Kubernetes Pod and Service Networks

Objectives

- Interconnect application pods inside the same cluster by using Kubernetes services.

The Software-defined Network

Kubernetes implements software-defined networking (SDN) through Open Virtual Network (OVN)-Kubernetes to manage the network infrastructure of the cluster. OVN-Kubernetes creates a virtual network that encompasses all cluster nodes. The virtual network enables communication between any container or pod within the cluster. Cluster node processes that Kubernetes pods manage have access to the OVN-Kubernetes network. However, the OVN-Kubernetes network remains inaccessible from outside the cluster or by regular processes on cluster nodes.

With the OVN-Kubernetes model, you can manage network services through the abstraction of multiple networking layers. With OVN-Kubernetes, you can manage network traffic and network resources programmatically to enable organization teams to determine how to expose their application.

The OVN-Kubernetes implementation creates a model that is compatible with traditional networking practices. The model makes pods that are similar to virtual machines in terms of port allocation, IP address leasing, and reservation.

With the OVN-Kubernetes design, you do not need to modify how application components communicate with each other. The OVN-Kubernetes design assists in containerizing legacy applications. If your application consists of multiple services that communicate over the TCP/UDP stack, then this approach remains effective because containers within a pod use the same network stack.

The following diagram illustrates how all pods connect to a shared network:

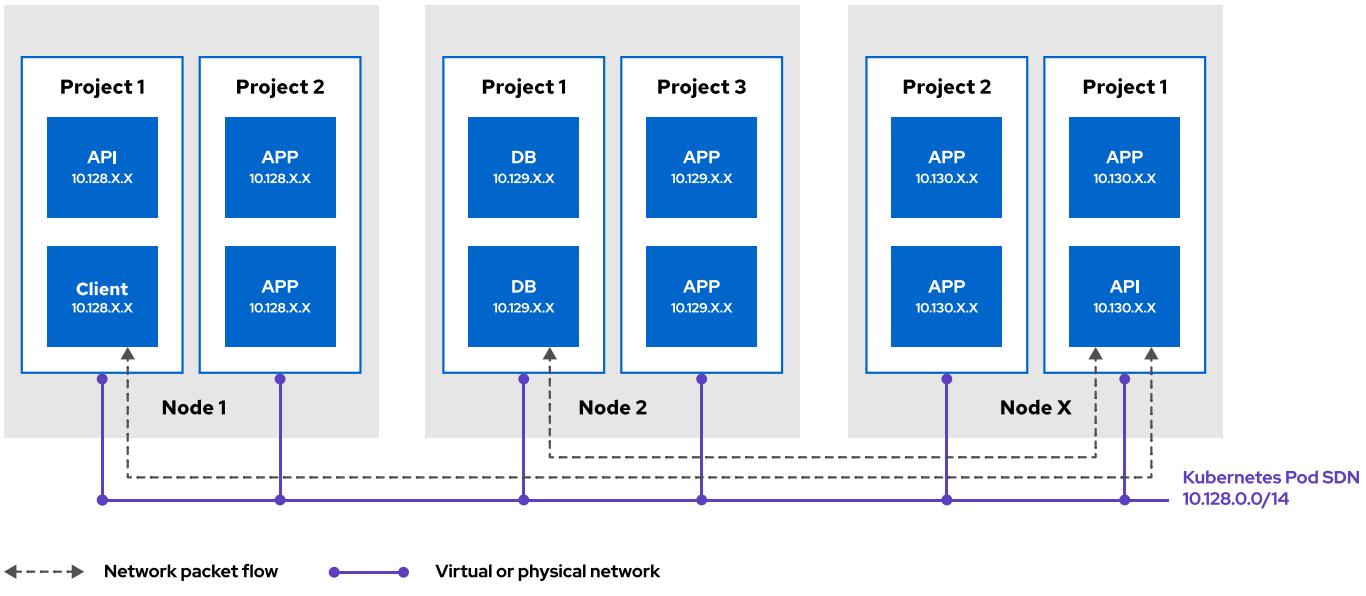

Figure 4.6: How OVN-Kubernetes manages the network

Among the features of OVN-Kubernetes, open standards enable vendors to propose solutions for centralized management, dynamic routing, and tenant isolation.

