *OVS* port connecting the *Pod*.

OpenShift ovs-subnet

The *OpenShift ovs-subnet* is the original *OpenShift SDN* plugin. This plugin provides basic connectivity for the *Pods*. In the *OpenShift* official documentation, this network connectivity is sometimes referred to as a “flat” Pod network. That may cause some confusion with season network engineers. For any network engineer, the term “flat” network will be interpreted as a network where there are no subnetting and sharing of the same broadcast domain. That would be a very bad network design and would be prone to constant broadcast storms. Fortunately, that is not the case with *ovs-subnet*.

With the *OpenShift SDN ovs-subnet* plugin, each *Node* still receives a dedicated /23 cluster network subnet (see #1, #2, and #3 of Figure 3-5). Then, the local *OpenShift SDN* instance sets up *OpenFlow* entries for each cluster network subnet defined by the *SDN Master* (#4, #5, and #6 of Figure 3-5 provide a conceptual representation of these).

The reason it is described as a “flat” Pod network is because there are no filters or restrictions and every Pod can communicate with every other Pod and Service in the cluster. So, from the networking perspective, this will be a fully meshed and unfiltered network. In this case, any Pod in Node 1 (#8 of Figure 3-5) will have reachability to the Pods in Node 2 and Node 3 (see #9 and #10 in Figure 3-5) and vice versa.

NOTE Even when *Pods* may have reachability to any other *Pod* in the cluster, they will only see open the *Ports* explicitly enabled by the



destination *Pod* definition. For example, a *Pod* definition opening TCP Port 8080 will only allow traffic to TCP 8080 to arrive to the container inside the *Pod* and will block everything else.

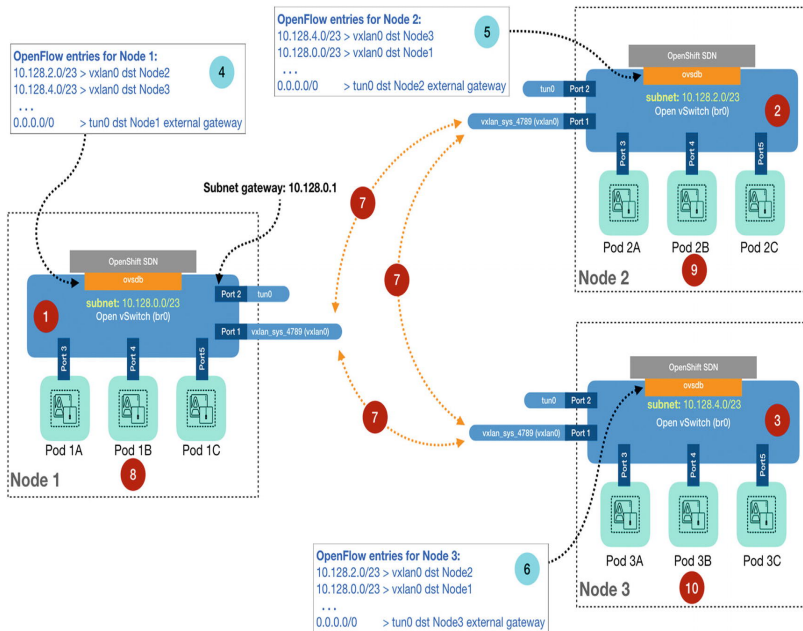


Figure 3-5 Representation of OpenShift SDN ovs-subnet plugin

OpenShift ovs-multitenant

With *OpenShift ovs-multitenant* plugin, each *Project* receives a unique *VXLAN ID*, also known as a *Virtual Network ID (VNID)*. All the *Pods* and *Services* of a *Project* are assigned to the corresponding *VNID*. By doing this, it maintains project-level traffic isolation. Meaning, *Pods* and *Services* of one *Project* can only communicate with *Pods* and *Services* in the same *Project*. By definition, there is no way for *Pods* or *Services* from one *Project* to send traffic into another *Project*.

The underlying cluster network subnet allocation remains the same. Each Node receives a dedicated /23 cluster network subnet (see #1, #2, and #3 of Figure 3-6). After this, the local OpenShift SDN instance sets up the OpenFlow entries for each cluster network subnet defined by the SDN Master (see #4, #5, and #6 of Figure 3-6).



