Red Hat
Training and
Certification

# <Subject for cover goes here>

Travis Michette

# Table of Contents

# Chapter 1. Software Defined Network Concepts

## 1.1. Defining Software Defined Networks

This section presents the reasoning why OpenShift Container Platform depends on Software-Defined Networks and what they are.

### 1.1.1. Why using Software-Defined Networks in a cloud environment?

Software-defined networks enable dynamic and programmatic network configuration in order to improve network performance and monitoring.

It is a different approach than addressing static architecture of traditional networks and may be employed to centralize network intelligence in one network component by disassociating the forwarding process of network packets (data plane) from the routing process (control plane).

It relies on a OpenFlow protocol to manage network devices and all packages transported by them.

#### 1.1.1.1. SDN Control Plane

The implementation of a SDN control plane might be centralized, hierarchical, or distributed.

A centralized solution, where a single control entity has a global view of the network, simplifies the implementation of the control logic, it has scalability limitations as the size and dynamics of the network increase.

A hierarchical solution distributes controllers to operate on a partitioned network view, while decisions that require network-wide knowledge are taken by a logically centralized root controller.

In distributed approaches, controllers operate on their local view or they may exchange synchronization messages to enhance their knowledge. Distributed solutions are more suitable for supporting adaptive SDN applications.

#### 1.1.1.2. SDN Data Plane

The data plane is responsible for processing data-carrying packets using a set of rules specified by the control plane. The data plane may be implemented in physical hardware switches or in software implementations, such as Open vSwitch. The memory capacity of hardware switches may limit the number of rules that can be stored where as software implementations may have higher capacity.

### 1.1.2. OpenShift and the software defined network

OpenShift supports multiple SDN plugins to support specific use cases.

If not explicitely defined, OpenShift deploys the OVN-Kubernetes SDN plugin. Alternatively, it also provides OpenShift SDN plugin that was used in previous versions of OpenShift and it provides backward compatibility to old clusters.

> These plugins must be configured during the installation of OpenShift! No changes can be done afterwards.

On the top of these SDN implementations, there are other extensions that supports internal network operations like:

1. DNS server (OpenDNS)
2. Load balancers in a bare metal environment (MetalLB)
3. Encrypted web traffic (Ingress Controller).

If you need to address multiple SDN plugins, you need to use Multus plugin.

Among others that are described in this link:

https://access.redhat.com/documentation/en-us/openshift_container_platform/4.12/html/networking/networking-operators-overview

Comparing two major SDN implementations used in OpenShift:

| Feature | OVN-Kubernetes | OpenShift SDN |
|---|---|---|
| Egress IPs | Supported | Supported |
| Egress firewall | Supported | Supported |
| Egress router | Supported | Supported |
| IPsec encryption | Supported | Not supported |
| IPv6 | Supported | Not supported |
| Kubernetes network policy | Supported | Partially supported |
| Kubernetes network policy logs | Supported | Not supported |
| Multicast | Supported | Supported |

However OVN-Kubernetes has some limitations:

1. In a dual stack network (IPv4/IPv6) must use the same network device as their gateway.
2. As a side effect, both network stacks must configure routing tables with the default gateway.
3. Session stickyness uses the last request as the initial moment to timeout stickyness.

## 1.2. Workshop: Customizing OVN-Kubernetes network

*Example 1.* **Exercise: Hands-On Inspecting network configuration on pods and services**

*Listing 1. Log into the OpenShift cluster*

```
[student@workstation ~]$ oc login -u admin -p redhat
https://api.ocp4.example.com:6443
Login successful.

You have access to 70 projects, the list has been suppressed. You can list all
projects with 'oc projects'

Using project "default"
```

*Listing 2. Connect to the `openshift-network-operator` project*

```
[student@workstation ~]$ oc project openshift-network-operator
Using project "openshift-network-operator" project on server
"https://api.ocp4.example.com:6443"
```

*Listing 3. Evaluating network configuration*

```
[student@workstation ~]$ oc get network/cluster -o yaml
apiVersion: config.openshift.io/v1
kind: Network
metadata:
  creationTimestamp: "2023-05-03T18:07:31Z"
  generation: 2
  name: cluster
  resourceVersion: "2938"
  uid: 824a923d-ea68-4284-8919-958a3455b49e
spec:
  clusterNetwork:
  - cidr: 10.8.0.0/14
    hostPrefix: 23
  externalIP:
    policy: {}
  networkType: OVNKubernetes
  serviceNetwork:
  - 172.30.0.0/16
status:
  clusterNetwork:
  - cidr: 10.8.0.0/14
    hostPrefix: 23
  clusterNetworkMTU: 1400
```

```
  networkType: OVNKubernetes
  serviceNetwork:
  - 172.30.0.0/16
```

The networkType field in the spec section defines that we are using OVNKubernetes as OCP network plugin. The clusterNetwork field from the spec section defines which IP addresses will be assigned to pods created by OCP THe serviceNetwork field from the spec section defines which IP addresses will be assigned to services created by OCP

*Listing 4. Checking IP addresses are assigned in your pods.*

```
[student@workstation ~]$ oc get pods -o wide -A
NAMESPACE                                           NAME
READY   STATUS      RESTARTS     AGE     IP              NODE       NOMINATED NODE
READINESS GATES
metallb-system                                      controller-77794f9b74-twbw5
2/2     Running     2            12d     10.8.0.86       master01   <none>
<none>
metallb-system                                      metallb-operator-controller-
manager-547ff8dd4-hbpvl              1/1     Running     1            12d
10.8.0.84       master01   <none>           <none>
metallb-system                                      metallb-operator-webhook-server-
85d58867dc-bb6wl                  1/1     Running     1            12d     10.8.0.85
master01   <none>           <none>
```

There are some pods that must be exposed to external IP addresses as they need to communicate with each other to support

Note the IP address for the vast majority of these pods are within the 10.8.0.0 network, as defined in the previous configuration.

None of these pods can be accessed using any external source as they are a software defined network that only pods can connect.