Kubernetes Networking

Networking in Kubernetes provides a scalable method for communication between containers and offers the following capabilities:

- Highly coupled container-to-container communications
- Pod-to-pod communications
- Pod-to-service communications
- External-to-service communication: Covered in the section called " Scale and Expose Applications to External Access "

Kubernetes automatically assigns an IP address to each pod. However, pod IP addresses remain unstable because pods are ephemeral. Pods continuously create and delete themselves across the nodes in the cluster. For example, when you deploy a new version of your application, Kubernetes deletes the existing pods and then deploys new ones.

All containers within a pod share networking resources. The IP address and MAC address that are assigned to the pod are shared among all containers in the pod. Thus, all containers within a pod can access each other's ports via the loopback address, which is `localhost`. Ports that are bound to `localhost` are available to all containers that are running within the pod but not to containers outside it.

By default, pods can communicate with each other even if they run on different cluster nodes or belong to different Kubernetes namespaces. Every pod is assigned an IP address in a flat shared networking namespace that has full communication with other physical computers and containers across the network. All pods are assigned a unique IP address from a Classless Inter-Domain Routing (CIDR) range of host addresses. The shared address range places all pods in the same subnet.

Because all the pods are on the same subnet, pods on all nodes can communicate with pods on any other node without the aid of Network Address Translation (NAT). Kubernetes also provides a service subnet, which links the stable IP address of a service resource to a set of specified pods. The traffic is forwarded in a transparent way to the pods; an agent, which OVN-Kubernetes manages, handles routing rules to direct traffic to pods that match the service resource selectors. Thus, pods function similarly to virtual machines or to physical hosts regarding port allocation, networking, naming, service discovery, load balancing, application configuration, and migration.

The following illustration gives further insight into how the infrastructure components work along with the pod and service subnets to enable network access between pods inside an OpenShift instance.

