chapter 6

contrail networking and test tools installation

2020-10-19

In previous chapters, we have gone through most important topics about SDN and DPDK in general, DPDK vRouter architectures and packet processing details and so on. When you read these topics you may wonder how to get a running setup with DPDK environment in it, so you can play around, test those theories and familiarize yourself about you’ve learned. Indeed, these topics are important, but unfortunately they are by themselves not so straightforward, so even after we’ve put great effort to illustrate, some of them may still sound confusing, especially when you get down to the implementation details.

In this chapter, we will mostly focus on the hands-ons and lab testings to familiarize ourselves with these concept and theories.

* We’ll start from introducing steps we’ve used to install a latest version of contrail networking cluster.
* On top of it we give details of building a testing environment. Basically that includes a few VMs running OPENFV PROX software, and on each VM based on its role the PROX software is configured as a traffic generator or a a traffic receiver.
* We’ll go ahead to introduce some of the commonly used DPDK command line tools/scripts that prints useful information to help us understand what we’ve learned so far.
* Then we’ll explore the DPDK log file and analyze some interesting log entries.
* At last, we’ll do some case studies. We start some traffic in our setup using the testing tools, and use DPDK command line tools to analyze the vRouter running status.

After reading this chapter, you will have a deeper and concrete understanding to the main concepts we’ve covered in this book. We’ll start with the contrail installation.

# contrail installation

In this book, we’ve been focusing on DPDK vRouter that runs in each individual compute node, which basically runs in a relatively standalone mode. If you look at forwarding plane as a whole, they are a distributed system. Actually as we’ve briefed in chapter 1, contrail is a complex distributed system involving a lot more different software modules especially in control plane. Again, each of the software module can be a completely different distributed system by themselves. Explaining and understanding details about how things works in distributed system is never easy, and so is the installation process. It won’t be a surprise if you run into some installation issues in your lab. Generally speaking, it is always much more efficient to follow a detailed, verfied process with step by step instructions to "avoid the issues", than just starting with a "try-and-see" mode.

There are many different way of installing contrail system. In this section, we’ll give a detail steps about installing contrail with openstack and kolla ansible.

Kolla is an OpenStack project which provides tools to build container images for OpenStack services. Kolla Ansible provides Ansible playbooks to deploy the Kolla images. The contrail-kolla-ansible playbook works in conjunction with contrail-ansible-deployer to install OpenStack and Contrail Networking containers.

TODO: add brief introduction to each steps, add comment to the yaml file line by line. etc

## cluster diagram

ADD digram with setup details here.

## re-image servers

## configure bond and vlan

scp a7s2/\* root@a7s2:/etc/sysconfig/network-scripts/  
scp a7s3/\* root@a7s3:/etc/sysconfig/network-scripts/  
scp a7s4/\* root@a7s4:/etc/sysconfig/network-scripts/  
scp a7s5/\* root@a7s5:/etc/sysconfig/network-scripts/

service network restart

Once the restart is successful, the following IP addresses should be pingable: 8.0.0.1, 8.0.0.2, 8.0.0.3, 8.0.0.4

## instances.yaml (kiran)

global\_configuration:  
CONTAINER\_REGISTRY: svl-artifactory.juniper.net/contrail-nightly  
REGISTRY\_PRIVATE\_INSECURE: True  
provider\_config:  
bms:  
 ssh\_pwd: c0ntrail123  
 ssh\_user: root  
 ntpserver: 10.84.5.100  
 domainsuffix: englab.juniper.net  
instances:  
a7s2:  
 provider: bms  
 ip: 10.84.27.2  
 roles:  
 openstack\_control:  
 openstack\_network:  
 openstack\_storage:  
 openstack\_monitoring:  
 config\_database:  
 config:  
 control:  
 analytics\_database:  
 analytics:  
 webui:  
a7s3:  
 provider: bms  
 ip: 10.84.27.3  
 ssh\_user: root  
 ssh\_pwd: c0ntrail123  
 roles:  
 openstack\_compute:  
 vrouter:  
 PHYSICAL\_INTERFACE: bond0.101  
 CPU\_CORE\_MASK: "0x1fe"  
 DPDK\_UIO\_DRIVER: uio\_pci\_generic  
 HUGE\_PAGES: 32000  
 AGENT\_MODE: dpdk  
a7s4:  
 provider: bms  
 ip: 10.84.27.4  
 ssh\_user: root  
 ssh\_pwd: c0ntrail123  
 roles:  
 openstack\_compute:  
 vrouter:  
 PHYSICAL\_INTERFACE: bond0.101  
 CPU\_CORE\_MASK: "0x1fe"  
 DPDK\_UIO\_DRIVER: uio\_pci\_generic  
 HUGE\_PAGES: 32000  
 AGENT\_MODE: dpdk  
a7s5:  
 provider: bms  
 ip: 10.84.27.5  
 ssh\_user: root  
 ssh\_pwd: c0ntrail123  
 roles:  
 openstack\_compute:  
 vrouter:  
 PHYSICAL\_INTERFACE: bond0.101  
contrail\_configuration:  
CONTRAIL\_VERSION: master.1209  
OPENSTACK\_VERSION: queens  
CLOUD\_ORCHESTRATOR: openstack  
CONTROLLER\_NODES: 8.0.0.1  
OPENSTACK\_NODES: 8.0.0.1  
CONTROL\_NODES: 8.0.0.1  
KEYSTONE\_AUTH\_HOST: 8.0.0.200  
KEYSTONE\_AUTH\_ADMIN\_PASSWORD: c0ntrail123  
RABBITMQ\_NODE\_PORT: 5673  
KEYSTONE\_AUTH\_URL\_VERSION: /v3  
IPFABRIC\_SERVICE\_IP: 8.0.0.200  
VROUTER\_GATEWAY: 8.0.0.254  
two\_interface: true  
ENCAP\_PRIORITY: VXLAN,MPLSoUDP,MPLSoGRE  
AUTH\_MODE: keystone  
CONFIG\_API\_VIP: 10.84.27.51  
ssh\_user: root  
ssh\_pwd: c0ntrail123  
METADATA\_PROXY\_SECRET: c0ntrail123  
CONFIG\_NODEMGR\_\_DEFAULTS\_\_minimum\_diskGB: 2  
CONFIG\_DATABASE\_NODEMGR\_\_DEFAULTS\_\_minimum\_diskGB: 2  
DATABASE\_NODEMGR\_\_DEFAULTS\_\_minimum\_diskGB: 2  
XMPP\_SSL\_ENABLE: no  
LOG\_LEVEL: SYS\_DEBUG  
AAA\_MODE: rbac  
kolla\_config:  
kolla\_globals:  
 kolla\_internal\_vip\_address: 8.0.0.200  
 kolla\_external\_vip\_address: 10.84.27.51  
 contrail\_api\_interface\_address: 8.0.0.1  
 keepalived\_virtual\_router\_id: "111"  
 enable\_haproxy: "yes"  
 enable\_ironic: "no"  
 enable\_swift: "no"  
kolla\_passwords:  
 keystone\_admin\_password: c0ntrail123  
 metadata\_secret: c0ntrail123  
 keystone\_admin\_password: c0ntrail123

## installation steps

1. install pre-requisite packages on a7s2

* yum -y remove python-netaddr  
  yum -y install epel-release python-pip gcc python-cffi python-devel bcrypt==3.1.7 sshpass python-wheel  
  pip install wheel requests  
  yum -y install git  
  pip install ansible==2.5.2.0

1. install ansible deployer

* git clone http://github.com/tungstenfabric/tf-ansible-deployer  
  cd tf-ansible-deployer

1. edit version and copy instances.yaml to /root/tf-ansible-deployer/config (Ref for version: <https://svl-artifactory.juniper.net/artifactory/contrail-nightly/contrail-vrouter-agent-dpdk/>)
2. install contrail

* ansible-playbook -i inventory/ -e orchestrator=openstack playbooks/configure\_instances.yml  
  ansible-playbook -i inventory/ playbooks/install\_openstack.yml  
  ansible-playbook -i inventory/ -e orchestrator=openstack playbooks/install\_contrail.yml

1. install openstack client

* pip install --ignore-installed python-openstackclient python-ironicclient openstack-heat  
  source /etc/kolla/kolla-toolbox/admin-openrc.sh

TODO: polish this paragraph to introduct next topic. Now you have an up and running DPDK setup. You can login to the setup from webUI to confirm everything is working fine.

GIVE webUI capture here.

You can also login to each individual nodes and run contrail-status to verify the running status of each components of it. If everything works, congratulations! You now have your own lab to exercise lots of things we’ve been introducing in this book. Next, we’ll go over the steps of setting up testing tools to send and receive traffic - the PROX and rapid script.

