chapter 6

contrail networking and test tools installation

2020-10-07

# 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

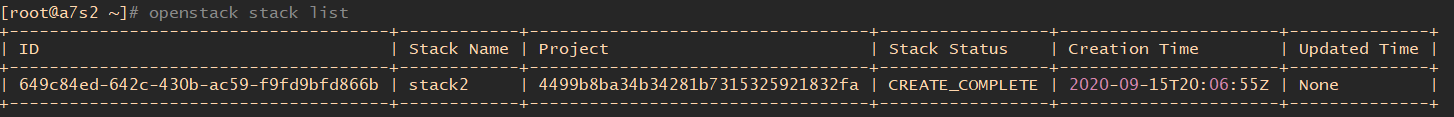
1. (Optionally) if you’re using ceph backend:

* 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

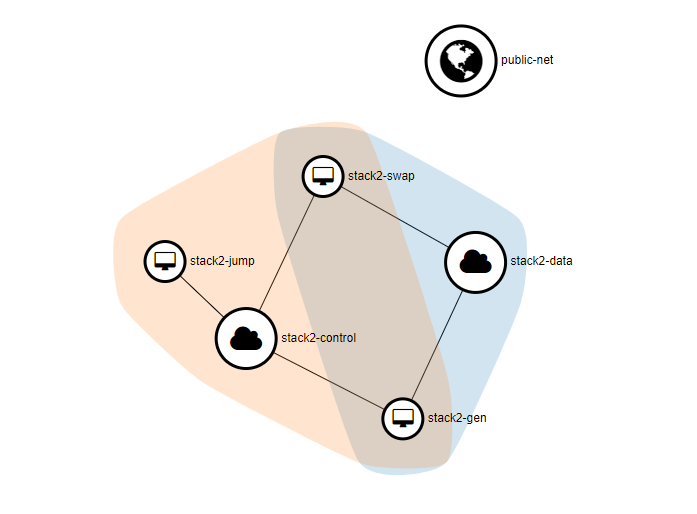
1. adjust the heat template files based on your environment
   * environment.yaml
   * build-rapid.yml
   * configure.rapid.sh
2. create heat stack:

* 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.



