Using Virtual Functions with DPDK in OpenShift 4

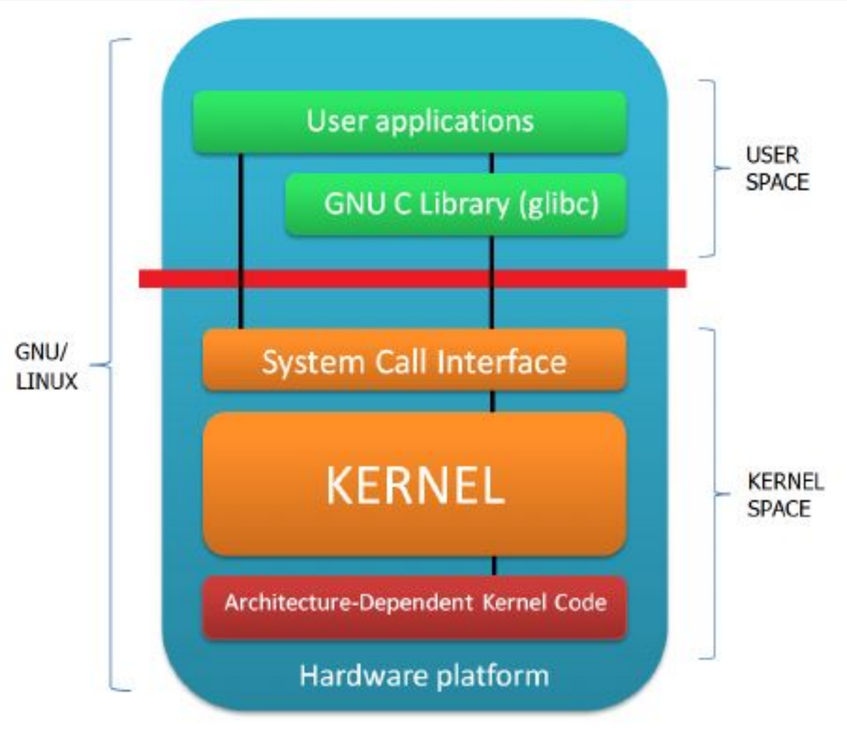
DPDK Summit 2021

Ip Sam
Architect, Red Hat Certified Architect

Wuxin Zeng
Consultant, Red Hat

Linux Architecture

- System memory in Linux is divided into two spaces.
- Kernel space is reserved for the highest of trusted functions in a system
- User space is where the “normal” programs reside.

