Yardstick

Release Latest

Open Platform for NFV

Feb 04, 2020

CONTENTS

CHAPTER

ONE

YARDSTICK RELEASE NOTE

1.1 Yardstick Release Notes

1.1.1 Abstract

This document compiles the release notes for the Iruya release of OPNFV Yardstick.

1.1.2 Version History

Date	*Version*	*Comment*
Jan 10, 2020	9.0.0	Yardstick for Iruya release

1.1.3 Important Notes

The software delivered in the OPNFV Yardstick Project, comprising the *Yardstick framework*, and the *Yardstick test cases* is a realization of the methodology in ETSI-ISG NFV-TST001.

The *Yardstick* framework is *installer*, *infrastructure* and *application* independent.

1.1.4 OPNFV Iruya Release

This Iruya release provides *Yardstick* as a framework for NFVI testing and OPNFV feature testing, automated in the OPNFV CI pipeline, including:

- Documentation generated with Sphinx
  - User Guide
  - Developer Guide
  - Release notes (this document)
  - Results
- Automated Yardstick test suite (daily, weekly)
  - Jenkins Jobs for OPNFV community labs

- Automated Yardstick test results visualization
  - Dashboard using Grafana (user:opnfv/password: opnfv), influxDB is used as backend
- Yardstick framework source code
- Yardstick test cases yaml files
- Yardstick plug-in configuration yaml files, plug-in install/remove scripts

For Iruya release, the *Yardstick framework* is used for the following testing:

- OPNFV platform testing - generic test cases to measure the categories:
  - Compute
  - Network
  - Storage
- OPNFV platform network service benchmarking (NSB)
  - NSB
- Test cases for the following OPNFV Projects:
  - Container4NFV
  - High Availability
  - IPv6
  - KVM
  - Parser
  - StorPerf
  - VSperf

The *Yardstick framework* is developed in the OPNFV community, by the Yardstick team.

Note: The test case description template used for the Yardstick test cases is based on the document ETSI-ISG NFV-TST001; the results report template used for the Yardstick results is based on the IEEE Std 829-2008.

1.1.5 Release Data

Project	Yardstick
Repo/tag	yardstick/opnfv-9.0.0
Yardstick Docker image tag	opnfv-9.0.0
Release designation	Iruya 9.0
Release date	Jan 10, 2020
Purpose of the delivery	OPNFV Iruya 9.0.0

1.1.6 Deliverables

Documents

- User Guide: <yardstick:userguide>

- Developer Guide: <yardstick:devguide>

Software Deliverables

- The Yardstick Docker image: https://hub.docker.com/r/opnfv/yardstick (tag: opnfv-9.0.0)

List of Contexts

Context	Description
Heat	Models orchestration using OpenStack Heat
Node	Models Baremetal, Controller, Compute
Standalone	Models VM running on Non-Managed NFVi
Kubernetes	Models VM running on Non-Managed NFVi

List of Runners

Runner	Description
Arithmetic	Steps every run arithmetically according to specified input value
Duration	Runs for a specified period of time
Iteration	Runs for a specified number of iterations
IterationIPC	Runs a configurable number of times before it returns. Each iteration has a configurable timeout.
Sequence	Selects input value to a scenario from an input file and runs all entries sequentially
Dynamictp	A runner that searches for the max throughput with binary search
Search	A runner that runs a specific time before it returns

List of Scenarios

Category	Delivered
Availability	Attacker: • baremetal, process HA tools: • check host, openstack, process, service • kill process • start/stop service Monitor: • command, process
Compute	• cpuload • cyclictest • lmbench • lmbench_cache • perf • unixbench • ramspeed • cachestat • memeoryload • computecapacity • SpecCPU2006
Networking	• iperf3 • netperf • netperf_node • ping • ping6 • pktgen • sfc • sfc with tacker • networkcapacity • netutilization • nstat • pktgenDPDK
Parser	Tosca2Heat
Storage	• fio • bonnie++ • storagecapacity
StorPerf	storperf
NSB	vFW thoughput test case

New Test cases

opnfv_yardstick_tc015: Processing speed with impact on energy consumption and CPU load.

The purpose of TC015 is to evaluate the IaaS compute performance with regards to CPU processing speed with its

impact on the energy consumption. It measures score of single cpu running and parallel running. Energy consumption and cpu load are monitored while the cpu test is running. The purpose is also to be able to spot the trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations, different server types.

