Chapter 5

Contrail Networking and Test Tools Installation

The previous chapters have gone through most of the important topics about SDN and DPDK in general, DPDK vRouter architectures, vRouter packet processing details, and so on. When you read these topics you may wonder how to get a running Contrail networking environment with a few DPDK vRouters in it, so you can play around, test those theories and familiarize yourself about what you’ve learned. Indeed, those topics are important, 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.

This and the next chapters mostly focuse on hands-on lab testing to verify some of the most important DPDK vRouter concepts and working mechanisms.

* In this chapter, we’ll start from introducing steps we’ve used to install the latest version of Contrail networking cluster.
* On top of that, in the rest part of this chapter we’ll start to build a testing environment. That includes a few VMs running OPENFV PROX software. On each VM, based on its role, the PROX software is configured as either a traffic generator or a traffic receiver.
* In the next chapter, We’ll introduce some of the commonly used DPDK tools, scripts, and log entries to provides useful information to help you understand how things run in a DPDK environment.
* In the end of next chapter, we’ll go over some case studies. We use PROX and rapid we’ve installed to start different traffic patterns in the setup, and then use DPDK tools to analyze what we are seeing.

After reading these two chapters, you will have a deeper understanding of some of the main concepts covered in this book. Let’s start with Contrail installation.

# Contrail Installation

## contrail installation methods

We’ve been focusing on the DPDK vRouter that runs in each individual compute node, which basically runs in a relatively standalone mode. But if you look at the forwarding plane as a whole, those nodes are actually a distributed system. In fact, as we’ve briefed in Chapter 1, the whole TF cluster is a complex distributed system involving a lot more different software modules, especially in control plane. Again, each of the software modules can be a completely different distributed system by themselves. The Cassandra database that the TF cluster uses is one such example. 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, verified process with step-by-step instructions to *avoid* the issues, than starting with a *try-and-see* mode and then try to fix the issues.

Currently, the TF cluster has been widely integrated with many majority deployment systems and platforms. Therefore, depending on your environment, there can be totally different ways of installing the Contrail system. Here is an incomplete list of currently supported installation methods:

* Installing Contrail with OpenStack and Kolla Ansible
* Installing Contrail with RHOSP
* Installing Kubernetes Contrail Cluster using the Contrail Command UI
* Installing and provisioning Contrail VMware vRealize Orchestrator Plugin
* Installing a Standalone Red Hat OpenShift Container Platform 3.11 Cluster with Contrail Using Contrail OpenShift Deployer
* Installing a Nested Red Hat OpenShift Container Platform 3.11 Cluster Using Contrail Ansible Deployer
* Installing Contrail with OpenStack or Kubernetes by using Juju Charms

For example, in the second method, you can install Contrail with Redhat OpenStack platform director 13 (RHOSPd), which is a toolset based on the OpenStack project TripleO (OOO, or OpenStack on OpenStack). A TF environment built out of RHOSPd uses concepts of *undercloud* and *overcloud*. Basically undercloud is a single server containing complete OpenStack components, whose role is just to deploy and manage an overcloud, which is a tenant-facing environment that hosts the *resulting* OpenStack and TF nodes. This deployment is currently used by many major service providers in production.

However, the installation process of such a deployment involves the understanding of RHOSPd, TripleO, and of lots of different types of network isolation topologies, which add many unnecessary complexities to our lab setup. In this section, we’ll give a detail steps about the first method - 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.

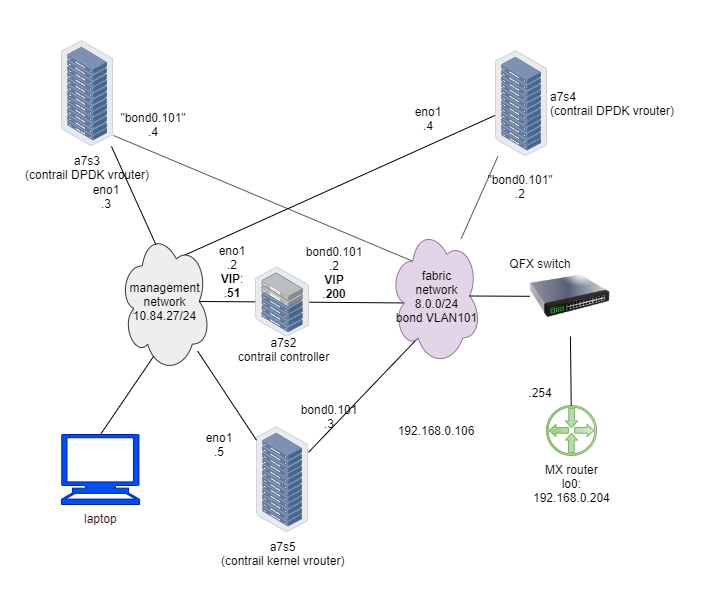


Figure 5.1 The Contrail DPDK vRouter Test Bed

## Configure Bond and VLAN

To enable bond interface in centos, under /etc/sysconfig/network-scripts/ of all the nodes where the bond interface is needed, add these configuration files:

