Chapter 6 Contrail DPDK vRouter Toolbox and case study

# DPDK vRouter Tool Box

You’ve read a lot of details about DPDK and Contrail DPDK vRouter implementations. You should understand that performance boost is its main benefit but it has pros and cons. One problem commonly raised is the lack of tools during troubleshooting process, especially in the case of traffic loss problems. Within the traditional Linux world, there are tons of well-known tools to trace the packet, from displaying packet statistics in and out of NIC, showing drop counters, to performing packet capture for deeper level packet decoding. Examples of these tools are ifconfig, ip, bmon, tcpdump, tshark, etc. With DPDK, however, none of the traditional tools can be used directly, and the reason is obvious: whichever interface bound to DPDK becomes invisible to the Linux stack, hence they are also hidden from the perspective of these tools. In production, you need some new tools developed to fill this gap, so that you can narrow the packet loss related issues when the outage is ongoing. Fortunately, today’s Contrail DPDK vRouter is equipped with quite a few such tools. In this section we’ll look at some of them.

## Contrail-tools Docker: vRouter tools box

Contrail-tools is a Docker container located in the compute node, where all of the vRouter tools and utilities are available. Apparently, from the user perspective, this is more convenient than distributing tools in multiple containers. This design was introduced a few releases before Contrail networking R2008. As more and more existing tools migrated into it and new tools added in, this container now really becomes a centralized tool box, whenever you want to check any running states of the vRouter data plane. Let’s first take a look at how to open this box.

To enter the container, just run the contrail-tools script (same name as of the Docker) in a compute node:

[root@a7s3 ~]# contrail-tools  
Unable to find image 'svl-artifactory.juniper.net/contrail-nightly/contrail-tools:2008.108' locally  
2008.108: Pulling from contrail-nightly/contrail-tools  
f34b00c7da20: Already exists  
b3779b5a313a: Already exists  
4b95f42cde64: Already exists  
8b329f8ee1e6: Already exists  
2986115b3d27: Already exists  
10c5940c4895: Already exists  
dec794e181cd: Already exists  
226c056c5788: Already exists  
d391962e0038: Pull complete  
Digest: sha256:2d68d8cd010ba76c265c3b7458fcf12c459d46ec71357b45118dfc4610f40338  
Status: Downloaded newer image for svl-artifactory.juniper.net/contrail-nightly/contrail-tools:2008.108  
(contrail-tools)[root@a7s3 /]$

Now you are inside of the container. From here you can test all of the old vRouter tools you may have been familiar with, for example, to print the packet dropping statistics:

(contrail-tools)[root@a7s3 /]$ dropstats | grep -iEv 0$|^$  
Flow Action Drop 1792  
Flow Queue Limit Exceeded 305  
Invalid NH 12  
No L2 Route 1

We use grep to remove all counters with a zero value. When you are done, just exit the Docker and it will be killed.

(contrail-tools)[root@a7s3 /]$ exit  
exit  
[root@a7s3 ~]#

You can also pass the tool command as parameters to the script, execute the command, get its output, and exit the Docker all with one go.

[root@a7s3 ~]# contrail-tools dropstats | grep -iE route  
No L2 Route 68129939  
[root@a7s3 ~]#

As the time of the writing of this book, there are nearly twenty tools available in this container. Let’s take a look at what’s in the package.

First, in the container locate the package name:

[root@a7s3 ~]# contrail-tools  
lcontrail-tools)[root@a7s3 /]$ rpm -qa | grep contrail-tool  
contrail-tools-2008-108.el7.x86\_64

Then, based on the package name, you can list all available tools in it:

(contrail-tools)[root@a7s3 /]$ repoquery -l contrail-tools-2008-108.el7.x86\_64 | grep bin  
/usr/bin/dpdkinfo  
/usr/bin/dpdkvifstats.py  
/usr/bin/dropstats  
/usr/bin/flow  
/usr/bin/mirror  
/usr/bin/mpls  
/usr/bin/nh  
/usr/bin/pkt\_droplog.py  
/usr/bin/qosmap  
/usr/bin/rt  
/usr/bin/sandump  
/usr/bin/vif  
/usr/bin/vifdump  
/usr/bin/vrfstats  
/usr/bin/vrftable  
/usr/bin/vrinfo  
/usr/bin/vrmemstats  
/usr/bin/vrouter  
/usr/bin/vxlan

In previous chapters you’ve read about the dpdk\_nic\_bind.py script, which is a tool to bind a specific driver for a NIC. In the rest of this section, we’ll introduce some more tools that are especially useful in DPDK environment.

## Vif Command and Scripts

The first one from our Contrail DPDK tool box is THE vif command. Before talking about it, let’s see how to list all interfaces in the compute running DPDK vRouter. Let’s first try the Linux ip or ifconfig command in our DPDK compute running PROX gen VM:

[root@a7s3 ~]# ip link  
1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000  
 link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00  
2: eno1: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc mq state UP mode DEFAULT group default qlen 1000  
 link/ether 0c:c4:7a:4c:16:c2 brd ff:ff:ff:ff:ff:ff  
3: eno2: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc mq state UP mode DEFAULT group default qlen 1000  
 link/ether 0c:c4:7a:4c:16:c3 brd ff:ff:ff:ff:ff:ff  
8: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT group default  
 link/ether 02:42:56:4f:cc:6e brd ff:ff:ff:ff:ff:ff  
25: vhost0: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc pfifo\_fast state UNKNOWN mode DEFAULT group default qlen 1000  
 link/ether 90:e2:ba:c3:af:20 brd ff:ff:ff:ff:ff:ff

Well, you can see some interfaces were output:

* the loopback interface (lo)
* management interface (eno1)
* vhost0 interface
* docker interface (docker0)
* physical NIC which is not in use (eno2)

However, some most important interfaces are not shown at all:

* The physical fabric interface: the bond interface in our setup
* The VM virtual interfaces: the tapxxx interfaces

If you compare this with what you’d see with the same IP command in a kernel mode vRouter compute without DPDK, there’s a big difference:

[root@a7s5-kiran ~]# ip link  
1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000  
 link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00  
2: eno1: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc mq state UP mode DEFAULT group default qlen 1000  
 link/ether 0c:c4:7a:47:d7:b4 brd ff:ff:ff:ff:ff:ff  
3: eno2: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc mq state UP mode DEFAULT group default qlen 1000  
 link/ether 0c:c4:7a:47:d7:b5 brd ff:ff:ff:ff:ff:ff  
4: enp2s0f0: <BROADCAST,MULTICAST,SLAVE,UP,LOWER\_UP> mtu 1500 qdisc mq master bond0 state UP mode DEFAULT group default qlen 1000  
 link/ether 00:1b:21:bb:f9:46 brd ff:ff:ff:ff:ff:ff  
5: enp2s0f1: <BROADCAST,MULTICAST,SLAVE,UP,LOWER\_UP> mtu 1500 qdisc mq master bond0 state UP mode DEFAULT group default qlen 1000  
 link/ether 00:1b:21:bb:f9:46 brd ff:ff:ff:ff:ff:ff  
6: bond0: <BROADCAST,MULTICAST,MASTER,UP,LOWER\_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group default qlen 1000  
 link/ether 00:1b:21:bb:f9:46 brd ff:ff:ff:ff:ff:ff  
12: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT group default  
 link/ether 02:42:d6:c6:2c:12 brd ff:ff:ff:ff:ff:ff  
41: pkt1: <UP,LOWER\_UP> mtu 65535 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000  
 link/void c2:6e:97:ef:cd:b2 brd 00:00:00:00:00:00  
42: pkt3: <UP,LOWER\_UP> mtu 65535 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000  
 link/void 8e:44:4e:2e:28:0c brd 00:00:00:00:00:00  
43: pkt2: <UP,LOWER\_UP> mtu 65535 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000  
 link/void a6:2a:01:7c:db:65 brd 00:00:00:00:00:00  
44: vhost0: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc pfifo\_fast state UNKNOWN mode DEFAULT group default qlen 1000  
 link/ether 00:1b:21:bb:f9:46 brd ff:ff:ff:ff:ff:ff  
45: bond0.101@bond0: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group default qlen 1000  
 link/ether 00:1b:21:bb:f9:46 brd ff:ff:ff:ff:ff:ff  
46: pkt0: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc pfifo\_fast state UNKNOWN mode DEFAULT group default qlen 1000  
 link/ether 5e:a0:f8:77:25:97 brd ff:ff:ff:ff:ff:ff  
49: tap0160123b-14: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc mq state UP mode DEFAULT group default qlen 1000  
 link/ether fe:01:60:12:3b:14 brd ff:ff:ff:ff:ff:ff

Here except for lo, management interface, and whatever we saw from the DPDK compute, you can also see these other important interfaces:

* bond interface and its subinterface: bond0, bond0.101
* bond interface’s member interfaces: enp2s0f0, enp2s0f1
* VM tap interface: tap0160123b-14
* pkt0 interface

NOTE The pkt1, pkt2, pkt3 interfaces are created by vRouter but not used in DPDK setup.

The reason you can see these differences, as we’ve mentioned many times throughout this book, is that when DPDK is in charge of the NIC card the Linux kernel is mostly bypassed. The NIC card’s feature and functions are exposed by another special driver directly connected to the user space PMD driver running in the DPDK layer, so the traditional applications that rely on the interfaces sitting in the Linux kernel to do their job, are no longer useful.

We’ll talk more about this later. For now, let’s look at the vif command with -l|--list and --get option. The vif --list lists all interfaces located in the vRouter and --get just retrieves one of them. Here is the capture from the same DPDK compute:

[root@a7s3 ~]# contrail-tools vif --get 3  
Vrouter Interface Table  
  
......  
vif0/3 PMD: tap41a9ab05-64 NH: 32  
 Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.1.104  
 Vrf:3 Mcast Vrf:3 Flags:PL3L2DMonEr QOS:-1 Ref:12  
 RX queue packets:2306654691 errors:0  
 RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0  
 RX packets:2306869103 bytes:285898139558 errors:0  
 TX packets:47613036 bytes:5739655392 errors:0  
 ISID: 0 Bmac: 02:41:a9:ab:05:64  
  
[root@a7s3 ~]# contrail-tools vif -l  
Vrouter Interface Table  
......  
  
vif0/0 PCI: 0000:00:00.0 (Speed 20000, Duplex 1) NH: 4  
 Type:Physical HWaddr:90:e2:ba:c3:af:20 IPaddr:0.0.0.0  
 Vrf:0 Mcast Vrf:65535 Flags:TcL3L2VpVofEr QOS:-1 Ref:18  
 RX device packets:106218495224 bytes:12108991404264 errors:0  
 RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0  
 Fabric Interface: eth\_bond\_bond0 Status: UP Driver: net\_bonding  
 Slave Interface(0): 0000:02:00.0 Status: UP Driver: net\_ixgbe  
 Slave Interface(1): 0000:02:00.1 Status: UP Driver: net\_ixgbe  
 Vlan Id: 101 VLAN fwd Interface: vfw  
 RX packets:53109240518 bytes:5842056828972 errors:0  
 TX packets:53459418469 bytes:5880886194306 errors:0  
 Drops:291  
 TX device packets:106919210258 bytes:12189494593618 errors:0  
  
vif0/1 PMD: vhost0 NH: 5  
 Type:Host HWaddr:90:e2:ba:c3:af:20 IPaddr:8.0.0.4  
 Vrf:0 Mcast Vrf:65535 Flags:L3DEr QOS:-1 Ref:13  
 RX device packets:436036 bytes:400358720 errors:0  
 RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0  
 RX packets:436036 bytes:400358720 errors:0  
 TX packets:447092 bytes:88525732 errors:0  
 Drops:3  
 TX device packets:447092 bytes:88518904 errors:0  
  
vif0/2 Socket: unix  
 Type:Agent HWaddr:00:00:5e:00:01:00 IPaddr:0.0.0.0  
 Vrf:65535 Mcast Vrf:65535 Flags:L3Er QOS:-1 Ref:3  
 RX port packets:71548 errors:0  
 RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0  
 RX packets:71548 bytes:6153128 errors:0  
 TX packets:14936 bytes:1359697 errors:0  
 Drops:0  
  
vif0/3 PMD: tap41a9ab05-64 NH: 38  
 Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.1.104  
 Vrf:2 Mcast Vrf:2 Flags:L3L2DEr QOS:-1 Ref:12  
 RX queue packets:17708866065 errors:3874701360  
 RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 3874691664 9696  
 RX packets:17708865121 bytes:1062531327800 errors:0  
 TX packets:17563478684 bytes:1053808124972 errors:0  
 ISID: 0 Bmac: 02:41:a9:ab:05:64  
 Drops:3874701393  
  
vif0/4 PMD: tapd2d7bb67-c1 NH: 35  
 Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.0.104  
 Vrf:3 Mcast Vrf:3 Flags:PL3L2DEr QOS:-1 Ref:12  
 RX queue packets:3060 errors:205  
 RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 205 0  
 RX packets:5478 bytes:528770 errors:0  
 TX packets:5402 bytes:423320 errors:0  
 Drops:445

