NFVBENCH

Release Latest

Nov 09, 2018

Contents

CHAPTER 1

Introduction

NFVbench is a python application that is designed to run in a compact and portable format inside a container and on production pods. As such it only uses open sourec software with minimal hardware requirements (just a NIC card that is DPDK compatible). Traffic generation is handled by TRex on 2 physical ports (2x10G or higher) forming traffic loops up to VNF level and following a path that is common to all NFV applications: external source to top of rack switch(es) to conpute node(s) to vswitch (if applicable) to VNF(s) and back.

Configuration of benchmarks is through a hierarchy of yaml configuraton files and command line arguments.

Results are available in different formats: - text output with tabular results - json result in file or in REST reply (most detailed)

Logging is available in a log file.

Benchmark results and logs can be optionally sent to one or more remote fluentd aggeregators using json format.

CHAPTER 2

OPNFV NFVbench Euphrates Design

2.1 Introduction

NFVbench can be decomposed in the following components: - Configuration - Staging - Traffic generation - Traffic generator results analysis

2.2 Configuration

This component is in charge of getting the configuration options from the user and consolidate them with the default configuration into a running configuration.

default configuration + user configuration options = running configuration

User configuration can come from: - CLI configuration shortcut arguments (e.g –frame-size) - CLI configuration file (–config [file]) - CLI configuration string (–config [string]) - REST request body - custom platform pluging

The precedence order for configuration is (from highest precedence to lowest precedence) - CLI confguration or REST configuration - custom platform plugin - default configuration

The custom platform plugin is an optional python class that can be used to override default configuration options with default platform options which can be either hardcoded or calculated at runtime from platform specific sources

(such as platform deployment configuration files). A custom platform plugin class is a child of the parent class nfvbench.config_plugin.ConfigPlugin.

2.3 Staging

The staging component is in charge of staging the OpenStack resources that are used for the requested packet path. For example, for a PVP packet path, this module will create 2 Neutron networks and one VM instance connected to these 2 networks. Multi-chaining and VM placement is also handled by this module.

Main class: nfvbench.chaining.ChainManager

2.4 Traffic Generation

The traffic generation component is in charge of contrilling the TRex traffic generator using its python API. It includes tasks such as: - traffic check end to end to make sure the packet path is clear in both directions before starting a benchmark - programming the Trex traffic flows based on requested parameters - fixed rate control - NDR/PDR binary search

Main class: nfvbench.traffic_client.TrafficClient

2.5 Traffic Generator Results Analysis

At the end of a traffic generation session, this component collects the results from TRex and packages them in a format that is suitable for the various output formats (JSON, REST, file, fluentd). In the case of multi-chaining, it handles aggregation of results across chains.

Main class: nfvbench.stats_manager.StatsManager

2.6 Versioning

NFVbench uses semver compatible git tags such as “1.0.0”. These tags are also called project tags and applied at important commits on the master branch exclusively. Rules for the version numbers follow the semver 2.0 specification (https://semver.org). These git tags are applied indepently of the OPNFV release tags which are applied only on the stable release branches (e.g. “opnfv-5.0.0”).

In general it is recommeneded to always have a project git version tag associated to any OPNFV release tag content obtained from a sync from master.

NFVbench Docker containers will be versioned based on the OPNF release tags or based on NFVbench project tags.

2.7 Traffic Description

The general packet path model followed by NFVbench requires injecting traffic into an arbitrary number of service chains, where each service chain is identified by 2 edge networks (left and right). In the current multi-chaining model:

- all service chains share the same left and right edge networks
- each port associated to the traffic generator is dedicated to send traffic to one edge network

In an OpenStack deployment, this corresponds to all chains sharing the same 2 neutron networks. If VLAN encapsulation is used, all traffic sent to a port will have the same VLAN id.

2.7.1 Basic Packet Description

The code to create the UDP packet is located in TRex.create_pkt() (nfvbench/traffic_gen/trex.py).

NFVbench always generates UDP packets (even when doing L2 forwarding). The final size of the frame containing each UDP packet will be based on the requested L2 frame size. When taking into account the minimum payload size requirements from the traffic generator for the latency streams, the minimum L2 frame size is 64 byte (no vlan tagging) or 68 bytes (with vlan tagging).

2.7.2 Flows Specification

Mac Addresses

The source MAC address is always the local port MAC address (for each port). The destination MAC address is based on the configuration and can be:

- the traffic generator peer port MAC address in the case of L2 loopback at the switch level or when using a loopback cable
- the dest MAC as specified by the configuration file (EXT chain no ARP)
- the dest MAC as discovered by ARP (EXT chain)
- the VM MAC as dicovered from Neutron API (PVP, PVVP chains)

NFVbench does not currently range on the MAC addresses.

IP addresses

The source IP address is fixed per chain. The destination IP address is variable within a distinct range per chain.

UDP ports

The source and destination ports are fixed for all packets and can be set in the configuratoon file (default is 53).

Payload User Data

The length of the user data is based on the requested L2 frame size and takes into account the size of the L2 header - including the VLAN tag if applicable.

2.7.3 IMIX Support

In the case of IMIX, each direction is made of 4 streams: - 1 latency stream - 1 stream for each IMIX frame size

The IMIX ratio is encoded into the number of consecutive packets sent by each stream in turn.

2.7.4 Service Chains and Streams

A stream identifies one "stream" of packets with same characteristics such as rate and destination address. NFVbench will create 2 streams per service chain per direction:

- 1 latency stream set to 1000pps
- 1 main traffic stream set to the requested Tx rate less the latency stream rate (1000pps)

For example, a benchmark with 1 chain (fixed rate) will result in a total of 4 streams. A benchmark with 20 chains will results in a total of 80 streams (fixed rate, it is more with IMIX).

The overall flows are split equally between the number of chains by using the appropriate destination MAC address.

For example, in the case of 10 chains, 1M flows and fixed rate, there will be a total of 40 streams. Each of the 20 non-latency stream will generate packets corresponding to 50,000 flows (unique src/dest address tuples).

2.8 NDR/PDR Binary Search

The NDR/PDR binary search algorithm used by NFVbench is based on the algorithm used by the FD.io CSIT project, with some additional optimizations.

2.8.1 Algorithm Outline

The ServiceChain class (nfvbench/service_chain.py) is responsible for calculating the NDR/PDR or all frame sizes requested in the configuration. Calculation for 1 frame size is delegated to the TrafficClient class (nfvbench/traffic_client.py)

Call chain for calculating the NDR-PDR for a list of frame sizes:

- **ServiceChain.run()**
  - **ServiceChain._get_chain_results()**
    - **for every frame size:**
      - **ServiceChain.__get_result_per_frame_size()**

        TrafficClient.get_ndr_pdr()

        TrafficClient.__range_search() recursive binary search

The search range is delimited by a left and right rate (expressed as a % of line rate per direction). The search always start at line rate per port, e.g. in the case of 2x10Gbps, the first iteration will send 10Gbps of traffic on each port.

The load_epsilon configuration parameter defines the accuracy of the result as a % of line rate. The default value of 0.1 indicates for example that the measured NDR and PDR are within 0.1% of line rate of the actual NDR/PDR (e.g. 0.1% of 10Gbps is 10Mbps). It also determines how small the search range must be in the binary search. Smaller values of load_epsilon will result in more iterations and will take more time but may not always be beneficial if the absolute value falls below the precision level of the measurement. For example a value of 0.01% would translate to an absolute value of 1Mbps (for a 10Gbps port) or around 10kpps (at 64 byte size) which might be too fine grain.

The recursion narrows down the range by half and stops when:

- the range is smaller than the configured load_epsilon value
- or when the search hits 100% or 0% of line rate

2.8.2 Optimization

Binary search algorithms assume that the drop rate curve is monotonically increasing with the Tx rate. To save time, the algorithm used by NFVbench is capable of calculating the optimal Tx rate for an arbitrary list of target maximum drop rates in one pass instead of the usual 1 pass per target maximum drop rate. This saves time linearly to the number target drop rates. For example, a typical NDR/PDR search will have 2 target maximum drop rates:

- NDR = 0.001%
- PDR = 0.1%

The binary search will then start with a sorted list of 2 target drop rates: [0.1, 0.001]. The first part of the binary search will then focus on finding the optimal rate for the first target drop rate (0.1%). When found, the current target drop rate is removed from the list and iteration continues with the next target drop rate in the list but this time starting from the upper/lower range of the previous target drop rate, which saves significant time. The binary search continues until the target maximum drop rate list is empty.

2.8.3 Results Granularity

The binary search results contain per direction stats (forward and reverse). In the case of multi-chaining, results contain per chain stats. The current code only reports aggregated stats (forward + reverse for all chains) but could be enhanced to report per chain stats.

2.8.4 CPU Limitations

One particularity of using a software traffic generator is that the requested Tx rate may not always be met due to resource limitations (e.g. CPU is not fast enough to generate a very high load). The algorithm should take this into consideration:

- always monitor the actual Tx rate achieved as reported back by the traffic generator
- actual Tx rate is always <= requested Tx rate
- the measured drop rate should always be relative to the actual Tx rate
- if the actual Tx rate is < requested Tx rate and the measured drop rate is already within threshold (<NDR/PDR threshold) then the binary search must stop with proper warning because the actual NDR/PDR might probably be higher than the reported values

CHAPTER 3

NFVbench Release Notes

3.1 RELEASE NOTES

3.1.1 Release 2.0

Major release highlights:

- Dedicated chain networks
- VxLAN support with VTEP in the traffic generator
- Enhanced chain analysis
- Code refactoring and enhanced unit testing
- Miscellaneous enhancement

Dedicated chain networks

NFVbench 1.x only supported shared networks across chains. For example, 20xPVP would create only 2 networks (left and right) shared by all chains. With NFVbench 2.0, chain networks will become dedicated (unshared) by default with an option in the nfvbench configuration to shared them. A 20xPVP run will create 2x20 networks instead.

