DPDK

DATA PLANE DEVELOPMENT KIT

HowTo Guides

Release 19.08.2

Nov 15, 2019

CHAPTER

ONE

LIVE MIGRATION OF VM WITH SR-IOV VF

1.1 Overview

It is not possible to migrate a Virtual Machine which has an SR-IOV Virtual Function (VF).

To get around this problem the bonding PMD is used.

The following sections show an example of how to do this.

1.2 Test Setup

A bonded device is created in the VM. The virtio and VF PMD's are added as slaves to the bonded device. The VF is set as the primary slave of the bonded device.

A bridge must be set up on the Host connecting the tap device, which is the backend of the Virtio device and the Physical Function (PF) device.

To test the Live Migration two servers with identical operating systems installed are used. KVM and Qemu 2.3 is also required on the servers.

In this example, the servers have Niantic and or Fortville NIC's installed. The NIC's on both servers are connected to a switch which is also connected to the traffic generator.

The switch is configured to broadcast traffic on all the NIC ports. A *Sample switch configuration* can be found in this section.

The host is running the Kernel PF driver (ixgbe or i40e).

The ip address of host_server_1 is 10.237.212.46

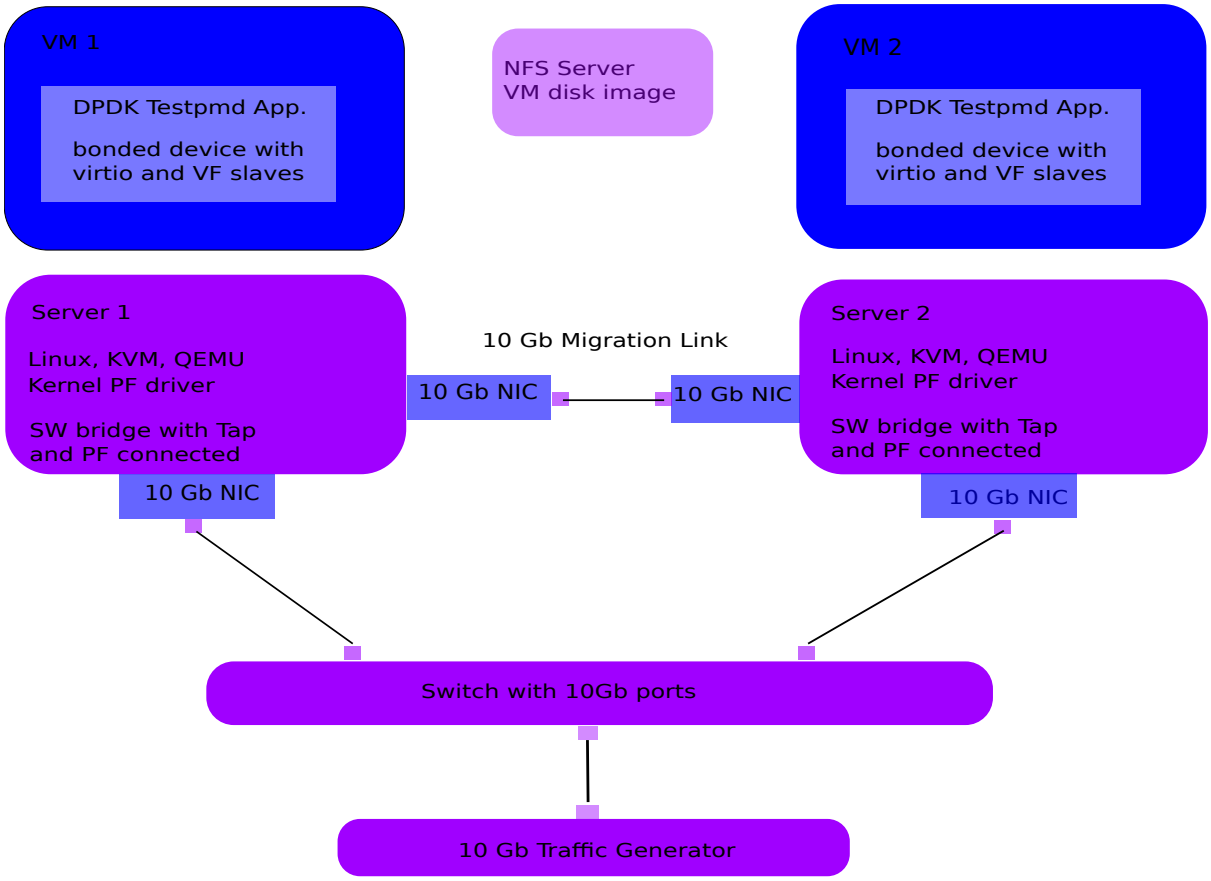
The ip address of host_server_2 is 10.237.212.131

1.3 Live Migration steps

The sample scripts mentioned in the steps below can be found in the *Sample host scripts* and *Sample VM scripts* sections.

1.3.1 On host_server_1: Terminal 1

```
cd /root/dpdk/host_scripts
./setup_vf_on_212_46.sh
```



For Fortville NIC

```
./vm_virtio_vf_i40e_212_46.sh
```

For Niantic NIC

```
./vm_virtio_vf_one_212_46.sh
```

1.3.2 On host_server_1: Terminal 2

```
cd /root/dpdk/host_scripts
./setup_bridge_on_212_46.sh
./connect_to_qemu_mon_on_host.sh
(qemu)
```

1.3.3 On host_server_1: Terminal 1

In VM on host_server_1:

```
cd /root/dpdk/vm_scripts
./setup_dpdk_in_vm.sh
./run_testpmd_bonding_in_vm.sh

testpmd> show port info all
```

The `mac_addr` command only works with kernel PF for Niantic

```
testpmd> mac_addr add port 1 vf 0 AA:BB:CC:DD:EE:FF
```

The syntax of the `testpmd` command is:

Create bonded device (mode) (socket).

Mode 1 is active backup.

Virtio is port 0 (P0).

VF is port 1 (P1).

Bonding is port 2 (P2).

```
testpmd> create bonded device 1 0
Created new bonded device net_bond_testpmd_0 on (port 2).
testpmd> add bonding slave 0 2
testpmd> add bonding slave 1 2
testpmd> show bonding config 2
```

The syntax of the `testpmd` command is:

set bonding primary (slave id) (port id)

Set primary to P1 before starting bonding port.

```
testpmd> set bonding primary 1 2
testpmd> show bonding config 2
testpmd> port start 2
Port 2: 02:09:C0:68:99:A5
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Port 2 Link Up - speed 10000 Mbps - full-duplex

testpmd> show bonding config 2
```

Primary is now P1. There are 2 active slaves.

Use P2 only for forwarding.

```
testpmd> set portlist 2
testpmd> show config fwd
testpmd> set fwd mac
testpmd> start
testpmd> show bonding config 2
```

Primary is now P1. There are 2 active slaves.

```
testpmd> show port stats all
```

VF traffic is seen at P1 and P2.

```
testpmd> clear port stats all
testpmd> set bonding primary 0 2
testpmd> remove bonding slave 1 2
testpmd> show bonding config 2
```

Primary is now P0. There is 1 active slave.

```
testpmd> clear port stats all
testpmd> show port stats all
```

No VF traffic is seen at P0 and P2, VF MAC address still present.

```
testpmd> port stop 1
testpmd> port close 1
```

Port close should remove VF MAC address, it does not remove perm_addr.

The `mac_addr` command only works with the kernel PF for Niantic.

```
testpmd> mac_addr remove 1 AA:BB:CC:DD:EE:FF
testpmd> port detach 1
Port '0000:00:04.0' is detached. Now total ports is 2
testpmd> show port stats all
```

No VF traffic is seen at P0 and P2.

1.3.4 On host_server_1: Terminal 2

```
(qemu) device_del vf1
```

1.3.5 On host_server_1: Terminal 1

In VM on host_server_1:

```
testpmd> show bonding config 2
```

Primary is now P0. There is 1 active slave.

```
testpmd> show port info all
testpmd> show port stats all
```

1.3.6 On host_server_2: Terminal 1

```
cd /root/dpdk/host_scripts
./setup_vf_on_212_131.sh
./vm_virtio_one_migrate.sh
```

1.3.7 On host_server_2: Terminal 2

```
./setup_bridge_on_212_131.sh
./connect_to_qemu_mon_on_host.sh
(qemu) info status
VM status: paused (inmigrate)
(qemu)
```

1.3.8 On host_server_1: Terminal 2

Check that the switch is up before migrating.

```
(qemu) migrate tcp:10.237.212.131:5555
(qemu) info status
VM status: paused (postmigrate)
```

For the Niantic NIC.

```
(qemu) info migrate
capabilities: xbzrle: off rdma-pin-all: off auto-converge: off zero-blocks: off
Migration status: completed
total time: 11834 milliseconds
downtime: 18 milliseconds
setup: 3 milliseconds
transferred ram: 389137 kbytes
throughput: 269.49 mbps
remaining ram: 0 kbytes
```

```
total ram: 1590088 kbytes
duplicate: 301620 pages
skipped: 0 pages
normal: 96433 pages
normal bytes: 385732 kbytes
dirty sync count: 2
(qemu) quit
```

For the Fortville NIC.

```
(qemu) info migrate
capabilities: xbzrle: off rdma-pin-all: off auto-converge: off zero-blocks: off
Migration status: completed
total time: 11619 milliseconds
downtime: 5 milliseconds
setup: 7 milliseconds
transferred ram: 379699 kbytes
throughput: 267.82 mbps
remaining ram: 0 kbytes
total ram: 1590088 kbytes
duplicate: 303985 pages
skipped: 0 pages
normal: 94073 pages
normal bytes: 376292 kbytes
dirty sync count: 2
(qemu) quit
```

1.3.9 On host_server_2: Terminal 1

In VM on host_server_2:

Hit Enter key. This brings the user to the testpmd prompt.

```
testpmd>
```

1.3.10 On host_server_2: Terminal 2

```
(qemu) info status
VM status: running
```

For the Niantic NIC.

```
(qemu) device_add pci-assign,host=06:10.0,id=vf1
```

For the Fortville NIC.

```
(qemu) device_add pci-assign,host=03:02.0,id=vf1
```

1.3.11 On host_server_2: Terminal 1

In VM on host_server_2:

```
testpmd> show port info all
testpmd> show port stats all
testpmd> show bonding config 2
testpmd> port attach 0000:00:04.0
Port 1 is attached.
Now total ports is 3
Done

testpmd> port start 1
```

The `mac_addr` command only works with the Kernel PF for Niantic.

```
testpmd> mac_addr add port 1 vf 0 AA:BB:CC:DD:EE:FF
testpmd> show port stats all.
testpmd> show config fwd
testpmd> show bonding config 2
testpmd> add bonding slave 1 2
testpmd> set bonding primary 1 2
testpmd> show bonding config 2
testpmd> show port stats all
```

VF traffic is seen at P1 (VF) and P2 (Bonded device).

```
testpmd> remove bonding slave 0 2
testpmd> show bonding config 2
testpmd> port stop 0
testpmd> port close 0
testpmd> port detach 0
Port '0000:00:03.0' is detached. Now total ports is 2

testpmd> show port info all
testpmd> show config fwd
testpmd> show port stats all
```

VF traffic is seen at P1 (VF) and P2 (Bonded device).

1.4 Sample host scripts

1.4.1 setup_vf_on_212_46.sh

Set up Virtual Functions on host_server_1

```
#!/bin/sh
# This script is run on the host 10.237.212.46 to setup the VF

# set up Niantic VF
cat /sys/bus/pci/devices/0000\:09\:00.0/sriov_numvfs
echo 1 > /sys/bus/pci/devices/0000\:09\:00.0/sriov_numvfs
cat /sys/bus/pci/devices/0000\:09\:00.0/sriov_numvfs
rmmod ixgbevf

# set up Fortville VF
cat /sys/bus/pci/devices/0000\:02\:00.0/sriov_numvfs
echo 1 > /sys/bus/pci/devices/0000\:02\:00.0/sriov_numvfs
cat /sys/bus/pci/devices/0000\:02\:00.0/sriov_numvfs
rmmod i40evf
```

1.4.2 vm_virtio_vf_one_212_46.sh

Setup Virtual Machine on host_server_1

```
#!/bin/sh

# Path to KVM tool
KVM_PATH="/usr/bin/qemu-system-x86_64"

# Guest Disk image
DISK_IMG="/home/username/disk_image/virt1_sml.disk"

# Number of guest cpus
```

```
VCPUS_NR="4"

# Memory
MEM=1536

taskset -c 1-5 $KVM_PATH \
 -enable-kvm \
 -m $MEM \
 -smp $VCPUS_NR \
 -cpu host \
 -name VM1 \
 -no-reboot \
 -net none \
 -vnc none -nographic \
 -hda $DISK_IMG \
 -netdev type=tap,id=net1,script=no,downscript=no,ifname=tap1 \
 -device virtio-net-pci,netdev=net1,mac=CC:BB:BB:BB:BB:BB \
 -device pci-assign,host=09:10.0,id=vf1 \
 -monitor telnet::3333,server,nowait
```

1.4.3 setup_bridge_on_212_46.sh

Setup bridge on host_server_1

```
#!/bin/sh
# This script is run on the host 10.237.212.46 to setup the bridge
# for the Tap device and the PF device.
# This enables traffic to go from the PF to the Tap to the Virtio PMD in the VM.

# ens3f0 is the Niantic NIC
# ens6f0 is the Fortville NIC

ifconfig ens3f0 down
ifconfig tap1 down
ifconfig ens6f0 down
ifconfig virbr0 down

brctl show virbr0
brctl addif virbr0 ens3f0
brctl addif virbr0 ens6f0
brctl addif virbr0 tap1
brctl show virbr0

ifconfig ens3f0 up
ifconfig tap1 up
ifconfig ens6f0 up
ifconfig virbr0 up
```

1.4.4 connect_to_qemu_mon_on_host.sh

```
#!/bin/sh
# This script is run on both hosts when the VM is up,
# to connect to the Qemu Monitor.

telnet 0 3333
```

1.4.5 setup_vf_on_212_131.sh

Set up Virtual Functions on host_server_2

```
#!/bin/sh
# This script is run on the host 10.237.212.131 to setup the VF

# set up Niantic VF
cat /sys/bus/pci/devices/0000\:06\:00.0/sriov_numvfs
echo 1 > /sys/bus/pci/devices/0000\:06\:00.0/sriov_numvfs
cat /sys/bus/pci/devices/0000\:06\:00.0/sriov_numvfs
rmmod ixgbevf

# set up Fortville VF
cat /sys/bus/pci/devices/0000\:03\:00.0/sriov_numvfs
echo 1 > /sys/bus/pci/devices/0000\:03\:00.0/sriov_numvfs
cat /sys/bus/pci/devices/0000\:03\:00.0/sriov_numvfs
rmmod i40evf
```

1.4.6 vm_virtio_one_migrate.sh

Setup Virtual Machine on host_server_2

```
#!/bin/sh
# Start the VM on host_server_2 with the same parameters except without the VF
# parameters, as the VM on host_server_1, in migration-listen mode
# (-incoming tcp:0:5555)

# Path to KVM tool
KVM_PATH="/usr/bin/qemu-system-x86_64"

# Guest Disk image
DISK_IMG="/home/username/disk_image/virt1_sml.disk"

# Number of guest cpus
VCPUS_NR="4"

# Memory
MEM=1536

taskset -c 1-5 $KVM_PATH \
 -enable-kvm \
 -m $MEM \
 -smp $VCPUS_NR \
 -cpu host \
 -name VM1 \
 -no-reboot \
 -net none \
 -vnc none -nographic \
 -hda $DISK_IMG \
 -netdev type=tap,id=net1,script=no,downscript=no,ifname=tap1 \
 -device virtio-net-pci,netdev=net1,mac=CC:BB:BB:BB:BB:BB \
 -incoming tcp:0:5555 \
 -monitor telnet::3333,server,nowait
```