Here, the vRouter interfaces are:

* vif0/0: this connects to the bond interface
* vif0/1: this connects to vhost0, the interface in linux kernel
* vif0/2: this connects to the pkt0 interface toward vrouter agent
* vif0/3: this is the vRouter interface connecting the data interface of our PROX VM: tap41a9ab05-64
* vif0/4: this is the vRouter interface connecting the control and management interface of our PROX VM: tapd2d7bb67-c1

Now you should understand the importance of the vif command, especially in the DPDK vRouter. It shows interfaces from vRouter’s perspective, and reveals the one-to-one connection mapping between vRouter and fabric or the VM tap interface. The latter would otherwise be invisible.

Besides that, it also displays key information. The Vrf numbers and packet counters are the most commonly used data points, but among various counters, usually we focus on the RX/TX packets/bytes counters which displays data received or sent in packets or bytes. Depending on your environment, sometimes you may also see non-zero numbers in the RX/TX queue packets/errors counter that gives inter-lcore packet statistics. This usually happens when two lcores are involved in the packet forwarding path. Let’s use this command intensively in the rest of this chapter and analyze them to understand some of the important vRouter working mechanisms.

The vif tool also supports some other options such as --help to display a brief list of all currently supported options:

[root@a7s3 ~]# contrail-tools vif --help  
Usage: vif [--create <intf\_name> --mac <mac>]  
 [--add <intf\_name> --mac <mac> --vrf <vrf>  
 --type [vhost|agent|physical|virtual|monitoring]  
 --transport [eth|pmd|virtual|socket]  
 --xconnect <physical interface name>  
 --policy, --vhost-phys, --dhcp-enable]  
 --vif <vif ID> --id <intf\_id> --pmd --pci]  
 [--delete <intf\_id>|<intf\_name>]  
 [--get <intf\_id>][--kernel][--core <core number>][--rate] [--get-drop-stats]  
 [--set <intf\_id> --vlan <vlan\_id> --vrf <vrf\_id>]  
 [--list][--core <core number>][--rate]  
 [--sock-dir <sock dir>]  
 [--clear][--id <intf\_id>][--core <core\_number>]  
 [--help]

We won’t cover every option and its usage and usually you don’t need to know any except --get and -l|--list. But there is one more (--add) which we’ll talk about shortly. The --clear option will reset all counters, and this is handy to set a quick clean baseline for later observations, which we’ll give an example later. For others, refer to Juniper documentation at, <https://www.juniper.net/documentation/en_US/contrail20/topics/task/configuration/vrouter-cli-utilities-vnc.html>.

Now let’s look at two useful scripts that are developed based on the vif command: dpdkvifstats.py and vifdump.

### dpdkvifstats.py script

We’ve seen that the vif command displays all interfaces and their traffic statistics (RX/TX packets/bytes/errors, RX queue packets/errors, etc.) in the form of a list. During testing or troubleshooting, you can collect this data to evaluate the vRouter’s forwarding performance, its running status, is it losing packets or not, etc. In production, you always need to examine the traffic passing through a compute. Same thing in lab, once you start traffic from the PROX or any other traffic generator, the first thing you want to check is the traffic rate on the interfaces. In fact there are at least two common tasks in practice:

* monitor the traffic forwarding rate (instead of only number of packets)
* compare statistics between different vif interfaces

Starting from R2008, a Python script named dpdkvifstat.py is provided, which collects the statistics from vif output, calculates the changing rate of all counters in pps (packet per second) and bps (bit per second), based on both per-lcore and total statistics. It then displays the result in a table format. This makes the output prettier and the comparison across vif interfaces much easier to read.

In fact the vif command also provides --list --rate options to display traffic rates. However, it is lacking of itemized per-lcore statistics and the display is not easy to be collected in a file.

To demonstrate how the script works (see Figure 5.14), in our testbed we have configured PROX to send traffic at a constant speed of 125000 bytes per second (Bps) with minimum packet size of 60 bytes. That calculates to about 1.48K packet per second (PPS).



Figure 5.14 PROX gen sending traffic with a speed of 125000 Bytes per second

Let’s take a look at the dpdkvifstats.py script output in Figure 5.15.

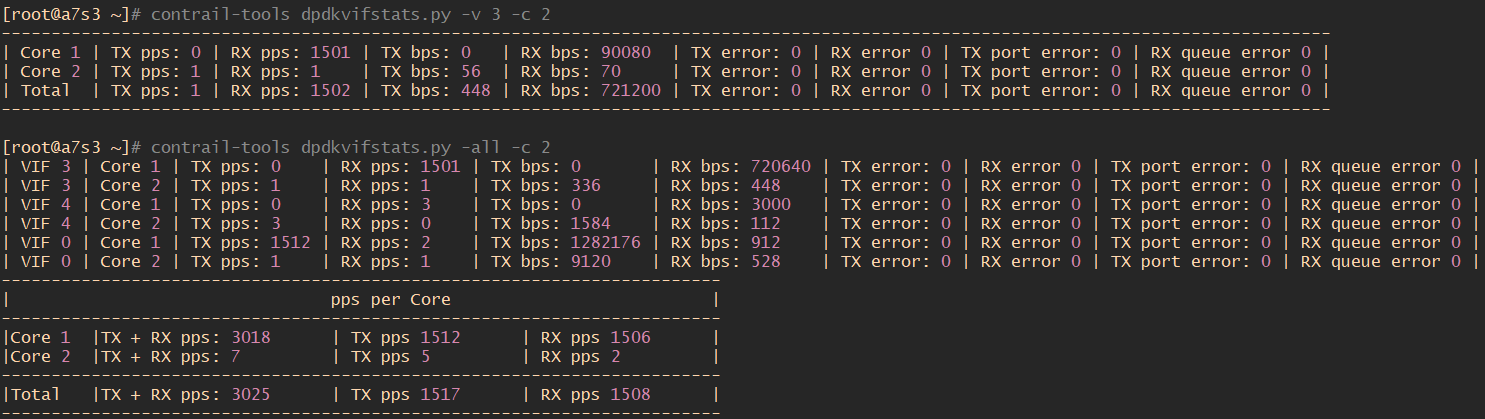


Figure 5.15 dpdkvifstats.py

We typically run the script two times. First, to show traffic rate for vif0/3 (-v), then to show traffic rate for all (-a) vif interfaces for comparison purposes. In both executions, per-lcore statistics of a specific interface are given separately. With the -v option, the total value of the interface is also given, which is in addition of counters from all cores. This gives a per-interface statistic. With -a, the script also calculates RX/TX/RX+TX traffic rate for each lcore across all interfaces in the end. This gives the overall lcore forwarding load in the DPDK vRouter.

To understand the output, first let’s review the DPDK vRouter CPU cores allocation.

In Chapter 3, you’ve learned about DPDK vRouter architectures and you know how the packet processing works. Basically:

**vRouter creates the same number of lcores and DPDK queues as the number of CPUs allocated to it.**

In this compute, for testing purposes, we’ve allocated two CPU cores to vRouter DPDK forwarding lcores. CPU allocation to DPDK vRouter forwarding lcores is configurable via vRouter configuration files. Refer to Chapter 4 for CPU allocation details. For each vRouter interface, two DPDK queues are created, each served by a forwarding lcore in DPDK process. That is why in the output for each vif interface there are two lines statistics, for Core 1 and Core 2 respectively.

This capture shows vRouter interface vif 0/3 processed 1501 pps traffic in the first forwarding lcore, that is 720640 bits per second for 60 bytes packet size. These are the majority of the traffic forwarded out of fabric interface vif 0/0, with a similar rate of 1512 pps. These overlay packets received from the VM will be tunneled in extra underlay encapsulations, MPLSoUDP in this case, so the vif 0/0 bps number (1282176) will be a little bit bigger comparing with the number on the VM interface.

This script conveniently gives a straightforward overview about the current traffic profile from vRouter’s perspective. To compare with the original vif output which the script is based on, let’s check what the raw data looks like without the dpdkvifstats.py script:

[root@a7s3 ~]# vif --clear; sleep 1; vif --get 3 --core 10; vif --get 3 --core 11  
  
Vif stats cleared successfully on all cores for all interfaces  
......  
  
vif0/3 PMD: tap41a9ab05-64 NH: 34  
 Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.1.104  
 Vrf:2 Mcast Vrf:2 Flags:PL3L2DEr QOS:-1 Ref:12  
 RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0  
 Core 10 RX packets:1488 bytes:89280 errors:0  
 Core 10 TX packets:0 bytes:0 errors:0  
 ISID: 0 Bmac: 02:41:a9:ab:05:64  
 Drops:131  
  
......  
vif0/3 PMD: tap41a9ab05-64 NH: 34  
 Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.1.104  
 Vrf:2 Mcast Vrf:2 Flags:PL3L2DEr QOS:-1 Ref:12  
 Core 11 RX queue packets:1496 errors:0  
 RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0  
 Core 11 RX packets:0 bytes:0 errors:0  
 Core 11 TX packets:0 bytes:0 errors:0  
 ISID: 0 Bmac: 02:41:a9:ab:05:64  
 Drops:131

We captured the interface data, waited for one second, and then captured it again. After that we can calculate the differences of all the counters between the two captures to get the increasing rate of each counter:

* pps - packets per second: 1488 pps
* Bps - bytes per second: 1488 \* 60 = 89280 Bps
* bps - bit per second: 89280 \* 8 = 714240 bps

These numbers are consistent with what is seen in the dpdkvifstats.py script. But to monitor multiple vif interfaces you have to repeat these steps multiple times. Compare this with having a handy script doing everything for you!

As mentioned, dpdkvifstats.py script is useful to quickly retrieve a snapshot of current traffic profile. When everything goes well it is fine. In the case of traffic loss, you often need to first capture the packets themselves, then from one of the packet captures you can decode the payload and take a deeper look to analyze the issue. Now, you may say, oh you mean tcpdump! Well, yes and no. Please remember the fact that we are in a setup where the NIC card is invisible to most of the Linux applications, including tcpdump!

Next let’s go over another DPDK vRouter packet capture script: vifdump.

### vifdump script

In many Linux machines, tcpdump comes with the OS as part of a standard package. With it you can capture whatever packets sensed by a NIC, which can be either physical NIC or virtual NIC, like a tuntap interface. Both NICs are visible to the kernel. In DPDK environments, the difficulty of an interface not being visible to the kernel makes tcpdump unworkable, unless you just want it to read packets from a file.

