NFV Reference Design For A Containerized vEPC Application

Solution Summary

Intel Corporation

Data Center Network Solutions Group

Yu Zhou *Solutions Technology Engineer*

Cindy Spear *Solutions Technology Engineer*

Xuekun Hu *Platform Application Engineer*

Ivan Coughlan *Network Software Architect*

Kuralamudhan Ramakrishnan *Network Software Engineer*

Arindam Saha *Solutions Technology Manager*

ZTE Corporation

Telco Cloud and Core Network Product Group

Jinlei Zhu *System Architect*

Lin Yang *System Architect*

Jianfeng Zhou *System Architect*

Minghe Zhao *System Architect*

Mingxing Zhang *System Architect*

Wei Luo *Project Manager*

Revision History

Date	Revision	Comments
June 12, 2017	1.0	Version 1.0

Contents

Figures

Tables

1.0 Introduction

Intel Corporation, the world's biggest semiconductor company, is spearheading the transformation of the network and communications industry from traditional dedicated hardware equipment to software defined networks (SDN) and network function virtualization (NFV) leveraging the economies of scale and innovations of high volume servers. In this reference design, Intel provides an open source container platform on which any commercial containerized VNF can be deployed.

ZTE Corporation, the world's leading integrated tele-communications solutions provider, is one of China's largest communications equipment makers. In this reference design, ZTE provides a containerized virtual Evolved Packet Core (vEPC) VNF, integrated on Intel's delivered open source container platform.

Intel and ZTE partnered with CMCC (China Mobile Communications Corporation) to develop a bare metal reference design that lays the foundation for CMCC to offer efficient, easily deployable containerized services and fast evolution of their cloud infrastructure. Intel and ZTE are planning to expand the solution to utilize both the bare metal container as well as the containers in VMs reference designs, leading to a true unified infrastructure that can support multiple clouds.

This paper describes the reference design and a demo utilizing the design. For the demo, Intel provided an open source container platform to deploy proprietary VNFs from multiple vendors. ZTE integrated their containerized vEPC application with improved performance by using Intel EPA (Enhanced Platform Awareness) features. The demo of the bare metal reference design, while showcasing a proof point for the industry, bolsters the build-up of cloud-native services leading to large-scale platform deployment in the near future. Intel and ZTE collaborated to setup the demo in CMCC lab to present how 1) A proprietary VNF with cloud-native and micro-service architecture can be deployed on an open source container platform. 2) A containerized VNF can support heavy data traffic through acceleration technologies like DPDK, SR-IOV, and virtio.

CMCC uses the joint work described in this paper to explore the feasibility and necessity of technology fusion for container and NFV, Cloud transformation, and leverage the experience to contribute to the OPNFV OpenRetriever project.

2.0 Intel Container Platform for Telco VNF

Kubernetes is an open source system for automating deployment, scaling, and management of containerized applications. It originated from Google and is the anchor project in the Cloud Native Computing Foundation (CNCF) which is governed by the Linux Foundation (LF). The reference design makes use of several open source components, including Kubernetes, Docker, SR-IOV CNI Plugin, Multus CNI Plugin, and NFD. Refer to Table 3. Software Versions within this document for further details.

For network management, Kubernetes uses the Container Network Interface (CNI) API specifications and plugin framework. The CNI is primarily a specification governing the API for network provider integration and a framework to host those provider's plugin implementations. It originated from CoreOS and is widely adopted by Container Orchestration Engines (COE) such as Mesos, Cloud Foundry, and of course Kubernetes.

In NFV use cases, one key requirement is the functionality to provide multiple network interfaces to the virtualized operating environment of the Virtual Network Function (VNF). This is required for many various reasons, such as:

1. Functional separation of management, control and data network planes.
2. Support for termination of different and mutually exclusive network protocol stacks.
3. Support for implementation of different network SLAs.
4. Support for physical platform resource isolation.

Currently, while the CNI does provide a mechanism to support the multiple network interfaces requirement, Kubernetes does not. To address this gap, Intel introduced the **Multus CNI plugin** [13].

Another challenge for the Containerized VNF is the lack of support for physical platform resource isolation to guarantee network I/O performance and limit the impacts of operating in a co-located cloud environment. Single Root I/O Virtualization (SR-IOV) [11] is a technology that is currently widely used in VM-based NFV deployments, due to the increased network performance it provides by enabling direct access to the network hardware. Adding support for SR-IOV in Containers significantly impacts the network performance above what is available already. For this reason, Intel introduced the **SR-IOV CNI plugin** [12]. With this plugin, a Kubernetes pod can 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 hosts kernel and the second mode supports DPDK VNFs which execute the VF driver and network protocol stack in user space to achieve a network data plane performance which greatly exceeds the ability of the kernel network stack.

Data centers are heterogeneous by nature, comprised of nodes possessing a range of compute, network and other platform capabilities, features and configurations, due to incremental additions of varied systems over time.

For performance sensitive applications, such as a VNF, to achieve optimum performance or indeed successfully execute on the server at all, it is required that it be able to express its requirements for specific platform capabilities and in some circumstances for consumable platform resources to be allocated to it.

To cater to these needs, Intel introduced **Enhanced Platform Awareness (EPA)** [16][17] into the Kubernetes environment which exposes key features in the Intel silicon to enhance performance and security. EPA has been available within the OpenStack environment since the Kilo release.

Kubernetes scheduling criteria mostly only takes compute resources, CPU and memory, into consideration. To address this, , Intel introduced **Node Feature Discovery (NFD)** [14], a project which is in the Kubernetes incubator and aims to expose the capabilities and consumable resources existing on the servers in the cluster to the Kubernetes scheduler enabling it to make enhanced workload placement decisions. For example, NFD today can be used to discover servers with CPUs capable of executing AVX instructions and/or are capable of providing SR-IOV network plumbing.

Intel is committed to enabling NFV use cases on Container platforms, evolving the solutions over time, driving significant core changes into the Kubernetes system and establishing standards in order that we achieve continuous improvement of the overall ecosystem and user experience.

To this end, Intel has planned NFV enabling work in Kubernetes for DPDK/Huge Pages, CPU Pinning & Isolation (CPU Core Manager for Kubernetes), RDT, FPGA, QAT (IPSEC use case) Network QoS and multiple interface support evolution.

2.1 Acceleration for Container Network Path

The original container network path is through veth pair. It is a low efficient network solution for traffic forwarding. Intel now provides two alternative ways to enhance container data path. 1) SR-IOV. 2) Virtio-user. Intel integrates both features on DPDK – a highly efficient user space polling mode driver (PMD). Therefore, the container platform can support heavy traffic VNF (i.e. vEPC, vRouter) for Telco Cloud. User tasks to specific SR-IOV or Virtio-user depends on their usage model. For further details, refer to document **SR-IOV for NFV Solutions – Practical Considerations and Thoughts** [11].

1. For SRIOV – what should be emphasized here is to transition from kernel mode to user mode, which can provide a much higher performing data path.

