WHITE PAPER

Intel Corporation
Data Center Solution Networking Group

Enabling New Features with Kubernetes for NFV

Authors

Muhammad Siddiqui
Solution Software Engineer

Tarek Radi
Lead Solution Enabling Manager

Lukasz Obuchowicz
Solution Software Engineer

Pawel Rutkowski
Solution Software Engineer

Table of Contents

1.0 Introduction

Network Function Virtualization (NFV) is gaining traction in enterprises and Communication Service Providers (CSPs) as it offers faster service enablement and programmability in deploying applications which require consistency, determinism and predictability. Traditionally, the common virtualization method uses virtual machines (VM), wherein applications run on an operating system's hypervisor. However, running multiple guest Operating Systems (OSs) on top of the hypervisor is not always the most efficient use of resources. Using container technology is an alternate approach to virtualization which mitigates some of the inefficiencies of VM based virtualization.

Software containers provide several benefits over VMs. For example, they are virtualized at the OS level which can make them quick to deploy with no processing overhead on the guest OS. Software applications can be packaged into container images, which have smaller footprints and better portability than VMs.

The nature of containers allows the deployment of microservices, where each part of a service is decoupled into a separate container to provide modular development, easy deployment and scaling models. In many cases, a microservices approach can be suitable when building cloud infrastructures, especially those that consist of cloud applications that add more functionality or features over time.

Kubernetes (K8s) is an open source container manager and orchestrator that was originally designed by Google, but then donated to the Cloud Native Computing Foundation. It automates the deployment, scaling and operational functions associated with application containers, and does so across a cluster of hosts (physical machines). The open source nature of the Kubernetes project allows the community to contribute and improve upon it.

Kubernetes is evolving quickly, but still lacks a number of features that are important for the management and performance of container-based virtual network functions (VNF) in an NFV environment. To help address this, Intel Corporation recently made public four new repositories containing features that are compatible plugins to Kubernetes:

1) Multus Container Networking Interface (CNI) Plugin (available on GitHub. See [6])

2) Single Root I/O virtualization (SR-IOV) CNI Plugin (available on GitHub. See [9])

3) Node Feature Discovery (NFD) (available on GitHub. See [10])

4) CPU Core Manager for Kubernetes (available on GitHUb. See[11])

See sections 2.3 and 3 for more information about why these features were implemented and a summary of what they do. This document provides a summary of each of these new features and shows how they can be used in a NFV deployment. The techniques presented here can potentially be used for any containerized NFV use-case. For more information about use-cases that Kubernetes enables, see [13].

2.0 Kubernetes Overview, Limitations and Improvements

2.1 Kubernetes Components

The following general diagram shows a typical K8s deployment. This section summarizes the key components in Figure 1 for context. For more details, see [14]

Containers create an isolation boundary at the application level for portability and ease of packaging. Docker is a container runtime technology that manages these containers along with all their dependencies and the libraries required to successfully run these containers. This allows running a software application in a container that is isolated from the host machine. A Docker container is another form of virtualization compared to VMs. For a comparison of Containers vs VMs, see the diagrams at [15].

Kubernetes is an open-source platform that automates the deployment, management, and scaling of application containers across clusters of servers (hosts), providing container-centric infrastructure [1]. A Kubernetes cluster consists of one or more masters, and one or more nodes. A Kubernetes cluster can achieve High Availability (HA) of the infrastructure when it has multiple masters.

Kubernetes Master is the main controlling unit of the cluster that manages and schedules pods on to the worker nodes. The main components of K8s master are:

- **The API server:** this services REST operations and provides the frontend to the cluster's shared state through which all other components interact. [2]
- **etcd:** this is a distributed key-value data store that reliably stores the configuration data of the cluster and represents the overall state of the Kubernetes cluster.
- **The scheduler:** this decides the target node on to which a pod would be scheduled, and makes that decision based on the available resources.
- **The controller manager** communicates with the API server to create, update and delete the resources they manage e.g. pods, service etc.

A **Kubernetes node,** also referred to as worker or minion, is the machine where pods are deployed. The node runs the services that are necessary to run application containers and be managed by one or multiple Kubernetes masters. The main components of a Kubernetes node are:

- **Container runtime:** Docker and Rocket are examples of a container runtime.
- **Kubelet:** this is an agent that is responsible for registering a node to the Kubernetes cluster. It provides the Kubernetes Master with the running state of the Kubernetes node it is running on. It takes care of creating and deleting pods, and ensures the health of the application containers.
- **The kube-proxy** is an implementation of a network proxy and reflects services as defined in the Kubernetes API on each node. It can do simple Transport Control Protocol (TCP), User Datagram Protocol (UDP) stream forwarding across a set of backends [17].