Enhanced chain analysis

The new chain analysis improves at multiple levels:

- there is now one table for each direction (forward and reverse) that both read from left to right
- per-chain packet counters and latency
- all-chain aggregate packet counters and latency
- supports both shared and dedicated chain networks

Code refactoring and enhanced unit testing

The overall code structure is now better partitioned in the following functions:

- staging and resource discovery
- traffic generator
- stats collection

The staging algorithm was rewritten to be:

- a lot more robust to errors and to handle better resource reuse use cases. For example when a network with a matching name is discovered the new code will verify that the network is associated to the right VM instance
- a lot more strict when it comes to the inventory of MAC addresses. For example the association from each VM MAC to a chain index for each Trex port is handled in a much more strict manner.

Although not all code is unit tested, the most critical parts are unit tested with the use of the mock library. The resulting unit test code can run in isolation without needing a real system under test.

3.1.2 OPNFV Fraser Release

Over 30 Jira tickets have been addressed in this release (Jira NFVBENCH-55 to NFVBENCH-78)

The Fraser release adds the following new features:

- support for benchmarking non-OpenStack environments (with external setup and no OpenStack openrc file)
- PVVP packet path with SRIOV at the edge and vswitch between VMs
- support logging events and results through fluentd

Enhancements and main bug fixes:

- end to end connectivity for larger chain count is now much more accurate for large chain count - avoiding excessive drops
- use newer version of TRex (2.32)
- use newer version of testpmd DPDK
- NDR/PDR uses actual TX rate to calculate drops - resulting in more accurate results
- add pylint to unit testing
- add self sufficient and standalone unit testing (without actual testbed)

3.1.3 OPNFV Euphrates Release

This is the introductory release for NFVbench. In this release, NFVbench provides the following features/capabilities:

- standalone installation with a single Docker container integrating the open source TRex traffic generator
- can measure data plane performance for any NFVi full stack
- **can setup automatically service chains with the following packet paths:**
  - PVP (physical-VM-physical)
  - PVVP (physical-VM-VM-physical) intra-node and inter-node
- **can setup multiple service chains**
  - N * PVP

    - N * PVVP
- supports any external service chain (pre-set externally) that can do basic IPv4 routing
- **can measure**
    - drop rate and latency for any given fixed rate
    - NDR (No Drop Rate) and PDR (Partial Drop Rate) with configurable drop rates
- **traffic specification**
    - any fixed frame size or IMIX
    - uni or bidirectional traffic
    - any number of flows
    - vlan tagging can be enabled or disabled
- **user interface:**
    - CLI
    - REST+socketIO
- fully configurable runs with yaml-JSON configuration
- detailed results in JSON format
- summary tabular results
- can send logs and results to one or more fluentd aggregators (per configuration)

CHAPTER 4

Developer Guide

CHAPTER 5

Configuration Guide

CHAPTER 6

NFVbench User Guide

The NFVbench tool provides an automated way to measure the network performance for the most common data plane packet flows on any OpenStack system. It is designed to be easy to install and easy to use by non experts (no need to be an expert in traffic generators and data plane performance testing).

6.1 Table of Content

6.1.1 Features

Data Plane Performance Measurement Features

NFVbench supports the following main measurement capabilities:

- **supports 2 measurement modes:**
  - *fixed rate* mode to generate traffic at a fixed rate for a fixed duration
  - NDR (No Drop Rate) and PDR (Partial Drop Rate) measurement mode
- configurable frame sizes (any list of fixed sizes or 'IMIX')
- built-in packet paths (PVP, PVVP)
- built-in loopback VNFs based on fast L2 or L3 forwarders running in VMs
- configurable number of flows and service chains
- configurable traffic direction (single or bi-directional)

NDR is the highest throughput achieved without dropping packets. PDR is the highest throughput achieved without dropping more than a pre-set limit (called PDR threshold or allowance, expressed in %).

Results of each run include the following data:

- Aggregated achieved throughput in bps
- Aggregated achieved packet rate in pps (or fps)

- Actual drop rate in %
- Latency in usec (min, max, average in the current version)

Built-in OpenStack support

NFVbench can stage OpenStack resources to build 1 or more service chains using direct OpenStack APIs. Each service chain is composed of:

- 1 or 2 loopback VM instances per service chain
- 2 Neutron networks per loopback VM

OpenStack resources are staged before traffic is measured using OpenStack APIs (Nova and Neutron) then disposed after completion of measurements.

The loopback VM flavor to use can be configured in the NFVbench configuration file.

Note that NFVbench does not use OpenStack Heat nor any higher level service (VNFM or NFVO) to create the service chains because its main purpose is to measure the performance of the NFVi infrastructure which is mainly focused on L2 forwarding performance.

External Chains

NFVbench supports settings that involve externally staged packet paths with or without OpenStack:

- run benchmarks on existing service chains at the L3 level that are staged externally by any other tool (e.g. any VNF capable of L3 routing)
- run benchmarks on existing L2 chains that are configured externally (e.g. pure L2 forwarder such as DPDK testpmd)

Direct L2 Loopback (Switch or wire loopback)

NFVbench supports benchmarking of pure L2 loopbacks (see "–l2-loopback vlan" option)

- Switch level loopback
- Port to port wire loopback

In this mode, NFVbench will take a vlan ID and send packets from each port to the other port (dest MAC set to the other port MAC) using the same VLAN ID on both ports. This can be useful for example to verify that the connectivity to the switch is working properly.

Traffic Generation

NFVbench currently integrates with the open source TRex traffic generator:

- TRex (pre-built into the NFVbench container)

Supported Packet Paths

Packet paths describe where packets are flowing in the NFVi platform. The most commonly used paths are identified by 3 or 4 letter abbreviations. A packet path can generally describe the flow of packets associated to one or more service chains, with each service chain composed of 1 or more VNFs.

The following packet paths are currently supported by NFVbench:

- PVP (Physical interface to VM to Physical interface)
- PVVP (Physical interface to VM to VM to Physical interface)
- N*PVP (N concurrent PVP packet paths)
- N*PVVP (N concurrent PVVP packet paths)

The traffic is made of 1 or more flows of L3 frames (UDP packets) with different payload sizes. Each flow is identified by a unique source and destination MAC/IP tuple.

Loopback VM

NFVbench provides a loopback VM image that runs CentOS with 2 pre-installed forwarders:

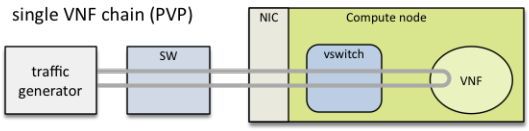
- DPDK testpmd configured to do L2 cross connect between 2 virtual interfaces
- FD.io VPP configured to perform L3 routing between 2 virtual interfaces

Frames are just forwarded from one interface to the other. In the case of testpmd, the source and destination MAC are rewritten, which corresponds to the mac forwarding mode (–forward-mode=mac). In the case of VPP, VPP will act as a real L3 router, and the packets are routed from one port to the other using static routes.

Which forwarder and what Nova flavor to use can be selected in the NFVbench configuration. Be default the DPDK testpmd forwarder is used with 2 vCPU per VM. The configuration of these forwarders (such as MAC rewrite configuration or static route configuration) is managed by NFVbench.

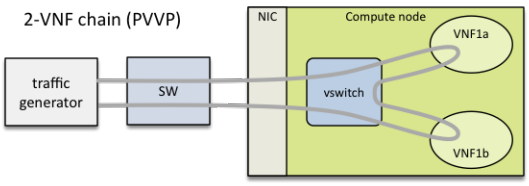
PVP Packet Path

This packet path represents a single service chain with 1 loopback VNF and 2 Neutron networks:

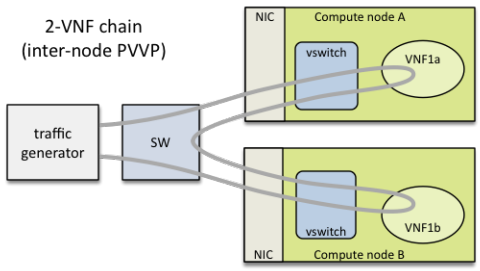


PVVP Packet Path

This packet path represents a single service chain with 2 loopback VNFs in sequence and 3 Neutron networks. The 2 VNFs can run on the same compute node (PVVP intra-node):



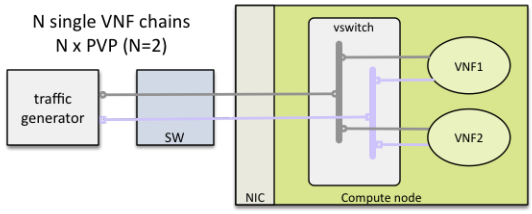
or on different compute nodes (PVVP inter-node) based on a configuration option:



Multi-Chaining (N*PVP or N*PVVP)

Multiple service chains can be setup by NFVbench without any limit on the concurrency (other than limits imposed by available resources on compute nodes). In the case of multiple service chains, NFVbench will instruct the traffic generator to use multiple L3 packet streams (frames directed to each path will have a unique destination MAC address).

Example of multi-chaining with 2 concurrent PVP service chains:



This innovative feature will allow to measure easily the performance of a fully loaded compute node running multiple service chains.

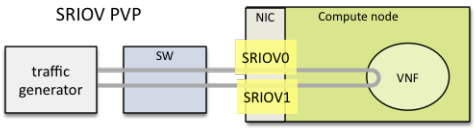
Multi-chaining is currently limited to 1 compute node (PVP or PVVP intra-node) or 2 compute nodes (for PVVP inter-node). The 2 edge interfaces for all service chains will share the same 2 networks. The total traffic will be split equally across all chains.

SR-IOV

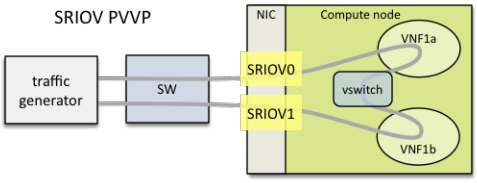
By default, service chains will be based on virtual switch interfaces.

