Technology Guide

Kubernetes Operator - Intel® Ethernet Operator

Authors

Emma Kenny

Abdul Halim

Padraig Connolly

Richard Walsh

1 Introduction

This document describes the Intel® Ethernet Operator, a Kubernetes Operator developed using the Operator SDK, designed to enable the network resource management of the advanced Ethernet technology features introduced by the Intel® Ethernet 800 Series Network Adapters. This operator does not support Intel® Ethernet 700 Series or 500 Series Network Adapters.

The Intel Ethernet Operator has been verified with the following network adapters:

- Intel® Ethernet Network Adapter E810-CQDA1/CQDA2
- Intel® Ethernet Network Adapter E810-XXVDA4
- Intel® Ethernet Network Adapter E810-XXVDA2

The Intel Ethernet Operator packages two core components, the Firmware/Dynamic Device Personalization (FW/DDP) daemon and the Unified Flow Operator (UFO). Each provide a declarative API, which allows Kubernetes users and administrators to configure and manage an Intel E810 series adapter to optimize network performance.

The operator provides the Kubernetes user or administrator with the capabilities of updating device firmware, configuring DDP profiles, and utilizing the Generic Flow API (RTE_FLOW) provided by the Data Plane Development Kit (DPDK). All this functionality provides the benefit of workload-specific network optimizations in a cloud-native environment.

This document is part of the Network Transformation Experience Kits.

Table of Contents

Figures

Tables

Document Revision History

Revision	Date	Description
001	February 2023	Initial release.

1.1 Terminology

Table 1. Terminology

Abbreviation	Description
CRD	Custom Resource Definition
CR	Custom Resource
DCF	Device Configuration Function
DDP	Dynamic Device Personalization
DPDK	Data Plane Development Kit
FW	Firmware
NVM	Non-Volatile Memory
SR-IOV	Single Root I/O Virtualization
UFO	Unified Flow Operator
UFT	Unified Flow Tool
VF	Virtual Function

1.2 Reference Documentation

Table 2. Reference Documents

Reference	Source
Intel® Ethernet Operator documentation	https://github.com/intel/intel-ethernet-operator/blob/main/docs/intelethernet-operator.md
Intel® Ethernet Operator Solution Brief	https://networkbuilders.intel.com/solutionslibrary/intel-ethernet-operator-overview-solution-brief
Intel® Ethernet Operator Source Code GitHub Repository	https://github.com/intel/intel-ethernet-operator/releases/tag/v22.11
Network and Edge Container Bare Metal Reference System Architecture User Guide	https://networkbuilders.intel.com/solutionslibrary/network-and-edge-container-bare-metal-reference-system-architecture-user-guide

2 Overview

2.1 Technology Description

The Intel Ethernet Operator includes several Kubernetes Custom Resource Definitions (CRDs) that define the network interface configuration state of individual nodes.

These definitions serve as templates for the administrator to create Custom Resources (CRs) to configure:

- Network adapter firmware versions
- Network adapter DDP profiles
- Flow rules

The CRs can target individual or multiple nodes in the cluster, and their corresponding network interfaces. The CRs are Kubernetes API compatible and can easily be applied to a Kubernetes client.

Once applied, each configuration CR is reconciled by its corresponding controller pod, which is also deployed as part of the operator. The reconcile loop recognizes the applied configuration, compares it against the current configuration, and then applies the configuration changes via the necessary tools. For example, when a CR is applied that requests an update of the firmware for a network adapter on a specified node, the controller will communicate with that node and initiate the update via the Non-volatile Memory (NVM) utility running on that node. This is explained in detail later in this document.