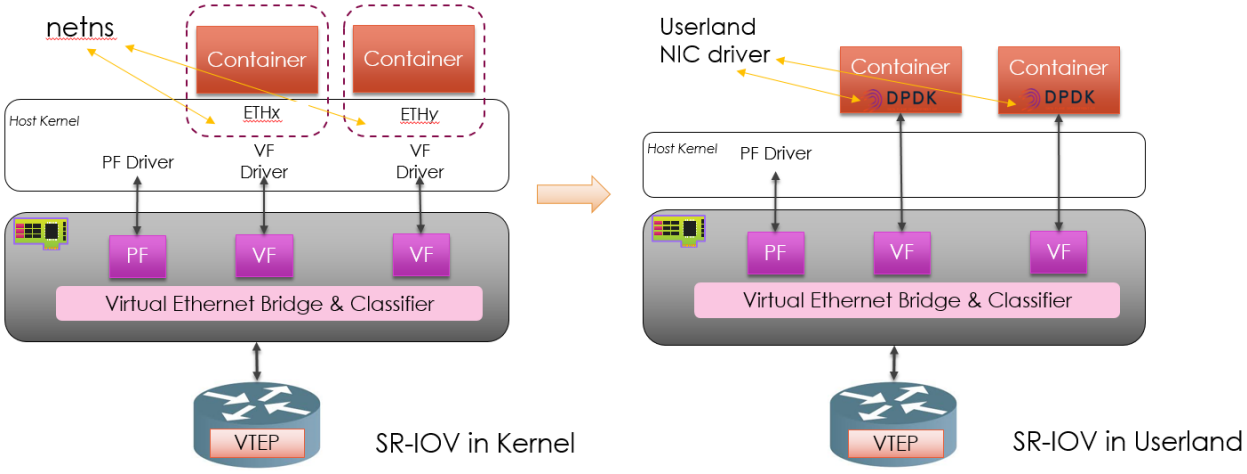

Figure 1 : SR-IOV VF promoted from Kernel to User space

2. For Virtio, it is a native Linux kernel driver. Intel also implements it in DPDK as a user space PMD.

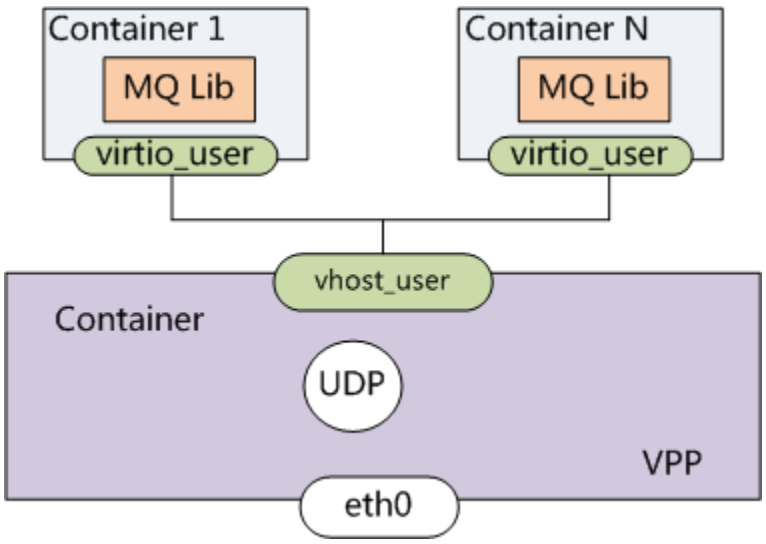

Figure 2 : Virtio Network Device Communication

The overview of virtio-user implementation in DPDK.

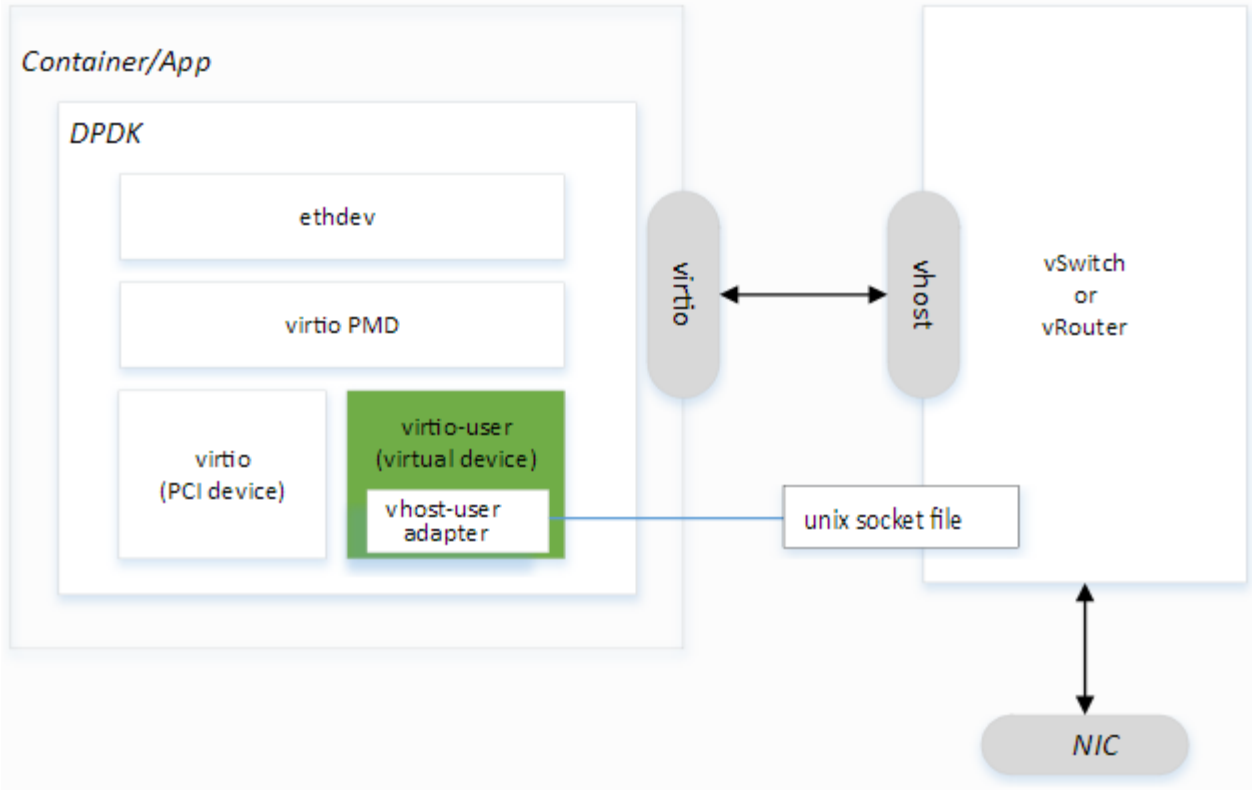

Figure 3 : Virtio Network Device in User Space

For different virtio PCI devices we usually use as a para-virtualization I/O in the context of QEMU/VM. The basic idea here is to present a kind of virtual set of devices, which can be attached and initialized by DPDK. The device emulation layer by QEMU in VM's context is saved by just registering a new kind of virtual device in DPDK's Ethernet layer. To simplify configuration, we reuse the already-existing virtio PMD code (driver/net/virtio).

