Advanced Networking Features in Kubernetes* and Container Bare Metal

Application Note

December 2018

Document Number: 606835-001

Contents

Figures

Tables

Revision History

Date	Revision	Description
December 2018	001	Initial release.

1 Introduction

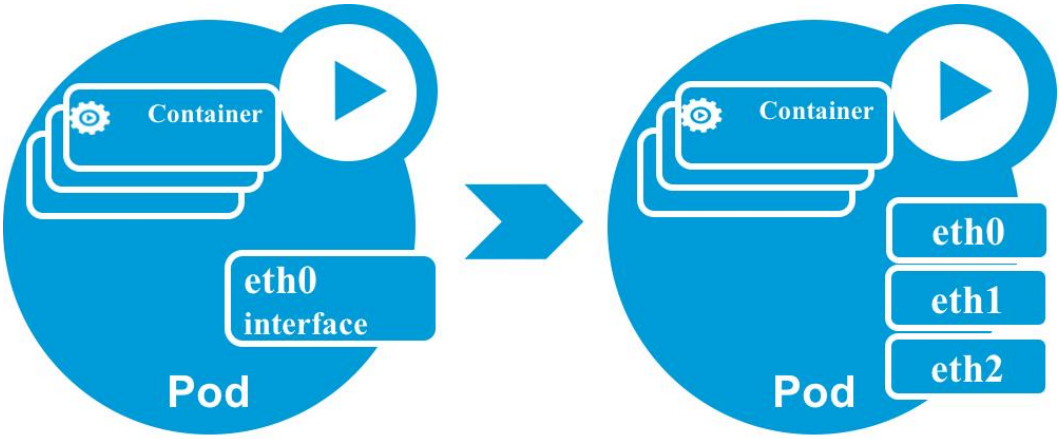
For some time, the Communications Service Provider (CommSP) industry has been moving to embrace the developments in Software Defined Networking (SDN) and Network Function Virtualization (NFV). The virtualization of the physical network functions into virtual machines (VM) and recently containers are adopted as a means to achieve greater scalability and resiliency. These new network functions are often referred to as Cloud Native Virtualized Network Functions (VNF).

Kubernetes* is the leading container orchestration engine (COE). It is an open source system for automating deployment, scaling, and management of containerized applications. Kubernetes was developed at Google* and is the anchor project in the Cloud Native Computing Foundation (CNCF), which is governed by the Linux* Foundation (LF).

While development continues in Kubernetes, one area for improvement is functionality to provide and support multiple network interfaces in VNFs. Traditionally, multiple network interfaces are employed by network functions to provide for separation of control, management and data/user network planes. They are also used to support different protocols or software stacks and different tuning and configuration requirements.

This document introduces an open source solution for Kubernetes called Multus, which addresses this need. Multus is a container network interface (CNI) plugin that can be used to create multiple network interfaces for pods in Kubernetes. A pod is a deployable unit of computing and is created and managed by Kubernetes. Multus is designed and developed to enable easier migration of current NFV use cases to a container environment. Figure 1 depicts the impact of Multus in creating multiple network interfaces for a pod.

Figure 1. Kubernetes Networking Before and After Multus



Intel is working with the Kubernetes community to develop a native Kubernetes approach for providing multiple network interfaces.

This document is designed for engineers working with Kubernetes who want to learn how to configure Multus in a virtual environment based on Virtual Network Functions (VNFs) or cloud-native network functions (CNFs). It also covers:

- How Multus can utilize single root I/O virtualization (SR-IOV) for accelerating the data/user plane throughput for high packet throughput and low processing latency.
- The open source software components required to utilize the Multus CNI plugin.

Note: This document does not describe how to set up a Kubernetes cluster. We recommend that you perform those steps as a prerequisite. For more setup and installation guidelines of a complete system, refer to the *Deploying Kubernetes and Container Bare Metal Platform for NFV Use Cases with Intel® Xeon® Scalable Processors User Guide* listed in Table 6.

This document is part of the Container Experience Kit for EPA. Container Experience Kits are collections of user guides, application notes, feature briefs, and other collateral that provide a library of best-practice documents for engineers who are developing container-based applications. Other documents in the EPA Container Experience Kit can be found at: https://networkbuilders.intel.com/network-technologies/container-experience-kits

2 Multiple Networking Interfaces using Multus

Multus is a container network interface (CNI) plugin specifically designed to provide support for multiple networking interfaces in a Kubernetes environment. CNI, a CNCF project, is a specification and supporting framework for creating plugins that create & configure network interfaces in Linux containers. Multus strictly adheres to the CNI specification described in the following link:
https://github.com/containernetworking/cni/blob/master/SPEC.md

Operationally, Multus behaves as a broker and arbiter of other CNI plugins, meaning it invokes other CNI plugins (such as Flannel, Calico, SR-IOV, or vHost CNI) to do the actual work of creating the network interfaces. When configuring Multus, you must identify one plugin as the master plugin, which is then used to configure and manage the primary network interface (eth0).

Only information from the primary network interface is returned to Kubernetes after all networking is configured for a pod. Any number of additional CNI plugins can then be used to create additional network interfaces, but Kubernetes is not informed of the details related to those interfaces. This has the consequence that, while many network interfaces have been created and can be utilized within the pod, Kubernetes is not in a position to support services or security policy, for example, on those additional network interfaces.

To understand how Multus works, it is important to review how Kubernetes networking functions operate. Kubernetes uses network plugins to orchestrate networking. Currently there are two flavors of network plugin for Kubernetes:

- **CNI plugins**: CNI plugins implement the CNI specification for interoperability of a container networking solution in a Linux environment.
- **Kubenet**: Kubenet implements a basic bridge cbr0 with host-local CNI plugin. It is also worth noting that kubenet is being deprecated, leaving CNI as the only supported framework.

The sequence diagram in Figure 2 shows the control flow for multiple network interface creation by the Multus CNI plugin. **Kubelet** is the primary agent that runs on each node in a Kubernetes cluster. Its main functions are to register the node with the Kubernetes control plane and provide lifecycle management to any pods that are subsequently scheduled to run on that node. It is also responsible for establishing the network interfaces for each pod. Kubelet does this by reading the Multus CNI configuration file and then uses these configurations to set up each pod's network. In this setup, Kubelet is configured to use CNI as its networking plugin.

When Kubelet is invoked to set up a pod, it calls its container runtime [for example, Docker* or CoreOS* Rocket (rkt)] to set up the pod. Kubelet also provides a network plugin wrapper to the container runtime to configure its network. In this case, it is the Multus CNI plugin. Multus can be used with a configuration file, with network objects, or

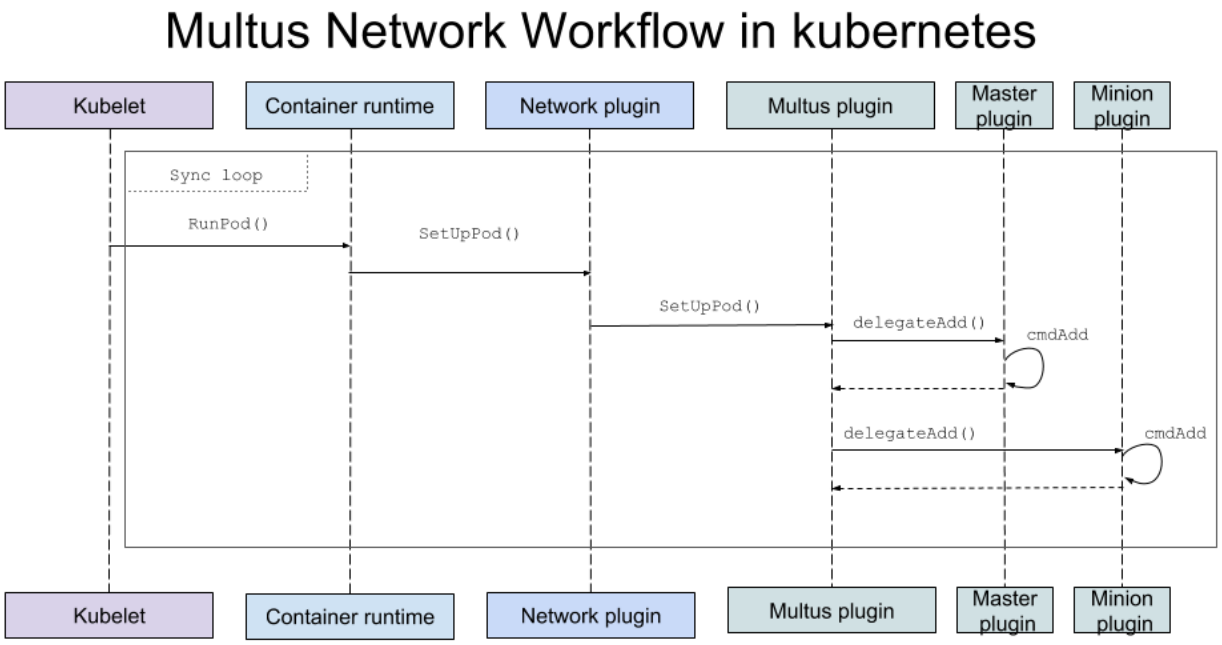
a combination of both. In any of these modes, Multus reads its configuration and offloads the actual tasks of setting up the network to other CNI plugins called as delegates. Network objects are explained in Section 4.1.