A **Kubernetes pod** represents a unit of deployment. A pod consists of either a single container or a small number of containers that are tightly coupled and share resources [3]. Each pod is assigned a unique IP address which is routable within a Kubernetes cluster. Every container in a pod shares the network namespace, including the IP address and network ports. If two containers should not share such networking components, then they should be designed into different pods. To learn more about Kubernetes, visit https://kubernetes.io

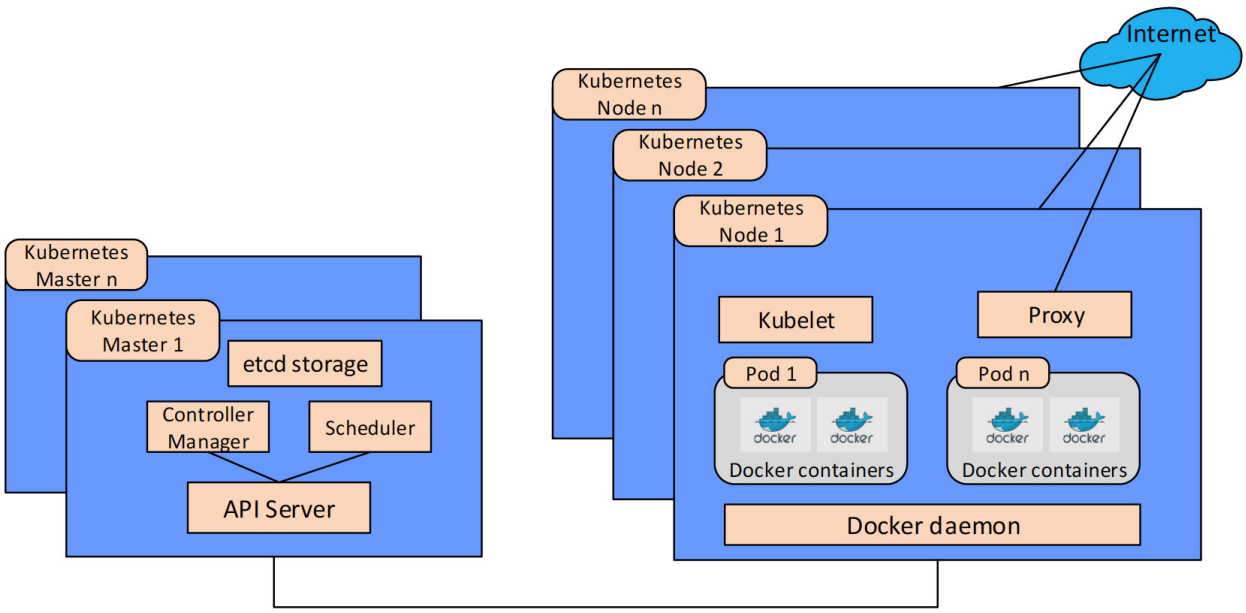

Figure 1. Generic Kubernetes Cluster

2.2 Container Networking

By default, **Docker** provides its own Linux-bridge (docker0). This bridge network is automatically created when the Docker is first installed. The Docker daemon connects containers to this bridge network. In our setup we are not using the default networking of Docker. Instead, we use the Kubernetes **Container Networking Interface** (CNI). See [18] for more information on CNI.

The CNI is a specification and a set of libraries for a plugin-based networking solution for containers. This networking solution has been adopted by various open source orchestration engines including Kubernetes. CNI concerns itself only with the network connectivity of containers and removing allocated resources when the containers are deleted [4]. It is written in Go programming language and supports several 3rd party plugins.

One of the commonly used CNI plugin is the **flannel plugin**; this provides basic overlay networking (unique and routable IP) for containers in a Kubernetes cluster. It gives a subnet to each host for use with container runtimes [5].

2.3 How Intel enhanced Kubernetes for NFV

In a typical NFV deployment, VNFs are usually connected to multiple network interfaces in order to a) provide VNFs with redundancy of the network, and b) segregate the control plane from the data plane traffic. One of the major current limitation in the CNI is the inability to connect more than one network interface to a K8s pod. This limitation does not facilitate the deployment of VNFs in a K8s environment. In this document, we introduce the **Multus CNI plugin** which resolves this limitation, allowing containerized VNFs in Kubernetes to have more than one network interface. For more details, see Section 3.1.

Another current challenge for a container-based virtual network function (VNF) solution is the lack of support for Single root I/O virtualization (SR-IOV) Virtual Function (VF) [7]. SR-IOV is a technology that is currently widely used in VM-based VNFs due to the increased performance it provides. We make SR-IOV possible for containers in CNI by implementing and deploying an **SR-IOV CNI plugin**. This plugin enables a Kubernetes pod to attach directly to an SR-IOV VF. The plugin also provides pods with support to bind the VF to a DPDK driver. All this provides containers with direct access to the network hardware and results in higher performance. For more details, see Section 3.2.

Additionally, at the time of writing this document, Kubernetes was unable to identify which nodes have high performance networking hardware such as SR-IOV capable Network Interface Cards (NICs). We solve this problem using **Node Feature Discovery (NFD)**, and show how the Kubernetes scheduler can schedule pods that have a requirement for high I/O traffic to nodes where the SR-IOV VFs are available. This helps ensure that hardware features within a Kubernetes cluster are utilized in a more efficient way. NFD is a run-once K8s job that detects hardware features that are available at the Kubernetes node level. Kubernetes can use this information for scheduling containerized VNFs such that they get deployed on nodes with the desirable hardware features. For more details, see Section 3.3.

