PREV
9. Day-2 Opera

NEXT
I Mode Ba…

W. Caban, *Architecting and Operating OpenShift Clusters*
https://doi.org/10.1007/978-1-4842-4985-7_10

10. Advanced Network Operations

William Caban [1]

(1) Columbia, MD, USA

The OpenShift platform provides defaults optimized for Cloud-native workloads. These have been covered throughout this book. As with many successful Open Source project, Kubernetes is being used in setups for which it was never designed. With the adoption of Kubernetes outside the web-based application, there is the need to support specialized hardware acceleration (i.e., GPUs, FPGAs, etc.), multiple NICs per Container, and much more. This chapter focuses on advanced networking features or capabilities for increasing network performance and for the onboarding of applications or microservices using nontraditional web protocols into *OpenShift*.

Network Optimizations

OpenShift SDN uses *OpenvSwitch*, *VXLAN* tunnels, *OpenFlow* rules, and *iptables* or *firewalld* rules. Some possible optimizations to this overlay network are based on best practices for fine-tuning a system in a high-performance environment.

JUMBO FRAMES AND VXLAN ACCELERATION

The standard Ethernet Maximum Transmission Unit (MTU) is 1500 Bytes. A regular IP UDP packet will consume 20 Bytes for the IP header (see #2 in Figure 10-1) and 8 Bytes for the UDP header (see #3 in Figure 10-1), and the remaining 1472 Bytes are available for payload (see #4 to #8 in Figure 10-1).

> **NOTE** The outer Ethernet header (14 Bytes) (see #1 in Figure 10-1) is not counted as part of the MTU.

In SDN networks using the VXLAN protocol, the whole Ethernet frame of traffic from *Pods* in one *Node* destined to *Pods* in another *Node* is encapsulated as IP UDP packets and forwarded to the *Node* running the destination *Pods*. For this, the VXLAN header (see #4 in Figure 10-1) is added to the original Layer 2 Ethernet frame (see #5 in Figure 10-1), minus its FCS, and all this content becomes the payload of the outer IP UDP packets (see #2 and #3 in Figure 10-1) and is sent to the remote *Node*.

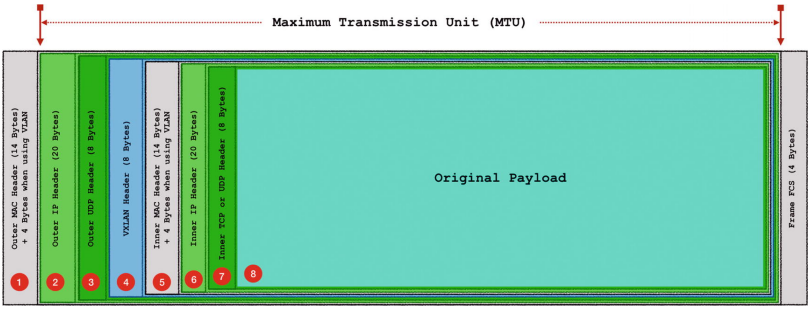

Figure 10-1 Format of a VXLAN packet

From the diagram in Figure 10-1, the effective MTU for payload (see #8) is 1422 Bytes. When working with environments with large streams of data to transfer among Pods on different Nodes, those streams of data need to be broken into very small chunks of 1422 Bytes or less. Each one of these packets undergoes an encapsulation process. Under high network utilization or high network throughput, this may lead to high CPU utilization and high latency.

To reduce the CPU utilization and latency under such circumstances, the recommendations are

- Use Jumbo frames (i.e., MTU 9000 or more) to be able to send more data per packet and reduce the number of packets and overhead required to move the data from one Node to the other.
- Use NIC cards supporting VXLAN acceleration so the encapsulation process is offloaded to hardware and CPU.

> **CAUTION** Not all the VXLAN-accelerated NICs support Jumbo frames. Consult the technical specification of your NIC provider. In those cases where VXLAN acceleration is not supported with Jumbo frames, the cluster administrators should avoid Jumbo frames as the NIC driver will determine the final behavior which may have a negative impact in performance.

TUNING NETWORK DEVICES

Advanced Linux system administrator with deep understanding of the Linux networking stack and the available tuning options for high-performance

computing may use similar techniques with OpenShift clusters.

> **CAUTION** Some of the following optimizations have limited availability or configuration options in some NIC drivers. Consult your hardware and driver technical information.