Multus then invokes **delegateAdd()** for each of these delegates (CNI plugins and their corresponding configurations). These, in turn, call their own **cmdAdd()** function to add a network interface for the pod. These delegates define which CNI plugin to be invoked and what their parameters are. The arguments of these delegate CNI plugins can be stored as CRD or TPR object in Kubernetes. (Refer to Section 4.1 for more details.)

Multus creates additional networks by taking network information that is supplied in the pod annotation. By evaluating the pod annotation, Multus will determine, which other CNI plugin should be invoked.

Note: The order of plugin invocation is important as it specifies the identity of the master plugin and that of the rest of the plugins, which are identified as minion plugins.

Figure 2. Multus Workflow in Kubernetes



Multus enables support for NFV use cases that require multiple network interfaces. Multus also permits the use of interfaces and software stacks that would otherwise not be possible in Kubernetes, such as SR-IOV and the Data Plane Development Kit (DPDK).

The next section introduces accelerating the NFV Data plane with SR-IOV and DPDK.

3 Improving Network Performance using Multus and SR-IOV/DPDK

Data Plane Development Kit (DPDK) is an open source collection of libraries and drivers that support fast-packet processing by routing packets around the OS kernel and minimizing the number of CPU cycles needed to send and receive packets. DPDK libraries include multicore framework, huge page memory, ring buffers and poll mode drivers for networking and other network functions.

SR-IOV is a PCI-SIG standardized method for isolating PCI Express* (PCIe*) native hardware resources for manageability and performance reasons. In effect, this means a single PCIe device, referred to as the physical function (PF), can appear as multiple separate PCIe devices, referred to as virtual functions (VF), with the required resource arbitration occurring in the device itself.

In the case of an SR-IOV-enabled network interface card (NIC), each VFs MAC and IP address can be independently configured and packet switching between the VFs occurs in the device hardware. The benefits of using SR-IOV networking devices in Kubernetes pods include:

- Direct communication with the NIC device allows for close to "bare-metal" performance.
- Support for multiple fast network packet processing simultaneous workloads in user space (based on DPDK, for example).
- Leveraging of NIC accelerators and offloads per workload.
- For more information, read the Intel white paper SR-IOV for NFV Solutions at: https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/sr-iov-nfv-tech-brief.pdf

Intel introduced the SR-IOV CNI plugin to allow a Kubernetes pod to be attached directly to an SR-IOV virtual function (VF) in one of two modes. The first mode uses the standard SR-IOV VF driver in the container host's kernel. The second mode supports DPDK VNFs that execute the VF driver and network protocol stack in user space.

DPDK allows the application to achieve packet processing performance that greatly exceeds the ability of the kernel network stack. For performance benchmark results, refer to *Deploying Kubernetes and Container Bare Metal Platform for NFV Use Cases with Intel® Xeon® Scalable Processors,* which is linked in Table 6.

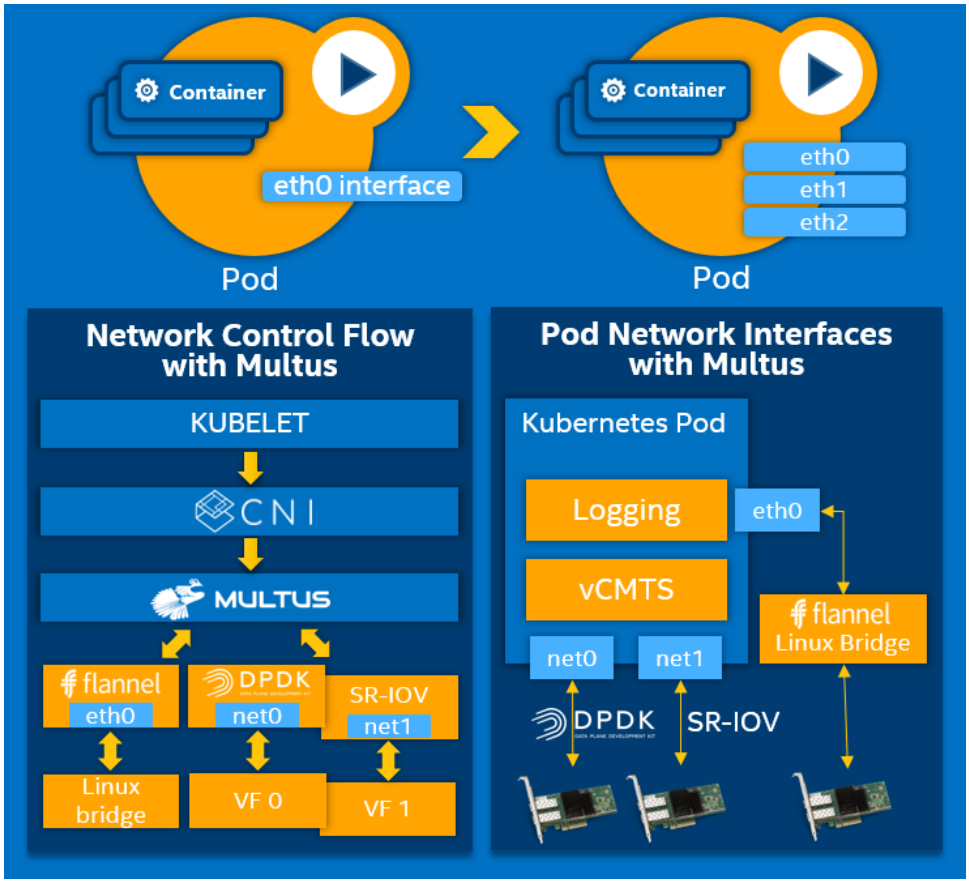
Consider the diagram in Figure 3 which shows a containerized virtual firewall with logging capability in a pod setup with three network interfaces. In this diagram, eth0 is used as the management interface for the pod, which allows that pod to communicate with any other pod within the Kubernetes cluster. eth0 is the main networking interface in Kubernetes.

In addition, two more SR-IOV VF interfaces (net0 and net1) are shown. These interfaces are created for the accelerating data plane networking. For example, a virtual firewall requires that two networks are isolated from each other using firewall rules. The VLAN technology is implemented between the virtual firewall and the 802.1Q switches and routers. The firewall recognizes VLAN IDs, and applies the firewall rules specific to each VLAN. This can include authenticating data or applying relevant policies established in the data plane network.

The advantages of this setup include:

- Logical segmentation of network
- Granular firewall rules specific to VLAN tagging
- Improved network throughput and low latency

Figure 3. Multus Networking with SR-IOV/ DPDK CNI



The next two sections detail how Multus can be configured in Kubernetes with the SR-IOV/DPDK CNI plugin.

4 Configuring Multus in Kubernetes

Multus introduces the concept of network objects. A network object represents a network into which a pod interface is attached. They are logical references for a network and they have a global scope. The Multus CNI follows the Kubernetes Network Custom Resource Definition to provide a standard method to create an additional network. These standards are created by the Kubernetes Network Plumbing Working Group (NPWG).

Network objects are considered as cluster-wide objects as they are stored in Kubernetes master registry.