|  |  |
| --- | --- |
| bond | members |
| $ cat ifcfg-bond0 SUBCHANNELS=1,2,3 NM\_CONTROLLED=no BOOTPROTO=none BONDING\_OPTS=miimon=100 mode=802.3ad xmit\_hash\_policy=layer3+4 DEVICE=bond0 BONDING\_MASTER=yes ONBOOT=yes | $ cat ifcfg-bond0.101 NM\_CONTROLLED=no NETMASK=255.255.255.0 BOOTPROTO=none DEVICE=bond0.101 VLAN=yes IPADDR=8.0.0.3 ONBOOT=yes |
| $ cat ifcfg-enp2s0f0 HWADDR=00:1b:21:bb:f9:46 SLAVE=yes NM\_CONTROLLED=no BOOTPROTO=none MASTER=bond0 DEVICE=enp2s0f0 ONBOOT=yes | $ cat ifcfg-enp2s0f1 HWADDR=00:1b:21:bb:f9:47 SLAVE=yes NM\_CONTROLLED=no BOOTPROTO=none MASTER=bond0 DEVICE=enp2s0f1 ONBOOT=yes |

Then restart network service to invoke these configurations:

service network restart

Once the restart is successful, you should see bond0 interface appearing in all nodes with one of these IP addresses in each node: 8.0.0.1 to 8.0.0.4. Now you should have IP connectivity in both the management network and fabric network.

Next you’ll need to install Ansible and use it to automate the rest part of the installations. Most of Ansible’s magic is done through its playbooks and configuration for all plays is done in a single file with a default name instances.yaml. This configuration file has multiple main sections. We’ll go over some of the main parameters in this file and then introduce the steps to run the playbooks. Here’s the configuration file for instances.yaml:

1 global\_configuration:  
2 CONTAINER\_REGISTRY: svl-artifactory.juniper.net/contrail-nightly  
3 REGISTRY\_PRIVATE\_INSECURE: True  
4 provider\_config:  
5 bms:  
6 ssh\_pwd: c0ntrail123  
7 ssh\_user: root  
8 ntpserver: 10.84.5.100  
9 domainsuffix: englab.juniper.net  
10 instances:  
11 a7s2:  
12 provider: bms  
13 ip: 10.84.27.2  
14 roles:  
15 openstack\_control:  
16 openstack\_network:  
17 openstack\_storage:  
18 openstack\_monitoring:  
19 config\_database:  
20 config:  
21 control:  
22 analytics\_database:  
23 analytics:  
24 webui:  
25 a7s3:  
26 provider: bms  
27 ip: 10.84.27.3  
28 ssh\_user: root  
29 ssh\_pwd: c0ntrail123  
30 roles:  
31 openstack\_compute:  
32 vrouter:  
33 PHYSICAL\_INTERFACE: bond0.101  
34 CPU\_CORE\_MASK: 0x1fe  
35 DPDK\_UIO\_DRIVER: uio\_pci\_generic  
36 HUGE\_PAGES: 32000  
37 AGENT\_MODE: dpdk  
38 a7s4:  
39 provider: bms  
40 ip: 10.84.27.4  
41 ssh\_user: root  
42 ssh\_pwd: c0ntrail123  
43 roles:  
44 openstack\_compute:  
45 vrouter:  
46 PHYSICAL\_INTERFACE: bond0.101  
47 CPU\_CORE\_MASK: 0x1fe  
48 DPDK\_UIO\_DRIVER: uio\_pci\_generic  
49 HUGE\_PAGES: 32000  
50 AGENT\_MODE: dpdk  
51 a7s5:  
52 provider: bms  
53 ip: 10.84.27.5  
54 ssh\_user: root  
55 ssh\_pwd: c0ntrail123  
56 roles:  
57 openstack\_compute:  
58 vrouter:  
59 PHYSICAL\_INTERFACE: bond0.101  
60 contrail\_configuration:  
61 CONTRAIL\_VERSION: 2008.108  
62 OPENSTACK\_VERSION: rocky  
63 CLOUD\_ORCHESTRATOR: openstack  
64 CONTROLLER\_NODES: 8.0.0.1  
65 OPENSTACK\_NODES: 8.0.0.1  
66 CONTROL\_NODES: 8.0.0.1  
67 KEYSTONE\_AUTH\_HOST: 8.0.0.200  
68 KEYSTONE\_AUTH\_ADMIN\_PASSWORD: c0ntrail123  
69 RABBITMQ\_NODE\_PORT: 5673  
70 KEYSTONE\_AUTH\_URL\_VERSION: /v3  
71 IPFABRIC\_SERVICE\_IP: 8.0.0.200  
72 VROUTER\_GATEWAY: 8.0.0.254  
73 two\_interface: true  
74 ENCAP\_PRIORITY: VXLAN,MPLSoUDP,MPLSoGRE  
75 AUTH\_MODE: keystone  
76 CONFIG\_API\_VIP: 10.84.27.51  
77 ssh\_user: root  
78 ssh\_pwd: c0ntrail123  
79 METADATA\_PROXY\_SECRET: c0ntrail123  
80 CONFIG\_NODEMGR\_\_DEFAULTS\_\_minimum\_diskGB: 2  
81 CONFIG\_DATABASE\_NODEMGR\_\_DEFAULTS\_\_minimum\_diskGB: 2  
82 DATABASE\_NODEMGR\_\_DEFAULTS\_\_minimum\_diskGB: 2  
83 XMPP\_SSL\_ENABLE: no  
84 LOG\_LEVEL: SYS\_DEBUG  
85 AAA\_MODE: rbac  
86 kolla\_config:  
87 kolla\_globals:  
88 kolla\_internal\_vip\_address: 8.0.0.200  
89 kolla\_external\_vip\_address: 10.84.27.51  
90 contrail\_api\_interface\_address: 8.0.0.1  
91 keepalived\_virtual\_router\_id: 111  
92 enable\_haproxy: yes  
93 enable\_ironic: no  
94 enable\_swift: no  
95 kolla\_passwords:  
96 keystone\_admin\_password: c0ntrail123  
97 metadata\_secret: c0ntrail123  
98 keystone\_admin\_password: c0ntrail123