Sample usage

- Start a testpmd on the host with a vhost-user port.

```
$(testpmd) -l 0-1 -n 4 --socket-mem 1024,1024 \
    --vdev 'eth_vhost0,iface=/tmp/sock0' \
    --file-prefix=host --no-pci -- -i
```

- Start a container instance with a virtio-user port.

```
$ docker run -i -t -v /tmp/sock0:/var/run/usvhost \
    -v /dev/hugepages:/dev/hugepages \
    dpdk-app-testpmd testpmd -l 6-7 -n 4 -m 1024 --no-pci \
    --vdev=virtio_user0,path=/var/run/usvhost \
    --file-prefix=container \
    -- -i --txqflags=0xf00 --disable-hw-vlan
```

2.2 Multus CNI Plugin

Basic introduction

- *Multus* is the Latin word for "Multi".
- As the name suggests, it acts as the Multi plugin in Kubernetes and provides the Multi interface support in a pod.
- It is compatible with other plugins like calico, weave and flannel, with different IPAM and networks.
- It is a contact between the container runtime and other plugins, it doesn't have any of its own net configuration and it calls other plugins like flannel/calico to do the real net conf job.
- Multus reuses the concept of invoking the delegates in flannel, it groups the multi plugins into delegates and invokes each other in sequential order, according to the JSON scheme in the CNI configuration.
- The number of plugins supported is dependent upon the number of delegates in the conf file.
- The Master plugin invokes "eth0" interface in the pod, the rest of plugins (Minion plugins e.g.: sriov,ipam) invoke interfaces as "net0", "net1".. "netn".
- The "masterplugin" is the only net conf option of the Multus CNI; it identifies the primary network. The default route will point to the primary network.

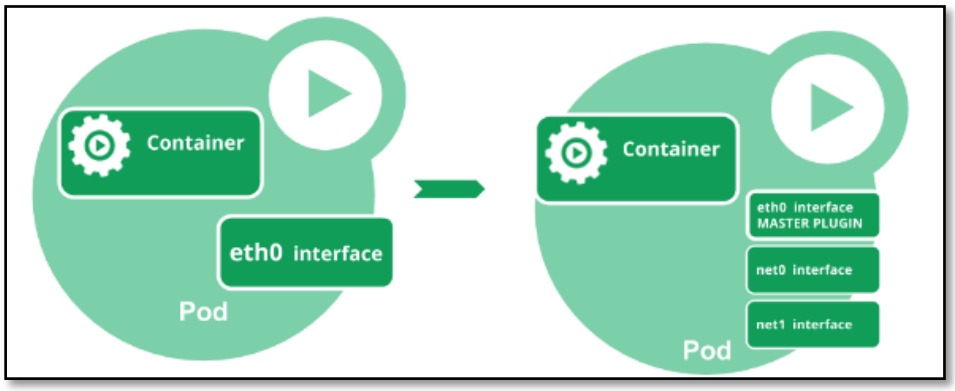

Figure 4 : Multi-Homed Pod

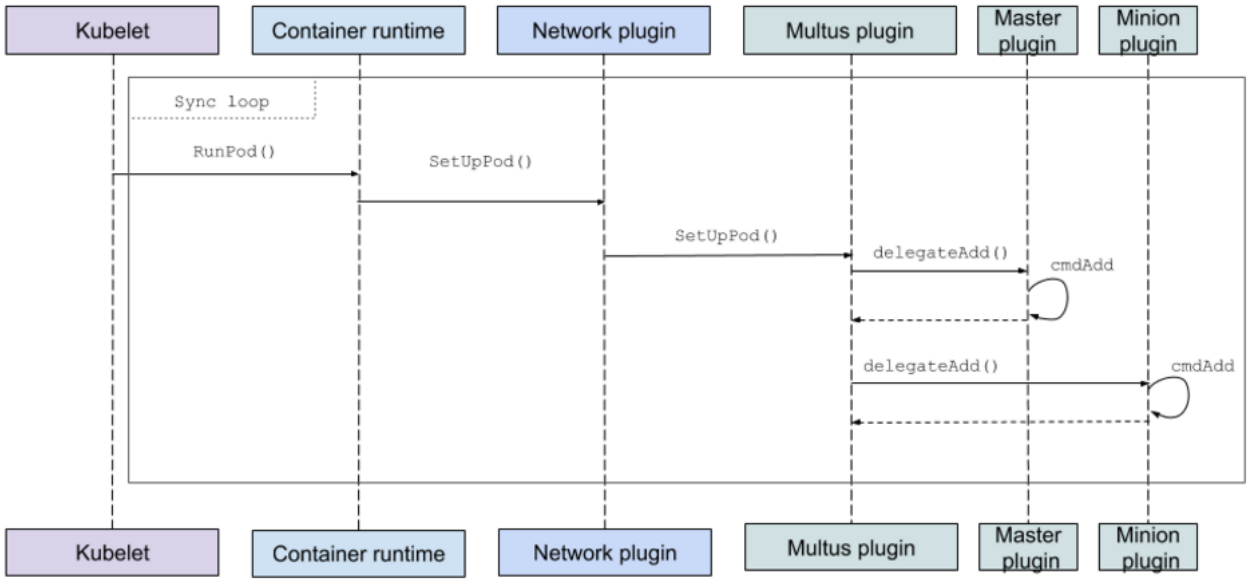

Figure 5 : Multus Network Workflow in Kubernetes

Network Configuration Reference

- *name* (string, required): the name of the network
- type (string, required): "multus"
- delegates (([]map,required): number of delegate details in the Multus
- masterplugin (bool,required): master plugin to report back the IP address and DNS to the container

Usage

```
# tee /etc/cni/net.d/multus-cni.conf <<-'EOF'
{
    "name": "multus-demo-network",
    "type": "multus",
    "delegates": [
```

```
        {
                "type": "sriov",
                #part of sriov plugin conf
                "if0": "enp12s0f0",
                "ipam": {
                        "type": "host-local",
                        "subnet": "10.56.217.0/24",
                        "rangeStart": "10.56.217.131",
                        "rangeEnd": "10.56.217.190",
                        "routes": [
                                { "dst": "0.0.0.0/0" }
                        ],
                        "gateway": "10.56.217.1"
                }
        },
        {
                "type": "ptp",
                "ipam": {
                        "type": "host-local",
                        "subnet": "10.168.1.0/24",
                        "rangeStart": "10.168.1.11",
                        "rangeEnd": "10.168.1.20",
                        "routes": [
                                { "dst": "0.0.0.0/0" }
                        ],
                        "gateway": "10.168.1.1"
                }
        },
        {
                "type": "flannel",
                "masterplugin": true,
                "delegate": {
                        "isDefaultGateway": true
                }
        }
    ]
}
EOF
```

Testing the Multus CNI with Kubernetes

- Kubelet must be configured to run with the CNI --network-plugin, with the following configuration information. Edit /etc/default/kubelet file and add KUBELET_OPTS:

```
KUBELET_OPTS="...
--network-plugin-dir=/etc/cni/net.d
--network-plugin=cni
"
```

- Restart the kubelet

```
# systemctl restart kubelet.service
```

Launching workloads in Kubernetes

- Launching the workload using yaml file in the Kubernetes master, with above configuration in the Multus CNI, each pod should have multiple interfaces.

  Note: To verify whether Multus CNI plugin is working fine, create a pod containing one "busybox" container and execute "ip link" command to check if interfaces management follows configuration.