NFVbench provides an option to select SR-IOV based virtual interfaces instead (thus bypassing any virtual switch) for those OpenStack system that include and support SR-IOV capable NICs on compute nodes.

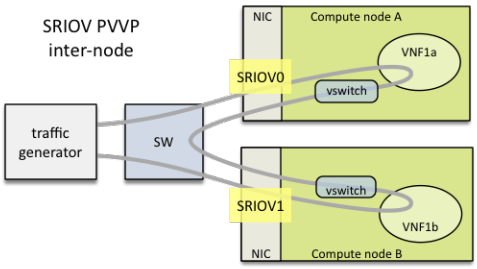
The PVP packet path will bypass the virtual switch completely when the SR-IOV option is selected:



The PVVP packet path will use SR-IOV for the left and right networks and the virtual switch for the middle network by default:

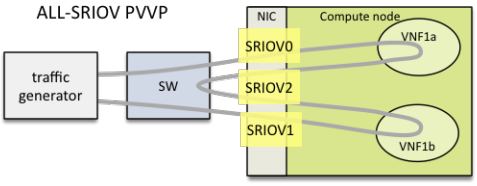


Or in the case of inter-node:

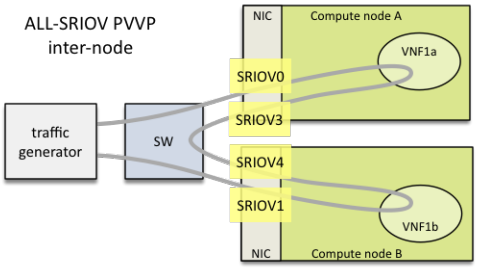


This packet path is a good way to approximate VM to VM (V2V) performance (middle network) given the high efficiency of the left and right networks. The V2V throughput will likely be very close to the PVVP throughput while its latency will be very close to the difference between the SR-IOV PVVP latency and the SR-IOV PVP latency.

It is possible to also force the middle network to use SR-IOV (in this version, the middle network is limited to use the same SR-IOV phys net):



The chain can also span across 2 nodes with the use of 2 SR-IOV ports in each node:



Other Misc Packet Paths

P2P (Physical interface to Physical interface - no VM) can be supported using the external chain/L2 forwarding mode.

V2V (VM to VM) is not supported but PVVP provides a more complete (and more realistic) alternative.

Supported Neutron Network Plugins and vswitches

Any Virtual Switch, Any Encapsulation

NFVbench is agnostic of the virtual switch implementation and has been tested with the following virtual switches:

- ML2/VPP/VLAN (networking-vpp)
- OVS/VLAN and OVS-DPDK/VLAN
- ML2/ODL/VPP (OPNFV Fast Data Stack)

6.1.2 Limitations

NFVbench only supports VLAN with OpenStack. NFVbench does not support VxLAN overlays.

6.1.3 Installation and Quick Start Guides

Requirements for running NFVbench

Hardware Requirements

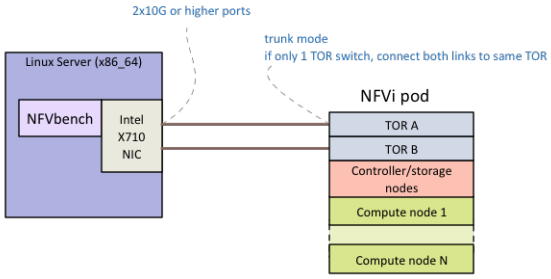
To run NFVbench you need the following hardware: - a Linux server - a DPDK compatible NIC with at least 2 ports (preferably 10Gbps or higher) - 2 ethernet cables between the NIC and the OpenStack pod under test (usually through a top of rack switch)

The DPDK-compliant NIC must be one supported by the TRex traffic generator (such as Intel X710, refer to the Trex Installation Guide for a complete list of supported NIC)

To run the TRex traffic generator (that is bundled with NFVbench) you will need to wire 2 physical interfaces of the NIC to the

- if you have only 1 TOR, wire both interfaces to that same TOR

- 1 interface to each TOR if you have 2 TORs and want to use bonded links to your compute nodes



Switch Configuration

The 2 corresponding ports on the switch(es) facing the Trex ports on the Linux server should be configured in trunk mode (NFVbench will instruct TRex to insert the appropriate vlan tag).

Using a TOR switch is more representative of a real deployment and allows to measure packet flows on any compute node in the rack without rewiring and includes the overhead of the TOR switch.

Although not the primary targeted use case, NFVbench could also support the direct wiring of the traffic generator to a compute node without a switch.

Software Requirements

You need Docker to be installed on the Linux server.

TRex uses the DPDK interface to interact with the DPDK compatible NIC for sending and receiving frames. The Linux server will need to be configured properly to enable DPDK.

DPDK requires a uio (User space I/O) or vfio (Virtual Function I/O) kernel module to be installed on the host to work. There are 2 main uio kernel modules implementations (igb_uio and uio_pci_generic) and one vfio kernel module implementation.

To check if a uio or vfio is already loaded on the host:

```
lsmod | grep -e igb_uio -e uio_pci_generic -e vfio
```

If missing, it is necessary to install a uio/vfio kernel module on the host server:

- find a suitable kernel module for your host server (any uio or vfio kernel module built with the same Linux kernel version should work)
- load it using the modprobe and insmod commands

Example of installation of the igb_uio kernel module:

```
modprobe uio
insmod ./igb_uio.ko
```

Finally, the correct iommu options and huge pages to be configured on the Linux server on the boot command line:

- enable intel_iommu and iommu pass through: “intel_iommu=on iommu=pt”

- for Trex, pre-allocate 1024 huge pages of 2MB each (for a total of 2GB): "hugepagesz=2M hugepages=1024"

More detailed instructions can be found in the DPDK documentation (https://media.readthedocs.org/pdf/dpdk/latest/dpdk.pdf).

NFVbench Installation and Quick Start Guide

Make sure you satisfy the *hardware and software requirements <requirements>* before you start .

1. Container installation

To pull the latest NFVbench container image:

```
docker pull opnfv/nfvbench
```

2. Docker Container configuration

The NFVbench container requires the following Docker options to operate properly.

Docker options	Description
-v /lib/modules/$(uname -r):/lib/modules/$(uname -r)	needed by kernel modules in the container
-v /usr/src/kernels:/usr/src/kernels	needed by TRex to build kernel modules when needed
-v /dev:/dev	needed by kernel modules in the container
-v $PWD:/tmp/nfvbench	optional but recommended to pass files between the host and the docker space (see examples below) Here we map the current directory on the host to the /tmp/nfvbench director in the container but any other similar mapping can work as well
–net=host	(optional) needed if you run the NFVbench server in the container (or use any appropriate docker network mode other than "host")
–privileged	(optional) required if SELinux is enabled on the host
-e HOST="127.0.0.1"	(optional) required if REST server is enabled
-e PORT=7556	(optional) required if REST server is enabled
-e CONFIG_FILE="/root/nfvbenchconfig.json	(optional) required if REST server is enabled

It can be convenient to write a shell script (or an alias) to automatically insert the necessary options.

The minimal configuration file required must specify the openrc file to use (using in-container path), the PCI addresses of the 2 NIC ports to use for generating traffic and the line rate (in each direction) of each of these 2 interfaces.

Here is an example of mimimal configuration where: the openrc file is located on the host current directory which is mapped under /tmp/nfvbench in the container (this is achieved using -v $PWD:/tmp/nfvbench) the 2 NIC ports to use for generating traffic have the PCI addresses "04:00.0" and "04:00.1"

```
{
    "openrc_file": "/tmp/nfvbench/openrc",
    "traffic_generator": {
        "generator_profile": [
            {
                "interfaces": [
                    {
                        "pci": "04:00.0",
                        "port": 0,
                    },
                    {
                        "pci": "04:00.1",
                        "port": 1,
                    }
                ],
                "intf_speed": "",
                "ip": "127.0.0.1",
                "name": "trex-local",
                "software_mode": false,
                "tool": "TRex"
            }
        ]
    }
}
```

The other options in the minimal configuration must be present and must have the same values as above.

3. Start the Docker container

As for any Docker container, you can execute NFVbench measurement sessions using a temporary container ("docker run" - which exits after each NFVbench run) or you can decide to run the NFVbench container in the background then execute one or more NFVbench measurement sessions on that container ("docker exec").

The former approach is simpler to manage (since each container is started and terminated after each command) but incurs a small delay at start time (several seconds). The second approach is more responsive as the delay is only incurred once when starting the container.

We will take the second approach and start the NFVbench container in detached mode with the name "nfvbench" (this works with bash, prefix with "sudo" if you do not use the root login)

First create a new working directory, and change the current working directory to there. A "nfvbench_ws" directory under your home directory is good place for that, and this is where the OpenStack RC file and NFVbench config file will sit.

To run NFVBench without server mode

```
cd ~/nfvbench_ws
docker run --detach --net=host --privileged -v $PWD:/tmp/nfvbench -v /dev:/dev -v /
↪lib/modules/$(uname -r):/lib/modules/$(uname -r) -v /usr/src/kernels:/usr/src/
↪kernels --name nfvbench opnfv/nfvbench
```

To run NFVBench enabling REST server (mount the configuration json and the path for openrc)

```
cd ~/nfvbench_ws
docker run --detach --net=host --privileged -e HOST="127.0.0.1" -e PORT=7556 -e␣
↪CONFIG_FILE="/tmp/nfvbench/nfvbenchconfig.json -v $PWD:/tmp/nfvbench -v /dev:/dev -
↪v /lib/modules/$(uname -r):/lib/modules/$(uname -r) -v /usr/src/kernels:/usr/src/
↪kernels --name nfvbench opnfv/nfvbench start_rest_server
```

(continues on next page)

(continued from previous page)

The create an alias to make it easy to execute nfvbench commands directly from the host shell prompt:

```
alias nfvbench='docker exec -it nfvbench nfvbench'
```

The next to last "nfvbench" refers to the name of the container while the last "nfvbench" refers to the NFVbench binary that is available to run in the container.