Some of the optimizations that may be considered are

- **Adjusting the number and size of RX and TX queues**: Improved throughput, latency, and multi-queue techniques can be used to distribute the processing of queues across multiple CPUs.
- **Interrupt coalescing**: Prevents interrupt storms and increases throughput or latency.
- **Adaptive RX and TX coalescing**: Interrupt delivery is optimized to improve latency or throughput based on packet rate.
- **Hardware-accelerated Receive Flow Steering (RFS)** : When supported by the NIC's driver, the NIC and the Kernel work together to determine which flows to send to which CPU for processing.
- **Adjusting IRQ affinity**: Optimizes for data locality for interrupts generated by the NIC.
- **Adjusting UDP receive queue size**: Increases throughput.
- **Generic Receive Offloading (GRO) and Large Receive Offloading (LRO)**
- **Receive Packet Steering (RPS) and Receive Side Scaling (RSS)**

Some of the Linux commands used for these optimizations are *sysctl* and *ethtool*.

These levels of optimizations are unique to each hardware and driver combination. As such, this book highlights the existence of these capabilities but leaves it to the reader to explore and test the ones suitable for their environment.

Routing Optimizations

The *OpenShift Router* can handle the *Routes* for multiple applications. This can be anywhere from one to thousands of applications. The actual number of Routes an OpenShift Router can handle is determined by the technology in use by the applications behind the *Routes*.

As seen in previous chapters, the OpenShift Router is based on HAProxy. One of the tunable parameters for HAProxy is the *maxconn* parameter which is configurable by using the *ROUTER_MAX_CONNECTION* [1] environment variable of the *OpenShift Router DeploymentConfig*. This parameter sets the per-process maximum number of concurrent connections.

> **NOTE** When configuring the *maxconn* parameter, consider *HAProxy* counts of the *frontend* connection and *backend* connection as two different connections. Because of this, a connection from an external client to an application load balanced by HAProxy counts as two.

Additional parameters [2] for the optimization of the OpenShift Router are

- CPU and interrupt affinity
- Increasing number of threads
- Setting up connection timeouts

> **NOTE** *OpenShift Routers*, by default, listen on ports 80 (HTTP) and 443 (HTTPS), but they can be configured to listen for HTTP and HTTPS traffic on other ports. This option is configured using the environment variable `ROUTER_SERVICE_HTTP_PORT` and the environment variable `ROUTER_SERVICE_HTTPS_PORT`.

ROUTE-SPECIFIC OPTIMIZATIONS ANNOTATIONS

In addition to the global configuration parameters of the HAProxy, OpenShift provides the ability to modify certain behavior on per-Route basis. This is done by using Route Annotations (see Table 10-1).

Table 10-1 OpenShift Route Annotations [3]

Variable	Descript
haproxy.router.openshift.io/balance	Load balancing algorithm *source*, *roundrob* or *leastco*
haproxy.router.openshift.io/disable_cookies	Disables t use of *cookies* to track *rela* connectio
router.openshift.io/cookie_name	Optional *cookie* to for *Route*

Variable	Descript
haproxy.router.openshift.io/pod-concurrent-connections	Sets the maximun number c connectic allowed f each back *Pod* from specific *Router*
haproxy.router.openshift.io/rate-limit-connections.concurrent-tcp	Limits th number c concurre TCP connectic by an IP address
haproxy.router.openshift.io/rate-limit-connections.rate-http	Limits th rate at wh an IP address c make HT requests

Variable	Descript
haproxy.router.openshift.io/rate-limit-connections.rate-tcp	Limits th rate at wh an IP address c make TCI connectic
haproxy.router.openshift.io/timeout	Sets a server-sic timeout f the *Route*
router.openshift.io/haproxy.health.check.interval	Sets inter for the backend health checks
haproxy.router.openshift.io/ip_whitelist	(See "*IP Whitelists* section)

Variable	Descript
haproxy.router.openshift.io/hsts_header	Sets a Str Transpor Security header fo the terminate or re-encrypt *Route*

IP WHITELISTS

OpenShift supports the use of special annotations to restrict which source IP address or network can access a specific *Route*. The *ip_whitelist* annotation (see Figure 10-2) is a space-separated list of whitelisted source IP addresses and CIDRs that are allowed to access the particular *Route*.

```
...
metadata:
  annotations:
    haproxy.router.openshift.io/ip_whitelist: 209.132.183.105 192.168.1.0/24 10.5.25.0/24
...
```

Figure 10-2 IP Whitelist annotations for a Route

OPENSHIFT ROUTER SHARDING

To horizontally scale the routing layer, *OpenShift* provides the capability to define and use *Router Shards* . In this case, the *Routes* are shared among a group of *Routers* based on a *selection expression* defining the *Shard*. There are two levels of Route sharding:

- Cluster administrators configure and manage sharding at cluster-wide level.