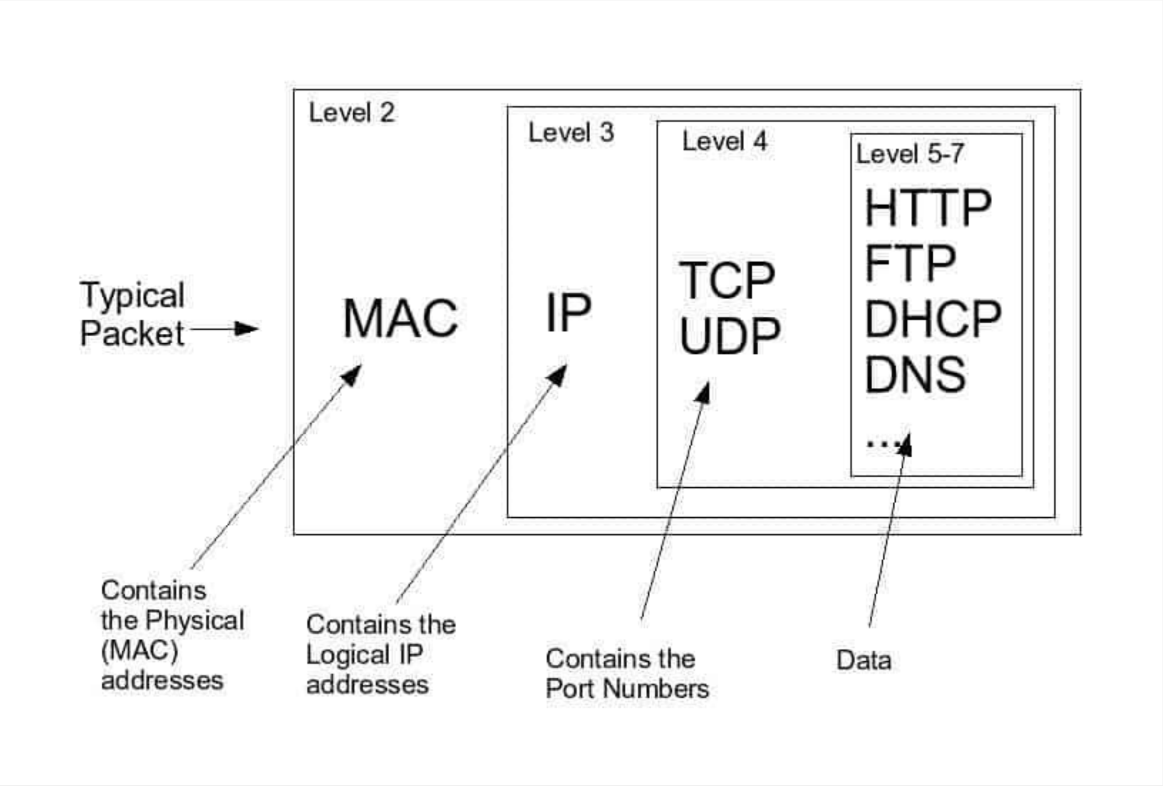


Network Packets Problems

- Network packets are moving inside the Host
- System gets notified of new packet and forward it to an allocated buffer
- Linux uses interrupt notifications handled by the kernel
- Many interrupts created when managing the packet
- More packets have to be processed more resources are consumed
- Poor performance

Network Packets

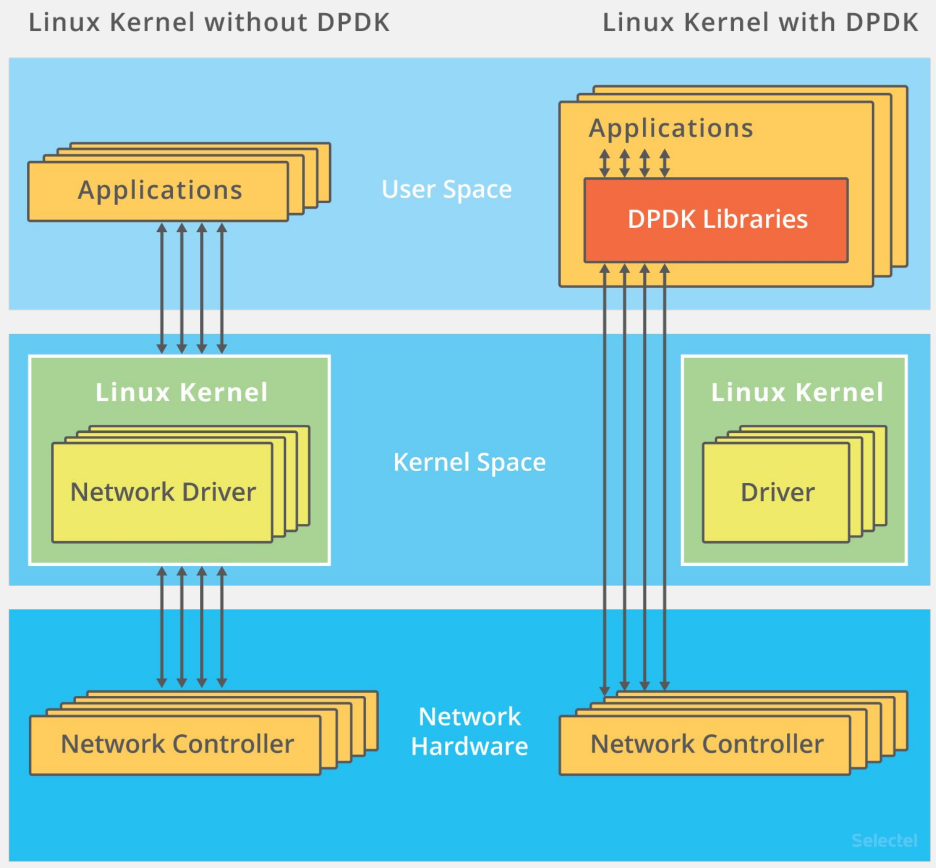
- Don’t want to manage network packet forwarding in Kernel Space
- Obtain high performance in system