Fortunately, we now know that each interface related to the vRouter data plane connects to a unique vRouter interface (vif). We can make use of this fact and create an alternative. vifdump is a shell script and when invoked it uses the --add option of the vif command to create a monitoring tun interface in the Linux kernel, and internally the vRouter will clone all data passing through the monitored vif interface to this kernel interface. vifdump will then start up the tcpdump program to capture the packets from the monitoring tun interface. From a user’s perspective, the script works the same way as with tcpdump. Here are two captures on vif0/3 toward VM, which is our PROX gen, and on vif0/0 toward the fabric interface:

[root@a7s3 ~]# contrail-tools vifdump -i 3 -n -c 3  
vif0/3 PMD: tap41a9ab05-64 NH: 32  
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode  
listening on mon3, link-type EN10MB (Ethernet), capture size 262144 bytes  
13:12:31.286528 IP 192.168.1.104.filenet-cm > 192.168.1.105.filenet-nch: UDP, length 82  
13:12:31.286532 IP 192.168.1.104.filenet-rmi > 192.168.1.105.filenet-pch: UDP, length 82  
13:12:31.286540 IP 192.168.1.104.filenet-rpc > 192.168.1.105.filenet-pa: UDP, length 82  
3 packets captured  
401 packets received by filter  
271 packets dropped by kernel  
vifdump: deleting vif 4348...  
  
[root@a7s3 ~]# contrail-tools vifdump -i 0 -n -c 3  
vif0/0 PCI: 0000:00:00.0 (Speed 20000, Duplex 1) NH: 4  
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode  
listening on mon0, link-type EN10MB (Ethernet), capture size 262144 bytes  
13:12:23.796516 IP 8.0.0.4.55184 > 8.0.0.2.4789: VXLAN, flags [I] (0x08), vni 8  
IP 192.168.1.104.filenet-pa > 192.168.1.105.filenet-nch: UDP, length 82  
13:12:23.796522 IP 8.0.0.4.54530 > 8.0.0.2.4789: VXLAN, flags [I] (0x08), vni 8  
IP 192.168.1.104.filenet-rmi > 192.168.1.105.filenet-pa: UDP, length 82  
13:12:23.796531 IP 8.0.0.4.63363 > 8.0.0.2.4789: VXLAN, flags [I] (0x08), vni 8  
IP 192.168.1.104.filenet-nch > 192.168.1.105.filenet-pch: UDP, length 82  
3 packets captured  
334 packets received by filter  
271 packets dropped by kernel  
vifdump: deleting vif 4351...  
[root@a7s3 ~]#

The shell script also uses UNIX trap to monitor signals and deletes the monitoring interface when the signals appear. The most used signal is SIGINT triggered by the keyboard’s ctrl-c to stop the capture. That is why we see the vifdump: deleting vif 4351… message at the end of each capture.

dpdkvifstats.py and vifdump are two scripts developed based on the vif command. With these tools we can collect general packet RX/TX counters and packet contents.

In the next section, we’ll take a look another powerful debug tool that is useful in the DPDK environment: dpdkinfo.

## dpdkinfo Command

We’ve talked about vif and dpdkvifstats.py tools. Now let’s introduce a relatively new tool that can be used to investigate lower level details of DPDK interfaces. It’s dpdkinfo and it was introduced in Contrail 2008. With it, Contrail operators can collect more information about DPDK vRouter fabric interface internal status, connectivity (physical NIC bond), DPDK library information, and some other statistics.

Let’s first run the tool with -h to get a brief menu of it:

(contrail-tools)[root@a7s3 /]$ dpdkinfo -h  
Usage: dpdkinfo  
 --help  
 --version|-v Show DPDK Version  
 --bond|-b Show Master/Slave bond information  
 --lacp|-l <all/conf> Show LACP information from DPDK  
 --mempool|-m <all/<mempool-name>> Show Mempool information  
 --stats|-n <eth> Show Stats information  
 --xstats|-x <=all/=0(Master)/=1(Slave(0))/=2(Slave(1))>  
 Show Extended Stats information  
 --lcore|-c Show Lcore information  
 --app|-a Show App information  
 Optional: --buffsz <value> Send output buffer size (less than 1000Mb)

From this help information we can see it provides information about DPDK interface in multiple areas. In this rest of this section, let’s take a look at some of the most useful options:

* --version|-v
* --bond|-b
* --lacp|-l
* --stats|-n
* --xstats|-x
* --lcore|-c

There are some other options like --app|-a, --mempool|-m that we won’t introduce in this book, and the list of supported functions may grow in each future releases. But you will get the basic idea of its usage and you can refer to the official documents for up to date information.

### version

The -v or --version option reports the basic version information of DPDK release in use:

(contrail-tools)[root@a7s3 /]$ dpdkinfo -v  
DPDK Version: DPDK 19.11.0  
vRouter version: {build-info: [{build-time: 2020-09-04 10:38:22.330666, build-hostname: 6fb64a1f86b9, build-user: root, build-version: 2004}]}

### bond and LACP status

The -b or --bond option displays detailed information about the bond interface managed by DPDK. The output is organized in a similar form as what you would see for the bond status managed by Linux kernel. Compare the output in Figure 5.15 with cat /proc/net/bonding/bond0 output from a compute running kernel mode vRouter.

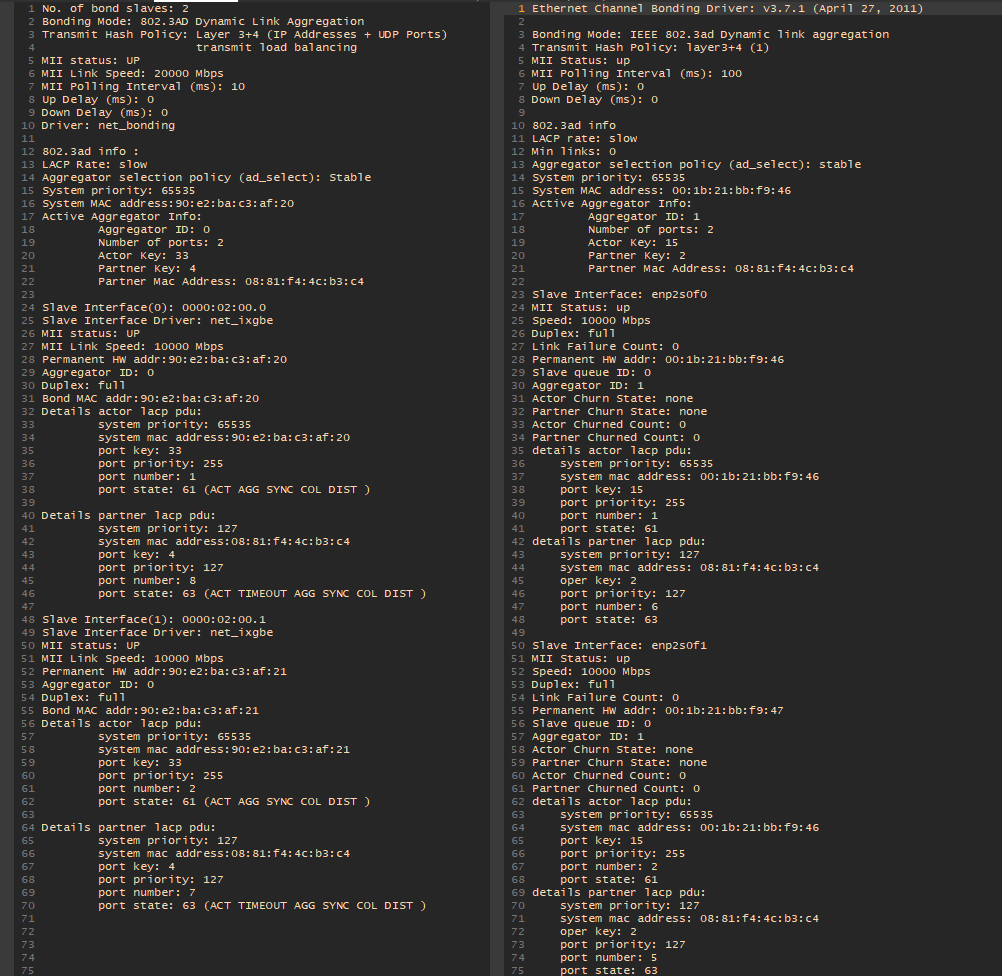


Figure 5.15 DPDK Info -b vs. cat /proc/net/bonding/bond0

Basically you can have same information as a Linux kernel bond0, such as bonding mode, transmit hash policy, system MAC and aggregator information, etc. In this example the current bonding mode is 802.3AD dynamic link aggregation, indicating the LACP protocol is configured between compute and peer devices (in our environment it’s a TOR switch). The Transmit Hash Policy shows Layer 3+4 (IP Addresses + UDP Ports) transmit load balancing, which the bond allows for traffic to a particular network peer to span multiple clients for load balancing purpose. This is achieved by calculating a hash value for each packet from the IP addresses and UDP ports in the outer header of the packet, and then distributing the packet based on the hash value.

The command output also displays each member link’s information, its current driver, MAC address, up/down status, etc.

Since LACP is running, for each member link LACP parameters are displayed. Another way to show this information is with -l|--lacp option:

[root@a7s3 ~]# contrail-tools dpdkinfo -l all  
LACP Rate: slow  
  
Fast periodic (ms): 900  
Slow periodic (ms): 29000  
Short timeout (ms): 3000  
Long timeout (ms): 90000  
Aggregate wait timeout (ms): 2000  
Tx period (ms): 500  
Update timeout (ms): 100  
Rx marker period (ms): 2000  
  
Slave Interface(0): 0000:02:00.0  
Details actor lacp pdu:  
 port state: 61 (ACT AGG SYNC COL DIST )  
  
Details partner lacp pdu:  
 port state: 63 (ACT TIMEOUT AGG SYNC COL DIST )  
  
Slave Interface(1): 0000:02:00.1  
Details actor lacp pdu:  
 port state: 61 (ACT AGG SYNC COL DIST )  
  
Details partner lacp pdu:  
 port state: 63 (ACT TIMEOUT AGG SYNC COL DIST )  
  
LACP Packet Statistics:  
 Tx Rx  
0000:02:00.0 13414 413  
0000:02:00.1 13414 414

Here, you can get more insight of LACP running stats, including all LACP timers and PDU statistics about number of packets exchanged with the peer device. Of course, here the counters are LACP PDU only. If you need all packets received and sent through the bond interface, you can use -n|--stats option.

### bond packet counters

The -n|--stats option is useful to look into the packet statistics of the bond interface. So far we’ve seen at least two ways of retrieving packet counters from a vif interface:

* vif --get X
* dpdkvifstats.py -v X

The DPDK bond interface is represented by the vRouter interface vif0/0, so you may think setting X to 0 in the above commands achieves the same effect. The problem is none of these tools print packet statistics for each member link of the bond. Let’s take a look at an example here:

[root@a7s3 ~]# contrail-tools dpdkinfo --stats eth  
 Master Info:  
 RX Device Packets:28360664, Bytes:3233321316, Errors:0, Nombufs:0  
 Dropped RX Packets:0  
 TX Device Packets:28361174, Bytes:3234763122, Errors:0  
 Queue Rx: [0]28360664  
 Tx: [0]28361174  
 Rx Bytes: [0]3233321316  
 Tx Bytes: [0]3234760294  
 Errors:  
 ---------------------------------------------------------------------  
  
 Slave Info(0000:02:00.0):  
 RX Device Packets:1421, Bytes:129257, Errors:0, Nombufs:0  
 Dropped RX Packets:0  
 TX Device Packets:28358167, Bytes:3234235595, Errors:0  
 Queue Rx: [0]1421  
 Tx: [0]28358167  
 Rx Bytes: [0]129257  
 Tx Bytes: [0]3234232767  
 Errors:  
 ---------------------------------------------------------------------  
  
 Slave Info(0000:02:00.1):  
 RX Device Packets:28359275, Bytes:3233195707, Errors:0, Nombufs:0  
 Dropped RX Packets:0  
 TX Device Packets:3039, Bytes:531175, Errors:0  
 Queue Rx: [0]28359275  
 Tx: [0]3039  
 Rx Bytes: [0]3233195707  
 Tx Bytes: [0]531175  
 Errors:  
 ---------------------------------------------------------------------

With the --stats eth option, dpdkinfo displays traffic distribution among all member links of a DPDK bond interfaces. For example, in this example, we are seeing the first member link (PCI bus 0000:02:00.0) receive 1421 packets, while the second member link (PCI bus 0000:02:00.1) receive 28359275 packets. It is obvious that the second member link carries most of the traffic. Maybe you are wondering why we ended up with imbalanced traffic distributions, because previously we’ve mentioned earlier that Transmit Hash Policy is set to load balancing across member links. The reason is that in this test environment we are sending just one UDP flow!