1.4.7 setup_bridge_on_212_131.sh

Setup bridge on host_server_2

```
#!/bin/sh
# This script is run on the host to setup the bridge
# for the Tap device and the PF device.
# This enables traffic to go from the PF to the Tap to the Virtio PMD in the VM.
```

```
# ens4f0 is the Niantic NIC
# ens5f0 is the Fortville NIC

ifconfig ens4f0 down
ifconfig tap1 down
ifconfig ens5f0 down
ifconfig virbr0 down

brctl show virbr0
brctl addif virbr0 ens4f0
brctl addif virbr0 ens5f0
brctl addif virbr0 tap1
brctl show virbr0

ifconfig ens4f0 up
ifconfig tap1 up
ifconfig ens5f0 up
ifconfig virbr0 up
```

1.5 Sample VM scripts

1.5.1 setup_dpdk_in_vm.sh

Set up DPDK in the Virtual Machine

```
#!/bin/sh
# this script matches the vm_virtio_vf_one script
# virtio port is 03
# vf port is 04

cat  /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
cat  /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages

ifconfig -a
/root/dpdk/usertools/dpdk-devbind.py --status

rmmod virtio-pci ixgbevf

modprobe uio
insmod /root/dpdk/x86_64-default-linux-gcc/kmod/igb_uio.ko

/root/dpdk/usertools/dpdk-devbind.py -b igb_uio 0000:00:03.0
/root/dpdk/usertools/dpdk-devbind.py -b igb_uio 0000:00:04.0

/root/dpdk/usertools/dpdk-devbind.py --status
```

1.5.2 run_testpmd_bonding_in_vm.sh

Run testpmd in the Virtual Machine.

```
#!/bin/sh
# Run testpmd in the VM

# The test system has 8 cpus (0-7), use cpus 2-7 for VM
# Use taskset -pc <core number> <thread_id>

# use for bonding of virtio and vf tests in VM
```

```
/root/dpdk/x86_64-default-linux-gcc/app/testpmd \
-l 0-3 -n 4 --socket-mem 350 --  --i --port-topology=chained
```

1.6 Sample switch configuration

The Intel switch is used to connect the traffic generator to the NIC's on host_server_1 and host_server_2.

In order to run the switch configuration two console windows are required.

Log in as root in both windows.

TestPointShared, run_switch.sh and load /root/switch_config must be executed in the sequence below.

1.6.1 On Switch: Terminal 1

run TestPointShared

```
/usr/bin/TestPointShared
```

1.6.2 On Switch: Terminal 2

execute run_switch.sh

```
/root/run_switch.sh
```

1.6.3 On Switch: Terminal 1

load switch configuration

```
load /root/switch_config
```

1.6.4 Sample switch configuration script

The `/root/switch_config` script:

```
# TestPoint History
show port 1,5,9,13,17,21,25
set port 1,5,9,13,17,21,25 up
show port 1,5,9,13,17,21,25
del acl 1
create acl 1
create acl-port-set
create acl-port-set
add port port-set 1 0
add port port-set 5,9,13,17,21,25 1
create acl-rule 1 1
add acl-rule condition 1 1 port-set 1
add acl-rule action 1 1 redirect 1
apply acl
create vlan 1000
add vlan port 1000 1,5,9,13,17,21,25
set vlan tagging 1000 1,5,9,13,17,21,25 tag
set switch config flood_ucast fwd
show port stats all 1,5,9,13,17,21,25
```

CHAPTER

TWO

LIVE MIGRATION OF VM WITH VIRTIO ON HOST RUNNING VHOST_USER

2.1 Overview

Live Migration of a VM with DPDK Virtio PMD on a host which is running the Vhost sample application (vhost-switch) and using the DPDK PMD (ixgbe or i40e).

The Vhost sample application uses VMDQ so SRIOV must be disabled on the NIC's.

The following sections show an example of how to do this migration.

2.2 Test Setup

To test the Live Migration two servers with identical operating systems installed are used. KVM and QEMU is also required on the servers.

QEMU 2.5 is required for Live Migration of a VM with vhost_user running on the hosts.

In this example, the servers have Niantic and or Fortville NIC's installed. The NIC's on both servers are connected to a switch which is also connected to the traffic generator.

The switch is configured to broadcast traffic on all the NIC ports.

The ip address of host_server_1 is 10.237.212.46

The ip address of host_server_2 is 10.237.212.131

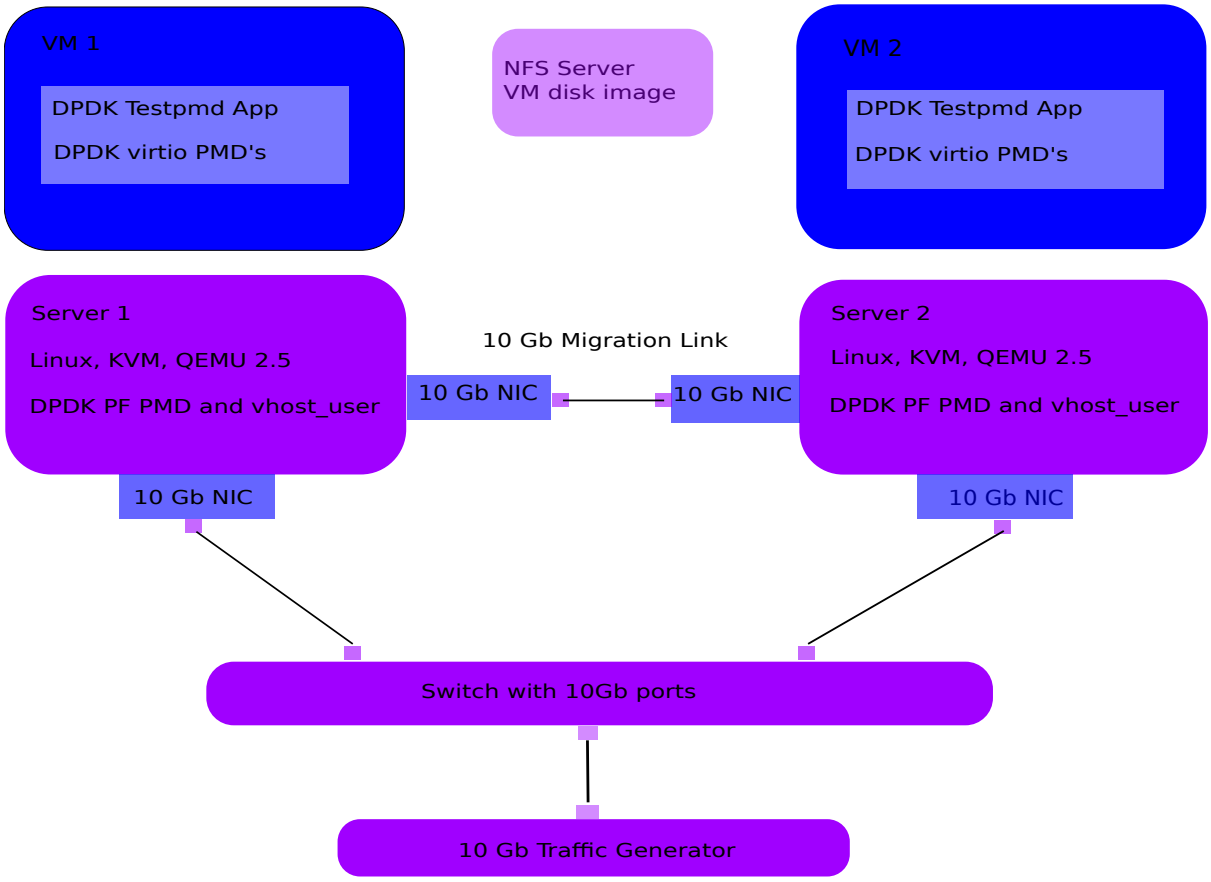
2.3 Live Migration steps

The sample scripts mentioned in the steps below can be found in the *Sample host scripts* and *Sample VM scripts* sections.

2.3.1 On host_server_1: Terminal 1

Setup DPDK on host_server_1

```
cd /root/dpdk/host_scripts
./setup_dpdk_on_host.sh
```



2.3.2 On host_server_1: Terminal 2

Bind the Niantic or Fortville NIC to igb_uio on host_server_1.

For Fortville NIC.

```
cd /root/dpdk/usertools
./dpdk-devbind.py -b igb_uio 0000:02:00.0
```

For Niantic NIC.

```
cd /root/dpdk/usertools
./dpdk-devbind.py -b igb_uio 0000:09:00.0
```

2.3.3 On host_server_1: Terminal 3

For Fortville and Niantic NIC's reset SRIOV and run the vhost_user sample application (vhost-switch) on host_server_1.

```
cd /root/dpdk/host_scripts
./reset_vf_on_212_46.sh
./run_vhost_switch_on_host.sh
```

2.3.4 On host_server_1: Terminal 1

Start the VM on host_server_1

```
./vm_virtio_vhost_user.sh
```

2.3.5 On host_server_1: Terminal 4

Connect to the QEMU monitor on host_server_1.

```
cd /root/dpdk/host_scripts
./connect_to_qemu_mon_on_host.sh
(qemu)
```

2.3.6 On host_server_1: Terminal 1

In VM on host_server_1:

Setup DPDK in the VM and run testpmd in the VM.

```
cd /root/dpdk/vm_scripts
./setup_dpdk_in_vm.sh
./run_testpmd_in_vm.sh

testpmd> show port info all
testpmd> set fwd mac retry
testpmd> start tx_first
testpmd> show port stats all
```

Virtio traffic is seen at P1 and P2.

2.3.7 On host_server_2: Terminal 1

Set up DPDK on the host_server_2.

```
cd /root/dpdk/host_scripts
./setup_dpdk_on_host.sh
```

2.3.8 On host_server_2: Terminal 2

Bind the Niantic or Fortville NIC to igb_uio on host_server_2.

For Fortville NIC.

```
cd /root/dpdk/usertools
./dpdk-devbind.py -b igb_uio 0000:03:00.0
```

For Niantic NIC.

```
cd /root/dpdk/usertools
./dpdk-devbind.py -b igb_uio 0000:06:00.0
```

2.3.9 On host_server_2: Terminal 3

For Fortville and Niantic NIC's reset SRIOV, and run the vhost_user sample application on host_server_2.

```
cd /root/dpdk/host_scripts
./reset_vf_on_212_131.sh
./run_vhost_switch_on_host.sh
```

2.3.10 On host_server_2: Terminal 1

Start the VM on host_server_2.

```
./vm_virtio_vhost_user_migrate.sh
```

2.3.11 On host_server_2: Terminal 4

Connect to the QEMU monitor on host_server_2.

```
cd /root/dpdk/host_scripts
./connect_to_qemu_mon_on_host.sh
(qemu) info status
VM status: paused (inmigrate)
(qemu)
```

2.3.12 On host_server_1: Terminal 4

Check that switch is up before migrating the VM.

```
(qemu) migrate tcp:10.237.212.131:5555
(qemu) info status
VM status: paused (postmigrate)

(qemu) info migrate
capabilities: xbzrle: off rdma-pin-all: off auto-converge: off zero-blocks: off
Migration status: completed
total time: 11619 milliseconds
downtime: 5 milliseconds
setup: 7 milliseconds
transferred ram: 379699 kbytes
throughput: 267.82 mbps
remaining ram: 0 kbytes
total ram: 1590088 kbytes
duplicate: 303985 pages
skipped: 0 pages
normal: 94073 pages
normal bytes: 376292 kbytes
dirty sync count: 2
(qemu) quit
```

2.3.13 On host_server_2: Terminal 1

In VM on host_server_2:

Hit Enter key. This brings the user to the testpmd prompt.

```
testpmd>
```

2.3.14 On host_server_2: Terminal 4

In QEMU monitor on host_server_2

```
(qemu) info status
VM status: running
```

2.3.15 On host_server_2: Terminal 1

In VM on host_server_2:

```
testpmd> show port info all
testpmd> show port stats all
```

Virtio traffic is seen at P0 and P1.

2.4 Sample host scripts

2.4.1 reset_vf_on_212_46.sh

```
#!/bin/sh
# This script is run on the host 10.237.212.46 to reset SRIOV

# BDF for Fortville NIC is 0000:02:00.0
cat /sys/bus/pci/devices/0000\:02\:00.0/max_vfs
echo 0 > /sys/bus/pci/devices/0000\:02\:00.0/max_vfs
cat /sys/bus/pci/devices/0000\:02\:00.0/max_vfs

# BDF for Niantic NIC is 0000:09:00.0
cat /sys/bus/pci/devices/0000\:09\:00.0/max_vfs
echo 0 > /sys/bus/pci/devices/0000\:09\:00.0/max_vfs
cat /sys/bus/pci/devices/0000\:09\:00.0/max_vfs
```

2.4.2 vm_virtio_vhost_user.sh

```
#/bin/sh
# Script for use with vhost_user sample application
# The host system has 8 cpu's (0-7)

# Path to KVM tool
KVM_PATH="/usr/bin/qemu-system-x86_64"

# Guest Disk image
DISK_IMG="/home/user/disk_image/virt1_sml.disk"

# Number of guest cpus
VCPUS_NR="6"

# Memory
MEM=1024

VIRTIO_OPTIONS="csum=off,gso=off,guest_tso4=off,guest_tso6=off,guest_ecn=off"

# Socket Path
SOCKET_PATH="/root/dpdk/host_scripts/usvhost"

taskset -c 2-7 $KVM_PATH \
 -enable-kvm \
 -m $MEM \
 -smp $VCPUS_NR \
 -object memory-backend-file,id=mem,size=1024M,mem-path=/mnt/huge,share=on \
 -numa node,memdev=mem,nodeid=0 \
 -cpu host \
 -name VM1 \
 -no-reboot \
 -net none \
```

```
 -vnc none \
 -nographic \
 -hda $DISK_IMG \
 -chardev socket,id=chr0,path=$SOCKET_PATH \
 -netdev type=vhost-user,id=net1,chardev=chr0,vhostforce \
 -device virtio-net-pci,netdev=net1,mac=CC:BB:BB:BB:BB:BB,$VIRTIO_OPTIONS \
 -chardev socket,id=chr1,path=$SOCKET_PATH \
 -netdev type=vhost-user,id=net2,chardev=chr1,vhostforce \
 -device virtio-net-pci,netdev=net2,mac=DD:BB:BB:BB:BB:BB,$VIRTIO_OPTIONS \
 -monitor telnet::3333,server,nowait
```

2.4.3 connect_to_qemu_mon_on_host.sh

```
#!/bin/sh
# This script is run on both hosts when the VM is up,
# to connect to the Qemu Monitor.

telnet 0 3333
```

2.4.4 reset_vf_on_212_131.sh

```
#!/bin/sh
# This script is run on the host 10.237.212.131 to reset SRIOV

# BDF for Niantic NIC is 0000:06:00.0
cat /sys/bus/pci/devices/0000\:06\:00.0/max_vfs
echo 0 > /sys/bus/pci/devices/0000\:06\:00.0/max_vfs
cat /sys/bus/pci/devices/0000\:06\:00.0/max_vfs

# BDF for Fortville NIC is 0000:03:00.0
cat /sys/bus/pci/devices/0000\:03\:00.0/max_vfs
echo 0 > /sys/bus/pci/devices/0000\:03\:00.0/max_vfs
cat /sys/bus/pci/devices/0000\:03\:00.0/max_vfs
```