- Create "multus-test.yaml" file containing in the configuration below. Created pod will consist of one "busybox" container running "top" command.

```
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
```

- *Create pod using command:*

```
# kubectl create -f multus-test.yaml
pod "multus-test" created
```

- *Run "ip link" command inside the container:*

```
# 1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
3: eth0@if41: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 26:52:6b:d8:44:2d brd ff:ff:ff:ff:ff:ff
20: net0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq qlen 1000
    link/ether f6:fb:21:4f:1d:63 brd ff:ff:ff:ff:ff:ff
21: net1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq qlen 1000
    link/ether 76:13:b1:60:00:00 brd ff:ff:ff:ff:ff:ff
```

Table 1. Multus interface table.

Interface name	**Description**
lo	loopback
eth0@if41	Flannel network tap interface
net0	VF assigned to the container by *SR_IOV*

	CNI plugin
net1	VF assigned to the container by SR_IOV CNI plugin

Of Note

- Multus CNI is a short term solution for Kubernetes to support multiple networks.
- For the long term solution for Multiple Network support in Kubernetes, Intel is working along with the Kubernetes Network Special Interest Group. Intel is actively involved with other stakeholders in the community for this proposal.
- Kubernetes Multiple Network proposal link; https://docs.google.com/document/d/1TW3P4c8auWwYy-w_5afIPDcGNLK3LZf0m14943eVfVg/edit?ts=58877ea7

  Intel Multiple Network use case:

  https://docs.google.com/presentation/d/1yYIzwymWBhCf-FW-qVSqf1692g649y3hHEl39-mC5Tk/edit#slide=id.g1ee67508b9_2_0

2.3 SR-IOV CNI Plugin

SR-IOV is a specification that leverage a PCIe device to appear to be multiple distinct physical PCIe devices. SR-IOV introduces the idea of physical functions (PFs) and virtual functions (VFs). Physical function (PFs) are fully-featured PCIe functions, while virtual functions are lightweight PCIe function. Each VF can be assigned to one container, and configured with separate MAC, VLAN and IP, etc. The SR-IOV CNI plugin enables the K8s pods to attach to an SR-IOV VF. The plugin looks for the first available VF on the designated port in the Multus configuration file. The plugin also supports the Data Plane Development Kit (DPDK) driver i.e. vfio-pci for these VFs, which can provide high performance networking interfaces to the K8s pods for data plane acceleration in containerized VNF. Otherwise, the driver of these VFs should be 'i40evf' in kernel space.

Enable SR-IOV

- Given Intel ixgbe NIC on CentOS, Fedora or RHEL

```
# vi /etc/modprobe.conf
options ixgbe max_vfs=8,8
```

Network configuration reference

- name (string, required): the name of the network

- type (string, required): "sriov"
- master (string, required): name of the PF
- ipam (dictionary, required): IPAM configuration to be used for this network.(kernel mode)
- dpdk (dictionary required only in userspace)
    - kernel_driver (string, required): name of the NIC driver e.g i40evf
    - dpdk_driver (string, required): name of the DPDK driver e.g. vfio-pci
    - dpdk_tool (string, required): absolute path of dpdk bind script e.g. dpdk-devbind.py

Extra arguments

- vf (int, optional): VF index, default value is 0
- vlan (int, optional): VLAN ID for VF device

Usage in kernel mode using IPAM

```
# cat > /etc/cni/net.d/10-mynet.conf <<EOF
{
    "name": "mynet",
    "type": "sriov",
    "master": "eth1",
    "ipam": {
        "type": "fixipam",
        "subnet": "10.55.206.0/26",
        "routes": [
            { "dst": "0.0.0.0/0" }
        ],
        "gateway": "10.55.206.1"
    }
}
EOF
```

Add container to network

```
# CNI_PATH=`pwd`/bin
# cd scripts
# CNI_PATH=$CNI_PATH
CNI_ARGS="IgnoreUnknown=1;IP=10.55.206.46;VF=1;MAC=66:d8:02:77:aa:aa" ./priv-net-run.sh ifconfig
contid=148e21a85bcc7aaf
netnspath=/var/run/netns/148e21a85bcc7aaf
```

```
eth0      Link encap:Ethernet  HWaddr 66:D8:02:77:AA:AA
          inet addr:10.55.206.46  Bcast:0.0.0.0  Mask:255.255.255.192
          inet6 addr: fe80::64d8:2ff:fe77:aaaa/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 b)  TX bytes:558 (558.0 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:0
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
```

Remove container from network:

```
# CNI_PATH=$CNI_PATH ./exec-plugins.sh del $contid /var/run/netns/$contid
```

For example:

```
# CNI_PATH=$CNI_PATH ./exec-plugins.sh del 148e21a85bcc7aaf
/var/run/netns/148e21a85bcc7aaf
```

The diagram below shows a pod with 3 interfaces and includes Multus, Flannel and SR-IOV CNI plugins working together.

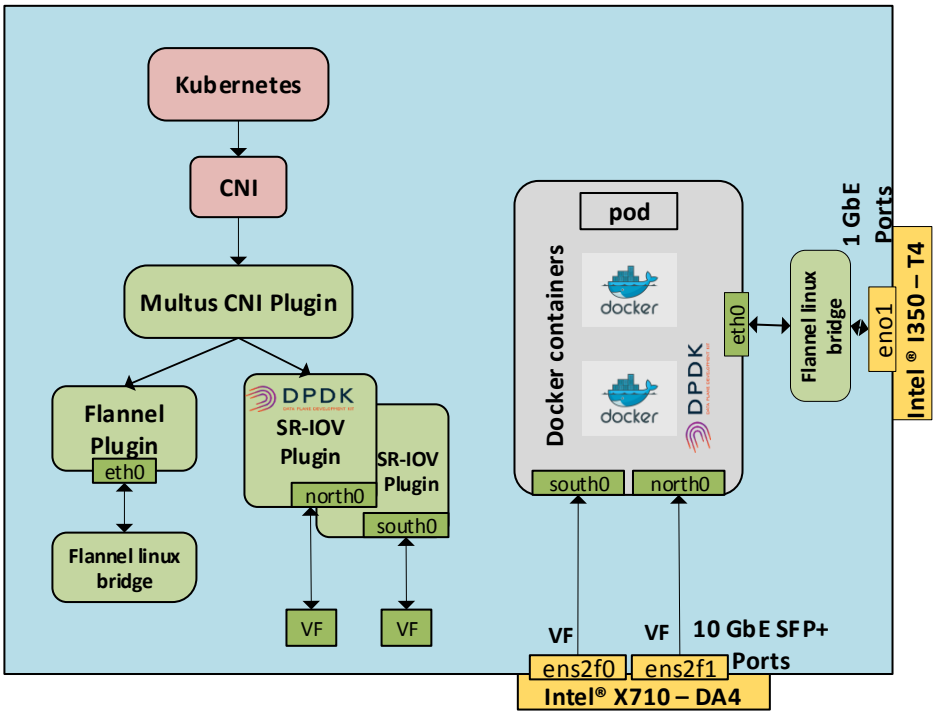

Figure 6 : Flannel and SR-IOV Plugins

- **Flannel interface** – is the 'masterplugin' as default gateway associated with the 'eth0' in a K8s pod.
- **SR-IOV VF Interface using kernel driver** – this VF is instantiated from the host machine's physical port 'ens2f0', the first port on the Intel® X710-DA4 NIC. This interface name inside pod is 'south0', and it can be assigned an IP address using IPAM.
- **SR-IOV VF interface using DPDK driver – VF** is instantiated from the host machine's physical port 'ens2f1', which is the second port on the Intel® X710-DA4 NIC. This interface name inside pod is 'north0'.