*Listing 5. Checking IP addresses are assigned in your services.*

```
[student@workstation ~]$ oc get svc -o wide -A
NAMESPACE                                           NAME
TYPE           CLUSTER-IP       EXTERNAL-IP                                   PORT(S)
AGE   SELECTOR
default                                             kubernetes
ClusterIP      172.30.0.1       <none>                                        443/TCP
70d   <none>
default                                             openshift
ExternalName   <none>           kubernetes.default.svc.cluster.local   <none>
```

```
70d   <none>
kube-system                                       kubelet
ClusterIP      None             <none>
10250/TCP,10255/TCP,4194/TCP          70d   <none>
metallb-system                                    controller-monitor-service
ClusterIP      None             <none>                                9120/TCP
12d   app=metallb,component=controller
metallb-system                                    metallb-operator-controller-
manager-service   ClusterIP      172.30.40.172    <none>
443/TCP                               8h    control-plane=controller-manager
```

The field name presents the service name in the project described in the namespace column. Also notice that the CLUSTER-IP column has IP addresses defined in the serviceNetwork field mentioned previously

A pod is bound to a service using a label that matches with a selector field in the service. For example, in the `openshift-apiserver` project:

.

```
[student@workstation ~]$ oc describe pods -n openshift-apiserver | grep -A5 Labels
Labels:                 *apiserver=true*
                        app=openshift-apiserver-a
                        openshift-apiserver-anti-affinity=true
                        pod-template-hash=7c949fbc9d
                        revision=1
```

```
[student@workstation ~]$ oc describe svc -n openshift-apiserver | grep -A5 Selector
Selector:          *apiserver=true*
...
----
[NOTE]
=====
To create a service that is bound to a pod, use the `oc expose pod` command.
=====
```

In a real world scenario do not create a pod and assign it to a service. Actually, create a controller (Deployment/DeploymentConfig/DaemonSet/StatefulSet) and bound all pods created by these controllers to a service that acts as a load balancer by creating a new service with the `oc expose` command.

If you have two unrelated pods sharing the same label defined in a service selector, the service loads balance the request among these pods. This is a technique admins can use to update versions of a deployment, for instance.

*Listing 6. Open openshift-dns project*

```
[student@workstation ~]$ oc project openshift-dns
```

*Listing 7. Evaluate the DNS Configuration used by OpenShift DNS internal server*

```
[student@workstation ~]$ oc get cm/dns-default -o yaml
apiVersion: v1
data:
  Corefile: |
    .:5353 {
        bufsize 512
        errors
        log . {
            class error
        }
        health {
            lameduck 20s
        }
        ready
        kubernetes cluster.local in-addr.arpa ip6.arpa {
            pods insecure
            fallthrough in-addr.arpa ip6.arpa
        }
        prometheus 127.0.0.1:9153
        forward . /etc/resolv.conf {
            policy sequential
        }
        cache 900 {
            denial 9984 30
        }
        reload
    }
    hostname.bind:5353 {
        chaos
    }
kind: ConfigMap
metadata:
  creationTimestamp: "2023-05-03T18:27:11Z"
  labels:
    dns.operator.openshift.io/owning-dns: default
  name: dns-default
  namespace: openshift-dns
```

```
  ownerReferences:
  - apiVersion: operator.openshift.io/v1
    controller: true
    kind: DNS
    name: default
    uid: dcf60f95-1192-4b17-8c68-a67297989ca3
  resourceVersion: "7616"
  uid: d4a2878e-ae6e-498a-9c22-b1ad20708526
```

Internally there is a DNS entry that names cluster.local as the internal domain.

Pods and services in OpenShift get a DNS entry to simplify their access. If a pod or a service is available in the same project, the name of the pod or service can be used to access them.

*Listing 8. Creating a new project to explore OpenShift internal DNS server*

```
[student@workstation ~]$ oc new-project example-dns
```

*Listing 9. Deploy an application to be used to connect to another pod*

```
[student@workstation ~]$  oc create deploy hello-php --image
registry.ocp4.example.com:8443/redhattraining/php-hello-dockerfile
```

*Listing 10. Deploy an application to be used to access the previous hello world app*

```
[student@workstation ~]$  oc create deploy hello-nginx --image
registry.ocp4.example.com:8443/redhattraining/hello-world-nginx
```

Containers are supposed to be slim and some tools are not available, such as `ip` or `netstat`. You must rely on the oc command outputs.

*Listing 11. Check pods names*

```
[student@workstation ~]$ oc get pods -o wide
NAME                           READY   STATUS    RESTARTS   AGE   IP          NODE
NOMINATED NODE   READINESS GATES
hello-ngix-589864fd7d-vs5zx   1/1     Running   0          91m   10.8.0.83   master01
<none>           <none>
hello-php-567f7b5c7c-smhmq     1/1     Running   0          85m   10.8.0.87   master01
<none>           <none>
```

*Listing 12. Connect to one of the running pods*

```
[student@workstation ~]$ oc rsh hello-php-589864fd7d-vs5zx
```

```
sh-4.4$
```

*Listing 13. Access the other pod by using the curl command*

```
sh-4.4$ curl 10.8.0.83:8080
<html>
  <body>
    <h1>Hello, world from nginx!</h1>
  </body>
</html>
```

Alternatively, you can use the DNS entry provided by the DNS operator. The DNS entry is like pod-ip-address.my-namespace.pod.cluster-domain.example.

*Listing 14. Accessing using the DNS entry*

```
sh-4.4$ curl 10-8-0-83.example-dns.pod.cluster.local:8080
<html>
  <body>
    <h1>Hello, world from nginx!</h1>
  </body>
</html>
```

As the IP address is dynamically generated, it is not recommended to use the DNS entry of your pod. Instead, create a service that acts like a load balancer: .Exit from the pod

```
sh-4.4$ exit
[student@workstation ~]$
```

*Listing 15. Create a service*

```
[student@workstation ~]$ oc expose deploy/hello-php --port 8080
service/hello exposed
```

*Listing 16. Check the service name*

```
[student@workstation ~]$ oc get svc
NAME    TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
hello-php   ClusterIP   172.30.123.133   <none>    8080/TCP   29s
```

*Listing 17. Connect to the pod and check the access using the new service DNS entry*

```
[student@workstation ~]$ oc rsh hello-php-589864fd7d-vs5zx
```

```
sh-4.4$
```

*Listing 18. Access using the new service DNS entry*

```
sh-4.4$ curl hello-php:8080
<html>
  <body>
    <h1>Hello, world from nginx!</h1>
  </body>
</html>
```

*Listing 19. Creating a new project to explore OpenShift DNS resolution for different projects*

```
[student@workstation ~]$ oc new-project example-external
Now using project "example-external" on server "https://api.ocp4.example.com:6443".
```

*Listing 20. Creating an HTTPD deploy to test contents*

```
[student@workstation ~]$ oc create deploy httpd --image
registry.ocp4.example.com:8443/ubi8/httpd-24
....
```

*Listing 21. Creating a service to connect externally the pods*

```
[student@workstation ~]$ oc expose deploy/httpd --port 8080
....
```