3 Components

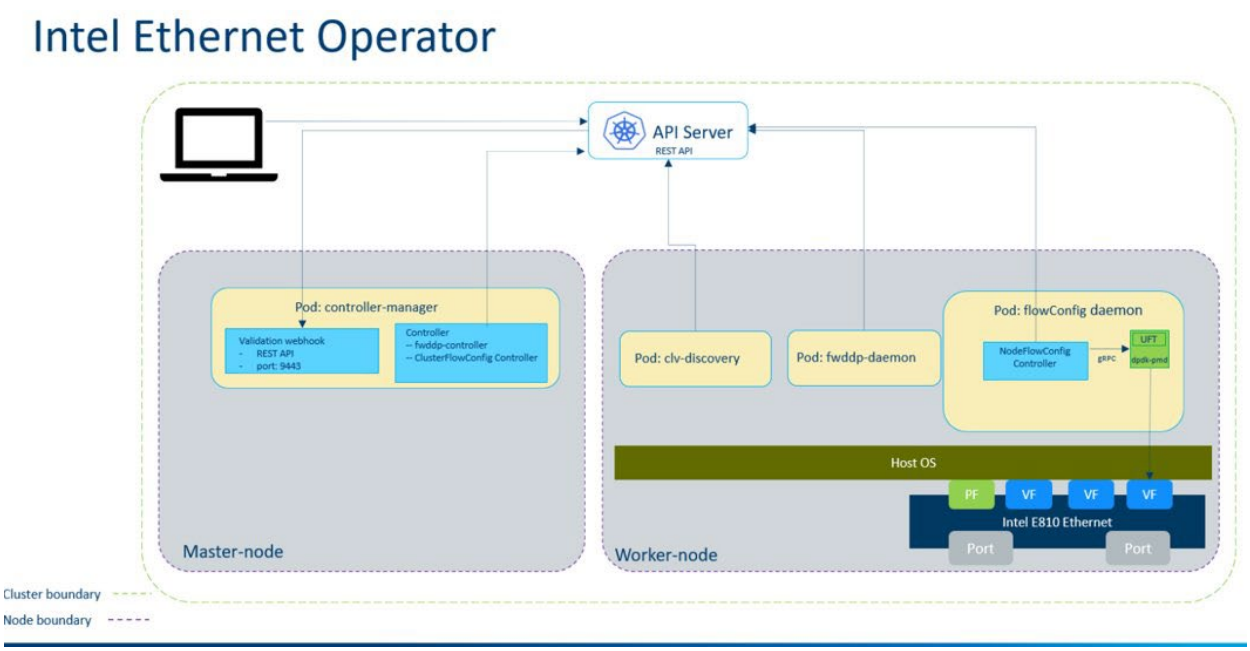

Figure 1. Sample Ethernet Operator Deployment Diagram

3.1 Controller Manager

The controller manager pod is the initial pod of the operator. It is responsible for deploying other assets within the operator, exposing the APIs, handling the CRs, and executing the validation webhook. It contains the logic for accepting and splitting the FW/DDP CRs into node CRs and reconciling the status of each CR. The validation webhook of the controller manager is responsible for checking each CR for invalid arguments.

3.2 Device Discovery

The clv-discovery pod is a DaemonSet deployed on each worker node in the cluster. It is responsible for checking if a supported hardware device is discovered on the platform and then labeling that worker node accordingly.

To get the worker nodes that contain discovered devices (when the Intel Ethernet Operator is deployed), run:

```
kubectl get EthernetNodeConfig -A

NAMESPACE                  NAME       UPDATE
intel-ethernet-operator    worker-1   InProgress
```

intel-ethernet-operator worker-2 InProgressTo get more information about the currently loaded FW/DDP packages, run:

```
kubectl describe EthernetNodeConfig worker-1 -n intel-ethernet-operator
```

To obtain the list of supported devices found (when the Intel Ethernet Operator is deployed), run:

```
kubectl describe configmap supported-clv-devices -n intel-ethernet-operator
```

3.3 FW/DDP Daemon

The FW/DDP daemon pod is a DaemonSet, which is deployed as part of the operator. It is deployed on each worker node that has the appropriate label present that indicates that a supported Intel® Ethernet 800 Series Network Adapter has been detected. The pod consists of a reconcile loop, which monitors and acts on the changes in each node's EthernetNodeConfig. The logic implemented in this Daemon handles updating the device firmware and DDP profile. This logic is also responsible for draining the nodes, taking them out of commission and rebooting as required by an update.

3.4 Cluster Flow Configuration Controller

The Cluster Flow Configuration controller watches for flow rule changes via a cluster wide CRD. This CRD is of kind ClusterFlowConfig. Based on the pod selector outlined in the CRD, the controller will determine the interface details for the target set of pods and create a NodeFlowConfig CR object, which contains a complete flow rule specification.

3.5 Node Flow Configuration Controller

The Node Flow Configuration Controller watches for changes in flow rules via a node specific CRD. This CRD is of kind NodeFlowConfig and should be named (`metadata.name`) based upon the node that it is deployed on. Once the operator detects a change to this CR related to the Intel E810 Series Flow Configuration, it then attempts to create/update/delete rules via the Unified Flow Tool (UFT) over an internal gRPC API call.