To verify it is working:

```
nfvbench --version
nfvbench --help
```

4. NFVbench configuration

Create a new file containing the minimal configuration for NFVbench, we can call it any name, for example "my_nfvbench.cfg" and paste the following yaml template in the file:

```
openrc_file:
traffic_generator:
    generator_profile:
        - name: trex-local
          tool: TRex
          ip: 127.0.0.1
          cores: 3
          software_mode: false,
          interfaces:
            - port: 0
              switch_port:
              pci:
            - port: 1
              switch_port:
              pci:
          intf_speed:
```

NFVbench requires an `openrc` file to connect to OpenStack using the OpenStack API. This file can be downloaded from the OpenStack Horizon dashboard (refer to the OpenStack documentation on how to retrieve the openrc file). The file pathname in the container must be stored in the "openrc_file" property. If it is stored on the host in the current directory, its full pathname must start with /tmp/nfvbench (since the current directory is mapped to /tmp/nfvbench in the container).

The required configuration is the PCI address of the 2 physical interfaces that will be used by the traffic generator. The PCI address can be obtained for example by using the "lspci" Linux command. For example:

```
[root@sjc04-pod6-build ~]# lspci | grep 710
0a:00.0 Ethernet controller: Intel Corporation Ethernet Controller X710 for 10GbE␣
↪SFP+ (rev 01)
0a:00.1 Ethernet controller: Intel Corporation Ethernet Controller X710 for 10GbE␣
↪SFP+ (rev 01)
0a:00.2 Ethernet controller: Intel Corporation Ethernet Controller X710 for 10GbE␣
↪SFP+ (rev 01)
0a:00.3 Ethernet controller: Intel Corporation Ethernet Controller X710 for 10GbE␣
↪SFP+ (rev 01)
```

Example of edited configuration with an OpenStack RC file stored in the current directory with the "openrc" name, and PCI addresses "0a:00.0" and "0a:00.1" (first 2 ports of the quad port NIC):

```
openrc_file: /tmp/nfvbench/openrc
traffic_generator:
    generator_profile:
        - name: trex-local
          tool: TRex
          ip: 127.0.0.1
          cores: 3
          software_mode: false,
          interfaces:
            - port: 0
              switch_port:
              pci: "0a:00.0"
            - port: 1
              switch_port:
              pci: "0a:00.1"
          intf_speed:
```

Warning: You have to put quotes around the pci addresses as shown in the above example, otherwise TRex will read it wrong.

Alternatively, the full template with comments can be obtained using the –show-default-config option in yaml format:

```
nfvbench --show-default-config > my_nfvbench.cfg
```

Edit the nfvbench.cfg file to only keep those properties that need to be modified (preserving the nesting).

Make sure you have your nfvbench configuration file (my_nfvbench.cfg) and OpenStack RC file in your pre-created working directory.

5. Run NFVbench

To do a single run at 10,000pps bi-directional (or 5kpps in each direction) using the PVP packet path:

```
nfvbench -c /tmp/nfvbench/my_nfvbench.cfg --rate 10kpps
```

NFVbench options used:

- `-c /tmp/nfvbench/my_nfvbench.cfg` : specify the config file to use (this must reflect the file path from inside the container)
- `--rate 10kpps` : specify rate of packets for test for both directions using the kpps unit (thousands of packets per second)

This should produce a result similar to this (a simple run with the above options should take less than 5 minutes):

```
[TBP]
```

7. Terminating the NFVbench container

When no longer needed, the container can be terminated using the usual docker commands:

```
docker kill nfvbench
docker rm nfvbench
```

6.1.4 Example of Results

Example run for fixed rate

```
nfvbench -c /nfvbench/nfvbenchconfig.json --rate 1%
```

```
========== NFVBench Summary ==========
Date: 2017-09-21 23:57:44
NFVBench version 1.0.9
Openstack Neutron:
  vSwitch: BASIC
  Encapsulation: BASIC
Benchmarks:
> Networks:
  > Components:
    > TOR:
        Type: None
    > Traffic Generator:
        Profile: trex-local
        Tool: TRex
    > Versions:
      > TOR:
      > Traffic Generator:
          build_date: Aug 30 2017
          version: v2.29
          built_by: hhaim
          build_time: 16:43:55
  > Service chain:
    > PVP:
      > Traffic:
          Profile: traffic_profile_64B
          Bidirectional: True
          Flow count: 10000
          Service chains count: 1
          Compute nodes: []

            Run Summary:

              +-----------------+-------------+----------------------+----------------
↪------+----------------------+
              |   L2 Frame Size |  Drop Rate  |   Avg Latency (usec) |   Min Latency␣
↪(usec) |   Max Latency (usec) |
              ␣
↪+=================+=============+======================+======================+=======================
              |              64 |   0.0000%   |                   53 |                ␣
↪   20 |                  211 |
              +-----------------+-------------+----------------------+----------------
↪------+----------------------+


            L2 frame size: 64
            Chain analysis duration: 60.076 seconds
```

(continues on next page)

(continued from previous page)

```
           Run Config:

             +-------------+---------------------------+-----------------------+----
↪-------------+---------------------------+-----------------------+-----------------
↪+
             |  Direction  |  Requested TX Rate (bps)  |  Actual TX Rate (bps)  | ␣
↪RX Rate (bps)  |  Requested TX Rate (pps)  |  Actual TX Rate (pps)  |  RX Rate␣
↪(pps)  |
           ␣
↪+=============+===========================+=======================+=================+=============
             |   Forward   |       100.0000 Mbps       |      95.4546 Mbps      | ␣
↪95.4546 Mbps   |        148,809 pps        |      142,045 pps       |   142,045 pps␣
↪  |
             +-------------+---------------------------+-----------------------+----
↪-------------+---------------------------+-----------------------+-----------------
↪+
             |   Reverse   |       100.0000 Mbps       |      95.4546 Mbps      | ␣
↪95.4546 Mbps   |        148,809 pps        |      142,045 pps       |   142,045 pps␣
↪  |
             +-------------+---------------------------+-----------------------+----
↪-------------+---------------------------+-----------------------+-----------------
↪+
             |    Total    |       200.0000 Mbps       |     190.9091 Mbps      | ␣
↪190.9091 Mbps  |        297,618 pps        |      284,090 pps       |   284,090 pps␣
↪  |
             +-------------+---------------------------+-----------------------+----
↪-------------+---------------------------+-----------------------+-----------------
↪+

           Chain Analysis:

             +------------------+----------+-----------------+---------------+------
↪---------+-----------------+---------------+---------------+
             |     Interface    |  Device  |  Packets (fwd)  |   Drops (fwd) |  Drop
↪% (fwd)  |  Packets (rev)  |   Drops (rev) |  Drop% (rev)  |
           ␣
↪+==================+==========+=================+===============+===============+=============
             | traffic-generator |   trex   |    8,522,729    |               | ␣
↪         |    8,522,729    |             0 |    0.0000%    |
             +------------------+----------+-----------------+---------------+------
↪---------+-----------------+---------------+---------------+
             | traffic-generator |   trex   |    8,522,729    |             0 |    0.
↪0000%    |    8,522,729    |               |               |
             +------------------+----------+-----------------+---------------+------
↪---------+-----------------+---------------+---------------+
```

Example run for NDR/PDR with package size 1518B

```
nfvbench -c /nfvbench/nfvbenchconfig.json -fs 1518
```

```
========== NFVBench Summary ==========
Date: 2017-09-22 00:02:07
NFVBench version 1.0.9
Openstack Neutron:
  vSwitch: BASIC
```

(continues on next page)

(continued from previous page)

```
  Encapsulation: BASIC
Benchmarks:
> Networks:
  > Components:
    > TOR:
        Type: None
    > Traffic Generator:
        Profile: trex-local
        Tool: TRex
    > Versions:
      > TOR:
      > Traffic Generator:
          build_date: Aug 30 2017
          version: v2.29
          built_by: hhaim
          build_time: 16:43:55
  > Measurement Parameters:
      NDR: 0.001
      PDR: 0.1
  > Service chain:
    > PVP:
      > Traffic:
          Profile: custom_traffic_profile
          Bidirectional: True
          Flow count: 10000
          Service chains count: 1
          Compute nodes: []

            Run Summary:

              +-----+-----------------+------------------+------------------+---------
↪--------+----------------------+----------------------+----------------------+
              |  -  |   L2 Frame Size |  Rate (fwd+rev)  |  Rate (fwd+rev)  |  Avg␣
↪Drop Rate  |   Avg Latency (usec) |   Min Latency (usec) |  Max Latency (usec)  |
              ␣
↪+=====+=================+==================+==================+==================+=================
              | NDR |            1518 |   19.9805 Gbps   |  1,623,900 pps   |     0.
↪0001%     |                  342 |                   30 |         704          |
              +-----+-----------------+------------------+------------------+---------
↪--------+----------------------+----------------------+----------------------+
              | PDR |            1518 |   20.0000 Gbps   |  1,625,486 pps   |     0.
↪0022%     |                  469 |                   40 |        1,266         |
              +-----+-----------------+------------------+------------------+---------
↪--------+----------------------+----------------------+----------------------+


            L2 frame size: 1518
            Chain analysis duration: 660.442 seconds
            NDR search duration: 660 seconds
            PDR search duration: 0 seconds
```

6.1.5 Advanced Usage

This section covers a few examples on how to run NFVbench with multiple different settings. Below are shown the most common and useful use-cases and explained some fields from a default config file.

How to change any NFVbench run configuration (CLI)

NFVbench always starts with a default configuration which can further be refined (overridden) by the user from the CLI or from REST requests.