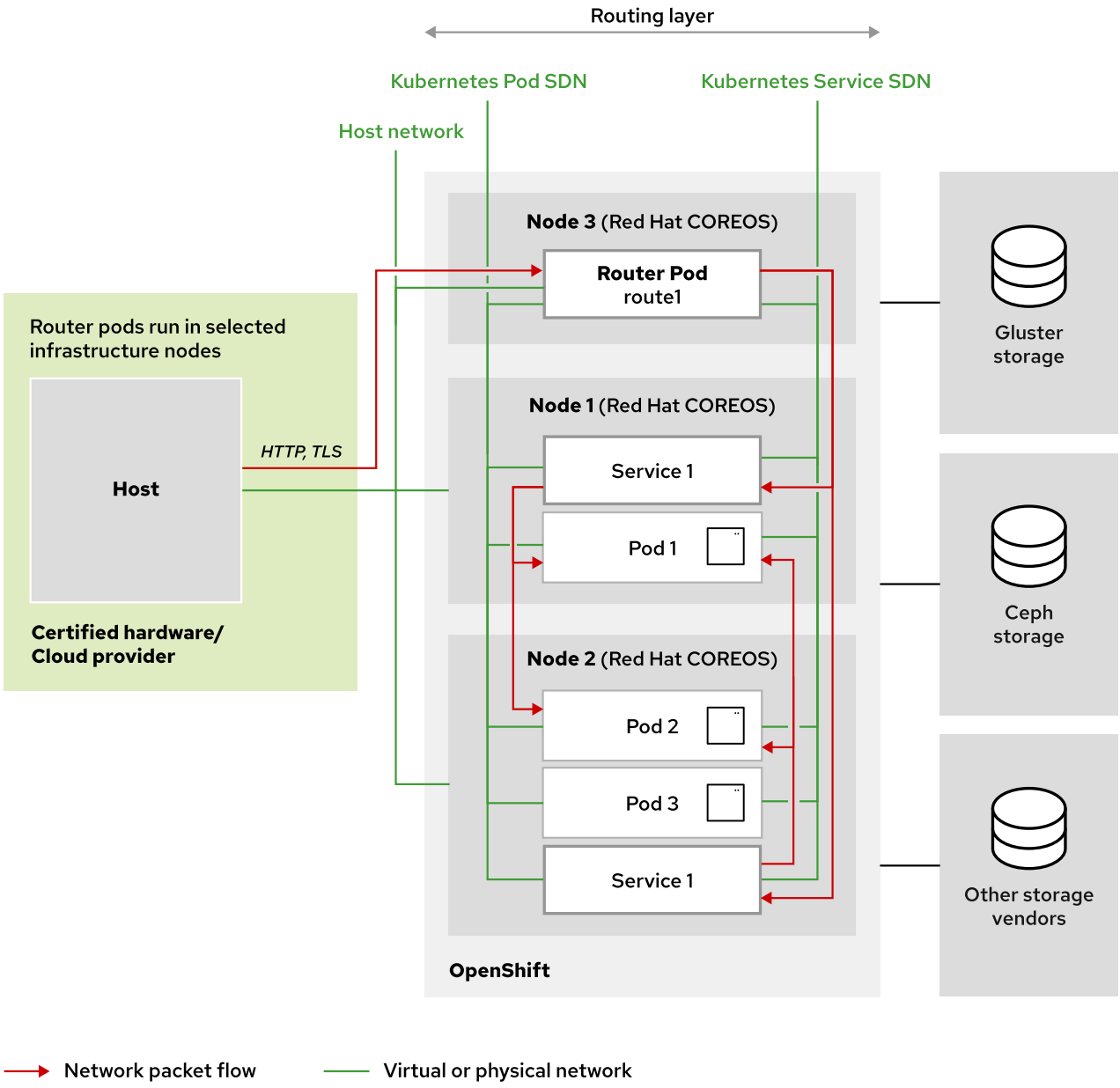

Figure 4.7: Network access between pods in a cluster

The shared networking namespace of pods enables a straightforward communication model. However, the dynamic nature of pods presents a problem. Pods can be added on the fly to handle increased traffic. Likewise, pods can be dynamically scaled down. If a pod fails, then Kubernetes automatically replaces the pod with a new one. These events change pod IP addresses.