3.6 Unified Flow Tool

When a change to the Flow Configuration is required, the Flow Config Controller will communicate with the UFT container running a DPDK Device Configuration Function (DCF) application. The DCF VF should be set to VF 0. The UFT application accepts the configuration as an input. The application then programs the device using a trusted VF created for this device. It is the responsibility of the user to provide the trusted VFs as an allocatable K8s resource using the Single Root I/O Virtualization (SR-IOV) Network Operator.

4 Deployment

4.1 Prerequisites

- Intel® Ethernet 800 Series Network Adapters (as stated in the Introduction section)

Vanilla Kubernetes Cluster	RedHat Openshift Cluster
Bare Metal Reference Architecture v21.08 or later with remote_fp profile	RedHat Openshift Container Platform (RHOCP) Version 4.9, 4.10, 4.11
Kubernetes Version 1.25.3	Out of Tree ICE Driver

- On a jumphost where the operator will be deployed:
  - Operator SDK v1.25.0
  - Golang v1.19
  - Container image utility: Docker or Podman
  - Node Feature Discovery

4.2 Known Limitations

- The certified release version 0.1.0 currently only supports Firmware Update functionality. Due to the use of in-tree driver, the DDP Update and Flow Rule Configuration is not supported.

4.3 Building the Operator from Source

To build the operator, the images must be built from source here https://github.com/intel/intel-ethernet-operator. If you are using prebuilt images from the Red Hat Catalog, then you can skip this step.

- VERSION is the version reference to be applied to the bundle i.e: 0.0.1
- IMAGE_REGISTRY is the address of the registry where the build images should be pushed i.e.: my.private.registry.com
- TLS_VERIFY defines whether the connection to the registry needs TLS verification. The default value is false
- TARGET_PLATFORM specifies the platform, which the operator will be built on. Supported values are OCP and K8S. The default value is OCP.

Note: If the operator is built on a platform other than OCP, the user must manually install the sriov-network-operator as described here: https://github.com/k8snetworkplumbingwg/sriov-network-operator

```
make VERSION=$(VERSION) IMAGE_REGISTRY=$(IMAGE_REGISTRY) TLS_VERIFY=$(TLS_VERIFY)
TARGET_PLATFORM=$(TARGET_PLATFORM) build_all push_all catalog-build catalog-push
```

4.4 Installing the bundle

Once you have the operator images built, and they are accessible inside the cluster, the operator can be installed by running the following commands:

Create a namespace for the operator:

```
kubectl create ns intel-ethernet-operator
```

Create the following Catalog Source yaml file:

- VERSION is the version of the image that you would like to use, that is: 0.0.1
- IMAGE_REGISTRY is the address of the registry where the build images are stored ie: my.private.registry.com

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: intel-ethernet-operators
  namespace: openshift-marketplace
spec:
  sourceType: grpc
  image: <IMAGE_REGISTRY>/intel-ethernet-operator-catalog:<VERSION>
  publisher: Intel
  displayName: Intel ethernet operators(Local)
```

Apply the Catalog Source file:

```
kubectl apply -f <filename>
```

Create the following yaml file, which includes Subscription and OperatorGroup:

```
---
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: intel-ethernet-operator
  namespace: intel-ethernet-operator
spec:
  targetNamespaces:
    - intel-ethernet-operator
---
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: intel-ethernet-subscription
  namespace: intel-ethernet-operator
spec:
  channel: alpha
  name: intel-ethernet-operator
  source: intel-ethernet-operators
  sourceNamespace: olm
  installPlanApproval: Automatic
```

Subscribe to and install the operator by applying the yaml file:

```
kubectl apply -f <filename>
```

You can check if the operator has deployed successfully, by running:

```
kubectl get pods -n intel-ethernet-operator
NAME                                                              READY   STATUS    RESTARTS
AGE
pod/clv-discovery-4vk8l                                           1/1     Running   0
22h
pod/fwddp-daemon-sjzlz                                            1/1     Running   0
22h
pod/intel-ethernet-operator-controller-manager-59645597f6-gktpm   1/1     Running   0
22h
pod/intel-ethernet-operator-controller-manager-59645597f6-jfsn9   1/1     Running   0
22h
NAME                                                          TYPE        CLUSTER-IP
EXTERNAL-IP   PORT(S)   AGE
service/intel-ethernet-operator-controller-manager-service   ClusterIP   10.104.6.72     <none>
443/TCP   22h
service/intel-ethernet-operator-webhook-service               ClusterIP   10.98.197.202   <none>
443/TCP   22h
NAME                           DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE
SELECTOR                              AGE
daemonset.apps/clv-discovery   1         1         1       1            1           <none>
22h
daemonset.apps/fwddp-daemon    1         1         1       1            1
ethernet.intel.com/intel-ethernet-present=   22h
```