After this point, it starts differentiating from the other plugins. When using *ovs-multitenant*, the *OpenShift SDN Master* monitors the creation and deletion of *Projects*. Upon the creation of a new *Project*, it allocates and assigns a *VXLAN ID* to the *Project*. This *VXLAN ID* is the one used to isolate the traffic of the *Project* (see #11 of Figure 3-6).

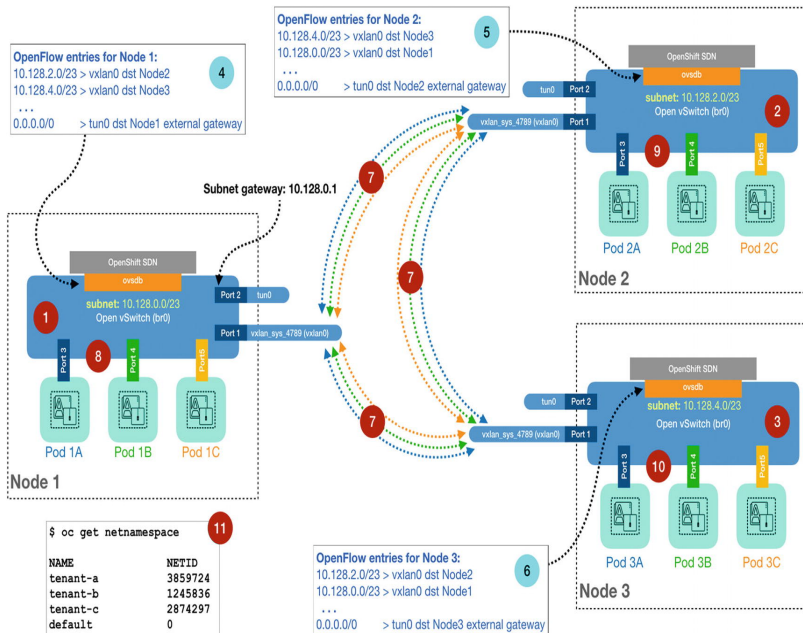


Figure 3-6 Representation of OpenShift SDN *ovs-multitenant* plugin

When a new *Pod* is instantiated in a cluster using the *ovs-multitenant* plugin, during the process of injecting the OpenFlow entries into *br0*, the *OpenShift SDN* includes OpenFlow rules to tag traffic coming from the *br0* port connecting the *Pod* with the *VNI* corresponding to its *Project*. In addition, it adds explicit rules to only allow traffic into the *Pod* if traffic's *VNI* matches the *Pod*'s *VNI* or is coming from a privileged *VNI* 0.

NOTE When using *ovs-multitenant*, the *VNI=0* is considered privileged traffic that can communicate with any *Project*, and any *Project* can send traffic to a *Project* with *VNI=0*. OpenShift assigns *Project "default"* to *VNI=0* (see #11 of Figure 3-6). Among other *Pods* and *Services*, *Project "default"* contains the *Pods* and *Services* for the internal *Container Registry (OCR)* and the *OpenShift Router*.



When sending traffic across the `vxlano` interface to a remote *Node*, the traffic is tagged with the correct VNID matching the source Pod Project VNID. The VNID is used as the *VXLAN Tunnel ID* (see #7 of Figure 3-6 where the colors represent the different *VNIDs*). The receiving *Node* uses the *VXLAN Tunnel ID* as the *VNID* tag for the traffic. This guarantees end-to-end isolation of traffic from different projects.

OpenShift ovs-networkpolicy

The *OpenShift ovs-networkpolicy* plugin, fully supported since OpenShift 3.7, is a modern SND that implements the *Kubernetes Network Policies* capabilities. In the default configuration, all *Pods* have reachability to any other *Pod* or *Service* in the cluster.

To restrict traffic to or from a *Pod* or to isolate *Pods*, a *NetworkPolicy* resource must be defined (see #8 in Figure 3-7). Once a *NetworkPolicy* is configured in a *Project* or *Namespace* selecting a particular *Pod*, there will be an implicit *deny-all* rule rejecting all the traffic to that *Pod* and only allowing traffic from connections explicitly allowed by the *NetworkPolicy*. These policies will not impact or affect any other *Pods* in the same *Project*, and those will continue to receive all traffic directed to them.