- Users can configure sharding for namespaces where they have admin privileges.

When using sharding, each *Router* in the group handles a portion of the traffic based on the assigned *Shard*.

> **NOTE** Based on the *selection expression*, the *Router Shards* can be unique, in which case a *Route* belongs to only one *Shard*, or there can be overlapping in which case some *Routes* can belong to more than one *Shard*.

When using *Router Sharding*, the first *Route* matching a particular *Shard* reserves the right to exist on that *Shard* permanently and even across restarts. Figure 10-3 illustrates both ways of configuring *Router Sharding*.

USING NAMESPACE LABELS

```
# Create new router (no replicas)
oc adm router router-dev --replicas=0

# Set the NAMESPACE selection expression
oc set env  dc/router-dev NAMESPACE_LABELS="router=dev"

# Run one replica of the router
oc scale dc/router-ns-dev --replicas=1

# Label a NAMESPACE to be matched by selection expression
oc label namespace myproject "router=dev"
```

USING ROUTE LABELS

```
# Create new router (no replicas)
oc adm router router-shard-2 --replicas=0

# Set the Shard selection expression
oc set env  dc/router-shard-2 ROUTE_LABELS="bizunit=prod"

# Run one replica of the router
oc scale dc/router-shard-2 --replicas=3

# Label a ROUTE to be matched by selection expression
oc label route myroute "bizunit=prod"
```

Figure 10-3 Router Sharding using Namespace or Route labels

> **NOTE** When using the *Namespace* labels, the *Service Account* assigned to the *Router* must have *cluster-reader* permission to access the labels in the *Namespaces*.

Supporting Non-HTTP/HTTPS/TLS Applications

There is a wide range of applications that cannot be classified as HTTP-, HTTPS-, or TLS-based applications. For example:

- Applications using specialized TCP protocols (i.e., database protocols)

- UDP-based applications
- Applications requiring direct access to the Pods IP

For these applications, OpenShift provides various mechanisms:

- Using IngressIP or ExternalIP
- Using NodePorts or HostPorts

USING INGRESSIP AND EXTERNALIP

When using an ingressIP and externalIP, OpenShift uses Kube-Proxy to configure all Nodes into accepting traffic destined to the particular IP address. When traffic destined for a particular ExternalIP arrives to a Node, it forwards the traffic internally to the Pods associated to the Service (see #2 and #5 on Figure 10-4).