*Listing 22. Creating a second project to explore OpenShift DNS resolution for different projects*

```
[student@workstation ~]$ oc new-project example-internal
Now using project "example-internal" on server "https://api.ocp4.example.com:6443".
...
```

*Listing 23. Creating a second pod to explore OpenShift DNS resolution for different projects*

```
[student@workstation ~]$ oc run -it cli --image
registry.ocp4.example.com:8443/openshift/origin-cli:4.12 --command -- /bin/bash
pod.apps/cli created
sh-4.4#
```

*Listing 24. Accessing the HTTPD welcome page*

```
sh-4.4# curl httpd.example-internal:8080
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
        <head>
                <title>Test Page for the HTTP Server on Red Hat Enterprise Linux</title>
```

*Listing 25. Exit from the pod*

```
sh-4.4# exit
[student@workstation ~]$
```

**Exercise: Hands-On Blocking access using NetworkPolicy**

Network policies are a broad equivalent to firewall rules. It does not provide port level restrictions but still it is a way to segregate resources in OpenShift, blocking access to pods.

*Listing 26. Creating a new project to explore OpenShift DNS resolution for different projects*

```
[student@workstation ~]$ oc new-project example-policies
Now using project "example-policies" on server "https://api.ocp4.example.com:6443".
```

*Listing 27. Creating an HTTPD deploy to test contents*

```
[student@workstation ~]$ oc create deploy httpd --image
registry.ocp4.example.com:8443/ubi8/httpd-24
....
```

*Listing 28. Creating a service to connect externally the pods*

```
[student@workstation ~]$ oc expose deploy/httpd --port 8080
....
```

*Listing 29. Creating a second pod to explore OpenShift DNS resolution for different projects*

```
[student@workstation ~]$ oc run -it cli --image
registry.ocp4.example.com:8443/openshift/origin-cli:4.12 --command -- /bin/bash
pod.apps/cli created
sh-4.4#
```

*Listing 30. Accessing the HTTPD welcome page*

```
sh-4.4# curl httpd:8080
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
        <head>
                <title>Test Page for the HTTP Server on Red Hat Enterprise Linux</title>
```

In the following NetworkPolicy definition all pods with a label `app=httpd` will be blocked if another pod tries to access it. In order to bypass the limitation it must have `access: true` label.

*Listing 31. Create and save the `network.yaml` text file as below using your preferred text editor:*

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: access-httpd
spec:
  podSelector:
    matchLabels:
      app: httpd
  ingress:
  - from:
    - podSelector:
        matchLabels:
          access: "true"
```

*Listing 32. Deploying the NetworkPolicy resource:*

```
[student@workstation ~]$ oc apply -f network.yaml
networkpolicy.networking.k8s.io/access-httpd created
```

*Listing 33. Creating a second pod to check that NetworkPolicy blocks the access to the pod*

```
[student@workstation ~]$ oc run -it cli --image
registry.ocp4.example.com:8443/openshift/origin-cli:4.12 --command -- /bin/bash
pod.apps/cli created
sh-4.4#
```

*Listing 34. Accessing the HTTPD welcome page. `curl` command line hangs because there is no response due to the NetworkPolicy rule.*

```
sh-4.4# curl httpd:8080
```

*Listing 35. Deploying a third pod to check that networkpolicy allows access to the pod due to a label:*

```
[student@workstation ~]$ oc run -it cli-allowed --labels="access=true" --image
registry.ocp4.example.com:8443/openshift/origin-cli:4.12 --command -- /bin/bash
```

```
pod.apps/cli created
sh-4.4#
```

*Listing 36. Accessing the HTTPD welcome page. `curl` command line hangs because there is no response due to the NetworkPolicy rule.*

```
sh-4.4# curl httpd:8080
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
        <head>
                <title>Test Page for the HTTP Server on Red Hat Enterprise Linux</title>
```

### 1.2.1. OVN-Kubernetes Architecture

OVN-Kubernetes architecture Introduction to OVN-Kubernetes architecture The following diagram shows the OVN-Kubernetes architecture.

OVN-Kubernetes architecture Figure 1. OVK-Kubernetes architecture The key components are:

Cloud Management System (CMS) - A platform specific client for OVN that provides a CMS specific plugin for OVN integration. The plugin translates the cloud management system's concept of the logical network configuration, stored in the CMS configuration database in a CMS-specific format, into an intermediate representation understood by OVN.

OVN Northbound database (nbdb) - Stores the logical network configuration passed by the CMS plugin.

OVN Southbound database (sbdb) - Stores the physical and logical network configuration state for OpenVswitch (OVS) system on each node, including tables that bind them.

ovn-northd - This is the intermediary client between nbdb and sbdb. It translates the logical network configuration in terms of conventional network concepts, taken from the nbdb, into logical data path flows in the sbdb below it. The container name is northd and it runs in the ovnkube-master pods.

ovn-controller - This is the OVN agent that interacts with OVS and hypervisors, for any information or update that is needed for sbdb. The ovn-controller reads logical flows from the sbdb, translates them into OpenFlow flows and sends them to the node's OVS daemon. The container name is ovn-controller and it runs in the ovnkube-node pods.

The OVN northbound database has the logical network configuration passed down to it by the cloud management system (CMS). The OVN northbound Database contains the current desired state of the network, presented as a collection of logical ports, logical switches, logical routers, and more. The ovn-northd (northd container) connects to the OVN northbound database and the OVN southbound database. It translates the logical network configuration in terms of conventional network concepts, taken from the OVN northbound Database, into logical data path flows in the OVN southbound database.

The OVN southbound database has physical and logical representations of the network and binding tables that link them together. Every node in the cluster is represented in the southbound database, and you can see the ports that are connected to it. It also contains all the logic flows, the logic flows are shared with the ovn-controller process that runs on each node and the ovn-controller turns those into OpenFlow rules to program Open vSwitch.

The Kubernetes control plane nodes each contain an ovnkube-master pod which hosts containers for the OVN northbound and southbound databases. All OVN northbound databases form a Raft cluster and all southbound databases form a separate Raft cluster. At any given time a single ovnkube-master is the leader and the other ovnkube-master pods are followers.

*Example 2.* **Exercise: Hands-On identifying OVN-Kubernetes resources**

1. Identify those pods that are part of the control plane node and compute nodes

*Listing 37. Log into the OpenShift cluster*

```
[student@workstation ~]$ oc login -u admin -p redhat
https://api.ocp4.example.com:6443
Login successful.