```
NAME                                                     READY   UP-TO-DATE   AVAILABLE
AGE
deployment.apps/intel-ethernet-operator-controller-manager   2/2     2            2
22h
NAME                                                                DESIRED   CURRENT
READY   AGE
replicaset.apps/intel-ethernet-operator-controller-manager-59645597f6   2         2         2
22h
```

4.5 Deploying the Flow Configuration Feature

The Flow configuration feature can be enabled by deploying the Flowconfig Daemon agent in the Intel Ethernet Operator. This agent requires that the supported network adapter's VFs are created via the SR-IOV Network Operator and the VF pool, with admin capability, is available along with the Network Attachment definition.

4.5.1 Create Trusted VFs using the SR-IOV Network Operator

Once the SR-IOV Network Operator is running on your cluster, we can examine the SriovNetworkNodeStates to view available Intel® Ethernet 800 Series Network Adapters. From this, we can find the network adapter information such as PCI address and interface names, which are used to define the SriovNetworkNodePolicy to create the required VF pools:

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
  name: uft-admin-policy
  namespace: intel-ethernet-operator
spec:
  deviceType: vfio-pci
  nicSelector:
    pfNames:
    - ens1f0#0-0
    - ens1f1#0-0
    vendor: "8086"
  nodeSelector:
    feature.node.kubernetes.io/network-sriov.capable: 'true'
  numVfs: 8
  priority: 99
  resourceName: cvl_uft_admin
```

The CR can be applied by running:

```
kubectl create -f sriov-network-policy.yaml
```

4.5.2 Check Node Status

Check the node status to confirm that the cvl_uft_admin resource pool registered the DCF capable VFs on the node:

```
# kubectl describe node worker-01 -n intel-ethernet-operator | grep -i allocatable -A 20
Allocatable:
  bridge.network.kubevirt.io/cni-podman0:  1k
  cpu:                                     108
  ephemeral-storage:                       468315972Ki
  hugepages-1Gi:                           0
  hugepages-2Mi:                           8Gi
  memory:                                  518146752Ki
  intel.com/cvl_uft_admin:                 2
  pods:                                    250
```

4.5.3 Create a DCF Capable SR-IOV Network

Next, create the SR-IOV Network Attachment Definition for the DCF VF pool:

```
cat <<EOF | kubectl apply -f -
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetwork
metadata:
  name: sriov-cvl-dcf
spec:
  trust: 'on'
  networkNamespace: intel-ethernet-operator
  resourceName: cvl_uft_admin
EOF
```

4.5.4 Build the UFT image

Next, build the UFT image:

```
export IMAGE_REGISTRY=<Your Image registry>
git clone https://github.com/intel/UFT.git
git checkout v22.07
make dcf-image
docker tag dcf-tool:v22.07 $IMAGE_REGISTRY/uft:v22.07
docker push $IMAGE_REGISTRY/uft:v22.07
```

4.5.5 Creating the Flow Config Node Agent Deployment CR

Note: The Admin VF pool prefix in DCFVfPoolName should match how it is shown in the node description from section 4.5.2:

```
cat <<EOF | kubectl apply -f -
apiVersion: flowconfig.intel.com/v1
kind: FlowConfigNodeAgentDeployment
metadata:
  labels:
    control-plane: flowconfig-daemon
  name: flowconfig-daemon-deployment
  namespace: intel-ethernet-operator
spec:
  DCFVfPoolName: intel.com/cvl_uft_admin
  NADAnnotation: sriov-cvl-dcf
EOF
```