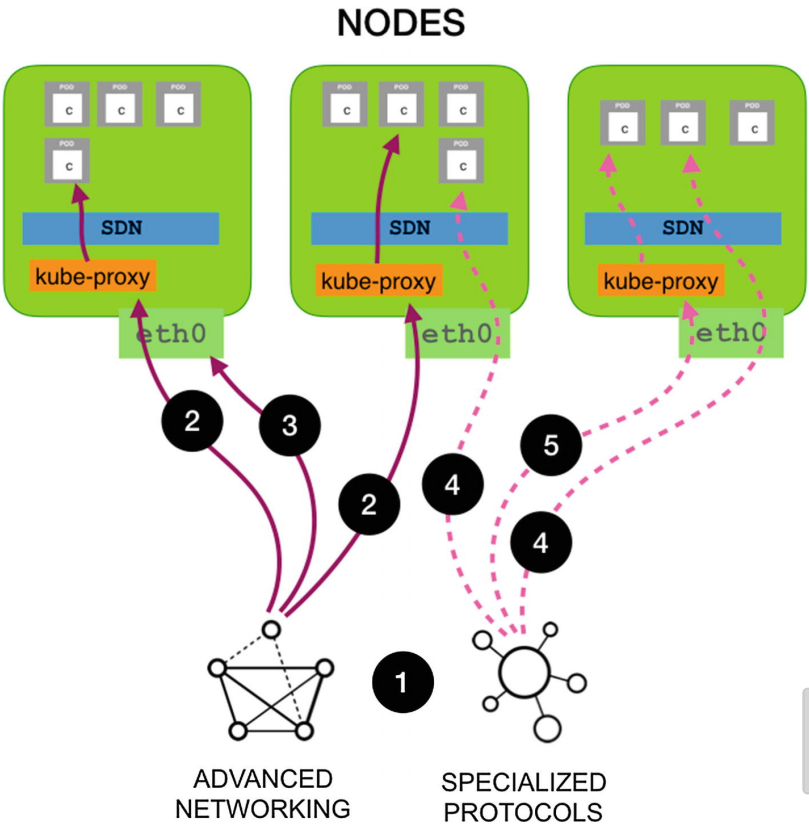

Figure 10-4 Traffic flow for non-http/https traffic

At first sight, both of these objects behave similarly but have a different default or intended purpose.

- **IngressIP**: This IP address is allocated from the *ingressIPNetworkCIDR* (default to 172.29.0.0/16 when not defined) for *Service* type *LoadBalancer*. This *CIDR* should not overlap with other IP ranges used in the Cluster.

- **ExternalIP**: This IP is allocated from a *CIDR* defined by the *externalIPNetworkCIDRs* variable in the `master-config.yaml` (see Figure 10-5). This can be a public IP address range or an organization-level visible and unique network CIDR.

> **NOTE** IP Addresses from the *externalIP* CIDR are not managed by *OpenShift*. It is up to the network administrator to make sure the traffic destined to these IP arrives to the *Nodes*.

```
vi /etc/origin/master/master-config.yaml
...
networkConfig:
  externalIPNetworkCIDRs:
  - 198.18.100.0/23
  - <network_2>/<cidr>
  - ...


# Restart Master service for changes to get into effect
$ master-restart api
$ master-restart controllers
```

Figure 10-5 Defining an externalIPNetworkCIDR

Creating a service type *LoadBalancer* (see Listing 10-1) gets an ingressIP by default (see Figure 10-6).

```
apiVersion: v1
kind: Service
metadata:
  name: pgsql-lb
spec:
  ports:
  - name: pgsql
```

```
    port: 5432
  type: LoadBalancer
  selector:
    name: pgsql
```

Listing 10-1 Creating a Service type LoadBalancer

```
apiVersion: v1
kind: Service
metadata:
  name: pgsql-lb
spec:
  ports:
  - name: pgsql
    port: 5432
  type: LoadBalancer
  selector:
    name: pgsql
```

and

```
$ oc expose dc pgsql --type=LoadBalancer --name=pgsql-lb2
```

```
# psql -h 172.29.223.161 sampledb pguser
Password for user pguser:
psql (9.2.24, server 9.6.10)
WARNING: psql version 9.2, server version 9.6.
         Some psql features might not work.
Type "help" for help.

sampledb=>
```

```
[root@ocp ~]# oc expose dc pgsql --type=LoadBalancer --name=pgsql-lb2
service/pgsql-lb2 exposed
[root@ocp ~]# oc get all
NAME                READY     STATUS    RESTARTS   AGE
pod/pgsql-1-vpm6t   1/1       Running   0          4m

NAME                          DESIRED   CURRENT   READY     AGE
replicationcontroller/pgsql-1   1         1         1         4m

NAME                                      TYPE           CLUSTER-IP       EXTERNAL-IP                     PORT(S)          AGE
service/pgsql                             ClusterIP      172.30.240.202   <none>                          5432/TCP         4m
service/pgsql-lb                          LoadBalancer   172.30.106.105   172.29.121.102,172.29.121.102   5432:30671/TCP   1m
service/pgsql-lb2                         LoadBalancer   172.30.106.248   172.29.223.161,172.29.223.161   5432:30318/TCP   4s
```

Figure 10-6 LoadBalancer Service, IngressIP, and ExternalIP

The *LoadBalancer* resource can be created using a YAML file (see #1 in Figure 10-6) or using the OpenShift client command (see #2 in Figure 10-6). The resulting LoadBalancer object will be assigned an IP from the *ingressIPNetworkCIDR*.

> **NOTE** The *CIDR* for `ingressIPNetworkCIDR` can be modified on the `master-config.yaml`.