With more flows we’ll see the balance happens. let’s send ten flows, but before that let’s clear the current counters to make our second comparison easier:

[root@a7s3 ~]# contrail-tools vif --clear  
  
Vif stats cleared successfully on all cores for all interfaces

Now, start the rapid script to send 64 flows, and check same dpdkinfo command output again:

[root@a7s3 ~]# contrail-tools dpdkinfo -n eth  
 Master Info:  
 RX Device Packets:471211, Bytes:53724144, Errors:0, Nombufs:0  
 Dropped RX Packets:0  
 TX Device Packets:471189, Bytes:53719798, Errors:0  
 Queue Rx: [0]471211  
 Tx: [0]471190  
 Rx Bytes: [0]53724144  
 Tx Bytes: [0]53719884  
 Errors:  
 ---------------------------------------------------------------------  
  
 Slave Info(0000:02:00.0):  
 RX Device Packets:228370, Bytes:26033818, Errors:0, Nombufs:0  
 Dropped RX Packets:0  
 TX Device Packets:220073, Bytes:25090326, Errors:0  
 Queue Rx: [0]228370  
 Tx: [0]220076  
 Rx Bytes: [0]26033818  
 Tx Bytes: [0]25090640  
 Errors:  
 ---------------------------------------------------------------------  
  
 Slave Info(0000:02:00.1):  
 RX Device Packets:242872, Bytes:27693860, Errors:0, Nombufs:0  
 Dropped RX Packets:0  
 TX Device Packets:251148, Bytes:28633120, Errors:0  
 Queue Rx: [0]242872  
 Tx: [0]251158  
 Rx Bytes: [0]27693860  
 Tx Bytes: [0]28634260  
 Errors:  
 ---------------------------------------------------------------------

From the member link packet statistics, you can see that the traffic is balanced on both links.

Now that you understand that the -stats|-n option provides the insight of member link usage reflected by a few RX/TX counters. Base on this information you can determine the load balance status of a DPDK bond interface. So far, all of the packet counters we’ve seen, no matter under master or members, are almost the same ones as what are provided by the vif command. In practice, if you need to get more extensive statistics, there is another option: xstats|-x. Let’s check it out:

[root@a7s3 ~]# contrail-tools dpdkinfo -xall | grep -v : 0  
Master Info:  
Rx Packets: Rx Bytes:  
 rx\_good\_packets: 852475379 rx\_good\_bytes: 97185979648  
 rx\_q0packets: 852475379 rx\_q0bytes: 97185979648  
Tx Packets: Tx Bytes:  
 tx\_good\_packets: 852853117 tx\_good\_bytes: 97253818091  
 tx\_q0packets: 852853127 tx\_q0bytes: 97253769503  
Errors:  
Others:  
 ------------------------------------------------------------------  
  
Slave Info(0):0000:02:00.0 Slave Info(1):0000:02:00.1  
Rx Packets: Rx Packets:  
 rx\_good\_packets: 412875343 rx\_good\_packets: 439600104  
 rx\_q0packets: 412875343 rx\_q0packets: 439600104  
 rx\_size\_64\_packets: 5939 rx\_size\_64\_packets: 19  
 **rx\_size\_65\_to\_127\_packets: 412869003 rx\_size\_65\_to\_127\_packets: 439553375** rx\_size\_128\_to\_255\_packets: 191 rx\_size\_128\_to\_255\_packets: 42367  
 rx\_size\_256\_to\_511\_packets: 206 rx\_size\_256\_to\_511\_packets: 1173  
 rx\_broadcast\_packets: 5882 rx\_size\_512\_to\_1023\_packets: 1242  
 rx\_multicast\_packets: 6124 rx\_size\_1024\_to\_max\_packets: 1922  
 rx\_total\_packets: 412875340 rx\_multicast\_packets: 396  
Tx Packets: rx\_total\_packets: 439600098  
 tx\_good\_packets: 399807799 Tx Packets:  
 tx\_q0packets: 399807802 tx\_good\_packets: 453045397  
 tx\_total\_packets: 399807792 tx\_q0packets: 453045399  
 tx\_size\_64\_packets: 3552 tx\_total\_packets: 453045389  
 **tx\_size\_65\_to\_127\_packets: 399717757 tx\_size\_65\_to\_127\_packets: 453035768**  
 tx\_size\_128\_to\_255\_packets: 59597 tx\_size\_128\_to\_255\_packets: 6448  
 tx\_size\_256\_to\_511\_packets: 10695 tx\_size\_256\_to\_511\_packets: 9  
 tx\_size\_512\_to\_1023\_packets: 831 tx\_size\_512\_to\_1023\_packets: 1680  
 tx\_size\_1024\_to\_max\_packets: 15360 tx\_size\_1024\_to\_max\_packets: 1484  
 tx\_multicast\_packets: 6365 tx\_multicast\_packets: 6365  
 tx\_broadcast\_packets: 2941 Rx Bytes:  
Rx Bytes: rx\_good\_bytes: 50119065424  
 rx\_good\_bytes: 47066921976 rx\_q0bytes: 50119065424  
 rx\_q0bytes: 47066921976 rx\_total\_bytes: 50119064740  
 rx\_total\_bytes: 47066921752 Tx Bytes:  
Tx Bytes: tx\_good\_bytes: 51649995369  
 tx\_good\_bytes: 45603831138 tx\_q0bytes: 51649996187  
 tx\_q0bytes: 45603781752 Errors:  
Errors: Others:  
Others: rx\_l3\_l4\_xsum\_error: 439588641  
 rx\_l3\_l4\_xsum\_error: 412856784 out\_pkts\_untagged: 474447816  
 out\_pkts\_untagged: 549754060  
 ------------------------------------------------------------------

As you can see, the output is *very* extensive – perhaps ten times more than what vif, dpdkvifstats.py and dpdkinfo -n eth provide. In fact, to shorten the output, we’ve removed all counters with a zero value in them, and also edited the output format to compact all texts into two columns. If you go through it quickly, you will be able to tell the fact that the majority part of the traffic is composed of packets with sizes between 65 to 127 bytes, and that is what we are sending from the rapid script. Increasing traffic packet size from rapid will end up with a different result:

[root@a7s3 ~]# contrail-tools dpdkinfo -xall | grep -v : 0  
Master Info:  
....  
 --------------------------------------------------------------------  
Slave Info(0):0000:02:00.0 Slave Info(1):0000:02:00.1  
Rx Packets: Rx Packets:  
 rx\_good\_packets: 7902180 rx\_good\_packets: 7896450  
 rx\_q0packets: 7902180 rx\_q0packets: 7896450  
 rx\_size\_64\_packets: 302 rx\_size\_64\_packets: 1  
 rx\_size\_65\_to\_127\_packets: 1731 rx\_size\_65\_to\_127\_packets: 389  
 **rx\_size\_128\_to\_255\_packets: 7900126 rx\_size\_128\_to\_255\_packets: 7895820**  
 rx\_size\_256\_to\_511\_packets: 15 rx\_size\_256\_to\_511\_packets: 66  
 rx\_size\_512\_to\_1023\_packets: 3 rx\_size\_512\_to\_1023\_packets: 69  
 rx\_size\_1024\_to\_max\_packets: 3 rx\_size\_1024\_to\_max\_packets: 105  
 rx\_broadcast\_packets: 299 rx\_multicast\_packets: 20  
 rx\_multicast\_packets: 312 rx\_total\_packets: 7896450  
 rx\_total\_packets: 7902180 Tx Packets:  
Tx Packets: tx\_good\_packets: 8272747  
 tx\_good\_packets: 7536810 tx\_q0packets: 8272747  
 tx\_q0packets: 7536810 tx\_total\_packets: 8272747  
 tx\_total\_packets: 7536810 tx\_size\_65\_to\_127\_packets: 179  
 tx\_size\_64\_packets: 181 **tx\_size\_128\_to\_255\_packets: 8272496** tx\_size\_65\_to\_127\_packets: 290 tx\_size\_256\_to\_511\_packets: 17  
 **tx\_size\_128\_to\_255\_packets: 7535143** tx\_size\_512\_to\_1023\_packets: 53  
 tx\_size\_256\_to\_511\_packets: 223 tx\_size\_1024\_to\_max\_packets: 2  
 tx\_size\_512\_to\_1023\_packets: 90 tx\_multicast\_packets: 324  
 tx\_size\_1024\_to\_max\_packets: 883 Rx Bytes:  
 tx\_multicast\_packets: 323 rx\_good\_bytes: 1405706413  
 tx\_broadcast\_packets: 150 rx\_q0bytes: 1405706413  
Rx Bytes: rx\_total\_bytes: 1405706413  
 rx\_good\_bytes: 1406393359 Tx Bytes:  
 rx\_q0bytes: 1406393359 tx\_good\_bytes: 1472542701  
 rx\_total\_bytes: 1406393359 tx\_q0bytes: 1472542701  
Tx Bytes: Errors:  
 tx\_good\_bytes: 1342701308 Others:  
 tx\_q0bytes: 1342698774 rx\_l3\_l4\_xsum\_error: 7895846  
Errors: out\_pkts\_untagged: 3532820029  
Others:  
 rx\_l3\_l4\_xsum\_error: 7901213  
 out\_pkts\_untagged: 3249154601  
 --------------------------------------------------------------------

We won’t discuss all the counters listed in this output, for now, just add dpdkinfo with these two options -n|stats and -x|xstats in your DPDK vRouter troubleshooting toolkits. Considering to use them to collect information whenever you run into traffic loss issues during your lab test or production deployment.

Next we’ll explore another interesting option -c|--lcore.

### lcore

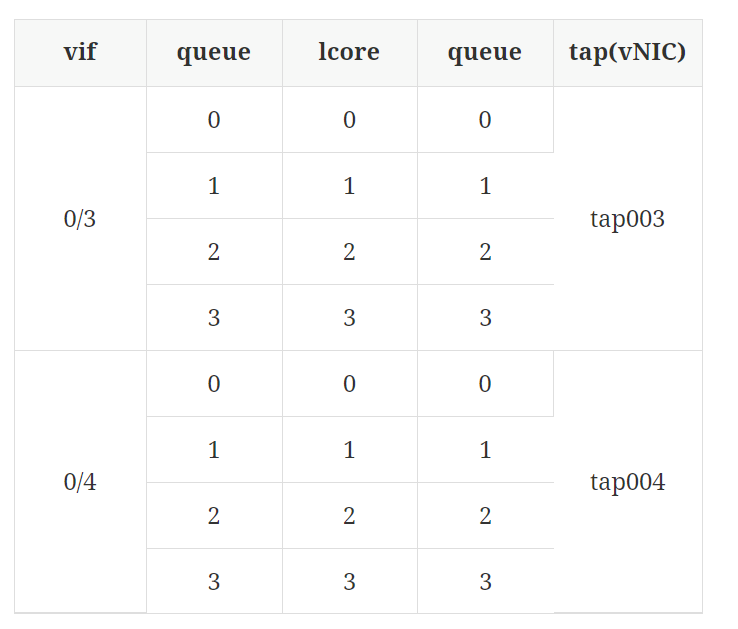
There are several key concepts we’ve been trying to illustrate in this book. Among others, at least four of them are often mentioned together: lcore, interface, and queue, but before we start introducing the fourth, the -c|--lcore option, let’s briefly review these concepts.