Definitions for the configuration are:

* line 1-3: global configurations
* line 2: the registry from which to pull Contrail containers
* line 3: set to True if containers that are pulled from a private registry (named CONTAINER\_REGISTRY) are not accessible
* line 4-9: provider-specific settings
* line 5: bare metal server (bms) environment
* line 6-9: ssh password, user name, ntpserver and domainsuffix
* line 10-59: Instances means the node on which the containers will be launched. here we defined 4 nodes, named a7s2, a7s3, a7s4 and a7s5 respectively
* line 11-24: this is the configuration section for node a7s2
* line 12-14: this server’s provider type (baremetal server), ip address, and roles
* line 14-24: roles of containers that will be installed in this node, according to the configuration, this server a7s2 will be installed with all “controller” software modules, in both OpenStack and Contrail
* line 25-37: parameters for our first DPDK compute node. OpenStack compute components and Contrail vRouter will be installed in it.
* line 33: under vRouter, bond0.101 will be the PHYSICAL\_INTERFACE, which is also called a *fabric interfa*ce which carries all the underlay data packets
* line 34-37: these are the DPDK specific configurations. For kernel based vRouter these are not needed
* line 34: CPU\_CORE\_MASK defines DPDK vRouter forwarding lcore pinning. The hex code 0x1fe, if converted to its binary format, is 0b000111111110. That means physical CPU core NO.1 through 8 is used as forwarding lcores lcore#10 through lcore#17
* line 35: DPDK\_UIO\_DRIVER specifies which UIO driver to use. Here it is uio\_pci\_generic. Another popular option is UIO driver is igb\_uio.
* line 36: HUGE\_PAGES defines number of huge pages. Here we allocate 32000 huge pages. considering page size 2M it will be 64G memory usage in total. free -h command output in compute node will confirm this
* line 37: agent mode set to dpdk
* line 38-50: the second DPDK vRouter on server a7s4
* line 51-59: defines the third vRouter, this is a kernal based one, so we don’t need any DPDK specific parameters
* line 60-85: contrail\_configuration section contains parameters for Contrail services
* line 61-62: Contrail and OpenStack versions
* line 63: the cloud orchestrator. It can be OpenStack or vcenter. our setup is with OpenStack only.
* line 64-66: who is the controller node. In our setup both OpenStack and contrail controllers are installed in same node
* line 71, 76: these are the two virtual IPs configured
* line 80-82: these are needed only for lab setup. Without these parameters, contrail-status command will print warning to indicate that the storage space is not big enough
* line 86-98: the parameters for Kolla
* line 87-94: refers to OpenStack service
* line 88-89: VIPs configured for management and data control network respectively. One usage of these VIPs is to make it possible to access the OpenStack horizon service (webUI) from management network, by default all OpenStack services listen on the IP in data/ctrl network. With these VIPs configured and used by keepalived, HAproxy can forward the access request coming from the management network to the Horizon service.

## installation Steps

Once the YAML file is carefully prepared, the installation process is relatively easy. Basically you select one node as the deployment node, the node from where you want to automate the installation of all other nodes. In practice, use the controller node as the deployment node.

In this node you need to install some pre-requisite software, such as python libraries, ansible, git, etc., and the python modules (python-wheel) that are used by Ansible, and ansible is our automation tool. git is used to clone a github repository which includes all Ansible playbooks. Then you use Ansible to automate the software installation in all the nodes based on the playbook and your configuration file instances.yaml. The details start here:

1. Install pre-requisite packages on deployment node, in this case, it’s the controller a2s2:

* 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

2. Use git to clone install the Ansible deployer folder into deployment node:

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

3. Place the prepared configuration file instances.yaml to tf-ansible-deployer/config:

4. Install Contrail with Ansible:

* 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

5. Install the OpenStack client:

* pip install --ignore-installed python-openstackclient python-ironicclient openstack-heat

After everything loads, you will have an up and running 4-node Contrail cluster (one controller node and three vRouter/compute nodes). You can log in to the setup via a webUI or ssh session to check system status.

## Post-installation Verification

Here is the Contrail web UI for a working setup in Figure 5.2.

![contrail-web](data:image/png;base64;base64,)

Figure 5.2 Contrail WebUI Dashboard

You can also log in to each individual node with ssh, and the run contrail-status command to verify the running status of all the components as shown in Figure 5.3.