Finally, as we have seen in non-containerized NFV deployments, performance can often be optimized when CPU pinning is enabled. CPU pinning is a key Enhanced Platform Awareness (EPA) feature in OpenStack. EPA facilitates better decision making related to VM placement and helps drive tangible improvements for Cloud tenants [16]. More details about EPA can be found in [8]. Currently, Kubernetes does not support CPU pinning. In this document, we show how K8s container-based VNFs can be affinitized to dedicated CPU cores using **CPU Core Manager for Kubernetes**. The CPU Core Manager for Kubernetes manages pools of CPU cores and constrains workloads to specific CPU cores within those pool. This is a highly desirable feature in an NFV environment for VNFs that have low latency and high performance requirements. For more details, see Section 3.4. requirements.

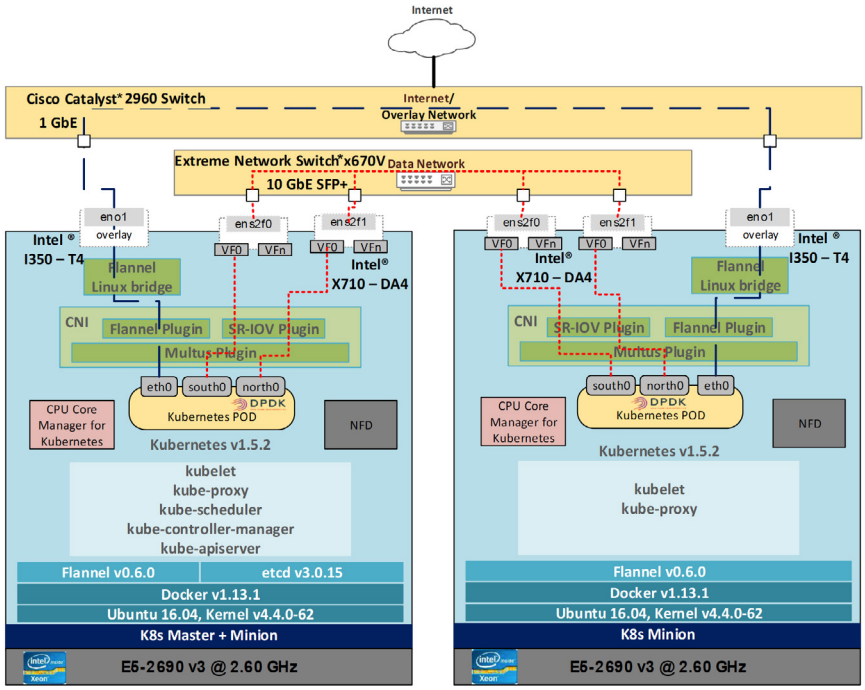

Figure 2. Setup

3.0 Setup

The Network Function Virtualization Infrastructure (NFVI) used to demonstrate these new features in K8s and CNI consists of two industry-standard, high volume server Intel® Xeon® CPU E5-2690 v3 @ 2.60 GHz processor-based servers running the Ubuntu 16.04 Desktop OS. See Appendix A for details about the hardware. Hyper-Threading and virtualization is enabled on both host machines. Docker v1.13.1 is used as the container run-time and Kubernetes v1.5.2 is installed on these servers to provide a container orchestration platform. One server is configured as the Kubernetes master plus minion, whereas the second server is configured as minion node only. Figure 2 shows the physical topology and the software stack used to develop and test the new features presented in this document. From that figure, one can notice different networks. Table 1 summarizes the 3 networks that we used and which interfaces are connected to each network.

3.1 Feature 1: Multus CNI Plugin

Multus is the Latin word for “Multi”. As the name suggests, it allows K8s pods to be multi-homed. The Multus CNI plugin can be used in conjunction with other CNI plugins e.g. ptp, local-host, calico, flannel etc. The Multus CNI plugin can also work with different internet protocol address management (IPAM) configurations and networks. It groups multiple plugins into delegates and invokes each plugin in sequential order according to the CNI configuration file. It calls each plugin used in the Multus CNI file to do the network configuration of its interface [6].

The left part of Figure 3 shows a pod without the Multus CNI plugin. When the Multus CNI plugin is enabled, a ‘masterplugin’ gets instantiated. See the right side of the figure. This masterplugin is responsible for managing the "eth0" interface of the pod. In other words, the masterplugin identifies the primary network and sets the default route via this network. This is the only network configuration option of the multus plugin.The other plugins are referred to as minion plugins, and they are responsible for invoking other interfaces, e.g. ‘net0’, ‘net1’ [6]. Naturally, there can only be one masterplugin, and this is specified in the Multus config file using the tag "masterplugin": true. as shown in the sample Multus configuration file in Appendix C.

The Multus workflow begins in the Kubelet. A properly configured Kubelet knows the CNI plugin’s directory. CNI processes the plugin configuration file in lexicographical order. The Kubelet calls CNI to create network interfaces for the newly created containers. Furthermore, the CNI calls the Multus masterplugin which then invokes the other minion CNI plugins sequentially to create the desired number of interfaces in the pod. One of these minion CNI plugins could be flannel, IPAM, or the new SR-IOV CNI Plugin defined in the following section.