To verify that the Flow Config Daemon is up and running on the nodes, run the following command:

```
kubectl get pods -n intel-ethernet-operator
NAME                                                  READY   STATUS    RESTARTS   AGE
clv-discovery-kwjkb                                   1/1     Running   0          44h
clv-discovery-tpqzb                                   1/1     Running   0          44h
flowconfig-daemon-worker-01                           2/2     Running   0          44h
fwddp-daemon-m8d4w                                    1/1     Running   0          44h
intel-ethernet-operator-controller-manager-79c4d5bf6d-bjlr5   1/1     Running   0          44h
intel-ethernet-operator-controller-manager-79c4d5bf6d-txj5q   1/1     Running   0          44h

kubectl logs -n intel-ethernet-operator flowconfig-daemon-worker-01 -c uft
Generating server_conf.yaml file...
Done!
server :
    ld_lib : "/usr/local/lib64"
ports_info :
    - pci  : "0000:18:01.0"
      mode : dcf
do eal init ...
[{'pci': '0000:18:01.0', 'mode': 'dcf'}]
[{'pci': '0000:18:01.0', 'mode': 'dcf'}]
the dcf cmd line is: a.out -c 0x30 -n 4 -a 0000:18:01.0,cap=dcf -d /usr/local/lib64 --file-
prefix=dcf --
EAL: Detected 96 lcore(s)
EAL: Detected 2 NUMA nodes
EAL: Detected shared linkage of DPDK
EAL: Multi-process socket /var/run/dpdk/dcf/mp_socket
EAL: Selected IOVA mode 'VA'
EAL: No available 1048576 kB hugepages reported
EAL: VFIO support initialized
EAL: Using IOMMU type 1 (Type 1)
EAL: Probe PCI driver: net_iavf (8086:1889) device: 0000:18:01.0 (socket 0)
EAL: Releasing PCI mapped resource for 0000:18:01.0
EAL: Calling pci_unmap_resource for 0000:18:01.0 at 0x2101000000
EAL: Calling pci_unmap_resource for 0000:18:01.0 at 0x2101020000
EAL: Using IOMMU type 1 (Type 1)
EAL: Probe PCI driver: net_ice_dcf (8086:1889) device: 0000:18:01.0 (socket 0)
ice_load_pkg_type(): Active package is: 1.3.30.0, ICE COMMS Package (double VLAN mode)
TELEMETRY: No legacy callbacks, legacy socket not created
grpc server start ...
now in server cycle
```

5 Functionality Use Cases

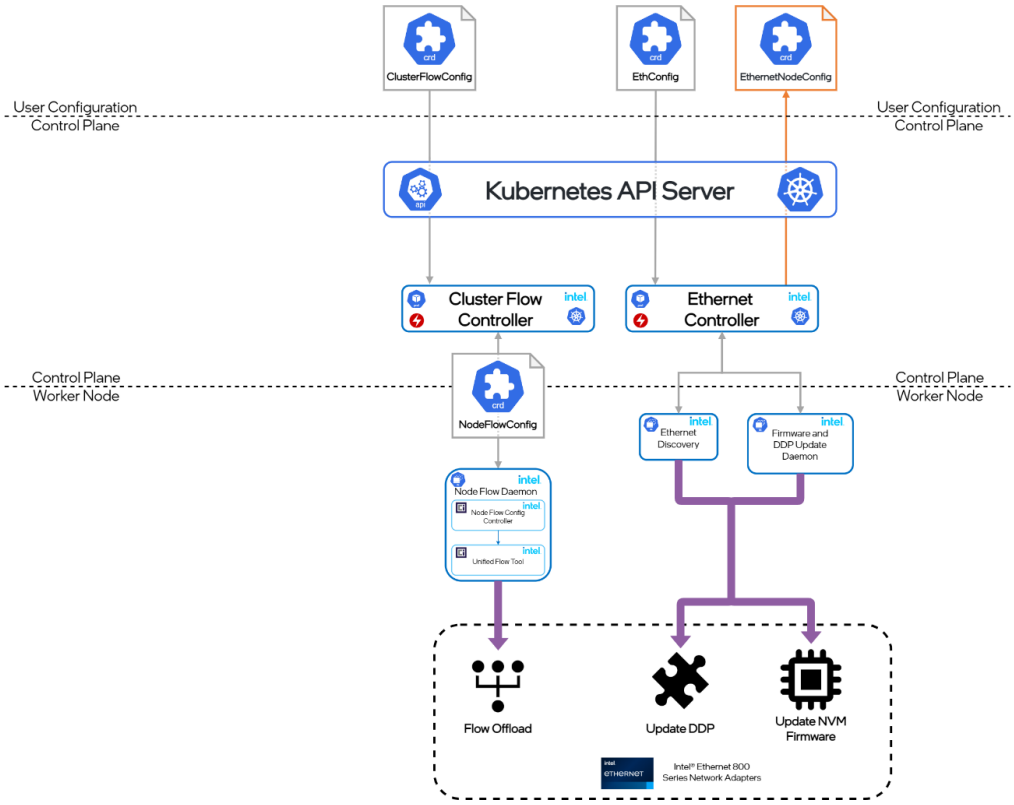

Figure 2. Use Cases for Firmware, DDP, and Flow Functionality

5.1 Firmware Update

When the operator detects a change to a CR related to the update of the Intel® Ethernet 800 Series Network Adapters firmware, it will try to perform an update. The user is expected to provide the firmware for the Intel Ethernet Network Adapter E810 device in the form of a tar.gz file. The user is also responsible to verify that the firmware version is compatible with the device. The user can place the firmware on an accessible local HTTP server and provide the URL in the CR. If the file is provided correctly, and the firmware is to be updated, the Ethernet Configuration Daemon will update the Intel Ethernet Network Adapter E810 device using the NVM utility provided.