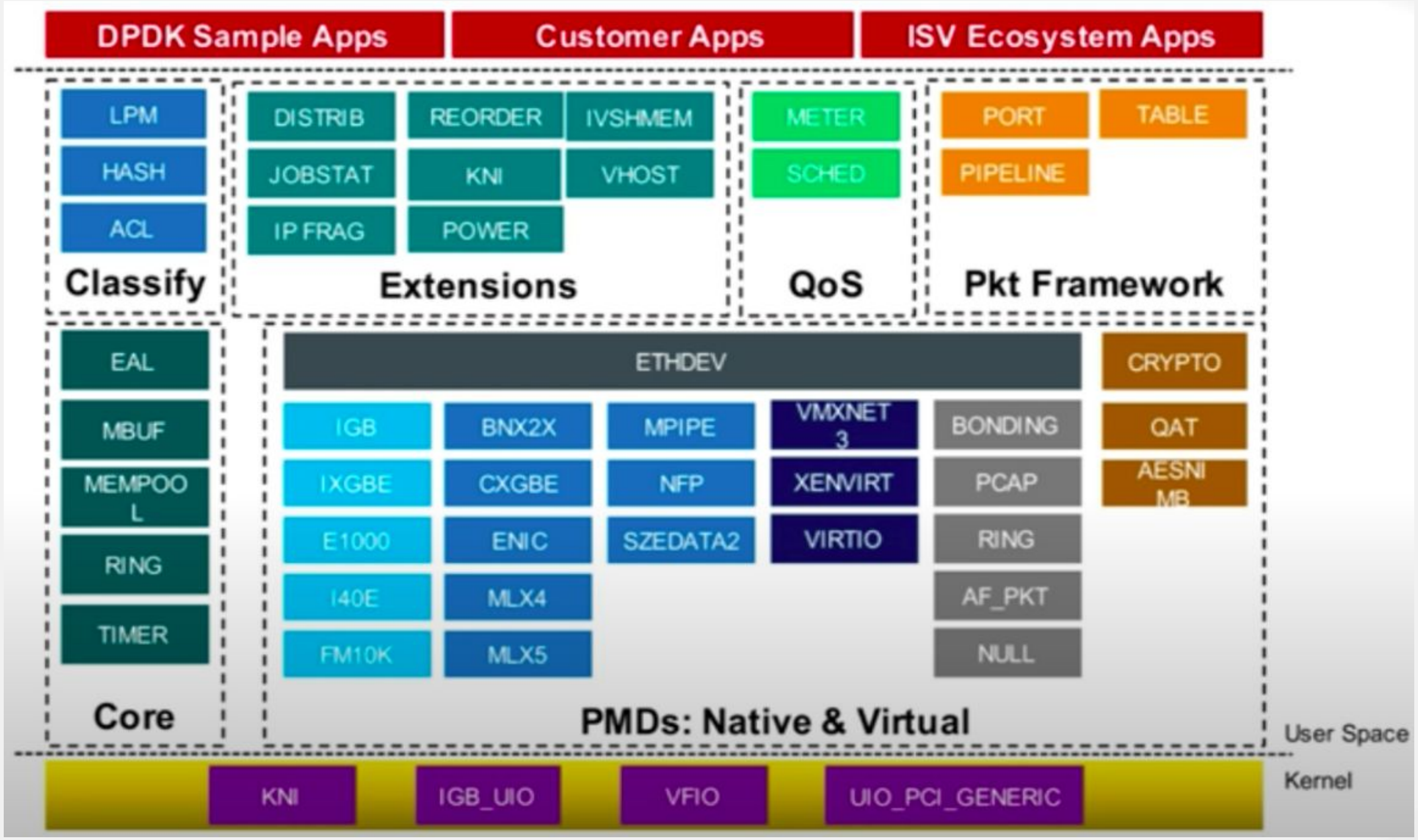
Solution

- Avoid using Kernel Space by using DPDK

DPDK Architecture



DPDK Libraries

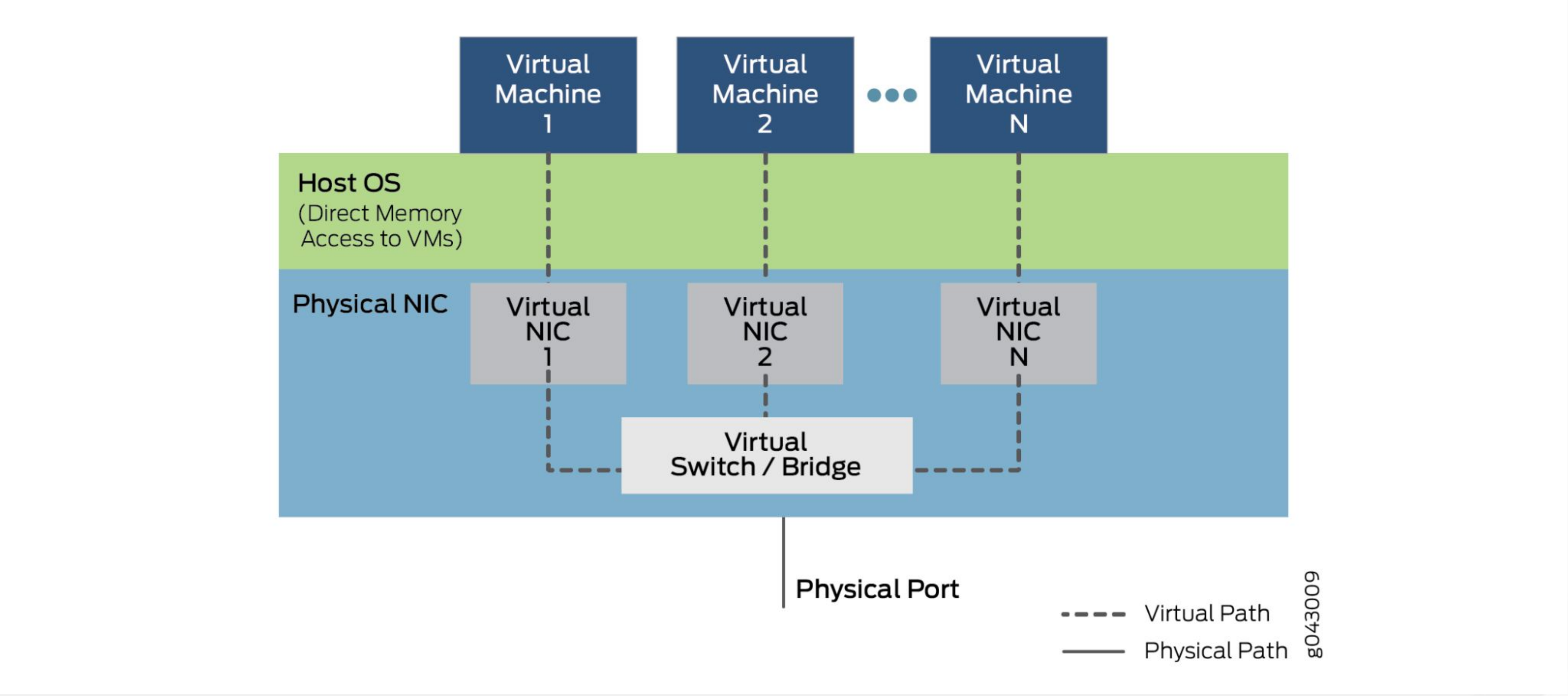


Red Hat

DPDK Benefits

- Reduce uncertainty about latency
- Improve network performance
- Reserve CPU cores that are close to the memory that the process needs
- Cache big pages of memory