This section describes the two Multus configuration options for selecting networks in Kubernetes:

- Configure Multus using network objects
- Configure Multus using configuration file

The Multus configuration using network objects allows the user to select the network per pod instead of selecting the network per node. Below are some sample scenarios that you might want to set up:

- Pod A spec with network object annotation “SRIOV” connected to SR-IOV networks with the default network.
- Pod B spec without any network object annotation, but having “Weave” as default network, connected to Weave network.

Note: Multus configured to use a configuration file works the same as any other CNI plugin.

The following section describes how to configure Multus with network object using a virtual firewall use case that involves invoking three networking interfaces as described in Figure 3. These include Flannel, SR-IOV, and SR-IOV with virtual LAN (VLAN) tagging.

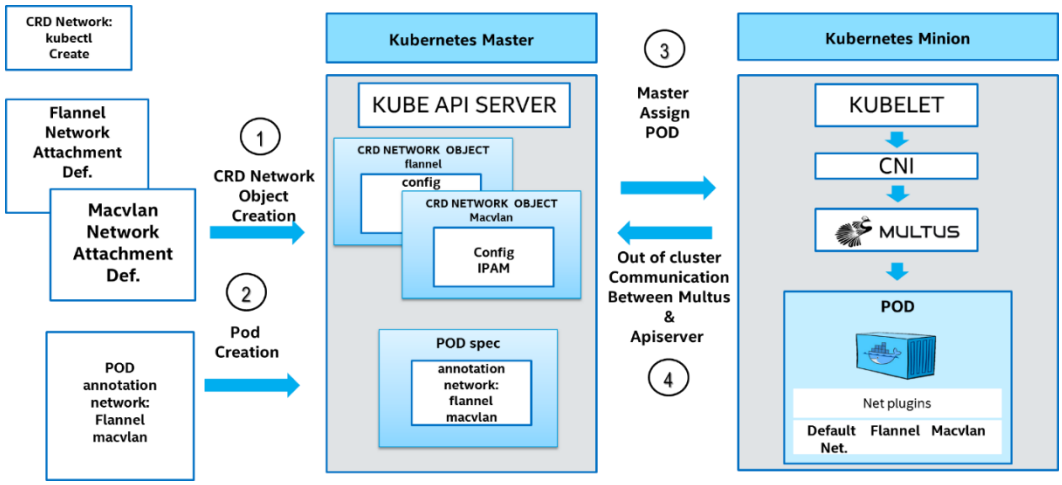
4.1 Configure Multus using Network Objects

A custom resource is an extension in Kubernetes that stores a collection of objects of a particular kind such as network objects. Custom resource definition (CRD) is a built-in feature in Kubernetes that offers a simple way to create custom resources. This mechanism provides a facility to describe a new API entity to the Kubernetes API server. CRDs provides a stable object with the introduction to new features such as pluralization of resource names and the ability to create non-namespaced CRDs.

In this section, the "network” object is created as a custom resource using CRD. Multus uses these network objects to create network interfaces in Kubernetes.

Kubelet is responsible for establishing the network interfaces for each pod; it does this by invoking the configured CNI plugin. When Multus is invoked, it recovers pod annotations related to Multus. It then uses these annotations to recover a Kubernetes CRD object. The Kubernetes CRD is an object that informs the Kubelet of which plugins to invoke and the required configurations to be passed. The identity of the primary plugin as well as the order of plugin invocation is important. The flow chart of activities required to create Multus network interfaces in Kubernetes is demonstrated in Figure 4.

Figure 4. Flow Chart to Create Multus Network Interfaces in Kubernetes



To set up multiple network interfaces, you must first define the required network objects and then create them. Instructions for defining and creating the CRD network objects are described in the following section.

4.1.1 Defining CRD-Based Network Objects

1. Create a CRD network object specification as shown in the following steps and save it as a `crdnetwork.yaml` file:

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: network-attachment-definitions.k8s.cni.cncf.io
spec:
  group: k8s.cni.cncf.io
  version: v1
  scope: Namespaced
  names:
    plural: network-attachment-definitions
    singular: network-attachment-definition
    kind: NetworkAttachmentDefinition
    shortNames:
    - net-attach-def
  validation:
    openAPIV3Schema:
```

```
      properties:
        spec:
          properties:
            config:
                 type: string
```

2. Run the following kubectl command to create the Custom Resource Definition:

```
# kubectl create -f ./crdnetwork.yaml
customresourcedefinition "network.kubernetes.com" created
```

3. Run the following kubectl command to get a command to check the Network CRD creation:

```
# kubectl get CustomResourceDefinition
NAME                                              CREATED AT
network-attachment-definitions.k8s.cni.cncf.io    2018-07-
24T09:23:43Z
```

4. Save the below following YAML to `flannel-network.yaml`:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: flannel-conf
spec:
  config: '{
    "cniVersion": "0.3.0",
    "type": "flannel",
    "delegate": {
      "isDefaultGateway": true
    }
  }'
```

5. Create the custom resource definition:

```
# kubectl create -f customCRD/flannel-network.yaml
networkattachmentdefinition.k8s.cni.cncf.io/flannel-conf created

# kubectl get net-attach-def
NAME                            AGE
flannel-conf                 1d
```

6. Get the custom network object details:

```
# kubectl get net-attach-def flannel-conf -o yaml
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  clusterName: ""
  creationTimestamp: 2018-08-01T13:46:26Z
  generation: 1
  name: flannel-conf
  namespace: default
  resourceVersion: "6926159"
```

```
  selfLink: /apis/k8s.cni.cncf.io/v1/namespaces/default/network-
attachment-definitions/flannel-conf
  uid: 4951aab8-9591-11e8-8236-408d5c537d27
spec:
  config: '{ "cniVersion": "0.3.0", "type": "flannel",
"delegate": { "isDefaultGateway":
    true } }'
```

4.1.2 How to Create Network Objects

After completing the steps in the previous section, the CRD network object definition was added to the API server. It is now possible to create the network objects. Network objects should contain the network args parameter in JSON format.

In the following example, the plugin and args fields are set to the object of kind *Network*. The object of kind *Network* is derived from the metadata.name of the CRD object that was created in the previous steps.

1. Save the following YAML to file `flannel-network.yaml`:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: flannel-conf
spec:
  config: '{
    "cniVersion": "0.3.0",
    "type": "flannel",
    "delegate": {
      "isDefaultGateway": true
    }
  }’
```

2. Run kubectl create command to create flannel-conf network object:

```
# kubectl create -f customCRD/flannel-network.yaml
networkattachmentdefinition.k8s.cni.cncf.io/flannel-conf created
```

3. Verify the network objects using kubectl:

```
# kubectl get net-attach-def
NAME                              AGE
flannel-conf                    1d
```

4. You can also view the raw JSON data. The instructions that follow show how it contains the custom plugin and args fields from the yaml that was used to create it:

```
# kubectl get net-attach-def flannel-conf -o yaml
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  clusterName: ""
  creationTimestamp: 2018-08-01T13:46:26Z
```

```
  generation: 1
  name: flannel-conf
  namespace: default
  resourceVersion: "6926159"
  selfLink: /apis/k8s.cni.cncf.io/v1/namespaces/default/network-
attachment-definitions/flannel-conf
  uid: 4951aab8-9591-11e8-8236-408d5c537d27
spec:
  config: '{ "cniVersion": "0.3.0", "type": "flannel",
"delegate": { "isDefaultGateway":
    true } }'
```

The plugin field should be the name of the CNI plugin and args should have the Flannel args, it should be in the JSON format as shown above. Network objects for Calico, Weave, Romana, and Cilium can also be created similarly.

4.1.3 Using Multus for Multiple Networks in Kubernetes

The previous section described how to create a network attachment definition in Kubernetes using CRD.

This section discusses how to use Multus for multiple networks in Kubernetes, by taking the reference workload from Figure 4.

1. Ensure the /etc/cni/net.d file uses the `00-multus.conf` file as shown below.