You have access to 70 projects, the list has been suppressed. You can list all
projects with 'oc projects'

Using project "default"
```

*Listing 38. Connect to the `openshift-ovn-kubernetes` project*

```
[student@workstation ~]$ oc project openshift-ovn-kubernetes
Using project "openshift-ovn-kubernetes" project on server
"https://api.ocp4.example.com:6443"
```

*Listing 39. List pods running on the project*

```
[student@workstation ~]$ oc get pods
[student@workstation ~]$ oc get pods
NAME                   READY   STATUS    RESTARTS   AGE
ovnkube-master-pwpqq   6/6     Running   30         70d
ovnkube-node-lgzz7     5/5     Running   25         70d
```

*Listing 40. List all containers running on a pod*

```
[student@workstation ~]$ oc get pods
[student@workstation ~]$ oc get pods
```

```
NAME                  READY   STATUS    RESTARTS   AGE
ovnkube-master-pwpqq  6/6     Running   30         70d
ovnkube-node-lgzz7    5/5     Running   25         70d
```

.. .. Image: **ubi7**

1. Run the container

### 1.2.2. OpenShift SDN configuration

OpenShift SDN is an alternative SDN that is supported by Red Hat. To enable it, you need to have it configured during OpenShift installation. Network operator configuration, broadly speaking, is the same as described in the previous sections but there are attributes that might require some customization, depending on which environment OpenShift is deployed.

To configure an OpenShift SDN during installation, add the following lines to the install-config.yaml file:

```
apiVersion: v1
baseDomain: example.com
metadata:
  name: ocp4
...
Output omitted
networking:
  clusterNetwork:
  - cidr: 10.128.0.0/14
    hostPrefix: 23
  machineNetwork:
  - cidr: 10.0.0.0/16
  networkType: OpenShiftSDN
  serviceNetwork:
  - 172.30.0.0/16
...
Output omitted
```

## 1.3. Implementing Multicast in OpenShift

To avoid huge traffic due to multicast packets, OpenShift disables it by default. However, some apps might require that multicast is enabled in a project. To support this condition, OpenShift uses a standardized configuration from Kubernetes to enable multicast.

### 1.3.1. Exercise: Implementing Multicast in OpenShift

*Listing 41. Create a project to run a multicast application*

```
[student@workstation ~]$ oc new-project multicast
```

*Listing 42. Create, using your preferred text editor the following content in a file named* `mlistener.yaml`*:*

```
apiVersion: v1
kind: Pod
metadata:
  name: mlistener
  labels:
    app: multicast-verify
spec:
  containers:
   - name: mlistener
     image: registry.access.redhat.com/ubi8
     command: ["/bin/sh", "-c"]
     args:
       ["dnf -y install socat hostname && sleep inf"]
     ports:
       - containerPort: 30102
         name: mlistener
         protocol: UDP
```

The previous code creates a pod named mlistener using as Red Hat Universal Base Container Image (ubi8), and installs a tool named socat. In a later moment, the socat command is used to capture multicast packet sent to the network.

*Listing 43. Create the pod:*

```
[student@workstation ~]$ oc create -f mlistener.yaml
```

*Listing 44. Create, using your preferred text editor the following content in a file named* `msender.yaml`*:*

```
apiVersion: v1
kind: Pod
metadata:
  name: msender
  labels:
    app: multicast-verify
spec:
  containers:
    - name: msender
      image: registry.access.redhat.com/ubi8
```

```
      command: ["/bin/sh", "-c"]
      args:
        ["dnf -y install socat && sleep inf"]
EOF
```

The previous code creates a pod named msender using as Red Hat Universal Base Container Image (ubi8), and installs a tool named socat. In a later moment, the socat command is used to send multicast packet.

Create the pod:

```
[student@workstation ~]$ oc create -f msender.yaml
```

Start the listener. In order to capture the messages the pod creates a handshake a multicast IP address (224.1.0.1) and identifies itself with its own IP address. However, as OpenShift uses its own software defined network, capture the IP address inside the SDN.

*Listing 45. Capture the IP address from the listener and store it in the POD_IP environment variable*

```
[student@workstation ~]$ POD_IP=$(oc get pods mlistener -o jsonpath='{.status.podIP}')
```

*Listing 46. Start the listener*

```
[student@workstation ~]$ oc exec mlistener -it -- socat UDP4-RECVFROM:30102,ip-add-
membership=224.1.0.1:$POD_IP,fork EXEC:hostname
```

Leave the command running and open a new terminal window.

Start the multicast transmitter. In order to send messages socat requires that CIDR is provided to connect to the network. To capture it, use the following command: .Capture CIDR from the SDN:

```
[student@workstation ~]$ CIDR=$(oc get Network.config.openshift.io cluster -o
jsonpath='{.status.clusterNetwork[0].cidr}')
```

*Listing 47. Start sending packet to the multicast address*

```
[student@workstation ~]$  oc exec msender -it -- /bin/bash -c "echo | socat STDIO UDP4-
DATAGRAM:224.1.0.1:30102,range=$CIDR,ip-multicast-ttl=64"
```

Check that in the terminal window that your started the listener, no messages are coming. This is because OpenShift blocks multicast within its SDN.

Run the following command to enable multicast within the project: .Enable multicast on the project.

```
[student@workstation ~]$ oc annotate namespace multicast *k8s.ovn.org/multicast-
enabled=true*
```

```
Re-run the sender again.
```

```
In the window running the sender, restart it by hitting Ctrl+C and re run the command:
[source,bash]
```

```
[student@workstation ~]$  oc exec msender -it -- /bin/bash -c "echo | socat STDIO UDP4-
DATAGRAM:224.1.0.1:30102,range=$CIDR,ip-multicast-ttl=64"
^C
[student@workstation ~]$  oc exec msender -it -- /bin/bash -c "echo | socat STDIO UDP4-
DATAGRAM:224.1.0.1:30102,range=$CIDR,ip-multicast-ttl=64"
mlistener
```

Now the listener sends a message with its own hostname. :pygments-style: tango :source-highlighter: pygments :toc: :toclevels: 7 :sectnums: :sectnumlevels: 6 :numbered: :chapter-label: :icons: font :icons: font :imagesdir: ./images/

# 1.4. Implementing Ingress Node Firewall

Network within a data center must have strict rules, even within the same cluster, like removing ICMP capabilities from those nodes or blocking some ports that malwares might use to access your systems.

In these cases, OpenShift offers the Ingress Node Firewall Operator that allows admins to deploy firewall rules that prevents external agents to access your cluster. It uses eBPF to protect network connections and requires some dependencies, as cert-manager operator and bpfd operator.

## 1.4.1. Installing requirements

### 1.4.1.1. Installing cert-manager operator

This operator allows encrypted network connections using certificates

1. In the OpenShift Container Platform web console, click Operators → OperatorHub.
2. Select cert-manager from the list of available Operators, and then click Install.
3. On the Install Operator page, under Installed Namespace, select Operator recommended Namespace.
4. Click Install.

#### 1.4.1.2. Installing bpfd operator

1. In the OpenShift Container Platform web console, click Operators → OperatorHub.
2. Select Bpfd Operator from the list of available Operators, and then click Install.
3. On the Install Operator page, under Installed Namespace, select Operator recommended Namespace.
4. Click Install.

In order to create all the infrastructure to connect using eBPF, run the following commands:

*Listing 48. Create a new project*