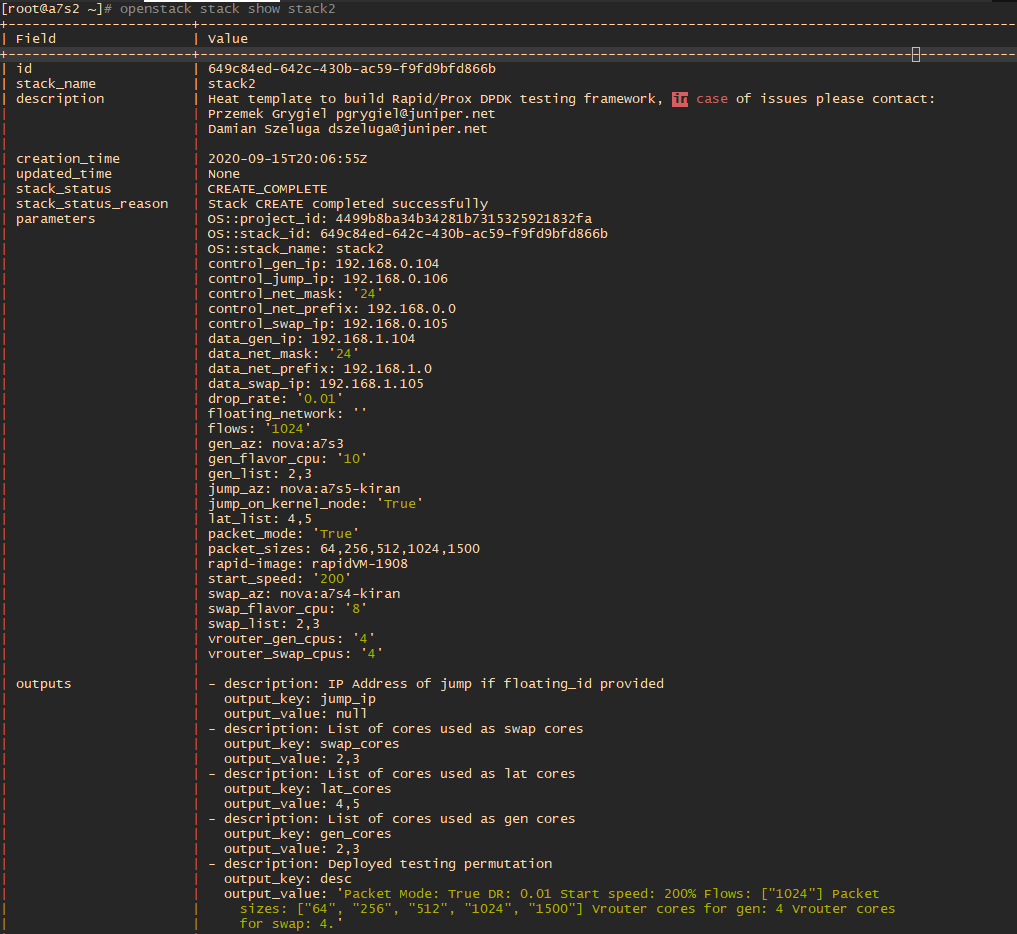
openstack stack list



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:

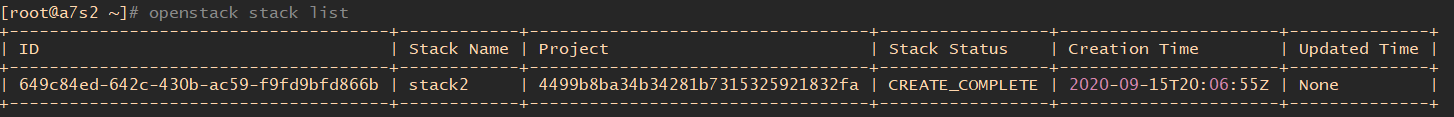
[root@a7s3 ~]# docker exec -it nova\_libvirt virsh list  
 Id Name State  
----------------------------------------------------  
 2 instance-00000041 running