```
{
      "name": "defaultnetwork",
      "type": "multus",
      "LogLevel": "debug",
      "LogFile": "/var/log/multus.log",
      "kubeconfig": "/etc/kubernetes/node-kubeconfig.yaml",
      "delegates": [{
            "cniVersion": "0.3.0",
            "name": "defaultnetwork",
            "type": "flannel",
            "isDefaultGateway": true
      }],
      "confDir": "/etc/cni/multus/net.d"
}
```

2. Save the following YAML to `sriov-network.yaml`:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: sriov-with-ipam
spec:
  config: '{
    "cniVersion": "0.3.0",
    "type": "sriov",
    "if0": "enp12s0f1",
```

```
    "ipam": {
        "type": "host-local",
        "subnet": "10.56.217.0/24",
        "rangeStart": "10.56.217.171",
        "rangeEnd": "10.56.217.181",
        "routes": [
            { "dst": "0.0.0.0/0" }
            ],
            "gateway": "10.56.217.1"
        }
    }
  }'
```

3. Save the following YAML to file `sriov-vlanid-l2enable-network.yaml`:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: sriov-vlanid-l2enable-conf
spec:
  config: '{
    "cniVersion": "0.3.0",
    "type": "sriov",
    "if0": "enp2s0",
    "vlan": 210,
    "l2enable": true
  }'
```

4. Complete the following two steps to create the network object “sriov-vlanid-l2enable-conf” and “sriov-conf”:

```
# kubectl create -f ./sriov-vlanid-l2enable-network.yaml
network "sriov-vlanid-l2enable-conf" created

# kubectl create -f ./sriov-network.yaml
network "sriov-conf" created
```

5. Verify the network objects using kubectl:

```
# kubectl get net-attach-def
NAME                          AGE
flannel-conf                  1d
sriov-vlanid-l2enable-conf    1d
sriov-conf                    1d
```

At this stage, the network resources have been configured and created. The next step is to deploy pods using these network resources.

4.1.4 Deploy Pods with Multiple Interfaces

1. Create a sample pod specification `pod-multi-network.yaml` file with the following contents. Here, flannel-conf network object acts as the primary network:

```
# cat pod-multi-network.yaml

apiVersion: v1
kind: Pod
metadata:
  name: multus-multi-net-pod
          annotations:
    k8s.v1.cni.cncf.io/networks: '[
            { "name": "sriov-conf " },
            { "name": " sriov-vlanid-l2enable-conf ",
              "interface": "north" }
    ]'

spec:  # specification of the pod's contents
  containers:
  - name: multus-multi-net-pod
    image: "busybox"
    command: ["top"]
    stdin: true
    tty: true
```

2. Create multiple network-based pods from the master node:

```
# kubectl create -f ./pod-multi-network.yaml

pod "multus-multi-net-pod" created
```

3. Retrieve the details of the running pod from the master:

```
# kubectl get pods

NAME                    READY     STATUS    RESTARTS   AGE
multus-multi-net-pod    1/1       Running   0          30s
```

4.2 Configure Multus using Configuration File

Multus can be configured to read the network configuration from its configuration file instead of network objects. In this case, all pods in the node will have the same network interface.

4.2.1 Multus Configuration Parameters

Multus accepts the configuration parameters in JSON format described in Table 1. Certain parameters are required and thus need to be provided by the user when configuring a Kubernetes minion node. For more information, refer to the Multus Readme file.

Table 1. Multus Configuration Parameters

Parameter name	Type	Required	Description
Name	string	Yes	The name of the network
type	string	Yes	"multus"
kubeconfig	string	No	kubeconfig file for the out of cluster communication with kube-apiserver
delegates	[]map	Yes	Delegate objects (underlying CNI plugin configuration). This is ignored if kubeconfig is added.

4.3 Verifying Pod Networks

Once the configuration is complete, you can verify the pod networks are working as expected. The pod created with the specification defined in Section 4.1.3 should have created three interfaces with the provided configurations. To verify these configurations, complete the following steps:

1. Run the `ifconfig` command inside the container:

```
# kubectl exec -it multus-multi-net-poc - ifconfig

eth0      Link encap:Ethernet  HWaddr 06:21:91:2D:74:B9
          inet addr:192.168.42.3  Bcast:0.0.0.0
Mask:255.255.255.0
          inet6 addr: fe80::421:91ff:fe2d:74b9/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

net0      Link encap:Ethernet  HWaddr D2:94:98:82:00:00
          inet addr:10.56.217.171  Bcast:0.0.0.0
Mask:255.255.255.0
          inet6 addr: fe80::d094:98ff:fe82:0/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:120 (120.0 B)  TX bytes:648 (648.0 B)
```

```
north     Link encap:Ethernet  HWaddr BE:F2:48:42:83:12
          inet6 addr: fe80::bcf2:48ff:fe42:8312/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1420 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1276 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:95956 (93.7 KiB)  TX bytes:82200 (80.2 KiB)
```

As shown in the output screen above, there are three interfaces created along with a loopback interface.

The description of the pod interfaces can be found in the following table:

Interface name	Description
lo	Loopback
eth0@if41	Flannel network tap interface
net0	VF0 of NIC 1 assigned to the container by SR-IOV CNI plugin
north	VF0 of NIC 2 assigned with VLAN ID 210 to the container by SR-IOV CNI plugin

2. Verify that the VLAN ID of the VF that is assigned from the SR-IOV NIC matches the VLAN tag given in file `sriov-vlanid-l2enable-network.yaml` file in Section 4.1.2. This is confirmed by checking the SR-IOV NIC information using IPROUTE2 utility. The VLAN ID is highlighted in the sample below:

```
# ip link show enp2s0

20: enp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq
state UP mode DEFAULT group default qlen 1000
    link/ether 24:8a:07:e8:7d:40 brd ff:ff:ff:ff:ff:ff
    vf 0 MAC 00:00:00:00:00:00, vlan 210, spoof checking off,
link-state auto
    vf 1 MAC 00:00:00:00:00:00, vlan 4095, spoof checking off,
link-state auto
    vf 2 MAC 00:00:00:00:00:00, vlan 4095, spoof checking off,
link-state auto
    vf 3 MAC 00:00:00:00:00:00, vlan 4095, spoof checking off,
link-state auto
```

After completing the steps above, a Kubernetes pod should be configured with three network interfaces: “eth0”, “net0” and “north” along with a loopback interface “lo”. In the above instruction, the multi networking in Kubernetes is showcased with network object creation via the Multus CNI plugin. Refer to the github link for more information: https://github.com/Intel-Corp/multus-cni/

5 Orchestrating SR-IOV Network Interfaces in Kubernetes

5.1 Overview

SR-IOV enabled network interface cards (NICs) allow sharing of a physical NIC port transparently amongst many VNFs in many virtual environments. Each VF can be assigned to one container, and configured with separate MAC, VLAN, and IP.

The SR-IOV network device plugin along with Multus and SR-IOV CNI plugin enable Kubernetes pods to attach to an SR-IOV VF. The SR-IOV network device plugin discovers and registers available SR-IOV capable VFs in host machine to Kubernetes API as extended resources. Pods that require one or more SR-IOV VFs will request for these VFs in their specification. Kubelet takes care of resource allocation and accounting of all VFs that registered with the SR-IOV network device plugin.

To apply specific network configurations to these VFs, a pod may add network attachment annotations. This annotation simply refers to a CRD object in API that describes CNI plugin and its configurations. The Multus plugin as explained in previous section parses this CRD, then delegates to other CNI plugin to configure an interface in the pod. To configure SR-IOV VF, a network CRD must be created with SRIOV-CNI as its delegate plugin. Multus then provides allocated VF information of the pod to the SRIOV CNI.

5.2 Prepare SR-IOV Network Resources

5.2.1 Hardware

- Intel® Ethernet Converged Network Adapter X710 (codename Fortville) SFP+ 4x10G Quad-port

5.2.2 Software

- Operating Systems
  - CentOS* Linux* 7
  - Kernel version 4.13.7-1.el7.elrepo.x86_64 (Updated)
- Network driver
  - Intel® Ethernet Converged Network Adapter X710 or XL710 Linux Drivers for PF and VF
  - https://downloadcenter.intel.com/product/82947/Intel-Ethernet-Controller-X710-Series

- Download link: https://downloadcenter.intel.com/download/24411/Intel-Network-Adapter-Driver-for-PCIe-40-Gigabit-Ethernet-Network-Connections-Under-Linux-?product=24586

5.2.3 Enable IOMMU Support

I/O Memory Management Unit (IOMMU) support is not enabled by default in the CentOS* 7.0 distribution. IOMMU support is required for a VF to function properly when assigned to a Virtual environment, such as a VM or a Container. The following kernel boot parameter is required to enable IOMMU support for Linux kernels:

```
intel_iommu=on
```

Append to the **GRUB_CMDLINE_LINUX** entry in `/etc/default/grub` configuration file.