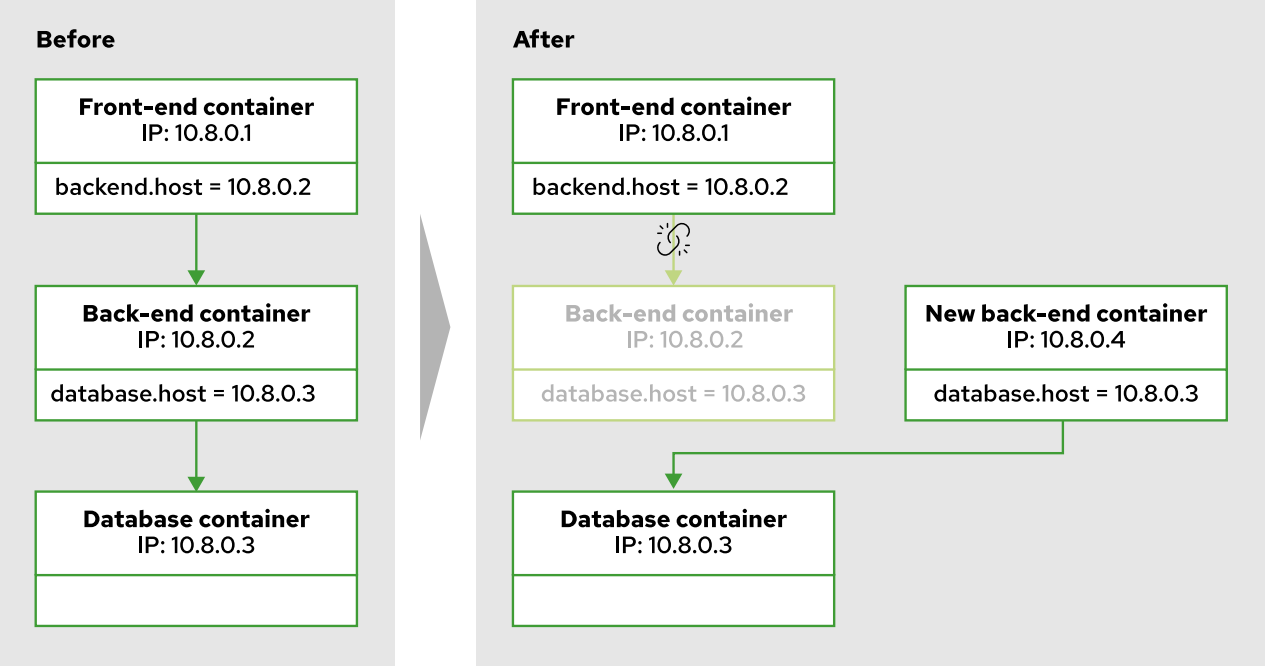

Figure 4.8: Problem with direct access to pods

In the diagram, the `Before` side shows a `Front-end` container that is running in a pod with a `10.8.0.1` IP address. The container also shows a `Back-end` container that is running in a pod with a `10.8.0.2` IP address. In this example, an event occurs that causes the `Back-end` container to fail. A pod can fail for many reasons. In response to the failure, Kubernetes creates a pod for the `Back-end` container that uses a new IP address of `10.8.0.4`.

From the `After` side of the diagram, the `Front-end` container now has an invalid reference to the `Back-end` container because of the IP address change. Kubernetes resolves this problem with `service` resources.

Using Services

Containers inside Kubernetes pods must not connect directly to each other's dynamic IP address. Instead, Kubernetes assigns a stable IP address to a service resource that is linked to a set of specified pods. The service acts as a virtual network load balancer for the pods that are linked to it.

If pods are restarted, replicated, or rescheduled to different nodes, then the service endpoints are updated. The updated endpoints provide scalability and fault tolerance for your application. Unlike the IP addresses of pods, the IP addresses of services do not change. Service IP stability provides scalability and fault tolerance for your application.

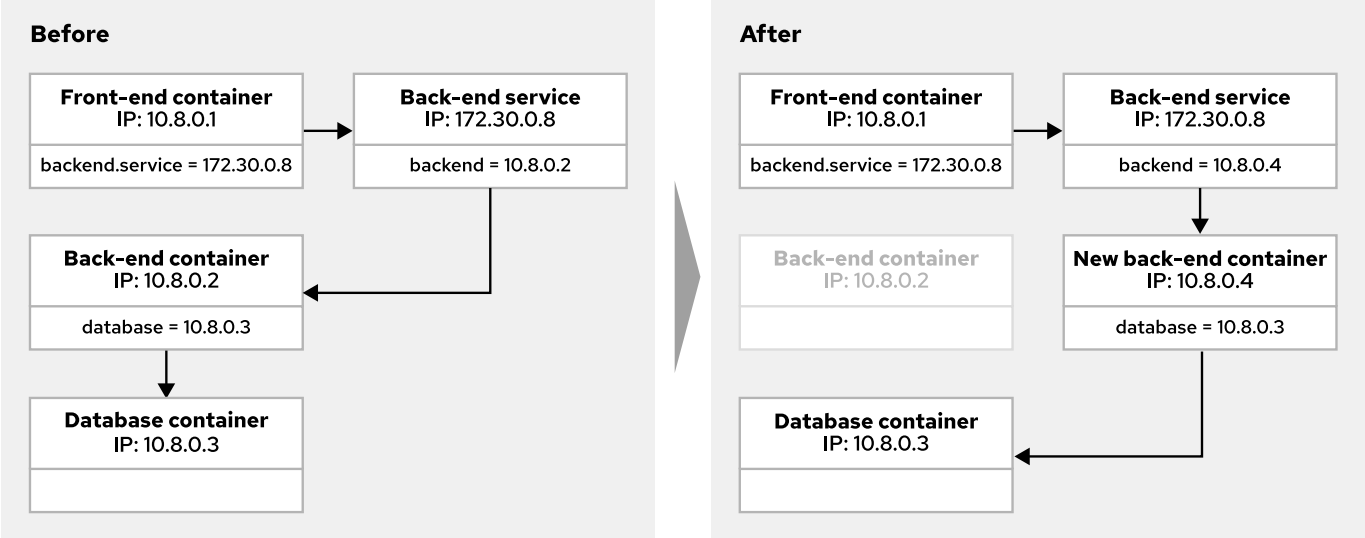

Figure 4.9: Services resolve pod failure issues

In the diagram, the `Before` side shows that a `Front-end` container references the stable IP address of the `Back-end` service, instead of to the IP address of the pod that is running the `Back-end` container. When the `Back-end` container fails, Kubernetes creates a pod with the `New back-end` container to replace the failed pod. In response to the change, Kubernetes removes the failed pod from the service endpoints. Kubernetes then adds the IP address of the `New back-end` container pod to the service endpoints.