2.4.5 vm_virtio_vhost_user_migrate.sh

```
#/bin/sh
# Script for use with vhost user sample application
# The host system has 8 cpu's (0-7)

# Path to KVM tool
KVM_PATH="/usr/bin/qemu-system-x86_64"

# Guest Disk image
DISK_IMG="/home/user/disk_image/virt1_sml.disk"

# Number of guest cpus
VCPUS_NR="6"

# Memory
MEM=1024

VIRTIO_OPTIONS="csum=off,gso=off,guest_tso4=off,guest_tso6=off,guest_ecn=off"

# Socket Path
SOCKET_PATH="/root/dpdk/host_scripts/usvhost"

taskset -c 2-7 $KVM_PATH \
 -enable-kvm \
```

```
-m $MEM \
-smp $VCPUS_NR \
-object memory-backend-file,id=mem,size=1024M,mem-path=/mnt/huge,share=on \
-numa node,memdev=mem,nodeid=0 \
-cpu host \
-name VM1 \
-no-reboot \
-net none \
-vnc none \
-nographic \
-hda $DISK_IMG \
-chardev socket,id=chr0,path=$SOCKET_PATH \
-netdev type=vhost-user,id=net1,chardev=chr0,vhostforce \
-device virtio-net-pci,netdev=net1,mac=CC:BB:BB:BB:BB:BB,$VIRTIO_OPTIONS \
-chardev socket,id=chr1,path=$SOCKET_PATH \
-netdev type=vhost-user,id=net2,chardev=chr1,vhostforce \
-device virtio-net-pci,netdev=net2,mac=DD:BB:BB:BB:BB:BB,$VIRTIO_OPTIONS \
-incoming tcp:0:5555 \
-monitor telnet::3333,server,nowait
```

2.5 Sample VM scripts

2.5.1 setup_dpdk_virtio_in_vm.sh

```
#!/bin/sh
# this script matches the vm_virtio_vhost_user script
# virtio port is 03
# virtio port is 04

cat  /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
cat  /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages

ifconfig -a
/root/dpdk/usertools/dpdk-devbind.py --status

rmmod virtio-pci

modprobe uio
insmod /root/dpdk/x86_64-default-linux-gcc/kmod/igb_uio.ko

/root/dpdk/usertools/dpdk-devbind.py -b igb_uio 0000:00:03.0
/root/dpdk/usertools/dpdk-devbind.py -b igb_uio 0000:00:04.0

/root/dpdk/usertools/dpdk-devbind.py --status
```

2.5.2 run_testpmd_in_vm.sh

```
#!/bin/sh
# Run testpmd for use with vhost_user sample app.
# test system has 8 cpus (0-7), use cpus 2-7 for VM

/root/dpdk/x86_64-default-linux-gcc/app/testpmd \
-l 0-5 -n 4 --socket-mem 350 -- --burst=64 --i
```

CHAPTER

THREE

FLOW BIFURCATION HOW-TO GUIDE

Flow Bifurcation is a mechanism which uses hardware capable Ethernet devices to split traffic between Linux user space and kernel space. Since it is a hardware assisted feature this approach can provide line rate processing capability. Other than KNI, the software is just required to enable device configuration, there is no need to take care of the packet movement during the traffic split. This can yield better performance with less CPU overhead.

The Flow Bifurcation splits the incoming data traffic to user space applications (such as DPDK applications) and/or kernel space programs (such as the Linux kernel stack). It can direct some traffic, for example data plane traffic, to DPDK, while directing some other traffic, for example control plane traffic, to the traditional Linux networking stack.

There are a number of technical options to achieve this. A typical example is to combine the technology of SR-IOV and packet classification filtering.

SR-IOV is a PCI standard that allows the same physical adapter to be split as multiple virtual functions. Each virtual function (VF) has separated queues with physical functions (PF). The network adapter will direct traffic to a virtual function with a matching destination MAC address. In a sense, SR-IOV has the capability for queue division.

Packet classification filtering is a hardware capability available on most network adapters. Filters can be configured to direct specific flows to a given receive queue by hardware. Different NICs may have different filter types to direct flows to a Virtual Function or a queue that belong to it.

In this way the Linux networking stack can receive specific traffic through the kernel driver while a DPDK application can receive specific traffic bypassing the Linux kernel by using drivers like VFIO or the DPDK `igb_uio` module.

3.1 Using Flow Bifurcation on Mellanox ConnectX

The Mellanox devices are natively bifurcated, so there is no need to split into SR-IOV PF/VF in order to get the flow bifurcation mechanism. The full device is already shared with the kernel driver.

The DPDK application can setup some flow steering rules, and let the rest go to the kernel stack. In order to define the filters strictly with flow rules, the flow_isolated_mode can be configured.

There is no specific instructions to follow. The recommended reading is the ../prog_guide/rte_flow guide. Below is an example of testpmd commands for receiving VXLAN 42 in 4 queues of the DPDK port 0, while all other packets go to the kernel:

```
testpmd> flow isolate 0 true
testpmd> flow create 0 ingress pattern eth / ipv4 / udp / vxlan vni is 42 / end \
         actions rss queues 0 1 2 3 end / end
```

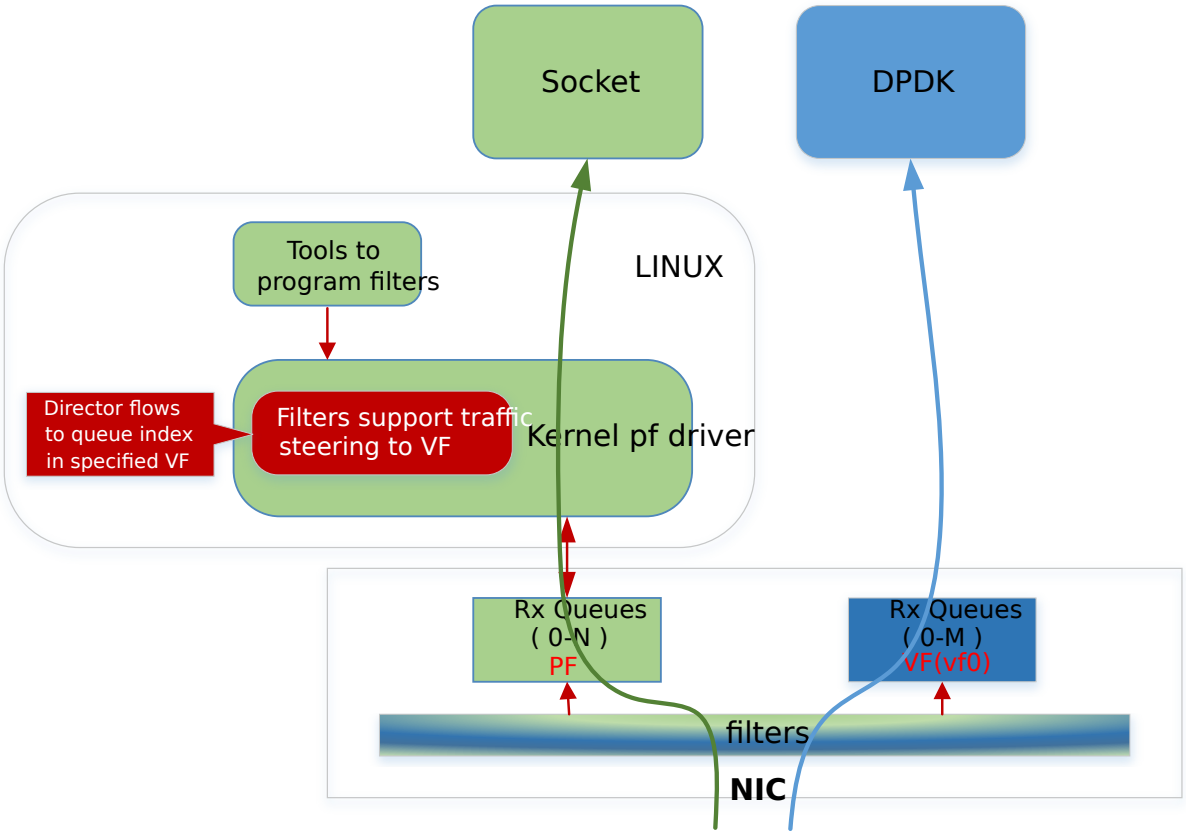


Fig. 3.1: Flow Bifurcation Overview

3.2 Using Flow Bifurcation on IXGBE in Linux

On Intel 82599 10 Gigabit Ethernet Controller series NICs Flow Bifurcation can be achieved by SR-IOV and Intel Flow Director technologies. Traffic can be directed to queues by the Flow Director capability, typically by matching 5-tuple of UDP/TCP packets.

The typical procedure to achieve this is as follows:

1. Boot the system without iommu, or with `iommu=pt`.
2. Create Virtual Functions:

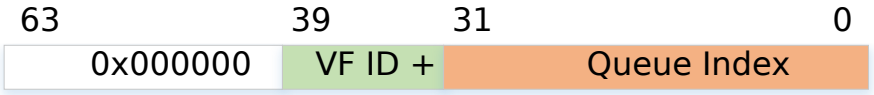
```
echo 2 > /sys/bus/pci/devices/0000:01:00.0/sriov_numvfs
```

3. Enable and set flow filters:

```
ethtool -K eth1 ntuple on
ethtool -N eth1 flow-type udp4 src-ip 192.0.2.2 dst-ip 198.51.100.2 \
        action $queue_index_in_VF0
ethtool -N eth1 flow-type udp4 src-ip 198.51.100.2 dst-ip 192.0.2.2 \
        action $queue_index_in_VF1
```

Where:

- `$queue_index_in_VFn`: Bits 39:32 of the variable defines VF id + 1; the lower 32 bits indicates the queue index of the VF. Thus:
  - `$queue_index_in_VF0 = (0x1 & 0xFF) << 32 + [queue index].`
  - `$queue_index_in_VF1 = (0x2 & 0xFF) << 32 + [queue index].`



4. Compile the DPDK application and insert `igb_uio` or probe the `vfio-pci` kernel modules as normal.
5. Bind the virtual functions:

```
modprobe vfio-pci
dpdk-devbind.py -b vfio-pci 01:10.0
dpdk-devbind.py -b vfio-pci 01:10.1
```

6. Run a DPDK application on the VFs:

```
testpmd -l 0-7 -n 4 -- -i -w 01:10.0 -w 01:10.1 --forward-mode=mac
```

In this example, traffic matching the rules will go through the VF by matching the filter rule. All other traffic, not matching the rules, will go through the default queue or scaling on queues in the PF. That is to say UDP packets with the specified IP source and destination addresses will go through the DPDK application. All other traffic, with different hosts or different protocols, will go through the Linux networking stack.

Note:

- The above steps work on the Linux kernel v4.2.
- The Flow Bifurcation is implemented in Linux kernel and ixgbe kernel driver using the following patches:
  - ethtool: Add helper routines to pass vf to rx_flow_spec
  - ixgbe: Allow flow director to use entire queue space
- The Ethtool version used in this example is 3.18.

3.3 Using Flow Bifurcation on I40E in Linux

On Intel X710/XL710 series Ethernet Controllers Flow Bifurcation can be achieved by SR-IOV, Cloud Filter and L3 VEB switch. The traffic can be directed to queues by the Cloud Filter and L3 VEB switch's matching rule.

- L3 VEB filters work for non-tunneled packets. It can direct a packet just by the Destination IP address to a queue in a VF.
- Cloud filters work for the following types of tunneled packets.
  - Inner mac.
  - Inner mac + VNI.
  - Outer mac + Inner mac + VNI.
  - Inner mac + Inner vlan + VNI.
  - Inner mac + Inner vlan.

The typical procedure to achieve this is as follows:

1. Boot the system without iommu, or with `iommu=pt`.
2. Build and insert the `i40e.ko` module.
3. Create Virtual Functions:

```
echo 2 > /sys/bus/pci/devices/0000:01:00.0/sriov_numvfs
```

4. Add udp port offload to the NIC if using cloud filter:

```
ip li add vxlan0 type vxlan id 42 group 239.1.1.1 local 10.16.43.214 dev <name>
ifconfig vxlan0 up
ip -d li show vxlan0
```

Note: Output such as `add vxlan port 8472,index 0 success` should be found in the system log.

---

5. Examples of enabling and setting flow filters:

   - L3 VEB filter, for a route whose destination IP is 192.168.50.108 to VF 0's queue 2.

```
ethtool -N <dev_name> flow-type ip4 dst-ip 192.168.50.108 \
        user-def 0xffffffff00000000 action 2 loc 8
```

   - Inner mac, for a route whose inner destination mac is 0:0:0:0:9:0 to PF's queue 6.

```
ethtool -N <dev_name> flow-type ether dst 00:00:00:00:00:00 \
        m ff:ff:ff:ff:ff:ff src 00:00:00:00:09:00 m 00:00:00:00:00:00 \
        user-def 0xffffffff00000003 action 6 loc 1
```

   - Inner mac + VNI, for a route whose inner destination mac is 0:0:0:0:9:0 and VNI is 8 to PF's queue 4.

```
ethtool -N <dev_name> flow-type ether dst 00:00:00:00:00:00 \
        m ff:ff:ff:ff:ff:ff src 00:00:00:00:09:00 m 00:00:00:00:00:00 \
        user-def 0x800000003 action 4 loc 4
```

   - Outer mac + Inner mac + VNI, for a route whose outer mac is 68:05:ca:24:03:8b, inner destination mac is c2:1a:e1:53:bc:57, and VNI is 8 to PF's queue 2.

```
ethtool -N <dev_name> flow-type ether dst 68:05:ca:24:03:8b \
        m 00:00:00:00:00:00 src c2:1a:e1:53:bc:57 m 00:00:00:00:00:00 \
        user-def 0x800000003 action 2 loc 2
```

   - Inner mac + Inner vlan + VNI, for a route whose inner destination mac is 00:00:00:00:20:00, inner vlan is 10, and VNI is 8 to VF 0's queue 1.

```
ethtool -N <dev_name> flow-type ether dst 00:00:00:00:01:00 \
        m ff:ff:ff:ff:ff:ff src 00:00:00:00:20:00 m 00:00:00:00:00:00 \
        vlan 10 user-def 0x800000000 action 1 loc 5
```

   - Inner mac + Inner vlan, for a route whose inner destination mac is 00:00:00:00:20:00, and inner vlan is 10 to VF 0's queue 1.

```
ethtool -N <dev_name> flow-type ether dst 00:00:00:00:01:00 \
        m ff:ff:ff:ff:ff:ff src 00:00:00:00:20:00 m 00:00:00:00:00:00 \
        vlan 10 user-def 0xffffffff00000000 action 1 loc 5
```

Note:

- If the upper 32 bits of 'user-def' are `0xffffffff`, then the filter can be used for programming an L3 VEB filter, otherwise the upper 32 bits of 'user-def' can carry the tenant ID/VNI if specified/required.

- Cloud filters can be defined with inner mac, outer mac, inner ip, inner vlan and VNI as part of the cloud tuple. It is always the destination (not source) mac/ip that these filters use. For all these examples dst and src mac address fields are overloaded dst == outer, src == inner.
- The filter will direct a packet matching the rule to a vf id specified in the lower 32 bit of user-def to the queue specified by 'action'.
- If the vf id specified by the lower 32 bit of user-def is greater than or equal to `max_vfs`, then the filter is for the PF queues.