```
$ cat /etc/default/grub
GRUB_TIMEOUT=5
GRUB_DEFAULT=0
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="crashkernel=auto nomodeset rhgb quiet
intel_iommu=on iommu=pt hugepages=4096 pci=realloc pci=assign-
busses"
GRUB_DISABLE_RECOVERY="true"
```

Update grub configuration using `grub2-mkconfig`

```
$ grub2-mkconfig -o /boot/grub2/grub.cfg
```

Reboot the machine for the IOMMU change to take effect.

5.2.4 Create Virtual Functions in CentOS* 7

In Linux Kernel version 3.8.x and above, VF can be created by writing an appropriate value to the `sriov_numvfs` parameter via sysfs interface as shown below:

```
$ echo 4 > /sys/bus/pci/devices/0000\:02\:00.0/sriov_numvfs
$ echo 4 > /sys/bus/pci/devices/0000\:02\:00.1/sriov_numvfs
$ echo 4 > /sys/bus/pci/devices/0000\:02\:00.2/sriov_numvfs
$ echo 4 > /sys/bus/pci/devices/0000\:02\:00.3/sriov_numvfs
```

In this example, we created 4 VFs for each PFs using their PCI address.

Note: The number of VFs created using the sysfs interface are not persistent from one boot to the next. To ensure that the desired number of VFs are created each time the server is power-cycled, we recommend that you create your own systemd service or udev rules to run a script during system boot to create the desired number of VF.

5.2.5 Create VF with User Space Driver

A PF can be registered with user space driver as well, for example, a DPDK Kernel module (igb_uio). You can create VF for PF that are registered with igb_uio driver as follows:

```
$ echo 4 > /sys/bus/pci/devices/0000\:02\:00.0/max_vfs
```

5.2.6 VFs with Kernel Network Stack

By default, when VFs are created for Intel® X710 NIC, the VFs are registered with i40evf kernel module. VF with Linux network driver uses the kernel network stack for all packet processing.

5.2.7 VFs with DPDK Network Stack

To take advantage of user space packet processing with DPDK, a VF needs to be registered with either igb_uio or vfio-pci kernel module. The example shown below binds a VF with PC address 03:02.0 with vfio-pci module using dpdk-devbind.py script bundled with DPDK distribution.

```
$ $RTE_SDK/dpdk-devbind.py -b vfio-pci 03:02.0
```

Note: $RTE_SDK is the path to the DPDK source code. For more information, refer to the Linux Drivers section in the *DPDK User Guide* here.

5.3 Deploy SR-IOV Network Device Plugin

5.3.1 Build Docker* Image

```
$ git clone https://github.com/intel/sriov-network-device-
plugin.git
$ cd sriov-network-device-plugin
$ make image
```

Once the image is built successfully, a Docker image with tag ***nfvpe/sriov-device-plugin*** will be available in the machine where the build was run. To verify that the Docker image is created, run the following command:

```
$ docker images | grep nfvpe
nfvpe/sriov-device-plugin   latest              4e0489ee413d
13 seconds ago      503MB
```

5.3.2 Create Host Configuration File

The SR-IOV Network device plugin creates device plugin endpoints based on the configurations given in file ***/etc/pcidp/config.json***. This configuration file is in json format as shown below:

```
$ cat <<EOF > /etc/pcidp/config.json
{
    "resourceList":
    [
        {
            "resourceName": "sriov_net_A",
            "rootDevices": ["02:00.0", "02:00.2"],
            "sriovMode": true,
            "deviceType": "netdevice"
        },
        {
            "resourceName": "sriov_net_B",
            "rootDevices": ["02:00.1", "02:00.3"],
            "sriovMode": true,
            "deviceType": "vfio"
        }
    ]
}
EOF
```

The "resourceList" should contain a list of config objects. Each config object describe a resource pool for the device plugin to discover and export appropriate device information for Kubernetes.

In this example configuration above, we are creating two resource pools, sriov_net_A and sriov_net_B. The sriov_net_A consists of all VFs under the PFs with PCI address "02:00.0" and "02:00.2". The "deviceType" parameter describes the type of driver all of the VFs registered for this resource pool.

Supported configuration parameters are described in the following table.

Table 2. SR-IOV Network Device Plugin Configuration Parameters

Field	Required	Description	Type - Accepted values	Example
"resourceName"	Yes	Endpoint resource name	string - must be unique and should not contain special characters	"sriov_net_A"
"rootDevices"	Yes	List of PCI address for a resource pool	A list of string - in sysfs pci address format	["02:00.0", "02:00.2"]
"sriovMode"	No	Whether the root devices are SR-IOV capable or not	bool - true OR false[default]	true
"deviceType"	No	Device driver type	string - "netdevice"|"uio"|"vfio"	"netdevice"

5.3.3 Deploy as Kubernetes Daemonset

The SR-IOV Network device plugin can be deployed as a Kubernetes Daemonset using the specification file available at https://github.com/intel/sriov-network-device-plugin/blob/master/images/sriovdp-daemonset.yaml.

```
$ kubectl create -f images/sriovdp-daemonset.yaml
serviceaccount/sriov-device-plugin created
daemonset.extensions/kube-sriov-device-plugin-amd64 created
$ kubectl get pods --all-namespaces
NAMESPACE     NAME                                    READY
STATUS    RESTARTS   AGE
kube-system   kube-sriov-device-plugin-amd64-s5r6x   1/1
Running   0          10s
```

Once the Daemonset is running, we can verify that the configured VF resources are listed for the Kubernetes node:

```
$ kubectl describe nodes <node_name>
.
.
.
Capacity:
 cpu:                    8
 ephemeral-storage:      184447308Ki
 hugepages-1Gi:          0
 hugepages-2Mi:          8Gi
 intel.com/sriov_net_A:  8
 intel.com/sriov_net_B:  8
 memory:                 16371628Ki
 pods:                   1k
Allocatable:
 cpu:                    8
 ephemeral-storage:      169986638772
 hugepages-1Gi:          0
 hugepages-2Mi:          8Gi
 intel.com/sriov_net_A:  8
 intel.com/sriov_net_B:  8
 memory:                 7880620Ki
 pods:                   1k
.
.
.
```

5.4 Network Configurations

To apply network configuration to the VF in Kubernetes pod, we need the CNI meta plugin Multus to get allocated VF information of a pod. Multus passes this information to SRIOV-CNI, which takes care of moving the VF interface to the pod network namespaces, and applies any other appropriate configurations. The example

deployment directory, located in https://github.com/intel/sriov-network-device-plugin/tree/master/deployments, contains sample deployment specification files used in the following examples.

5.4.1 Multus and CRDs

We are assuming the Kubernetes cluster has been configured and deployed with Multus already.

5.4.2 Install SR-IOV CNI Plugin

```
$ git clone https://github.com/intel/sriov-cni.git
$ cd sriov-cni
$ make
$ cp ./build/sriov /opt/cni/bin/
```

For further information on configuring and deploying sriov-cni, please refer to the README.md in the sriov-cni repository.

5.4.3 SR-IOV Network Attachment CRD

The network attachment CRD specification:

```
$ cat <<EOF > sriov-net-a.yaml
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: sriov-net-a
  annotations:
    k8s.v1.cni.cncf.io/resourceName: intel.com/sriov_net_A
spec:
  config: '{
  "type": "sriov",
  "vlan": 1000,
  "ipam": {
    "type": "host-local",
    "subnet": "10.56.217.0/24",
    "rangeStart": "10.56.217.171",
    "rangeEnd": "10.56.217.181",
    "routes": [{
      "dst": "0.0.0.0/0"
    }],
    "gateway": "10.56.217.1"
  }
}'
EOF
```

Create the CRD:

```
$ kubectl create -f sriov-net-a.yaml
```

A Pod specification requesting SR-IOV network VF:

```
$ cat <<EOF > pod-tc1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: testpod1
  labels:
    env: test
  annotations:
    k8s.v1.cni.cncf.io/networks: sriov-net-a
spec:
  containers:
  - name: appcntr1
    image: centos/tools
    imagePullPolicy: IfNotPresent
    command: [ "/bin/bash", "-c", "--" ]
    args: [ "while true; do sleep 300000; done;" ]
    resources:
      requests:
        intel.com/sriov net A: '1'
      limits:
        intel.com/sriov_net_A: '1'
  restartPolicy: "Never"
EOF
```

Deploy the pod:

```
$ kubectl -f pod-tc1.yaml
```

5.5 Verify

```
$ kubectl get pods
NAME                    READY     STATUS    RESTARTS   AGE
testpod1              1/1       Running   0          3s
```

Verify that VF interfaces are configured with the settings described in the SR-IOV-CRD:

```
$ kubectl exec -it testpod1 -- ip addr show

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state
UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
```

```
3: eth0@if17511: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc
noqueue state UP
    link/ether 0a:58:c0:a8:4a:b1 brd ff:ff:ff:ff:ff:ff link-
netnsid 0
    inet 192.168.74.177/24 scope global eth0
       valid_lft forever preferred_lft forever
17508: net0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc
mq state DOWN qlen 1000
    link/ether ce:d8:06:08:e6:3f brd ff:ff:ff:ff:ff:ff
    inet 10.56.217.179/24 scope global net0
       valid_lft forever preferred_lft forever
```

6 Orchestrating Userspace CNI in Kubernetes

6.1 Overview

The Userspace CNI is a Container Network Interface (CNI) plugin designed to implement userspace networking (as opposed to kernel space networking), such as DPDK based applications. The current implementation supports OVS-DPDK or VPP vSwitches along with the Multus CNI plugin in Kubernetes for the bare metal container deployment model. It enhances high performance container Networking solution and Dataplane Acceleration for NFV Environment.

Userspace networking requires additional considerations. For one, the interface needs to be created/configured on a local vSwitch (running on the host). There may also be a desire to add the interface to a specfic network on the host through the local vSwitch. Second, when the interface is inserted into the container, it is not owned by the kernel, so additional work needs to be done in the container to consume the interface and add to a network within the container. The Userspace CNI is designed to work with these additional considerations by provisioning the local vSwitch (for example, OVS-DPDK/VPP), and by pushing data into the container so the interface can be consumed.

6.2 Prepare Userspace CNI

6.2.1 Software

- Operating System
  - CentOS* Linux* 7
  - Kernel version 3.10.0-862.14.4.el7.x86_64
- Data Plane Development Kit (DPDK) 17.11.3
- Open vSwitch* 2.10.90
- Vector Packet Processing (VPP) 18.07

6.2.2 Build Userspace CNI

1. Download and build Userspace CNI:

```
$ cd $GOPATH/src/
$ go get github.com/intel/userspace-cni-network-plugin
$ cd github.com/intel/userspace-cni-network-plugin
$ make install
```

2. Copy the newly built userspace binary to the CNI directory and copy get-prefix.sh script from userspace-cni-network-plugin repo to /var/lib/cni/vhostuser/:

```
$ cp userspace/userspace /opt/cni/bin
$ cp tests/get-prefix.sh /var/lib/cni/vhostuser/
```

6.3 Use Userspace CNI with Open vSwitch* with DPDK

6.3.1 Start and Configure Open vSwitch*

1. Start OVS database server:

```
$ ovs-ctl --no-ovs-vswitchd start
```

2. Configure OVS dpdk-socket-mem. Note this example is on a dual socket system and allocates 2048MB to each socket. Adjust to suit your system:

```
$ ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-
socket-mem="2048,2048"
```

3. Enable support for DPDK ports:

```
$ ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-
init=true
```

4. Specify the CPU cores where dpdk lcore threads should be spawned:

```
$ ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-lcore-
mask=0xc
```

5. Set the CPU affinity of Poll Mode Driver threads:

```
$ ovs-vsctl --no-wait set Open_vSwitch . other_config:pmd-cpu-
mask=0x3
```

6. Start OVS daemon:

```
$ ovs-ctl --no-ovsdb-server --db-
sock="/usr/local/var/run/openvswitch/db.sock" restart
```

7. Add OVS bridge br0:

```
$ ovs-vsctl add-br br0 -- set bridge br0 datapath_type=netdev
```

6.3.2 Create Userspace Network Object

```
$ cd $GOPATH/src/github.com/intel/userspace-cni-network-
plugin/examples
$ kubectl create -f crd-userspace-net-ovs-no-ipam.yaml
```

6.3.3 Docker Images

1. Create two separate directories for your Dockerfiles:

```
$ mkdir dpdk-l3fwd dpdk-pktgen
```

2. Create the l3fwd Dockerfile:

```
$ cat <<EOF > dpdk-l3fwd/Dockerfile
```

```
FROM ubuntu:bionic

RUN apt-get update && apt-get install -y --no-install-recommends
\\
  build-essential make wget vim dpdk libnuma-dev \\
  && apt-get clean && rm -rf /var/lib/apt/lists/*

ENV DPDK_VERSION=17.11.3 \\
  RTE_SDK=/usr/src/dpdk \\
  RTE_TARGET=x86_64-native-linuxapp-gcc

RUN wget http://fast.dpdk.org/rel/dpdk-\${DPDK_VERSION}.tar.xz &&
tar xf dpdk-\${DPDK_VERSION}.tar.xz && \\
  mv dpdk-stable-\${DPDK_VERSION} \${RTE_SDK}

RUN sed -i s/CONFIG_RTE_EAL_IGB_UIO=y/CONFIG_RTE_EAL_IGB_UIO=n/
\${RTE_SDK}/config/common_linuxapp \\
  && sed -i s/CONFIG_RTE_LIBRTE_KNI=y/CONFIG_RTE_LIBRTE_KNI=n/
\${RTE_SDK}/config/common_linuxapp \\
  && sed -i s/CONFIG_RTE_KNI_KMOD=y/CONFIG_RTE_KNI_KMOD=n/
\${RTE_SDK}/config/common_linuxapp

RUN cd \${RTE_SDK} && make install T=\${RTE_TARGET} && make -C
examples

WORKDIR \${RTE_SDK}/examples/l3fwd/\${RTE_TARGET}/app/
EOF
```

3. Build the l3fwd Docker image:

```
$ docker build -t dpdk-l3fwd ./dpdk-l3fwd
```

4. Create the pktgen Dockerfile:

```
$ cat <<EOF > dpdk-pktgen/Dockerfile
FROM debian:jessie

RUN apt-get update && apt-get install -y --no-install-recommends
\\
  gcc build-essential make wget curl git liblua5.2-dev libedit-
dev libpcap-dev libncurses5-dev libncursesw5-dev pkg-config vim
libnuma-dev ca-certificates \\
  && apt-get clean && rm -rf /var/lib/apt/lists/*

ENV DPDK_VERSION=17.11.3 \\
  RTE_SDK=/usr/src/dpdk \\
  RTE_TARGET=x86_64-native-linuxapp-gcc \\
  PKTGEN_COMMIT=pktgen-3.4.8 \\
  PKTGEN_DIR=/usr/src/pktgen

RUN wget http://fast.dpdk.org/rel/dpdk-\${DPDK_VERSION}.tar.xz &&
tar xf dpdk-\${DPDK_VERSION}.tar.xz && \\
  mv dpdk-stable-\${DPDK_VERSION} \${RTE_SDK}
```

```
RUN sed -i s/CONFIG_RTE_EAL_IGB_UIO=y/CONFIG_RTE_EAL_IGB_UIO=n/
\${RTE_SDK}/config/common_linuxapp \\
  && sed -i s/CONFIG_RTE_LIBRTE_KNI=y/CONFIG_RTE_LIBRTE_KNI=n/
\${RTE_SDK}/config/common_linuxapp \\
  && sed -i s/CONFIG_RTE_KNI_KMOD=y/CONFIG_RTE_KNI_KMOD=n/
\${RTE_SDK}/config/common_linuxapp