```
[student@workstation ~]$ oc new-project bpfd
```

*Listing 49. Create a self signed certificate issuer*

```
oc apply -f https://raw.githubusercontent.com/bpfd-dev/bpfd/main/bpfd-
operator/config/bpfd-deployment/cert-issuer.yaml
```

*Listing 50. Create a self signed certificate*

```
oc apply -f https://raw.githubusercontent.com/bpfd-dev/bpfd/main/bpfd-
operator/config/bpfd-deployment/certs.yaml
```

### 1.4.2. Installing the Ingress Node Firewall Operator

1. In the OpenShift Container Platform web console, click Operators → OperatorHub.
2. Select Ingress Node Firewall Operator from the list of available Operators, and then click Install.
3. On the Install Operator page, under Installed Namespace, select Operator recommended Namespace.
4. Click Install.

Verify that the Ingress Node Firewall Operator is installed successfully:

1. Navigate to the Operators → Installed Operators page. .Ensure that Ingress Node Firewall Operator is listed in the openshift-ingress-node-firewall project with a Status of InstallSucceeded.

> During installation an Operator might display a Failed status. If the installation later succeeds with an InstallSucceeded message, you can ignore the Failed message.
>
> If the Operator does not have a Status of InstallSucceeded, troubleshoot using the following steps:

> Inspect the Operator Subscriptions and Install Plans tabs for any failures or errors under Status. Navigate to the Workloads → Pods page and check the logs for pods in the openshift-ingress-node-firewall project. Check the namespace of the YAML file. If the annotation is missing, you can add the annotation workload.openshift.io/allowed=management to the Operator namespace with the following command:
>
> $ oc annotate ns/openshift-ingress-node-firewall workload.openshift.io/allowed=management Note For single-node OpenShift clusters, the openshift-ingress-node-firewall namespace requires the workload.openshift.io/allowed=management annotation.

### 1.4.3. Configuring the Ingress Node Firewall

To configure and deploy your ingress node firewall each node must have a pod that manages these firewall rules. Kubernetes provides a resource named DaemonSet that deploys on each node only one pod. An `IngressNodeFirewallConfig` custom resource configures all pods managed by the DaemonSet. There are some rules to deploy it though:

1. There is only one `IngressNodeFirewallConfig` custom resource for the entire cluster.
2. The resource needs to be created inside the `openshift-ingress-node-firewall` project and be named `ingressnodefirewallconfig`.

The operator will consume this resource and create ingress node firewall daemonset daemon which runs on all nodes that match the nodeSelector.

The following configuration example states that a firewall is configured on all worker nodes.

```
apiVersion: ingressnodefirewall.openshift.io/v1alpha1
kind: IngressNodeFirewallConfig
metadata:
  name: ingressnodefirewallconfig
  namespace: openshift-ingress-node-firewall
spec:
  nodeSelector:
    node-role.kubernetes.io/worker: ""
```

Create the `ingressnodefirewallconfig.yaml` text file with your preferred text editor with the previous content.

And run the following command:

```
[student@workstation ~]$ oc create -f ingressnodefirewallconfig.yaml
ingressnodefirewallconfig.ingressnodefirewall.openshift.io/ingressnodefirewallconfig
```

```
created
```

To configure a firewall rule in all worker nodes, create a firewall rule. The following one creates a rule that is valid only in the `eth0` interface and it accepts access from the network 1.1.1.1 and these connections are allowed to access ports 100 to 200.

```
apiVersion: ingressnodefirewall.openshift.io/v1alpha1
kind: IngressNodeFirewall
metadata:
  name: ingressnodefirewall-demo-1
spec:
  interfaces:
  - eth0
  nodeSelector:
    node-role.kubernetes.io/worker: ""
  ingress:
  - sourceCIDRs:
       - 1.1.1.1/24
    rules:
    - order: 10
      protocolConfig:
        protocol: TCP
        tcp:
          ports: "100-200"
      action: Allow
```

The nodeSelector field must either match with the `IngressNodeFirewallConfig` custom resource or bound to a group of nodes that are worker nodes.

To check what network devices are available on a host, access it using a debug pod:

```
[student@workstation ~]$ oc debug node/master01
Starting pod/master01-debug ...
To use host binaries, run `chroot /host`
Pod IP: 192.168.50.10
If you don't see a command prompt, try pressing enter.
sh-4.4#
```

To identify which network devices are available run the following command:

```
sh-4.4# ip a
...
8: *br-ex*: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 8192 qdisc noqueue state UNKNOWN group
```

```
default qlen 1000
    link/ether 52:54:00:00:32:0a brd ff:ff:ff:ff:ff:ff
    inet 192.168.50.10/24 brd 192.168.50.255 scope global dynamic noprefixroute br-ex
       valid_lft 457866971sec preferred_lft 457866971sec
    inet 169.254.169.2/29 brd 169.254.169.7 scope global br-ex
       valid_lft forever preferred_lft forever
    inet6 fe80::5054:ff:fe00:320a/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
...
```

The IP address used by the environment is 192.168.50.xx, and as such, the **br-ex** network bridge is the IP address that all requests are sent.

To check that a ping works from the master node to a worker node run:

```
sh-4.4# ping worker01
...output omitted...
```

To exit the debug pod:

```
sh-4.4# exit
```

For the purpose of this exercise, create the following content into the `ingressnodefirewall.yaml` file:

```
apiVersion: ingressnodefirewall.openshift.io/v1alpha1
kind: IngressNodeFirewall
metadata:
 name: ingressnodefirewall-no-pings
spec:
 interfaces:
 - *br-ex*
 nodeSelector:
   matchLabels:
     node-role.kubernetes.io/worker: ""
 ingress:
 - sourceCIDRs:
      - 192.168.50.0/24
   rules:
   - order: 10
     protocolConfig:
       protocol: ICMP
       icmp:
         icmpType: 8
         icmpCode: 0
```