Network	Network Description	Network Interface Card	Interface name on the Host
External/Internet	Provides Internet/remote access to the host machines and the K8s pods	Intel® Ethernet Server Adapter I350-T4V2	eno1
Overlay Network (VxLAN)	Virtual extensible local area network (VxLAN) used as an overlay network for Kubernetes pods	Intel® Ethernet Server Adapter I350-T4v2	eno1
Data Network	Internal network used for data traffic using SR-IOV VFs.	Intel® Ethernet Converged Network Adapter X710-DA4	ens2f0 / ens2f1

Table 1. Networks used in this setup

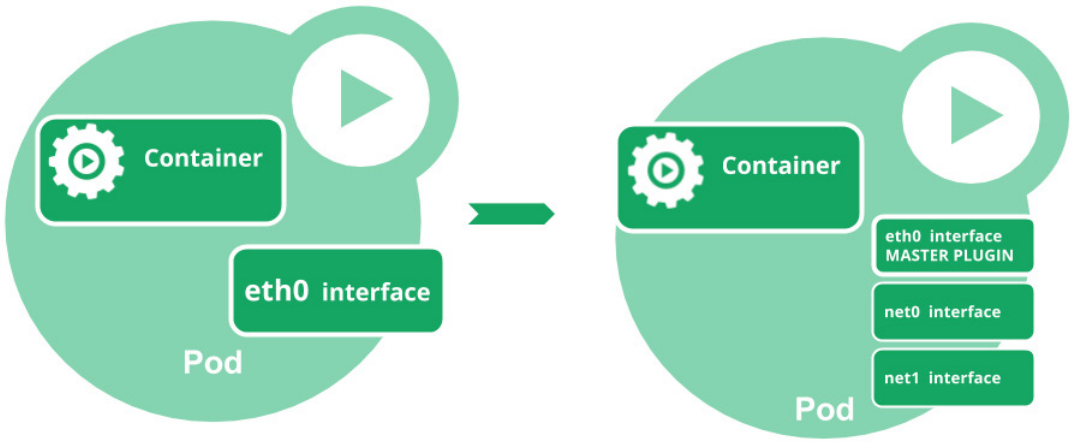

Figure 3. Before and after adding a Multus Plugin to the CNI (Image Source here)

3.2 Feature 2: SR-IOV CNI Plugin

The SR-IOV CNI plugin enables the K8s pods to attach to an SR-IOV VF [9]. The plugin looks for the first available VF on the designated port in the multus configuration file, based on the VF index number, and assigns that VF to the container. The plugin also supports the Data Plane Development Kit (DPDK) driver, i.e. vfio-pci, for these VFs. When this driver is specified in the multus configuration file, the plugin can automatically unbind the VF from its kernel driver, then associate it to the pod and bind it to the DPDK driver in user space. When the pod is deleted the VF will be bound back to the kernel driver. In the case of the X710-DA4 Network Interface Card (NIC), this would be the **'i40evf'**.

Note: SR-IOV VFs can be used with the DPDK driver to provide high performance networking interfaces to the K8s pods. These DPDK interfaces can then be used to run DPDK applications.

SR-IOV plugin workflow starts after CNI namespace is created for the new pod. The configured VF is moved to the new CNI namespace. The plugin sets the interface name as indicated in the 'name' configuration option in the CNI config file. Finally, the VF state is set to UP.

To demonstrate both the Multus and SR-IOV CNI plugins, we create a pod with three interfaces as shown in Figure 4 below. See Appendix C for the Multus Configuration file that results in this setup.

The Multus Config file in Appendix C specifies the following three interfaces:

1. **Flannel interface.** This interface was created using the existing flannel plugin, and has the 'masterplugin' option set to true. In this example, this flannel interface was associated with the **'eth0'** interface of the pod. This is the interface used to reach the default gateway.
2. **SR-IOV VF interface using kernel driver.** In this example, this VF was instantiated from the host machine's physical port **'ens2f0'**. This was the first port on the Intel® X710-DA4 NIC. The name of the VF interface inside the pod was **'south0'**. An IP address can be assigned to this interface using IPAM.
3. **SR-IOV VF interface using DPDK driver.** In this example, this VF was instantiated from the host machine's physical port **'ens2f1'**. This was the second port on the Intel® X710-DA4 NIC. The name of the VF interface inside the pod was **'north0'**. The interface was bound to the DPDK driver **'vfio-pci'**. If the pod ever gets deleted, the VF will be recycled and bound back to its kernel driver, in this case **'i40evf'**.