1.1.7 Version Change

Module Version Changes

This is the seventh tracked release of Yardstick. It is based on following upstream versions:

- OpenStack Stein

Document Version Changes

This is the seventh tracked version of the Yardstick framework in OPNFV. It includes the following documentation updates:

- Yardstick User Guide:
- Yardstick Developer Guide
- Yardstick Release Notes for Yardstick: this document

Feature additions

1.1.8 Scenario Matrix

1.1.9 Test results

Test results are available in:

- jenkins logs on CI: https://build.opnfv.org/ci/view/yardstick/

Known Issues/Faults

Corrected Faults

1.1.10 Iruya 9.0.0 known restrictions/issues

1.1.11 Useful links

- wiki project page: https://wiki.opnfv.org/display/yardstick/Yardstick
- wiki Yardstick Iruya release planning page: https://wiki.opnfv.org/display/yardstick/Release+Iruya
- Yardstick repo: https://git.opnfv.org/yardstick
- Yardstick CI dashboard: https://build.opnfv.org/ci/view/yardstick
- Yardstick grafana dashboard: http://testresults.opnfv.org/grafana/
- Yardstick IRC channel: #opnfv-yardstick

CHAPTER

TWO

YARDSTICK USER GUIDE

2.1 Introduction

Welcome to Yardstick's documentation !

Yardstick is an OPNFV Project.

The project's goal is to verify infrastructure compliance, from the perspective of a Virtual Network Function (*VNF*).

The Project's scope is the development of a test framework, *Yardstick*, test cases and test stimuli to enable Network Function Virtualization Infrastructure (*NFVI*) verification.

Yardstick is used in OPNFV for verifying the OPNFV infrastructure and some of the OPNFV features. The *Yardstick* framework is deployed in several OPNFV community labs. It is *installer*, *infrastructure* and *application* independent.

See also:

Pharos for information on OPNFV community labs and this Presentation for an overview of *Yardstick*

2.1.1 About This Document

This document consists of the following chapters:

- Chapter *Introduction* provides a brief introduction to *Yardstick* project's background and describes the structure of this document.
- Chapter *Methodology* describes the methodology implemented by the *Yardstick* Project for *NFVI* verification.
- Chapter *Architecture* provides information on the software architecture of *Yardstick*.
- Chapter *Yardstick Installation* provides instructions to install *Yardstick*.
- Chapter *Yardstick Usage* provides information on how to use *Yardstick* to run and create testcases.
- Chapter *Installing a plug-in into Yardstick* provides information on how to integrate other OPNFV testing projects into *Yardstick*.
- Chapter *Store Other Project's Test Results in InfluxDB* provides inforamtion on how to run plug-in test cases and store test results into community's InfluxDB.
- Chapter *Grafana dashboard* provides inforamtion on *Yardstick* grafana dashboard and how to add a dashboard into *Yardstick* grafana dashboard.
- Chapter *Yardstick Restful API* provides inforamtion on *Yardstick* ReST API and how to use *Yardstick* API.
- Chapter *Yardstick User Interface* provides inforamtion on how to use yardstick report CLI to view the test result in table format and also values pinned on to a graph

- Chapter *Network Services Benchmarking (NSB)* describes the methodology implemented by the Yardstick - Network service benchmarking to test real world usecase for a given VNF.
- Chapter *NSB Installation* provides instructions to install *Yardstick - Network Service Benchmarking (NSB) testing*.
- Chapter *Yardstick - NSB Testing - Operation* provides information on running *NSB*
- Chapter *Yardstick Test Cases* includes a list of available *Yardstick* test cases.

2.1.2 Contact Yardstick

Feedback? Contact us

2.2 Methodology

2.2.1 Abstract

This chapter describes the methodology implemented by the Yardstick project for verifying the *NFVI* from the perspective of a *VNF*.

2.2.2 ETSI-NFV

The document ETSI GS NFV-TST001, "Pre-deployment Testing; Report on Validation of NFV Environments and Services", recommends methods for pre-deployment testing of the functional components of an NFV environment.

The Yardstick project implements the methodology described in chapter 6, "Pre- deployment validation of NFV infrastructure".

The methodology consists in decomposing the typical *VNF* work-load performance metrics into a number of characteristics/performance vectors, which each can be represented by distinct test-cases.

The methodology includes five steps:

- ***Step1:*** **Define Infrastruture - the Hardware, Software and corresponding** configuration target for validation; the OPNFV infrastructure, in OPNFV community labs.
- ***Step2:*** **Identify *VNF* type - the application for which the** infrastructure is to be validated, and its requirements on the underlying infrastructure.
- ***Step3:*** **Select test cases - depending on the workload that represents the** application for which the infrastruture is to be validated, the relevant test cases amongst the list of available Yardstick test cases.
- ***Step4:*** **Execute tests - define the duration and number of iterations for the** selected test cases, tests runs are automated via OPNFV Jenkins Jobs.
- *Step5:* Collect results - using the common API for result collection.

See also:

Yardsticktst for material on alignment ETSI TST001 and Yardstick.

2.2.3 Metrics

The metrics, as defined by ETSI GS NFV-TST001, are shown in *Table1*, *Table2* and *Table3*.

In OPNFV Colorado release, generic test cases covering aspects of the listed metrics are available; further OPNFV releases will provide extended testing of these metrics. The view of available Yardstick test cases cross ETSI definitions in *Table1*, *Table2* and *Table3* is shown in *Table4*. It shall be noticed that the Yardstick test cases are examples, the test duration and number of iterations are configurable, as are the System Under Test (SUT) and the attributes (or, in Yardstick nomemclature, the scenario options).