2.4 Node Feature Discovery (NFD)

Node Feature Discovery (NFD) is a Kubernetes project that is part of Kubernetes Incubator. 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.

The current set of feature sources are the following.

- CPUID for x86 CPU details
- Intel Resource Director Technology (RDT)
- Intel P-State driver
- Network (SRIOV VF detection on Minion node)

NFD will label/tag on each node. The published node labels encode a few pieces of information. User can get these information from below command line

```
# kubectl get nodes -o json | jq .items[].metadata.labels
{
  "node.alpha.kubernetes-incubator.io/node-feature-discovery.version":
"v0.1.0",
  "node.alpha.kubernetes-incubator.io/nfd-cpuid-<feature-name>":  "true",
  "node.alpha.kubernetes-incubator.io/nfd-rdt-<feature-name>": "true",
  "node.alpha.kubernetes-incubator.io/nfd-pstate-<feature-name>":
"true",
  "node.alpha.kubernetes-incubator.io/nfd-network-<feature-name>":
"true"
```

```
}
```

Note: only features that are available on a given node are labeled, so the only label value published for features is the string "true".

- CPUID Features (Partial List)

Feature name	Description
ADX	Multi-Precision Add-Carry Instruction Extensions (ADX)
AESNI	Advanced Encryption Standard (AES) New Instructions (AES-NI)
AVX	Advanced Vector Extensions (AVX)
AVX2	Advanced Vector Extensions 2 (AVX2)
BMI1	Bit Manipulation Instruction Set 1 (BMI)
BMI2	Bit Manipulation Instruction Set 2 (BMI2)
SSE4.1	Streaming SIMD Extensions 4.1 (SSE4.1)
SSE4.2	Streaming SIMD Extensions 4.2 (SSE4.2)
SGX	Software Guard Extensions (SGX)

- Intel Resource Director Technology (RDT) Features

Feature name	Description
RDTMON	Intel Cache Monitoring Technology (CMT) and Intel Memory Bandwidth Monitoring (MBM)
RDTL3CA	Intel L3 Cache Allocation Technology
RDTL2CA	Intel L2 Cache Allocation Technology

- P-State

Feature name	Description
TURBO	Power enhancement on Host machine

- Network Features

Feature name	Description
SRIOV	Single Root Input/Output Virtualization (SR-IOV) enabled Network Interface Card

Note: There is an example script in this repo, https://github.com/Intel-Corp/node-feature-discovery.git, that demonstrates how to deploy a job that runs NFD containers on each of the nodes, which discover hardware capabilities on the nodes they're running and assign proper label to the nodes accordingly.

```
./label-nodes.sh
```

Targeting Nodes with Specific Features - Nodes with specific features can be targeted using the **nodeSelector** field.

```
{
```

```
    "apiVersion": "v1",
    "kind": "Pod",
    "metadata": {
        "labels": {
            "env": "test"
        },
        "name": "golang-test"
    },
    "spec": {
        "containers": [
            {
                "image": "golang",
                "name": "go1",
            }
        ],
        "nodeSelector": {
                "node.alpha.kubernetes-incubator.io/nfd-pstate-turbo":
"true"
        }
    }
}
```

3.0 ZTE Cloud Native vEPC

3.1 ZTE Core Network NFV Solution Overview

ZTE's cloud native NFV solution enables operators to face industry differentiated competition, via a low cost and rapid response solution to customer needs, in order to thrive within an environment of fierce market competition.

ZTE's cloud native NFV solution is a containerized micro-service architecture. The micro-service architecture decouples the network service into a group of micro services that can be deployed and distributed independently. The network service can be completed based on the corresponding micro services, which provides a reliable basis for fast delivery of the service. New service can reuse micro-service components for rapid development and release. The containerized micro-service architecture makes the service very easy to be packaged, published and run anywhere, which can enable the service to quickly start and stop and easily perform horizontal expansion and failure recovery. The containerization of the service also provides a good environment for DevOps. The ZTE cloud native NFV solution supports automation of service development and deployment through continuous integration and self-healing systems, improving operational efficiency and reducing operational costs.

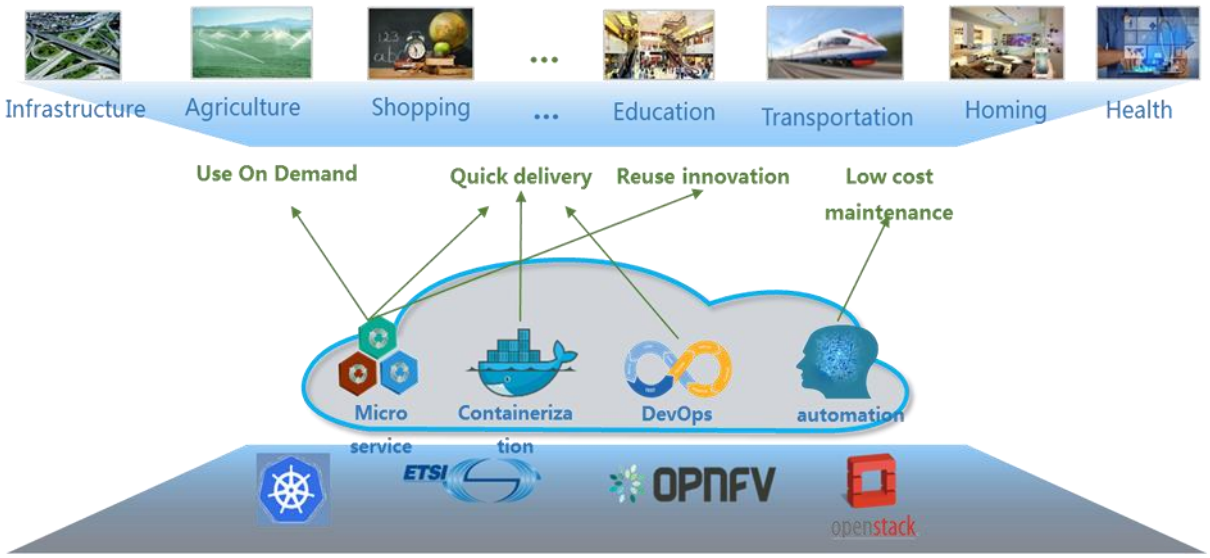

Figure 7 : ZTE NFV Solution Overview

The ZTE cloud native NFV solution is expanded on the basis of the ETSI NFV architecture. ZTE's cloud native VNF is independent to the NFVI platform including the aforementioned Kubernetes and OpenStack NFVI platform.

3.2 ZTE Cloud Native vEPC Solution

The cloud native vEPC solution is an independent micro-service such as a CDB, load balancer, etc. The Micro-service external interface is designed as IP-reachable lightweight API interface. Every micro service is designed to be stateless, and the information, such as status information, is stored in an independent CDB.