6. Compile the DPDK application and insert `igb_uio` or probe the `vfio-pci` kernel modules as normal.
7. Bind the virtual function:

```
modprobe vfio-pci
dpdk-devbind.py -b vfio-pci 01:10.0
dpdk-devbind.py -b vfio-pci 01:10.1
```

8. run DPDK application on VFs:

```
testpmd -l 0-7 -n 4 -- -i -w 01:10.0 -w 01:10.1 --forward-mode=mac
```

Note:

- The above steps work on the i40e Linux kernel driver v1.5.16.
- The Ethtool version used in this example is 3.18. The mask `ff` means 'not involved', while `00` or no mask means 'involved'.
- For more details of the configuration, refer to the cloud filter test plan

CHAPTER

FOUR

GENERIC FLOW API - EXAMPLES

This document demonstrates some concrete examples for programming flow rules with the `rte_flow` APIs.

- Detail of the rte_flow APIs can be found in the following link: ../prog_guide/rte_flow.
- Details of the TestPMD commands to set the flow rules can be found in the following link: TestPMD Flow rules

4.1 Simple IPv4 drop

4.1.1 Description

In this example we will create a simple rule that drops packets whose IPv4 destination equals 192.168.3.2. This code is equivalent to the following testpmd command (wrapped for clarity):

```
testpmd> flow create 0 ingress pattern eth / vlan /
                  ipv4 dst is 192.168.3.2 / end actions drop / end
```

4.1.2 Code

```
/* create the attribute structure */
struct rte_flow_attr attr = { .ingress = 1 };
struct rte_flow_item pattern[MAX_PATTERN_IN_FLOW];
struct rte_flow_action actions[MAX_ACTIONS_IN_FLOW];
struct rte_flow_item_eth eth;
struct rte_flow_item_vlan vlan;
struct rte_flow_item_ipv4 ipv4;
struct rte_flow *flow;
struct rte_flow_error error;

/* setting the eth to pass all packets */
pattern[0].type = RTE_FLOW_ITEM_TYPE_ETH;
pattern[0].spec = &eth;

/* set the vlan to pass all packets */
pattern[1] = RTE_FLOW_ITEM_TYPE_VLAN;
pattern[1].spec = &vlan;

/* set the dst ipv4 packet to the required value */
ipv4.hdr.dst_addr = htonl(0xc0a80302);
pattern[2].type = RTE_FLOW_ITEM_TYPE_IPV4;
pattern[2].spec = &ipv4;

/* end the pattern array */
```

```
pattern[3].type = RTE_FLOW_ITEM_TYPE_END;

/* create the drop action */
actions[0].type = RTE_FLOW_ACTION_TYPE_DROP;
actions[1].type = RTE_FLOW_ACTION_TYPE_END;

/* validate and create the flow rule */
if (!rte_flow_validate(port_id, &attr, pattern, actions, &error))
    flow = rte_flow_create(port_id, &attr, pattern, actions, &error);
```

4.1.3 Output

Terminal 1: running sample app with the flow rule disabled:

```
./filter-program disable
[waiting for packets]
```

Terminal 2: running scapy:

```
$scapy
welcome to Scapy
>> sendp(Ether()/Dot1Q()/IP(src='176.80.50.4', dst='192.168.3.1'), \
         iface='some interface', count=1)
>> sendp(Ether()/Dot1Q()/IP(src='176.80.50.5', dst='192.168.3.2'), \
         iface='some interface', count=1)
```

Terminal 1: output log:

```
received packet with src ip = 176.80.50.4
received packet with src ip = 176.80.50.5
```

Terminal 1: running sample the app flow rule enabled:

```
./filter-program enabled
[waiting for packets]
```

Terminal 2: running scapy:

```
$scapy
welcome to Scapy
>> sendp(Ether()/Dot1Q()/IP(src='176.80.50.4', dst='192.168.3.1'),  \
         iface='some interface', count=1)
>> sendp(Ether()/Dot1Q()/IP(src='176.80.50.5', dst ='192.168.3.2'), \
         iface='some interface', count=1)
```

Terminal 1: output log:

```
received packet with src ip = 176.80.50.4
```

4.2 Range IPv4 drop

4.2.1 Description

In this example we will create a simple rule that drops packets whose IPv4 destination is in the range 192.168.3.0 to 192.168.3.255. This is done using a mask.

This code is equivalent to the following testpmd command (wrapped for clarity):

```
testpmd> flow create 0 ingress pattern eth / vlan /
                  ipv4 dst spec 192.168.3.0 dst mask 255.255.255.0 /
                  end actions drop / end
```

4.2.2 Code

```
struct rte_flow_attr attr = {.ingress = 1};
struct rte_flow_item pattern[MAX_PATTERN_IN_FLOW];
struct rte_flow_action actions[MAX_ACTIONS_IN_FLOW];
struct rte_flow_item_eth eth;
struct rte_flow_item_vlan vlan;
struct rte_flow_item_ipv4 ipv4;
struct rte_flow_item_ipv4 ipv4_mask;
struct rte_flow *flow;
struct rte_flow_error error;

/* setting the eth to pass all packets */
pattern[0].type = RTE_FLOW_ITEM_TYPE_ETH;
pattern[0].spec = &eth;

/* set the vlan to pass all packets */
pattern[1] = RTE_FLOW_ITEM_TYPE_VLAN;
pattern[1].spec = &vlan;

/* set the dst ipv4 packet to the required value */
ipv4.hdr.dst_addr = htonl(0xc0a80300);
ipv4_mask.hdr.dst_addr = htonl(0xffffff00);
pattern[2].type = RTE_FLOW_ITEM_TYPE_IPV4;
pattern[2].spec = &ipv4;
pattern[2].mask = &ipv4_mask;

/* end the pattern array */
pattern[3].type = RTE_FLOW_ITEM_TYPE_END;

/* create the drop action */
actions[0].type = RTE_FLOW_ACTION_TYPE_DROP;
actions[1].type = RTE_FLOW_ACTION_TYPE_END;

/* validate and create the flow rule */
if (!rte_flow_validate(port_id, &attr, pattern, actions, &error))
    flow = rte_flow_create(port_id, &attr, pattern, actions, &error);
```

4.2.3 Output

Terminal 1: running sample app flow rule disabled:

```
./filter-program disable
[waiting for packets]
```

Terminal 2: running scapy:

```
$scapy
welcome to Scapy
>> sendp(Ether()/Dot1Q()/IP(src='176.80.50.4', dst='192.168.3.1'), \
        iface='some interface', count=1)
>> sendp(Ether()/Dot1Q()/IP(src='176.80.50.5', dst='192.168.3.2'), \
        iface='some interface', count=1)
>> sendp(Ether()/Dot1Q()/IP(src='176.80.50.6', dst='192.168.5.2'), \
        iface='some interface', count=1)
```

Terminal 1: output log:

```
received packet with src ip = 176.80.50.4
received packet with src ip = 176.80.50.5
received packet with src ip = 176.80.50.6
```

Terminal 1: running sample app flow rule enabled:

```
./filter-program enabled
[waiting for packets]
```

Terminal 2: running scapy:

```
$scapy
welcome to Scapy
>> sendp(Ether()/Dot1Q()/IP(src='176.80.50.4', dst='192.168.3.1'), \
         iface='some interface', count=1)
>> sendp(Ether()/Dot1Q()/IP(src='176.80.50.5', dst='192.168.3.2'), \
         iface='some interface', count=1)
>> sendp(Ether()/Dot1Q()/IP(src='176.80.50.6', dst='192.168.5.2'), \
         iface='some interface', count=1)
```

Terminal 1: output log:

```
received packet with src ip = 176.80.50.6
```

4.3 Send vlan to queue

4.3.1 Description

In this example we will create a rule that routes all vlan id 123 to queue 3.

This code is equivalent to the following testpmd command (wrapped for clarity):

```
testpmd> flow create 0 ingress pattern eth / vlan vid spec 123 /
                 end actions queue index 3 / end
```

4.3.2 Code

```
struct rte_flow_attr attr = { .ingress = 1 };
struct rte_flow_item pattern[MAX_PATTERN_IN_FLOW];
struct rte_flow_action actions[MAX_ACTIONS_IN_FLOW];
struct rte_flow_item_eth eth;
struct rte_flow_item_vlan vlan;
struct rte_flow_action_queue queue = { .index = 3 };
struct rte_flow *flow;
struct rte_flow_error error;

/* setting the eth to pass all packets */
pattern[0].type = RTE_FLOW_ITEM_TYPE_ETH;
pattern[0].spec = &eth;

/* set the vlan to pas all packets */
vlan.vid = 123;
pattern[1] = RTE_FLOW_ITEM_TYPE_VLAN;
pattern[1].spec = &vlan;

/* end the pattern array */
pattern[2].type = RTE_FLOW_ITEM_TYPE_END;

/* create the queue action */
actions[0].type = RTE_FLOW_ACTION_TYPE_QUEUE;
actions[0].conf = &queue;
actions[1].type = RTE_FLOW_ACTION_TYPE_END;

/* validate and create the flow rule */
if (!rte_flow_validate(port_id, &attr, pattern, actions, &error))
    flow = rte_flow_create(port_id, &attr, pattern, actions, &error);
```

4.3.3 Output

Terminal 1: running sample app flow rule disabled:

```
./filter-program disable
[waiting for packets]
```

Terminal 2: running scapy:

```
$scapy
welcome to Scapy
>> sendp(Ether()/Dot1Q(vlan=123)/IP(src='176.80.50.4', dst='192.168.3.1'), \
         iface='some interface', count=1)
>> sendp(Ether()/Dot1Q(vlan=50)/IP(src='176.80.50.5', dst='192.168.3.2'),  \
         iface='some interface', count=1)
>> sendp(Ether()/Dot1Q(vlan=123)/IP(src='176.80.50.6', dst='192.168.5.2'), \
         iface='some interface', count=1)
```

Terminal 1: output log:

```
received packet with src ip = 176.80.50.4 sent to queue 2
received packet with src ip = 176.80.50.5 sent to queue 1
received packet with src ip = 176.80.50.6 sent to queue 0
```

Terminal 1: running sample app flow rule enabled:

```
./filter-program enabled
[waiting for packets]
```

Terminal 2: running scapy:

```
$scapy
welcome to Scapy
>> sendp(Ether()/Dot1Q(vlan=123)/IP(src='176.80.50.4', dst='192.168.3.1'), \
         iface='some interface', count=1)
>> sendp(Ether()/Dot1Q(vlan=50)/IP(src='176.80.50.5', dst='192.168.3.2'),  \
         iface='some interface', count=1)
>> sendp(Ether()/Dot1Q(vlan=123)/IP(src='176.80.50.6', dst='192.168.5.2'), \
         iface='some interface', count=1)
```

Terminal 1: output log:

```
received packet with src ip = 176.80.50.4 sent to queue 3
received packet with src ip = 176.80.50.5 sent to queue 1
received packet with src ip = 176.80.50.6 sent to queue 3
```

CHAPTER

FIVE

PVP REFERENCE BENCHMARK SETUP USING TESTPMD

This guide lists the steps required to setup a PVP benchmark using testpmd as a simple forwarder between NICs and Vhost interfaces. The goal of this setup is to have a reference PVP benchmark without using external vSwitches (OVS, VPP, ...) to make it easier to obtain reproducible results and to facilitate continuous integration testing.

The guide covers two ways of launching the VM, either by directly calling the QEMU command line, or by relying on libvirt. It has been tested with DPDK v16.11 using RHEL7 for both host and guest.

5.1 Setup overview

In this diagram, each red arrow represents one logical core. This use-case requires 6 dedicated logical cores. A forwarding configuration with a single NIC is also possible, requiring 3 logical cores.

5.2 Host setup

In this setup, we isolate 6 cores (from CPU2 to CPU7) on the same NUMA node. Two cores are assigned to the VM vCPUs running testpmd and four are assigned to testpmd on the host.

5.2.1 Host tuning

1. On BIOS, disable turbo-boost and hyper-threads.
2. Append these options to Kernel command line:

```
intel_pstate=disable mce=ignore_ce default_hugepagesz=1G hugepagesz=1G hugepages=6 isolcpu
```

3. Disable hyper-threads at runtime if necessary or if BIOS is not accessible:

```
cat /sys/devices/system/cpu/cpu*[0-9]/topology/thread_siblings_list \
    | sort | uniq \
    | awk -F, '{system("echo 0 > /sys/devices/system/cpu/cpu"$2"/online")}'
```

4. Disable NMIs:

```
echo 0 > /proc/sys/kernel/nmi_watchdog
```

5. Exclude isolated CPUs from the writeback cpumask:

```
echo ffffff03 > /sys/bus/workqueue/devices/writeback/cpumask
```

6. Isolate CPUs from IRQs:

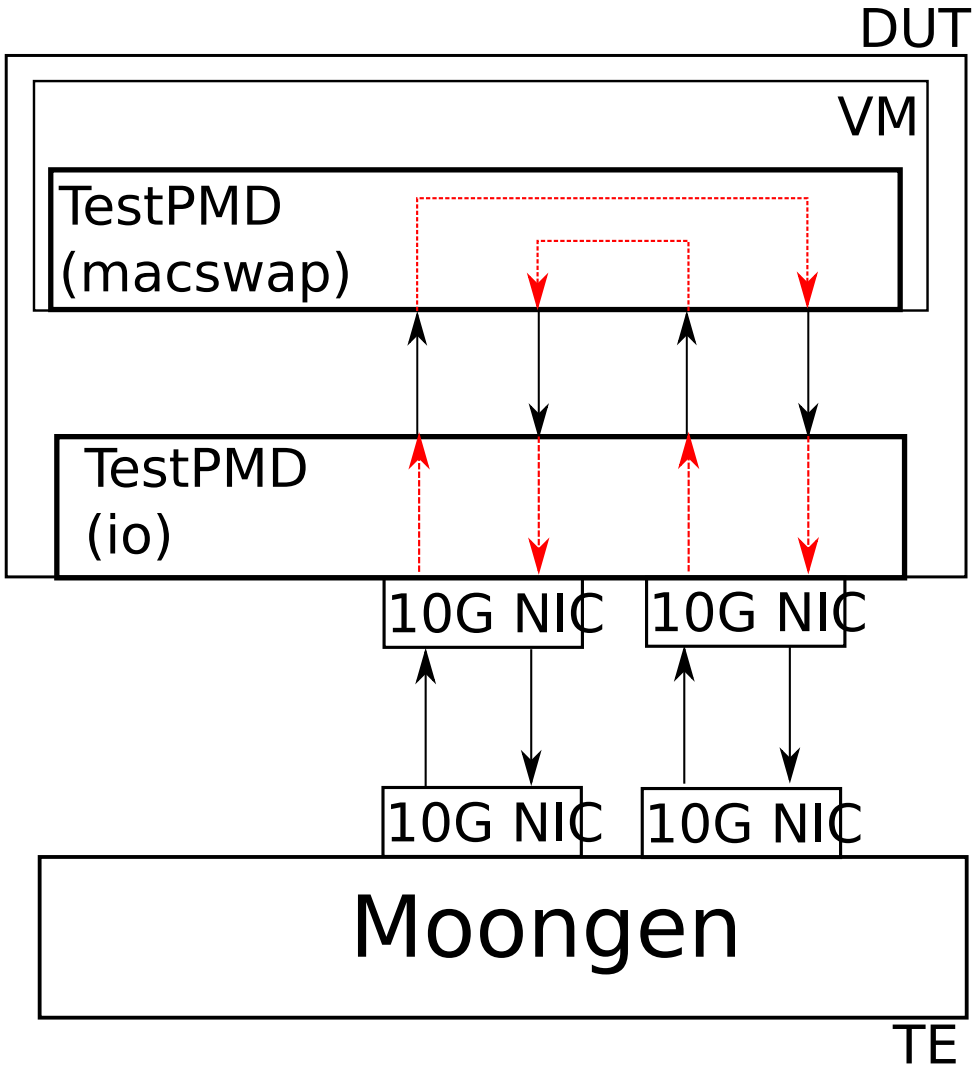


Fig. 5.1: PVP setup using 2 NICs

```
clear_mask=0xfc #Isolate CPU2 to CPU7 from IRQs
for i in /proc/irq/*/smp_affinity
do
  echo "obase=16;$(( 0x$(cat $i) & ~$clear_mask ))" | bc > $i
done
```

5.2.2 Qemu build

Build Qemu:

```
git clone git://git.qemu.org/qemu.git
cd qemu
mkdir bin
cd bin
../configure --target-list=x86_64-softmmu
make
```

5.2.3 DPDK build

Build DPDK:

```
git clone git://dpdk.org/dpdk
cd dpdk
export RTE_SDK=$PWD
make install T=x86_64-native-linux-gcc DESTDIR=install
```

5.2.4 Testpmd launch

1. Assign NICs to DPDK:

   ```
   modprobe vfio-pci
   $RTE_SDK/install/sbin/dpdk-devbind -b vfio-pci 0000:11:00.0 0000:11:00.1
   ```

   Note: The Sandy Bridge family seems to have some IOMMU limitations giving poor performance results. To achieve good performance on these machines consider using UIO instead.