Table 1 - Performance/Speed Metrics

Category	Performance/Speed
Compute	• Latency for random memory access • Latency for cache read/write operations • Processing speed (instructions per second) • Throughput for random memory access (bytes per second)
Network	• Throughput per NFVI node (frames/byte per second) • Throughput provided to a VM (frames/byte per second) • Latency per traffic flow • Latency between VMs • Latency between NFVI nodes • Packet delay variation (jitter) between VMs • Packet delay variation (jitter) between NFVI nodes
Storage	• Sequential read/write IOPS • Random read/write IOPS • Latency for storage read/write operations • Throughput for storage read/write operations • Energy consumption in Watts (transversal to all others scenario)
Energy	

Table 2 - Capacity/Scale Metrics

Category	Capacity/Scale
Compute	• Number of cores and threads- Available memory size • Cache size • Processor utilization (max, average, standard deviation) • Memory utilization (max, average, standard deviation) • Cache utilization (max, average, standard deviation)
Network	• Number of connections • Number of frames sent/received • Maximum throughput between VMs (frames/byte per second) • Maximum throughput between NFVI nodes (frames/byte per second) • Network utilization (max, average, standard deviation) • Number of traffic flows
Storage	• Storage/Disk size • Capacity allocation (block-based, object-based) • Block size • Maximum sequential read/write IOPS • Maximum random read/write IOPS • Disk utilization (max, average, standard deviation)

Table 3 - Availability/Reliability Metrics

Category	Availability/Reliability
Compute	• Processor availability (Error free processing time) • Memory availability (Error free memory time) • Processor mean-time-to-failure • Memory mean-time-to-failure • Number of processing faults per second
Network	• NIC availability (Error free connection time) • Link availability (Error free transmission time) • NIC mean-time-to-failure • Network timeout duration due to link failure • Frame loss rate
Storage	• Disk availability (Error free disk access time) • Disk mean-time-to-failure • Number of failed storage read/write operations per second

Table 4 - Yardstick Generic Test Cases

Category	Performance/Speed	Capacity/Scale	Availability/Reliability
Compute	TC003[1] TC004 TC010 TC012 TC014 TC015 TC069	TC003[1] TC004 TC024 TC055	TC013[1] TC015[1]
Network	TC001 TC002 TC009 TC011 TC042 TC043	TC044 TC073 TC075	TC016[1] TC018[1]
Storage	TC005	TC063	TC017[1]

Note: The description in this OPNFV document is intended as a reference for users to understand the scope of the Yardstick Project and the deliverables of the Yardstick framework. For complete description of the methodology, please refer to the ETSI document.

2.3 Architecture

2.3.1 Abstract

This chapter describes the Yardstick framework software architecture. We will introduce it from Use Case View, Logical View, Process View and Deployment View. More technical details will be introduced in this chapter.

2.3.2 Overview

[1] To be included in future deliveries.

Architecture overview

Yardstick is mainly written in Python, and test configurations are made in YAML. Documentation is written in reStructuredText format, i.e. .rst files. Yardstick is inspired by Rally. Yardstick is intended to run on a computer with access and credentials to a cloud. The test case is described in a configuration file given as an argument.

How it works: the benchmark task configuration file is parsed and converted into an internal model. The context part of the model is converted into a Heat template and deployed into a stack. Each scenario is run using a runner, either serially or in parallel. Each runner runs in its own subprocess executing commands in a VM using SSH. The output of each scenario is written as json records to a file or influxdb or http server, we use influxdb as the backend, the test result will be shown with grafana.

Concept

Benchmark - assess the relative performance of something

Benchmark configuration file - describes a single test case in yaml format

Context - The set of Cloud resources used by a scenario, such as user names, image names, affinity rules and network configurations. A context is converted into a simplified Heat template, which is used to deploy onto the Openstack environment.

Data - Output produced by running a benchmark, written to a file in json format

Runner - Logic that determines how a test scenario is run and reported, for example the number of test iterations, input value stepping and test duration. Predefined runner types exist for re-usage, see *Runner types*.

Scenario - Type/class of measurement for example Ping, Pktgen, (Iperf, LmBench, ...)

SLA - Relates to what result boundary a test case must meet to pass. For example a latency limit, amount or ratio of lost packets and so on. Action based on *SLA* can be configured, either just to log (monitor) or to stop further testing (assert). The *SLA* criteria is set in the benchmark configuration file and evaluated by the runner.

Runner types

There exists several predefined runner types to choose between when designing a test scenario:

Arithmetic: Every test run arithmetically steps the specified input value(s) in the test scenario, adding a value to the previous input value. It is also possible to combine several input values for the same test case in different combinations.

Snippet of an Arithmetic runner configuration:

```
runner:
    type: Arithmetic
    iterators:
    -
      name: stride
      start: 64
      stop: 128
      step: 64
```

Duration: The test runs for a specific period of time before completed.

Snippet of a Duration runner configuration:

```
runner:
  type: Duration
  duration: 30
```

Sequence: The test changes a specified input value to the scenario. The input values to the sequence are specified in a list in the benchmark configuration file.