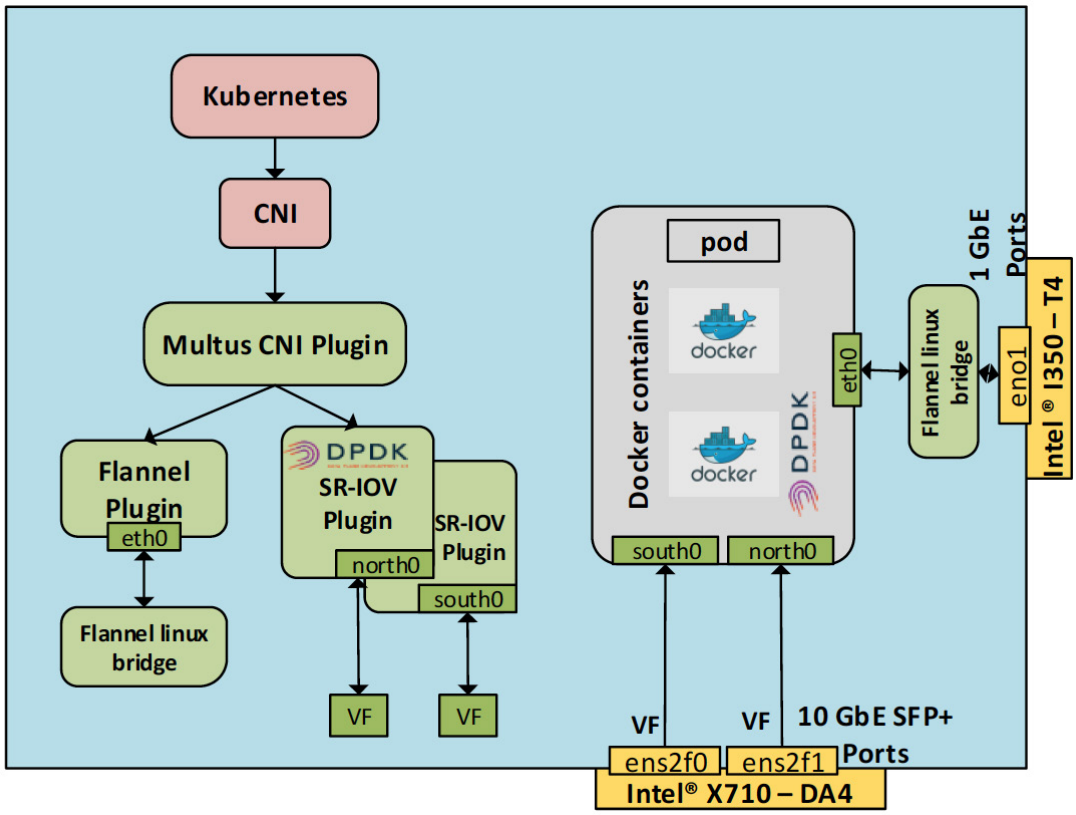

Figure 4. Mutus plugin allowing for 2 types of plugins per pod: Flannel and SR-IOV plugins

```
#!/bin/bash
# Getting the pci-addr from the container json file
#ifname name should match with what's used in the multus configuration
  file

ifname="north0"
cid="$(sed -ne '/hostname/p' /proc/1/task/1/mountinfo | awk -F '/'
  '{print $6}')"
cid="$cid-$ifname"
export PCIADDR="$(awk -F '"' '{print $4}' /sriov-cni/$cid)"

# echo $PCIADDR
0000:0b:06.0
```

After the pod is created, the user can use the above script to get the PCI address of the DPDK interface. The PCI address can then be used to run a DPDK-based application. The script requires the name of the DPDK interface inside the pod, which in our case was **'north0'**.

3.3 Feature 3: Node Feature Discovery (NFD)

Node Feature Discovery (NFD) is a Kubernetes project that is part of Kubernetes Incubator [10]. NFD detects hardware features available on each node in a Kubernetes cluster, and advertises those features using node labels. Feature discovery is done as a one-shot job. The node feature discovery script launches a job which deploys a single pod on each unlabeled node in the cluster. When each pods run, it contacts the Kubernetes API server to add labels to the node. Currently the NFD can detect the following four features in the node: 1) cpuid, 2) Intel® Resource Director (RDT), 3) p-state and 4) network (to detect SR-IOV VFs on the node).

Labels are key/value pairs that are attached to objects, such as pods or nodes. Labels are used to specify identifying attributes of objects that may be relevant to the end user. They can be used to organize objects into specific subsets. Labels are a part of the metadata information that is attached to each node's description [12]. All this information is kept within **etcd**.

The NFD project creates new labels and tags each node with information about its hardware features. The published node labels encode a few pieces of information:

- A namespace,
- The source for each label,
- The discovered feature name for the underlying source,
- The version of this discovery code that wrote the label

Labels generated by NFD can be checked from the master node using the following command:

```
# kubectl get nodes -o json | jq .items[].metadata.labels
{
  "node.alpha.kubernetes-incubator.io/node-feature-discovery.version":
  "a9af7ff-dirty",
  "node.alpha.kubernetes-incubator.io/nfd-pstate-turbo": "true",
  "node.alpha.kubernetes-incubator.io/nfd-network-sriov": "true",
  "node.alpha.kubernetes-incubator.io/nfd-network-sriov-configured":
  "true",
  "node.alpha.kubernetes-incubator.io/nfd-cpuid-SSSE3": "true",
...
```

The explanation of the network labels is as follows:

- **.../nfd-network-sriov": "true"** – the node has at least 1 network interface with SR-IOV capabilities
- **.../nfd-network-sriov-configured": "true"** – the node has at least one network interface with SR-IOV VFs configured and that interface is in UP state.

The Kubernetes scheduler can now use the information contained in each node labels to deploy pods according to the requirements specified in the pod specification. Appendix E shows a sample pod specification that uses the **'nodeSelector'** option with the NFD created label to deploy a K8s pod to a node which has at least one SR-IOV VF available. If the scheduler does not find a single node in the cluster that has the required SR-IOV VFs, then the creation of that pod will fail.