2. Launch the testpmd application:

   ```
   $RTE_SDK/install/bin/testpmd -l 0,2,3,4,5 --socket-mem=1024 -n 4 \
       --vdev 'net_vhost0,iface=/tmp/vhost-user1' \
       --vdev 'net_vhost1,iface=/tmp/vhost-user2' -- \
       --portmask=f -i --rxq=1 --txq=1 \
       --nb-cores=4 --forward-mode=io
   ```

   With this command, isolated CPUs 2 to 5 will be used as lcores for PMD threads.

3. In testpmd interactive mode, set the portlist to obtain the correct port chaining:

   ```
   set portlist 0,2,1,3
   start
   ```

5.2.5 VM launch

The VM may be launched either by calling QEMU directly, or by using libvirt.

Qemu way

Launch QEMU with two Virtio-net devices paired to the vhost-user sockets created by testpmd. Below example uses default Virtio-net options, but options may be specified, for example to disable mergeable buffers or indirect descriptors.

```
<QEMU path>/bin/x86_64-softmmu/qemu-system-x86_64 \
    -enable-kvm -cpu host -m 3072 -smp 3 \
    -chardev socket,id=char0,path=/tmp/vhost-user1 \
    -netdev type=vhost-user,id=mynet1,chardev=char0,vhostforce \
    -device virtio-net-pci,netdev=mynet1,mac=52:54:00:02:d9:01,addr=0x10 \
    -chardev socket,id=char1,path=/tmp/vhost-user2 \
    -netdev type=vhost-user,id=mynet2,chardev=char1,vhostforce \
    -device virtio-net-pci,netdev=mynet2,mac=52:54:00:02:d9:02,addr=0x11 \
    -object memory-backend-file,id=mem,size=3072M,mem-path=/dev/hugepages,share=on \
    -numa node,memdev=mem -mem-prealloc \
    -net user,hostfwd=tcp::1002$1-:22 -net nic \
    -qmp unix:/tmp/qmp.socket,server,nowait \
    -monitor stdio <vm_image>.qcow2
```

You can use this qmp-vcpu-pin script to pin vCPUs.

It can be used as follows, for example to pin 3 vCPUs to CPUs 1, 6 and 7, where isolated CPUs 6 and 7 will be used as lcores for Virtio PMDs:

```
export PYTHONPATH=$PYTHONPATH:<QEMU path>/scripts/qmp
./qmp-vcpu-pin -s /tmp/qmp.socket 1 6 7
```

Libvirt way

Some initial steps are required for libvirt to be able to connect to testpmd's sockets.

First, SELinux policy needs to be set to permissive, since testpmd is generally run as root (note, as reboot is required):

```
cat /etc/selinux/config

# This file controls the state of SELinux on the system.
# SELINUX= can take one of these three values:
#     enforcing  - SELinux security policy is enforced.
#     permissive - SELinux prints warnings instead of enforcing.
#     disabled   - No SELinux policy is loaded.
SELINUX=permissive

# SELINUXTYPE= can take one of three two values:
#     targeted - Targeted processes are protected,
#     minimum  - Modification of targeted policy.
#                Only selected processes are protected.
#     mls      - Multi Level Security protection.
SELINUXTYPE=targeted
```

Also, Qemu needs to be run as root, which has to be specified in `/etc/libvirt/qemu.conf`:

```
user = "root"
```

Once the domain created, the following snippet is an extract of he most important information (hugepages, vCPU pinning, Virtio PCI devices):

```
<domain type='kvm'>
  <memory unit='KiB'>3145728</memory>
  <currentMemory unit='KiB'>3145728</currentMemory>
  <memoryBacking>
```

```
    <hugepages>
      <page size='1048576' unit='KiB' nodeset='0'/>
    </hugepages>
    <locked/>
  </memoryBacking>
  <vcpu placement='static'>3</vcpu>
  <cputune>
    <vcpupin vcpu='0' cpuset='1'/>
    <vcpupin vcpu='1' cpuset='6'/>
    <vcpupin vcpu='2' cpuset='7'/>
    <emulatorpin cpuset='0'/>
  </cputune>
  <numatune>
    <memory mode='strict' nodeset='0'/>
  </numatune>
  <os>
    <type arch='x86_64' machine='pc-i440fx-rhel7.0.0'>hvm</type>
    <boot dev='hd'/>
  </os>
  <cpu mode='host-passthrough'>
    <topology sockets='1' cores='3' threads='1'/>
    <numa>
      <cell id='0' cpus='0-2' memory='3145728' unit='KiB' memAccess='shared'/>
    </numa>
  </cpu>
  <devices>
    <interface type='vhostuser'>
      <mac address='56:48:4f:53:54:01'/>
      <source type='unix' path='/tmp/vhost-user1' mode='client'/>
      <model type='virtio'/>
      <driver name='vhost' rx_queue_size='256' />
      <address type='pci' domain='0x0000' bus='0x00' slot='0x10' function='0x0'/>
    </interface>
    <interface type='vhostuser'>
      <mac address='56:48:4f:53:54:02'/>
      <source type='unix' path='/tmp/vhost-user2' mode='client'/>
      <model type='virtio'/>
      <driver name='vhost' rx_queue_size='256' />
      <address type='pci' domain='0x0000' bus='0x00' slot='0x11' function='0x0'/>
    </interface>
  </devices>
</domain>
```

5.3 Guest setup

5.3.1 Guest tuning

1. Append these options to the Kernel command line:

   ```
   default_hugepagesz=1G hugepagesz=1G hugepages=1 intel_iommu=on iommu=pt isolcpus=1,2 rcu_n
   ```

2. Disable NMIs:

   ```
   echo 0 > /proc/sys/kernel/nmi_watchdog
   ```

3. Exclude isolated CPU1 and CPU2 from the writeback cpumask:

   ```
   echo 1 > /sys/bus/workqueue/devices/writeback/cpumask
   ```

4. Isolate CPUs from IRQs:

```
clear_mask=0x6 #Isolate CPU1 and CPU2 from IRQs
for i in /proc/irq/*/smp_affinity
do
  echo "obase=16;$(( 0x$(cat $i) & ~$clear_mask ))" | bc > $i
done
```

5.3.2 DPDK build

Build DPDK:

```
git clone git://dpdk.org/dpdk
cd dpdk
export RTE_SDK=$PWD
make install T=x86_64-native-linux-gcc DESTDIR=install
```

5.3.3 Testpmd launch

Probe vfio module without iommu:

```
modprobe -r vfio_iommu_type1
modprobe -r vfio
modprobe  vfio enable_unsafe_noiommu_mode=1
cat /sys/module/vfio/parameters/enable_unsafe_noiommu_mode
modprobe vfio-pci
```

Bind the virtio-net devices to DPDK:

```
$RTE_SDK/usertools/dpdk-devbind.py -b vfio-pci 0000:00:10.0 0000:00:11.0
```

Start testpmd:

```
$RTE_SDK/install/bin/testpmd -l 0,1,2 --socket-mem 1024 -n 4 \
    --proc-type auto --file-prefix pg -- \
    --portmask=3 --forward-mode=macswap --port-topology=chained \
    --disable-rss -i --rxq=1 --txq=1 \
    --rxd=256 --txd=256 --nb-cores=2 --auto-start
```

5.4 Results template

Below template should be used when sharing results:

```
Traffic Generator: <Test equipment (e.g. IXIA, Moongen, ...)>
Acceptable Loss: <n>%
Validation run time: <n>min
Host DPDK version/commit: <version, SHA-1>
Guest DPDK version/commit: <version, SHA-1>
Patches applied: <link to patchwork>
QEMU version/commit: <version>
Virtio features: <features (e.g. mrg_rxbuf='off', leave empty if default)>
CPU: <CPU model>, <CPU frequency>
NIC: <NIC model>
Result: <n> Mpps
```

CHAPTER

SIX

VF DAEMON (VFD)

VFd (the VF daemon) is a mechanism which can be used to configure features on a VF (SR-IOV Virtual Function) without direct access to the PF (SR-IOV Physical Function). VFd is an *EXPERIMENTAL* feature which can only be used in the scenario of DPDK PF with a DPDK VF. If the PF port is driven by the Linux kernel driver then the VFd feature will not work. Currently VFd is only supported by the ixgbe and i40e drivers.

In general VF features cannot be configured directly by an end user application since they are under the control of the PF. The normal approach to configuring a feature on a VF is that an application would call the APIs provided by the VF driver. If the required feature cannot be configured by the VF directly (the most common case) the VF sends a message to the PF through the mailbox on ixgbe and i40e. This means that the availability of the feature depends on whether the appropriate mailbox messages are defined.

DPDK leverages the mailbox interface defined by the Linux kernel driver so that compatibility with the kernel driver can be guaranteed. The downside of this approach is that the availability of messages supported by the kernel become a limitation when the user wants to configure features on the VF.

VFd is a new method of controlling the features on a VF. The VF driver doesn't talk directly to the PF driver when configuring a feature on the VF. When a VF application (i.e., an application using the VF ports) wants to enable a VF feature, it can send a message to the PF application (i.e., the application using the PF port, which can be the same as the VF application). The PF application will configure the feature for the VF. Obviously, the PF application can also configure the VF features without a request from the VF application.

Compared with the traditional approach the VFd moves the negotiation between VF and PF from the driver level to application level. So the application should define how the negotiation between the VF and PF works, or even if the control should be limited to the PF.

It is the application's responsibility to use VFd. Consider for example a KVM migration, the VF application may transfer from one VM to another. It is recommended in this case that the PF control the VF features without participation from the VF. Then the VF application has no capability to configure the features. So the user doesn't need to define the interface between the VF application and the PF application. The service provider should take the control of all the features.

The following sections describe the VFd functionality.

Note: Although VFd is supported by both ixgbe and i40e, please be aware that since the hardware capability is different, the functions supported by ixgbe and i40e are not the same.

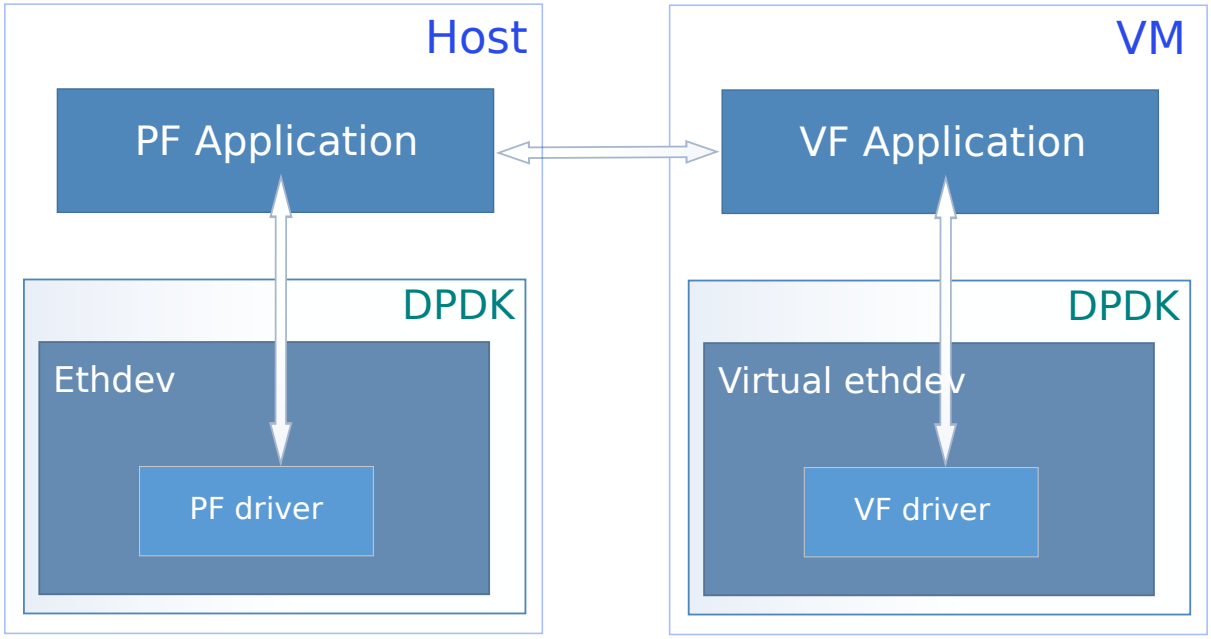


Fig. 6.1: VF daemon (VFd) Overview

6.1 Preparing

VFd only can be used in the scenario of DPDK PF + DPDK VF. Users should bind the PF port to `igb_uio`, then create the VFs based on the DPDK PF host.

The typical procedure to achieve this is as follows:

1. Boot the system without iommu, or with `iommu=pt`.
2. Bind the PF port to `igb_uio`, for example:

```
dpdk-devbind.py -b igb_uio 01:00.0
```

3. Create a Virtual Function:

```
echo 1 > /sys/bus/pci/devices/0000:01:00.0/max_vfs
```

4. Start a VM with the new VF port bypassed to it.
5. Run a DPDK application on the PF in the host:

```
testpmd -l 0-7 -n 4 -- -i --txqflags=0
```

6. Bind the VF port to `igb_uio` in the VM:

```
dpdk-devbind.py -b igb_uio 03:00.0
```

7. Run a DPDK application on the VF in the VM:

```
testpmd -l 0-7 -n 4 -- -i --txqflags=0
```

6.2 Common functions of IXGBE and I40E

The following sections show how to enable PF/VF functionality based on the above testpmd setup.

6.2.1 TX loopback

Run a testpmd runtime command on the PF to set TX loopback:

```
set tx loopback 0 on|off
```

This sets whether the PF port and all the VF ports that belong to it are allowed to send the packets to other virtual ports.

Although it is a VFd function, it is the global setting for the whole physical port. When using this function, the PF and all the VFs TX loopback will be enabled/disabled.

6.2.2 VF MAC address setting

Run a testpmd runtime command on the PF to set the MAC address for a VF port:

```
set vf mac addr 0 0 A0:36:9F:7B:C3:51
```

This testpmd runtime command will change the MAC address of the VF port to this new address. If any other addresses are set before, they will be overwritten.