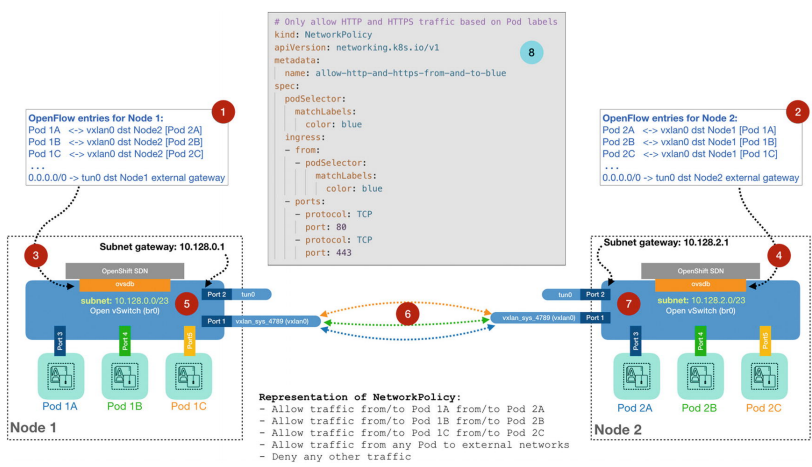


Figure 3-7 Representation of the OpenShift SDN ovs-networkpolicy plugin

For each Node, these network policies are enforced by *OpenFlow* entries in the *bro0* switch (see #1 in Figure 3-7 for representation).



The *NetworkPolicy* resource provides robust network policy mechanisms. As such, it is up to the *cluster-admin* or *Project admin* to define the desired policies for a *Project*. The additive property of these objects enables for multiple *NetworkPolicy* objects to be combined together to create advanced and complex network policies.

As with any other *Kubernetes* resource, the *NetworkPolicy* resource is expressed in YAML format.

Consider Listing 3-1 for an example *NetworkPolicy* definition to deny all traffic from and to any *Pod* in a *Project* or *Namespace*. After applying this policy, all *Pods* in the particular *Project* become isolated.

```
# Deny All Traffic (isolate all Pods in namespace)
# oc create -f 3.1_deny-all.yaml -n <your-namespace>
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: deny-all
spec:
  podSelector:
    ingress: []
```

Listing 3-1 NetworkPolicy to deny all traffic and isolate Pods

Figure 3-8 shows the process of applying this *NetworkPolicy*. On #1 in Figure 3-8, the output shows there are two *Pods*. In #2 in Figure 3-8, a *tcpping* Python function is used to demonstrate a *TCP* connection to the *PostgreSQL* *Pod* is possible. Then the policy is applied in #3 in Figure 3-8. On #4 and #5 in Figure 3-8, there is a validation that the *NetworkPolicy* has been created. Finally, #6 in Figure 3-8 shows the execution of *tcpping*, and this time the connection is blocked.



```

$ oc get pods -o wide --show-labels
NAME          ... IP          NODE    1          ... LABELS
podcool-1-gl86q ... 10.128.2.9   ccp-n3.shift.zone ... app=podcool,deployment=podcool-1,deploymentconfig=podcool
postgresql-1-5s54b ... 10.129.0.8   ccp-n2.shift.zone ... deployment=postgresql-1,deploymentconfig=postgresql,name=postgresql

$ oc exec podcool-1-gl86q -- python -c "from topping import topping ; topping(d_host='postgresql',d_port=5432, maxCount=3, DEBUG=True)"
Connected to postgresql[5432]: tcp_seq=0 time=2.92 ms
Connected to postgresql[5432]: tcp_seq=1 time=1.41 ms    2
Connected to postgresql[5432]: tcp_seq=2 time=1.22 ms

TCP Ping Results: Connections (Total/Pass/Fail/Avg): [3/3/0/1.85] (Failed: 0%)

$ oc create -f deny-all.yaml -n demo-policy
networkpolicy.networking.k8s.io/demo-policy created    3

$ oc get networkpolicy
NAME          POD-SELECTOR  AGE    4
deny-all     <none>        1m

$ oc describe networkpolicy deny-all    5
Name:          deny-all
Namespace:     demo-policy
Created on:    2019-01-17 18:40:52 -0500 EST
Labels:        <none>
Annotations:   <none>
Spec:
  PodSelector:  <none> (Allowing the specific traffic to all pods in this namespace)
  Allowing ingress traffic:
    <none> (Selected pods are isolated for ingress connectivity)
  Allowing egress traffic:
    <none> (Selected pods are isolated for egress connectivity)
  Policy Types: Ingress

$ oc exec podcool-1-gl86q -- python -c "from topping import topping ; topping(d_host='postgresql',d_port=5432, maxCount=3, DEBUG=True)"
Connection timed out!    6
Connection timed out!
Connection timed out!

TCP Ping Results: Connections (Total/Pass/Fail/Avg): [3/0/3/1001.993] (Failed: 100.00%)