Assigning an externalIP to a service is achieved by adding it to the `spec.externalIPs` definitions of the services (see Figure 10-7).

```
[root@ocp ~]# oc get svc
NAME                                TYPE           CLUSTER-IP       EXTERNAL-IP                     PORT(S)          AGE
glusterfs-dynamic-22a6a050-7efe-11e9-b815-001a4a160101   ClusterIP      172.30.55.81     <none>                          1/TCP            32m
pgsql                               ClusterIP      172.30.240.202   <none>                          5432/TCP         32m
pgsql-lb                            LoadBalancer   172.30.106.105   172.29.121.102,172.29.121.102   5432:30671/TCP   30m
pgsql-lb2                           LoadBalancer   172.30.106.248   172.29.223.161,172.29.223.161   5432:30318/TCP   28m
[root@ocp ~]# oc patch svc pgsql-lb2 -p '{"spec":{"externalIPs":["198.18.101.200"]}}'
service/pgsql-lb2 patched
[root@ocp ~]# oc get svc
NAME                                TYPE           CLUSTER-IP       EXTERNAL-IP                     PORT(S)          AGE
glusterfs-dynamic-22a6a050-7efe-11e9-b815-001a4a160101   ClusterIP      172.30.55.81     <none>                          1/TCP            33m
pgsql                               ClusterIP      172.30.240.202   <none>                          5432/TCP         34m
pgsql-lb                            LoadBalancer   172.30.106.105   172.29.121.102,172.29.121.102   5432:30671/TCP   31m
pgsql-lb2                           LoadBalancer   172.30.106.248   172.29.223.161,198.18.101.200   5432:30318/TCP   30m
[root@ocp ~]#
```

Figure 10-7 Assigning externalIP to a Service

USING NODEPORTS AND HOSTPORTS

Another way to bring traffic into the Pods is by using a NodePort or HostPort. These two objects are similar in their behavior with respect to allocating ports in the actual Nodes. The difference is how the Ports are allocated in all Nodes from a range or allocated on the Node where the Pod is running.

- **NodePort**: Will allocate a port from the range 30000–32767 in all Nodes. (Note: It is possible to request a specific port in this range). The NodePort can be allocated for a Service or a specific Pod (see Figure 10-8).

- **HostPort**: Will allocate the specified port in the Node where it is running (see Figure 10-9).

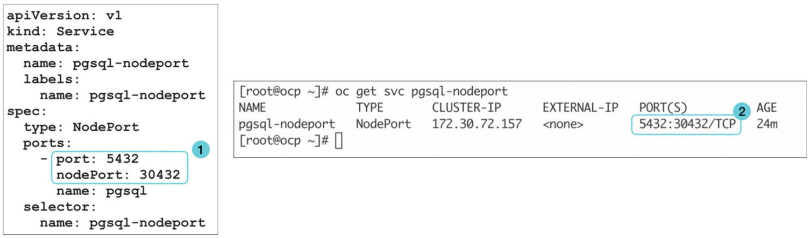

Figure 10-8 Using NodePort

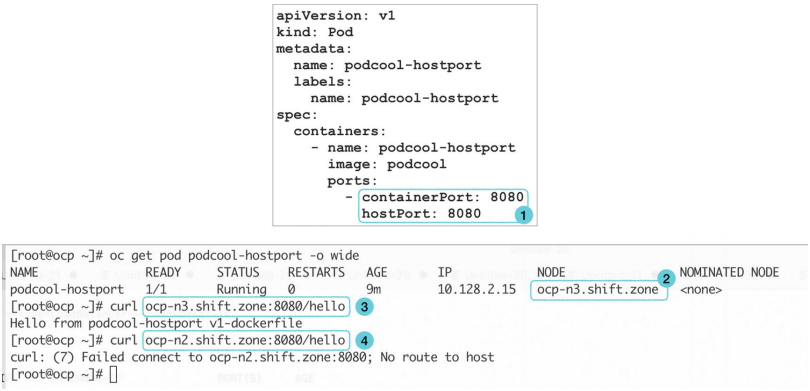

Figure 10-9 Using HostPort

Multiple NIC per POD

OpenShift 4.1 and later support the ability to provide multiple network interfaces to Pods. This capability is provided by the *Multus CNI*.[4]

Multus CNI is a meta plugin for Kubernetes which enables the creation of multiple network interfaces per *Pod*. Each interface can be using a different CNI plugin.