```
      action: Deny
```

The rule blocks pings within all worker nodes.

Install the rule:

```
[student@workstation ~]$ oc apply -f ingressnodefirewall.yaml
```

Check whether your worker nodes were affected:

```
[student@workstation ~]$  oc get ingressnodefirewallnodestates worker01 -o yaml -n
openshift-ingress-node-firewall
kind: IngressNodeFirewallNodeState
metadata:
  creationTimestamp: "2023-07-17T17:53:00Z"
  generation: 8
  name: worker01
  namespace: openshift-ingress-node-firewall
  ownerReferences:
  - apiVersion: ingressnodefirewall.openshift.io/v1alpha1
    kind: IngressNodeFirewall
    name: ingressnodefirewall-zero-trust
    uid: 647b0bb6-ab94-450a-9d69-d072fb46f0fd
  resourceVersion: "295940"
  uid: 8475a056-afd4-41dc-948b-1cc50d53d680
spec:
  interfaceIngressRules:
    ens3:
    - rules:
      - action: Deny
        order: 10
        protocolConfig:
          icmp:
            icmpType: 8
          protocol: ICMP
      sourceCIDRs:
      - 192.168.50.0/24
status:
  *syncStatus: Synchronized*
```

To check whether the ping is blocked in the worker nodes: .Connect to the worker01 node

```
[student@workstation ~]$ oc debug node/worker01
Warning: would violate PodSecurity "restricted:latest": host namespaces
```

```
(hostNetwork=true, hostPID=true, hostIPC=true), privileged (container "container-00" must
not set securityContext.privileged=true), allowPrivilegeEscalation != false (container
"container-00" must set securityContext.allowPrivilegeEscalation=false), unrestricted
capabilities (container "container-00" must set
securityContext.capabilities.drop=["ALL"]), restricted volume types (volume "host" uses
restricted volume type "hostPath"), runAsNonRoot != true (pod or container "container-00"
must set securityContext.runAsNonRoot=true), runAsUser=0 (container "container-00" must
not set runAsUser=0), seccompProfile (pod or container "container-00" must set
securityContext.seccompProfile.type to "RuntimeDefault" or "Localhost")
Starting pod/worker01-debug ...
To use host binaries, run `chroot /host`
Pod IP: 192.168.50.15
sh-4.4#
```

*Listing 51. Execute the `ping` command*

```
sh-4.4# ping worker02
PING worker02.ocp4.example.com (192.168.50.16) 56(84) bytes of data.
```

## 1.5. Implementing Egress Firewall

OpenShift projects are open to connect to any microservice hosted on Internet. Despite most companies use third party microservices to support their operation, this openness might bring security concerns and it would be better addressed by an API Gateway like 3scale. To block access to external microservices per project basis, OpenShift provides firewalls rules that prevents anythat a deployed pod in a project cannot have external access.

> Different from `NetworkPolicy` custom resource, that prevents access inside the cluster, an `EgressFirewall` custom resource prevents access to external sites.

### 1.5.1. Configuring an `EgressFirewall` custom resource

EgressFirewall custom resources are part of the OVN-Kubernetes network therefore, they are already available on OpenShift. It does not require any extra operator to work like an Ingress Node Firewall.

#### 1.5.1.1. Exercise: Deploying an EgressFirewall to prevent DNS lookups

*Listing 52. Create a new project*

```
[student@workstation ~]$ oc new-project egress-fw
```

In this example, a pod that hosts a simple hello world application is deployed.

*Listing 53. Create a deployment resource on OpenShift that access external resources*

```
[student@workstation ~]$ oc run test-egress --image=quay.io/openshifttest/hello-openshift
```

To check there is connectivity from the project to an external IP address, run a `ping` command to access Google's DNS IP address (4.4.4.4)

*Listing 54. Execute a ping command to a known address*

```
[student@workstation ~]$ oc debug -ti deploy/test-ingress -- ping -c2 4.4.4.4
PING 4.4.4.4 (4.4.4.4) 56(84) bytes of data.
64 bytes from 4.4.4.4: icmp_seq=1 ttl=53 time=4.88 ms
64 bytes from 4.4.4.4: icmp_seq=2 ttl=53 time=2.74 ms
```

### 1.5.2. Deploy an EgressFirewall custom resource to block access to Google

Create the `egress.yaml` text file with your preferred text editor with the following content. In any EgressFirewall custom resource deployed on OpenShift, its name must be `default`.

*Listing 55. Save the following content to the `egress.yaml` file.*

```
apiVersion: k8s.ovn.org/v1
kind: EgressFirewall
metadata:
  name: default
spec:
  egress:
  - type: Allow
    to:
      cidrSelector: 8.8.8.8/32
  - type: Deny
    to:
      cidrSelector: 0.0.0.0/0
```

Any egress firewall definition supports TCP, UDP and SCTP protocols. If needed, the instead of using regular IP address, EgressFirewall custom resources also support DNS entries. Instead of using the `cidrSelector`, the `dnsName` must be used instead. Also, it is possible to use a `nodeSelector` attribute in the same level of the `cidrSelector` to restrict which nodes are supposed to use the rule. This approach is useful to block access to the stage and prod environments but allow access to a dev environment.

And run the following command:

*Listing 56. Create the Egress Firewall*

```
[student@workstation ~]$ oc create -f egress.yaml
```

```
egressfirewall.k8s.ovn.org/default created
```

To evaluate the configuration run the following command line from the project:

*Listing 57. Test the egress firewall rule.*

```
[student@workstation ~]$ oc exec -ti test-egress -- ping -c2 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=53 time=4.88 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=53 time=2.74 ms
```

From the output, ping is working for the 8.8.8.8 IP address. Running on a different IP address, the request fails:

```
[student@workstation ~]$ oc exec -ti test-egress -- ping -c2 4.4.4.4
PING 4.4.4.4 (4.4.4.4) 56(84) bytes of data.

--- 1.1.1.1 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1012ms
```

Running from the same container any DNS entry also fails as the egress firewall blocks access to the IP address translated from the internal OpenShift DNS server.