```

Figure 3-8 Applying NetworkPolicy to isolate Pods by blocking all traffic to them

Following the same exercise, consider Listing 3-2. This *NetworkPolicy* allows every *Pod* to communicate to any other *Pod* in the same *Project* and enables access to the *default* Project.

NOTE When using *NetworkPolicy* resources, the communication with *Project “default”* is required to get to the *OpenShift Routers*. This rule must be explicitly allowed by the defined policy.

```

# Allow traffic between Pods in the same Project and
with the default project (i.e. to access the routers)
# oc label namespace default name=default
# oc create -f 3.2_allow-same-project-and-default.yaml
-n <your-namespace>

kind: NetworkPolicy
apiVersion: extensions/v1beta1
metadata:
  name: allow-same-and-default-namespace
spec:
  ingress:
    - from:
      - podSelector: {}
      - from:
        - namespaceSelector:
            matchLabels:
              name: default

```



Listing 3-2 NetworkPolicy to allow traffic within Pods in the Project and with the default Namespace

Figure [3-9](#) documents the application of Listing [3-2](#) *NetworkPolicy* (#1) to restore the communication with the PostgreSQL Pod (#3).

```
$ oc create -f allow-same-and-default-ns.yml -n demo-policy
networkpolicy.extensions/allow-same-and-default-namespace created 1

$ oc describe networkpolicy allow-same-and-default-namespace
Name:
allow-same-and-default-namespace
Namespace: demo-policy
Created on: 2019-01-17 18:47:50 -0500 EST 2
Labels:
Annotations: <none>
Spec:
  PodSelector: <none> (Allowing the specific traffic to all pods in this namespace)
  Allowing ingress traffic:
    To Port: <any> (traffic allowed to all ports)
    From:
      PodSelector: <none>
      -----
      To Port: <any> (traffic allowed to all ports)
      From:
        NamespaceSelector: name=default
  Allowing egress traffic:
    <none> (Selected pods are isolated for egress connectivity)
  Policy Types: Ingress

$ oc exec podcool-1-gl86g -- python -c "from topping import topping ; topping(d_host='postgresql',d_port=5432, maxCount=3, DEBUG=True)"
Connected to postgresql[5432]: tcp_seq=0 time=3.34 ms
Connected to postgresql[5432]: tcp_seq=1 time=1.88 ms
Connected to postgresql[5432]: tcp_seq=2 time=1.09 ms 3

TCP Ping Results: Connections (Total/Pass/Fail/Avg): [3/3/0/3.103] (Failed: 0%)
$
```

Figure 3-9 Applying NetworkPolicy to allow traffic among Pods and with Project default

FLANNEL

Flannel is one of the simplest SDN implementations of the *Kubernetes network model*. It supports various overlay protocols (or backends) ranging from *VXLAN* to *host-gw*, and many others.⁹ The OpenShift-supported *Flannel* configuration uses the *host-gw* backend.¹⁰

NOTE In OpenShift, the support of the *Flannel* plugin is limited to deployments of *OpenShift Container Platform* over the Red Hat OpenStack Platform.¹¹

The *host-gw* backend requires Layer2 connectivity between the Nodes so *flanneld* can forward the packets to the corresponding *Node* as *next-hop*. The *Flannel* SDN initialization in OpenShift is as follows:

- Each *Node* runs a *flanneld* agent which reads the configuration from the *etcd* database (see #11 of Figure [3-10](#)).
- The *flanneld* agent allocates a unique /24 subnet from the configured *Network* and registers the allocated *Node* host subnet into the *etcd* database (see #12 of Figure [3-10](#)).



- The first IP of the subnet is assigned as the interface *docker0* (#1 of Figure [3-10](#)) which becomes the *default gateway* for the local *Pods*.
- For each allocated host subnet in *etcd*, Flannel host-gw backend injects a subnet route with the remote *Node* *eth0* IP Address as the *next-hop* gateway address to reach that subnet (see #2 of Figure [3-10](#)).