As seen in Figure 10-9, when *Multus CNI* receives the request for the creation of a new network interface for the *Pod*, it sends that request to the primary Kubernetes CNI (see #4 in Figure 10-10) for the creation of the `eth0` interface. In addition, it interprets the *Pod annotations* to invoke additional *CNIs* to add other interfaces (see #6 in Figure 10-10) to the Pod.

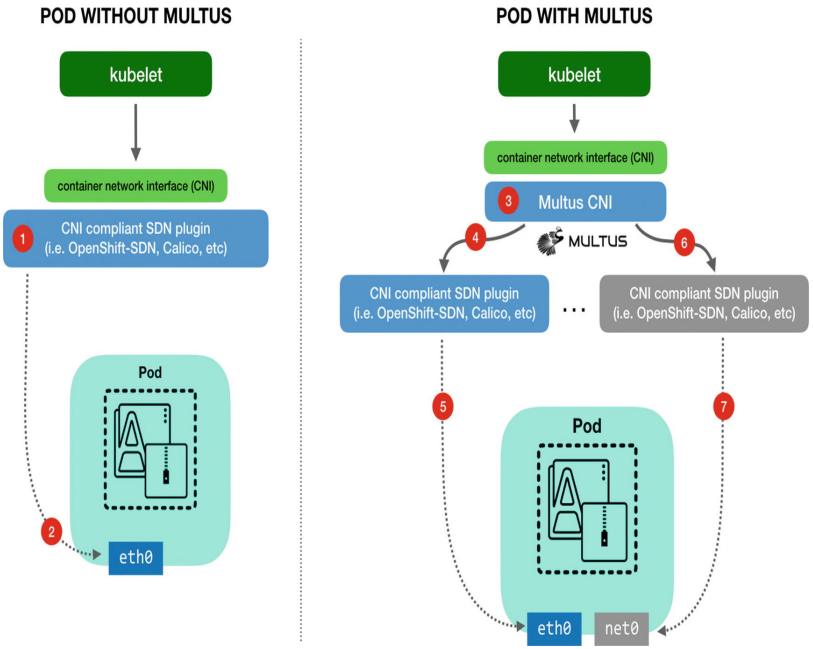

Figure 10-10 Multus CNI logical diagram

Multus requires the creation of a `NetworkAttachmentDefinition` defining the additional CNI (see #1 in Figure 10-11). The Pod must be annotated with the additional CNIs to use to provide additional interfaces (see #2 in Figure 10-11). At the Pod level, the new network interface is created (see #4 in Figure 10-11).

Figure 10-11 Defining NetworkAttachmentDefinition and using Multus CNI

OpenShift ServiceMesh

The OpenShift ServiceMesh is based on the upstream project Maistra.[5]

Some of the components of OpenShift ServiceMesh are

- **Istio**: Based on the Istio[6] project; enables the intelligent control of the flow of traffic; enables the authentication, authorization, and encryption of communication between microservices; enforces policies; and enables observability of the communication among the microservices of an application
- **Envoy**: Service proxy used by Istio and based on Envoy Proxy[7] project
- **Jaeger**: Distributed tracing capability based on the Jaeger[8] project (see #2 in Figure 10-12)

- **Kiali**[9] : Graphical interface integrating the components of OpenShift ServiceMesh (see #1 in Figure 10-12)
- **Grafana**: Used for the Istio mesh dashboards (see #3 and #4 in Figure 10-12)
- **Prometheus**: Used to collect Istio mesh metrics
- **Elasticsearch**: Used as the backend storage for the Istio metrics

> **NOTE** At the time of this writing, installing OpenShift ServiceMesh[10] in OCP 3.11.x is still considered a *Technology Preview* capability.

Some of the OpenShift ServiceMesh consoles are shown in Figure 10-12.