Lcore: lcore is a thread in vRouter DPDK process running in user space

interface:This is the endpoint of connections between the vRouter and the other VM, or between vRouter and the outside of the compute. At the vRouter and VM end, the interfaces are called vif and tap interfaces respectively. There are also bond0 physical interface in DPDK user space and vhost0 interface in Linux kernel. The former is the physically NIC bundle connecting to the peer device, and the latter gives the host an IP address and through which the vRouter agent can exchange control plane messages with the controller.

Queue: For each interface there are some queues created. They are essentially allocated memory to hold the packets.

The CPU cores connect all these objects together. As of the writing of this book, the implementation is to have one-to-one mapping between the number of CPU cores allocated to vRouter and the number of interface queues. For example, if four CPUs are allocated to the DPDK vRouter forwarding threads (the lcores), then four lcores will be created, and four DPDK interface queues will be created for each vif interface. The same rule applies to the VM. You assign four CPU cores to a VM, then by default, Nova will create four queues for a tap interface in the VM. That said, of course, multiple queue as a feature needs to be turned on in Nova. We can illustrate with the following table:



This is just a simple example. In production deployment there are a lot more conditions to consider, and a lot of confusions rise. Common questions are:

* What if the tap interface queue number is different than the vif queue number? What will happen when there are eight lcores but one of our VMs is running four queues in its tap interface?
* Will vif0/3 queue0 always be served by lcore0, instead of other lcores? If not, how to determine which vif queue goes to which lcore? Is there a chance that imbalanced lcores to queue mapping happens, so that some lcores are overloaded and some lcores are relatively idle?

To answer these questions, we need a tool to reveal the secret of actual mapping between lcores and queues from different vif interfaces. This is the moment for -c|--lcore option of dpdkinfo to show its power. Again, let’s start with an example:

[root@a7s3 ~]# contrail-tools dpdkinfo -c  
No. of forwarding lcores: 2  
No. of interfaces: 4  
Lcore 0:  
 Interface: bond0.101 Queue ID: 0  
 Interface: vhost0 Queue ID: 0  
  
Lcore 1:  
 Interface: bond0.101 Queue ID: 1  
 Interface: tap41a9ab05-64 Queue ID: 0

Let’s start from the first line. In this example, we have allocated two CPU cores to DPDK vRouter forwarding lcores, so we have two forwarding lcores running in total.

Then, the second line provides the number of vRouter interfaces in the compute. We have four of them in total. One vif0/4 connecting to VM tap interface tap41a9ab05-64, and three mandatory, vif0/0, vif0/1, vif0/2, connecting to bond, vhost0, and pkt0 respectively. Here, we have created just one VM (actually this is nothing but the PROX gen VM we’ve created earlier) with only one tap interface.

Let’s focus on the third line onward. The output is listing all forwarding lcores that are currently configured in vRouter, and for each lcore it lists interfaces that each lcore is associated with, in another words, interfaces this core is serving.

Please note that there are some inconsistencies in terms of the lcore numbering in different tools:

* In dpdkvifstats.py script, the forwarding lcore number starts from one, so Core 1 refers to the first forwarding lcore.
* In dpdkinfo -c output, forwarding lcore number starts from zero, so Lcore 0 refers to the first forwarding lcore.
* In vif output, forwarding lcore number starts from ten, so --core 10 refers to the first forwarding lcore.

This can cause confusion in our discussions. To make it consistent, in the rest of this chapter we’ll use the first forwarding lcore, fwd lcore#10, or simply lcore#10; the second forwarding lcore, fwd lcore#11, or simply lcore#11, and so on, to indicate Lcore 0, Lcore 1 in dpdkinfo-c output, Core 1, Core 2 in dpdkvifstats.py script output, and Core 10, Core 11 in vif output, respectively. Here’s a better visualization.

|  |  |  |  |
| --- | --- | --- | --- |
| vif | dpdkinfo -c | dpdkvifstats.py | meaning |
| Core 10 | Lcore 0 | Core 1 | 1st forwarding lcore: lcore#10 |
| Core 11 | Lcore 1 | Core 2 | 2nd forwarding lcore: lcore#11 |

Okay, as you may have realized, in the VM interface we use just one queue, which means the multiple queue feature on the VM interface is *not* enabled. Therefore the VM tap interface has only one queue connecting to its peering vRouter interface. Correspondingly, only one queue in the vRouter interface is needed and only one lcore is required to serve the packet forwarding in the vif interface.

First, let’s look at the bond0 and vhost0 interfaces. The bond0 are the physical interfaces, and it will always have multiple queues enabled, that is why it has two queues, and both lcores serve it. The vhost0 interface is a control plane Linux interface. As the time of writing of this book, the implementation is to hard-code vhost0 with one queue only. The first forwarding thread lcore#10 got it. This is not the focus in this section but it’s good to know to understand the whole output.

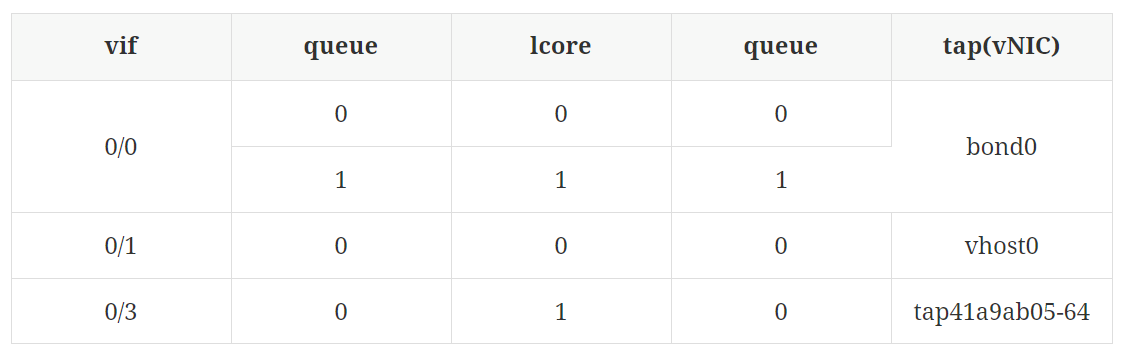
Finally, let’s look at the last line, the VM tap interface. From the output, you can see it is the second forwarding lcore (lcore#11) being assigned to this VM interface. You probably wonder is it just randomly chosen out of the two lcores or is some algorithm used? It is not like that. Currently the allocation basically follows a simple method:

**The least used lcore, in term of number of interface queues it is serving, will be assigned to serve the next interface queue.**

Based on what we just explained, lcore#10 took two interfaces (bond0.101 and vhost0) while lcore#11 took just one (bond0.101), so it’s lcore#11’s turn to take the next interface and queue.

The vNIC queues are assigned to logical cores in the following algorithm: the forwarding core that is currently polling the least number of queues is selected, with a tie won by the core with the lowest number (the first forwarding core lcore#10).

You’ll see more examples in later sections, when we’ll test out the tie breaker and other things. For now this mapping looks like this matrix:



Now that we’ve gone through the dpdkinfo command and demonstrated some of the most commonly used options, you can quickly display a lot of useful information about DPDK and DPDK vRouter running status. We’ll review this again later in our test case studies. The information is important before working on any deployment or troubleshooting task in the setup. However, when things go wrong, instead of just relying on the DPDK commands output, you may also want to check into the log messages to verify the current running status is as what you’ve expected it to be.

Next we’ll take a look at DPDK vRouter log messages.

## DPDK vRouter Log Files

Contrail’s DPDK vrouter data plane log file is named contrail-vrouter-dpdk.log. Depending on the version or installation methods, it can be located in different folders or even with a totally different name. For example:

* In latest TripleO deployment: /var/log/containers/contrail/dpdk/contrail-vrouter-dpdk.log
* In latest Ansible deployment: /var/log/contrail/contrail-vrouter-dpdk.log
* In older 3.x Ubuntu deployemnt: /var/log/contrail.log

This log file contains lots of good information that is helpful to understand the current running status. Of course, understanding the log messages is important during a troubleshooting process.

### DPDK vrouter parameters

Each time the vrouter is started, the main configuration parameters are listed in the log file during the vrouter initialization stage. You can see the DPDK library version that has be used to build the DPDK vrouter binary program:

2020-09-15 20:27:22,381 VROUTER: vRouter version: {build-info:  
[{build-time: 2020-09-15 01:07:25.101398, build-hostname:  
contrail-build-r2008-rhel-115-generic-20200914170527.novalocal, build-user:  
contrail-builder, build-version: 2008}]}  
2020-09-15 20:27:22,382 VROUTER: DPDK version: DPDK 19.11.0  
2020-09-15 20:27:23,046 VROUTER: Log file : /var/log/contrail/contrail-vrouter-dpdk.log  
2020-09-15 20:27:23,046 VROUTER: Bridge Table limit: 262144  
2020-09-15 20:27:23,046 VROUTER: Bridge Table overflow limit: 53248  
2020-09-15 20:27:23,046 VROUTER: Flow Table limit: 524288  
2020-09-15 20:27:23,046 VROUTER: Flow Table overflow limit: 105472  
2020-09-15 20:27:23,046 VROUTER: MPLS labels limit: 5120  
2020-09-15 20:27:23,046 VROUTER: Nexthops limit: 32768  
2020-09-15 20:27:23,046 VROUTER: VRF tables limit: 4096  
2020-09-15 20:27:23,046 VROUTER: Packet pool size: 16384  
2020-09-15 20:27:23,046 VROUTER: PMD Tx Descriptor size: 128  
2020-09-15 20:27:23,046 VROUTER: PMD Rx Descriptor size: 128  
2020-09-15 20:27:23,046 VROUTER: Maximum packet size: 9216  
2020-09-15 20:27:23,046 VROUTER: Maximum log buffer size: 200  
2020-09-15 20:27:23,046 VROUTER: VR\_DPDK\_RX\_RING\_SZ: 2048  
2020-09-15 20:27:23,046 VROUTER: VR\_DPDK\_TX\_RING\_SZ: 2048  
2020-09-15 20:27:23,046 VROUTER: VR\_DPDK\_YIELD\_OPTION: 0  
2020-09-15 20:27:23,046 VROUTER: VR\_SERVICE\_CORE\_MASK: 0x10  
2020-09-15 20:27:23,046 VROUTER: VR\_DPDK\_CTRL\_THREAD\_MASK: 0x10  
2020-09-15 20:27:23,046 VROUTER: Unconditional Close Flow on TCP RST: 0  
2020-09-15 20:27:23,046 VROUTER: EAL arguments:  
2020-09-15 20:27:23,046 VROUTER: -n 4  
2020-09-15 20:27:23,046 VROUTER: --socket-mem 1024

You can see the complete list of vRouter start up parameters on this Contrail vRouter, for example:

* build-version 2008
* it’s running DPDK Version 19.11.0
* Nexthops limit parameter is configured as 32768, decreased from the default value (65536)
* CPU core #4 is pinned to be used by control and service thread (VR\_SERVICE\_CORE\_MASK: 0x10)

We can compare this information with what we can display with these command line tools and see if they are consistent:

* contrail-version
* dpdkinfo -v
* vrouter --info
* taskset

Any inconsistency will provide a clue to proceed in that area.