[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

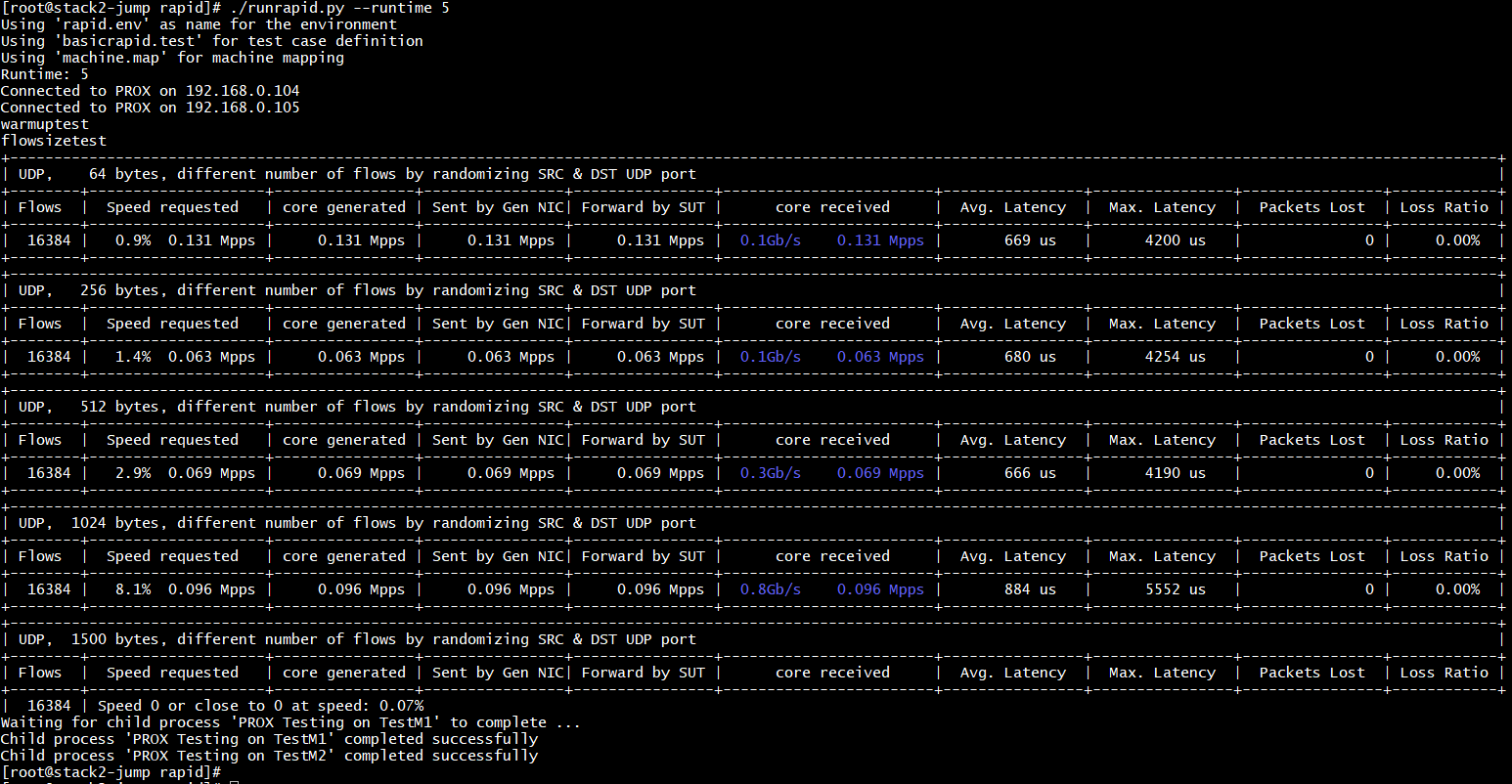
A few other command line options are supported, which can be listed by -h:

[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. Warmup can be use as preload of flow table if we don’t want to wait till all flows will be created during a test. It is important in case of large flow table e.g. 1M flows tests.
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

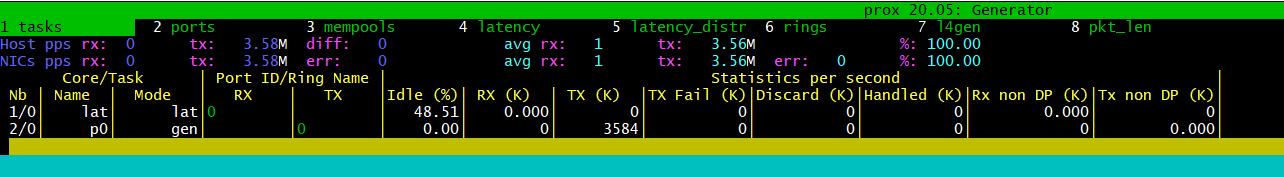
OK. We just introduced rapid. The script support very extensive options in the configuration files which beyond the scope of this book, but we’ve got the idea how it works basically. Please remember that Rapid and PROX and two different applications. Rapid script does all the magics and make your life 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:

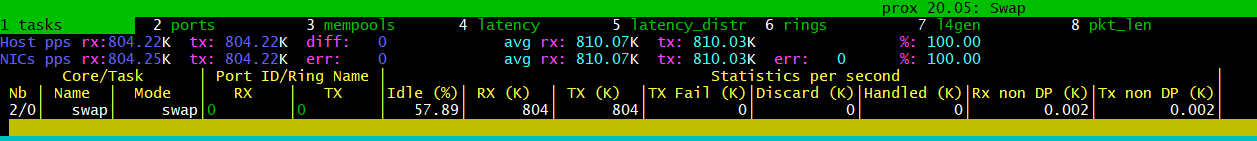
* setuping the DPDK environment (RTE EAL)
* initializing (rte) devices,
* initializing mempools, port addresses, queue numbers and rings on cores
* initializing DPDK ports
* initializing tasks
* start the test and display a ncurse based text UI