At first have a look at the default config:

```
nfvbench --show-default-config
```

It is sometimes useful derive your own configuration from a copy of the default config:

```
nfvbench --show-default-config > nfvbench.cfg
```

At this point you can edit the copy by:

- removing any parameter that is not to be changed (since NFVbench will always load the default configuration, default values are not needed)
- edit the parameters that are to be changed changed

A run with the new confguration can then simply be requested using the -c option and by using the actual path of the configuration file as seen from inside the container (in this example, we assume the current directory is mapped to /tmp/nfvbench in the container):

```
nfvbench -c /tmp/nfvbench/nfvbench.cfg
```

The same -c option also accepts any valid yaml or json string to override certain parameters without having to create a configuration file.

NFVbench provides many configuration options as optional arguments. For example the number of flows can be specified using the –flow-count option.

The flow count option can be specified in any of 3 ways:

- by providing a confguration file that has the flow_count value to use (-c myconfig.yaml and myconfig.yaml contains 'flow_count: 100k')
- by passing that yaml paremeter inline (-c "flow_count: 100k") or (-c "{flow_count: 100k}")
- by using the flow count optional argument (–flow-count 100k)

Showing the running configuration

Because configuration parameters can be overriden, it is sometimes useful to show the final configuration (after all oevrrides are done) by using the –show-config option. This final configuration is also called the "running" configuration.

For example, this will only display the running configuration (without actually running anything):

```
nfvbench -c "{flow_count: 100k, debug: true}" --show-config
```

Connectivity and Configuration Check

NFVbench allows to test connectivity to devices used with the selected packet path. It runs the whole test, but without actually sending any traffic. It is also a good way to check if everything is configured properly in the configuration file and what versions of components are used.

To verify everything works without sending any traffic, use the –no-traffic option:

```
nfvbench --no-traffic
```

Used parameters:

- `--no-traffic` or `-0` : sending traffic from traffic generator is skipped

Fixed Rate Run

Fixed rate run is the most basic type of NFVbench usage. It can be used to measure the drop rate with a fixed transmission rate of packets.

This example shows how to run the PVP packet path (which is the default packet path) with multiple different settings:

```
nfvbench -c nfvbench.cfg --no-cleanup --rate 100000pps --duration 30 --interval 15 --
↪json results.json
```

Used parameters:

- `-c nfvbench.cfg` : path to the config file
- `--no-cleanup` : resources (networks, VMs, attached ports) are not deleted after test is finished
- `--rate 100000pps` : defines rate of packets sent by traffic generator
- `--duration 30` : specifies how long should traffic be running in seconds
- `--interval 15` : stats are checked and shown periodically (in seconds) in this interval when traffic is flowing
- `--json results.json` : collected data are stored in this file after run is finished

---

Note: It is your responsibility to clean up resources if needed when `--no-cleanup` parameter is used. You can use the nfvbench_cleanup helper script for that purpose.

The `--json` parameter makes it easy to store NFVbench results. The –show-summary (or -ss) option can be used to display the results in a json results file in a text tabular format:

```
nfvbench --show-summary results.json
```

This example shows how to specify a different packet path:

```
nfvbench -c nfvbench.cfg --rate 1Mbps --inter-node --service-chain PVVP
```

Used parameters:

- `-c nfvbench.cfg` : path to the config file
- `--rate 1Mbps` : defines rate of packets sent by traffic generator
- `--inter-node` : VMs are created on different compute nodes, works only with PVVP flow
- `--service-chain PVVP` or `-sc PVVP` : specifies the type of service chain (or packet path) to use

---

Note: When parameter `--inter-node` is not used or there aren't enough compute nodes, VMs are on the same compute node.

Rate Units

Parameter `--rate` accepts different types of values:

- packets per second (pps, kpps, mpps), e.g. `1000pps` or `10kpps`
- load percentage (%), e.g. `50%`
- bits per second (bps, kbps, Mbps, Gbps), e.g. `1Gbps`, `1000bps`
- NDR/PDR (ndr, pdr, ndr_pdr), e.g. `ndr_pdr`

NDR/PDR is the default rate when not specified.

NDR and PDR

The NDR and PDR test is used to determine the maximum throughput performance of the system under test following guidelines defined in RFC-2544:

- NDR (No Drop Rate): maximum packet rate sent without dropping any packet
- PDR (Partial Drop Rate): maximum packet rate sent while allowing a given maximum drop rate

The NDR search can also be relaxed to allow some very small amount of drop rate (lower than the PDR maximum drop rate). NFVbench will measure the NDR and PDR values by driving the traffic generator through multiple iterations at different transmission rates using a binary search algorithm.

The configuration file contains section where settings for NDR/PDR can be set.

```
# NDR/PDR configuration
measurement:
    # Drop rates represent the ratio of dropped packet to the total number of packets␣
↪sent.
    # Values provided here are percentages. A value of 0.01 means that at most 0.01%␣
↪of all
    # packets sent are dropped (or 1 packet every 10,000 packets sent)

    # No Drop Rate; Default to 0.001%
    NDR: 0.001
    # Partial Drop Rate; NDR should always be less than PDR
    PDR: 0.1
    # The accuracy of NDR and PDR load percentiles; The actual load percentile that␣
↪match NDR
    # or PDR should be within `load_epsilon` difference than the one calculated.
    load_epsilon: 0.1
```

Because NDR/PDR is the default `--rate` value, it is possible to run NFVbench simply like this:

```
nfvbench -c nfvbench.cfg
```

Other possible run options:

```
nfvbench -c nfvbench.cfg --duration 120 --json results.json
```

Used parameters:

- `-c nfvbench.cfg` : path to the config file
- `--duration 120` : specifies how long should be traffic running in each iteration
- `--json results.json` : collected data are stored in this file after run is finished

Multichain

NFVbench allows to run multiple chains at the same time. For example it is possible to stage the PVP service chain N-times, where N can be as much as your compute power can scale. With N = 10, NFVbench will spawn 10 VMs as a part of 10 simultaneous PVP chains.

The number of chains is specified by `--service-chain-count` or `-scc` flag with a default value of 1. For example to run NFVbench with 3 PVP chains:

```
nfvbench -c nfvbench.cfg --rate 10000pps -scc 3
```

It is not necessary to specify the service chain type (-sc) because PVP is set as default. The PVP service chains will have 3 VMs in 3 chains with this configuration. If `-sc PVVP` is specified instead, there would be 6 VMs in 3 chains as this service chain has 2 VMs per chain. Both **single run** or **NDR/PDR** can be run as multichain. Running multichain is a scenario closer to a real life situation than runs with a single chain.

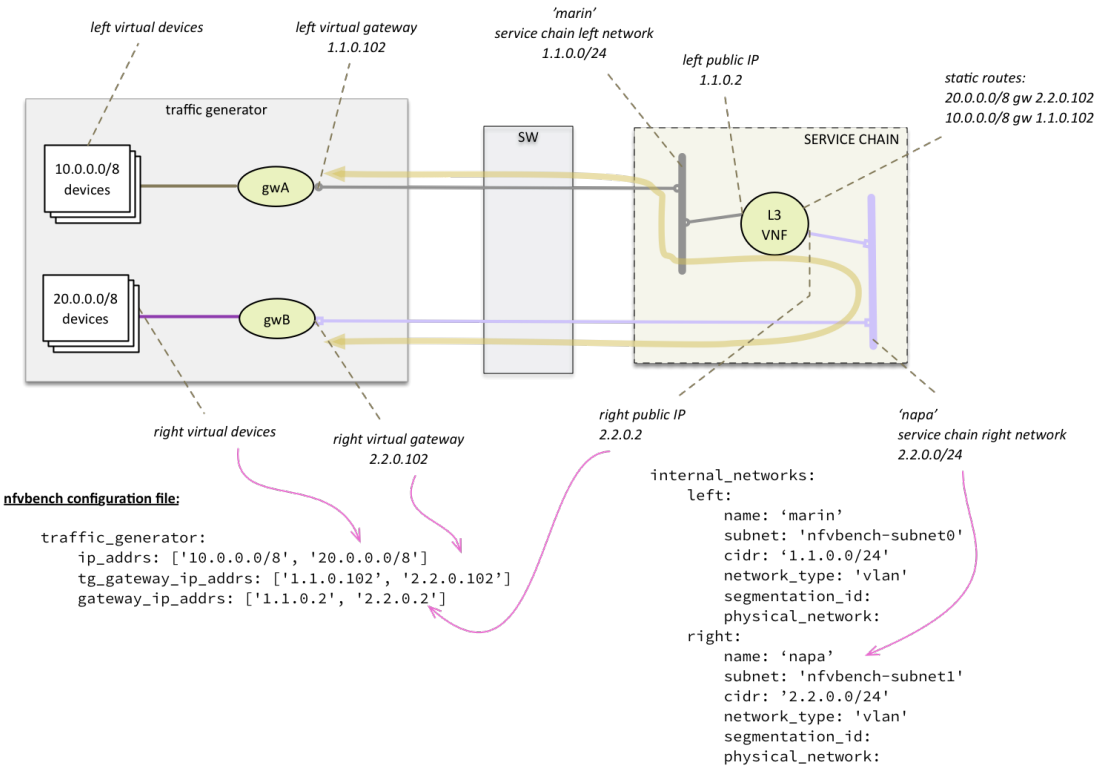
External Chain

NFVbench can measure the performance of 1 or more L3 service chains that are setup externally. Instead of being setup by NFVbench, the complete environment (VMs and networks) has to be setup prior to running NFVbench.

Each external chain is made of 1 or more VNFs and has exactly 2 end network interfaces (left and right network interfaces) that are connected to 2 neutron networks (left and right networks). The internal composition of a multi-VNF service chain can be arbitrary (usually linear) as far as NFVbench is concerned, the only requirement is that the service chain can route L3 packets properly between the left and right networks.