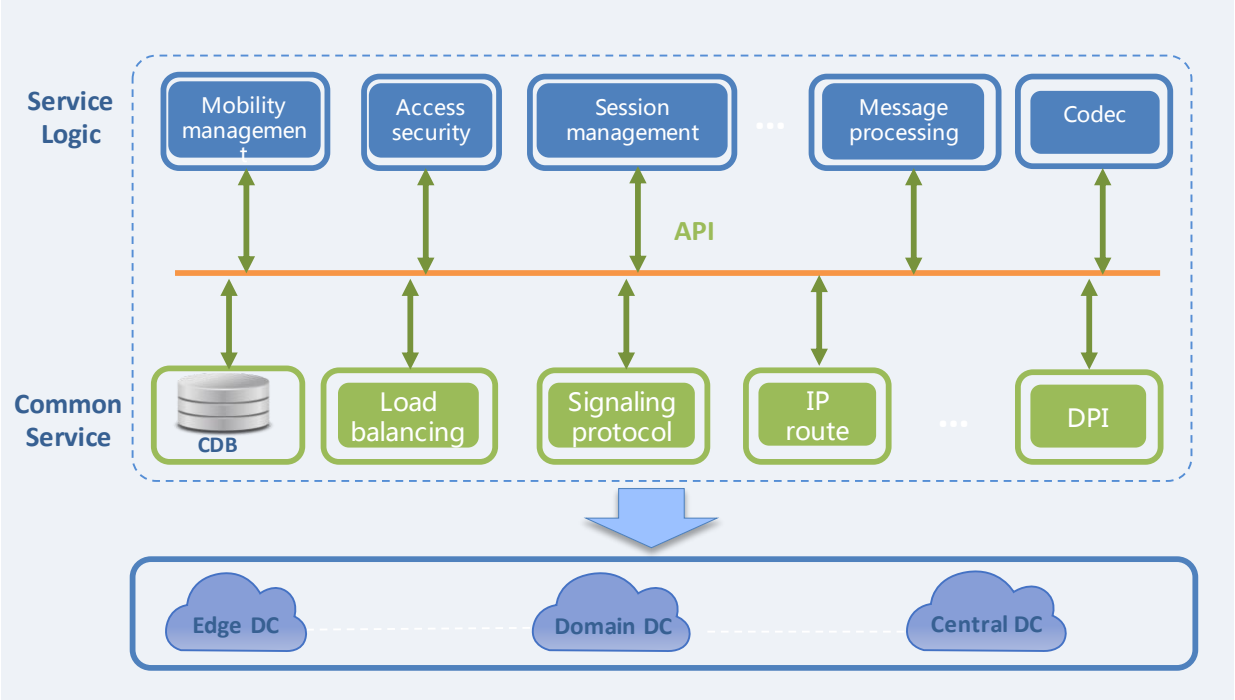

Figure 8 : Cloud Native vEPC Architecture

The cloud native vEPC solution can quickly achieve the desired product characteristics for each scene based on the target scenario. For example: according to the traditional core

network scenarios, and enterprise network scenarios, the corresponding EPC can be quickly developed.

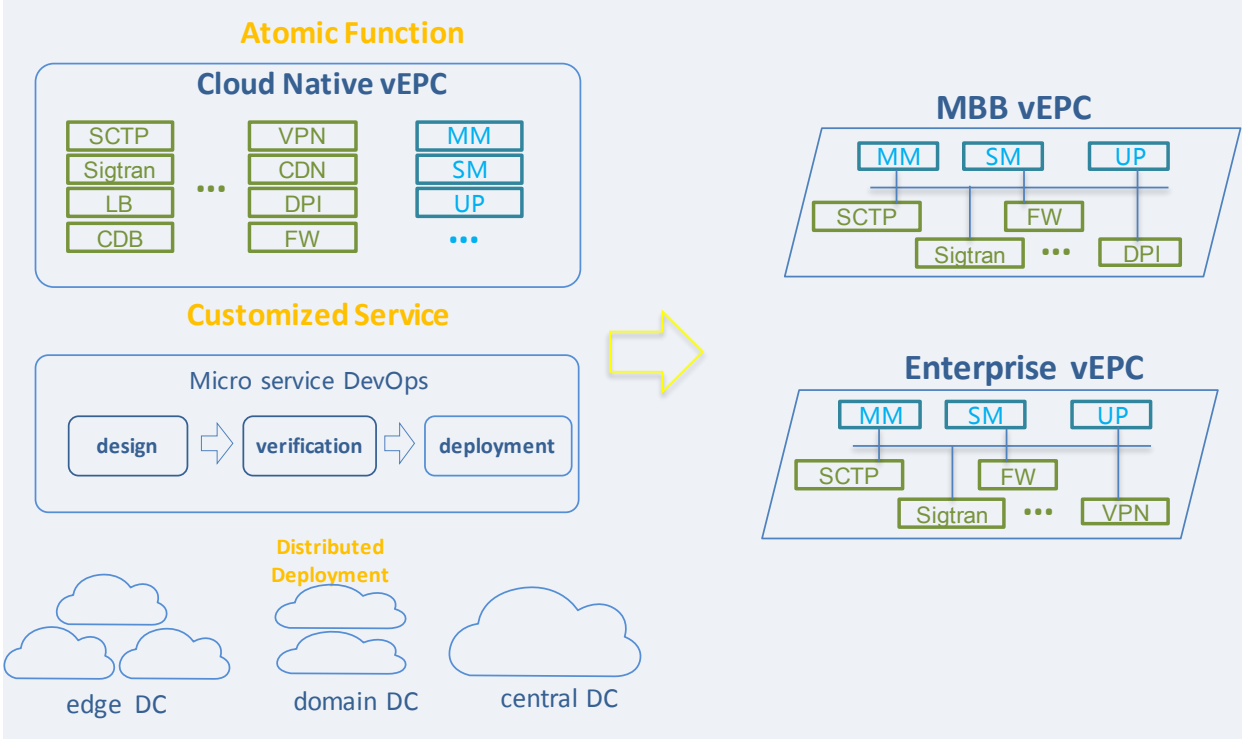

Figure 9 : Cloud Native vEPC Scenarios

4.0 User Scenario and Demonstration

Three scenarios have been created for the purpose of evaluating the ZTE cloud native vEPC solution's functionality based on Intel container platform:

1. A vEPC deployment and basic function scenario.
2. A vEPC scale in/out scenario.
3. A vEPC self-healing scenario.

ZTE vEPC cloud native solution include two parts, one is vManager which is the application life cycle manager, and the other is the vEPC application. This section will show how to setup the vManager and a step by step guide on how to replicate the different scenarios above.

4.1 vManager Deployment Guide

Follow the steps below to setup the vManager's yaml file and then create and run vManager.