With *Flannel host-gw* backend, the traffic flow from a Pod in one *Node* to a Pod in another *Node* is as follows:

1. The Pod sends traffic to its default gateway. For example, Pod 1A in Node 1 sends traffic to Pod 2C in Node 2 (Figure [3-10](#)). Pod 1A sends traffic to its default gateway, which happens to be the *docker0* interface (#1 of Node 1 in Figure [3-10](#)).
2. From the *docker0* interface, the traffic is routed by the *host routing table* (#5 in Figure [3-10](#)). Since there is a specific route for the destination subnet (#2 in Figure [3-10](#)), the traffic is sent to the registered *next-hop* address (#7 in Figure [3-10](#)) which, in this example, happens to be Node 2 *eth0* IP Address.
3. Once the traffic is received by the remote *Node* (Node 2), the destination *IP Address* is evaluated by the host routing table (#9 in Figure [3-10](#)) so the traffic is sent to *docker0* interface which finally forwards the traffic to Pod 2C (#10 in Figure [3-10](#)).

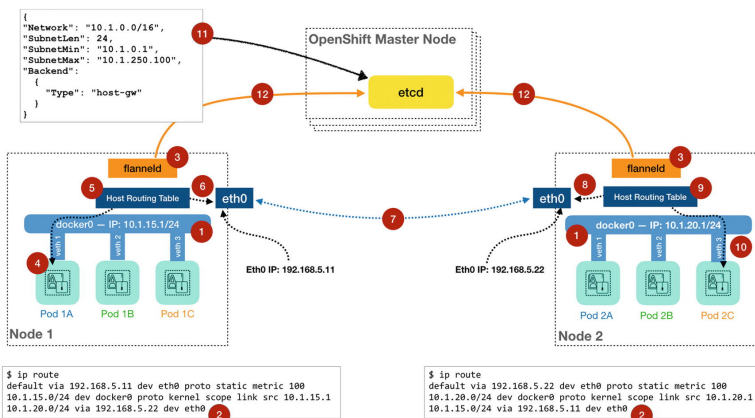


Figure 3-10 Flannel SDN with host-gw backend in OpenShift



Because *Flannel* with the *host-gw* backend does not use additional encapsulations, it maintains certain level of performance, and the *host-gw* backend is considered a good option when deploying Kubernetes over virtualized platforms that have their own SDN solutions. This is to avoid the performance penalties which might be experienced when using SDNs over SDNs, resulting in what is known as double encapsulation.

OPENSIFT WITH THIRD-PARTY SDN

OpenShift configurations with third-party SDN are maintained by their respective third-party vendors. To illustrate the use of third-party SDNs with OpenShift in this section, we focus on the *Open Source Calico*¹² SDN solution.

OpenShift with Calico SDN

The *Calico SDN CNI* provides another *SDN* alternative supporting *NetworkPolicy* resources for *ingress* and *egress* policy rules. Calico can be used with or without an encapsulated overlay network. In OpenShift, by default it uses IP over IP encapsulation.

Calico relies on routing principles from the native Linux network stack to move traffic from one *Node* to another. It can be used with Nodes using *Layer2* or *Layer3* connectivity.

As with other *Kubernetes SDN* solutions, *Calico* maintains its configuration and state in the cluster *etcd* database and relies on the *BGP* protocol at each *Node* to communicate the routing information.

TIP A best practice for large-scale cluster deployments with Calico is to have a dedicated *etcd* instance for it, different from the cluster *etcd*.

NOTE If BGP is supported by the *top-of-rack (TOR)* switches interconnecting the cluster, *Calico* can peer with the *TOR* over BGP. The default BGP ASN is 64512. This ASN value is configurable by CLI.¹³



By default, *Calico* allocates a /26 subnet to each *Node*, and as IPs are consumed by the *Node*, it dynamically allocates additional blocks to the *Node*. This is possible thanks to the use of a dynamic routing protocol, in this case BGP, on each *Node*.

Various components come together to create the Calico architecture (see Figure 3-11):

1. **CNI Plugin:**

- a. **Calico-CNI:** The Calico CNI plugin implements the Kubernetes CNI specification.
- b. **Calico-IPAM:** The Calico IPAM assigns IP address to the Pods.