3.4 Feature 4: CPU Core Manager for Kubernetes

CPU Core Manager for Kubernetes is a tool for managing CPU core pinning and isolation. It is a command-line program that performs various functions, such as host configuration, managing groups of CPU cores and constraining workloads to specific CPU cores.

CPU Core Manager for Kubernetes creates core isolation through applying CPU masks. These masks represent cores on which the workload can be executed. The state of each core (e.g. allocated or free) is maintained through the host file system which incorporates a system lock to avoid any conflicts. This core state is structured as a directory hierarchy where pools are represented as directories. Appendix F shows an example of the directory hierarchy created by CPU Core Manager for Kubernetes. Workloads can acquire slots from these directory structures. These slots represent physical allocable cores in the form of a list of their logical core IDs. In this instance, a pool is a representation of a named group of CPU core lists.

CPU Core Manager for Kubernetes creates three pools: 1) a data plane pool, 2) a control plane pool, and 3) an infra pool. The data plane pool is exclusive whereas the control plane and infra pools are shared. When there is no pool mentioned in the pod specification, the CPU Core Manager for Kubernetes will use cores from the infra pool. It uses Opaque Integer Resources (OIR) to advertise the number of slots on the relative Kubernetes node. The number of slots advertised is equal to the number of CPU lists available under the data plane pool.

CPU Core Manager for Kubernetes has an "isolate" sub-command. This can be referenced in the pod specification in the arguments such that the pod consumes one of the available CPUs from a specified pool. In case that the data plane pool is specified, the OIR created in the CPU Core Manager for Kubernetes command will be consumed. Exactly one OIR will be consumed per container, this ensures the correct number of containers are allowed to run on a node. In the case of a shared pool any CPU may be selected regardless of the current process allocations. The isolate command will fail in the case where an exclusive pool is requested and there are no available CPUs left in that pool.

CPU Core Manager for Kubernetes isolates requested workloads to a core in the specified pool, however this does not prevent system tasks from running on that core. The use of the Linux kernel parameter 'isolcpus' is the recommended way to ensure that cores are isolated from system tasks. CPU Core Manager for Kubernetes reads the value of 'isolcpus' in the file located at `/proc/cmdline`. These isolated CPUs are then used in the creation of the dataplane and controlplane pools. All other unisolated CPUs are used for the infra pool. If isolcpus is not set, the CPU Core Manager for Kubernetes arbitrarily chooses CPUs to assign to the pools.

In this setup, we used K8s nodes which had 2 CPU sockets each. Figure 5 in the Appendix B shows the CPU configuration of those K8s nodes. Each CPU socket has 12 physical cores. Intel® Hyper-Threading (HT) technology was enabled, and that's why that figure shows 24 logical cores for each NUMA node. At the time this document was written, the CPU Core Manager for Kubernetes could only work with single socket systems. For this setup, we overcame this limitation by disabling all cores on NUMA node 1 (in other words, we disabled the 2nd CPU socket). The first 12 logical cores in NUMA node 0 (i.e. 0, 2, 4 up to 22) are the physical cores. The next 12 logical cores in NUMA node 0 (i.e. 24, 26, 28 up to 46) are the respective hyper-thread (HT) siblings of the first 12 physical cores. In this setup, we isolated four physical cores and their corresponding hyper- thread silbings from NUMA node 0 using 'isolcpus' parameter in the `/etc/default/grub` file as shown below.

```
.. isolcpus=8,32,10,34,12,36,14,38"
```

Note: By default the CPU Core Manager for Kubernetes uses the following 2 directories:

1) The **`/opt/bin/kcm`** *directory to install all of its binaries*

2) The **`/etc/kcm`** *directory to store configuration files*

These are required when deploying a pod that uses the CPU Core Manager for Kubernetes.

The CPU Core Manager for Kubernetes is then initialized to allocate 3 cores plus their HT siblings to the data plane pool, and 1 core and its HT sibling to the control plane pool. The rest of the unisolated cores on NUMA node 0 were used for the infra pool. The following two flags were used while initializing CPU Core Manager for Kubernetes to allocate the desired number of CPU cores to the data plane and control plane pools.

```
--num-dp-cores=<num>: number of physical cores to assign to the
dataplane pool.
--num-cp-cores=<num>: number of physical cores to assign to the
controlpane pool.
```

After the CPU Core Manager for Kubernetes is initialized, it deploys **'NodeReport'** pods on each node. This pod reports the allocation of CPU cores to all three pools. A sample report showing all three pools and their allocated CPU cores is shown in the Appendix D. We then proceeded with running our application inside a container using the 'isolate' sub-command and specify a pool name. In the example below we chose to run the **memtester** application and used the dataplane pool so that the memtester application is affinitized to use exclusive CPU cores.

```
[…]
spec:
  containers:
  - args:
    - "apt update && apt install memtester -y && /opt/bin/kcm isolate --
   conf-dir=/etc/kcm --pool=dataplane memtester -- 100M > /dev/null"
```

```
[…]
```

In this example, one core and its HT sibling were consumed to run the memtester application. The process ID (PID) of the application was written to the tasks file of the used dataplane core.