### Polling core allocation

In Chapter 3 we introduced the DPDK vRouter process. It is a multiple thread application and the threads fall into different categories based on their roles. This is also reflected by some log entries. Before diving into the logs, let’s do a quick review of the three thread categories:

* Control threads: These are generated by DPDK libraries and are used during Contrail vRouter startup for DPDK initialization. Control threads are not our focus in this book.
* Service threads: These two service threads are totally hard-coded named lcore0 through lcore9. Each lcore has its own role. For example, lcore9 serves netlink connections between agent and vRouter data plane. Details of each lcore’s rule is out of this book’s scope. You just need to know that they are used to serve communication between the vrouter agent and vrouter forwarding plane.
* Forwarding threads: After service threads, from lcore10 and onward, the forwarding threads are the horsepower that performs the packet forwarding tasks and determines the performance of the DPDK vRouter. *This is the main focus of our book.*

NOTE In service threads, lcore3 to lcore7 are never used in Contrail DPDK vRouter.

Okay, now let’s take a look at an interesting log entry:

2020-09-16 09:06:50,886 VROUTER: --lcores  (0-2)@(10,34),(8-9)@(10,34),\*10@2,11@4,12@6,13@8\*

Here, we understand that the string --lcores means a service thread, or a forwarding thread. Following this string is a few coupled numbers connected by @ - NUMBER@NUMBER, which are seperated by commas. How to decode these? Well, to understand this you need to understand CPU pinning. To achieve maximum performance we’re pinning each of the service and forwarding threads (or lcores) with a few specific CPU cores, so each thread will be served by dedicated CPUs that are isolated from any other system tasks. So this log reads:

* Service threads, that is lcore0 to lcore2, and lcore8-lcore9 in the message, are all pinned to two CPU cores: core#10 and CPU core#34. The pinning is configured by the SERVICE\_CORE\_MASK parameter.
* Forwarding threads, lcore10 to lcore13, are allocated and are pinned to CPU core#2, core#4, core#6 and core#8, respectively. This is configured from the CPU\_LIST parameter.

### Internal Load Balancing

In some situations the polling core performs a new hash calculation to distribute the polled packets to another processing core. This is a DPDK pipeline model implemented in the vrouter.

This distribution behavior can be observed in the following messages in DPDK log file:

2020-01-07 13:08:01,403 VROUTER: Lcore 10: distributing MPLSoGRE packets to [11,12,13]  
2020-01-07 13:08:01,403 VROUTER: Lcore 11: distributing MPLSoGRE packets to [10,12,13]  
2020-01-07 13:08:01,403 VROUTER: Lcore 12: distributing MPLSoGRE packets to [10,11,13]  
2020-01-07 13:08:01,404 VROUTER: Lcore 13: distributing MPLSoGRE packets to [10,11,12]

Here the logs show MPLSoGRE, but it actually applies to both MPLSoGRE or VxLAN packets. This is historically due to when only MPLSoGRE was supported. So, it remains like that in the software code. Here it means both MPLSoGRE and VxLAN packets will be distributed via hashing by the polling core. See Figure 5.16.

![image](data:image/emf;base64;base64,)

Figure 5.16 Virtual Interface queues

Each time a new virtual interface is connected to the vrouter, a vif port is created on the vrouter with the same number of queues as the number of polling CPUs (specified in CPU\_LIST parameter). Each queue is created and handled by only one of the vrouter polling cores. So, for each vif, we have a one-to-one mapping between vrouter polling cores and RX queues. This mapping can be seen from dpdkinfo -c command output which we’ve introduced. The same can be observed in the DPDK vrouter logs:

2019-09-24 16:36:50,011 VROUTER: Adding vif 8 (gen. 37) virtual device tap66e68bc1-a9  
....  
2019-09-24 16:36:50,012 VROUTER: lcore 12 RX from HW queue 0  
2019-09-24 16:36:50,012 VROUTER: lcore 13 RX from HW queue 1  
2019-09-24 16:36:50,012 VROUTER: lcore 10 RX from HW queue 2  
2019-09-24 16:36:50,012 VROUTER: lcore 11 RX from HW queue 3

Here, the vif interface 0/8 is created in order to connect the virtual NIC tap66e68bc1-a9 to the vrouter. Because four forwarding lcores are configured, this vif is created with 4 queues, namely q0 to q3, which are respectively handled by polling cores 12,13,10, and 11.

When a polling queue is enabled on the vrouter, a ring activation message is generated in the Contrail DPDK log file.

The vrings correspond to both transmit and receive queues:

* The transmit queues are the even numbers. Divide them by two to get the queue number: vring 0 is TX queue 0, vring 2 is TX queue 1, … and so on.
* The receive queues are the odd numbers. Divide them by two (discard the remainder) to get the queue number: vring 1 is RX queue 0, vring 3 is RX queue 1, … and so on.
* Ready state 1 = enabled, ready state 0 = disabled.

![image](data:image/emf;base64;base64,)

Figure 5.17 Vrings Correspond to Both Transmit and Receive Queues

In this next example, only 1 RX (and TX) queue is enabled on the vrouter vif interface. A single queue virtual machine interface is connected to the vrouter port:

2019-09-24 16:37:46,693 UVHOST: Client \_tap66e68bc1-a9: setting vring 0 ready state 1  
2019-09-24 16:37:46,693 UVHOST: Client \_tap66e68bc1-a9: setting vring 1 ready state 1  
2019-09-24 16:37:46,693 UVHOST: Client \_tap66e68bc1-a9: setting vring 2 ready state 0  
2019-09-24 16:37:46,693 UVHOST: Client \_tap66e68bc1-a9: setting vring 3 ready state 0  
2019-09-24 16:37:46,693 UVHOST: Client \_tap66e68bc1-a9: setting vring 4 ready state 0  
2019-09-24 16:37:46,693 UVHOST: Client \_tap66e68bc1-a9: setting vring 5 ready state 0  
2019-09-24 16:37:46,693 UVHOST: Client \_tap66e68bc1-a9: setting vring 6 ready state 0  
2019-09-24 16:37:46,693 UVHOST: Client \_tap66e68bc1-a9: setting vring 7 ready state 0

And in the next example, four RX (and TX) queues are enabled on the vrouter vif interface, but a virtual machine interface having more than four queues is connected to the vrouter port:

2019-09-24 16:37:46,693 UVHOST: Client \_tap66e68bc1-a9: setting vring 0 ready state 1  
2019-09-24 16:37:46,693 UVHOST: Client \_tap66e68bc1-a9: setting vring 1 ready state 1  
2019-09-24 16:37:46,693 UVHOST: Client \_tap66e68bc1-a9: setting vring 2 ready state 1  
2019-09-24 16:37:46,693 UVHOST: Client \_tap66e68bc1-a9: setting vring 3 ready state 1  
2019-09-24 16:37:46,693 UVHOST: Client \_tap66e68bc1-a9: setting vring 4 ready state 1  
2019-09-24 16:37:46,693 UVHOST: Client \_tap66e68bc1-a9: setting vring 5 ready state 1  
2019-09-24 16:37:46,693 UVHOST: Client \_tap66e68bc1-a9: setting vring 6 ready state 1  
2019-09-24 16:37:46,693 UVHOST: Client \_tap66e68bc1-a9: setting vring 7 ready state 1  
2019-09-24 16:37:46,693 UVHOST: vr\_uvhm\_set\_vring\_enable: Can not disable TX queue 4 (only 4 queues)  
2019-09-24 16:37:46,693 UVHOST: Client \_tap66e68bc1-a9: handling message 18  
2019-09-24 16:37:46,693 UVHOST: vr\_uvhm\_set\_vring\_enable: Can not disable RX queue 4 (only 4 queues)

As there are more than four queues on the virtual machine interface, some queues must not be enabled on the virtual machine NIC. Unfortunately, these queues can ’t be disabled on the virtual machine. Therefore, this setup is faulty.

![image](data:image/emf;base64;base64,)

Figure 5.18 A faulty vRouter and VM queuing setup

# DPDK vRouter Case Studies

In previous sections, we’ve introduced some DPDK tools and explained some important log entries to help collect DPDK vRouter running status. Let’s drill down into these tools.

## Single Queue

Having understood the lcore mapping basics, let’s test some traffic.

### One way single flow: VM to fabric

To make it very simple, we are sending single uni-directional UDP flow from the PROX gen VM. You can list current flows you have in vRouter to confirm:

[root@a7s3 ~]# contrail-tools flow -l  
 Flow table(size 161218560, entries 629760)  
  
 ......  
 Index Source:Port/Destination:Port Proto(V)  
 -----------------------------------------------------------------------------------  
 40196<=>436016 192.168.0.106:59514 6 (3)  
 192.168.0.104:22  
 (Gen: 1, K(nh):27, Action:F, Flags:, TCP:SSrEEr, QOS:-1, S(nh):36, Stats:503/35823,  
 SPort 56703, TTL 0, Sinfo 8.0.0.3)  
  
 436016<=>40196 192.168.0.104:22 6 (3)  
 192.168.0.106:59514  
 (Gen: 1, K(nh):27, Action:F, Flags:, TCP:SSrEEr, QOS:-1, S(nh):27, Stats:511/71619,  
 SPort 49812, TTL 0, Sinfo 4.0.0.0)  
  
 62792<=>172020 192.168.0.106:48664 6 (3)  
 192.168.0.104:8474  
(Gen: 1, K(nh):27, Action:F, Flags:, TCP:SSrEEr, QOS:-1, S(nh):36, Stats:3828/296117,  
 SPort 63470, TTL 0, Sinfo 8.0.0.3)  
  
 172020<=>62792 192.168.0.104:8474 6 (3)  
 192.168.0.106:48664  
(Gen: 1, K(nh):27, Action:F, Flags:, TCP:SSrEEr, QOS:-1, S(nh):27, Stats:2739/274615,  
 SPort 52648, TTL 0, Sinfo 4.0.0.0)  
  
 38232<=>257372 192.168.1.105:32768 17 (2)  
 192.168.1.104:32770  
 (Gen: 5, K(nh):30, Action:F, Flags:, QOS:-1, S(nh):37, Stats:0/0, SPort 61739,  
 TTL 0, Sinfo 0.0.0.0)  
  
 257372<=>38232 192.168.1.104:32770 17 (2)  
 192.168.1.105:32768  
 (Gen: 5, K(nh):30, Action:F, Flags:, QOS:-1, S(nh):30, Stats:390003/48360372,  
 SPort 62464, TTL 0, Sinfo 3.0.0.0)

Here, you can see six vRouter flows, which are in fact three groups. The first two groups with index pairs 40196/436016 and 62792/172020, are generated by the control messages from the rapid jump VM into the PROX gen VM. The last group of flows with index pairs 38232/257372 is our single flow test traffic. The stats 39003/48360372 show the traffic flow is sent from gen VM (192.168.1.104:32770) to swap VM (192.168.1.105:32768).

In Contrail vRouter, flows are generated in pairs. For any traffic, even if it is one direction only, vRouter will generate a reverse flow for it. This is because in the real world most of the traffic is bidirectional, so having a separate entry built for each direction is required. In this case, from PROX, we are generating uni-directional traffic so only the flow of that direction has packet stats. The pairing flow entry is generated as well, but packet statistics show nothing.