To run NFVbench on such external service chains:

- explicitly tell NFVbench to use external service chain by adding `-sc EXT` or `--service-chain EXT` to NFVbench CLI options
- specify the number of external chains using the `-scc` option (defaults to 1 chain)
- **specify the 2 end point networks of your environment in `external_networks` inside the config file.**
  - The two networks specified there have to exist in Neutron and will be used as the end point networks by NFVbench ('napa' and 'marin' in the diagram below)
- specify the router gateway IPs for the external service chains (1.1.0.2 and 2.2.0.2)
- specify the traffic generator gateway IPs for the external service chains (1.1.0.102 and 2.2.0.102 in diagram below)
- specify the packet source and destination IPs for the virtual devices that are simulated (10.0.0.0/8 and 20.0.0.0/8)



L3 routing must be enabled in the VNF and configured to:

- reply to ARP requests to its public IP addresses on both left and right networks
- route packets from each set of remote devices toward the appropriate dest gateway IP in the traffic generator using 2 static routes (as illustrated in the diagram)

Upon start, NFVbench will: - first retrieve the properties of the left and right networks using Neutron APIs, - extract the underlying network ID (typically VLAN segmentation ID), - generate packets with the proper VLAN ID and measure traffic.

Note that in the case of multiple chains, all chains end interfaces must be connected to the same two left and right networks. The traffic will be load balanced across the corresponding gateway IP of these external service chains.

Multiflow

NFVbench always generates L3 packets from the traffic generator but allows the user to specify how many flows to generate. A flow is identified by a unique src/dest MAC IP and port tuple that is sent by the traffic generator. Flows are generated by ranging the IP adresses but using a small fixed number of MAC addresses.

The number of flows will be spread roughly even between chains when more than 1 chain is being tested. For example, for 11 flows and 3 chains, number of flows that will run for each chain will be 3, 4, and 4 flows respectively.

The number of flows is specified by `--flow-count` or `-fc` flag, the default value is 2 (1 flow in each direction). To run NFVbench with 3 chains and 100 flows, use the following command:

```
nfvbench -c nfvbench.cfg --rate 10000pps -scc 3 -fc 100
```

Note that from a vswitch point of view, the number of flows seen will be higher as it will be at least 4 times the number of flows sent by the traffic generator (add flow to VM and flow from VM).

IP addresses generated can be controlled with the following NFVbench configuration options:

```
ip_addrs: ['10.0.0.0/8', '20.0.0.0/8']
ip_addrs_step: 0.0.0.1
tg_gateway_ip_addrs: ['1.1.0.100', '2.2.0.100']
tg_gateway_ip_addrs_step: 0.0.0.1
gateway_ip_addrs: ['1.1.0.2', '2.2.0.2']
gateway_ip_addrs_step: 0.0.0.1
```

`ip_addrs` are the start of the 2 ip address ranges used by the traffic generators as the packets source and destination packets where each range is associated to virtual devices simulated behind 1 physical interface of the traffic generator. These can also be written in CIDR notation to represent the subnet.

`tg_gateway_ip_addrs` are the traffic generator gateway (virtual) ip addresses, all traffic to/from the virtual devices go through them.

`gateway_ip_addrs` are the 2 gateway ip address ranges of the VMs used in the external chains. They are only used with external chains and must correspond to their public IP address.

The corresponding `step` is used for ranging the IP addresses from the *ip_addrs'*, `tg_gateway_ip_addrs` and `gateway_ip_addrs` base addresses. 0.0.0.1 is the default step for all IP ranges. In `ip_addrs`, 'random' can be configured which tells NFVBench to generate random src/dst IP pairs in the traffic stream.

Traffic Configuration via CLI

While traffic configuration can be modified using the configuration file, it can be inconvenient to have to change the configuration file everytime you need to change a traffic configuration option. Traffic configuration options can be overridden with a few CLI options.

Here is an example of configuring traffic via CLI:

```
nfvbench --rate 10kpps --service-chain-count 2 -fs 64 -fs IMIX -fs 1518 --unidir
```

This command will run NFVbench with a unidirectional flow for three packet sizes 64B, IMIX, and 1518B.

Used parameters:

- `--rate 10kpps` : defines rate of packets sent by traffic generator (total TX rate)
- `-scc 2` or `--service-chain-count 2` : specifies number of parallel chains of given flow to run (default to 1)
- `-fs 64` or `--frame-size 64`: add the specified frame size to the list of frame sizes to run
- `--unidir` : run traffic with unidirectional flow (default to bidirectional flow)

MAC Addresses

NFVbench will dicover the MAC addresses to use for generated frames using: - either OpenStack discovery (find the MAC of an existing VM) in the case of PVP and PVVP service chains - or using dynamic ARP discovery (find MAC from IP) in the case of external chains.

Status and Cleanup of NFVbench Resources

The –status option will display the status of NFVbench and list any NFVbench resources. You need to pass the OpenStack RC file in order to connect to OpenStack.

```
# nfvbench --status -r /tmp/nfvbench/openrc
2018-04-09 17:05:48,682 INFO Version: 1.3.2.dev1
2018-04-09 17:05:48,683 INFO Status: idle
2018-04-09 17:05:48,757 INFO Discovering instances nfvbench-loop-vm...
2018-04-09 17:05:49,252 INFO Discovering flavor nfvbench.medium...
2018-04-09 17:05:49,281 INFO Discovering networks...
2018-04-09 17:05:49,365 INFO No matching NFVbench resources found
#
```

The Status can be either “idle” or “busy (run pending)”.

The –cleanup option will first discover resources created by NFVbench and prompt if you want to proceed with cleaning them up. Example of run:

```
# nfvbench --cleanup -r /tmp/nfvbench/openrc
2018-04-09 16:58:00,204 INFO Version: 1.3.2.dev1
2018-04-09 16:58:00,205 INFO Status: idle
2018-04-09 16:58:00,279 INFO Discovering instances nfvbench-loop-vm...
2018-04-09 16:58:00,829 INFO Discovering flavor nfvbench.medium...
2018-04-09 16:58:00,876 INFO Discovering networks...
2018-04-09 16:58:00,960 INFO Discovering ports...
2018-04-09 16:58:01,012 INFO Discovered 6 NFVbench resources:
+----------+------------------+--------------------------------------+
| Type     | Name             | UUID                                 |
|----------+------------------+--------------------------------------|
| Instance | nfvbench-loop-vm0 | b039b858-777e-467e-99fb-362f856f4a94 |
| Flavor   | nfvbench.medium  | a027003c-ad86-4f24-b676-2b05bb06adc0 |
| Network  | nfvbench-net0    | bca8d183-538e-4965-880e-fd92d48bfe0d |
| Network  | nfvbench-net1    | c582a201-8279-4309-8084-7edd6511092c |
| Port     |                  | 67740862-80ac-4371-b04e-58a0b0f05085 |
| Port     |                  | b5db95b9-e419-4725-951a-9a8f7841e66a |
+----------+------------------+--------------------------------------+
2018-04-09 16:58:01,013 INFO NFVbench will delete all resources shown...
Are you sure? (y/n) y
2018-04-09 16:58:01,865 INFO Deleting instance nfvbench-loop-vm0...
2018-04-09 16:58:02,058 INFO     Waiting for 1 instances to be fully deleted...
2018-04-09 16:58:02,182 INFO     1 yet to be deleted by Nova, retries left=6...
2018-04-09 16:58:04,506 INFO     1 yet to be deleted by Nova, retries left=5...
2018-04-09 16:58:06,636 INFO     1 yet to be deleted by Nova, retries left=4...
2018-04-09 16:58:08,701 INFO Deleting flavor nfvbench.medium...
2018-04-09 16:58:08,729 INFO Deleting port 67740862-80ac-4371-b04e-58a0b0f05085...
2018-04-09 16:58:09,102 INFO Deleting port b5db95b9-e419-4725-951a-9a8f7841e66a...
2018-04-09 16:58:09,620 INFO Deleting network nfvbench-net0...
2018-04-09 16:58:10,357 INFO Deleting network nfvbench-net1...
#
```

The –force-cleanup option will do the same but without prompting for confirmation.

6.1.6 NFVbench Fluentd Integration

NFVbench has an optional fluentd integration to save logs and results.

Configuring Fluentd to receive NFVbench logs and results

The following configurations should be added to Fluentd configuration file to enable logs or results.

To receive logs, and forward to a storage server:

In the example below nfvbench is the tag name for logs (which should be matched with logging_tag under NFVbench configuration), and storage backend is elasticsearch which is running at localhost:9200.

```
<match nfvbench.**>
@type copy
<store>
    @type elasticsearch
    host localhost
    port 9200
    logstash_format true
    logstash_prefix nfvbench
    utc_index false
    flush_interval 15s
</store>
</match>
```

To receive results, and forward to a storage server:

In the example below resultnfvbench is the tag name for results (which should be matched with result_tag under NFVbench configuration), and storage backend is elasticsearch which is running at localhost:9200.

```
<match resultnfvbench.**>
@type copy
<store>
    @type elasticsearch
    host localhost
    port 9200
    logstash_format true
    logstash_prefix resultnfvbench
    utc_index false
    flush_interval 15s
</store>
</match>
```

Configuring NFVbench to connect Fluentd

To configure NFVbench to connect Fluentd, fill following configuration parameters in the configuration file

Configuration	Description
logging_tag	Tag for NFVbench logs, it should be the same tag defined in Fluentd configuration
result_tag	Tag for NFVbench results, it should be the same tag defined in Fluentd configuration
ip	ip address of Fluentd server
port	port number of Fluentd serverd

An example of configuration for Fluentd working at 127.0.0.1:24224 and tags for logging is nfvbench and result is resultnfvbench

```
fluentd:
    # by default (logging_tag is empty) nfvbench log messages are not sent to fluentd
    # to enable logging to fluents, specify a valid fluentd tag name to be used for␣
↪the
```