With the service in place, requests from the `Front-end` container to the `Back-end` container continue to function. The service updates dynamically with the IP address change. A service provides a permanent, static IP address for a group of pods that belong to the same deployment or replica set for an application. Until you delete the service, the assigned IP address does not change, and the cluster does not reuse it.

Most real-world applications do not run as a single pod. Applications need to scale horizontally. Multiple pods run the same containers to meet growing user demand. A Deployment resource manages multiple pods that execute the same container. A service provides a single IP address for the entire set. The service performs load balancing for client requests among member pods.

With services, containers in one pod can establish network connections to containers in another pod. The pods that the service tracks do not need to exist on the same compute node or in the same namespace or project. Because a service provides a stable IP address for other pods to use, a pod does not need to discover the new IP address of another pod after a restart. The service provides a stable IP address to use, no matter which compute node runs the pod after each restart.

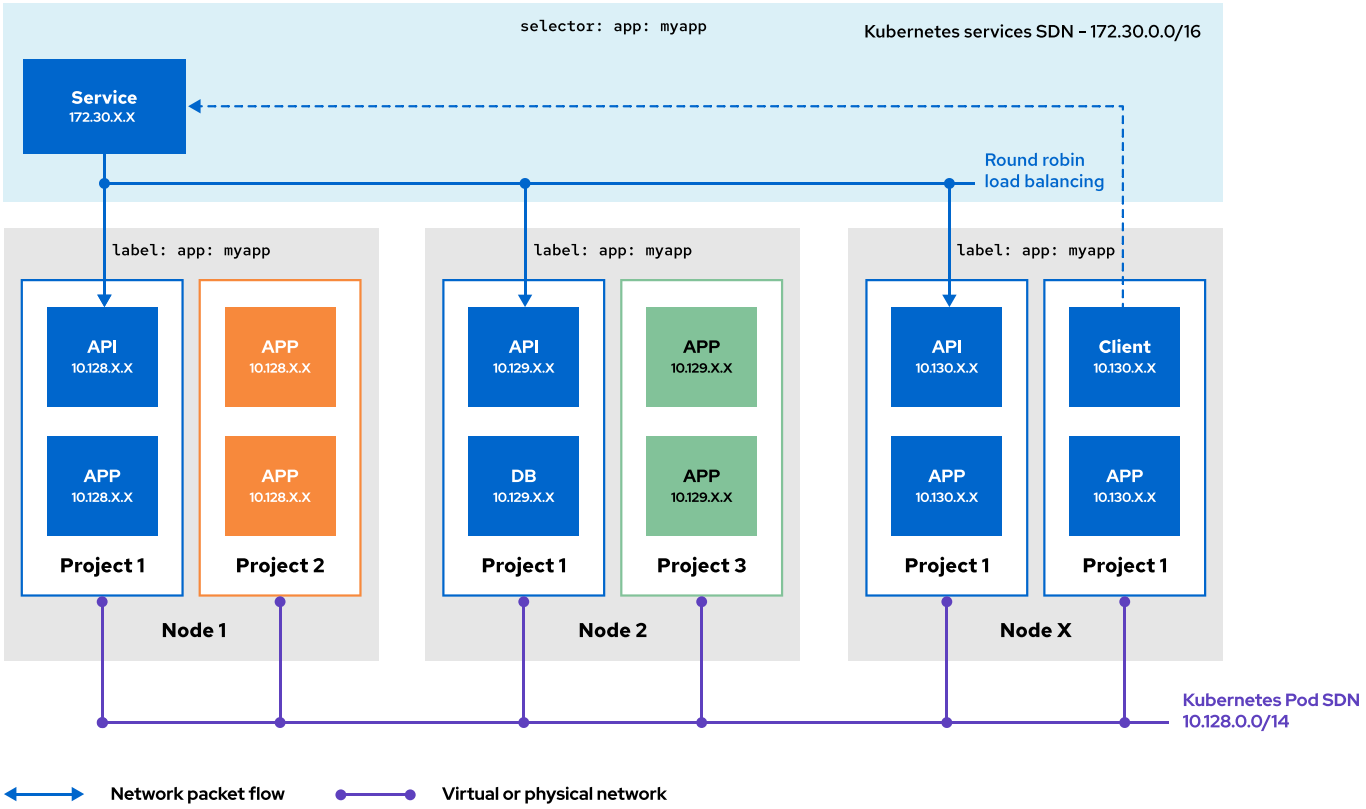

Figure 4.10: Service with pods on many nodes

The *Service* object provides a stable IP address for the *Client* container on *Node X*. The stable IP address enables the *Client* container to send a request to any of the *API* containers.

Kubernetes uses labels on pods to select those pods that are associated with a service. To include a pod in a service, the pod labels must include each of the `selector` fields of the service.