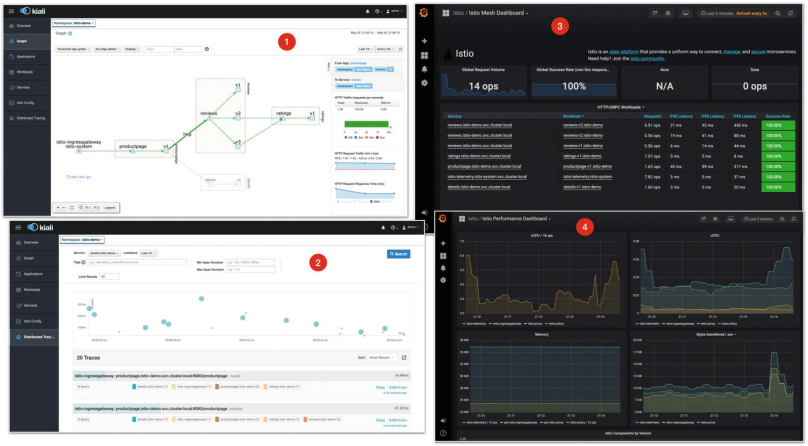

Figure 10-12 OpenShift ServiceMesh

The main functionality of the OpenShift ServiceMesh requires injecting the Istio sidecar, the Envoy proxy, into the Pod. This requires the proper annotation of the *Deployment* configuration (see #1 in Figure 10-13).

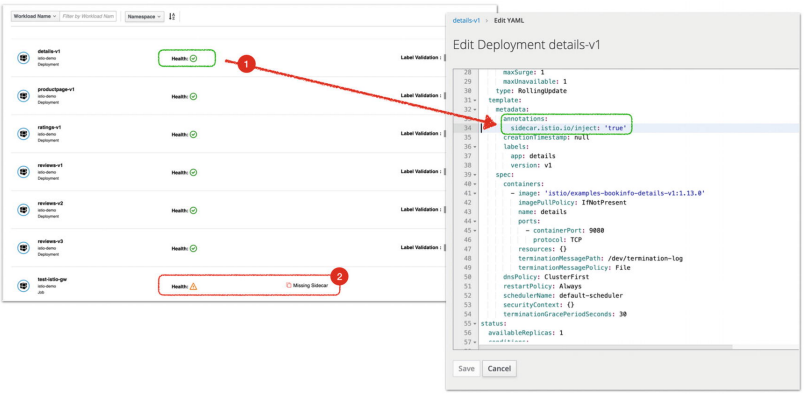

Figure 10-13 OpenShift ServiceMesh annotations for Istio sidecar

Once the Istio proxy sidecar is injected into the Pod (see #5 in Figure 10-14), all traffic incoming or outgoing to that Pod goes over the Istio-proxy sidecar container.

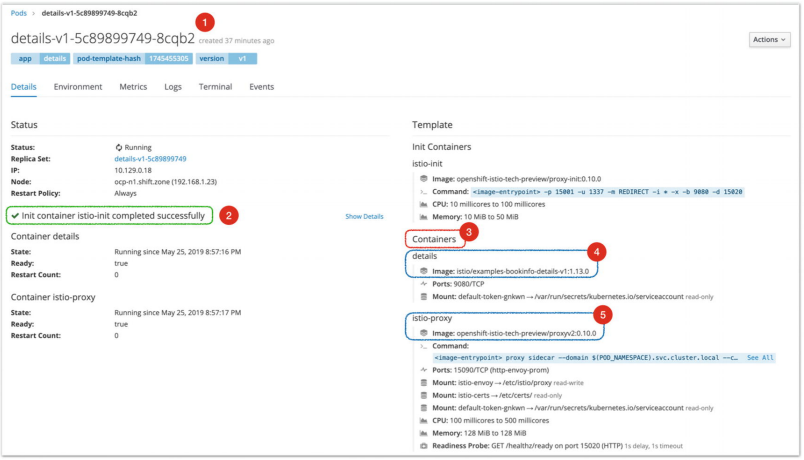

Figure 10-14 OpenShift ServiceMesh Istio-Proxy sidecar injection

From the *OpenShift ServiceMesh* perspective, when using *OpenShift Routes* in conjunction with the *Istio Gateway* resources, the traffic flow will be as follows:

- External traffic arrives to the Route (see #1 in Figure 10-15) which points to a *LoadBalancer* type *Service* (see #2 in Figure 10-15).
    - A *LoadBalancer* resource gets allocated an IngressIP and the cluster administrator could also assign an ExternalIP.