- Install the host operating system, Docker engine and other related software.
- Get the vManager software image from the ZTE software center, then upload it to the Kubernetes master node's /home/version/zte directory.
- Put the image into the Docker Registry:

```
# docker tag vmanager masternode:5000/vmanager
# docker push masternode:5000/vmanager
```

- Edit the vmanager.yaml file.

```
apiVersion: v1
kind: Service
metadata:
  name: vmanagersvc
  labels:
    app: vmanager
spec:
  type: NodePort
  ports:
  - port: 8080
    targetPort: 80
    protocol: TCP
    name: http
  selector:
```

```
    app: vmanager
----
apiVersion: v1
kind: ReplicationController
metadata:
  name: vmanager
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: vmanager
    spec:
      containers:
      - name: vmanager
        image: masternode:5000/vmanager
        ports:
        - containerPort: 80
```

- Run the create command, and vmanager will launch.

```
$ kubectl create -f ./vmanager.yaml
$ kubectl get pods|grep vmanager
NAME             READY     STATUS    RESTARTS   AGE
vmanager          1/1       Running   0          2m
```

4.2 vEPC Deployment Scenario

In this scenario, we will show the deployment of the vEPC application and the basic function of the vEPC application. All the deployment of the vEPC will be done on the vManager's webpage, making it very easy and convenient for the user.

Then, we show a video download test case for the vEPC. We prepared a video server, a video client and a simulate eNodeB proxy first. After the vEPC is deployed, we import the prepared service configuration file for the vEPC, then the video client will connect to the video server through the simulate eNodeB proxy.

The steps to deploy vEPC application are as follows:

- On the ZTE vManager Project image repository page, upload the vEPC image files of App as initial version 1.

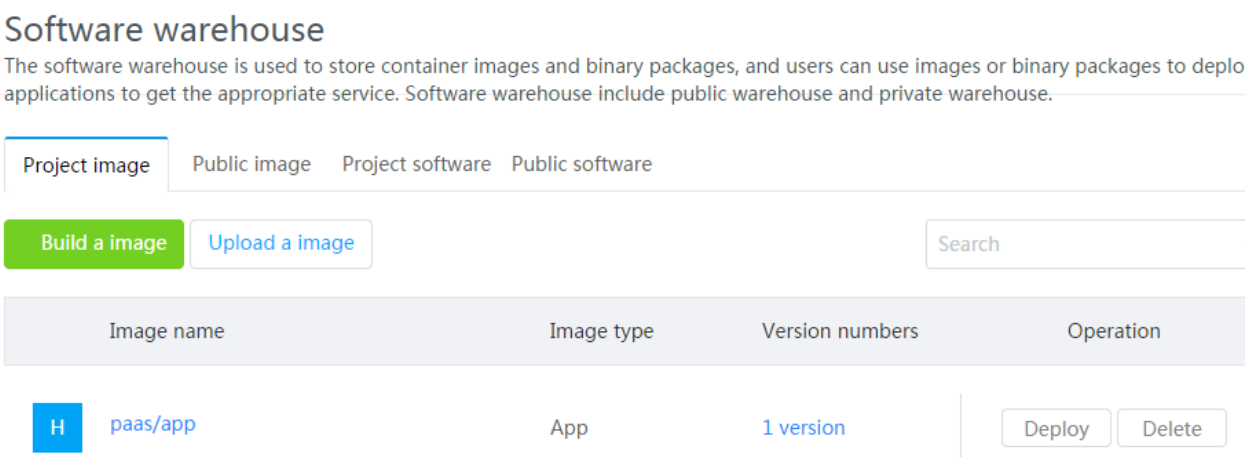

Figure 10 : App Image in Software Warehouse

- Upload the vEPC blueprint file using the vManager Blueprint center.

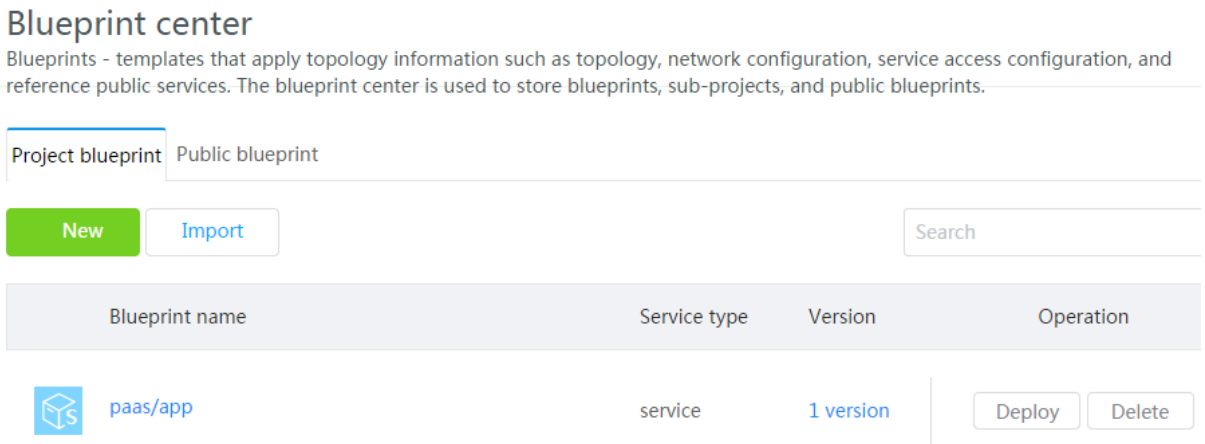

Figure 11 : App Blueprint in Blue Print Center

- Deploy the imported vEPC blueprint, and wait for about 30 seconds. The vEPC app will be deployed successfully.

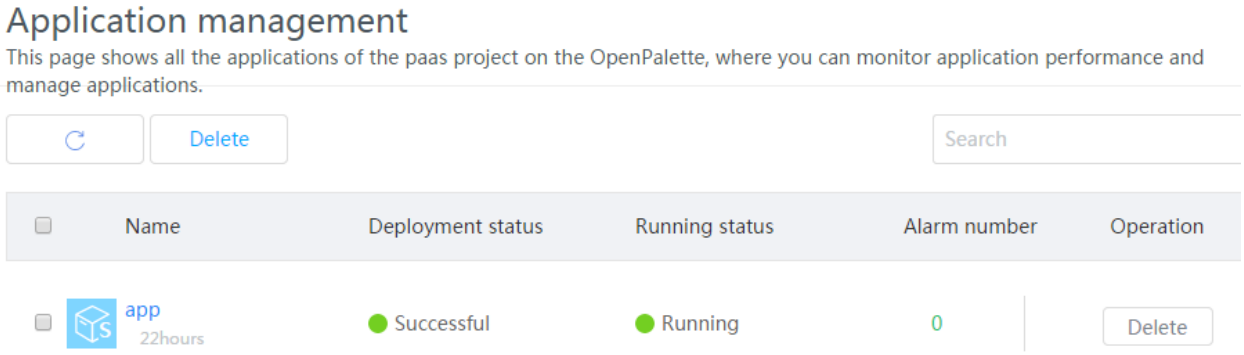

Figure 12 : App Deployed Successfully

The steps to configure and test the vEPC network are as follows:

- To configure vEPC's apn/service ip/ip pool/route information, connect to vEPC's CLI windows, and import the prepared vEPC service configuration file.
- The UE of a simulated eNodeB proxy connects to vEPC network normally.
- The video server is deployed to the vEPC's PDN network.
- Video client through the simulated eNodeB proxy connects to the video server and plays the video.

Benefit shown using the containerization and micro service architecture:

The entire process, from deployment of the vEPC application to the video client playing video files normally, can be completed within 2 minutes.