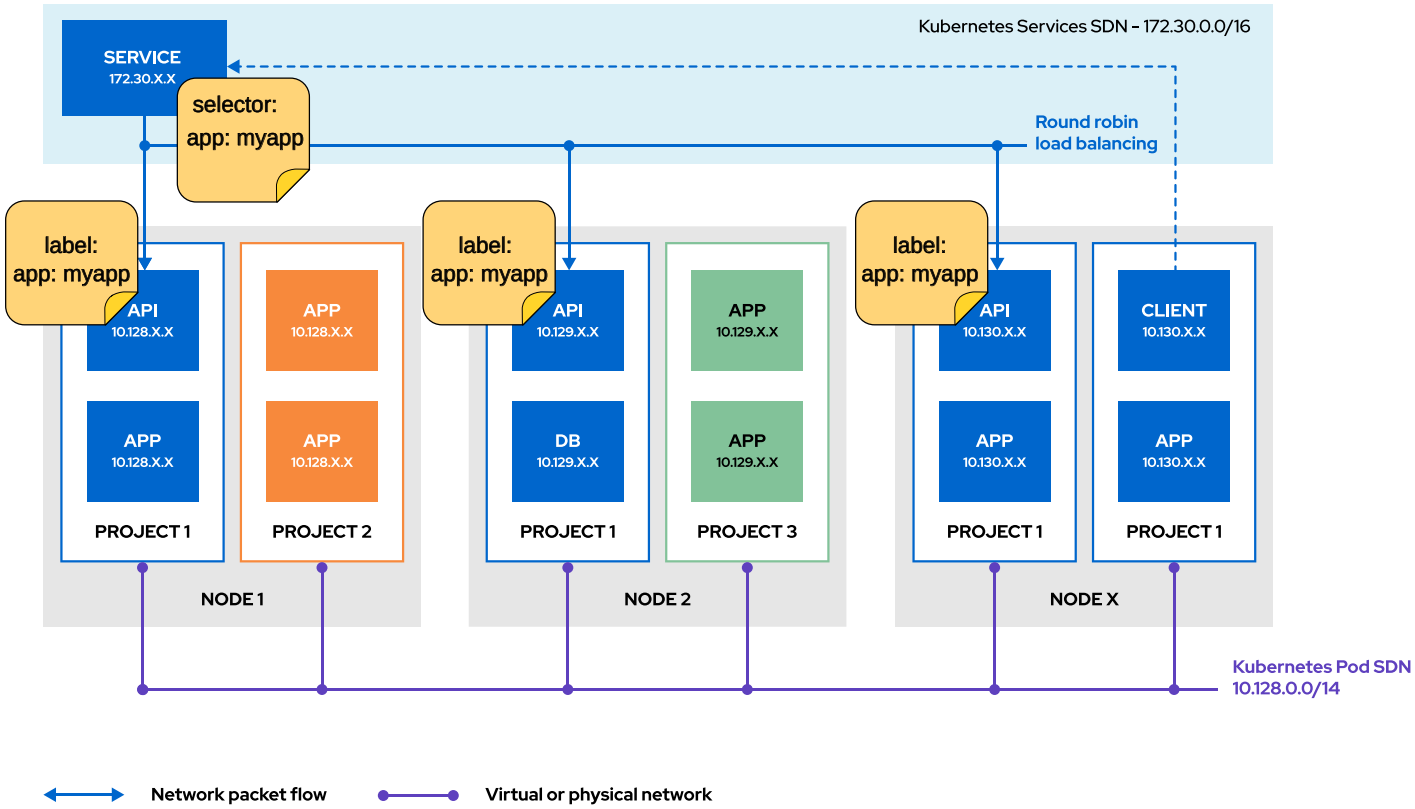

Figure 4.11: Service selector match to pod labels

In this example, the selector has a key-value pair of `app: myapp`. Thus, pods with a matching label of `app: myapp` are included in the set that is associated with the service. The *selector* attribute of a service identifies the set of pods that form the endpoints for the service. Each pod in the set is an endpoint for the service.

To create a service for a deployment, use the `oc expose` command:

```
[user@host ~]$ oc expose deployment/<deployment-name> [--selector <selector>]
[--port <port>][--target-port <target port>][--protocol <protocol>][--name <name>]
```

The `oc expose` command uses the `--selector` option to specify label selectors. When you omit the `--selector` option, the command applies a selector that matches the replication controller or the replica set.

The `--port` option specifies the port that the service listens on. This port is available only to pods within the cluster. If a port value is not provided, then the port is copied from the configuration of the deployment.

The `--target-port` option specifies the name or number of the container port that the service uses to communicate with the pods. If you do not provide a port value, then the port is copied from the configuration of the deployment.

The `--protocol` option determines the network protocol for the service. TCP is used by default.

The `--name` option explicitly names the service. If you do not specify a name, then the service uses the same name as the deployment.

To view the selector that a service uses, use the `-o wide` option with the `oc get` command.

```
[user@host ~]$ oc get service db-pod -o wide
NAME    TYPE        CLUSTER-IP      EXTERNAL-IP PORT(S)     AGE     SELECTOR
db-pod  ClusterIP   172.30.108.92   <none>      3306/TCP    108s    app=db-pod
```

In this example, `db-pod` is the name of the service. Pods must use the `app=db-pod` label to be included in the host list for the `db-pod` service. To see the endpoints that a service uses, use the `oc get endpoints` command.

```
[user@host ~]$ oc get endpoints
NAME     ENDPOINTS                         AGE
db-pod   10.8.0.86:3306,10.8.0.88:3306   27s
```