- Traffic is then delivered to the *Istio Gateway* on the destination *Project* or *Namespace* as seen in #3 on Figure 10-15.
  - The *Istio Gateway* is considered the edge of the *Mesh* for incoming and outgoing connections. It describes the ports and protocol (HTTP/HTTPS/TCP) it will accept traffic for.
- Traffic accepted at the *Istio Gateway* is forwarded based on the *VirtualService* definition (see #5 on Figure 10-15).
  - The *VirtualService* defines one or more destinations where the traffic should go inside the *ServiceMesh* to reach the actual destination (i.e., a *Service* or *Pod*).
- Any *Service* or *Pod* annotated for the *Istio Proxy* will have the Istio sidecar injected into the Pods (see yellow pentagon shapes in Figure 10-15).
- After this point, the metrics and visibility provided by the *Istio Proxy* are available over the *Kiali* console.

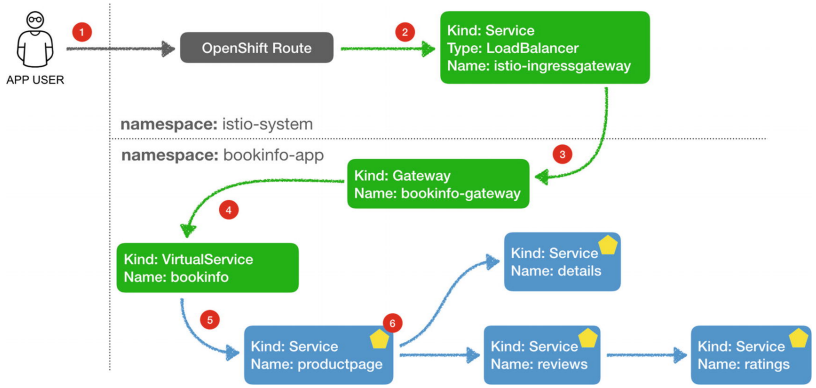

Figure 10-15 Traffic flow with OpenShift ServiceMesh

> **CAUTION** At the moment of this writing, the Istio-Proxy (Envoy) has limited support for non-TCP traffic. Applications relying on non-TCP protocols should investigate the impact of these limitations to avoid service disruption.

Summary

This chapter covers some of the advanced network optimizations available in the OpenShift Container Platform (OCP). Some of these optimizations, like hardware acceleration, are dependent on the availability of underlying infrastructure supporting the capability. Other optimizations are more in the fine-tuning of configuration attributes to increase performance and scalability of the capability, like the optimizations available for the *OpenShift Routers*.

In addition to the optimization, this chapter describes the use of IngressIP, ExternalIPs, NodePorts, and HostPorts to bring specialized IP protocols to Services and Pods running on the platform.

Finally, the chapter explored advanced functionalities provided by OpenShift Multi-Network capabilities with Multus and the OpenShift ServiceMesh with Istio, Jaeger, and other upstream projects.

Some of the optimizations described in this chapter are intended for OpenShift bare-metal deployments. Chapter 11 provides a glimpse of the installation of OpenShift 4.1 using the User Provisioned Infrastructure (UPI) deployment option. This new OpenShift version provides the support for advanced networking capabilities like the multiple network for Containers using Multus, OpenShift ServiceMesh, and many others.

Footnotes

1 Additional information about configuring the maximum number of connection is available at the online documentation: https://docs.openshift.com/container-platform/3.11/scaling_performance/routing_optimization.html#scaling-performance-optimizing-router-haproxy-maxconn

2 OpenShift HAProxy optimization parameters: https://docs.openshift.com/container-platform/3.11/scaling_performance/routing_optimization.html#scaling-performance-optimizing-router-haproxy

3 Additional information and the updated list of possible Route Annotations are available from the online documentation: https://docs.openshift.com/container-platform/3.11/architecture/networking/routes.html#route-specific-annotations

4 Additional information on providing multiple network interface to Pods can be found at the OCP 4.1 online documentation: https://docs.openshift.com/container-platform/4.1/networking/managing-multinetworking.html

5 For more details of the Maistra project, refer to the online documentation at https://maistra.io/docs/

6 Upstream Istio project is available at https://istio.io

7 Upstream Envoy Proxy project is available at www.envoyproxy.io (http://www.envoyproxy.io)

8 Upstream Jaeger project is available at www.jaegertracing.io (http://www.jaegertracing.io)

9 Upstream Kiali project is available at www.kiali.io (http://www.kiali.io)

10 The instructions for the installation and configuration of the OpenShift Service Mesh are available at the online documentation: https://docs.openshift.com/container-platform/3.11/servicemesh-install/servicemesh-install.html#installing-service-mesh

PREV
9. Day-2 Operations

NEXT
11. OCP 4.1 UPI Mode Ba…