(continues on next page)

(continued from previous page)

```
    # log records
    logging_tag: nfvbench

    # by default (result_tag is empty) nfvbench results are not sent to fluentd
    # to enable sending nfvbench results to fluentd, specify a valid fluentd tag name
    # to be used for the results records, which is different than logging_tag
    result_tag: resultnfvbench

    # IP address of the server, defaults to loopback
    ip: 127.0.0.1

    # port # to use, by default, use the default fluentd forward port
    port: 24224
```

Example of logs and results

An example of log obtained from fluentd by elasticsearch:

```
{
  "_index": "nfvbench-2017.10.17",
  "_type": "fluentd",
  "_id": "AV8rhnCjTgGF_dX8DiKK",
  "_version": 1,
  "_score": 3,
  "_source": {
    "loglevel": "INFO",
    "message": "Service chain 'PVP' run completed.",
    "@timestamp": "2017-10-17T18:09:09.516897+0000",
    "runlogdate": "2017-10-17T18:08:51.851253+0000"
  },
  "fields": {
    "@timestamp": [
      1508263749516
    ]
  }
}
```

For each packet size and rate a result record is sent. Users can label those results by passing –user-label parameter to NFVbench run

And the results of this command obtained from fluentd by elasticsearch:

```
{
  "_index": "resultnfvbench-2017.10.17",
  "_type": "fluentd",
  "_id": "AV8rjYlbTgGF_dX8Drl1",
  "_version": 1,
  "_score": null,
  "_source": {
    "compute_nodes": [
      "nova:compute-3"
    ],
    "total_orig_rate_bps": 200000000,
    "@timestamp": "2017-10-17T18:16:43.755240+0000",
    "frame_size": "64",
```

(continues on next page)

(continued from previous page)


```
    "forward_orig_rate_pps": 148809,
    "flow_count": 10000,
    "avg_delay_usec": 6271,
    "total_tx_rate_pps": 283169,
    "total_tx_rate_bps": 190289668,
    "forward_tx_rate_bps": 95143832,
    "reverse_tx_rate_bps": 95145836,
    "forward_tx_rate_pps": 141583,
    "chain_analysis_duration": "60.091",
    "service_chain": "PVP",
    "version": "1.0.10.dev1",
    "runlogdate": "2017-10-17T18:10:12.134260+0000",
    "Encapsulation": "VLAN",
    "user_label": "nfvbench-label",
    "min_delay_usec": 70,
    "profile": "traffic_profile_64B",
    "reverse_rx_rate_pps": 68479,
    "reverse_rx_rate_bps": 46018044,
    "reverse_orig_rate_pps": 148809,
    "total_rx_rate_bps": 92030085,
    "drop_rate_percent": 51.6368455626846,
    "forward_orig_rate_bps": 100000000,
    "bidirectional": true,
    "vSwitch": "OPENVSWITCH",
    "sc_count": 1,
    "total_orig_rate_pps": 297618,
    "type": "single_run",
    "reverse_orig_rate_bps": 100000000,
    "total_rx_rate_pps": 136949,
    "max_delay_usec": 106850,
    "forward_rx_rate_pps": 68470,
    "forward_rx_rate_bps": 46012041,
    "reverse_tx_rate_pps": 141586
  },
  "fields": {
    "@timestamp": [
      1508264203755
    ]
  },
  "sort": [
    1508264203755
  ]
}
```


6.1.7 Testing SR-IOV

NFVbench supports SR-IOV with the PVP packet flow (PVVP is not supported). SR-IOV support is not applicable for external chains since the networks have to be setup externally (and can themselves be pre-set to use SR-IOV or not).

Pre-requisites

To test SR-IOV you need to have compute nodes configured to support one or more SR-IOV interfaces (also knows as PF or physical function) and you need OpenStack to be configured to support SR-IOV. You will also need to know: - the name of the physical networks associated to your SR-IOV interfaces (this is a configuration in Nova compute) - the

VLAN range that can be used on the switch ports that are wired to the SR-IOV ports. Such switch ports are normally configured in trunk mode with a range of VLAN ids enabled on that port

For example, in the case of 2 SR-IOV ports per compute node, 2 physical networks are generally configured in OpenStack with a distinct name. The VLAN range to use is is also allocated and reserved by the network administrator and in coordination with the corresponding top of rack switch port configuration.

Configuration

To enable SR-IOV test, you will need to provide the following configuration options to NFVbench (in the configuration file). This example instructs NFVbench to create the left and right networks of a PVP packet flow to run on 2 SRIOV ports named "phys_sriov0" and "phys_sriov1" using resp. segmentation_id 2000 and 2001:

```
internal_networks:
   left:
       segmentation_id: 2000
       physical_network: phys_sriov0
   right:
       segmentation_id: 2001
       physical_network: phys_sriov1
```

The segmentation ID fields must be different. In the case of PVVP, the middle network also needs to be provisioned properly. The same physical network can also be shared by the virtual networks but with different segmentation IDs.

NIC NUMA socket placement and flavors

If the 2 selected ports reside on NICs that are on different NUMA sockets, you will need to explicitly tell Nova to use 2 numa nodes in the flavor used for the VMs in order to satisfy the filters, for example:

```
flavor:
  # Number of vCPUs for the flavor
  vcpus: 2
  # Memory for the flavor in MB
  ram: 8192
  # Size of local disk in GB
  disk: 0
  extra_specs:
      "hw:cpu_policy": dedicated
      "hw:mem_page_size": large
      "hw:numa_nodes": 2
```

Failure to do so might cause the VM creation to fail with the Nova error "Instance creation error: Insufficient compute resources: Requested instance NUMA topology together with requested PCI devices cannot fit the given host NUMA topology."

6.1.8 NFVbench Server mode and NFVbench client API

NFVbench can run as an HTTP server to:

- optionally provide access to any arbitrary HTLM files (HTTP server function) - this is optional
- service fully parameterized aynchronous run requests using the HTTP protocol (REST/json with polling)
- service fully parameterized run requests with interval stats reporting using the WebSocket/SocketIO protocol.

Start the NFVbench server

To run in server mode, simply use the –server <http_root_path> and optionally the listen address to use (–host <ip>, default is 0.0.0.0) and listening port to use (–port <port>, default is 7555).

If HTTP files are to be serviced, they must be stored right under the http root path. This root path must contain a static folder to hold static files (css, js) and a templates folder with at least an index.html file to hold the template of the index.html file to be used. This mode is convenient when you do not already have a WEB server hosting the UI front end. If HTTP files servicing is not needed (REST only or WebSocket/SocketIO mode), the root path can point to any dummy folder.

Once started, the NFVbench server will be ready to service HTTP or WebSocket/SocketIO requests at the advertised URL.

Example of NFVbench server start in a container:

```
# get to the container shell (assume the container name is "nfvbench")
docker exec -it nfvbench bash
# from the container shell start the NFVbench server in the background
nfvbench -c /tmp/nfvbench/nfvbench.cfg --server /tmp &
# exit container
exit
```

HTTP Interface

<http-url>/echo (GET)

This request simply returns whatever content is sent in the body of the request (body should be in json format, only used for testing)

Example request:

```
curl -XGET '127.0.0.1:7556/echo' -H "Content-Type: application/json" -d '{"nfvbench":
↪"test"}'
Response:
{
  "nfvbench": "test"
}
```

<http-url>/status (GET)

This request fetches the status of an asynchronous run. It will return in json format:

- a request pending reply (if the run is still not completed)
- an error reply if there is no run pending
- or the complete result of the run

The client can keep polling until the run completes.

Example of return when the run is still pending:

```
{
  "error_message": "nfvbench run still pending",
  "status": "PENDING"
}
```

Example of return when the run completes:

```
{
  "result": {...}
  "status": "OK"
}
```

<http-url>/start_run (POST)

This request starts an NFVBench run with passed configurations. If no configuration is passed, a run with default configurations will be executed.

Example request: curl -XPOST 'localhost:7556/start_run' -H "Content-Type: application/json" -d @nfvbenchconfig.json

See "NFVbench configuration JSON parameter" below for details on how to format this parameter.

The request returns immediately with a json content indicating if there was an error (status=ERROR) or if the request was submitted successfully (status=PENDING). Example of return when the submission is successful:

```
{
  "error_message": "NFVbench run still pending",
  "request_id": "42cccb7effdc43caa47f722f0ca8ec96",
  "status": "PENDING"
}
```

If there is already an NFVBench running then it will return:

```
{
 "error_message": "there is already an NFVbench request running",
 "status": "ERROR"
}
```

WebSocket/SocketIO events

List of SocketIO events supported:

Client to Server

start_run:

> sent by client to start a new run with the configuration passed in argument (JSON). The configuration can be any valid NFVbench configuration passed as a JSON document (see "NFVbench configuration JSON parameter" below)

Server to Client

run_interval_stats:

> sent by server to report statistics during a run the message contains the statistics {'time_ms': time_ms, 'tx_pps': tx_pps, 'rx_pps': rx_pps, 'drop_pct': drop_pct}

ndr_found:

(during NDR-PDR search) sent by server when the NDR rate is found the message contains the NDR value {'rate_pps': ndr_pps}

ndr_found:

(during NDR-PDR search) sent by server when the PDR rate is found the message contains the PDR value {'rate_pps': pdr_pps}

run_end:

sent by server to report the end of a run the message contains the complete results in JSON format

NFVbench configuration JSON parameter

The NFVbench configuration describes the parameters of an NFVbench run and can be passed to the NFVbench server as a JSON document.