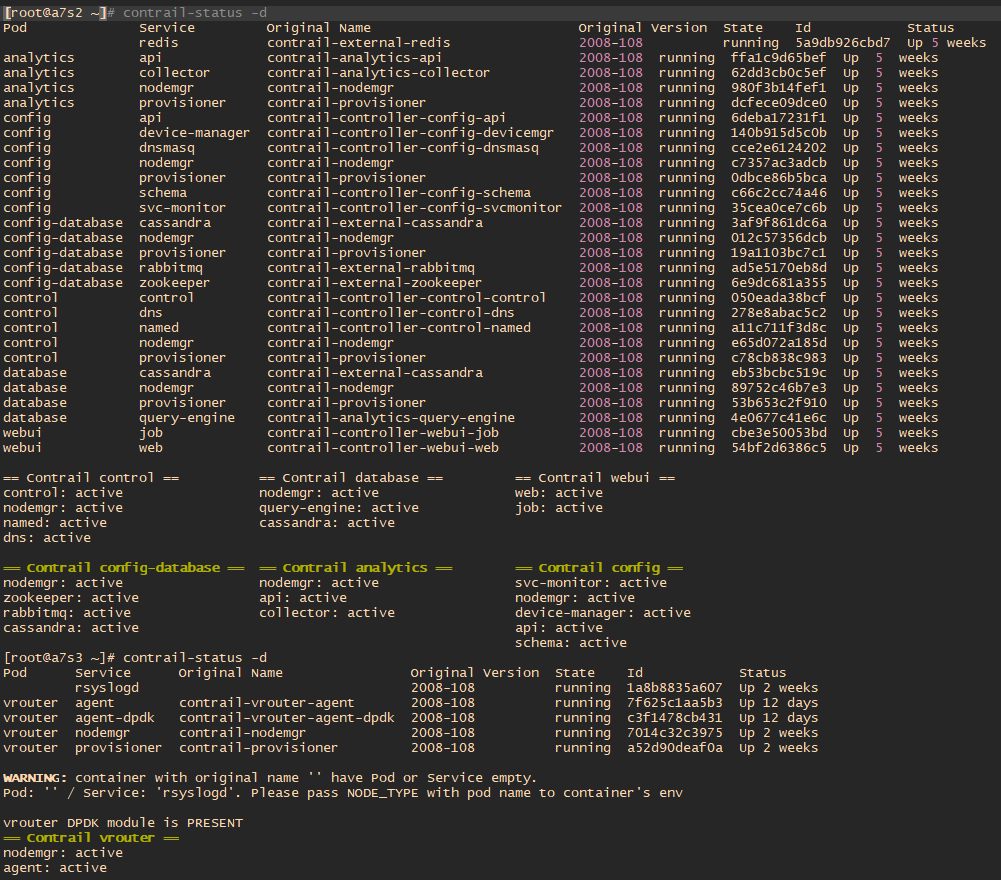


Figure 5.3 contrail-status command output

If everything works, congratulations! You now have your own lab to play in.

Now let’s go over the steps of setting up testing tools to send and receive traffic - the PROX and rapid script.

# DPDK vRouter Test Tools

**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 supports 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. This chapter introduces you to how to use it to test vRouter performance in DPDK environments.

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 (swap VM, or loop VM)

This book calls them *gen* and *swap* VM respectively. One special feature used here is that the swap PROX is configured in such a way that once it 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 this case it’s the DPDK vRouter.

**Rapid**

Rapid (Rapid Automated Performance Indication for Dataplane) is a group of wrapper scripts interacting with PROX to simplify and automate the configuration of PROX. It’s a set of files and scripts offering an even easier way to do sanity checks of the data plane performance.

Rapid is both powerful and configurable. A typical workflow works as follows:

* A script name runrapid.py will send the proper configuration files to the gen and swap VMs involved in the testing, so each one 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 visible onscreen and logged in the log and csv files.
* The same tests will be done for different packet sizes and different amounts of flows, under certain latency and packet drop rates.

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, and pause the test, as well as collect statistics.

**PROX and Rapid Test Setup**

A typical PROX and Rapid testing setup looks like Figure 5.4.

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

Figure 5.4 PROX and Rapid Test Diagram

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

* PROX generated 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 generation and collecting results.

**Hardware Requirements**

Here’s a brief summary of hardware requirements for different VMs:

* Swap VM: This is where the DUT (vRouter) is located. Based on the test requirements, 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 DUTs, the traffic generator VM and the compute should allocate much more CPU resources than the DUT.
* Jump VM: No high speed VM is required, and it can be run on the kernel or the 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 a practical environment.