Snippet of a Sequence runner configuration:

```
runner:
  type: Sequence
  scenario_option_name: packetsize
  sequence:
  - 100
  - 200
  - 250
```

Iteration: Tests are run a specified number of times before completed.

Snippet of an Iteration runner configuration:

```
runner:
  type: Iteration
  iterations: 2
```

2.3.3 Use-Case View

Yardstick Use-Case View shows two kinds of users. One is the Tester who will do testing in cloud, the other is the User who is more concerned with test result and result analyses.

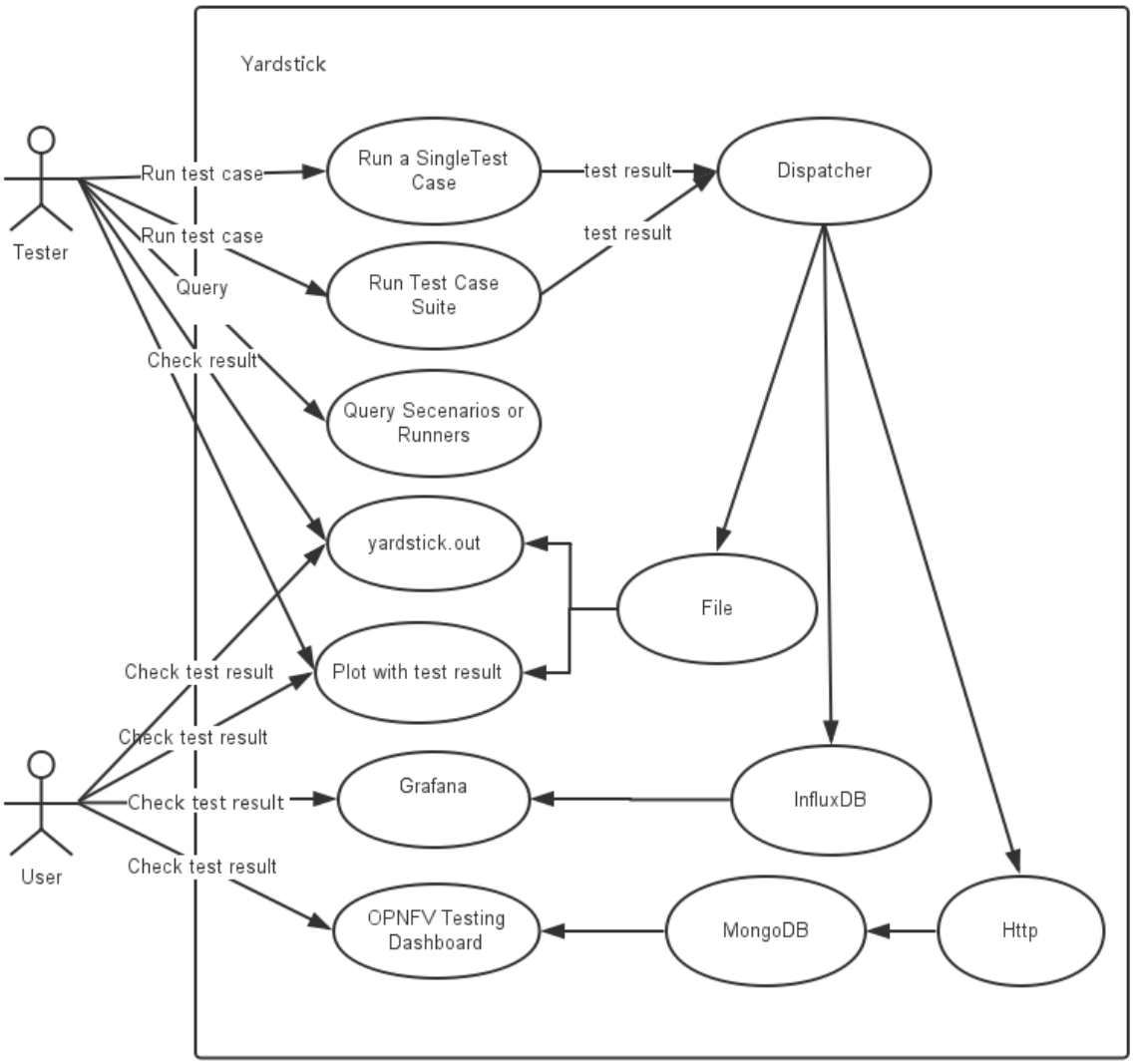
For testers, they will run a single test case or test case suite to verify infrastructure compliance or bencnmark their own infrastructure performance. Test result will be stored by dispatcher module, three kinds of store method (file, influxdb and http) can be configured. The detail information of scenarios and runners can be queried with CLI by testers.

For users, they would check test result with four ways.

If dispatcher module is configured as file(default), there are two ways to check test result. One is to get result from yardstick.out (default path: /tmp/yardstick.out), the other is to get plot of test result, it will be shown if users execute command "yardstick-plot".

If dispatcher module is configured as influxdb, users will check test result on Grafana which is most commonly used for visualizing time series data.