# dpdk vRouter test tools: prox and rapid

## introduction

**PROX.**

PROX (Packet pROcessing eXecution Engine) is an OPNFV project application built on top of DPDK. It is capable of performing various operations on packets in a highly configurable manner. It also support performance statistics that can be used for performance investigations. Because of the rich feature set it supports, it can be used to create flexible software architectures through small and readable configuration files. In this chapter we’ll introduce how to use it to test vrouter performance in DPDK environment.

In a typical test you need two VMs running PROX. VM1 is generating packets, sending them to VM2 which will perform a "swap" operation on all packets, so that they are sent back to VM1.

* "traffic generator" VM ("gen" VM)
* "traffic receiver and looping VM" VM ("swap" VM, or "loop" VM)

In this book we will call them "gen" and "swap" VM respectively. One special feature we used here is that, the "swap" PROX is configured in such a way that, once receives the packets sent from the generator, it will "swap", or "loop" them back to the generator VM, so the latter can collect them and calculate how much traffic got forwarded by the DUT - in our case it is the DPDK vRouter.

**rapid.**

Rapid(Rapid Automated Performance Indication for Dataplane) is a groups of "wrapper" scripts interacting with PROX to simplify and automate the configuration of PROX. It is a set of files and scripts offering an even easier way to do a sanity check of the dataplane performance.

rapid is very powerful and configurable. A typical workflow is like below:

* A script name runrapid.py will send the proper configuration files to the gen and swap VMs involved in the testing, so each one will knows its role ("generator" or "swapper") in the test.
* It then starts PROX within both VMs, as generator and swapper respectively.
* While the test is ongoing it collects the results from PROX. Results are printed on the screen and logged in the log and csv files.
* The same tests will be done for different packet sizes and/or different amounts of flows.

The rapid scripts are typically installed in a third VM, called "jump" VM in this book. The purpose of this VM is to control the traffic generator to start, stop, pause the test as well as collecting the statistics.

**PROX and rapid test setup.**

A typical prox and rapid testing setup looks like this:

![testing diagram](data:image/png;base64;base64,)

testing diagram

The test setup consists of three compute nodes, running the above mentioned 3 VMs respectively:

* "PROX generate VM" runs on compute-A: This is the "traffic generator" VM for traffic generation
* "PROX looping VM" runs on compute-B: This is the "swap" VM for looping traffic out of the same interface where it came in. This is the DUT (device under test) where the vRouter is running.
* "rapid jump VM" runs on compute-C: This is the VM where rapid scripts are installed, it is responsible for control traffic genaration and collecting results

**Hardware requirements.**

Here is a brief summary of hardware requirements for different VM:

* swap VM: this is where the DUT (vRouter) is located. Based on the test requirement a specific amount of hardware resources should be allocated and all applications that could unnecessarily consume the hardware resources should be removed.
* gen VM: In order to saturate the DUT, the traffic generator VM and the compute should be allocated much more CPU resources than the DUT.
* Jump VM: no high speed VM is required, can be run on kernel or DPDK compute)
* Optionally, the generator and receiver computes can run on a bonded interface configured with 802.3ad LACP mode. This is a common configuration recommended in practical environment.

By default, multi-queue is enabled on both Prox gen and swap VMs via openstack flavor. You can refer to chapter 3 for more details about "multi-queue" feature and its configurations. Additionally, Rapid scripts also provides CPU pining to protect PROX PMDs against CPU stealing by other processes and the VM Operating System.

## installation: manual steps

**creating openstack resources.**

As mentioned earlier, to perform the test we need two VM both running PROX. One sending traffic and the other one receive and swap it back. Same exact PROX application is running but with different configuration files.

Apparently, the IP level connectivity is required in order for the two VM to be able to exchange packets with each other. In our case, the two VM will be spawned by openstack nova. Needless to say, all supporting objects and resources associcated to the VMs, like IPAM, subnet, virtual-network and VM flavor (size of CPU/memory/storage/etc), also need to be created out of openstack infrastructure, either from horizon webUI or openstack CLIes. A quick list of the common tasks are listed here:

* create IPAMs/subnets/virtual networks
* create flavors
* create images
* create host aggregates
* create instances
* create key-pairs

On top of these, installing PROX inside of the VMs, like with many other open source projects, often requires downloading the source code and compile it in your platform. That means you download the PROX source codes, compile it to get the execute, then configure and run the application. In this section we’ll introduce how PROX is installed in our setup we built for this book, You can find more details in PROX website here: <https://wiki.opnfv.org/display/SAM/PROX+installation>

The software and CPU model we use here are shown below:

[root@a7s3 ~]# cat /etc/centos-release  
CentOS Linux release 7.7.1908 (Core)

[root@a7s3 ~]# uname -a  
Linux a7s3 3.10.0-1062.el7.x86\_64 #1 SMP Wed Aug 7 18:08:02 UTC 2019 x86\_64 x86\_64 x86\_64 GNU/Linux

[root@a7s3 ~]# lscpu | grep Model  
Model: 62  
Model name: Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz

In our lab setup the VM OS is the same as the host, and the emulated CPU Model is Intel Xeon E3-12xx:

[root@stack2-gen ~]# cat /etc/centos-release  
CentOS Linux release 7.7.1908 (Core)

[root@stack2-gen ~]# uname -a  
Linux stack2-gen.novalocal 3.10.0-1062.18.1.el7.x86\_64 #1 SMP Tue Mar 17 23:49:17 UTC 2020 x86\_64 x86\_64 x86\_64 GNU/Linux

[root@stack2-gen ~]# lscpu | grep -i Model  
Model: 58  
Model name: Intel Xeon E3-12xx v2 (Ivy Bridge, IBRS)

There is a good chance that your servers and VM may have totally different hardware and software architectures. The steps below are tested and working fine in our setup, but depending on your environment it may works just fine or run into some errors. Check PROX online document for more detailed instructions.

**Compiling and building DPDK.**

PROX is a dpdk application. When running, it connects to the DPDK libraries to implement most of its features. Therefore to build it we need a DPDK environment.

You can either build it inside of the VM where you want to run it, or build it directly in the host environment where the VM got spawned and copy it into the VM.

The steps to build DPDK in our setup is as below:

Add the following to the end of ~/.bashrc file

sudo yum install numactl-devel net-tools wget gcc unzip libpcap-devel \  
 ncurses-devel libedit-devel pciutils lua-devel kernel-devel  
  
export RTE\_SDK=/root/dpdk  
export RTE\_TARGET=x86\_64-native-linuxapp-gcc  
export RTE\_KERNELDIR=/lib/modules/`ls /lib/modules`/build  
export RTE\_UNBIND=$RTE\_SDK/tools/dpdk\_nic\_bind.py  
#Re-login or source that file  
. ~/.bashrc  
#Build DPDK  
git clone https://github.com/DPDK/dpdk  
cd dpdk  
git checkout v19.11  
make install T=$RTE\_TARGET

**Compiling PROX.**

Now with DPDK libraries built, we can start to download, extract and build the PROX application. Here are the steps:

git clone https://github.com/opnfv/samplevnf  
cd samplevnf/VNFs/DPPD-PROX  
git checkout origin/master  
make

When make succeeds, the compiled binary PROX will be available in build folder of current directory.

We’ll demonstrate this later.

**configuration files.**

The set of sample configuration files can be found in: ./config folder. Sample configs of PROX functioning as the "generator" is available in ./gen/ folder.

Assuming the current directory is where you’ve just built PROX, we can just launch PROX with a proper configuration file.

./build/prox -f <prox configuration file>

When it runs, a ncurse based UI will pop up and through it you will see update about the running states in real time. We’ll give an example on this later.

**Rapid installation.**

Rapid scripts can be downloaded from here: <https://github.com/opnfv/samplevnf/tree/master/VNFs/DPPD-PROX/helper-scripts/rapid> The scripts were developed in python, so you can run them directly and no need to compile.