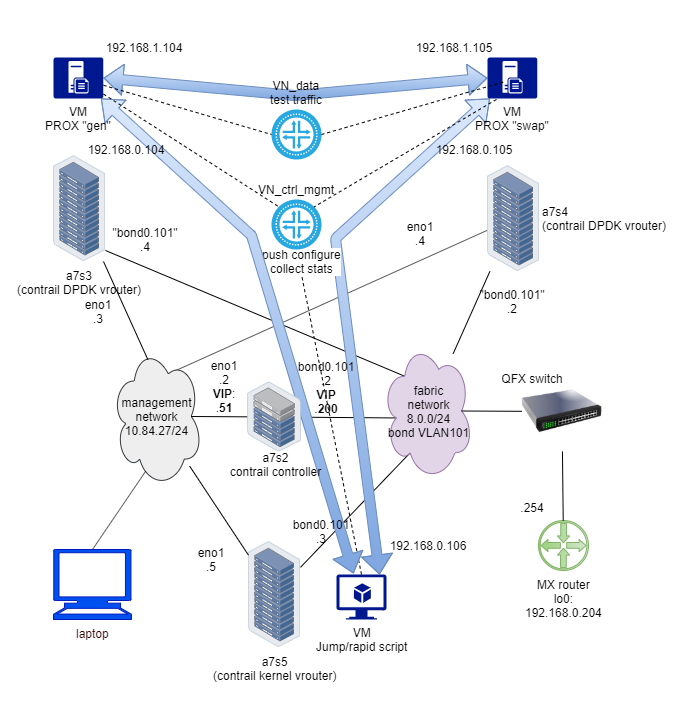


Figure 5.5 The PROX Setup in the Book’s Lab

By default, multi-queue is enabled on both the PROX gen and swap VMs via OpenStack image property. You can refer to Chapter 3 for more details about the multi-queue feature and its configurations. Additionally, rapid scripts also provide CPU pinning to protect PROX PMDs against CPU-stealing by other processes and the VM OS.

## Installation

**Creating OpenStack resources.**

As mentioned earlier, to perform the test we need two VMs both running PROX. One sending traffic and the other receiving and swapping it back. The same exact PROX application is running but here with different configuration files.

Apparently, the IP level connectivity is required in order for the two VMs to be able to exchange packets with each other. In this case, the two VMs will be spawned by OpenStack Nova. Needless to say, all supporting objects and resources associated 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 CLIs. A quick list of 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 compiling it on 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 you on how PROX is installed in our setup we built for this book.

MORE? You can find more details in PROX website here: <https://wiki.opnfv.org/display/SAM/PROX+installation>.

The software and CPU model used here are:

[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)

NOTE: There is a good chance that your servers and VMs have totally different hardware and software architectures. The steps here are tested and working fine in the book’s lab setup, but depending on your environment, you may run into some errors. Check PROX online documentation for more detailed instructions.

**Compiling and Building the DPDK**

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

You can either build it inside of the VM where it runs or build it directly in the host environment where the VM got spawned and copy it into the VM. The steps to build the DPDK in our setup follows.

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

NOTE The stable and recommended version of DPDK at the time of writing this book is 19.11.

**Compiling PROX**

Now with the DPDK libraries built, let’s 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 the build folder of the current directory. (We’ll demonstrate this shortly.)

**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 that the current directory is where you’ve just built PROX, you 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 updates about the running states in real time. We’ll give an example on this shortly.

**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 with no need to compile.

## Installation: Heat Automation

There are the steps of manually compiling PROX from source code.

Now here’s a list of tasks to create all the necessary objects required by the VMs from OpenStack. Doing this one time is not a big deal but 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 tedious and even painful. To simplify the building, creation, and configuration of PROX, as well as creating all necessary OpenStack resources, the number one choice for automation is *heat*. With *heat* all tasks are typically programmed in a template file, which calls all parameters from another environment file. In this book’s Appendix, we provide all sample template files as well as environment file and associated scripts, which are tested and proven to be working fine, at least in our setup. You can use them as a starting point, then make necessary customizations based on your environment to build your own automation. The VM, where the tools are running, including rapid scripts and PROX DPDK applications pre-compiled on it, has also been built as an image .

With all these automations carefully designed and tested, 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 the previous section.

Adjust the heat template, environment variables, and automation scripts based on your environment: [[1]](#footnote-1)( These files are available in this [github repository](https://github.com/damjanek/dpdk-prox-contrail).)

* + - environment.yaml
    - build-rapid.yml
    - configure.rapid.sh

2. Load rapid image into OpenStack 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

3. (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

4. Create heat stack with the prepared yaml files:

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

Wait for a few minutes and use the openstack stack list command to check the stack creation progress shown in Figure 5.6.

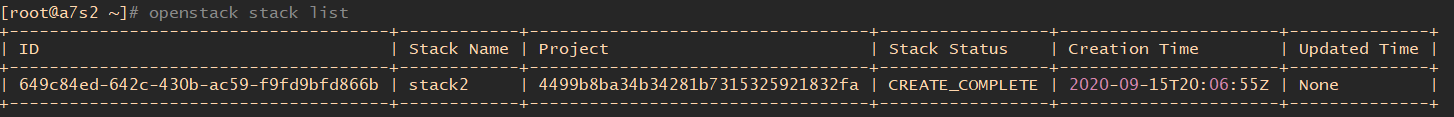


Figure 5.6 OpenStack Stack List

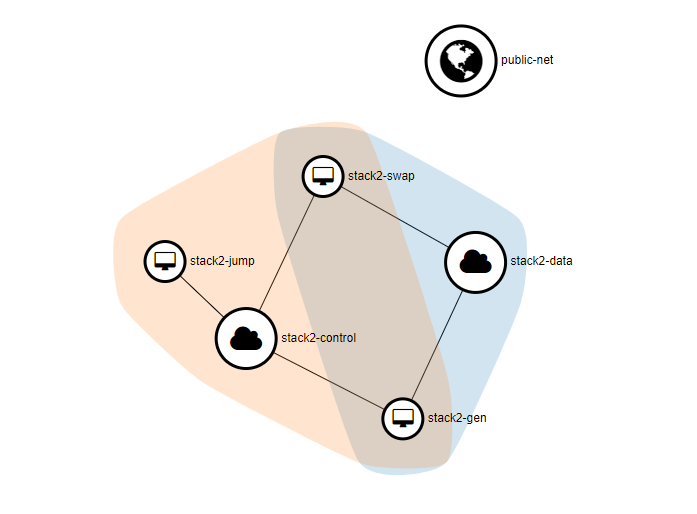


Figure 5.7 OpenStack Topology (graph)