2. **calico-node:** The *calico-node* is a privileged container running as *DaemonSet* in every *Node* (see #1 of Figure 3-11). This container has three elements:

- a. **confd:** Monitors the *etcd* database for state updates and generates the corresponding new *BGP* configuration for *BIRD*.
- b. **BIRD and BIRD6:** *BGP* agents running at each *Node* and distribute the routes across. *BIRD* is for IPv4 addresses and *BIRD6* for IPv6 addresses.
- c. **Felix:** Agent doing the routing and policy calculation. It writes the corresponding routes and ACLs to the *Node* host routing table and *iptables*, respectively.

3. **calico-kube-controller:** This container runs as a *Pod* on top of Kubernetes and maintains Calico in sync with *Kubernetes* when using *NetworkPolicy*.



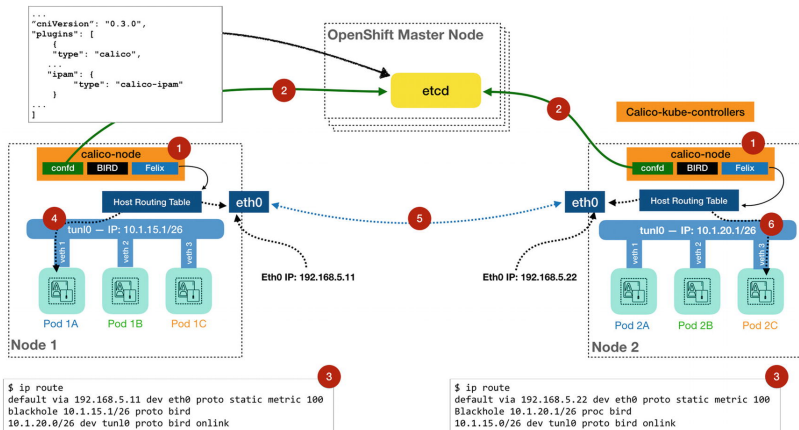


Figure 3-11 Representation of Calico SDN in OpenShift

From #3 in Figure 3-11, we can see an extract of the resulting host routing table when using *Calico*. Local Pod-to-Pod traffic has direct communication inside the host. To reach a *Pod* in a remote Node, the traffic from a *Pod* gets to the local *tunl0* interface (#4 in Figure 3-11) and gets routed by the *host routing table* to the *next-hop* IP Address which is the remote Node. At the remote Node, the packet is routed by the host routing table and delivered to the *tunl0* interface (#6 in Figure 3-11) where it finally reaches the remote *Pod*.

North-South Traffic

When considering the north-south traffic, out of the box, the available *OpenShift Router plugins*¹⁴ are

- HAProxy Template Router (default plugin)
- F5 BIG-IP Router plugin

In addition to the official supported plugins, at the time of this writing, a third-party supported OpenShift Router plugin is

- NGINX and NGINX Plus Router¹⁵

HAPROXY TEMPLATE ROUTER



The default *OpenShift Router* is one or more *Router Pods* running on Infrastructure Nodes (see #1 of output shown in Figure 3-12) and is deployed as a *Deployment Config* (see #5 of output shown in Figure 3-12).

```
$ oc get all --selector='router=router' -n default -o wide
```

NAME	READY	STATUS	...	1	NODE	...
pod/router-1-8nsg	1/1	Running	...		infranode1.demo.internal	...
pod/router-1-pxsg	1/1	Running	...		infranode2.demo.internal	...

NAME	DESIRED	CURRENT	READY	AGE	CONTAINERS	...	2
replicationcontroller/router-1	2	2	1d		router	...	
...	
SELECTOR	deployment=router-1,deploymentconfig=router,router=router	

3	NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	4
	service/router	ClusterIP	172.30.117.14	<none>	80/TCP,443/TCP,1936/TCP	...