Default configuration

The simplest JSON document is the empty dictionary "{}" which indicates to use the default NFVbench configuration:

- PVP
- NDR-PDR measurement
- 64 byte packets
- 1 flow per direction

The entire default configuration can be viewed using the –show-json-config option on the cli:

```
# nfvbench --show-config
{
    "availability_zone": null,
    "compute_node_user": "root",
    "compute_nodes": null,
    "debug": false,
    "duration_sec": 60,
    "flavor": {
        "disk": 0,
        "extra_specs": {
            "hw:cpu_policy": "dedicated",
            "hw:mem_page_size": 2048
        },
        "ram": 8192,
        "vcpus": 2
    },
    "flavor_type": "nfv.medium",
    "flow_count": 1,
    "generic_poll_sec": 2,
    "generic_retry_count": 100,
    "inter_node": false,
    "internal_networks": {
        "left": {
            "name": "nfvbench-net0",
            "subnet": "nfvbench-subnet0",
            "cidr": "192.168.1.0/24",
        },
```

(continues on next page)

(continued from previous page)


```
        "right": {
            "name": "nfvbench-net1",
            "subnet": "nfvbench-subnet1",
            "cidr": "192.168.2.0/24",
        },
        "middle": {
            "name": "nfvbench-net2",
            "subnet": "nfvbench-subnet2",
            "cidr": "192.168.3.0/24",
        }
    },
    "interval_sec": 10,
    "json": null,
    "loop_vm_name": "nfvbench-loop-vm",
    "measurement": {
        "NDR": 0.001,
        "PDR": 0.1,
        "load_epsilon": 0.1
    },
    "name": "(built-in default config)",
    "no_cleanup": false,
    "no_traffic": false,
    "openrc_file": "/tmp/nfvbench/openstack/openrc",
    "rate": "ndr_pdr",
    "service_chain": "PVP",
    "service_chain_count": 1,
    "sriov": false,
    "std_json": null,
    "traffic": {
        "bidirectional": true,
        "profile": "traffic_profile_64B"
    },
    "traffic_generator": {
        "default_profile": "trex-local",
        "gateway_ip_addrs": [
            "1.1.0.2",
            "2.2.0.2"
        ],
        "gateway_ip_addrs_step": "0.0.0.1",
        "generator_profile": [
            {
                "cores": 3,
                "interfaces": [
                    {
                        "pci": "0a:00.0",
                        "port": 0,
                        "switch_port": "Ethernet1/33",
                        "vlan": null
                    },
                    {
                        "pci": "0a:00.1",
                        "port": 1,
                        "switch_port": "Ethernet1/34",
                        "vlan": null
                    }
                ],
                "intf_speed": null,
```


(continues on next page)

(continued from previous page)

```
                "ip": "127.0.0.1",
                "name": "trex-local",
                "tool": "TRex"
            }
        ],
        "host_name": "nfvbench_tg",
        "ip_addrs": [
            "10.0.0.0/8",
            "20.0.0.0/8"
        ],
        "ip_addrs_step": "0.0.0.1",
        "mac_addrs": [
            "00:10:94:00:0A:00",
            "00:11:94:00:0A:00"
        ],
        "step_mac": null,
        "tg_gateway_ip_addrs": [
            "1.1.0.100",
            "2.2.0.100"
        ],
        "tg_gateway_ip_addrs_step": "0.0.0.1"
    },
    "traffic_profile": [
        {
            "l2frame_size": [
                "64"
            ],
            "name": "traffic_profile_64B"
        },
        {
            "l2frame_size": [
                "IMIX"
            ],
            "name": "traffic_profile_IMIX"
        },
        {
            "l2frame_size": [
                "1518"
            ],
            "name": "traffic_profile_1518B"
        },
        {
            "l2frame_size": [
                "64",
                "IMIX",
                "1518"
            ],
            "name": "traffic_profile_3sizes"
        }
    ],
    "unidir_reverse_traffic_pps": 1,
    "vlan_tagging": true,
}
```

Common examples of JSON configuration

Use the default configuration but use 10000 flows per direction (instead of 1):

```
{ "flow_count": 10000 }
```

Use default confguration but with 10000 flows, "EXT" chain and IMIX packet size:

```
{
   "flow_count": 10000,
   "service_chain": "EXT",
    "traffic": {
        "profile": "traffic_profile_IMIX"
    },
}
```

A short run of 5 seconds at a fixed rate of 1Mpps (and everything else same as the default configuration):

```
{
   "duration": 5,
   "rate": "1Mpps"
}
```

Example of interaction with the NFVbench server using HTTP and curl

HTTP requests can be sent directly to the NFVbench server from CLI using curl from any host that can connect to the server (here we run it from the local host).

This is a POST request to start a run using the default NFVbench configuration but with traffic generation disabled ("no_traffic" property is set to true):

```
[root@sjc04-pod3-mgmt ~]# curl -H "Accept: application/json" -H "Content-type:␣
↪application/json" -X POST -d '{"no_traffic":true}' http://127.0.0.1:7555/start_run
{
  "error_message": "nfvbench run still pending",
  "status": "PENDING"
}
[root@sjc04-pod3-mgmt ~]#
```

This request will return immediately with status set to "PENDING" if the request was started successfully.

The status can be polled until the run completes. Here the poll returns a "PENDING" status, indicating the run is still not completed:

```
[root@sjc04-pod3-mgmt ~]# curl -G http://127.0.0.1:7555/status
{
  "error_message": "nfvbench run still pending",
  "status": "PENDING"
}
[root@sjc04-pod3-mgmt ~]#
```

Finally, the status request returns a "OK" status along with the full results (truncated here):

```
[root@sjc04-pod3-mgmt ~]# curl -G http://127.0.0.1:7555/status
{
  "result": {
```

(continues on next page)

(continued from previous page)

```
    "benchmarks": {
      "network": {
        "service_chain": {
          "PVP": {
            "result": {
              "bidirectional": true,
              "compute_nodes": {
                "nova:sjc04-pod3-compute-4": {
                  "bios_settings": {
                    "Adjacent Cache Line Prefetcher": "Disabled",
                    "All Onboard LOM Ports": "Enabled",
                    "All PCIe Slots OptionROM": "Enabled",
                    "Altitude": "300 M",
...

    "date": "2017-03-31 22:15:41",
    "nfvbench_version": "0.3.5",
    "openstack_spec": {
      "encaps": "VxLAN",
      "vswitch": "VTS"
    }
  },
  "status": "OK"
}
[root@sjc04-pod3-mgmt ~]#
```

Example of interaction with the NFVbench server using a python CLI app (nfvbench_client)

The module client/client.py contains an example of python class that can be used to control the NFVbench server from a python app using HTTP or WebSocket/SocketIO.

The module client/nfvbench_client.py has a simple main application to control the NFVbench server from CLI. The "nfvbench_client" wrapper script can be used to invoke the client front end (this wrapper is pre-installed in the NFVbench container)

Example of invocation of the nfvbench_client front end, from the host (assume the name of the NFVbench container is "nfvbench"), use the default NFVbench configuration but do not generate traffic (no_traffic property set to true, the full json result is truncated here):

```
[root@sjc04-pod3-mgmt ~]# docker exec -it nfvbench nfvbench_client -c '{"no_traffic
↪":true}' http://127.0.0.1:7555
{u'status': u'PENDING', u'error_message': u'nfvbench run still pending'}
{u'status': u'PENDING', u'error_message': u'nfvbench run still pending'}
{u'status': u'PENDING', u'error_message': u'nfvbench run still pending'}

{u'status': u'OK', u'result': {u'date': u'2017-03-31 22:04:59', u'nfvbench_version': u
↪'0.3.5',
 u'config': {u'compute_nodes': None, u'compute_node_user': u'root', u'traffic_
↪generator': {u'tg_gateway_ip_addrs': [u'1.1.0.100', u'2.2.0.100'], u'ip_addrs_step
↪': u'0.0.0.1',
 u'step_mac': None, u'generator_profile': [{u'intf_speed': u'', u'interfaces': [{u'pci
↪': u'0a:00.0', u'port': 0, u'vlan': 1998, u'switch_port': None},

...

[root@sjc04-pod3-mgmt ~]#
```

The http interface is used unless –use-socketio is defined.

Example of invocation using Websocket/SocketIO, execute NFVbench using the default configuration but with a duration of 5 seconds and a fixed rate run of 5kpps.

```
[root@sjc04-pod3-mgmt ~]# docker exec -it nfvbench nfvbench_client -c '{"duration":5,
↪"rate":"5kpps"}' --use-socketio  http://127.0.0.1:7555 >results.json
```

6.1.9 Frequently Asked Questions

General Questions

Can NFVbench be used without OpenStack

Yes. This can be done using the EXT chain mode, with or without ARP (depending on whether your systen under test can do routing) and by setting the openrc_file property to empty in the NFVbench configuration.

Can NFVbench be used with a different traffic generator than TRex?

This is possible but requires developing a new python class to manage the new traffic generator interface.

Can I connect Trex directly to my compute node?

Yes.

Can I drive NFVbench using a REST interface?

NFVbench can run in server mode and accept HTTP or WebSocket/SocketIO events to run any type of measurement (fixed rate run or NDR_PDR run) with any run configuration.

Can I run NFVbench on a Cisco UCS-B series blade?

Yes provided your UCS-B series server has a Cisco VIC 1340 (with a recent firmware version). TRex will require VIC firmware version 3.1(2) or higher for blade servers (which supports more filtering capabilities). In this setting, the 2 physical interfaces for data plane traffic are simply hooked to the UCS-B fabric interconnect (no need to connect to a switch).

Troubleshooting

TrafficClientException: End-to-end connectivity cannot be ensured

Prior to running a benchmark, NFVbench will make sure that traffic is passing in the service chain by sending a small flow of packets in each direction and verifying that they are received back at the other end. This exception means that NFVbench cannot pass any traffic in the service chain.

The most common issues that prevent traffic from passing are: - incorrect wiring of the NFVbench/TRex interfaces - incorrect vlan_tagging setting in the NFVbench configuration, this needs to match how the NFVbench ports on the switch are configured (trunk or access port)

- if the switch port is configured as access port, you must disable vlan_tagging in the NFVbench configuration
- of the switch port is configured as trunk (recommended method), you must enable it