5. Once loaded you can use these different sub-commands of the 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

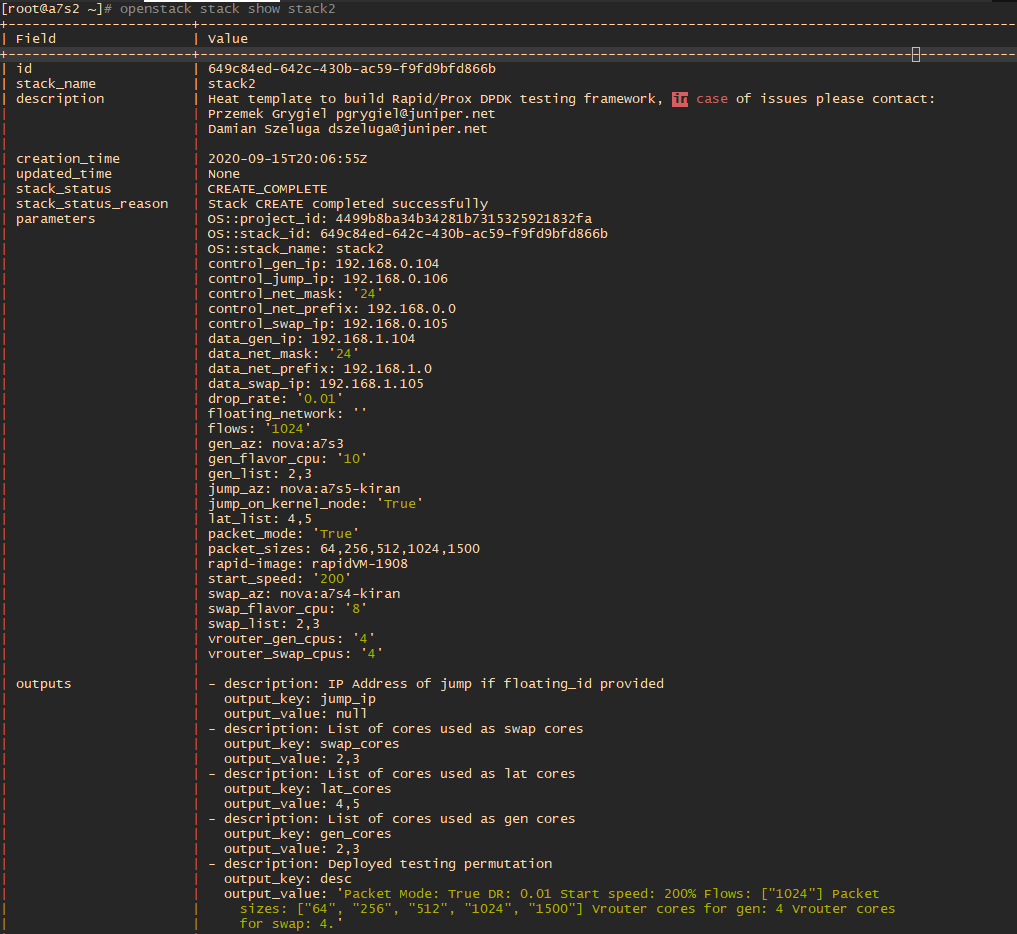


Figure 5.8 OpenStack Stack Show STACK

**Login to the VMs.**

So all three VMs, once up and running, will inherit the same log in credentials defined in the heat template and scripts. There are a few common ways to access a VM running in a specific compute node in Contrail/OpenStack integration environments:

* floating IP: This is a routable IP address that is visible from outside of the cluster which maps to an internal IP of the VM. Once the VM is launched, you can log in to a specific VMs 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 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 the VM console use the 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 the console, ssh session is usually preferred. Let’s take a look at each VM’s allocated interface IPs with openstack server list command:

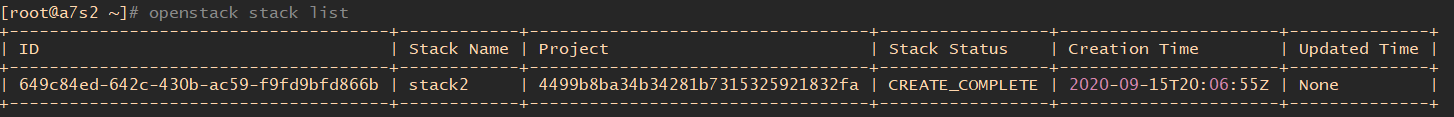


Figure 5.9 OpenStack Server List

Let’s examine our jump VM stack2-jump for a moment. OpenStack allocated an IP address 192.168.0.106 to its 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, you need to first locate the meta\_ip\_address allocated to the VM’s tap interface, or more specifically, the vif interface in the vRouter. You 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, where *N* is the same number as the number in the interface vif0/N. Therefore in this case, the 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 it also supports file copies with the scp tool. Remember, in many cases the VM does not have 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 can now talk about how to run the test with rapid. Remember rapid is installed in the jump VM, so we’ll need to execute the script from there.