## installation: heat automation

We have just introduced the steps of manually compiling PROX from source code. We also has assumed you know how to perform a list of tasks to create all necessary objects required by the VMs from openstack. Doing this one time is not a big deal. Suppose you are working in a dynamic environment where you often need to:

* quickly build up a PROX test environment to do some tests
* tear it down after the test is finished
* redo the same test all over again in another cluster

Repeating these manual steps will become a tedious and even painful job. You will soon prefer to be able to simplify the building, creation and configuration of PROX, as well as creating all necessary openstack resources. In openstack environment the NO. 1 choice for automation is heat. With heat, typically all tasks are programmed in a template file, with calls all parameters from another environment file. In appendix, we provide all sample template file as long as environment file and associcated scripts, which are tested and proved to be working fine in our setup. You can use them as a starting point, then make necessary customizations based on your environment to build your owen automation. The virtual machine, where the tools are running, including rapid scripts and PROX DPDK application that is pre-compiled in it, has also been built as an image . With all these automations carefully designed and tested, all what we need to do now becomes much simpler:

* download this pre-built image and load it into openstack image service
* create the heat stack with the sample template files

If everything goes well, you will have your whole PROX testing environment available in just a few minutes. The detail steps are listed below:

1. Prepare pre-built VM image, heat template files and scripts
   * VM image: this is the image with PROX compiled, as shown in previous section.
   * heat template: see appendix
2. load rapid image into opentack glance service

* openstack image create --disk-format qcow2 --container-format bare --public --file rapidVM.qcow2 rapidVM-1908  
  openstack image set --property hw\_vif\_multiqueue\_enabled="true" rapidVM-1908

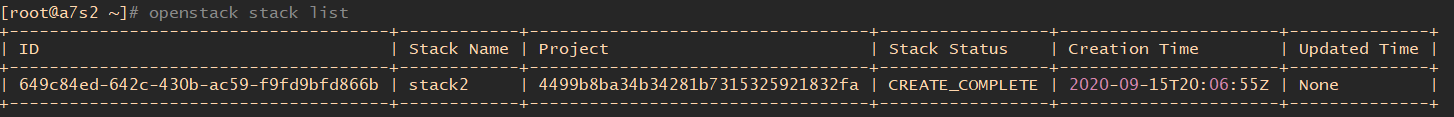
1. (Optionally) if you’re using ceph backend:

* qemu-img convert rapidVM-1908.qcow2 rapidVM-1908.raw  
  openstack image create --disk-format raw --container-format bare --public --file rapidVM.raw rapidVM-1908  
  openstack image set --property hw\_vif\_multiqueue\_enabled="true" rapidVM-1908

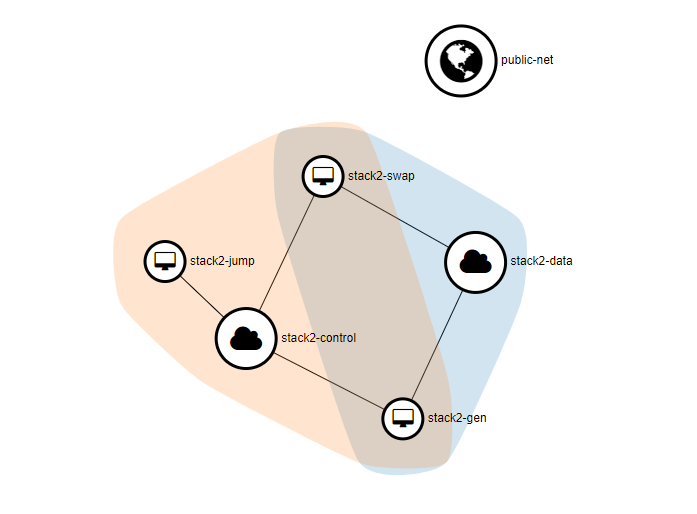
1. adjust the heat template files based on your environment
   * environment.yaml
   * build-rapid.yml
   * configure.rapid.sh
2. create heat stack:

* openstack stack create -t build-rapid.yml -e environment.yaml stack2

Wait for a few minutes and use openstack stack list command to check the stack creation status.



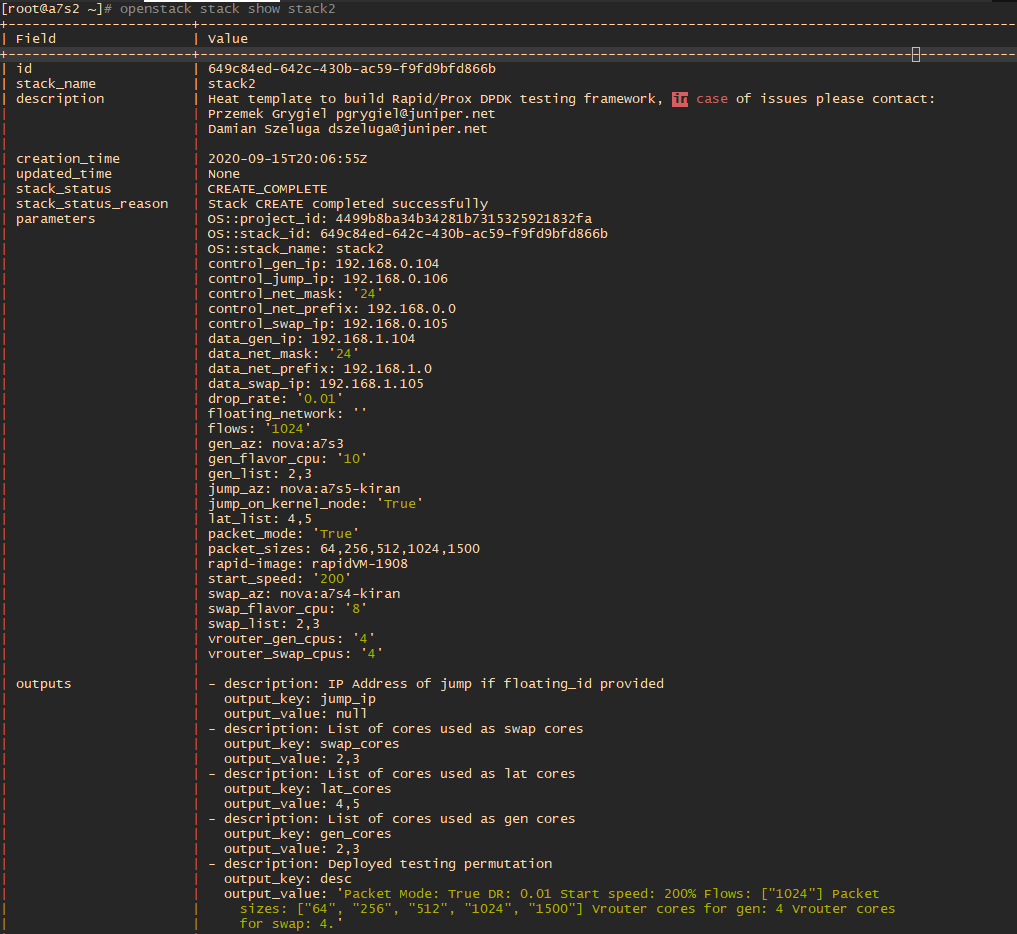
openstack stack list



openstack topology (graph)

Once succeeded, you can use different sub-command of openstack stack command to retrieve the parameters of the stack components.

openstack stack list STACK  
openstack stack resource list  
openstack stack resource list --filter type=OS::Nova::Server  
openstack stack show STACK  
openstack stack output show STACK



openstack stack show STACK

**login to the VMs.**

The image has been configured with a root password Login c0ntrail123. So all 3 VMs, once up and running, will inheritage the same login credential. In contrail/openstack integration environment There are a few common ways to access a VM running in a specific compute node:

* floating IP: This is an routable IP address that is visible from outside of the cluster which maps to an internal IP of the VM. Once VM is launched, you can login to a specific VM with this IP address from anywhere that is able to reach the IP.
* virsh console: virsh provides access to the VM console. This does not require any IP address to be configured.
* meta\_ip\_address: This is a non-routable private IP that visible only from a specific compute. This IP address is automatically generated and mapped to the VM’s tap interface IP.

In our test we didn’t configure any floating IP, so we will use console and meta\_ip\_address to access the VM. To access VM console use virsh console command from nova\_libvirt docker in the compute node:

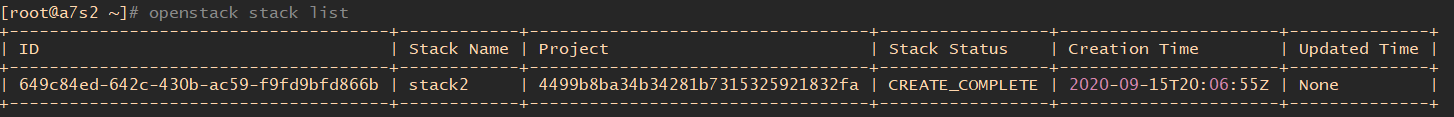
[root@a7s3 ~]# docker exec -it nova\_libvirt virsh list  
 Id Name State  
----------------------------------------------------  
 2 instance-00000041 running

[root@a7s3 ~]# docker exec -it nova\_libvirt virsh console 2  
Connected to domain instance-00000041  
Escape character is ^]

CentOS Linux 7 (Core)  
Kernel 3.10.0-1062.18.1.el7.x86\_64 on an x86\_64

stack2-gen login: root  
Password:  
Last login: Fri Sep 25 17:31:21 from 192.168.0.2  
[root@stack2-gen ~]#

Comparing with console, ssh session is usually preferred. Let’s take a look at each VM’s allocated interface IPs with openstack server list command:



openstack server list

let’s take our "jump" VM stack2-jump for instance. Openstack allocated an IP address 192.168.0.106 to it’s tap interface from the stack2-control virtual-network. However, this IP address is not directly reachable from the host. In order to ssh into the VM, we need to first locate the meta\_ip\_address allocated to the VM’s tap interface, or more specifically, the vif interface in vRouter. We can use vRouter vif command to confirm which vif interface has this IP.