6.2.3 VF MAC anti-spoofing

Run a testpmd runtime command on the PF to enable/disable the MAC anti-spoofing for a VF port:

```
set vf mac antispoof 0 0 on|off
```

When enabling the MAC anti-spoofing, the port will not forward packets whose source MAC address is not the same as the port.

6.2.4 VF VLAN anti-spoofing

Run a testpmd runtime command on the PF to enable/disable the VLAN anti-spoofing for a VF port:

```
set vf vlan antispoof 0 0 on|off
```

When enabling the VLAN anti-spoofing, the port will not send packets whose VLAN ID does not belong to VLAN IDs that this port can receive.

6.2.5 VF VLAN insertion

Run a testpmd runtime command on the PF to set the VLAN insertion for a VF port:

```
set vf vlan insert 0 0 1
```

When using this testpmd runtime command, an assigned VLAN ID can be inserted to the transmitted packets by the hardware.

The assigned VLAN ID can be 0. It means disabling the VLAN insertion.

6.2.6 VF VLAN stripping

Run a testpmd runtime command on the PF to enable/disable the VLAN stripping for a VF port:

```
set vf vlan stripq 0 0 on|off
```

This testpmd runtime command is used to enable/disable the RX VLAN stripping for a specific VF port.

6.2.7 VF VLAN filtering

Run a testpmd runtime command on the PF to set the VLAN filtering for a VF port:

```
rx_vlan add 1 port 0 vf 1
rx_vlan rm  1 port 0 vf 1
```

These two testpmd runtime commands can be used to add or remove the VLAN filter for several VF ports. When the VLAN filters are added only the packets that have the assigned VLAN IDs can be received. Other packets will be dropped by hardware.

6.3 The IXGBE specific VFd functions

The functions in this section are specific to the ixgbe driver.

6.3.1 All queues drop

Run a testpmd runtime command on the PF to enable/disable the all queues drop:

```
set all queues drop on|off
```

This is a global setting for the PF and all the VF ports of the physical port.

Enabling the `all queues drop` feature means that when there is no available descriptor for the received packets they are dropped. The `all queues drop` feature should be enabled in SR-IOV mode to avoid one queue blocking others.

6.3.2 VF packet drop

Run a testpmd runtime command on the PF to enable/disable the packet drop for a specific VF:

```
set vf split drop 0 0 on|off
```

This is a similar function as `all queues drop`. The difference is that this function is per VF setting and the previous function is a global setting.

6.3.3 VF rate limit

Run a testpmd runtime command on the PF to all queues' rate limit for a specific VF:

```
set port 0 vf 0 rate 10 queue_mask 1
```

This is a function to set the rate limit for all the queues in the `queue_mask` bitmap. It is not used to set the summary of the rate limit. The rate limit of every queue will be set equally to the assigned rate limit.

6.3.4 VF RX enabling

Run a testpmd runtime command on the PF to enable/disable packet receiving for a specific VF:

```
set port 0 vf 0 rx on|off
```

This function can be used to stop/start packet receiving on a VF.

6.3.5 VF TX enabling

Run a testpmd runtime command on the PF to enable/disable packet transmitting for a specific VF:

```
set port 0 vf 0 tx on|off
```

This function can be used to stop/start packet transmitting on a VF.

6.3.6 VF RX mode setting

Run a testpmd runtime command on the PF to set the RX mode for a specific VF:

```
set port 0 vf 0 rxmode AUPE|ROPE|BAM|MPE on|off
```

This function can be used to enable/disable some RX modes on the VF, including:

- If it accept untagged packets.
- If it accepts packets matching the MAC filters.
- If it accept MAC broadcast packets,
- If it enables MAC multicast promiscuous mode.

6.4 The I40E specific VFd functions

The functions in this section are specific to the i40e driver.

6.4.1 VF statistics

This provides an API to get the a specific VF's statistic from PF.

6.4.2 VF statistics resetting

This provides an API to rest the a specific VF's statistic from PF.

6.4.3 VF link status change notification

This provide an API to let a specific VF know if the physical link status changed.

Normally if a VF received this notification, the driver should notify the application to reset the VF port.

6.4.4 VF MAC broadcast setting

Run a testpmd runtime command on the PF to enable/disable MAC broadcast packet receiving for a specific VF:

```
set vf broadcast 0 0 on|off
```

6.4.5 VF MAC multicast promiscuous mode

Run a testpmd runtime command on the PF to enable/disable MAC multicast promiscuous mode for a specific VF:

```
set vf allmulti 0 0 on|off
```

6.4.6 VF MAC unicast promiscuous mode

Run a testpmd runtime command on the PF to enable/disable MAC unicast promiscuous mode for a specific VF:

```
set vf promisc 0 0 on|off
```

6.4.7 VF max bandwidth

Run a testpmd runtime command on the PF to set the TX maximum bandwidth for a specific VF:

```
set vf tx max-bandwidth 0 0 2000
```

The maximum bandwidth is an absolute value in Mbps.

6.4.8 VF TC bandwidth allocation

Run a testpmd runtime command on the PF to set the TCs (traffic class) TX bandwidth allocation for a specific VF:

```
set vf tc tx min-bandwidth 0 0 (20,20,20,40)
```

The allocated bandwidth should be set for all the TCs. The allocated bandwidth is a relative value as a percentage. The sum of all the bandwidth should be 100.

6.4.9 VF TC max bandwidth

Run a testpmd runtime command on the PF to set the TCs TX maximum bandwidth for a specific VF:

```
set vf tc tx max-bandwidth 0 0 0 10000
```

The maximum bandwidth is an absolute value in Mbps.

6.4.10 TC strict priority scheduling

Run a testpmd runtime command on the PF to enable/disable several TCs TX strict priority scheduling:

```
set tx strict-link-priority 0 0x3
```

The 0 in the TC bitmap means disabling the strict priority scheduling for this TC. To enable use a value of 1.

CHAPTER

SEVEN

VIRTIO_USER FOR CONTAINER NETWORKING

Container becomes more and more popular for strengths, like low overhead, fast boot-up time, and easy to deploy, etc. How to use DPDK to accelerate container networking becomes a common question for users. There are two use models of running DPDK inside containers, as shown in Fig. 7.1.

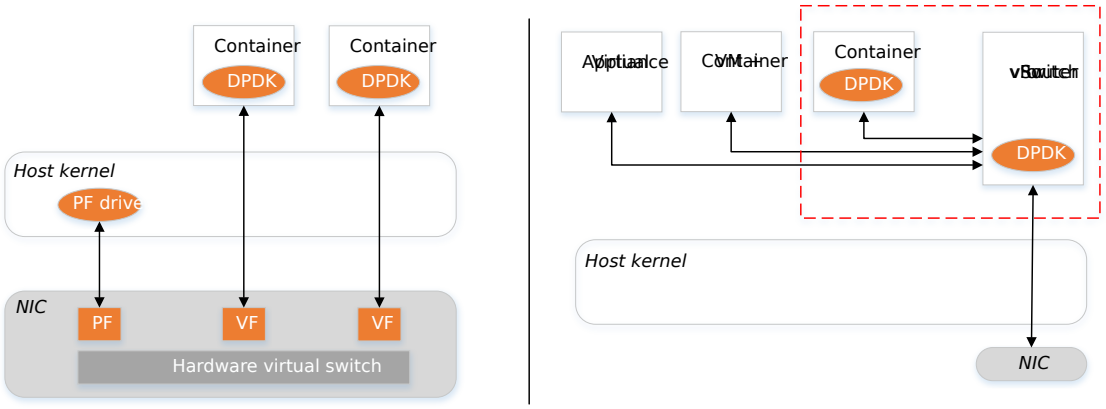


Fig. 7.1: Use models of running DPDK inside container

This page will only cover aggregation model.

7.1 Overview

The virtual device, virtio-user, with unmodified vhost-user backend, is designed for high performance user space container networking or inter-process communication (IPC).

The overview of accelerating container networking by virtio-user is shown in Fig. 7.2.

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 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 ether layer. And to minimize the change, we reuse already-existing virtio PMD code (driver/net/virtio/).

Virtio, in essence, is a shm-based solution to transmit/receive packets. How is memory shared? In VM's case, qemu always shares the whole physical layout of VM to vhost backend. But it's not feasible for a container, as a process, to share all virtual memory regions to backend. So only those virtual memory

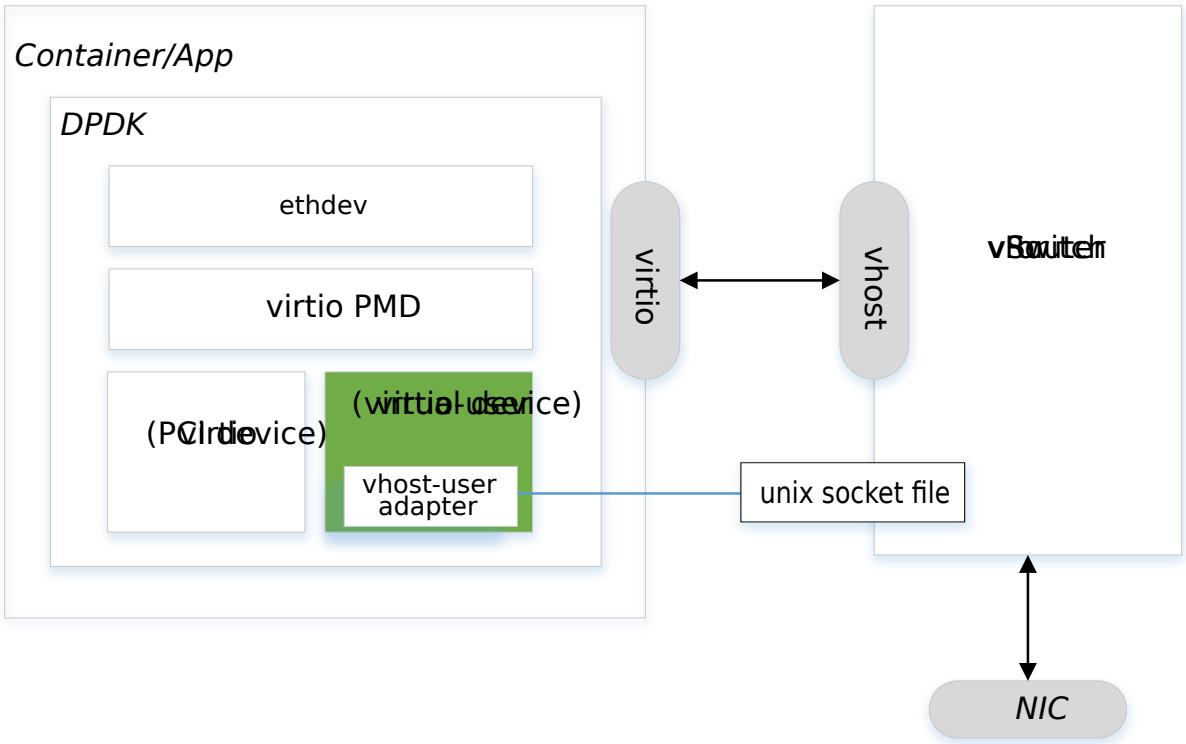


Fig. 7.2: Overview of accelerating container networking by virtio-user

regions (aka, hugepages initialized in DPDK) are sent to backend. It restricts that only addresses in these areas can be used to transmit or receive packets.

7.2 Sample Usage

Here we use Docker as container engine. It also applies to LXC, Rocket with some minor changes.

1. Compile DPDK.

```
make install RTE_SDK=`pwd` T=x86_64-native-linux-gcc
```

2. Write a Dockerfile like below.

```
cat <<EOT >> Dockerfile
FROM ubuntu:latest
WORKDIR /usr/src/dpdk
COPY . /usr/src/dpdk
ENV PATH "$PATH:/usr/src/dpdk/x86_64-native-linux-gcc/app/"
EOT
```

3. Build a Docker image.

```
docker build -t dpdk-app-testpmd .
```

4. 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
```

5. 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
```

Note: If we run all above setup on the host, it's a shm-based IPC.

7.3 Limitations

We have below limitations in this solution:

- Cannot work with –huge-unlink option. As we need to reopen the hugepage file to share with vhost backend.
- Cannot work with –no-huge option. Currently, DPDK uses anonymous mapping under this option which cannot be reopened to share with vhost backend.
- Cannot work when there are more than VHOST_MEMORY_MAX_NREGIONS(8) hugepages. If you have more regions (especially when 2MB hugepages are used), the option, –single-file-segments, can help to reduce the number of shared files.
- Applications should not use file name like HUGEFILE_FMT ("%smap_%d"). That will bring confusion when sharing hugepage files with backend by name.
- Root privilege is a must. DPDK resolves physical addresses of hugepages which seems not necessary, and some discussions are going on to remove this restriction.

CHAPTER

EIGHT

VIRTIO_USER AS EXCEPTIONAL PATH

The virtual device, virtio-user, was originally introduced with vhost-user backend, as a high performance solution for IPC (Inter-Process Communication) and user space container networking.

Virtio_user with vhost-kernel backend is a solution for exceptional path, such as KNI which exchanges packets with kernel networking stack. This solution is very promising in:

- Maintenance

  All kernel modules needed by this solution, vhost and vhost-net (kernel), are upstreamed and extensively used kernel module.

- Features

  vhost-net is born to be a networking solution, which has lots of networking related features, like multi queue, tso, multi-seg mbuf, etc.

- Performance

  similar to KNI, this solution would use one or more kthreads to send/receive packets to/from user space DPDK applications, which has little impact on user space polling thread (except that it might enter into kernel space to wake up those kthreads if necessary).

The overview of an application using virtio-user as exceptional path is shown in Fig. 8.1.

8.1 Sample Usage

As a prerequisite, the vhost/vhost-net kernel CONFIG should be chosen before compiling the kernel and those kernel modules should be inserted.

1. Compile DPDK and bind a physical NIC to igb_uio/uio_pci_generic/vfio-pci.

   This physical NIC is for communicating with outside.

2. Run testpmd.

   ```
   $(testpmd) -l 2-3 -n 4 \
           --vdev=virtio_user0,path=/dev/vhost-net,queue_size=1024 \
           -- -i --tx-offloads=0x0000002c --enable-lro \
           --txd=1024 --rxd=1024
   ```

   This command runs testpmd with two ports, one physical NIC to communicate with outside, and one virtio-user to communicate with kernel.