RUN sed -i s/CONFIG_RTE_APP_TEST=y/CONFIG_RTE_APP_TEST=n/
\${RTE_SDK}/config/common_linuxapp \\
  && sed -i s/CONFIG_RTE_TEST_PMD=y/CONFIG_RTE_TEST_PMD=n/
\${RTE_SDK}/config/common_linuxapp

RUN cd \${RTE_SDK} \\
  && make install T=\${RTE_TARGET} DESTDIR=install -j

RUN git config --global user.email "root@container" \\
  && git config --global user.name "root"

RUN git clone http://dpdk.org/git/apps/pktgen-dpdk \\
  && cd pktgen-dpdk \\
  && git checkout \${PKTGEN_COMMIT} \\
  && cd .. \\
  && mv pktgen-dpdk \${PKTGEN_DIR}

RUN cd \${PKTGEN_DIR} \\
  && make -j

WORKDIR \${PKTGEN_DIR}/
EOF
```

5. Build the pktgen Docker image:

```
$ docker build -t dpdk-pktgen ./dpdk-pktgen
```

6.3.4 Pod Spec

1. Create the l3fwd Pod Spec:

```
$ cat <<EOF > dpdk-l3fwd/pod.yaml
apiVersion: v1
kind: Pod
metadata:
  generateName: dpdk-l3fwd-
  annotations:
    k8s.v1.cni.cncf.io/networks: userspace-networkobj
spec:
  containers:
  - name: dpdk-l3fwd
    image: dpdk-l3fwd
    imagePullPolicy: IfNotPresent
    securityContext:
      privileged: true
      runAsUser: 0
```

```
    volumeMounts:
    - mountPath: /vhu/
      name: socket
    - mountPath: /dev/hugepages
      name: hugepage
    resources:
      requests:
        memory: 2Gi
      limits:
        hugepages-1Gi: 2Gi
    command: ["sleep", "infinity"]
  volumes:
  - name: socket
    hostPath:
      path: /var/lib/cni/vhostuser/
  - name: hugepage
    emptyDir:
      medium: HugePages
  securityContext:
    runAsUser: 0
  restartPolicy: Never
EOF
```

2. Create the pktgen Pod Spec:

```
$ cat <<EOF > dpdk-pktgen/pod.yaml
apiVersion: v1
kind: Pod
metadata:
  generateName: dpdk-pktgen-
  annotations:
    k8s.v1.cni.cncf.io/networks: userspace-networkobj
spec:
  containers:
  - name: dpdk-pktgen
    image: dpdk-pktgen
    imagePullPolicy: IfNotPresent
    securityContext:
      privileged: true
      runAsUser: 0
    volumeMounts:
    - mountPath: /vhu/
      name: socket
    - mountPath: /dev/hugepages
      name: hugepage
    resources:
      requests:
        memory: 2Gi
      limits:
        hugepages-1Gi: 2Gi
    command: ["sleep", "infinity"]
  volumes:
  - name: socket
```

```
    hostPath:
      path: /var/lib/cni/vhostuser/
  - name: hugepage
    emptyDir:
      medium: HugePages
  securityContext:
    runAsUser: 0
  restartPolicy: Never
EOF
```

6.3.5 Deploy

1. Create both pods:

```
$ kubectl create -f dpdk-l3fwd/pod.yaml
$ kubectl create -f dpdk-pktgen/pod.yaml
```

2. Verify your pods are running and note their unique name IDs:

```
$ kubectl get pods
NAME                 READY   STATUS    RESTARTS   AGE
dpdk-l3fwd-vj4hj     1/1     Running   0          51s
dpdk-pktgen-xrsz9    1/1     Running   0          2s
```

6.3.6 Verify

1. Using your pod ID, open a bash shell in the pktgen pod:

```
$ kubectl exec -it dpdk-pktgen-<ID> bash
```

2. Export the port ID prefix and start the pktgen application (adjust cores, memory, etc. to suit your system):

```
$ export ID=$(/vhu/get-prefix.sh)
$ ./app/x86_64-native-linuxapp-gcc/pktgen -l 10,11,12 --
vdev=virtio_user0,path=/vhu/${ID}/${ID:0:12}-net1 --no-pci --
socket-mem=1024,1024 --master-lcore 10 -- -m 11:12.0 -P
```

3. The pktgen application should launch. Among the statistics, make note of the pktgen source MAC address listed as 'Src MAC Address'.

```
Src MAC Address    :    f2:89:22:4e:28:3b
```

4. In another terminal, open a bash shell in the l3fwd pod:

```
$ kubectl exec -it dpdk-l3fwd-<ID> bash
```

5. Export the port ID prefix and start the l3fwd application. Set the destination MAC address using the 'dest' argument. This should be the Src MAC Address previously noted from the pktgen pod (adjust cores, memory, etc. to suit your system):

```
$ export ID=$(/vhu/get-prefix.sh)
```

```
$ ./l3fwd -c 0x10 --vdev=virtio_user0,path=/vhu/${ID}/${ID:0:12}-
net1 --no-pci --socket-mem=1024,1024 -- -p 0x1 -P --config
"(0,0,4)" --eth-dest=0,<pktgen-source-mac-add> --parse-ptype
```

6. The l3fwd app should start up. Among the information printed to the screen will be the 'Address'. This is the MAC address of the l3fwd port, make note of it.
7. Back on the pktgen pod, set the destination MAC address to that of the l3fwd port:

```
Pktgen:/> set 0 dst mac <l3fwd-mac-address>
```

8. Start traffic generation:

```
Pktgen:/> start 0
```

9. You should see the packet counts for Tx and Rx increase, verifying that packets are being transmitted by pktgen and are being sent back via l3fwd running in the other pod.
10. To exit:

```
Pktgen:/> stop 0
Pktgen:/> quit
```

6.4 Use Userspace CNI with VPP

6.4.1 Start VPP on the Host

Install VPP 18.07 on the host and run it:

```
# systemctl start vpp
```

6.4.2 Create Network Attachment Definitions

Create network attachment definition specification file for the Flannel control plane network:

```
cat <<EOF > flannel-network.yaml
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: flannel
spec:
  config: '{
    "cniVersion": "0.3.0",
    "name": "cbr0",
    "type": "flannel",
    "delegate": {
      "hairpinMode": true,
      "isDefaultGateway": true
    }
```

```
  }'
EOF
```

Create Network Attachment Definition specification file for VPP memif userspace network:

```
cat <<EOF > vpp-network.yaml
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: vpp
spec:
  config: '{
      "cniVersion": "0.3.1",
      "type": "userspace",
      "name": "memif-network",
      "host": {
          "engine": "vpp",
          "iftype": "memif",
          "netType": "bridge",
          "memif": {
              "role": "master",
              "mode": "ethernet"
          },
          "bridge": {
              "bridgeId": 4
          }
      },
      "container": {
          "engine": "vpp",
          "iftype": "memif",
          "netType": "interface",
          "memif": {
              "role": "slave",
              "mode": "ethernet"
          }
      },
      "ipam": {
          "type": "host-local",
          "subnet": "172.22.0.0/24",
          "rangeStart": "172.22.0.2",
          "rangeEnd": "172.22.0.254",
          "routes": [
              {
                  "dst": "0.0.0.0/0"
              }
          ],
          "gateway": "172.22.0.1"
      }
    }'
EOF
```

6.4.3 Create VPP Client Application Pod Specification

Create VPP client application pod specification file by executing the command:

```
cat <<EOF > vpp-client.yaml
apiVersion: v1
kind: Pod
metadata:
  generateName: vpp-client-
  annotations:
    k8s.v1.cni.cncf.io/networks: flannel,vpp
spec:
  containers:
  - name: vpp-client
    image: bmcfall/vpp-centos-userspace-cni:0.4.0
    volumeMounts:
    - mountPath: /hugepages
      name: hugepage
    - mountPath: /var/run/vpp/cni
      name: vpp-cni
    command: ["tail", "-f", "/dev/null"]
    securityContext:
      privileged: true
      runAsUser: 0
    resources:
      requests:
        memory: 1Gi
      limits:
        hugepages-2Mi: 1Gi
        memory: 1Gi
  volumes:
  - name: hugepage
    emptyDir:
      medium: HugePages
  - name: vpp-cni
    hostPath:
      path: /var/run/vpp/cni
  securityContext:
    runAsUser: 0
  restartPolicy: Never
EOF
```

6.4.4 Deploy

Deploy network attachment definitions created in previous steps. Execute this command:

```
kubectl create -f flannel-network.yaml -f vpp-network.yaml
```

Deploy 2 pods using previously created pod specification. Execute the following command (the same file is used two times intentionally):

```
kubectl create -f vpp-client.yaml -f vpp-client.yaml
```

6.4.5 Verify

Verify that pods are in running state:

```
# kubectl get pods
```

Example output:

```
NAME                                READY   STATUS      RESTARTS
AGE
vpp-client-6msjm                    1/1     Running     0
1m
vpp-client-kz2p6                    1/1     Running     0
1m
```

Note: Access to 2 terminal windows is recommended for the next steps.

In the first terminal window execute below command to attach to the first vpp-client pod:

```
# kubectl exec -it vpp-client-<ID1>
```

For example:

```
# kubectl exec -it vpp-client-6msjm
```

From the container shell, execute:

```
# /usr/bin/vpp -c /etc/vpp/startup.conf &> vppBoot.log &
```

Wait a couple of seconds for the VPP to start, then execute the following command to show list of default interfaces:

```
# vppctl show interface
```

Execute the commands below to add memif slave interface to the VPP instance running in the pod:

```
# mkdir -vp /var/run/vpp/cni/data
# cp /var/run/vpp/cni/$(sed -ne '/hostname/p'
/proc/1/task/1/mountinfo | awk -F '/' '{print $6}')/*
/var/run/vpp/cni/data/
# vpp-app
```

Repeat the above steps for the second pod. In another terminal window, run:

```
# kubectl exec -it vpp-client-<ID2>
# /usr/bin/vpp -c /etc/vpp/startup.conf &> vppBoot.log &
# mkdir -vp /var/run/vpp/cni/data
# cp /var/run/vpp/cni/$(sed -ne '/hostname/p'
/proc/1/task/1/mountinfo | awk -F '/' '{print $6}')/*
/var/run/vpp/cni/data/
# vpp-app
```

Now, as memif slave interfaces are properly added to the VPP instances running inside the pods, it's time to verify the connectivity between them.

In the shell of pod 1, execute:

```
# vppctl show int addr
```

Capture IP address from the output, for example:

```
local0 (dn):
memif1/0 (up):
  L3 172.22.0.32/24
```

In the second pod, execute VPP ping command, for example:

```
# vppctl ping 172.22.0.32
```

If everything is correctly working, you should see the following output:

```
64 bytes from 172.22.0.32: icmp_seq=1 ttl=64 time=110.9806 ms
64 bytes from 172.22.0.32: icmp_seq=2 ttl=64 time=72.1675 ms
64 bytes from 172.22.0.32: icmp_seq=3 ttl=64 time=95.4605 ms
64 bytes from 172.22.0.32: icmp_seq=4 ttl=64 time=96.2225 ms
64 bytes from 172.22.0.32: icmp_seq=5 ttl=64 time=128.2909 ms

Statistics: 5 sent, 5 received, 0% packet loss
```

This confirms that connection between two pods using VPP userspace memif interfaces works fine.

7 Conclusion

Having multiple network interfaces in a Kubernetes pod is an essential feature for many VNF applications. This can be accomplished today through the use of the Multus CNI and the other open source software components described in this document. Additionally, the availability of multiple network interfaces makes improved network throughput for container-based applications possible through the use of interfaces to SR-IOV and DPDK. Intel has contributed to the development of these technologies as part of its support for virtualized computing.

For more information on what Intel is doing with containers, go to https://networkbuilders.intel.com/network-technologies/intel-container-experience-kits

Appendix A References and Terminology

This appendix includes:

- Tables that show the hardware platform and software referenced in this document.
- Summary of the acronyms used in this document
- Links to reference documents.

A.1 Hardware

Table 3. Hardware

Item	Description	Notes
Platform	Intel® Server Board S2600WFQ	Intel® Xeon® processor-based dual-processor server board with 2 x 10 GbE integrated LAN ports
Processor	2x Intel® Xeon® Gold Processor 6138T	2.0 GHz; 125 W; 27.5 MB cache per processor 20 cores, 40 hyper-threaded cores per processor
Memory	192GB Total; Micron* MTA36ASF2G72PZ	12x16GB DDR4 2133MHz 16GB per channel, 6 Channels per socket
NIC	Intel® Ethernet Converged Network Adapter X710 (4x10G)	4 x 10 GbE ports Firmware version 5.50
Storage	Intel DC P3700 SSDPE2MD800G4	SSDPE2MD800G4 800 GB SSD 2.5in NVMe/PCIe
BIOS	Intel Corporation SE5C620.86B.0X.01.0007.060920171037 Release Date: 06/09/2017	Hyper-Threading - Enable Boot performance Mode – Max Performance Energy Efficient Turbo – Disabled Turbo Mode - Disabled C State - Disabled P State - Disabled Intel VT-x Enabled Intel VT-d Enabled

A.2 Software

Table 4. Software

Software Component	Description	References
Host Operating System	Ubuntu* 16.04.2 x86_64 (Server) Kernel: 4.4.0-62-generic	https://www.ubuntu.com/download/server
NIC Kernel Drivers	i40e v2.0.30 i40evf v2.0.30	https://sourceforge.net/projects/e1000/files/i40e%20stable
Data Plane Development Kit	DPDK 17.05	http://fast.dpdk.org/rel/dpdk-17.05.tar.xz
CPU Manager for Kubernetes*	v1.1.0 & v1.2.1	https://github.com/Intel-Corp/CPU-Manager-for-Kubernetes
Ansible*	Ansible 2.3.1.0	https://github.com/ansible/ansible/releases
Bare Metal Container Environment Setup scripts	Includes Ansible* scripts to deploy Kubernetes v1.6.6 & 1.8.4	https://github.com/intel/container-experience-kits
Docker*	v1.13.1	https://docs.docker.com/engine/installation/
SR-IOV Network device plugin	V2.0.0 Commit SHA: f01659ef33aee4eb262687669f861bffced740d9	https://github.com/intel/sriov-network-device-plugin
SRIOV-CNI	v0.2-alpha. commit ID: a2b6a7e03d8da456f3848a96c6832e6aefc968a6	https://www.ubuntu.com/download/server

A.3 Terminology

Table 5. Terminology

Term	Definition
CNCF	Cloud Native Computing Foundation
CNI	Container Network Interface
COE	Container Orchestration Engine
CRD	Custom Resource Definition
DPDK	Data Plane Development Kit
IPAM	IP Address Management
IPSec	Encryption Protocol for IP Networks

Term	Definition
LF	Linux Foundation
NFD	Node Feature Discovery
NFV	Network Functions Virtualization
NIC	Network Interface Card
PCIe	Peripheral Component Interconnect Express
PF	Physical Function
Pod	A group of one or more containers in Kubernetes
SDN	Software Defined Network
SLA	Service Level Agreement
SR-IOV	Single-Root Input/Output Virtualization
STDIN	Standard input
TPR	Third Party Resource
VETH	Virtual Ethernet interface
VF	Virtual Function
VLAN	Virtual LAN
VM	Virtual Machine
VNF	Virtual Network Function
VPP	Vector Packet Processing

A.4 Reference Documents

Table 6. Reference Documents

Document	Document No./Location
Deploying Kubernetes and Container Bare Metal Platform for NFV Use Cases with Intel® Xeon® Scalable Processors	https://builders.intel.com/docs/networkbuilders/deploying-kubernetes-and-container-bare-metal-platform-for-nfv-use-cases-with-intel-xeon-scalable-processors-user-guide.pdf
NFV Reference Design for A Containerized vEPC application	https://builders.intel.com/docs/networkbuilders/nfv-reference-design-for-a-containerized-vepc-application.pdf
Understanding CNI (Container Networking Interface)	http://www.dasblinkenlichten.com/understanding-cni-container-networking-interface/
EPA Container Experience Kit Documents	https://networkbuilders.intel.com/network-technologies/container-experience-kits