Note: The CPU Core Manager for Kubernetes does not treat HT siblings as separate cores. Instead, processes are pinned to the main physical core PLUS its HT sibling.

The following output from NodeReport shows that task #40386 (which is PID of the memtester application) has been assigned to the cores '10' and '34' which are HT siblings and belong to the dataplane pool. After such an assignment, these cores are no longer available for use by any other container until the current process is completed. When the memtester process terminated, the CPU Core Manager for Kubernetes program removed the process ID from the tasks file and made it available for use by other processes.

```
# kubectl get NodeReport minionnode -o json
...
"dataplane": {
                    "cpuLists": {
                        "10,34": {
                            "cpus": "10,34",
                            "tasks": [
                                40386
                            ]
                        },
...
```

4.0 Summary and Next Steps

The four new features summarized in this whitepaper are key contributions by Intel that facilitate a container-based NFV deployment. We highly recommend you consider them, at least for your proof of concepts.

1. Using the **Multus CNI plugin** you can enable containerized VNFs in Kubernetes to have more than one network interface, which is usually a fundamental requirement for VNFs, for network redundancy and the ability to segregate control plane from data plane traffic.
2. Using the **SR-IOV CNI plugin** enables Kubernetes pods to attach directly to an SR-IOV virtual function (VF), giving containerized VNFs high performance networking capabilities.
3. Using **Node Feature Discovery (NFD)** you can ensure that hardware features within a Kubernetes cluster are utilized in a more efficient way, and enable Kubernetes to use such information for scheduling containerized VNFs so that the pods get deployed on nodes with the desirable hardware features.
4. Finally, using the **CPU Core Manager for Kubernetes, you can** constrain workloads to specific CPU cores, which is a highly desirable feature in an NFV environment for VNFs that have low latency and high performance needs.

We encourage you to try out these four new features and provide Intel with feedback via github or by contacting your Intel representative.

5.0 References

#	Title	Link
1	Kubernetes Overview	https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/
2	Kubernetes API Server	https://kubernetes.io/docs/admin/kube-apiserver/
3	Kubernetes pod Overview	https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/
4	CNI Github Repository	https://github.com/containernetworking/cni
5	Flannel Github Repository	https://github.com/coreos/flannel
6	Multus CNI Plugin	https://github.com/Intel-Corp/multus-cni
7	SR_IOV	http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/sr-iov-nfv-tech-brief.pdf
8	Enhanced Platform Awareness	https://builders.intel.com/docs/networkbuilders/EPA_Enablement_Guide_V2.pdf
9	SR-IOV CNI Plugin	https://github.com/Intel-Corp/sriov-cni
10	Node Feature Discovery	https://github.com/Intel-Corp/node-feature-discovery
11	CPU Core Manager for Kubernetes	https://github.com/Intel-Corp/CPU-Manager-for-Kubernetes
12	Kubernetes Labels	https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/
13	Use cases for Kubernetes	https://thenewstack.io/dls/ebooks/TheNewStack_UseCasesForKubernetes.pdf
14	Kubernetes Components	https://kubernetes.io/docs/concepts/overview/components/
15	Containers vs Virtual Machines	https://docs.docker.com/get-started/#containers-vs-virtual-machines#containers-vs-virtual-machines
16	OpenStack EPA	https://wiki.openstack.org/wiki/Enhanced-platform-awareness-pcie
17	Kube Proxy	https://kubernetes.io/docs/admin/kube-proxy/
18	CNI Readme	https://github.com/containernetworking/cni/blob/master/README.md

Acronyms

Acronym	Expansion
CNI	Container Networking Interface
COTS	Commercial off the shelf
CSP	Communication Service Provider
DPDK	Data Plane Development Kit
HA	High Availability
HT	Hyper-Thread
IP	Internet Protocol
IPAM	IP Address Management
K8s	Kubernetes
NFD	Node Feature Discovery
NFV	Netwofk Function Virtualization
NFVI	Netwofk Function Virtualization Infrastructure
NIC	Network Interface Card
NUMA	Non Uniform Memory Access
OIR	Opaque Integer Resource
OS	Operating System
PCI	Peripheral Component Interconnect
PID	Process ID
RDT	Resource Director Technology
SR-IOV	Single Root I/O Virtualization
VF	Virtual Function
VM	Virtual Machine
VNF	Virtual Network Function
VxLAN	Virtual Extensible Local Area Network
TCP	Transport Control Protocol
UDP	User Datagram Protocol

Appendix A: Hardware Bill of Material

Hardware	Component	Specification
Kubernetes Master and Minion (Dell PowerEdge R730)	Processor	2x Intel® Xeon® processor E5-2690 v3, 2.60 GHz, total of 48 logical cores with Intel® Hyper-Threading Technology
	Memory	128 GB, DDR4-2133 RAM
	Intel® NIC, 1GbE	Intel® Ethernet Server Adapter I350-T4 (using Intel® Ethernet Controller I350)
	Intel® 10GbE	Intel® Ethernet Converged Network Adapter X710-DA4 (using Intel® Ethernet Controller XL710-AM1
	Hard Drive	SATA 8 TB HDD
Top-of-rack switch	10GbE Switch	Extreme Networks Summit* X670V-48t-BF-AC 10GbE Switch, SFP+ Connections
	1GbE Switch	Cisco catalyst 2960 Series