To update the NVM firmware of the Intel Ethernet Network Adapter E810 devices, the user can create a CR containing the information about which card should be updated. The physical functions of the devices will be updated in logical pairs. The user needs to provide the FW URL and checksum (SHA-1) in the CR.

```
apiVersion: ethernet.intel.com/v1
kind: EthernetClusterConfig
metadata:
  name: config
  namespace: <namespace>
spec:
  nodeSelectors:
    kubernetes.io/hostname: <hostname>
  deviceSelector:
    pciAddress: "<pci-address>"
  deviceConfig:
    fwURL: "<URL_to_firmware>"
    fwChecksum: "<file_checksum_SHA-1_hash>"
```

5.2 Dynamic Device Personalization Update

Once the operator detects a change to the CR, related to the update of the Intel® Ethernet 800 Series Network Adapters DDP profile, it will attempt to perform a DDP update. The DDP profile for the Intel Ethernet Network Adapter E810 devices is expected to be provided by the user. The user is also responsible for verifying that the DDP version is compatible with the device. The user can place the DDP package on an accessible local HTTP server and provide a URL for it in the CR. If the file is

provided correctly, and the DDP is to be updated, the Ethernet Configuration Daemon will update the DDP profile of the Intel Ethernet Network Adapter E810 device by placing it in the correct filesystem on the host.

To update the DDP profile, the user can create a CR containing the information about which device should be updated. All the physical functions of that network adapter will be updated.

```
apiVersion: ethernet.intel.com/v1
kind: EthernetClusterConfig
metadata:
  name: config
  namespace: <namespace>
spec:
  nodeSelectors:
    kubernetes.io/hostname: <hostname>
  deviceSelector:
    pciAddress: "<pci-address>"
  deviceConfig:
    ddpURL: "<URL_to_DDP>"
    ddpChecksum: "<file_checksum_SHA-1_hash>"
```

5.3 Flow Configuration

The Flow Configuration pod is a node agent deployed with a CRD FlowConfigNodeAgentDeployment provided by the Ethernet operator once it is up and running. The required DCF VF pools and their network attachment definitions are created using SR-IOV Network Operator APIs. The FlowConfigNodeAgent is deployed on each worker node that exposes DCF VF pool as extended node resource. The pod consists of a reconcile loop, which monitors the changes in each nodes CR and acts on those changes. The logic implemented in this Daemon takes care of updating the devices traffic flow configuration. It consists of three components: Cluster Flow Configuration Controller, Node Flow Configuration Controller, and the Unified Flow Tool.

Once the Flow config feature is enabled by deploying flow config daemons as described in section 4.5, the flow configuration rules for the cluster can be created using following steps.

5.3.1 Create Flow Configuration Rules

With trusted VFs and application VFs ready to be configured, there are two options to create flow rules:

The first option is to use a cluster wide ClusterFlowConfig CR, which will target nodes using a pod selector:

```
cat <<EOF | kubectl apply -f -
apiVersion: flowconfig.intel.com/v1
kind: ClusterFlowConfig
metadata:
  name: pppoes-sample
  namespace: intel-ethernet-operator
spec:
  rules:
    - pattern:
        - type: RTE_FLOW_ITEM_TYPE_ETH
        - type: RTE_FLOW_ITEM_TYPE_IPV4
          spec:
            hdr:
              src_addr: 10.56.217.9
          mask:
            hdr:
              src_addr: 255.255.255.255
        - type: RTE_FLOW_ITEM_TYPE_END
      action:
        - type: to-pod-interface
          conf:
            podInterface: net1
      attr:
        ingress: 1
        priority: 0
  podSelector:
      matchLabels:
        app: vagf
        role: controlplane

EOF
```

The second option is to use a node specific flow rule configuration by creating a NodeFlowConfig CR using the same name as the target node but with an empty specification

```
cat <<EOF | kubectl apply -f -
apiVersion: flowconfig.intel.com/v1
kind: NodeFlowConfig
metadata:
  name: worker-01
spec:
EOF
```