Physical NIC and Virtual Functions



Red Hat

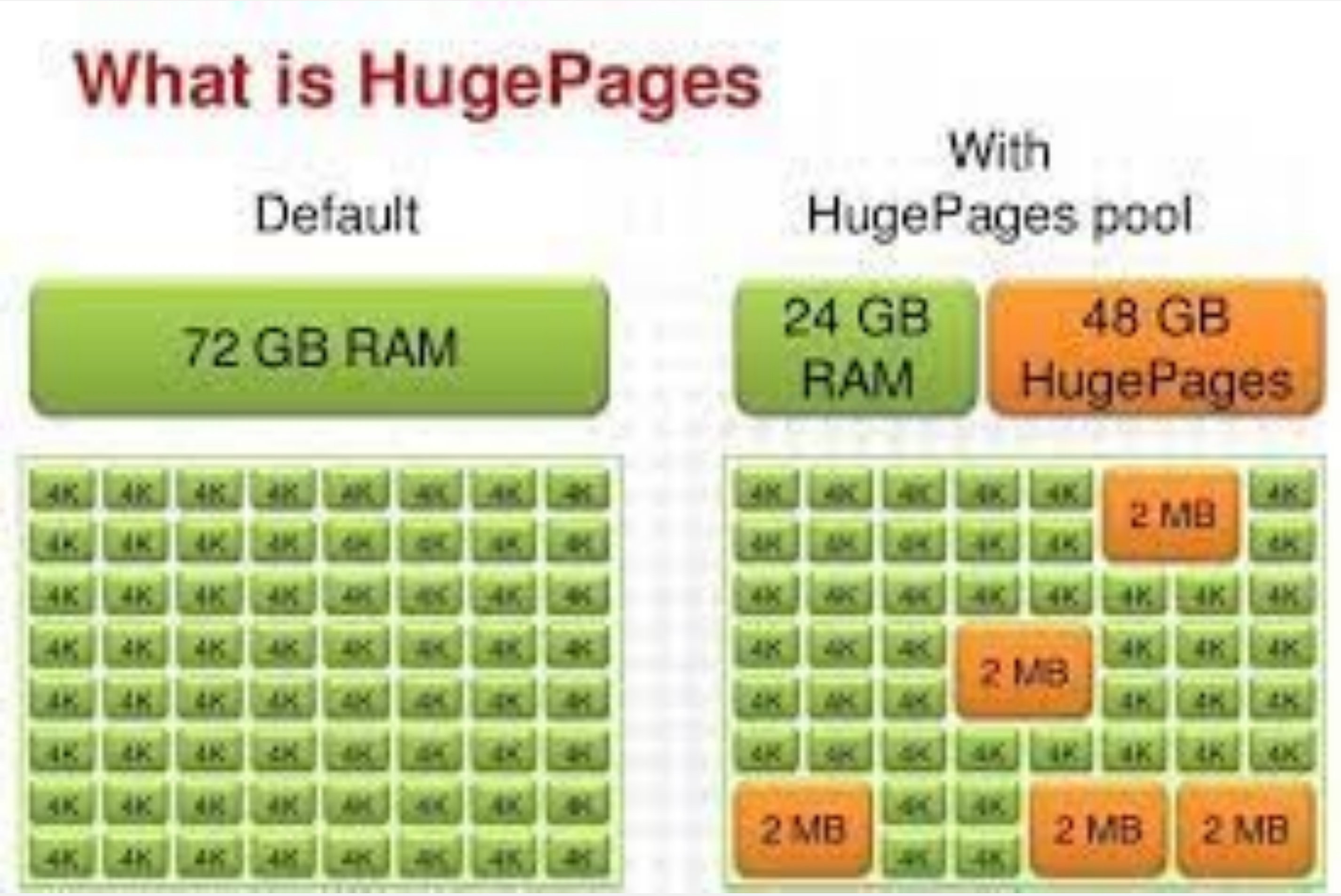
Prerequisites for Using VFs with DPDK

1. Configure SR-IOV
2. Configure HugePages

HugePages

- A huge page is a memory page that is larger than 4Ki
- Two common huge page sizes: 2Mi and 1Gi
- If a huge page pool is allocated, then huge pages need to be allocated.
- Transparent Huge Pages (THP) automates the management of huge pages without application knowledge
- THP leads to performance degradation on nodes with high memory utilization
- Applications in a pod can allocate and consume pre-allocated huge pages.

HugePages



Red Hat

HugePages Benefits

- Increased performance through increased TLB (Translation Lookaside Buffer) hits
- Pages are locked in memory and never swapped out
- Provides RAM for shared memory structures
- Contiguous pages are preallocated and only used for System shared memory
- Less bookkeeping work for the kernel for virtual memory because of larger page sizes

SR-IOV Network Operator

- Create namespace openshift-sriov-network-operator
- Create label openshift.io/run-level: "1"
- Create Operator Group and bind to that namespace
- Create subscription on the operator

SR-IOV Network Operator

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: openshift-sriov-network-operator
  labels:
    openshift.io/run-level: "1"
---
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: sriov-network-operators
  namespace: openshift-sriov-network-operator
spec:
  targetNamespaces:
  - openshift-sriov-network-operator
---
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: openshift-sriov-network-operator-subscription
  namespace: openshift-sriov-network-operator
spec:
 channel: "4.4"
 name: sriov-network-operator
 source: redhat-operators
 sourceNamespace: openshift-marketplace
```

SR-IOV Network Operator Namespace

Operator Group and Subscription

Add SR-IOV Labels to Nodes

The SR-IOV Network Node Policy requires this nodeSelector:

```
feature.node.kubernetes.io/network-sriov.capable: "true"
```

Node Feature Discovery Operator

We can install the NFD Operator to automatically label the nodes for us


```yaml
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: nfd
  namespace: openshift-operators
spec:
  channel: "4.4"
  name: nfd
  source: redhat-operators
  sourceNamespace: openshift-marketplace
```

Node Feature Discovery Operator

Node Feature Discovery CRD

```
apiVersion: nfd.openshift.io/v1alpha1
kind: NodeFeatureDiscovery
metadata:
  name: nfd-master-server
  namespace: openshift-operators
spec:
  namespace: openshift-nfd
```

Node Feature Discovery

Validate Node Labels

```
$ oc get node --show-labels
```

Config the SR-IOV Network Devices

- Specify physical NIC and the number of Virtual Functions per NIC
- Not all NICs have the same number of Virtual Functions
- Use Network Node State in SR-IOV Operator to get the VF information

```
oc get sriovnetworknodestate -n openshift-sriov-network-operator
<node name> -o yaml
```


Red Hat

SR-IOV Network Node State


```
- deviceID: '1572'
  driver: i40e
  mtu: 1500
  name: ens1f0
  pciAddress: '0000:12:00.0'
  totalvfs: 64
  vendor: '8086'
- deviceID: '1572'
  driver: i40e
  mtu: 1500
  name: ens1f1
  pciAddress: '0000:12:00.1'
  totalvfs: 64
  vendor: '8086'
- deviceID: '1572'
  driver: i40e
  mtu: 1500
  name: eno5
  pciAddress: '0000:5d:00.0'
  totalvfs: 64
  vendor: '8086'
```

Review SR-IOV Capable NICs in Web Console

Select Virtual Functions

- Sriov Network Node Policy created by the SR-IOV Operator.
- Set the nicSelector specification in this format

```
<pfname>#<first_vf>-<last_vf>
```

Select Device Type

- netdevice performs the device bind in the kernel space
- vfio-pci binds in the user space

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
  name: policy-onboard-dpdk
  namespace: openshift-sriov-network-operator
spec:
  resourceName: onboardDPDK
  nodeSelector:
    feature.node.kubernetes.io/network-sriov.capable: "true"
  mtu: 1500
  numVfs: 64
  nicSelector:
    pfNames: ["eno5#50-59", "eno6#50-59"]
  isRdma: false
  deviceType: vfio-pci
```


Red Hat

Select Device Type

SR-IOV Network Node Policy

NIC selection is done per Sriov Network Node Policy

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
  name: policy-pcie
  namespace: openshift-sriov-network-operator
spec:
  resourceName: pcieSRIOV
  nodeSelector:
    feature.node.kubernetes.io/network-sriov.capable: "true"
  mtu: 1500
  numVfs: 64
  nicSelector:
    pfNames: ["ens1f0#0-49", "ens1f1#0-49"]
  deviceType: netdevice
  isRdma: false
```

SR-IOV Network Attachment

- Define static IPs if you don't have a DHCP server in your network
- Enable the static IP address configuration in the capabilities

```yaml
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetwork
metadata:
  name: sriovnet1
  namespace: openshift-sriov-network-operator
spec:
  ipam: |
    {
      "type": "static",
      "addresses": [
        {
          "address": "192.168.99.0/24",
          "gateway": "192.168.99.1"
        }
      ],
      "dns": {
        "nameservers": ["8.8.8.8"],
        "domain": "test.lablocal",
        "search": ["test.lablocal"]
      }
    }
  vlan: 0
  spoofChk: 'on'
  trust: 'off'
  resourceName: onboardSRIOV
  networkNamespace: test-epa
  capabilities: '{"ips": true}'
```

Create SR-IOV Network

SR-IOV Testing

- Create a couple of pods using the SR-IOV network
- Pods are running on different nodes

```
apiVersion: v1
kind: Pod
metadata:
  name: test-sriov-1
  annotations:
    k8s.v1.cni.cncf.io/networks: '[
        {
                "name": "sriovnet1",
                "ips": ["192.168.99.11/24"]
        }
]'
spec:
  containers:
  - name: example-pod
    command: ["/bin/bash", "-c", "sleep 9000000"]
    image: centos/tools
  nodeName: <node1>
```