If dispatcher module is configured as http, users will check test result on OPNFV testing dashboard which use MongoDB as backend.



2.3.4 Logical View

Yardstick Logical View describes the most important classes, their organization, and the most important use-case realizations.

Main classes:

TaskCommands - "yardstick task" subcommand handler.

HeatContext - Do test yaml file context section model convert to HOT, deploy and undeploy Openstack heat stack.

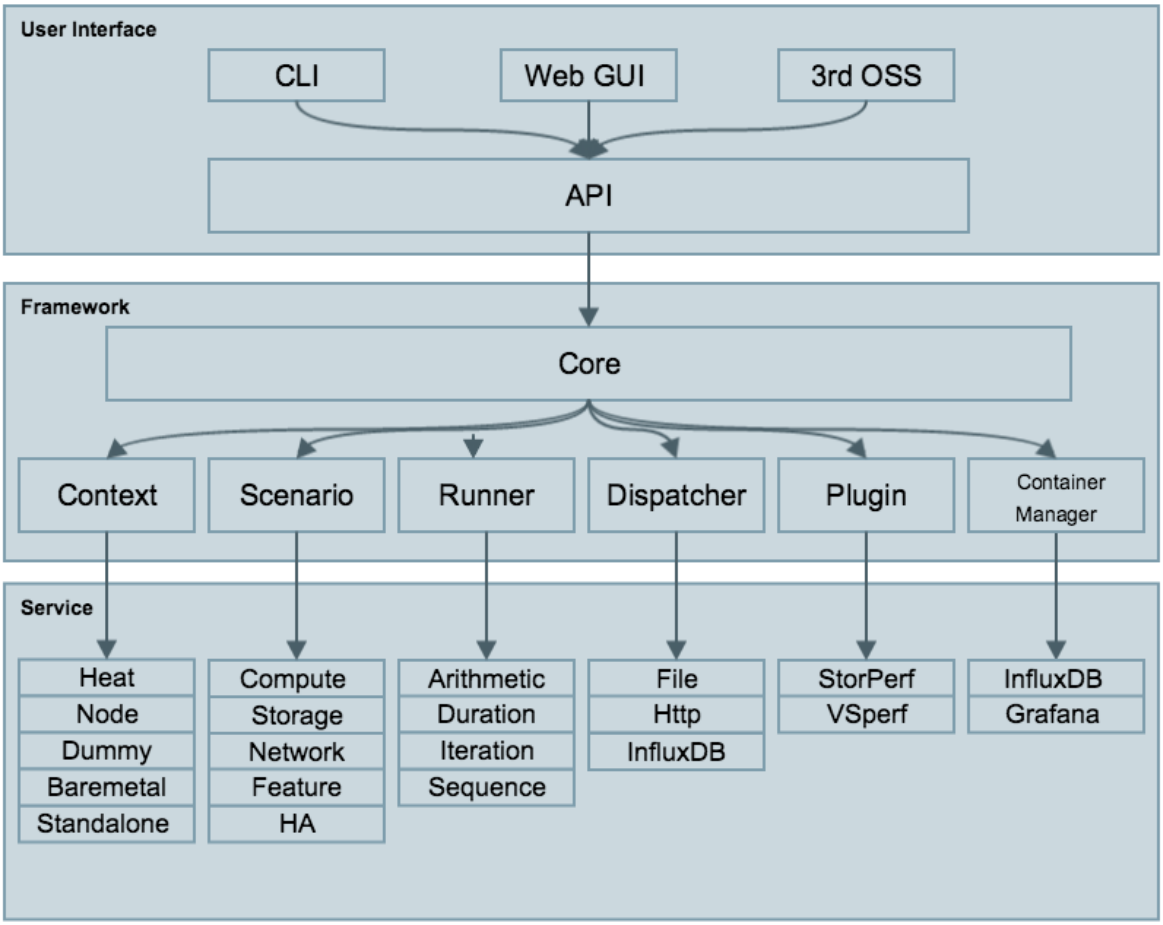
Runner - Logic that determines how a test scenario is run and reported.

TestScenario - Type/class of measurement for example Ping, Pktgen, (Iperf, LmBench, . . .)

Dispatcher - Choose user defined way to store test results.