Let’s clear the vif counters and collect the statistics using dpdkvifstats.py tool:

[root@a7s3 ~]# contrail-tools vif --clear  
  
Vif stats cleared successfully on all cores for all interfaces  
  
[root@a7s3 ~]# contrail-tools dpdkvifstats.py -v 3 -c 2  
 ------------------------------------------------------------------  
| Core 1 | TX pps: 0 | RX pps: 1504 | TX bps: 0 | RX bps: 90240  
| Core 2 | TX pps: 1 | RX pps: 1 | TX bps: 42 | RX bps: 56  
| Total | TX pps: 1 | RX pps: 1505 | TX bps: 336 | RX bps: 722368  
 ------------------------------------------------------------------  
  
[root@a7s3 ~]# contrail-tools dpdkvifstats.py -v 0 -c 2  
 --------------------------------------------------------------------  
| Core 1 | TX pps: 1512 | RX pps: 2 | TX bps: 166320 | RX bps: 132  
| Core 2 | TX pps: 1 | RX pps: 1 | TX bps: 112 | RX bps: 110  
| Total | TX pps: 1513 | RX pps: 3 | TX bps: 1331456 | RX bps: 1936  
 --------------------------------------------------------------------

From the first capture on the vRouter interface connecting to the PROX gen VM tap interface (-v 3), we are seeing that lcore#10 received the traffic – you can tell from the RX speed 1504 pps showing in Core 1 only. The second capture on the vRouter interface toward the bond interface (-v 0) confirms the same – it is the same lcore#10 (Core 1 here) that is sending the traffic to the bond interface, at speeds of 1512 pps, almost the same as the speed it received the traffic from the VM tap interface. The flow is illustrated here:

VM: tap41a9ab05-64 => vif0/3 => lcore#10 => vif0/0 => bond0

This seems to be weird, doesn’t it? Remember, previously based on the core-interface mapping given by dpdkinfo -c, we already know it was the lcore#11 serving our VM interface, not the other one. Accordingly, in the dpdkvifstats.py output, that should be Core 2 instead of Core 1. Let’s revisit the mapping:

[root@a7s3 ~]# contrail-tools dpdkinfo -c  
No. of forwarding lcores: 2  
No. of interfaces: 4  
Lcore 0:  
 Interface: bond0.101 Queue ID: 0  
 Interface: vhost0 Queue ID: 0  
  
Lcore 1:  
 Interface: bond0.101 Queue ID: 1  
 Interface: tap41a9ab05-64 Queue ID: 0

So we are right. The flow that is expected should be something like this:

VM: tap41a9ab05-64 => vif0/3 => lcore#11 => vif0/0 => bond0

Well, if you remember what you’ve read in Chapter 3, you probably will know the answer. When a packet flows from the PROX gen VM to the bond, vRouter uses a pipeline model to process the packet. What that really means is, the interface’s serving lcore, that is the second forwarding lcore, in our case based on dpdkinfo -c output, will poll it out of the vif interface. In Chapter 3, when we introduce the vRouter packet forwarding process, we’ve mentioned that when traffic flows from the vif connecting VM tap interface to vif0/0, all packets will be distributed by the polling lcore to other lcores for processing. The distribution is calculated based on the hash of the packet header.

Apparently, here the polling core, based on the mapping above, is lcore#11, and the only other lcore is the first forwarding lcore lcore#10. So packets from VM got polled by the lcore#11 and then distributed to the lcore#10, which then forwarded to the fabric interface vif0/0. Currently dpdkvifstats.py does not tell us much about these details, but if you collect vif output, you’ll see additional clues:

[root@a7s3 ~]# contrail-tools vif --get 3 --core 10  
Vrouter Interface Table  
  
......  
vif0/3 PMD: tap41a9ab05-64 NH: 38  
 Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.1.104  
 Vrf:2 Mcast Vrf:2 Flags:L3L2DEr QOS:-1 Ref:12  
 RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0  
 Core 10 RX packets:31272 bytes:1876320 errors:0  
 Core 10 TX packets:0 bytes:0 errors:0  
 Drops:18660668  
  
[root@a7s3 ~]# contrail-tools vif --get 3 --core 11  
Vrouter Interface Table  
  
......  
vif0/3 PMD: tap41a9ab05-64 NH: 38  
 Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.1.104  
 Vrf:2 Mcast Vrf:2 Flags:L3L2DEr QOS:-1 Ref:12  
 Core 11 RX queue packets:35384 errors:0  
 RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0  
 Core 11 RX packets:26 bytes:1092 errors:0  
 Core 11 TX packets:24 bytes:1008 errors:0  
 Drops:18660668

There is an RX queue counter, Core 11 RX queue packets:35384, that gives a little bit of a clue about this inter-core distribution. Core 11, our second forwarding lcore, polled the packet first from vif0/3 into its RX queue. Instead of processing the packet, it distributed the packets onto the first forwarding lcore, Core10, which, then processed them. That is why same amount of packets are counted as RX packets in Core 10. Therefore the full story is a flow like this:

(polling lcore) (processing lcore)  
VM: tap41a9ab05-64 => vif0/3 => lcore#11 => lcore#10 => vif0/0 => bond0

For the sake of completeness, we also captured the vif command on fabric interface vif0/0:

[root@a7s3 ~]# contrail-tools vif --get 0 --core 10  
Vrouter Interface Table  
  
......  
vif0/0 PCI: 0000:00:00.0 (Speed 20000, Duplex 1) NH: 4  
 Type:Physical HWaddr:90:e2:ba:c3:af:20 IPaddr:0.0.0.0  
 Vrf:0 Mcast Vrf:65535 Flags:TcL3L2VpVofEr QOS:-1 Ref:18  
 Core 10 RX device packets:199 bytes:49057 errors:0  
 RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0  
 Fabric Interface: eth\_bond\_bond0 Status: UP Driver: net\_bonding  
 Slave Interface(0): 0000:02:00.0 Status: UP Driver: net\_ixgbe  
 Slave Interface(1): 0000:02:00.1 Status: UP Driver: net\_ixgbe  
 Vlan Id: 101 VLAN fwd Interface: vfw  
 Core 10 RX packets:131 bytes:37595 errors:0  
 Core 10 TX packets:48756 bytes:5362888 errors:0  
 Drops:0  
 Core 10 TX device packets:49024 bytes:5730372 errors:0  
  
[root@a7s3 ~]# contrail-tools vif --get 0 --core 11  
Vrouter Interface Table  
  
......  
vif0/0 PCI: 0000:00:00.0 (Speed 20000, Duplex 1) NH: 4  
 Type:Physical HWaddr:90:e2:ba:c3:af:20 IPaddr:0.0.0.0  
 Vrf:0 Mcast Vrf:65535 Flags:TcL3L2VpVofEr QOS:-1 Ref:18  
 RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0  
 Fabric Interface: eth\_bond\_bond0 Status: UP Driver: net\_bonding  
 Slave Interface(0): 0000:02:00.0 Status: UP Driver: net\_ixgbe  
 Slave Interface(1): 0000:02:00.1 Status: UP Driver: net\_ixgbe  
 Vlan Id: 101 VLAN fwd Interface: vfw  
 Core 11 RX packets:67 bytes:9860 errors:0  
 Core 11 TX packets:181 bytes:162062 errors:0  
 Drops:0

Here after the first forwarding lcore processed the packets from vif0/0, it sent them out of vif0/0, which is reflected as TX packets and TX device packets.

One important thing that is worth to point out here is that, what we’ve tested and demonstrated here is the DPDK vRouter’s default behavior with the current parameters. Please keep in mind that vRouter is configurable. There is one vRouter configuration option introduced in release R2008 which will change this default pipeline model behavior. This option is --vr\_no\_load\_balance, and you can verify the vrouter-dpdk process running command line in your setup with ps command. With that configured, vRouter will change to the so-called run to complete model, which means that the same lcore that polled the packet will continue to process/forward it. This requires reboot of DPDK vRouter, and we’ll let you test the scenarios in your own lab.

This concludes the analysis of traffic forwarding in the direction of the VM to fabric. Next let’s take a look at the returning direction: from fabric (vif0/0) to VM (vif0/3).

### Returning Traffic: fabric to VM

Now let’s do the returning traffic. We configure the swap VM in such a way that it loops whatever it receives back to the sender. Here is the capture:

[root@a7s3 ~]# contrail-tools dpdkvifstats.py -v 0 -c 2  
 ---------------------------------------------------------------------------------  
 | Core 1 | TX pps: 85844 | RX pps: 16 | TX bps: 14936710 | RX bps: 1940 ..  
 | Core 2 | TX pps: 1 | **RX pps: 85846** | TX bps: 88 | RX bps: 14937132 ..  
 | Total | TX pps: 85845 | RX pps: 85862 | TX bps: 119494384 | RX bps: 119512576..  
 ---------------------------------------------------------------------------------

[root@a7s3 ~]# contrail-tools dpdkvifstats.py -v 3 -c 2  
 ---------------------------------------------------------------------------------  
 | Core 1 | TX pps: 0 | RX pps: 85274 | TX bps: 0 | RX bps: 10574058 ..  
 | Core 2 | **TX pps: 85278** | RX pps: 1 | TX bps: 10574431 | RX bps: 56 ..  
 | Total | TX pps: 85278 | RX pps: 85275 | TX bps: 84595448 | RX bps: 84592912 ..  
 ---------------------------------------------------------------------------------

Here, we are looking at the returning traffic from fabric back to the PROX gen VM.

Let’s focus on seeing the RX in vif 0/0 and TX in vif0/3, and the data shows lcore#11 received the packets from vif0/0 and forwarded them out of vif0/3, the forwarding path is illustrated below:

RX TX  
fabric: bond0 => vif0/0 => lcore#11 => vif0/3 => tap41a9ab05-64 => VM

To confirm if this lcore#11 that is “forwarding” packets is also the one that did the “polling”, we’ll need to look at the vif capture and looking for the “RX queue packets” as what we’ve seen before:

[root@a7s3 ~]# contrail-tools vif --get 0 --core 10  
Vrouter Interface Table  
  
......  
vif0/0 PCI: 0000:00:00.0 (Speed 20000, Duplex 1) NH: 4  
 Type:Physical HWaddr:90:e2:ba:c3:af:20 IPaddr:0.0.0.0  
 Vrf:0 Mcast Vrf:65535 Flags:TcL3L2VpVofEr QOS:-1 Ref:18  
 Core 10 RX device packets:3481584 bytes:619708685 errors:0 RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0  
 Fabric Interface: eth\_bond\_bond0 Status: UP Driver: net\_bonding  
 Slave Interface(0): 0000:02:00.0 Status: UP Driver: net\_ixgbe  
 Slave Interface(1): 0000:02:00.1 Status: UP Driver: net\_ixgbe  
 Vlan Id: 101 VLAN fwd Interface: vfw  
 Core 10 RX packets:676 bytes:106243 errors:0  
 Core 10 TX packets:3482241 bytes:605899226 errors:0  
 Drops:99  
 Core 10 TX device packets:3482474 bytes:619966089 errors:0  
  
[root@a7s3 ~]# contrail-tools vif --get 0 --core 11  
Vrouter Interface Table  
  
......  
vif0/0 PCI: 0000:00:00.0 (Speed 20000, Duplex 1) NH: 4  
 Type:Physical HWaddr:90:e2:ba:c3:af:20 IPaddr:0.0.0.0  
 Vrf:0 Mcast Vrf:65535 Flags:TcL3L2VpVofEr QOS:-1 Ref:18  
 RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0  
 Fabric Interface: eth\_bond\_bond0 Status: UP Driver: net\_bonding  
 Slave Interface(0): 0000:02:00.0 Status: UP Driver: net\_ixgbe  
 Slave Interface(1): 0000:02:00.1 Status: UP Driver: net\_ixgbe  
 Vlan Id: 101 VLAN fwd Interface: vfw  
 Core 11 RX packets:3594939 bytes:625517508 errors:0  
 Core 11 TX packets:166 bytes:133391 errors:0  
 Drops:99

There aren’t any “RX queue packets” counter like the one we saw in the data collected on the VM to fabric direction. Therefore in this direction of traffic sent from fabric to VM, we don’t see any “inter-core load balancing” behavior as we’ve elaborated before.