Appendix B: CPU Layout Of The K8s Host Machines

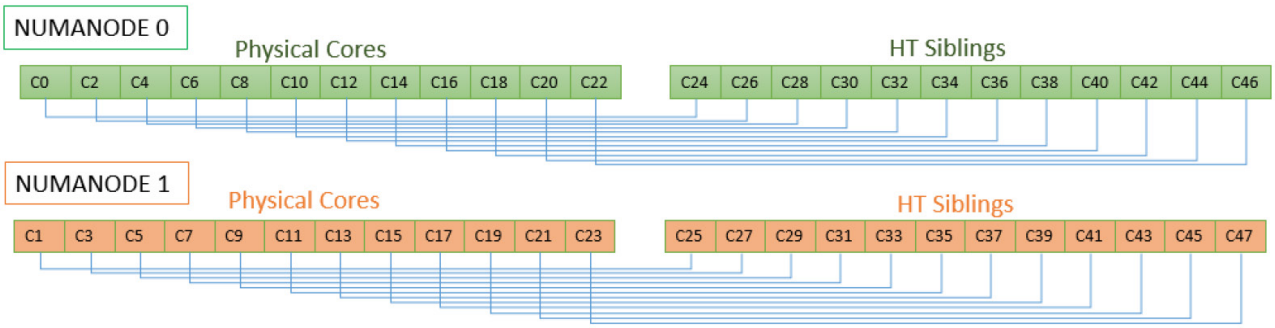

Figure 5. CPU Configuration of the Kubernetes Host Machines used in the solution

Note: Different vendors have different CPU configuration. The figure shows CPU configuration of the systems that we used in our solution.

Appendix C: Sample Multus Configuration file

```
# cat /etc/cni/net.d/multus-cni.conf                                    {
    "name": "Demo-Network",
    "type": "multus",
    "delegates": [
        {
                "type": "flannel",
                "masterplugin": true,
                "delegate": {
                        "isDefaultGateway": true
                }
        }
        {
                "type": "sriov",
                "if0": "ens2f0",
                "if0name": "south0",
                "ipam": {
                        "type": "host-local",
                        "subnet": "192.168.1.0/24",
                        "rangeStart": "192.168.1.10",
                        "rangeEnd": "192.168.1.30",
                        "routes": [
                                { "dst": "0.0.0.0/0" }
                        ],
                        "gateway": "192.168.1.1"
                }
        },
        {
                "type": "sriov",
                "if0": "ens2f1",
                "if0name": "north0",
                "dpdk": {
                     "kernel_driver":"i40evf",
                     "dpdk_driver":"vfio-pci",
                      "dpdk_tool":"/root/packages/dpdk/usertools/dpdk-devbind.py"
                }
        },
    ]
}
```

Appendix D: NodeReport After deploying CPU Core Manager for Kubernetes

```
# kubectl get NodeReport <node-name> -o json

.....
        "description": {
            "path": "/etc/kcm",
            "pools": {
                "controlplane": {
                    "cpuLists": {
                        "14,38": {
                            "cpus": "14,38",
                            "tasks": []
                        }
                    },
                    "exclusive": false,
                    "name": "controlplane"
                },
                "dataplane": {
                    "cpuLists": {
                        "10,34": {
                            "cpus": "10,34",
                            "tasks": []
                        },
                        "12,36": {
                            "cpus": "12,36",
                            "tasks": []
                        },
                        "8,32": {
                            "cpus": "8,32",
                            "tasks": []
                        }
                    },
                    "exclusive": true,
                    "name": "dataplane"
                },
                "infra": {
                    "cpuLists": {
                        "0,24,2,26,4,28,6,30,16,40,18,42,20,44,22,46": {
                            "cpus":
  "0,24,2,26,4,28,6,30,16,40,18,42,20,44,22,46",
                            "tasks": [
                                54417,
                                54451
                            ]
                        }
                    },
                    "exclusive": false,
                    "name": "infra"

                }
            }
        },
....
```

Appendix E: Sample pod specification using an NFD created Label

```
apiVersion: v1
kind: Pod
metadata:
  name: multus-test
spec:  # specification of the pod's contents
  restartPolicy: Never
  containers:
  - name: test1
    image: "busybox"
    command: ["top"]
    stdin: true
    tty: true
  nodeSelector:
    "node.alpha.kubernetes-incubator.io/nfd-network-sriov-configured":
   "true"
```

Appendix F: Example Directory Hierarchy Created by CPU Core Manager for Kubernetes

```
etc
|___ kcm
    |___lock               #system lock to prevent conflict
    |___pools
       |___controlplane
       |    |___0,1        #logical core IDS
       |    |    |___tasks #contains linux Process IDs
       |___dataplane
       |    |___2,3
       |    |    |___tasks
       |    |___4,5
       |    |    |___tasks
       |    |___exclusive #Determine exclusivity (exclusive == 1)
       |___infra
           |___6,7
           |    |___tasks
```

© 2017 Intel Corporation Printed in USA 07/17/HM/ICMCSW/PDF002 ♻ Please Recycle 336015-001US