For the node flow configuration, you can check status of the CR as follows:

```
# kubectl describe nodeflowconfig worker-01
Name:         worker-01
Namespace:    intel-ethernet-operator
Labels:       <none>
Annotations:  <none>
API Version:  flowconfig.intel.com/v1
Kind:         NodeFlowConfig
Metadata:
Status:
  Port Info:
    Port Id:    0
    Port Mode:  dcf
    Port Pci:   0000:18:01.0
Events:         <none>
```

Also, specifically for the second option using the node flow configuration, refer to the DCF port information above. This information can be used to identify which port on a node the Flow Rules should be applied to. To update the node flow configuration, use the following command:

```
cat <<EOF | kubectl apply -f -
apiVersion: flowconfig.intel.com/v1
kind: NodeFlowConfig
metadata:
  name: worker-01
  namespace: intel-ethernet-operator
spec:
  rules:
    - pattern:
        - type: RTE_FLOW_ITEM_TYPE_ETH
        - type: RTE_FLOW_ITEM_TYPE_IPV4
          spec:
            hdr:
              src_addr: 10.56.217.9
          mask:
            hdr:
              src_addr: 255.255.255.255
        - type: RTE_FLOW_ITEM_TYPE_END
      action:
        - type: RTE_FLOW_ACTION_TYPE_DROP
        - type: RTE_FLOW_ACTION_TYPE_END
      portId: 0
      attr:
        ingress: 1
EOF
```

Validate that flow rules are applied by the controller by inspecting the UFT logs:

```
kubectl logs flowconfig-daemon-worker uft
Generating server_conf.yaml file...
Done!
server :
    ld_lib : "/usr/local/lib64"
ports_info :
    - pci  : "0000:18:01.0"
      mode : dcf
do eal init ...
[{'pci': '0000:18:01.0', 'mode': 'dcf'}]
```

```
[{'pci': '0000:18:01.0', 'mode': 'dcf'}]
the dcf cmd line is: a.out -c 0x30 -n 4 -a 0000:18:01.0,cap=dcf -d /usr/local/lib64 --file-
prefix=dcf --
EAL: Detected 96 lcore(s)
EAL: Detected 2 NUMA nodes
EAL: Detected shared linkage of DPDK
EAL: Multi-process socket /var/run/dpdk/dcf/mp_socket
EAL: Selected IOVA mode 'VA'
EAL: No available 1048576 kB hugepages reported
EAL: VFIO support initialized
EAL: Using IOMMU type 1 (Type 1)
EAL: Probe PCI driver: net_iavf (8086:1889) device: 0000:18:01.0 (socket 0)
EAL: Releasing PCI mapped resource for 0000:18:01.0
EAL: Calling pci_unmap_resource for 0000:18:01.0 at 0x2101000000
EAL: Calling pci_unmap_resource for 0000:18:01.0 at 0x2101020000
EAL: Using IOMMU type 1 (Type 1)
EAL: Probe PCI driver: net_ice_dcf (8086:1889) device: 0000:18:01.0 (socket 0)
ice_load_pkg_type(): Active package is: 1.3.30.0, ICE COMMS Package (double VLAN mode)
TELEMETRY: No legacy callbacks, legacy socket not created
grpc server start ...
now in server cycle
flow.rte_flow_attr
flow.rte_flow_item
flow.rte_flow_item
flow.rte_flow_item_ipv4
flow.rte_ipv4_hdr
flow.rte_flow_item_ipv4
flow.rte_ipv4_hdr
flow.rte_flow_item
flow.rte_flow_action
flow.rte_flow_action
rte_flow_attr(group=0, priority=0, ingress=1, egress=0, transfer=0, reserved=0)
[rte_flow_item(type_=9, spec=None, last=None, mask=None), rte_flow_item(type_=11,
spec=rte_flow_item_ipv4(hdr=rte_ipv4_hdr(version_ihl=0, type_of_service=0, total_length=0,
packet_id=0, fragment_offset=0, time_to_live=0, next_proto_id=0, hdr_checksum=0,
src_addr=171497737, dst_addr=0)), last=None,
mask=rte_flow_item_ipv4(hdr=rte_ipv4_hdr(version_ihl=0, type_of_service=0, total_length=0,
packet_id=0, fragment_offset=0, time_to_live=0, next_proto_id=0, hdr_checksum=0,
src_addr=4294967295, dst_addr=0))), rte_flow_item(type_=0, spec=None, last=None, mask=None)]
[rte_flow_action(type_=7, conf=None), rte_flow_action(type_=0, conf=None)]
rte_flow_attr(group=0, priority=0, ingress=1, egress=0, transfer=0, reserved=0)
1
Finish ipv4: {'hdr': {'version_ihl': 0, 'type_of_service': 0, 'total_length': 0, 'packet_id':
0, 'fragment_offset': 0, 'time_to_live': 0, 'next_proto_id': 0, 'hdr_checksum': 0, 'src_addr':
165230602, 'dst_addr': 0}}
Finish ipv4: {'hdr': {'version_ihl': 0, 'type_of_service': 0, 'total_length': 0, 'packet_id':
0, 'fragment_offset': 0, 'time_to_live': 0, 'next_proto_id': 0, 'hdr_checksum': 0, 'src_addr':
4294967295, 'dst_addr': 0}}
rte_flow_action(type_=7, conf=None)
rte_flow_action(type_=0, conf=None)
Validate ok...
flow.rte_flow_attr
flow.rte_flow_item
flow.rte_flow_item
flow.rte_flow_item_ipv4
flow.rte_ipv4_hdr
flow.rte_flow_item_ipv4
flow.rte_ipv4_hdr
flow.rte_flow_item
flow.rte_flow_action
flow.rte_flow_action
rte_flow_attr(group=0, priority=0, ingress=1, egress=0, transfer=0, reserved=0)
[rte_flow_item(type_=9, spec=None, last=None, mask=None), rte_flow_item(type_=11,
spec=rte_flow_item_ipv4(hdr=rte_ipv4_hdr(version_ihl=0, type_of_service=0, total_length=0,
packet_id=0, fragment_offset=0, time_to_live=0, next_proto_id=0, hdr_checksum=0,
src_addr=171497737, dst_addr=0)), last=None,
mask=rte_flow_item_ipv4(hdr=rte_ipv4_hdr(version_ihl=0, type_of_service=0, total_length=0,
packet_id=0, fragment_offset=0, time_to_live=0, next_proto_id=0, hdr_checksum=0,
```

```
src_addr=4294967295, dst_addr=0))), rte_flow_item(type_=0, spec=None, last=None, mask=None)]
[rte_flow_action(type_=7, conf=None), rte_flow_action(type_=0, conf=None)]
rte_flow_attr(group=0, priority=0, ingress=1, egress=0, transfer=0, reserved=0)
rte_flow_attr(group=0, priority=0, ingress=1, egress=0, transfer=0, reserved=0)
1
Finish ipv4: {'hdr': {'version_ihl': 0, 'type_of_service': 0, 'total_length': 0, 'packet_id':
0, 'fragment_offset': 0, 'time_to_live': 0, 'next_proto_id': 0, 'hdr_checksum': 0, 'src_addr':
165230602, 'dst_addr': 0}}
Finish ipv4: {'hdr': {'version_ihl': 0, 'type_of_service': 0, 'total_length': 0, 'packet_id':
0, 'fragment_offset': 0, 'time_to_live': 0, 'next_proto_id': 0, 'hdr_checksum': 0, 'src_addr':
4294967295, 'dst_addr': 0}}
rte_flow_action(type_=7, conf=None)
rte_flow_action(type_=0, conf=None)
free attr
free item ipv4
free item ipv4
free list item
free list action
Flow rule #0 created on port 0
```