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

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

This is a symbolic link, by default this rapid folder links to: /opt/openstackrapid/samplevnf/VNFs/DPPD-PROX/helper-scripts/rapid/.

This will start rapid script and send traffic for ten seconds by default. the period of time for sending traffic can be adjusted by the --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 Figure 5.10.

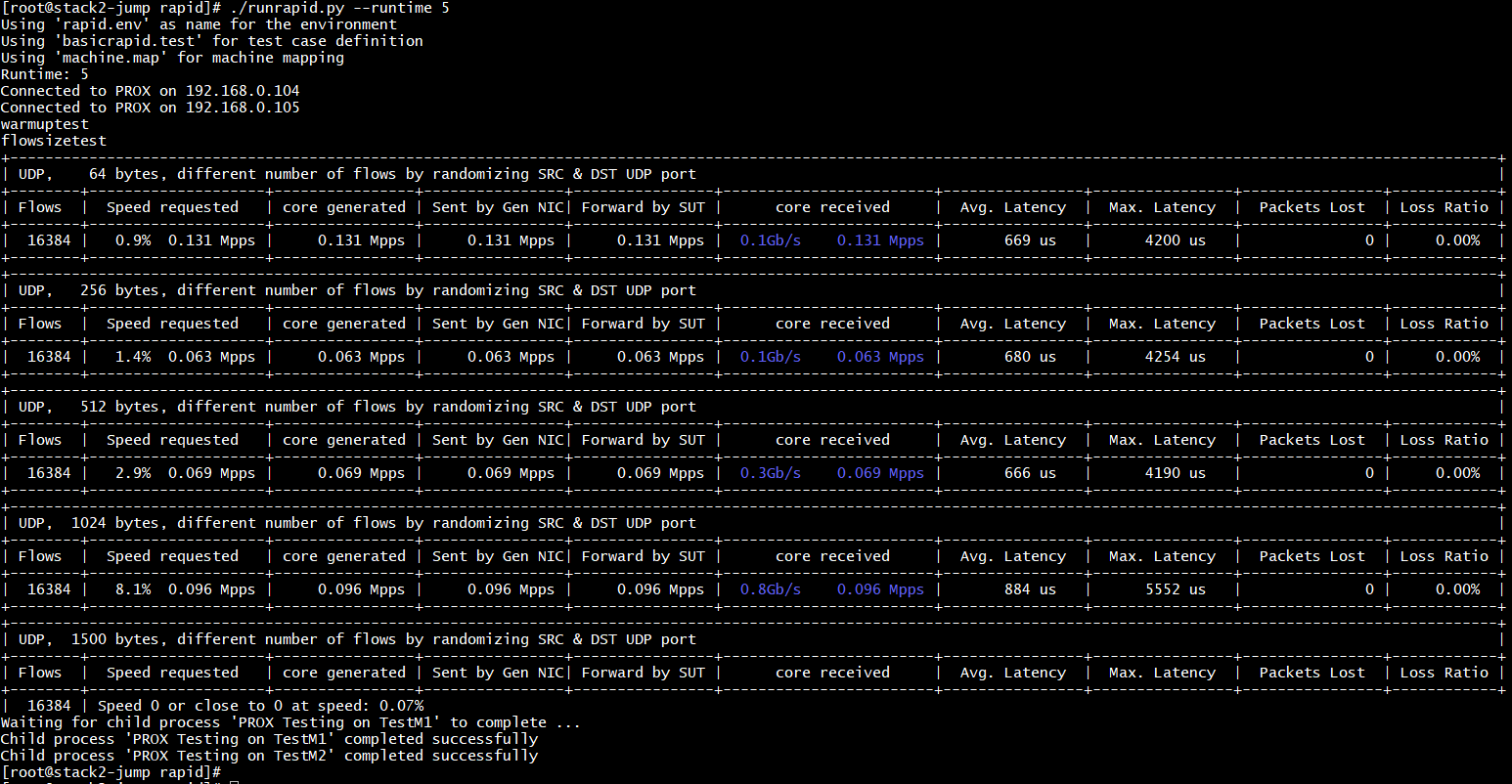


Figure 5.10 Runrapid.py Script Results

You can see that some preparation work was done before the actual test was started. First, the script read three 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 starts some small amount of traffic as warmup, to test the reachability between the source and destination, and also populate the MAC table or ARP table in devices along the path.

3. When everything is ready, the script starts the traffic and at the same time monitors the traffic’s 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 the two systems within a given allowed packet loss and accuracy which are defined in the \*.test files (for example, the basicrapid.test file for a simple test).

The script is highly configurable. The Appendix provides a sample basicrapid.test used in our lab. You can start with it and fine tune based on your needs. For example, in section [test2] of the file you can change the rate 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, 65536, 131072, 262144, 524280, 1048576  
flows=[16384, 65536]

## Run PROX Manually

Okay, we just introduced rapid. The script supports very extensive options in the configuration files which are beyond the scope of this book, but you should have the idea how it basically works. Please remember that rapid and PROX and two different applications. A rapid script does all the magic and make your life easier through the automation of PROX, and PROX is the foundation application that does the real work. 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) run 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 you can learn its booting sequences:

* setup 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 that shown in Figure 5.11.