- `--enable-lro`

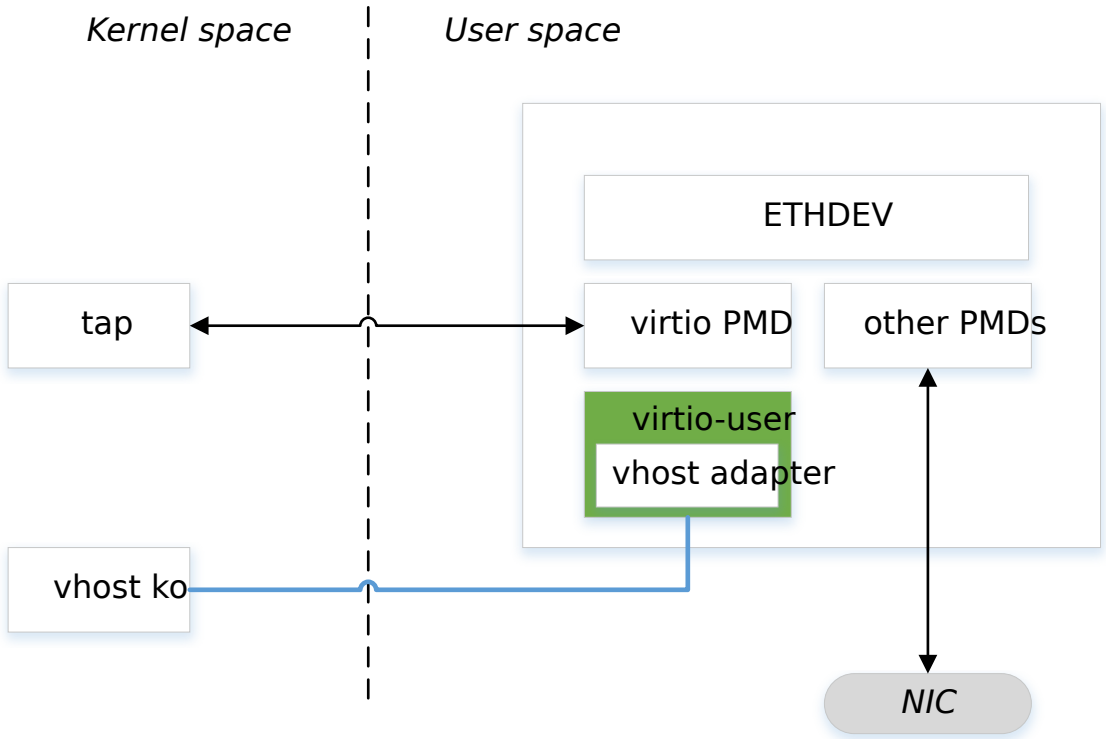


Fig. 8.1: Overview of a DPDK app using virtio-user as exceptional path

This is used to negotiate VIRTIO_NET_F_GUEST_TSO4 and VIRTIO_NET_F_GUEST_TSO6 feature so that large packets from kernel can be transmitted to DPDK application and further TSOed by physical NIC.

- `queue_size`

  256 by default. To avoid shortage of descriptors, we can increase it to 1024.

- `queues`

  Number of multi-queues. Each queue will be served by a kthread. For example:

```
$(testpmd) -l 2-3 -n 4 \
        --vdev=virtio_user0,path=/dev/vhost-net,queues=2,queue_size=1024 \
        -- -i --tx-offloads=0x0000002c --enable-lro \
        --txq=2 --rxq=2 --txd=1024 --rxd=1024
```

1. Enable Rx checksum offloads in testpmd:

```
(testpmd) port stop 0
(testpmd) port config 0 rx_offload tcp_cksum on
(testpmd) port config 0 rx_offload udp_cksum on
(testpmd) port start 0
```

2. Start testpmd:

```
(testpmd) start
```

3. Configure IP address and start tap:

```
ifconfig tap0 1.1.1.1/24 up
```

Note: The tap device will be named tap0, tap1, etc, by kernel.

Then, all traffic from physical NIC can be forwarded into kernel stack, and all traffic on the tap0 can be sent out from physical NIC.

8.2 Limitations

This solution is only available on Linux systems.

CHAPTER

NINE

DPDK PDUMP LIBRARY AND PDUMP TOOL

This document describes how the Data Plane Development Kit (DPDK) Packet Capture Framework is used for capturing packets on DPDK ports. It is intended for users of DPDK who want to know more about the Packet Capture feature and for those who want to monitor traffic on DPDK-controlled devices.

The DPDK packet capture framework was introduced in DPDK v16.07. The DPDK packet capture framework consists of the DPDK pdump library and DPDK pdump tool.

9.1 Introduction

The librte_pdump library provides the APIs required to allow users to initialize the packet capture framework and to enable or disable packet capture. The library works on a client/server model and its usage is recommended for debugging purposes.

The dpdk-pdump tool is developed based on the `librte_pdump` library. It runs as a DPDK secondary process and is capable of enabling or disabling packet capture on DPDK ports. The `dpdk-pdump` tool provides command-line options with which users can request enabling or disabling of the packet capture on DPDK ports.

The application which initializes the packet capture framework will act as a server and the application that enables or disables the packet capture will act as a client. The server sends the Rx and Tx packets from the DPDK ports to the client.

In DPDK the `testpmd` application can be used to initialize the packet capture framework and act as a server, and the `dpdk-pdump` tool acts as a client. To view Rx or Tx packets of `testpmd`, the application should be launched first, and then the `dpdk-pdump` tool. Packets from `testpmd` will be sent to the tool, which then sends them on to the Pcap PMD device and that device writes them to the Pcap file or to an external interface depending on the command-line option used.

Some things to note:

- The `dpdk-pdump` tool can only be used in conjunction with a primary application which has the packet capture framework initialized already. In dpdk, only `testpmd` is modified to initialize packet capture framework, other applications remain untouched. So, if the `dpdk-pdump` tool has to be used with any application other than the testpmd, the user needs to explicitly modify that application to call the packet capture framework initialization code. Refer to the `app/test-pmd/testpmd.c` code and look for `pdump` keyword to see how this is done.
- The `dpdk-pdump` tool depends on the libpcap based PMD which is disabled by default in the build configuration files, owing to an external dependency on the libpcap development files. Once the libpcap development files are installed, the libpcap based PMD can be enabled by setting `CONFIG_RTE_LIBRTE_PMD_PCAP=y` and recompiling the DPDK.

9.2 Test Environment

The overview of using the Packet Capture Framework and the `dpdk-pdump` tool for packet capturing on the DPDK port in Fig. 9.1.

9.3 Configuration

Modify the DPDK primary application to initialize the packet capture framework as mentioned in the above notes and enable the following config options and build DPDK:

```
CONFIG_RTE_LIBRTE_PMD_PCAP=y
CONFIG_RTE_LIBRTE_PDUMP=y
```

9.4 Running the Application

The following steps demonstrate how to run the `dpdk-pdump` tool to capture Rx side packets on dpdk_port0 in Fig. 9.1 and inspect them using `tcpdump`.

1. Launch testpmd as the primary application:

   ```
   sudo ./app/testpmd -c 0xf0 -n 4 -- -i --port-topology=chained
   ```

2. Launch the pdump tool as follows:

   ```
   sudo ./build/app/dpdk-pdump -- \
        --pdump 'port=0,queue=*,rx-dev=/tmp/capture.pcap'
   ```

3. Send traffic to dpdk_port0 from traffic generator. Inspect packets captured in the file capture.pcap using a tool that can interpret Pcap files, for example tcpdump:

   ```
   $tcpdump -nr /tmp/capture.pcap
   reading from file /tmp/capture.pcap, link-type EN10MB (Ethernet)
   11:11:36.891404 IP 4.4.4.4.whois++ > 3.3.3.3.whois++: UDP, length 18
   11:11:36.891442 IP 4.4.4.4.whois++ > 3.3.3.3.whois++: UDP, length 18
   11:11:36.891445 IP 4.4.4.4.whois++ > 3.3.3.3.whois++: UDP, length 18
   ```

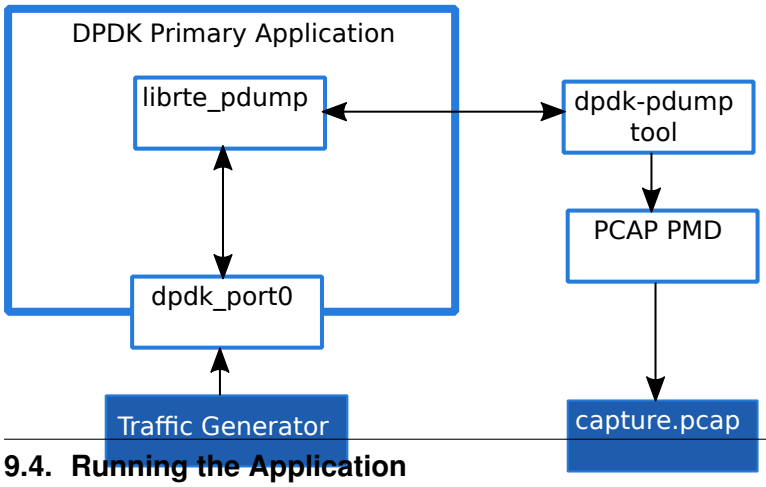


Fig. 9.1: Packet capturing on a DPDK port using the dpdk-pdump tool.

CHAPTER

TEN

DPDK TELEMETRY API USER GUIDE

This document describes how the Data Plane Development Kit(DPDK) Telemetry API is used for querying port statistics from incoming traffic.

10.1 Introduction

The `librte_telemetry` provides the functionality so that users may query metrics from incoming port traffic and global stats(application stats). The application which initializes packet forwarding will act as the server, sending metrics to the requesting application which acts as the client.

In DPDK, applications are used to initialize the `telemetry`. To view incoming traffic on featured ports, the application should be run first (ie. after ports are configured). Once the application is running, the service assurance agent (for example the collectd plugin) should be run to begin querying the API.

A client connects their Service Assurance application to the DPDK application via a UNIX socket. Once a connection is established, a client can send JSON messages to the DPDK application requesting metrics via another UNIX client. This request is then handled and parsed if valid. The response is then formatted in JSON and sent back to the requesting client.

10.1.1 Pre-requisites

- Python >= 2.5
- Jansson library for JSON serialization

10.2 Test Environment

`telemetry` offers a range of selftests that a client can run within the DPDK application.

Selftests are disabled by default. They can be enabled by setting the 'selftest' variable to 1 in rte_telemetry_initial_accept().

Note: this 'hardcoded' value is temporary.

10.3 Configuration

Enable the telemetry API by modifying the following config option before building DPDK:

```
CONFIG_RTE_LIBRTE_TELEMETRY=y
```

Note: Meson will pick this up automatically if `libjansson` is available.

10.4 Running the Application

The following steps demonstrate how to run the `telemetry` API to query all statistics on all active ports, using the `telemetry_client` python script to query. Note: This guide assumes packet generation is applicable and the user is testing with testpmd as a DPDK primary application to forward packets, although any DPDK application is applicable.

1. Launch testpmd as the primary application with `telemetry`.:

   ```
   ./app/testpmd --telemetry
   ```

2. Launch the `telemetry` python script with a client filepath.:

   ```
   python usertools/telemetry_client.py /var/run/some_client
   ```

   The client filepath is going to be used to setup our UNIX connection with the DPDK primary application, in this case `testpmd` This will initialize a menu where a client can proceed to recursively query statistics, request statistics once or unregister the file_path, thus exiting the menu.

3. Send traffic to any or all available ports from a traffic generator. Select a query option(recursive or singular polling or global stats). The metrics will then be displayed on the client terminal in JSON format.

4. Once finished, unregister the client using the menu command.

CHAPTER

ELEVEN

DEBUG & TROUBLESHOOT GUIDE

DPDK applications can be designed to have simple or complex pipeline processing stages making use of single or multiple threads. Applications can use poll mode hardware devices which helps in offloading CPU cycles too. It is common to find solutions designed with

- single or multiple primary processes
- single primary and single secondary
- single primary and multiple secondaries

In all the above cases, it is tedious to isolate, debug, and understand various behaviors which occur randomly or periodically. The goal of the guide is to consolidate a few commonly seen issues for reference. Then, isolate to identify the root cause through step by step debug at various stages.

Note: It is difficult to cover all possible issues; in a single attempt. With feedback and suggestions from the community, more cases can be covered.

11.1 Application Overview

By making use of the application model as a reference, we can discuss multiple causes of issues in the guide. Let us assume the sample makes use of a single primary process, with various processing stages running on multiple cores. The application may also make uses of Poll Mode Driver, and libraries like service cores, mempool, mbuf, eventdev, cryptodev, QoS, and ethdev.

The overview of an application modeled using PMD is shown in Fig. 11.1.

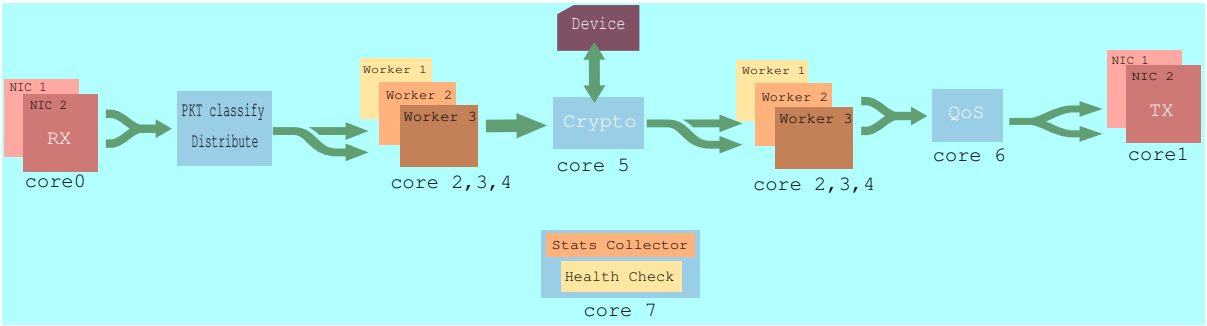


Fig. 11.1: Overview of pipeline stage of an application

11.2 Bottleneck Analysis

A couple of factors that lead the design decision could be the platform, scale factor, and target. This distinct preference leads to multiple combinations, that are built using PMD and libraries of DPDK. While the compiler, library mode, and optimization flags are the components are to be constant, that affects the application too.

11.2.1 Is there mismatch in packet (received < desired) rate?

RX Port and associated core Fig. 11.2.

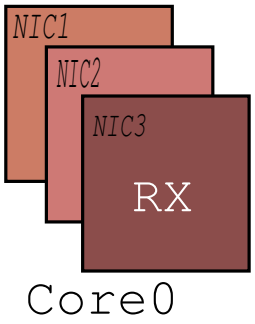


Fig. 11.2: RX packet rate compared against received rate.

1. Is the configuration for the RX setup correctly?
   - Identify if port Speed and Duplex is matching to desired values with `rte_eth_link_get`.
   - Check `DEV_RX_OFFLOAD_JUMBO_FRAME` is set with `rte_eth_dev_info_get`.
   - Check promiscuous mode if the drops do not occur for unique MAC address with `rte_eth_promiscuous_get`.
2. Is the drop isolated to certain NIC only?
   - Make use of `rte_eth_dev_stats` to identify the drops cause.
   - If there are mbuf drops, check nb_desc for RX descriptor as it might not be sufficient for the application.
   - If `rte_eth_dev_stats` shows drops are on specific RX queues, ensure RX lcore threads has enough cycles for `rte_eth_rx_burst` on the port queue pair.
   - If there are redirect to a specific port queue pair with, ensure RX lcore threads gets enough cycles.
   - Check the RSS configuration `rte_eth_dev_rss_hash_conf_get` if the spread is not even and causing drops.
   - If PMD stats are not updating, then there might be offload or configuration which is dropping the incoming traffic.
3. Is there drops still seen?
   - If there are multiple port queue pair, it might be the RX thread, RX distributor, or event RX adapter not having enough cycles.

- If there are drops seen for RX adapter or RX distributor, try using `rte_prefetch_non_temporal` which intimates the core that the mbuf in the cache is temporary.

11.2.2 Is there packet drops at receive or transmit?

RX-TX port and associated cores Fig. 11.3.