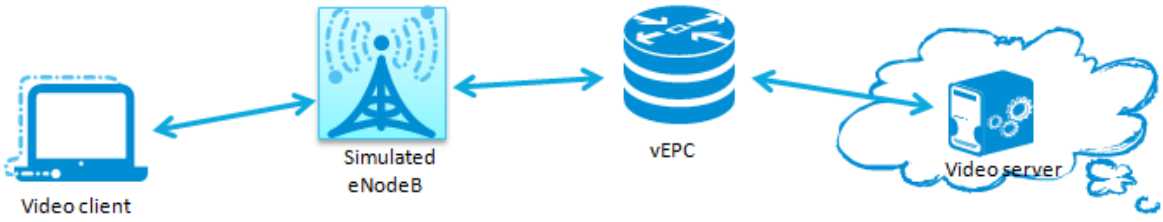

Figure 13 : vEPC Workflow

4.3 vEPC Scaled Scenario

In this scenario, we will see that the vEPC can scale in/out automatically according to vEPC PDP usage.

When the PDP usage ratio exceeds the scale-out threshold, the platform will create a new vEPC instance, and the new instance automatically joins the corresponding service cluster. When the PDP usage ratio is less than the scale-in threshold, the platform will delete a vEPC instance.

Benefit from the containerization, micro-service and stateless service architecture:

The vEPC network can scale in/out very quickly and no impact to the online sessions.

The steps of scale-out scenario are as follows:

- The maximum number of vEPC instance is set to 3, and the minimum number is set to 1.

- Configure the vEPC scale in/out policy, scale out:PDP usage ratio exceeds 80%, scale in: PDP use ratio is below 20%.
- The simulated eNodeB initiates simu-user sessions connection.
- When online PDP usage ratio exceeds 80%, the container platform creates a new vEPC instance.
- The new vEPC instance will automatically join the corresponding service cluster, and in this scale process, online video playback will not be affected.

The scale-in scenario steps as follows:

- The vEPC instance current number is 2, the maximum number of vEPC instance is set to 3 and the minimum number is set to 1.
- Configure the vEPC scale in/out policy, scale out:PDP usage ratio exceeds 80%, scale in: PDP use ratio is below 20%.
- The simulated eNodeB cancels initiated simu-user sessions.
- When online PDP usage ratio falls below 80%, the container platform will delete a vEPC instance.
- The vEPC service will re-load sharing, and in this sacle process, online video playback will not be affected.

4.4 Service Instance Redundancy Scenario

When one instance of vEPC service is abnormal or deleted, the Container platform will actively start a new service instance. The new service instance will automatically join the corresponding service cluster, in which the load balance service will distribute the user session to this service instance.

Benefit from stateless service architecture:

The failure of service instances does not have the impact on the session continuity of the vEPC.

The steps of this scenario steps are as follows:

- The platform expected vEPC instance number is set to 2.
- Through docker stop command, stop one vEPC instance.
- Container platform will start a new vEPC instance successfully, and in this scale process, online video playback will not be affected.

5.0 Ingredients

Table 2. Hardware Bill of Materials

Hardware	Component	Specification
Kubernetes Master and Minion Industry Server Based on IA	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

Table 3. Software Versions

Functionality	Product and Version
Kubernetes	1.5.2
Docker	1.13.1
DPDK	17.02
Multus CNI Plugin	https://github.com/Intel-Corp/multus-cni
SR-IOV CNI Plugin	https://github.com/Intel-Corp/sriov-cni
Node Feature Discovery (NFD)	https://github.com/Intel-Corp/node-feature-discovery
vEPC	ZXUN vEPC V6.17.10B5

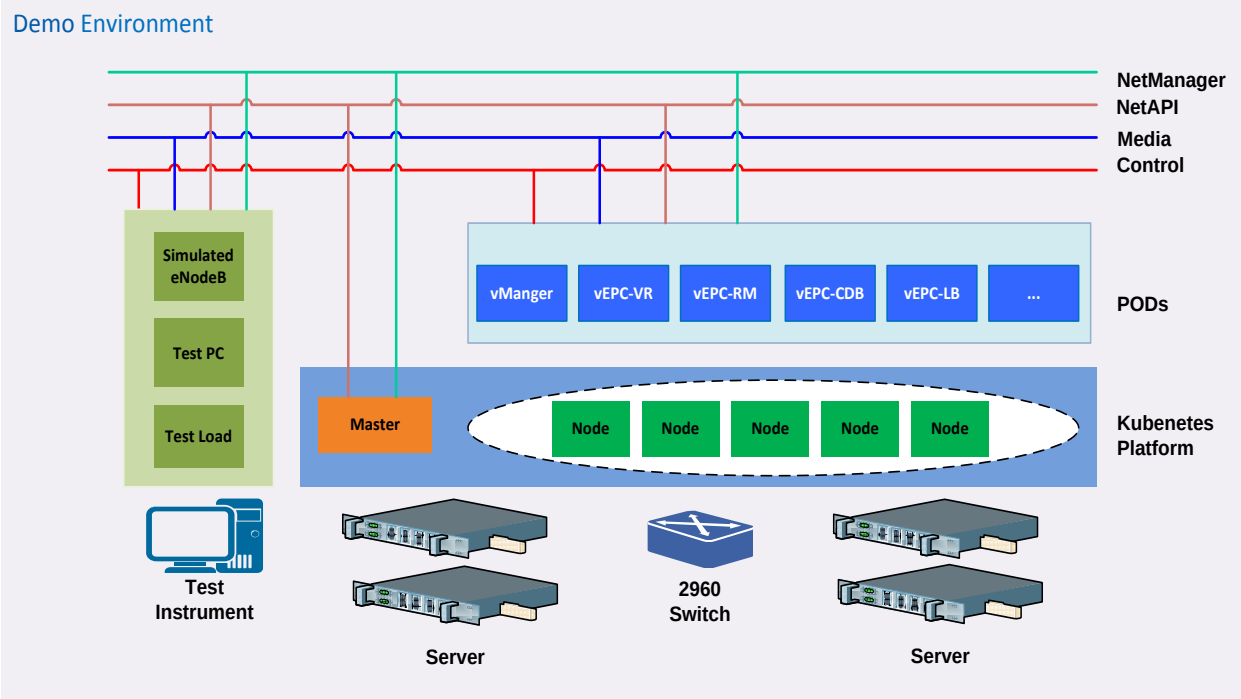

Figure 14 : vEPC Demo Topology

Demo topology networks:

#	Name	Attribute
1	NetManager	Traffic of kubernetes network mangager
2	NetAPI	Traffic of Kubernetes API
3	Media	Media traffic with POD
4	Control	Control traffic with POD

Demo topology components:

#	Name	Attribute
1	Master	Master of kubernetes
2	Simulated eNodeB	Simulated eNodeB for test
3	Test PC	PC for Test
4	Node	Node of kubernetes

5	**Test Load**	**Test Simulator for vEPC**

6.0 References

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

17	OpenStack EPA	https://wiki.openstack.org/wiki/Enhanced-platform-awareness-pcie

7.0 Acronyms

Acronym	Expansion
AES	Advanced Encryption Standard
AES-NI	Advanced Encryption Standard New Instructions
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	Network Function Virtualization
NFVI	Network 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
PMD	Poll Mode Driver
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