```
apiVersion: v1
kind: Pod
metadata:
  name: test-sriov-2
  annotations:
    k8s.v1.cni.cncf.io/networks: '[
        {
                "name": "sriovnet1",
                "ips": ["192.168.99.12/24"]
        }
]'
spec:
  containers:
  - name: example-pod
    command: ["/bin/bash", "-c", "sleep 9000000"]
    image: centos/tools
  nodeName: <node2>
```

SR-IOV Testing

```
$ oc rsh test-sriov-1

sh-4.2# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
group default 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
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
3: eth0@if179: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc
noqueue state UP group default
    link/ether 0a:58:0a:82:02:14 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.130.2.20/23 brd 10.130.3.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::4b5:27ff:fe7c:6a24/64 scope link
       valid_lft forever preferred_lft forever
135: net1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state
UP group default qlen 1000
    link/ether e2:bd:ea:8c:b0:49 brd ff:ff:ff:ff:ff:ff
    inet 192.168.99.11/24 brd 192.168.99.255 scope global net1
       valid_lft forever preferred_lft forever
    inet6 fe80::e0bd:eaff:fe8c:b049/64 scope link
       valid_lft forever preferred_lft forever

sh-4.2# ping 192.168.99.12
PING 192.168.99.12 (192.168.99.12) 56(84) bytes of data.
64 bytes from 192.168.99.12: icmp_seq=1 ttl=64 time=0.205 ms
64 bytes from 192.168.99.12: icmp_seq=2 ttl=64 time=0.099 ms
```

Jump to one pod and try to ping the other pod.

Red Hat

SR-IOV Testing

DPDK Configuration

- Intel NICs you should configure the vfio-pci driver
- Mellanox NIC needs to configure netdevice and set isRdma to true

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
  name: policy-onboard-dpdk
  namespace: openshift-sriov-network-operator
spec:
  resourceName: onboardDPDK
  nodeSelector:
    feature.node.kubernetes.io/network-sriov.capable: "true"
  mtu: 1500
  numVfs: 64
  nicSelector:
    pfNames: ["eno5#50-59", "eno6#50-59"]
  isRdma: false
  deviceType: vfio-pci
```

DPDK SR-IOV Network Configuration


```yaml
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetwork
metadata:
  name: dpdknet1
  namespace: openshift-sriov-network-operator
spec:
  ipam: |
    {
      "type": "static",
      "addresses": [
        {
          "address": "192.168.155.0/24",
          "gateway": "192.168.155.1"
        }
      ],
      "dns": {
        "nameservers": ["8.8.8.8"],
        "domain": "testdpdk.lablocal",
        "search": ["testdpdk.lablocal"]
      }
    }
  vlan: 0
  spoofChk: 'on'
  trust: 'off'
  resourceName: onboardDPDK
  networkNamespace: test-epa
  capabilities: '{"ips": true}'
```

DPDK Testing

- Have an Image that uses DPDK
- Create the POD including HugePages configuration and the SR-IOV network definition

DPDK Testing

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-dpdk-1
  namespace: test-epa
  annotations:
    k8s.v1.cni.cncf.io/networks: '[
        {
                "name": "dpdknet1",
                "ips": ["192.168.155.12/24"]
        }
]'
```

```yaml
spec:
  containers:
  - name: testpmd
    image: <image>
    securityContext:
     capabilities:
        add: ["IPC_LOCK"]
    volumeMounts:
    - mountPath: /dev/hugepages
      name: hugepage
    resources:
      limits:
        openshift.io/onboardDPDK: "1"
        memory: "1Gi"
        cpu: "4"
        hugepages-1Gi: "4Gi"
      requests:
        openshift.io/onboardDPDK: "1"
        memory: "1Gi"
        cpu: "4"
        hugepages-1Gi: "4Gi"
    command: ["sleep", "infinity"]
  nodeName: worker-epa-1.ocp.172.18.234.152.nip.io
  volumes:
  - name: hugepage
    emptyDir:
      medium: HugePages
```

Conclusion

- We can use the DPDK libraries and attach a network interface (virtual function) directly to the pod.
- DPDK libraries offers to free up the Kernel space from interrupts by processing the work in User space.
- We can leverage Red Hat's DPDK builder image from the Red Hat registry to simplify the application building process.
- The base image allows developers to build applications powered by DPDK with efficiency and quality.

Thank you

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.

linkedin.com/company/red-hat

youtube.com/user/RedHatVideos

facebook.com/redhatinc

twitter.com/RedHat