[root@a7s5-kiran ~]# contrail-tools vif -l | grep -B2 -A6 192.168.0.106  
  
vif0/3 OS: tap0160123b-14 NH: 28  
 Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.0.106  
 Vrf:2 Mcast Vrf:2 Flags:PL3L2DEr QOS:-1 Ref:6  
 RX packets:47246 bytes:2362255 errors:0  
 TX packets:42996 bytes:2133684 errors:0  
 ISID: 0 Bmac: 02:01:60:12:3b:14  
 Drops:3553

Good. vif0/3 has the IP, so this vif connects to the tap interface of our jump VM. In contrail vRouter, for each vif there is also a "hidden" meta\_data\_ip of "169.254.0.N", wherre N is the same number as in the vif0/N. Therefore in this case our meta\_data\_ip is "169.254.0.3". Let’s try to start a ssh session into it:

[root@a7s5-kiran ~]# ssh 169.254.0.3  
Password:  
Last login: Wed Sep 23 11:13:58 2020  
[root@stack2-jump ~]#

It works. The benefit of this approach is that, not only the interaction with the VM is much faster, but also it supports file copies with scp tool. Remember in many cases the VM does not has any Internet connection, so in case you need to copy files into (or out of) the VM, the meta\_data\_ip method will be especially useful.

## run rapid automation: runrapid.py

With the stack created and all VMs up and running, we now can introduce how to run test with rapid. Remember rapid is installed in the "jump" VM, so we’ll need to execute the script from there.

On Jump VM, go to /root/prox/helper-scripts/rapid/ folder, where you can locate a python script named "runrapid.py". To run test you can just run it without any other parameters:

cd /root/prox/helper-scripts/rapid/  
./runrapid.py

This will start rapid script and send traffic for 10 seconds by default. the period of time for sending traffic can be adjusted by --runtime option:

cd /root/prox/helper-scripts/rapid/  
./runrapid.py --runtime <time> # replace <time> with time per one execution in seconds

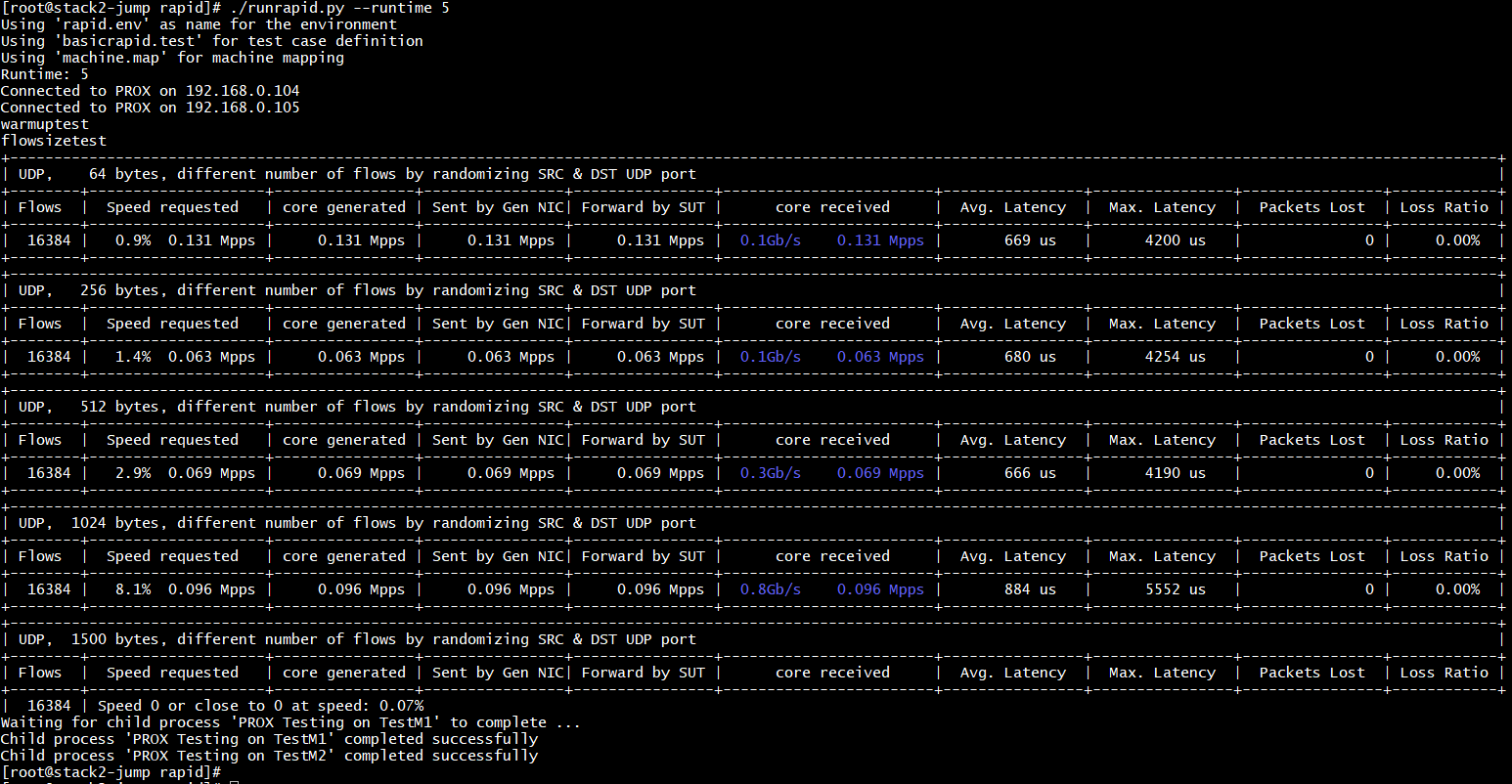
A few other command line options are supported, which can be listed by -h:

[root@stack2-jump rapid]# ./runrapid.py -h  
usage: runrapid [--version] [-v]  
 [--env ENVIRONMENT\_NAME]  
 [--test TEST\_NAME]  
 [--map MACHINE\_MAP\_FILE]  
 [--runtime TIME\_FOR\_TEST]  
 [--configonly False|True]  
 [--log DEBUG|INFO|WARNING|ERROR|CRITICAL]  
 [-h] [--help]

Command-line interface to runrapid

optional arguments:  
 -v, --version Show program's version number and exit  
 --env ENVIRONMENT\_NAME Parameters will be read from ENVIRONMENT\_NAME. Default is rapid.env.  
 --test TEST\_NAME Test cases will be read from TEST\_NAME. Default is basicrapid.test.  
 --map MACHINE\_MAP\_FILE Machine mapping will be read from MACHINE\_MAP\_FILE. Default is machine.map.  
 --runtime Specify time in seconds for 1 test run  
 --configonly If this option is specified, only upload all config files to the VMs, do not run the tests  
 --log Specify logging level for log file output, default is DEBUG  
 --screenlog Specify logging level for screen output, default is INFO  
 -h, --help Show help message and exit.

A typical runrapid.py script execution looks like this:



runrapid.py script

You can see that some preparation work were done before the actual test are started: . First, the script read 3 files, rapid.env, basicrapid.test and machine.map. The env file provides IP/MAC information of the gen and swap VM, and the .test file defines all detail behavior of the test.

1. Then, the script connects to both gen and swap VM.
2. The script start some small amount of traffic as "warmup". This is to test The reachability between the source and destination, and also populate MAC table or ARP table in devices along the path.
3. When everything is ready, the script starts the traffic in certain speed and at the same time monitor the traffic receiving rate in real time. Any packet drop rate higher than the defined threshold indicates the current traffic rate is too high to the DUT, so it will drop the rate in the next iteration. By binary search, eventually, it finds the maximum throughput between 2 systems within a given allowed packet loss and accuracy which are defined in the \*.test files (e.g. the basicrapid.test file for a simple test)

The script is highly configurable. In appendix We provide a sample "basicrapid.test" that we use in our lab. You can start with it and fine tune based on your need. For example, in section [test2] of the file you can change number of flow and packet size to define different test scenarios.