	SELECTOR	router=router

5	NAME	REVISION	DESIRED	CURRENT	TRIGGERED BY
	deploymentconfig.apps.openshift.io/router	1	2	2	config

Figure 3-12 Output showing the elements comprising the OpenShift Router service

These *Router* container images are based on *HAProxy* (see #6 of *Pod* definition extract shown in Figure 3-13). These *Pods* are defined to share the *Network Namespace* with the host *Infrastructure Node* (see #5 and #8 of extract shown in Figure 3-13).

apiVersion: v1	1
kind: Service	
metadata:	
...	
labels:	2
router: router	
name: router	
namespace: default	
...	
spec:	
clusterIP: 172.30.117.14	
ports:	
- name: 80-tcp	3
port: 80	
protocol: TCP	
targetPort: 80	
- name: 443-tcp	
port: 443	
protocol: TCP	
targetPort: 443	
- name: 1936-tcp	
port: 1936	
protocol: TCP	
targetPort: 1936	
selector:	
router: router	
sessionAffinity: None	
type: ClusterIP	

apiVersion: v1	
kind: Pod	4
metadata:	
annotations:	
openshift.io/deployment-config.latest-version: "1"	
openshift.io/deployment-config.name: router	
openshift.io/deployment.name: router-1	
openshift.io/scc: hostnetwork	5
labels:	
deployment: router-1	
deploymentconfig: router	
router: router	
...	
spec:	
containers:	6
...	
image: registry.redhat.io/openshift3/ose-haproxy-router:v3.11.51	
imagePullPolicy: IfNotPresent	
livenessProbe:	
failureThreshold: 3	
httpGet:	
host: localhost	
path: /healthz	
port: 1936	
scheme: HTTP	
...	
name: router	
ports:	7
- containerPort: 80	
hostPort: 80	
protocol: TCP	
- containerPort: 443	
hostPort: 443	
protocol: TCP	
- containerPort: 1936	
hostPort: 1936	
protocol: TCP	
name: stats	
protocol: TCP	
...	
hostNetwork: true	
nodeName: infranode1.demo.internal	8
nodeSelector:	
node-role.kubernetes.io/infra: "true"	
...	

Figure 3-13 Extract of an OpenShift Router Service and Pod definition

Sharing the *Network Namespace* enables these *Router Pods* to receive traffic over the *host-network*. By default, the *OpenShift Router* listens on TCP ports 80 (HTTP), 443 (HTTPS), and 1936 (HAProxy Stats) (see #3 and #7 in Figure



[3-13](#)). Once the traffic arrives to the Pod, it will match the corresponding Route object (see #1 and #2 of [Figure 3-14](#)).

During the creation of the Route resource (#1 in [Figure 3-14](#)) and at the addition or removal of a *Pod*, the *OpenShift Router* queries the *Service* resource (#3 in [Figure 3-14](#)) for the Endpoints associated to the *Service* based on label selectors (#5 in [Figure 3-14](#)). From here it obtains Endpoint information like name and IP of the *Pods*. The OpenShift Router uses this information to create the corresponding HAProxy configuration to load balance the traffic (#6 in [Figure 3-14](#)) destined to the particular *Route* (i.e., myapp-demo-app.example.com) across the available *Pods*.