```
[student@workstation ~]$ oc exec -ti test-egress -- curl https://docs.openshift.com -I
-m2
curl: (28) Failed to connect to docs.openshift.com port 443 after 1511 ms: Operation
timed out
command terminated with exit code 28
```

*Limitations of an egress firewall*

An egress firewall has the following limitations:

1. Egress firewall rules do not apply to traffic that goes through routers. Any user with permission to create a Route CR object can bypass egress firewall policy rules by creating a route that points to a forbidden destination.
2. No project can have more than one `EgressFirewall` sobject.
3. A maximum of one `EgressFirewall` resource with a maximum of 8,000 rules can be defined per project.
4. If you are using the OVN-Kubernetes network plugin with shared gateway mode in Red Hat OpenShift Networking, return ingress replies are affected by egress firewall rules. If the egress firewall rules drop the ingress reply destination IP, the traffic is dropped.

5. Violating any of these restrictions results in a broken egress firewall for the project, and might cause all external network traffic to be dropped.
6. An Egress Firewall resource can be created in the `kube-node-lease`, `kube-public`, `kube-system`, `openshift` and `openshift-` projects.
7. The egress firewall policy rules are evaluated in the order that they are defined, from first to last. The first rule that matches an egress connection from a pod applies. Any subsequent rules are ignored for that connection.

# Chapter 2. Troubleshooting Software Defined Network Issues

## 2.1. Workshop: Configuring Network logging level

Each OpenShift projects supports a different networking logging level. Eventually a firewall rule is too restrictive an it needs to be softened. To capture these logs, use the Namespace resource to configure the levels. For that purpose, create the following source code into a text file named `network-logging.yaml`.

```
kind: Namespace
apiVersion: v1
metadata:
  name: network-logging
  annotations:
    k8s.ovn.org/acl-logging: |-
      {
        "deny": "debug",
        "allow": "debug"
      }
```

After creating it, run the following command:

```
[student@workstation ~]$ oc apply -f network-logging.yaml
```

Alternatively, to create a namespace the process is similar to creating a project:

```
[student@workstation ~]$ oc new-project network-logging
```

Once the project is created, edit the namespace definition by using the `oc annotate` command:

```
[student@workstation ~]$ oc annotate namespace network-logging k8s.ovn.org/acl-logging='{
"deny": "debug", "allow": "debug" }'
```

To evaluate the logging level, adjust the `network-logging` project to configure two network policies that blocks the ingress and egress traffic from the project to another project, but allows communication within the same project. This way any attempt to communicate outside the current project will trigger a message

*Listing 58. Create the `network-policies.yaml` file with your preferred text editor:*

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
spec:
  podSelector:
    matchLabels:
  policyTypes:
  - Ingress
  - Egress
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-same-project
spec:
  podSelector: {}
  policyTypes:
   - Ingress
   - Egress
  ingress:
    - from:
        - podSelector: {}
  egress:
    - to:
       - namespaceSelector:
          matchLabels:
            namespace: network-logging
```

To deploy these network policies in the same project, run the following command:

```
[student@workstation ~]$ oc apply -f network-policies.yaml
networkpolicy.networking.k8s.io/deny-all created
networkpolicy.networking.k8s.io/allow-from-same-project created
```

*Listing 59. Create, using your preferred text editor, the following content in a file named `server.yaml`:*

```
apiVersion: v1
kind: Pod
metadata:
  name: server
spec:
  containers:
    - name: server
```

```
      image: registry.access.redhat.com/rhel7/rhel-tools
      command: ["/bin/sh", "-c"]
      args:
        ["sleep inf"]
```

The previous code creates a pod named server that contains some network tools to test our connectivity.

*Listing 60. Create the pod:*

```
[student@workstation ~]$ oc create -f server.yaml -n network-logging
```

*Listing 61. Create, using your preferred text editor the following content in a file named `client.yaml`:*

```
apiVersion: v1
kind: Pod
metadata:
  name: client
spec:
  containers:
    - name: client
      image: registry.access.redhat.com/rhel7/rhel-tools
      command: ["/bin/sh", "-c"]
      args:
        ["sleep inf"]
```

The previous code creates a pod named client using tools from RHEL. In a later moment execute some tools to check the connectivity.

Create a new project that another pod runs:

```
[student@workstation ~]$ oc new-project network-logging-2
```

Create the pod:

```
[student@workstation ~]$ oc create -f client.yaml -n network-logging-2
```

*Listing 62. Capture the IP address from the listener and store it in the POD_IP environment variable*

```
[student@workstation ~]$ POD_IP=$(oc get pods server -n network-logging -o
jsonpath='{.status.podIP}')
```

*Listing 63. Ping the server pod that is running on the network-logging project*

```
[student@workstation ~]$ oc exec -it client -n network-logging-2 -- /bin/ping -c 2
$POD_IP
PING 10.10.2.25 (10.10.2.25) 56(84) bytes of data.
--- 10.10.2.25 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1043ms

command terminated with exit code 1
```

To check the logs from the network logging, the openshift-ovn-kubernetes project offers ways to check the logs by checking the /var/log/ovn directory in one of these containers.

To check these packets that were blocked you need to evaluate each of these pods and identify which one was responsible to capture the log.

*Listing 64. Connect to all pods managed by the daemonset*

```
[student@workstation ~]$  oc debug daemonset/ovnkube-node -n openshift-ovn-kubernetess
If you don't see a command prompt, try pressing enter.
```

*Listing 65. Evaluate the logs:*

```
sh-4.4# cat /var/log/ovn/acl-audit-log.log
2023-07-18T15:43:03.467Z|00004|acl_log(ovn_pinctrl0)|INFO|name="network-
logging_ingressDefaultDeny", verdict=drop, severity=debug, direction=to-lport:
icmp,vlan_tci=0x0000,dl_src=0a:58:0a:0a:02:01,dl_dst=0a:58:0a:0a:02:19,nw_src=10.9.2.11,n
w_dst=10.10.2.25,nw_tos=0,nw_ecn=0,nw_ttl=63,nw_frag=no,icmp_type=8,icmp_code=0
2023-07-18T15:43:04.489Z|00005|acl_log(ovn_pinctrl0)|INFO|name="network-
logging_ingressDefaultDeny", verdict=drop, severity=debug, direction=to-lport:
icmp,vlan_tci=0x0000,dl_src=0a:58:0a:0a:02:01,dl_dst=0a:58:0a:0a:02:19,nw_src=10.9.2.11,n
w_dst=10.10.2.25,nw_tos=0,nw_ecn=0,nw_ttl=63,nw_frag=no,icmp_type=8,icmp_code=0
```

## 2.2. Workshop: Installing Network Observability

> The process described in the section is for testing purposes. Due to limitations in our infrastructure, the environment needs be customized to minimize resource consumption. In a real world condition though, you need to install Red Hat OpenShift Data Foundation or use an underlying object storage solution.