[test2]  
test=flowsizetest  
packetsizes=[64,256,512,1024,1500]  
# the number of flows in the list need to be powers of 2, max 2^20  
# Select from following numbers: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65535, 131072, 262144, 524280, 1048576  
flows=[16384, 65535]

## run PROX manually

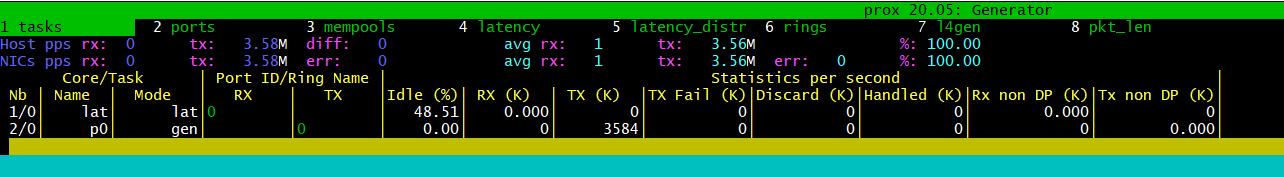
OK. We just introduced rapid. The script support very extensive options in the configuration files which beyond the scope of this book, but we’ve got the idea how it works basically. Please remember that rapid and PROX and two different applications. Rapid script does all the magics and make your life easier through automation of PROX, and PROX is the foundation application that does the "real" works. In fact, PROX can run tests just fine without Rapid. To launch PROX and start traffic, in the "gen" VM’s home folder (root in our case) start this command:

[root@stack2-gen ~]# /root/prox/build/prox -f /root/gen.cfg

PROX will parse its configuration file /root/gen.cfg and start to boot. from the booting messages in the screen we can learn its booting sequences:

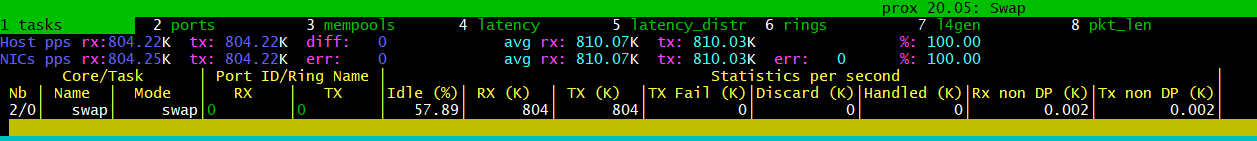
* setuping the DPDK environment (RTE EAL)
* initializing (rte) devices,
* initializing mempools, port addresses, queue numbers and rings on cores
* initializing DPDK ports
* initializing tasks
* start the test and display a ncurse based text UI

You will end up with a ncurse based UI like below:



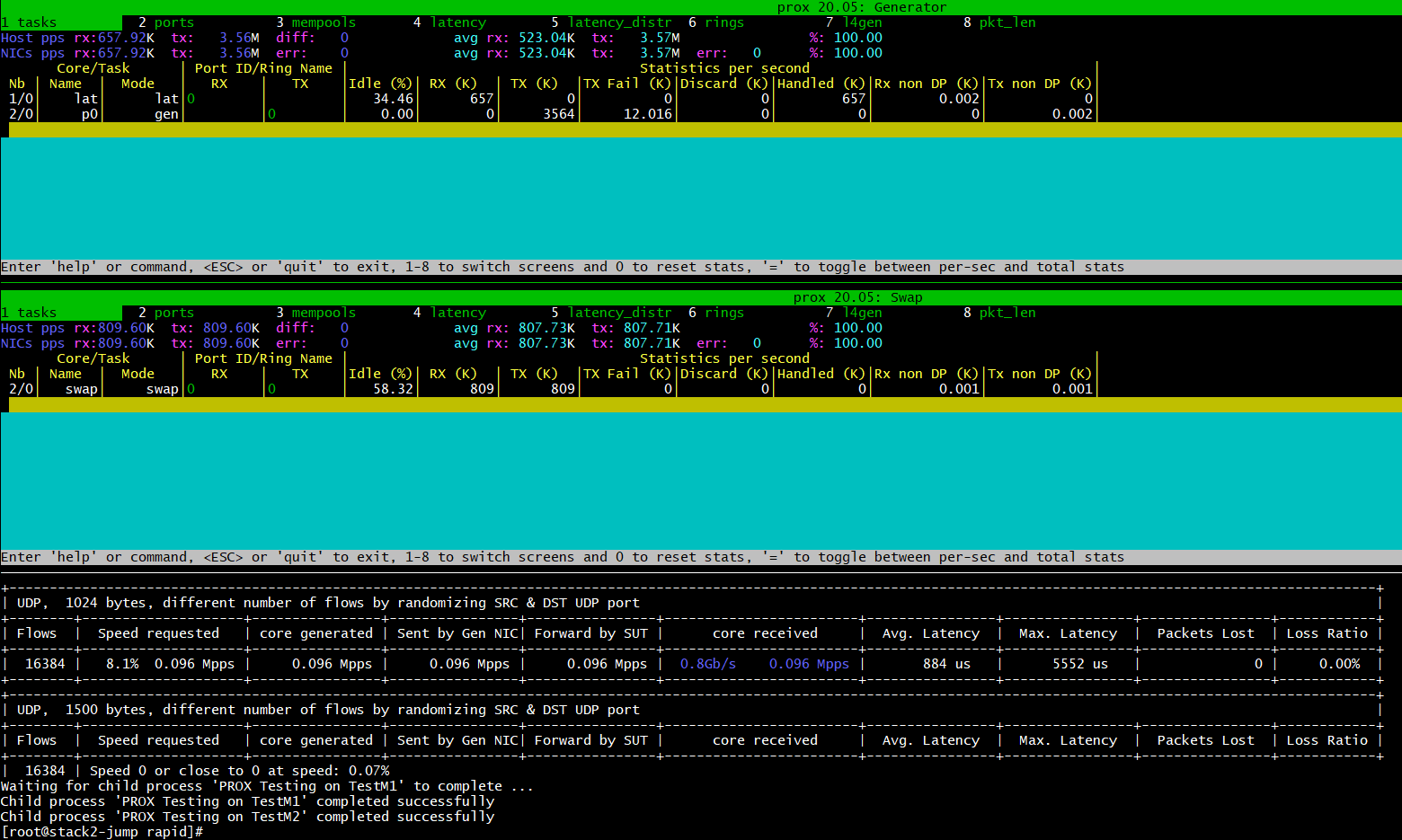
gen running UI

The display shows per task statistics which include: estimated idleness, per second statistics for packets received, transmitted or dropped; per core cache occupancy, cycles per packet, etc. These statistics can help pinpoint bottlenecks in the system. This information can then be used to optimize the configuration. There are quite a few other features include debugging support, scripting, Open vSwitch support, etc. Refer to PROX website for more details. For now, let’s look at how the traffic flows. Right now from the screenshot above we only see traffic being sent, but nothing gets received yet. Reason is we are now running PROX manually and we only starting the "gen" side, which is the traffic "sender" only. We need to start the "swap" VM as well as a "receiver", who will also "loop" the traffic back to the sender, so our first PROX application will see some "RX" statistics. Let’s do that. On the compute where "swap" VM is installed, execute the same prox command line, except this time we pass a different configuration file named swap.cfg:



swap running UI

Here you will end up with a similiar ncurse based text UI, after similiar booting process as of the sender. Once our "swap" end of PROX is up and running, immediately you will see both "RX" and "TX" counters keep updating on both side of the traffic:



gen and swap UI

That concludes our discussion of PROX and rapid as our testing tools. We’ll use this tools intensively in the rest of this chapter to generate different kinds of traffic in each tests. With the traffic running, we can dig deeper to understand the rules we’ve introduced about how vRouter works. Next we’ll introduce some of the commonly used tools that are designed for, or especially useful for verfications in DPDK vRouter environment.

# dpdk vRouter tool box

In this book you’ve read a lot of details about DPDK and contrail DPDK vRouter implementations. You should understand that performance boost is the main benefit it brings. As with almost everything, it has both pros and cons. One problem is that is commonly raised is the lack of tools during troubleshooting process, especially in the case of a traffic loss problems. Within 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 like 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 are also hidden from the perspective of these tools relying on it. In production, we need some new tools developed to fill this gap, so that we can narrow the packet loss related issues when the outage is ongoing. Fortunately, today contrail dpdk vRouter are equiped 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 into 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", which you’d like to open whenever you want to check any running states of the vRouter dataplane. Let’s first take a look at how to "open" this "box".