This example illustrates a service with two pods in the host list. The `oc get endpoints` command returns the service endpoints in the current selected project. Add the name of the service to the command to show only the endpoints of a single service. Use the `--namepace` option to view the endpoints in a different namespace.

Use the `oc describe deployment <deployment name>` command to view the deployment selector.

```
[user@host ~]$ oc describe deployment db-pod
Name:                   db-pod
Namespace:              deploy-services
CreationTimestamp:      Wed, 18 Jan 2023 17:46:03 -0500
Labels:                 app=db-pod
Annotations:            deployment.kubernetes.io/revision: 2
Selector:               app=db-pod
...output omitted...
```

You can view or parse the selector from the `YAML` or `JSON` output for the deployment resource from the `spec.selector.matchLabels` object. In this example, the `-o yaml` option of the `oc get` command returns the `selector` label that the deployment uses.

```
[user@host ~]$ oc get deployment/<deployment_name> -o yaml
...output ommitted...
  selector:
    matchLabels:
      app: db-pod
...output ommitted...
```

Kubernetes DNS for Service Discovery

Kubernetes uses an internal Domain Name System (DNS) server that the DNS operator deploys. The DNS operator creates a default cluster DNS name, and assigns DNS names to services that you define. The DNS operator implements the DNS API from the `operator.openshift.io` API group. The operator deploys CoreDNS, creates a service resource for CoreDNS, and configures the `kubelet` component to instruct pods to use the

CoreDNS service IP address for name resolution. When a service does not have a cluster IP address, the DNS operator assigns a DNS record that resolves to the set of IP addresses of the pods behind the service.

The DNS server discovers a service from a pod by using the internal DNS server, which is visible only to pods. Each service is dynamically assigned a *Fully Qualified Domain Name* (FQDN) that uses the following format:

```
SVC-NAME.PROJECT-NAME.svc.CLUSTER-DOMAIN
```

When a pod is created, Kubernetes provides the container with a `/etc/resolv.conf` file with similar contents to the following items:

```
[user@host ~]$ cat /etc/resolv.conf
nameserver 172.30.0.10
search deploy-services.svc.cluster.local svc.cluster.local cluster.local
options ndots:5
```

In this example, `deploy-services` is the project name for the pod, and `cluster.local` is the cluster domain.