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



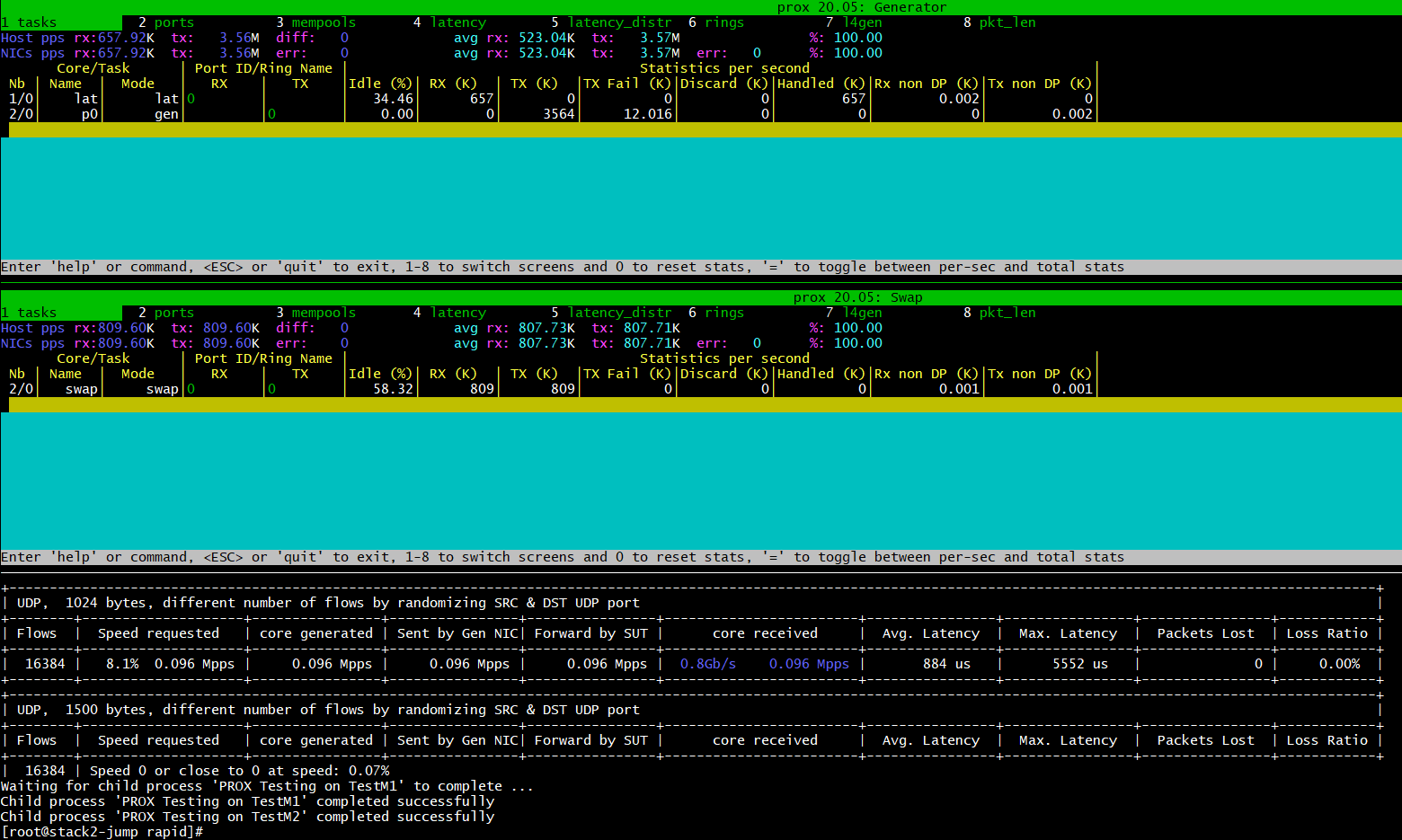
gen running UI

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

# contrail dpdk tools

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 are familiar with, for example, to list all vRouter interfaces:

[root@a7s3 ~]# contrail-tools vif -l  
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

vif0/0 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 is the bond interface
* vif0/1: this is 'vhost0', the interface toward linux kernel
* vif0/2: this is the netlink interface (hence the name "unix") toward vrouter agent
* vif0/3: this is the vRouter interface connecting the data interface of our PROX VM
* vif0/4: this is the vRouter interface connecting the control and management interface of our PROX VM

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-too  
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 two more tools that is useful in DPDK environment - dpdkvifstats.py and dpdkinfo.

## dpdkvifstats.py

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. 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 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 and prints in a table format. This makes the output looks much "prettier", and also makes comparison accross vif interfaces much easier. Let’s take a look:

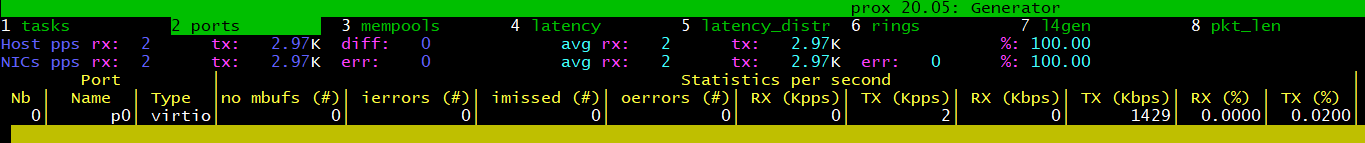
[root@a7s3 ~]# contrail-tools dpdkvifstats.py -v 3 -c 2  
 -------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------  
 |Core 1 | TX pps: 1 | RX pps: 1505 | TX bps: 42 | RX bps: 90296 | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |  
 -------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------  
 |Core 2 | TX pps: 1 | RX pps: 1505 | TX bps: 42 | RX bps: 90282 | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |  
 -------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------  
 |Total | TX pps: 2 | RX pps: 3010 | TX bps: 672 | RX bps: 1444624 | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |  
 -------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------  
  
 [root@a7s3 ~]# contrail-tools dpdkvifstats.py -a -c 2  
 | VIF 3 |Core 1 | TX pps: 1 | RX pps: 1499 | TX bps: 336 | RX bps: 719696 | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |  
 | VIF 3 |Core 2 | TX pps: 1 | RX pps: 1502 | TX bps: 448 | RX bps: 721088 | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |  
 | VIF 4 |Core 1 | TX pps: 0 | RX pps: 3 | TX bps: 0 | RX bps: 2912 | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |  
 | VIF 4 |Core 2 | TX pps: 3 | RX pps: 0 | TX bps: 1584 | RX bps: 112 | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |  
 | VIF 0 |Core 1 | TX pps: 1512 | RX pps: 4 | TX bps: 1330560 | RX bps: 3296 | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |  
 | VIF 0 |Core 2 | TX pps: 1514 | RX pps: 0 | TX bps: 1336864 | RX bps: 0 | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |  
 ------------------------------------------------------------------------  
 | pps per Core |  
 ------------------------------------------------------------------------  
 |Core 1 |TX + RX pps: 3019 | TX pps 1513 | RX pps 1506 |  
 |Core 2 |TX + RX pps: 3020 | TX pps 1518 | RX pps 1502 |  
 ------------------------------------------------------------------------  
 |Total |TX + RX pps: 6039 | TX pps 3031 | RX pps 3008 |  
 ------------------------------------------------------------------------

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!

Next, we’ll take a look another powerful dpdk tool: dpdkinfo.