To enter the container, just run 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, 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 20 tools available in this container. Let’s take a look at what’s in the package.

First, in the container we’ll 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, we 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 dpdk\_nic\_bind.py script, which is a tool to tell bind a specific driver for a NIC. In the rest of this section, we’ll introduce some more tools that is especially useful in DPDK environment.

## vif command and scripts

The first one from our contrail DPDK "tool box" is vif command. Before talking about it, let’s see how do we 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, we do see some interfaces got printed:

* the loop interface (lo)
* managment 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 we compare with what we would see with the same ip command in a kernel mode vRouter compute without DPDK, we will see the big differences:

[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 lo, management interface and whatever we saw from the DPDK compute, we also see these all other important interfaces:

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

pkt1, pkt2, pkt3 interfaces are created by vRouter but not used in dpdk setup

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

We’ll talk more about this later. for now, let’s look at the vif command with -l|--list and --get option. 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  
  
Flags: P=Policy, X=Cross Connect, S=Service Chain, Mr=Receive Mirror  
 Mt=Transmit Mirror, Tc=Transmit Checksum Offload, L3=Layer 3, L2=Layer 2  
 D=DHCP, Vp=Vhost Physical, Pr=Promiscuous, Vnt=Native Vlan Tagged  
 Mnp=No MAC Proxy, Dpdk=DPDK PMD Interface, Rfl=Receive Filtering Offload, Mon=Interface is Monitored  
 Uuf=Unknown Unicast Flood, Vof=VLAN insert/strip offload, Df=Drop New Flows, L=MAC Learning Enabled  
 Proxy=MAC Requests Proxied Always, Er=Etree Root, Mn=Mirror without Vlan Tag, HbsL=HBS Left Intf  
 HbsR=HBS Right Intf, Ig=Igmp Trap Enabled  
  
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 vif command, especially in DPDK vRouter. It shows interfaces from vRouter’s perspective, and reveals the one to one connection mapping between vRouter and fabric or VM tap interface. The latter would be "invisible" otherwise.

Besides that, it also prints other important information. The Vrf numbers and packet counters are the most commonly used data points. 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, sometime you may also see non-zero numbers in RX/TX queue  
packets/errors counter that gives inter lcore packet statistics. It is usually happens when two lcores are involved in the packet forwarding path. We’ll use this command intensively in the rest of this chapter, and we’ll analyze these counters and use them to understand some of the important vRouter working mechanisms.

vif tool also support some other options, use --help to print 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 talk about each every options and their usage and usually you don’t need to know anything except --get and -l|--list. There is one more (--add) which we’ll talk about shortly. For others you can refer to <https://www.juniper.net/documentation/en_US/contrail20/topics/task/configuration/vrouter-cli-utilities-vnc.html> for more details.

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

### dpdkvifstats.py script

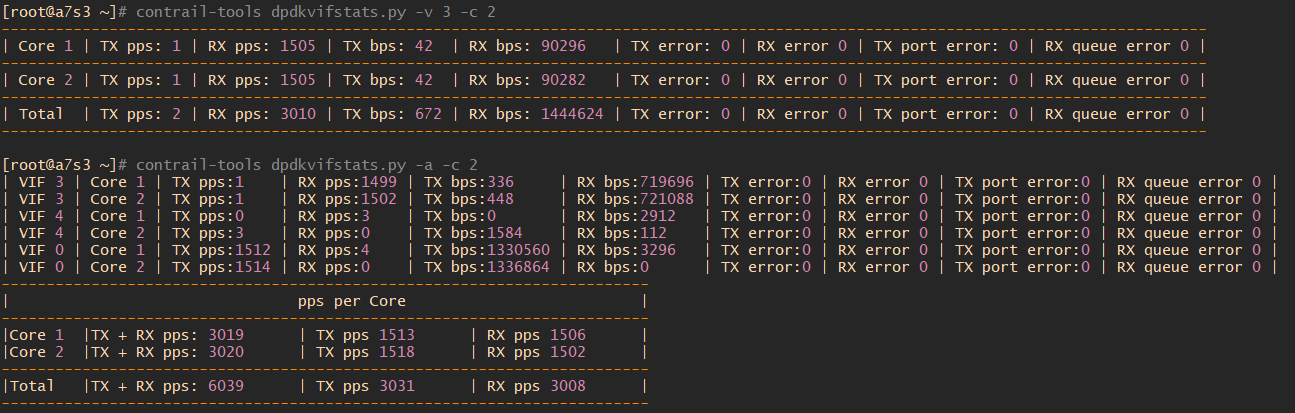
We’ve seen vif command prints all interfaces and its traffic statistics (RX/TX packets/bytes/errors, RX queue packets/errors, etc) in the form of a "list". During testing or troubleshooting, we can collect these data to evaluate the vRouter forwarding performance, its running status, is it losing packets or not, etc. In production, we always need to examine the traffic passing through a compute. Same thing in lab, once you start traffic from PROX or any other traffic generators, the first thing you want to check is the traffic rate on 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, calcuates the changing rate of all counters in pps and bps, then prints the result in a table format. This makes the output looks much "prettier", and also makes comparison accross vif interfaces much easier.

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

Let’s take a look:



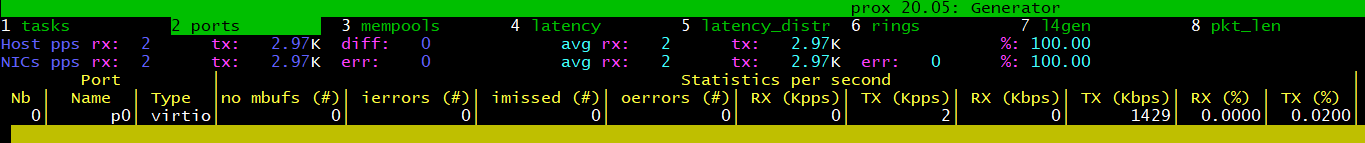
dpdkvifstats.py

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 same number of lcores and DPDK queues as the number of CPUs allocated to it**. In this compute, for testing purpose, we’ve allocated 2 CPU cores to vRouter dpdk forwarding lcores. Therefore, for each vRouter interface, 2 DPDK queues are created, each served by a forwarding lcore in DPDK process. That is why that in the output for each vif interface there are 2 lines statistics, for "Core 1" and "Core 2" respectively.

CPU allocatioin to DPDK vRouter forwarding lcores is configurable via options in vRouter configuration files. details of CPU allocation implementation is beyond the scope of this book.

Now let’s look at the counters. To demonstrate how the script works, 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 64 bytes. That calculates to about 1.4K packet per second (PPS).



PROX gen sending 125000 traffic with speed of Bytes per second

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

This is very straightforward. To comparing with the vif output, let’s check what the "raw" data looks like if without dpdkvifstats.py script:

[root@a7s3 ~]# date; contrail-tools vif --get 3; sleep 10; date; contrail-tools vif --get 3  
Wed Oct 7 07:08:36 PDT 2020  
  
......  
vif0/3 PMD: tap41a9ab05-64 NH: 38  
 Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.1.104  
 Vrf:3 Mcast Vrf:3 Flags:L3L2DEr QOS:-1 Ref:12  
 RX queue packets:1457762899 errors:0  
 RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0  
 RX packets:1457893340 bytes:87471243818 errors:0  
 TX packets:208763 bytes:10136442 errors:0  
 ISID: 0 Bmac: 02:41:a9:ab:05:64  
 Drops:33  
  
Wed Oct 7 07:08:47 PDT 2020  
  
......  
vif0/3 PMD: tap41a9ab05-64 NH: 38  
 Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.1.104  
 Vrf:3 Mcast Vrf:3 Flags:L3L2DEr QOS:-1 Ref:12  
 RX queue packets:1457797939 errors:0  
 RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0  
 RX packets:1457928405 bytes:87473347268 errors:0  
 TX packets:208788 bytes:10137492 errors:0  
 ISID: 0 Bmac: 02:41:a9:ab:05:64  
 Drops:33

We capture the interface data, wait for 10 seconds, and capture it again. After that we can calculate the differences of all counters between the two captures. We then divide each difference by 10 to get the increasing "rate" of each counter.