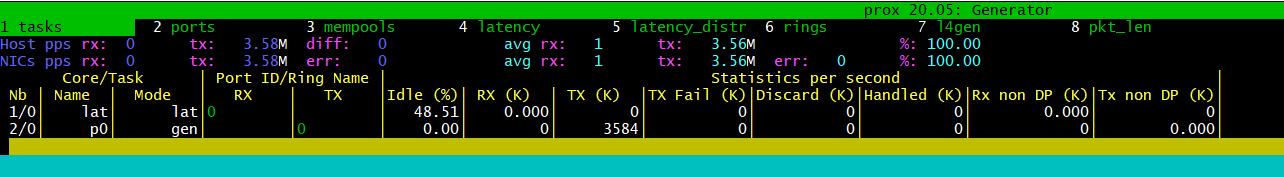


Figure 5.11 gen running UI

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, and more. Refer to the PROX website for more details. For now, let’s look at how the traffic flows.

Right now from Figure 5.11 you can 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. You need to start the swap VM as well as a receiver, which 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 the swap VM is installed, execute the same prox command line, except this time pass a different configuration file named swap.cfg. See Figure 5.12.

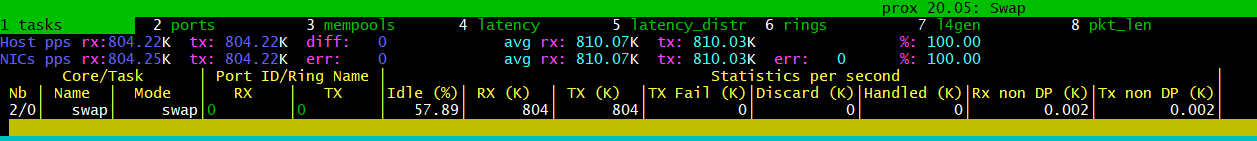


Figure 5.12 Swap Running UI

You will end up with a similar ncurse-based text UI, after a similar booting process of the sender. Once the swap end of PROX is up and running, you will immediately see both RX and TX counters (Figure 5.13) keep updating on both sides of the traffic.

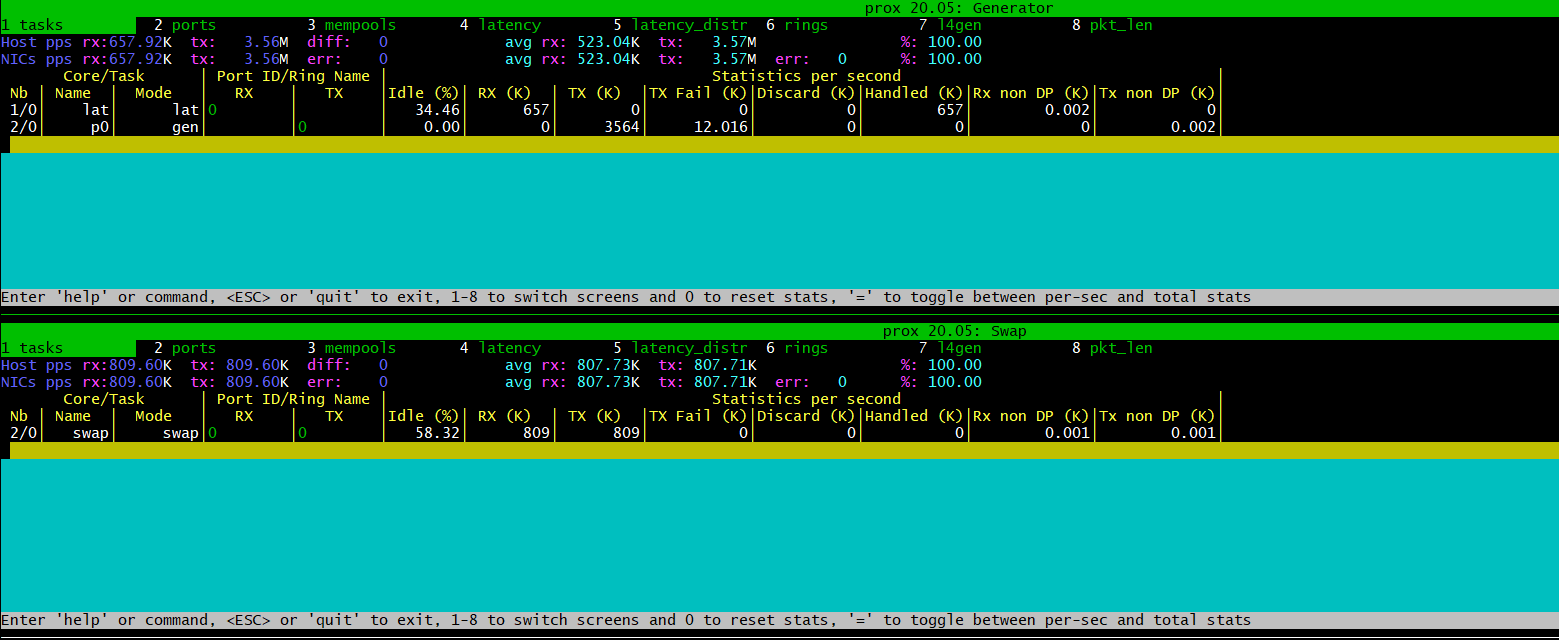


Figure 5.13 Gen and Swap UI

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















1. [↑](#footnote-ref-1)