The `nameserver` directive provides the IP address of the Kubernetes internal DNS server. The `options ndots` directive specifies the number of dots that must appear in a name to qualify for an initial absolute query. Alternative hostname values are derived by appending values from the `search` directive to the name that you send to the DNS server.

In the `search` directive in this example, the `svc.cluster.local` entry enables any pod to communicate with another pod in the same cluster by using the service name and project name:

```
SVC-NAME.PROJECT-NAME
```

The first entry in the `search` directive enables a pod to use the service name to specify another pod in the same project. In Red Hat OpenShift Container Platform (RHOCP), a project is also the namespace for the pod. The service name alone is sufficient for pods in the same RHOCP project:

```
SVC-NAME
```

Kubernetes Networking Drivers

Container Network Interface (CNI) plug-ins provide a common interface between the network provider and the container runtime. CNI defines the specifications for plug-ins that configure network interfaces inside containers. Plug-ins that are written to the specification enable different network providers to control the RHOCP cluster network.

Red Hat provides the following CNI plug-ins for a RHOCP cluster:

- OVN-Kubernetes: The default plug-in for first-time installations of RHOCP 4.10 and later versions.
- OpenShift SDN: An earlier plug-in from RHOCP 3.x; it is incompatible with some later features of RHOCP 4.x.

Certified CNI-plugins from other vendors are also compatible with an RHOCP cluster.

The SDN uses CNI plug-ins to create Linux namespaces to partition the usage of resources and processes on physical and virtual hosts. With this implementation, containers inside pods can share network resources, such as devices, IP stacks, firewall rules, and routing tables. The SDN allocates a unique routable IP to each pod, so that you can access the pod from any other service in the same network.

In OpenShift 4.18, OVN-Kubernetes serves as the default network provider and OpenShift SDN is deprecated.

OVN-Kubernetes uses OVN to manage the cluster network. A cluster that uses the OVN-Kubernetes plug-in also runs Open vSwitch (OVS) on each node. OVN configures OVS on each node to implement the declared network configuration.

The OpenShift Cluster Network Operator

RHOCP provides a Cluster Network Operator (CNO) that configures OpenShift cluster networking. The CNO serves as an OpenShift cluster operator that loads and configures CNI plug-ins. As a cluster administrator, execute the following command to observe the status of the CNO:

```
[user@host ~]$ oc get -n openshift-network-operator deployment/network-operator
NAME              READY   UP-TO-DATE  AVAILABLE   AGE
network-operator  1/1     1           1           41d
```

An administrator configures the CNO at installation time. To view the configuration, use the following command:

```
[user@host ~]$ oc describe network.config/cluster
Name: cluster
...output omitted...
Spec:
Cluster Network:
Cidr: 10.128.0.0/14 (1)
Host Prefix: 23
External IP:
Policy:
Network Type: OVNKubernetes
Service Network:
172.30.0.0/16 (2)
...output omitted...
```

(1) The Cluster Network CIDR defines the range of IP addresses for all pods in the cluster.

2. The Service Network CIDR defines the range of IP addresses for all services in the cluster.

REFERENCES

For more information, refer to the *Understanding Networking* chapter in the Red Hat OpenShift Container Platform 4.18 *Networking* documentation at https://docs.redhat.com/en/documentation/openshift_container_platform/4.18/html/networking_overview/understanding-networking

For more information, refer to the *Cluster Network Operator in OpenShift Container Platform* chapter in the Red Hat OpenShift Container Platform 4.18 *Networking* documentation at https://docs.redhat.com/en/documentation/openshift_container_platform/4.18/html/networking_operators/cluster-network-operator

For more information, refer to the *About the OVN-Kubernetes Network Plug-in* chapter in the Red Hat OpenShift Container Platform 4.18 *Networking* documentation at https://docs.redhat.com/en/documentation/openshift_container_platform/4.18/html/ovn-kubernetes_network_plugin/index

Cluster Networking