* pps - packets per second: (1457928405-1457893340)/10 = 3506.5
* Bps - bytes per second: (87473347268-87471243818)/10 = 210345
* bps - bit per second: 210345 \* 8 = 1682760

…​TODO: the number still does not match to script result well…​

To monitor multiple vif interfaces we have to repeat these steps multiple times. Compare these manual works with having a handy script doing everything for you!

dpdkvifstats.py script is useful to quickly retrieve a snapshot of current traffic profile at the moment and basically that’s it. When everything goes well that is fine. In the case of traffic loss, we often need to first "capture" the packets themselves. Then based on the packet capture, we can decode the payload, and analyze the issue. Now you may say: oh you mean the tcpdump! Well, Yes and no. Please remember the fact that we are in a setup where NIC card is invisible to most of the linux applications - including tcpdump! Next let’s briefly go over this DPDK vRouter packet capture script: vifdump.

### vifdump script: TODO

In many linux machine tcpdump comes with the OS as part of a standard packets. With that you can capture whatever packets sensed by a NIC, which can be either physical NIC or virtual NIC like a tuntap interface, both NIC are visible to the kernel. In DPDK environment, the difficulty of an interface not being visible to the kernel makes tcpdump not able to work, unless you just want it to read packets from a file. Fortunately, we now know that each interface related to vRouter dataplane connects to a unique vRouter interface (vif). We can make use of this fact and create something alternative. vifdump is a shell script, when invoked, it use --add option of vif command to creates a "monitoring" tun interface in linux kernel, and connects it to the vif interface that we want to monitor, then start up tcpdump program to capture the packets from the "monitoring" tun interface. From a user perspective, the script works the same way as with tcpdump:

vRouter: vif0/3 ------+----- tapxxx: VM  
 |  
 vif0/4348----+----- mon3: host

[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 use trap to monitor user key-board triggered interrupt, and delete the monitoring interface when capture is stopped.

we need to dig deeper into some internal data structure to narrow down the problem. In next section, we’ll take a look another powerful debug tool that is useful in 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. dpdkinfo is introduced since Contrail 20.08. Using this tool Contrail operators can collect information about DPDK vRouter 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, they are:

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

There are some other options like --app|-a, --mempool|-m we won’t introduced 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 the official documents for other usage informations.

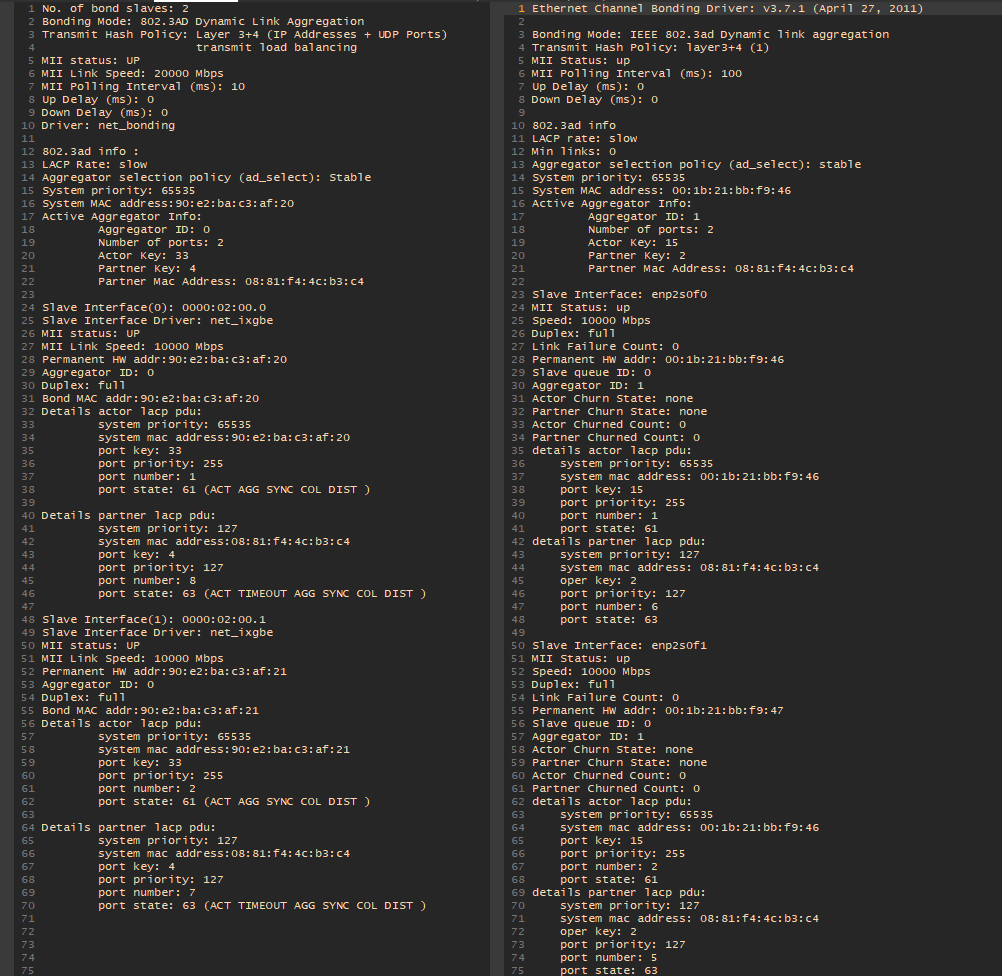
### version

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

-b or --bond option print detail information about the bond interface managed by DPDK. The output is organized in a similiar form as what you would see for the bond status managed by linux kernel. Compare this output below with cat /proc/net/bonding/bond0 output from a compute running kernel mode vRouter:



dpdkinfo -b vs. cat /proc/net/bonding/bond0

Basically now you can have same information as of 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 LACP protocol is configured between compute and peer device (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 slaves 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 commmand outupt also displays each member(slave) link’s information, its currrent driver, MAC address, up/down status, etc. You may notice that the slave interface is identified using PCI bus number (0000:02:00.0 and 0000:02:00.1) instead of interface name as comparing with linux bond. Again the reason is that the interface name is created by linux kernel, which is "bypassed" in dpdk.

### dpdk\_nic\_bind: TODO

(vrouter-agent-dpdk)[root@a7s3 /]$ /opt/contrail/bin/dpdk\_nic\_bind.py --status

Network devices using DPDK-compatible driver  
============================================  
0000:02:00.0 '82599ES 10-Gigabit SFI/SFP+ Network Connection' drv=uio\_pci\_generic unused=ixgbe  
0000:02:00.1 '82599ES 10-Gigabit SFI/SFP+ Network Connection' drv=uio\_pci\_generic unused=ixgbe

Network devices using kernel driver  
===================================  
0000:04:00.0 'I350 Gigabit Network Connection' if=eno1 drv=igb unused=uio\_pci\_generic \*Active\*  
0000:04:00.1 'I350 Gigabit Network Connection' if=eno2 drv=igb unused=uio\_pci\_generic

Other network devices  
=====================  
<none>

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

### bond packet counters

-n|--stats option is useful to look into packet statistics of bond interface. So far we’ve seen at least 2 ways of retreiving packet counters from a vif interface:

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

DPDK bond interface is represented by 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 prints 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) received 1421 packets, while the second member link (PCI bus 0000:02:00.1) received 28359275 packets. It is obvious that the second member link carries most part of the traffic. Maybe you are wondering why we end 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 in this test environment we are sending just one UDP flow!

With more flows we’ll see the balance happens. let’s send 10 flows, but before that let’s clear the current counters to make our second camparison easier:

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

Now we start 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, we are sure the traffic get balanced on both links.

Now you understand the -stats|-n option provides the insight of member link usage reflected by a few RX/TX counters. Base on these information we 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 vif command. In practice, if you need to get more extensive statistics, there is another option xstats|-x. Let’s go 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 give. In fact to shorten the output, we’ve removed all counters with a zero value in it, and also edited the output format to compact all texts in 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 size between 65 to 127 bytes, and that is what we are sending from 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 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. Consider 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. Before we start introducing -c|--lcore option, let’s briefly review these concepts.

lcore

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

interface

is the endpoints of connections between vRouter and 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 give the host an IP address and through with the vRouter agent can exchange control plane messages with the controller.

queue

for each interface there are some queues created. They are essentially some memories allocated to hold the packets.

The CPU cores connect all these objects together. As of the writing of this book, the implemention 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 4 CPUs are allocated to DPDK vRouter forwarding threads (the lcores), then 4 lcores will be created, and 4 DPDK interface queues will be created for each vif interface. Same rule applies to the VM - You assign 4 CPU cores to a VM, then by default, Nova will create 4 (virtio??) queues for a tap interface in the VM. That said, of course, multiple queue as a feature needs to be turned on in Nova at the first place. We can illustrate this with a table below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| vif | queue | lcore | queue | tap(vNIC) |
| 0/3 | 0 | 0 | 0 | tap003 |
| 1 | 1 | 1 |  |  |
| 2 | 2 | 2 |  |  |
| 3 | 3 | 3 |  |  |
| 0/4 | 0 | 0 | 0 | tap004 |
| 1 | 1 | 1 |  |  |
| 2 | 2 | 2 |  |  |
| 3 | 3 | 3 |  |  |

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

* What if the tap interface queue number is different than the vif queue number? What will happen when we have 8 lcores, but one of our VM are running 4 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 2 forwarding lcores running in total.