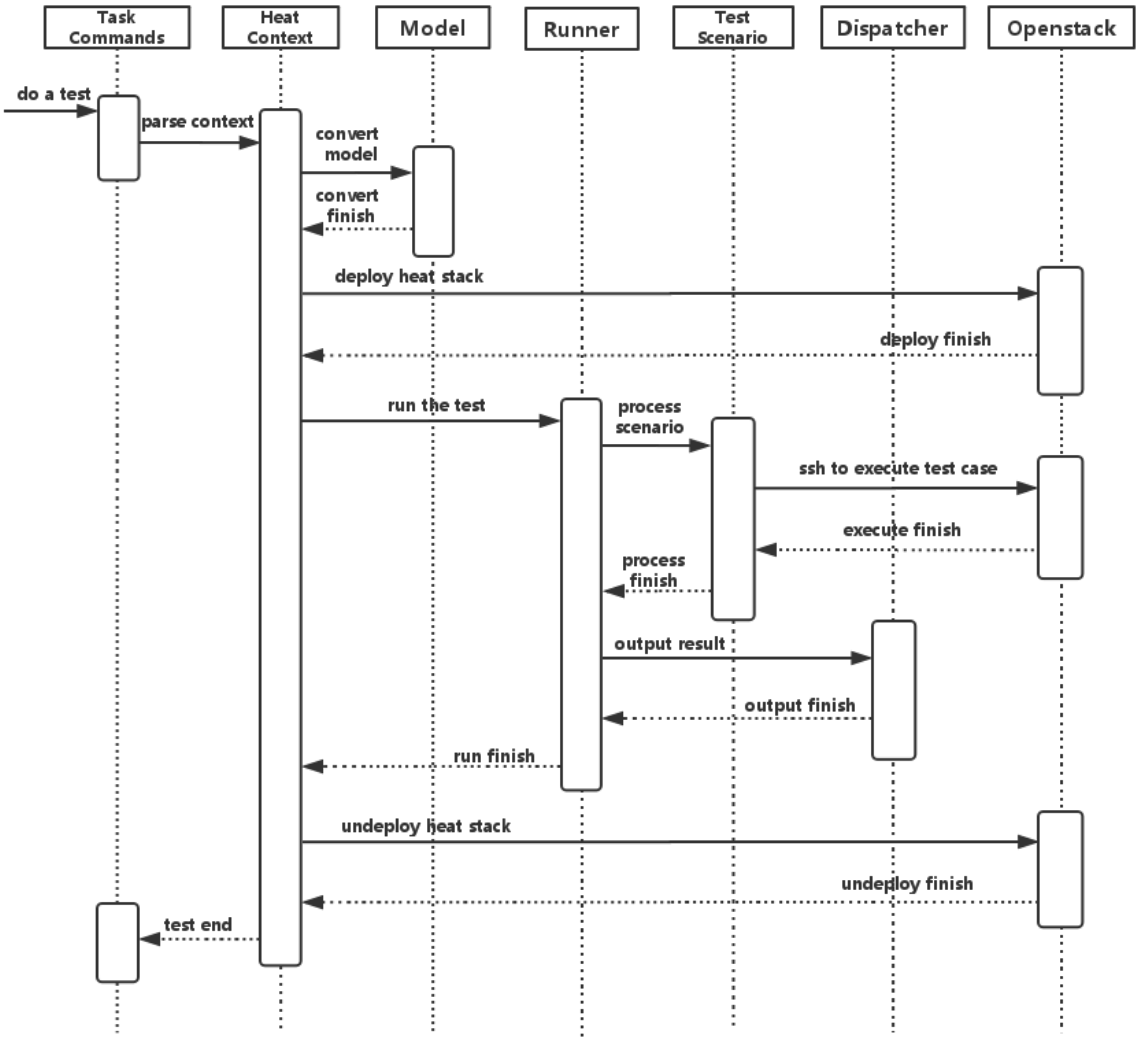
TaskCommands is the "yardstick task" subcommand's main entry. It takes yaml file (e.g. test.yaml) as input, and uses HeatContext to convert the yaml file's context section to HOT. After Openstack heat stack is deployed by HeatContext

with the converted HOT, TaskCommands use Runner to run specified TestScenario. During first runner initialization, it will create output process. The output process use Dispatcher to push test results. The Runner will also create a process to execute TestScenario. And there is a multiprocessing queue between each runner process and output process, so the runner process can push the real-time test results to the storage media. TestScenario is commonly connected with VMs by using ssh. It sets up VMs and run test measurement scripts through the ssh tunnel. After all TestScenaio is finished, TaskCommands will undeploy the heat stack. Then the whole test is finished.



2.3.5 Process View (Test execution flow)

Yardstick process view shows how yardstick runs a test case. Below is the sequence graph about the test execution flow using heat context, and each object represents one module in yardstick:

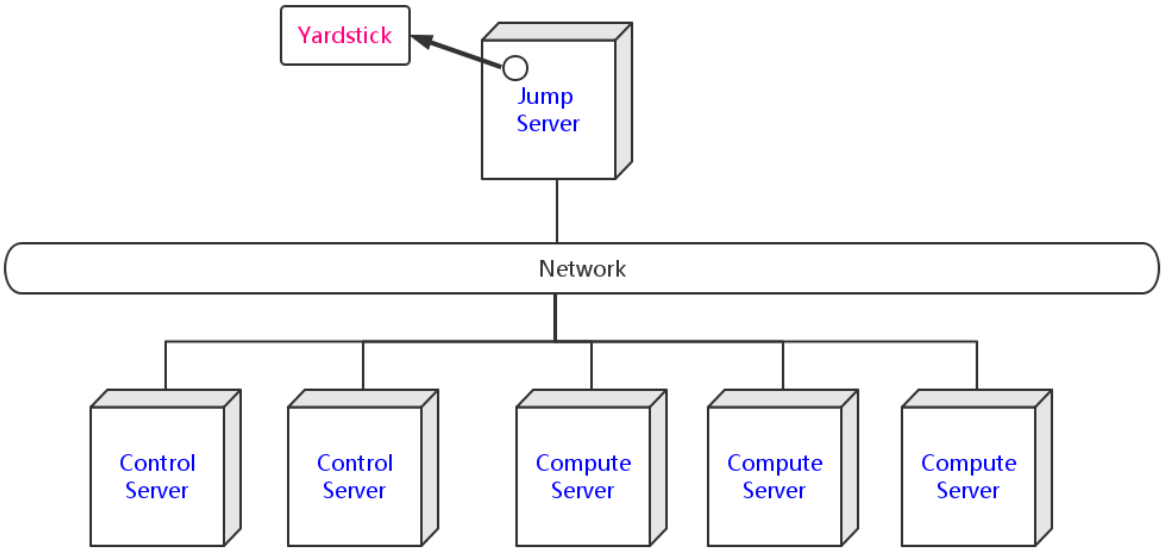


A user wants to do a test with yardstick. He can use the CLI to input the command to start a task. "TaskCommands" will receive the command and ask "HeatContext" to parse the context. "HeatContext" will then ask "Model" to convert the model. After the model is generated, "HeatContext" will inform "Openstack" to deploy the heat stack by heat template. After "Openstack" deploys the stack, "HeatContext" will inform "Runner" to run the specific test case.

Firstly, "Runner" would ask "TestScenario" to process the specific scenario. Then "TestScenario" will start to log on the openstack by ssh protocal and execute the test case on the specified VMs. After the script execution finishes, "TestScenario" will send a message to inform "Runner". When the testing job is done, "Runner" will inform "Dispatcher" to output the test result via file, influxdb or http. After the result is output, "HeatContext" will call "Openstack" to undeploy the heat stack. Once the stack is undepoyed, the whole test ends.

2.3.6 Deployment View

Yardstick deployment view shows how the yardstick tool can be deployed into the underlying platform. Generally, yardstick tool is installed on JumpServer(see *07-installation* for detail installation steps), and JumpServer is connected with other control/compute servers by networking. Based on this deployment, yardstick can run the test cases on these hosts, and get the test result for better showing.



2.3.7 Yardstick Directory structure

yardstick/ - Yardstick main directory.

***tests/ci/* - Used for continuous integration of Yardstick at different PODs and** with support for different installers.

***docs/* - All documentation is stored here, such as configuration guides,** user guides and Yardstick test case descriptions.

etc/ - Used for test cases requiring specific POD configurations.

***samples/* - test case samples are stored here, most of all scenario and** feature samples are shown in this directory.

***tests/* - The test cases run to verify the NFVI (*opnfv/*) are stored here.** The configurations of what to run daily and weekly at the different PODs are also located here.

***tools/* - Contains tools to build image for VMs which are deployed by Heat.** Currently contains how to build the yardstick-image with the different tools that are needed from within the image.

plugin/ - Plug-in configuration files are stored here.

***yardstick/* - Contains the internals of Yardstick: *Runners*,** *Scenarios*, *Contexts*, CLI parsing, keys, plotting tools, dispatcher, plugin install/remove scripts and so on.

***yardstick/tests* - The Yardstick internal tests (*functional/* and *unit/*)** are stored here.

2.4 Yardstick Installation

Yardstick supports installation by Docker or directly in Ubuntu. The installation procedure for Docker and direct installation are detailed in the sections below.

To use Yardstick you should have access to an OpenStack environment, with at least Nova, Neutron, Glance, Keystone and Heat installed.

The steps needed to run Yardstick are:

1. Install Yardstick.
2. Load OpenStack environment variables.
3. Create Yardstick flavor.
4. Build a guest image and load it into the OpenStack environment.
5. Create the test configuration `.yaml` file and run the test case/suite.

2.4.1 Prerequisites

The OPNFV deployment is out of the scope of this document and can be found in User Guide & Configuration Guide. The OPNFV platform is considered as the System Under Test (SUT) in this document.

Several prerequisites are needed for Yardstick:

1. A Jumphost to run Yardstick on
2. A Docker daemon or a virtual environment installed on the Jumphost
3. A public/external network created on the SUT
4. Connectivity from the Jumphost to the SUT public/external network

Note: *Jumphost* refers to any server which meets the previous requirements. Normally it is the same server from where the OPNFV deployment has been triggered.

Warning: Connectivity from Jumphost is essential and it is of paramount importance to make sure it is working before even considering to install and run Yardstick. Make also sure you understand how your networking is designed to work.

Note: If your Jumphost is operating behind a company http proxy and/or Firewall, please first consult *Proxy Support* section which is towards the end of this document. That section details some tips/tricks which *may* be of help in a proxified environment.

2.4.2 Install Yardstick using Docker (first option) (recommended)

Yardstick has a Docker image. It is recommended to use this Docker image to run Yardstick test.