Install a slimmed version of Loki (the current Logging solution provided by OpenShift storess all data collected from application and infrastructure logs in a robust storage systems).

To create a project that is used to install the Network Observability Operator on OpenShift, execute the following command:

```
[student@workstation ~]$ oc new-project netobserv
```

Loki requires a large storage to keep log information and Red Hat OpenShift Data Foundation (ODF), which is powered by Ceph, simplifies this kind of management.

The intent of ODF is to create a robust distributed storage in spread in multiple hosts.

It resembles Linux Logical Volume Management (LVM), but instead of creating a transparent layer in a single machine it goes beyond and create a common access point to store data in multiple hosts.

As we are in a proof of concept environment, the easiest path is to create a mount point that stores data from logs. Create the `pvc.yaml` file as follows:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: loki-store
spec:
  resources:
    requests:
      storage: 1G
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
```

The file create a `loki-store` PersistentVolumeClaim resource that trigger in OpenShift the need to create a mount point that stores data from Loki.

After creating it, run the following command:

```
[student@workstation ~]$ oc apply -f pvc.yaml
```

In the following step, three elements are created:

1. a `local-config.yaml` configuration file and stored in OpenShift as a ConfigMap. In a nutshell, it provides Loki configuration about:
    a. ports inside your cluster,
    b. directories used by Loki to store data
    c. overall resource usage
2. a `Pod` that runs Loki with the PVC created in the previous step and

*Listing 66. Create the `resources.yaml` file with your preferred text editor:*

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: loki-config
data:
  local-config.yaml: |
    auth_enabled: false
    server:
      http_listen_port: 3100
      grpc_listen_port: 9096
      http_server_read_timeout: 1m
      http_server_write_timeout: 1m
      log_level: error
    target: all
    common:
      path_prefix: /loki-store
      storage:
        filesystem:
          chunks_directory: /loki-store/chunks
          rules_directory: /loki-store/rules
      replication_factor: 1
      ring:
        instance_addr: 127.0.0.1
        kvstore:
          store: inmemory
    compactor:
      compaction_interval: 5m
      retention_enabled: true
      retention_delete_delay: 2h
      retention_delete_worker_count: 150
    frontend:
      compress_responses: true
    ingester:
      chunk_encoding: snappy
      chunk_retain_period: 1m
    query_range:
      align_queries_with_step: true
      cache_results: true
      max_retries: 5
      results_cache:
        cache:
          enable_fifocache: true
          fifocache:
            max_size_bytes: 500MB
            validity: 24h
      parallelise_shardable_queries: true
```

```
    query_scheduler:
      max_outstanding_requests_per_tenant: 2048
    schema_config:
      configs:
        - from: 2022-01-01
          store: boltdb-shipper
          object_store: filesystem
          schema: v11
          index:
            prefix: index_
            period: 24h
    storage_config:
      filesystem:
        directory: /loki-store/storage
      boltdb_shipper:
        active_index_directory: /loki-store/index
        shared_store: filesystem
        cache_location: /loki-store/boltdb-cache
        cache_ttl: 24h
    limits_config:
      ingestion_rate_strategy: global
      ingestion_rate_mb: 4
      ingestion_burst_size_mb: 6
      max_label_name_length: 1024
      max_label_value_length: 2048
      max_label_names_per_series: 30
      reject_old_samples: true
      reject_old_samples_max_age: 15m
      creation_grace_period: 10m
      enforce_metric_name: false
      max_line_size: 256000
      max_line_size_truncate: false
      max_entries_limit_per_query: 10000
      max_streams_per_user: 0
      max_global_streams_per_user: 0
      unordered_writes: true
      max_chunks_per_query: 2000000
      max_query_length: 721h
      max_query_parallelism: 32
      max_query_series: 10000
      cardinality_limit: 100000
      max_streams_matchers_per_query: 1000
      max_concurrent_tail_requests: 10
      retention_period: 24h
      max_cache_freshness_per_query: 5m
      max_queriers_per_tenant: 0
      per_stream_rate_limit: 3MB
      per_stream_rate_limit_burst: 15MB
```

```yaml
      max_query_lookback: 0
      min_sharding_lookback: 0s
      split_queries_by_interval: 1m
---
apiVersion: v1
kind: Pod
metadata:
  name: loki
  labels:
    app: loki
spec:
  securityContext:
    runAsGroup: 1000
    runAsUser: 1000
    fsGroup: 1000
  volumes:
    - name: loki-store
      persistentVolumeClaim:
        claimName: loki-store
    - name: loki-config
      configMap:
        name: loki-config
  containers:
    - name: loki
      image: grafana/loki:2.6.1
      volumeMounts:
        - mountPath: "/loki-store"
          name: loki-store
        - mountPath: "/etc/loki"
          name: loki-config
      securityContext:
        seccompProfile:
          type: RuntimeDefault
        allowPrivilegeEscalation: false
        capabilities:
          drop:
            - ALL
---
kind: Service
apiVersion: v1
metadata:
  name: loki
spec:
  selector:
    app: loki
  ports:
    - port: 3100
```

```
    protocol: TCP
```

To deploy these resources in the same project, run the following command:

```
[student@workstation ~]$ oc apply -f resources.yaml
```

In a production environment, install Loki with OperatorHub. To configure, deploy the `LokiStack` custom resource available in the web console installation.

From this point the configuration can be followed as it is in a production environment

Deploy the Network Observability Operator from the OperatorHub.

*Navigate to the* **Flow Collector** *tab, and click* **Create FlowCollector**. *Make the following selections in the form view:*

The current configuration works for the current environment, however some customization might be needed in a production environment.

*Click Create.*

To confirm this was successful, when you navigate to Observe you should see **Network Traffic** listed in the options.

To verify the traffic **clear all filters** to check the which projects are creating traffic inside the entire cluster. The Overview tab provides a multitude of graphics with information on how much traffic is going through the cluster, which are the top 5 flow rates.

To check which projects and pods are creating traffic, open the Traffic flow tab.

To evaluate a graphical view of data flow, select the Topology tab.

To create a load in a project, use the following command to deploy a pod that deploy multiple pods to check the traffic. The namespace created by this application is `kube-traffic-generator`.

```
oc apply -f https://raw.githubusercontent.com/netobserv/documents/main/examples/kube-traffic-generator/traffic.yaml
```

To avoid overload from our environment, run the following command:

oc delete -f https://raw.githubusercontent.com/netobserv/documents/main/examples/kube-traffic-generator/traffic.yaml