Then, the second line give number of vRouter interfaces in the compute. We have 4 of them in total. One vif0/4 connecting to VM tap interface tap41a9ab05-64, 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.

Starting from the third line onward are what we’ll focus now. The output is listing all forwarding lcores that are currently configured in vRouter, and for each lcore it list interfaces that this lcore is associcated with - in another word, interfaces this core is "serving".

Please note that there are some inconsistencies in term of the lcore numbering in different tools. \* In dpdkvifstats.py script, forwarding lcore number starts from "1", so "Core 1" refers to the first forwarding lcore. \* In dpdkinnfo -c output, forwarding lcore number starts from "0", so "Lcore 0" refers to the first forwarding lcore. \* In vif output, forwarding lcore number starts from "10", so "--core 10" refers to the first forwarding lcore.

This may cause some confusions 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.

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |

OK. 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 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. bond0 are the physical interfaces, and it will always has 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 worth to know to understand the whole output.

Finally, Let’s look at the last line - the VM tap interface. From the output, we 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 2 lcores or some algorithms are 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 is `lcore#11’s turn to take the next interface and queue.

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).

We’ll see more examples in later sections, in there we’ll test out the "tie breaker" and other things. We can convert the above mapping into a table like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| vif | queue | lcore | queue | tap(vNIC) |
| 0/0 | 0 | 0 | 0 | bond0 |
| 1 | 1 | 1 |  |  |
| 0/1 | 0 | 0 | 0 | vhost0 |
| 0/3 | 0 | 1 | 0 | tap41a9ab05-64 |

Now we’ve went through dpdkinfo command and demonstrated some most commonly used options. With this command you can quickly print out a lot of useful information about DPDK and DPDK vRouter running status. We’ll review this again later in our test case studies. These information is important to know before we work 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 (TODO)

TODO: this is copied and rewritten based on Laurent’s chapter 5. will need to capture in same environment and rewrite again.

Contrails DPDK vrouter dataplane 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. Understanding the log messages are important during 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. We can see the DPDK library version that has be use to build the DPDK vrouter binary program.

Here is an example:

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"

Here we see a complete list of vRouter start up parameters of this Contrail vRouter, for example:

* build-version "2008"
* is 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 these information with what we can print 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’ve introduced that DPDK vRouter process is a multiple threads application and the threads falls into different categories based on their roles. This is also reflected by some log entries. Before we dive into the logs, Let’s do a quick review of the three thread categories:

Control threads

They are generated by DPDK libraries and are used during Contrail vRouter startup for DPDK initialization. control threads are not our focus in this book.

Cervice threads

There are totally hard-coded two service threads named lcore0 through lcore9. Each lcore has its own role. For example lcore9 serves netlink connection between agent and vRouter data plane. Details of each lcore’s rule is out of this book’s scope. We just need to know they are used to serve communication between vrouter agent and vrouter forwarding plane.

Forwarding threads

After service threads, from lcore10 and onward, the forwarding threads are the main horse power that performs the packet forwarding tasks and determines the performance of DPDK vRouter. This is the main focus of our book.

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

OK. Now let’s take a look at a 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 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 we need to understand CPU pinning. To achieve maximum performance we’re pinning the service and forwarding threads(or lcores) each 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 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 situation 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 due to historically only MPLS GRE was supported. So, it remains like that in the software code. Here is means both MPLSoGRE and VxLAN packet will be distributed via hashing by the polling core.

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

### 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 CPU (specified in CPU\_LIST parameter). Each queue created is handled by only one of the vrouter polling core. 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 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 4 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 2 to get the queue number. i.e. vring 0 is TX queue 0, vring 2 is TX queue 1, …
* the receive queues are the odd numbers. Divide them by 2 (discard the remainder) to get the queue number. i.e. vring 1 is RX queue 0, vring 3 is RX queue 1,
* ready state 1 = enabled. ready state 0 = disabled

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

In the example below, 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

In the example hereafter, 4 RX (and TX) queues are enabled on the vrouter vif interface. But a virtual machine interface having more than 4 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 4 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,)

# dpdk vRouter case studies (TODO: still writing)

In previous sections, we’ve introduced some dpdk tools and explained some important log entries. to help collecting DPDK vRouter running status.

## single queue

Having understood the lcore mapping basics, let’s start a test with some traffic flowing.

### 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. We can list current flows we have in vRouter to confirm this.

[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, we see 6 vRouter flows, which are in fact 3 groups. The first 2 groups with index pairs 40196/436016 and 62792/172020 are generated by the control messages from 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 shows 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 real world most of the traffic are bidirectional, so having an seperate entry built for each direction is required. In our case, from PROX we are generating uni-directional traffic, so only flow of that direction has packet statistics. it’s pairing flow entry is generated as well, but packet statistics shows nothing.

Let’s clear 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 "lcore#10" received the traffic - we can tell from the RX speed 1504 pps showing in "Core 1" only. The second capture on the vRouter interface toward bond interface (-v 0) confirmed the same - it is the same lcore#10 ("Core 1" here) that is sending the traffic to the bond interface, at speed of 1512 pps, almost the same as the speed it received the traffic from VM tap interface. This flow is illustrated here:

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

This seems to be "weird", does it? Remember previously based on the core-interface mapping given by dpdkinfo -c We already knew it was the lcore#11 serving our VM interface, not the other one. Accordingly, in 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 our 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 the when traffic flows from 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 much about these details, but if you collect vif output, you will see some more 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 a "RX queue" counter Core 11 RX queue packets:35384 gives a little bit 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 oto the first forwarding lcore, Core  
10, 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.

What we’ve tested and demonstrated is the DPDK vRouter default behavior with the current parameters it current takes. 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 we can verify the vrouter-dpdk process running command line in our setup with ps command. With that configured, vRouter will change to the so-called run to complete model, which means same lcore whichever polled the packet will continue to process/forward it. This requires reboot of DPDK vRouter, and We won’t test this scenarios in this book.

This concludes the analysis of traffic forwarding in the direction of 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 we add returning traffic. We configure the swap VM in such a way that it loops whatever it received back to the sender. Here is the capture:

[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 ..  
 ---------------------------------------------------------------------------------  
  
 [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..  
 ---------------------------------------------------------------------------------

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

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

So we 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 out of vif0/3. To confirm if this lcore is also the polling lcore, we’ll need to look at the vif capture:

[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

We do not see any RX queue packets as what we’ve seen in the data we collected on the VM to fabric direction. Therefore in this direction we don’t see any inter-core load balancing behavior as we’ve elaborated before.

This concludes our analysis to the bidirectional single flow traffic. As you can see, one benefit to have traffic generator/swapper built in lab environment is, we can fine tune the generator to send traffic in a very specific pattern, so that we can take a deep look at the counters and analyze the vRouter traffic forwarding behavior. This is very helpful for learning purpose. In production, you probably never expect to has such a "luxury" since the traffic pattern in the "field" is usually much more complex. But don’t worry, we 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, we’ll add more flows in our testbed and check the result.

### multiple flows

Here, we are sending 64 flows from PROX gen VM. To confirm the flow numbers we use 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

132 flows entries means 66 groups of flows in our test. The additional 2 groups of flows are the control flows between 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 to 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 our 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 single queue in VM tap interface, it’s hard to achieve good load balance between lcores on the vRouter interface facing the Virtual Machine. Sometime we need to enable "multiple queue" to make better use of all our DPDK forwarding lcores.

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

## multiple queues

Let’s look at a multiple queue example.

Based on the previous setup, this time we added one more queue in 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 mapping:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| vif | queue | lcore | queue | tap(vNIC) |
| 0/0 | 0 | 0 | 0 | bond0 |
| 1 | 1 | 1 |  |  |
| 0/1 | 0 | 0 | 0 | vhost0 |
| 0/3 | 0 | 1 | 0 | tap41a9ab05-64 |
| 1 | 0 | 1 |  |  |

So most items remains the same, except we have one more queque 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 know that each of our lcores is serving same amount of queues, therefore as a "tie breaker", which we’ve mentioned when we introduce 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, since we have multiple queues on both VM tap interface and the fabric interface. Traffic on all lcores are very well balanced. Please keep this in mind as an "ideal" traffic profile that we are expecting the vRouter to has. 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 knows what to look when the result is far worse than expected.