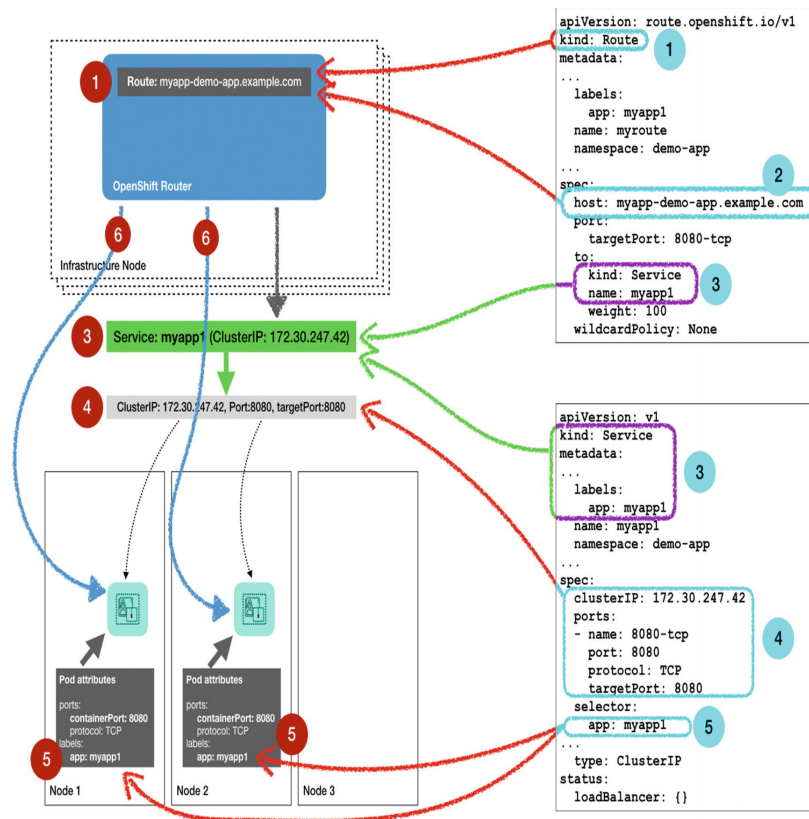


Figure 3-14 OpenShift Route to Service details

Summary



OpenShift Networking is comprised of multiple elements that can be grouped into two types of solutions: the solutions that provide the Software-Defined Networking (SDN) to move the east-west traffic, or traffic within the cluster, and the solutions that handle the north-south traffic, or the inbound traffic to applications hosted on the OpenShift cluster.

For both cases, for the east-west traffic and for the north-south traffic, there are the OpenShift native supported plugins and third-party validated plugins supported by those third-party vendors.

The next chapter, Chapter 4, explores the available options for providing storage to components and applications running on the platform.

Footnotes

1 The OpenShift Open Virtual Networking (OVN) plugin is considered a development preview. The current capabilities for OCP OVN are similar to the *ovs-networkpolicy*. More information about OVN can be found at the Kubernetes OVN upstream project under the Open vSwitch project:

<https://github.com/openvswitch/ovn-kubernetes>

2 Flannel is only supported when OCP is deployed over OpenStack environments which are using a VXLAN-based SDN to work around issues with the possible VXLAN over VXLAN encapsulation.

3 For an updated list of the supported third-party vendor, visit https://docs.openshift.com/container-platform/3.11/install_config/configuring_sdn.html#admin-guide-configuring-sdn-available-sdn-providers

4 Additional information about Big Switch Big Cloud Fabric Enterprise Cloud (BCF-EC) integration with OpenShift is available here: www.bigswitch.com/tech-partner/red-hat (<http://www.bigswitch.com/tech-partner/red-hat>)



5

For more information about the Cisco ACI CNI Plugin for OCP, refer to

www.cisco.com/c/en/us/td/docs/switches/datacenter/aci/apic/white_papers/ACI-CNI-Plugin-for-OpenShift-Architecture-and-Design-Guide.pdf
(http://www.cisco.com/c/en/us/td/docs/switches/datacenter/aci/apic/white_papers/Cisco-ACI-CNI-Plugin-for-OpenShift-Architecture-and-Design-Guide.pdf)

6

At the time of this writing, the Kuryr SDN is considered Technology

Preview; for more information, refer to

https://docs.openshift.com/container-platform/3.11/install_config/configuring_kuryrsdn.html

7

For details on expanding the Service network, refer to

https://docs.openshift.com/container-platform/3.11/install_config/configuring_sdn.html#expanding-the-service-network

8

Additional detail of *Kubernetes Network Policies* is available at

<https://kubernetes.io/docs/concepts/services-networking/network-policies/>

9

For a complete list of the backend protocols supported by the Flannel SDN, refer to

<https://github.com/coreos/flannel/blob/master/Documentation/backends>

[10](#) Additional details about OpenShift and Flannel are available at the following URL (note: a valid Red Hat support subscription is required to access this link): https://access.redhat.com/documentation/en-us/reference_architectures/2018/html/deploying_and_managing_openshift



[11](#) For more information of OpenShift Flannel, see https://docs.openshift.com/container-platform/3.11/install_config/configuring_sdn.html#using-flannel

[12](#) Additional information about project Calico can be found at www.projectcalico.org (<http://www.projectcalico.org>)

[13](#) For information on customizing the BGP ASN number, visit <https://docs.projectcalico.org/v3.4/usage/configuration/bgp#configuring-the-default-node-as-number>

[14](#) For an updated list of available Router plugins, visit https://docs.openshift.com/container-platform/3.11/architecture/networking/assembly_available_router_plugins.html

[15](#) Additional details about the NGINX and NGINX Plus OpenShift Routers are available at NGINX Inc Git repo: <https://github.com/nginxinc/nginx-openshift-router>