6 Summary

Prior to the Intel Ethernet Operator, there was no method of cluster-wide management of Intel Ethernet Network Adapter E810 Series devices. This meant, that if an administrator wished to change the configuration of a single network adapter (such as a firmware update) on a specific node in a Kubernetes cluster, they would have to apply the changes directly to the kernel of that node. This may not be much of an issue on a small or single-node cluster, but for a large multi-node cluster, this is not a feasible approach. For example, configuring a single node as shown in the example in section 5.3.1, the resulting configuration may not be easily visible from a cluster level, and easily forgotten, which will cause difficulty when debugging issues at that perspective. The issue is compounded when the administrator is required to apply the same configuration to multiple nodes at once, or even cluster-wide, as the above method is difficult to automate and to scale.

The Intel Ethernet Operator addresses the above problems, by allowing the administrator to target specific or multiple Intel® Ethernet 800 Series Network Adapters installed on any (or multiple) nodes in the cluster and reconfigure them from within the context of the Kubernetes Cloud Native environment.

The Intel Ethernet Operator is a functional tool to manage the update of Intel® Ethernet 800 Series Network Adapters FW and DDP profile as well as for programming the VF Flow configuration autonomously in a cloud native environment based on the user input. It is easy to use by providing simple steps to apply Custom Resources to configure various aspects of the device.

0223/DN/WIT/PDF 766597-001US