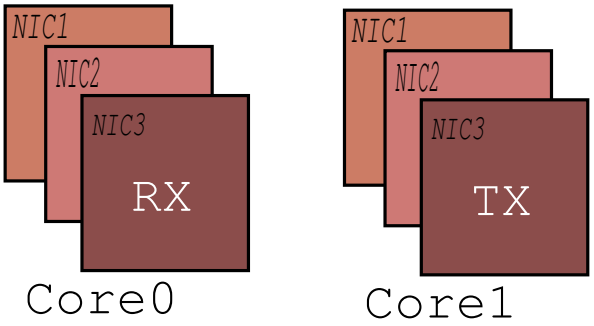


Fig. 11.3: RX-TX drops

1. At RX
   - Identify if there are multiple RX queue configured for port by `nb_rx_queues` using `rte_eth_dev_info_get`.
   - Using `rte_eth_dev_stats` fetch drops in q_errors, check if RX thread is configured to fetch packets from the port queue pair.
   - Using `rte_eth_dev_stats` shows drops in `rx_nombuf`, check if RX thread has enough cycles to consume the packets from the queue.
2. At TX
   - If the TX rate is falling behind the application fill rate, identify if there are enough descriptors with `rte_eth_dev_info_get` for TX.
   - Check the `nb_pkt` in `rte_eth_tx_burst` is done for multiple packets.
   - Check `rte_eth_tx_burst` invokes the vector function call for the PMD.
   - If oerrors are getting incremented, TX packet validations are failing. Check if there queue specific offload failures.
   - If the drops occur for large size packets, check MTU and multi-segment support configured for NIC.

11.2.3 Is there object drops in producer point for the ring library?

Producer point for ring Fig. 11.4.

1. Performance issue isolation at producer
   - Use `rte_ring_dump` to validate for all single producer flag is set to `RING_F_SP_ENQ`.
   - There should be sufficient `rte_ring_free_count` at any point in time.

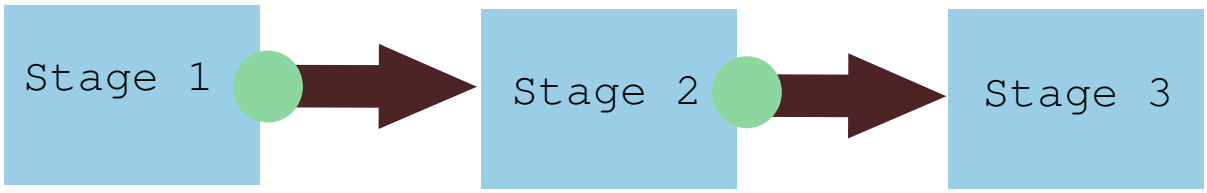


Fig. 11.4: Producer point for Rings

- Extreme stalls in dequeue stage of the pipeline will cause `rte_ring_full` to be true.

11.2.4 Is there object drops in consumer point for the ring library?

Consumer point for ring Fig. 11.5.

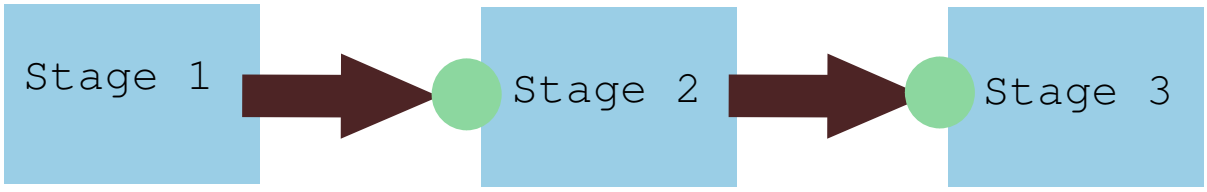


Fig. 11.5: Consumer point for Rings

1. Performance issue isolation at consumer
   - Use `rte_ring_dump` to validate for all single consumer flag is set to `RING_F_SC_DEQ`.
   - If the desired burst dequeue falls behind the actual dequeue, the enqueue stage is not filling up the ring as required.
   - Extreme stall in the enqueue will lead to `rte_ring_empty` to be true.

11.2.5 Is there a variance in packet or object processing rate in the pipeline?

Memory objects close to NUMA Fig. 11.6.

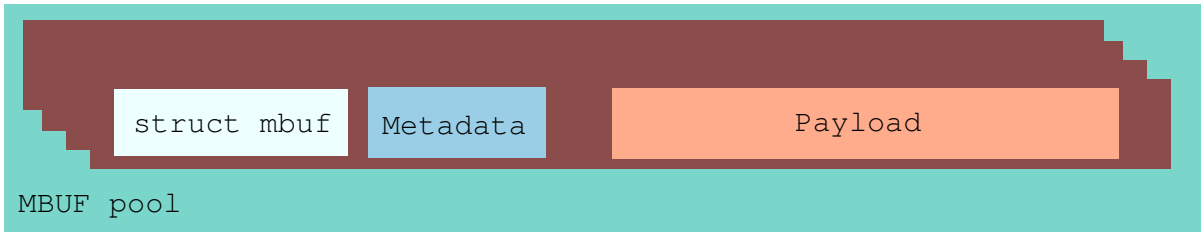


Fig. 11.6: Memory objects have to be close to the device per NUMA.

1. Stall in processing pipeline can be attributes of MBUF release delays. These can be narrowed down to
   - Heavy processing cycles at single or multiple processing stages.
   - Cache is spread due to the increased stages in the pipeline.
   - CPU thread responsible for TX is not able to keep up with the burst of traffic.

- Extra cycles to linearize multi-segment buffer and software offload like checksum, TSO, and VLAN strip.
- Packet buffer copy in fast path also results in stalls in MBUF release if not done selectively.
- Application logic sets `rte_pktmbuf_refcnt_set` to higher than the desired value and frequently uses `rte_pktmbuf_prefree_seg` and does not release MBUF back to mempool.

2. Lower performance between the pipeline processing stages can be
   - The NUMA instance for packets or objects from NIC, mempool, and ring should be the same.
   - Drops on a specific socket are due to insufficient objects in the pool. Use `rte_mempool_get_count` or `rte_mempool_avail_count` to monitor when drops occurs.
   - Try prefetching the content in processing pipeline logic to minimize the stalls.
3. Performance issue can be due to special cases
   - Check if MBUF continuous with `rte_pktmbuf_is_contiguous` as certain offload requires the same.
   - Use `rte_mempool_cache_create` for user threads require access to mempool objects.
   - If the variance is absent for larger huge pages, then try rte_mem_lock_page on the objects, packets, lookup tables to isolate the issue.

11.2.6 Is there a variance in cryptodev performance?

Crypto device and PMD Fig. 11.7.

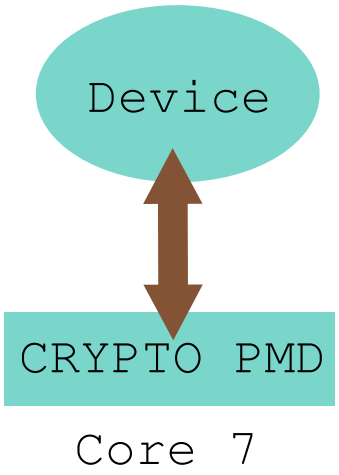


Fig. 11.7: CRYPTO and interaction with PMD device.

1. Performance issue isolation for enqueue
   - Ensure cryptodev, resources and enqueue is running on NUMA cores.
   - Isolate if the cause of errors for err_count using `rte_cryptodev_stats`.

- Parallelize enqueue thread for varied multiple queue pair.

2. Performance issue isolation for dequeue
   - Ensure cryptodev, resources and dequeue are running on NUMA cores.
   - Isolate if the cause of errors for err_count using `rte_cryptodev_stats`.
   - Parallelize dequeue thread for varied multiple queue pair.
3. Performance issue isolation for crypto operation
   - If the cryptodev software-assist is in use, ensure the library is built with right (SIMD) flags or check if the queue pair using CPU ISA for feature_flags AVX|SSE|NEON using `rte_cryptodev_info_get`.
   - If the cryptodev hardware-assist is in use, ensure both firmware and drivers are up to date.
4. Configuration issue isolation
   - Identify cryptodev instances with `rte_cryptodev_count` and `rte_cryptodev_info_get`.

11.2.7 Is user functions performance is not as expected?

Custom worker function Fig. 11.8.

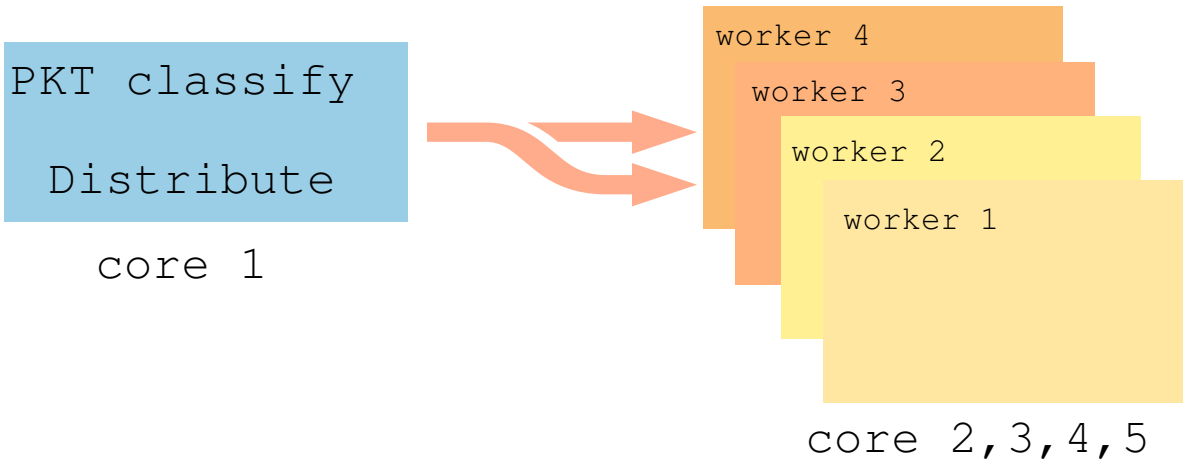


Fig. 11.8: Custom worker function performance drops.

1. Performance issue isolation
   - The functions running on CPU cores without context switches are the performing scenarios. Identify lcore with `rte_lcore` and lcore index mapping with CPU using `rte_lcore_index`.
   - Use `rte_thread_get_affinity` to isolate functions running on the same CPU core.
2. Configuration issue isolation
   - Identify core role using `rte_eal_lcore_role` to identify RTE, OFF and SERVICE. Check performance functions are mapped to run on the cores.
   - For high-performance execution logic ensure running it on correct NUMA and non-master core.

- Analyze run logic with `rte_dump_stack`, `rte_dump_registers` and `rte_memdump` for more insights.
- Make use of objdump to ensure opcode is matching to the desired state.

11.2.8 Is the execution cycles for dynamic service functions are not frequent?

service functions on service cores Fig. 11.9.

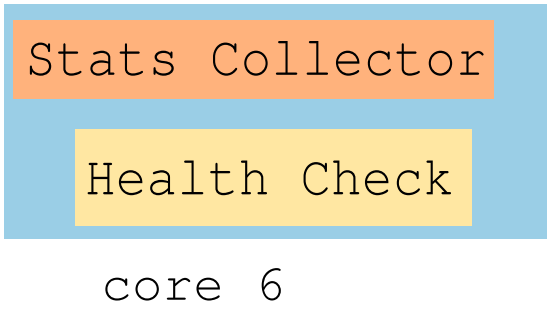


Fig. 11.9: functions running on service cores

1. Performance issue isolation
   - Services configured for parallel execution should have `rte_service_lcore_count` should be equal to `rte_service_lcore_count_services`.
   - A service to run parallel on all cores should return `RTE_SERVICE_CAP_MT_SAFE` for `rte_service_probe_capability` and `rte_service_map_lcore_get` returns unique lcore.
   - If service function execution cycles for dynamic service functions are not frequent?
   - If services share the lcore, overall execution should fit budget.
2. Configuration issue isolation
   - Check if service is running with `rte_service_runstate_get`.
   - Generic debug via `rte_service_dump`.

11.2.9 Is there a bottleneck in the performance of eventdev?

1. Check for generic configuration
   - Ensure the event devices created are right NUMA using `rte_event_dev_count` and `rte_event_dev_socket_id`.
   - Check for event stages if the events are looped back into the same queue.
   - If the failure is on the enqueue stage for events, check if queue depth with `rte_event_dev_info_get`.
2. If there are performance drops in the enqueue stage
   - Use `rte_event_dev_dump` to dump the eventdev information.
   - Periodically checks stats for queue and port to identify the starvation.

- Check the in-flight events for the desired queue for enqueue and dequeue.

11.2.10 Is there a variance in traffic manager?

Traffic Manager on TX interface Fig. 11.10.

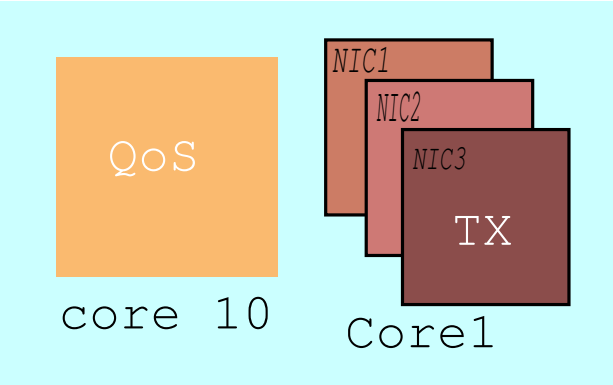


Fig. 11.10: Traffic Manager just before TX.

1. Identify the cause for a variance from expected behavior, is due to insufficient CPU cycles. Use `rte_tm_capabilities_get` to fetch features for hierarchies, WRED and priority schedulers to be offloaded hardware.
2. Undesired flow drops can be narrowed down to WRED, priority, and rates limiters.
3. Isolate the flow in which the undesired drops occur. Use `rte_tn_get_number_of_leaf_node` and flow table to ping down the leaf where drops occur.
4. Check the stats using `rte_tm_stats_update` and `rte_tm_node_stats_read` for drops for hierarchy, schedulers and WRED configurations.

11.2.11 Is the packet in the unexpected format?

Packet capture before and after processing Fig. 11.11.

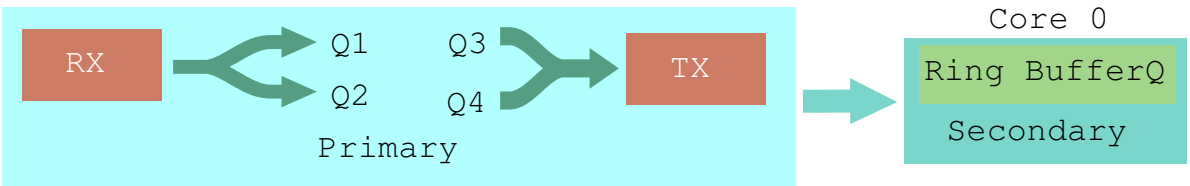


Fig. 11.11: Capture points of Traffic at RX-TX.

1. To isolate the possible packet corruption in the processing pipeline, carefully staged capture packets are to be implemented.
   - First, isolate at NIC entry and exit.

     Use pdump in primary to allow secondary to access port-queue pair. The packets get copied over in RX|TX callback by the secondary process using ring buffers.

- Second, isolate at pipeline entry and exit.

  Using hooks or callbacks capture the packet middle of the pipeline stage to copy the packets, which can be shared to the secondary debug process via user-defined custom rings.

Note: Use similar analysis to objects and metadata corruption.

11.2.12 Does the issue still persist?

The issue can be further narrowed down to the following causes.

1. If there are vendor or application specific metadata, check for errors due to META data error flags. Dumping private meta-data in the objects can give insight into details for debugging.
2. If there are multi-process for either data or configuration, check for possible errors in the secondary process where the configuration fails and possible data corruption in the data plane.
3. Random drops in the RX or TX when opening other application is an indication of the effect of a noisy neighbor. Try using the cache allocation technique to minimize the effect between applications.

11.3 How to develop a custom code to debug?

1. For an application that runs as the primary process only, debug functionality is added in the same process. These can be invoked by timer call-back, service core and signal handler.
2. For the application that runs as multiple processes. debug functionality in a standalone secondary process.