You may notice that there is a “Core 10 RX device packets” and a “Core 10 TX device packets” counter displayed on lcore#10 only. As the time of writing of this book, These counter shows the total packets received and sent by the NIC, so they have nothing to do with the inter-core load balancing behavior.

The reason is in this test vRouter is using VxLAN encapsulations for the fabric interface, which follows the run-to-completion model explained in Chapter 3, therefore, one forwarding lcore polls for packets, makes forwarding decision, and forwards the packet out of the other interface. This “rule” applies to the traffic flows from physical NIC toward the VM, regardless of number of lcores enabled. In this below test, we have enabled 4 lcores, and this conclusion remains:

[root@a7s3 ~]# contrail-tools dpdkvifstats.py -all -c 4

| VIF 3 |Core 1 | TX pps: 0 | RX pps: 0 | TX bps: 0 | RX bps: 0 | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |

| VIF 3 |Core 2 | TX pps: 1 | RX pps: 1 | TX bps: 448 | RX bps: 448 | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |

| VIF 3 |Core 3 | TX pps: 0 | RX pps: 75778 | TX bps: 0 | RX bps: 36373760 | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |

| VIF 3 |Core 4 | **TX pps: 75776** | RX pps: 0 | TX bps: 36372480 | RX bps: 0 | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |

| VIF 0 |Core 1 | TX pps: 10 | RX pps: 1 | TX bps: 12424 | RX bps: 448 | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |

| VIF 0 |Core 2 | TX pps: 0 | RX pps: 0 | TX bps: 0 | RX bps: 176 | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |

| VIF 0 |Core 3 | TX pps: 76810 | RX pps: 1 | TX bps: 67593384 | RX bps: 816 | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |

| VIF 0 |Core 4 | TX pps: 1 | **RX pps: 76839** | TX bps: 912 | RX bps: 67619992 | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |

------------------------------------------------------------------------

| pps per Core |

------------------------------------------------------------------------

|Core 1 |TX + RX pps: 19 | TX pps 10 | RX pps 9 |

|Core 2 |TX + RX pps: 2 | TX pps 1 | RX pps 1 |

|Core 3 |TX + RX pps: 152589 | TX pps 76810 | RX pps 75779 |

|Core 4 |TX + RX pps: 152627 | TX pps 75788 | RX pps 76839 |

------------------------------------------------------------------------

|Total |TX + RX pps: 305237 | TX pps 152609 | RX pps 152628 |

------------------------------------------------------------------------

This concludes our analysis to the bidirectional single flow traffic. As you can see, one benefit of having traffic generator/swapper built into lab environment is that you can fine tune the generator to send traffic in a very specific pattern, so that you can take a deep look at the counters and analyze the vRouter traffic forwarding behavior. This is very helpful for learning purpose, but in production, you probably never expect to have such a luxury since the traffic pattern in the field is usually much more complex. But don’t worry, you can add more and more complexities to our traffic pattern so eventually you will see something close to what you would see in real life.

Next, let’s add more flows in our testbed and check out the results.

### Multiple Flows

Now we are sending 64 flows from PROX gen VM. To confirm the flow numbers let’s use the flow -s command in contrail-tools:

[root@a7s3 ~]# contrail-tools flow -s  
 Flow Statistics  
 ---------------  
 Total Entries --- Total = 132, new = 0  
 Active Entries --- Total = 132, new = 0  
 Hold Entries --- Total = 0, new = 0  
 **Fwd flow Entries - Total = 132** drop flow Entries - Total = 0  
 NAT flow Entries - Total = 0  
  
 Rate of change of Active Entries  
 --------------------------------  
 current rate = 0  
 Avg setup rate = 0  
 Avg teardown rate = 0  
 Rate of change of Flow Entries  
 ------------------------------  
 current rate = 0

You can see 132 flows entries means 66 groups of flows in our test. The additional two groups of flows are the control flows between the jump VM and gen VM. Good, let’s collect the traffic statistics:

[root@a7s3 ~]# contrail-tools vif --clear  
  
 Vif stats cleared successfully on all cores for all interfaces  
  
 [root@a7s3 ~]# contrail-tools dpdkvifstats.py -all -c 2  
| **VIF 3** | **Core 1** | **TX pps: 1** | **RX pps: 85248** | **TX bps: 448** | **RX bps: 84566016** | VIF 3 | Core 2 | TX pps: 1 | RX pps: 1 | TX bps: 336 | RX bps: 560  
| **VIF 0** | **Core 1** | **TX pps: 85842** | **RX pps: 15** | **TX bps: 119490528** | **RX bps: 14744** | VIF 0 | Core 2 | TX pps: 0 | RX pps: 0 | TX bps: 0 | RX bps: 0  
 ------------------------------------------------------------------------  
 | pps per Core |  
 ------------------------------------------------------------------------  
 |Core 1 |TX + RX pps: 171133 | TX pps 85858 | RX pps 85275 |  
 |Core 2 |TX + RX pps: 2 | TX pps 1 | RX pps 1 |  
 ------------------------------------------------------------------------  
 |Total |TX + RX pps: 171135 | TX pps 85859 | RX pps 85276 |  
 ------------------------------------------------------------------------

Still, the lcore#10 processed the packets and forwarded them out of vif0/0. If you compare this result with our first test, where we have just one uni-directional flow, there is simply no difference. Shouldn’t we expect to see some load balance between lcores since we have more flows now? We should, but that is only when the VM tap interface has multiple queues. With just one queue, the mapping between our tap interface and lcores never changes. In this case it’s always lcore#11 polling the traffic and distributing to lcore#10, hence we’ll always see packet being forwarded by lcore#10 instead of lcore#11, regardless of number of flows and traffic volumes.

On the other direction, if we enable the returning traffic, we’ll see on VIF  
0 (vif0/0) the two lcores’ traffic are RX pps: 41547, and RX pps: 44257, which is well balanced because we have two queues enabled on the vif0/0:

[root@a7s3 ~]# contrail-tools dpdkvifstats.py -all -c 2  
 | VIF 3 | Core 1 | TX pps: 41249 | RX pps: 85182 | TX bps: 40919336 | RX bps: 84500544  
 | VIF 3 | Core 2 | TX pps: 43936 | RX pps: 1 | TX bps: 43584072 | RX bps: 336  
 | VIF 0 | Core 1 | TX pps: 85765 | **RX pps: 41547** | TX bps: 119382912 | RX bps: 57825008  
 | VIF 0 | Core 2 | TX pps: 3 | **RX pps: 44257** | TX bps: 18216 | RX bps: 61604304  
 ------------------------------------------------------------------------  
 | pps per Core |  
 ------------------------------------------------------------------------  
 |Core 1 |TX + RX pps: 253763 | TX pps 127025 | RX pps 126738 |  
 |Core 2 |TX + RX pps: 88197 | TX pps 43939 | RX pps 44258 |  
 ------------------------------------------------------------------------  
 |Total |TX + RX pps: 341960 | TX pps 170964 | RX pps 170996 |  
 ------------------------------------------------------------------------

With a single queue in the VM tap interface, it’s hard to achieve good load balance between lcores on the vRouter interface facing the VM. Sometimes you need to enable multiple queues to make better use of all your DPDK forwarding lcores.

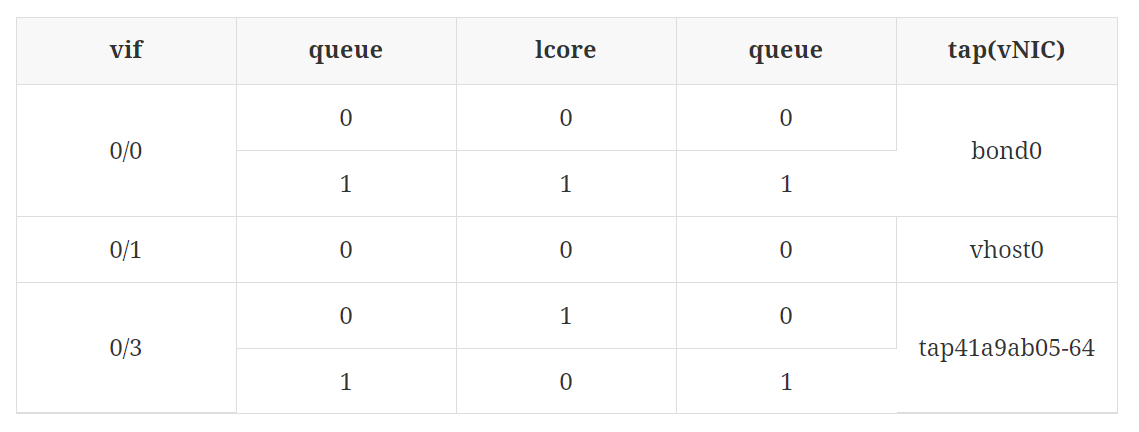
This concludes our analysis on one single queue test, and we’ll go ahead to test multiple queues.

## Multiple Queues

Finally, let’s look at a multiple queue example. Based on the previous setup, this time we added one more queue in the tap interface of VM gen and then collect the core interface mapping:

[root@a7s3 ~]# contrail-tools dpdkinfo -c  
No. of forwarding lcores: 2  
No. of interfaces: 5  
Lcore 0:  
 Interface: bond0.101 Queue ID: 0  
 Interface: vhost0 Queue ID: 0  
 Interface: tap41a9ab05-64 Queue ID: 1  
  
Lcore 1:  
 Interface: bond0.101 Queue ID: 1  
 Interface: tap41a9ab05-64 Queue ID: 0

Here is the table view of these mappings:



So most items remain the same, except we have one more queue added on tap and the vRouter interface to which it attaches, correspondingly, one core is allocated to serve this new queue. Before this new queue was created we already knew that each of our lcores is serving the same amount of queues, therefore as a tie breaker, which we’ve mentioned when we introduced dpdkinfo -c previously, the first forwarding lcore, lcore#10 with our notation, is allocated for the new queue.

Let’s check the traffic distribution between lcores with multiple queues on VM tap interface:

[root@a7s3 ~]# contrail-tools dpdkvifstats.py -all -c 2  
 | VIF 3 | Core 1 | TX pps: 41319 | RX pps: 42606 | TX bps: 40988672 | RX bps: 42264712  
 | VIF 3 | Core 2 | TX pps: 43889 | RX pps: 42604 | TX bps: 43537008 | RX bps: 42262288  
 | VIF 0 | Core 1 | TX pps: 42923 | RX pps: 41540 | TX bps: 59748824 | RX bps: 57815160  
 | VIF 0 | Core 2 | TX pps: 42918 | RX pps: 44320 | TX bps: 59741640 | RX bps: 61693328  
 ------------------------------------------------------------------------  
 | pps per Core |  
 ------------------------------------------------------------------------  
 |Core 1 |TX + RX pps: 168416 | TX pps 84258 | RX pps 84158 |  
 |Core 2 |TX + RX pps: 173731 | TX pps 86807 | RX pps 86924 |  
 ------------------------------------------------------------------------  
 |Total |TX + RX pps: 342147 | TX pps 171065 | RX pps 171082 |  
 ------------------------------------------------------------------------

Now we have multiple queues on both the VM tap interface and the fabric interface. Traffic on all lcores are well balanced. Please keep this in mind as an ideal traffic profile that we are expecting the vRouter to have. In production, we usually deal with more complicated vRouter lcore configurations and traffic profiles, so the lcore balancing may be appear as perfect as what we are seeing in lab environment, but at least you have a good baseline in your mind and know what to look for when the result is far worse than expected.