Prepare the Yardstick container

Install docker on your guest system with the following command, if not done yet:

```
wget -qO- https://get.docker.com/ | sh
```

Pull the Yardstick Docker image (`opnfv/yardstick`) from the public dockerhub registry under the OPNFV account in dockerhub, with the following docker command:

```
sudo -EH docker pull opnfv/yardstick:stable
```

After pulling the Docker image, check that it is available with the following docker command:

```
[yardsticker@jumphost ~]$ docker images
REPOSITORY          TAG       IMAGE ID        CREATED      SIZE
opnfv/yardstick     stable    a4501714757a    1 day ago    915.4 MB
```

Run the Docker image to get a Yardstick container:

```
docker run -itd --privileged -v /var/run/docker.sock:/var/run/docker.sock \
   -p 8888:5000 --name yardstick opnfv/yardstick:stable
```

Description of the parameters used with `docker run` command

Parameters	Detail
-itd	-i: interactive, Keep STDIN open even if not attached
	-t: allocate a pseudo-TTY detached mode, in the background
–privileged	If you want to build `yardstick-image` in Yardstick container, this parameter is needed
-p 8888:5000	Redirect the a host port (8888) to a container port (5000)
-v /var/run/docker.sock :/var/run/docker.sock	If you want to use yardstick env grafana/influxdb to create a grafana/influxdb container out of Yardstick container
–name yardstick	The name for this container

If the host is restarted

The yardstick container must be started if the host is rebooted:

```
docker start yardstick
```

Configure the Yardstick container environment

There are three ways to configure environments for running Yardstick, explained in the following sections. Before that, access the Yardstick container:

```
docker exec -it yardstick /bin/bash
```

and then configure Yardstick environments in the Yardstick container.

Using the CLI command `env prepare` (first way) (recommended)

In the Yardstick container, the Yardstick repository is located in the `/home/opnfv/repos` directory. Yardstick provides a CLI to prepare OpenStack environment variables and create Yardstick flavor and guest images automatically:

```
yardstick env prepare
```

Note: Since Euphrates release, the above command will not be able to automatically configure the `/etc/yardstick/openstack.creds` file. So before running the above command, it is necessary to create the `/etc/yardstick/openstack.creds` file and save OpenStack environment variables into it manually. If you have the openstack credential file saved outside the Yardstick Docker container, you can do this easily by mapping the credential file into Yardstick container using:

```
'-v /path/to/credential_file:/etc/yardstick/openstack.creds'
```

when running the Yardstick container. For details of the required OpenStack environment variables please refer to section *Export OpenStack environment variables*.

The `env prepare` command may take up to 6-8 minutes to finish building yardstick-image and other environment preparation. Meanwhile if you wish to monitor the env prepare process, you can enter the Yardstick container in a new terminal window and execute the following command:

```
tail -f /var/log/yardstick/uwsgi.log
```

Manually exporting the env variables and initializing OpenStack (second way)

Export OpenStack environment variables

Before running Yardstick it is necessary to export OpenStack environment variables:

```
source openrc
```

Environment variables in the `openrc` file have to include at least:

```
OS_AUTH_URL
OS_USERNAME
OS_PASSWORD
OS_PROJECT_NAME
EXTERNAL_NETWORK
```

A sample `openrc` file may look like this:

```
export OS_PASSWORD=console
export OS_PROJECT_NAME=admin
export OS_AUTH_URL=http://172.16.1.222:35357/v2.0
export OS_USERNAME=admin
export OS_VOLUME_API_VERSION=2
export EXTERNAL_NETWORK=net04_ext
```

Manual creation of Yardstick flavor and guest images

Before executing Yardstick test cases, make sure that Yardstick flavor and guest image are available in OpenStack. Detailed steps about creating the Yardstick flavor and building the Yardstick guest image can be found below.

Most of the sample test cases in Yardstick are using an OpenStack flavor called `yardstick-flavor` which deviates from the OpenStack standard `m1.tiny` flavor by the disk size; instead of 1GB it has 3GB. Other parameters are the same as in `m1.tiny`.

Create `yardstick-flavor`:

```
openstack flavor create --disk 3 --vcpus 1 --ram 512 --swap 100 \
   yardstick-flavor
```

Most of the sample test cases in Yardstick are using a guest image called `yardstick-image` which deviates from an Ubuntu Cloud Server image containing all the required tools to run test cases supported by Yardstick. Yardstick has a tool for building this custom image. It is necessary to have `sudo` rights to use this tool.

Also you may need install several additional packages to use this tool, by follwing the commands below:

```
sudo -EH apt-get update && sudo -EH apt-get install -y qemu-utils kpartx
```

This image can be built using the following command in the directory where Yardstick is installed:

```
export YARD_IMG_ARCH='amd64'
echo "Defaults env_keep += \'YARD_IMG_ARCH\'" | sudo tee --append \
   /etc/sudoers > /dev/null
sudo -EH tools/yardstick-img-modify tools/ubuntu-server-cloudimg-modify.sh
```

Warning: Before building the guest image inside the Yardstick container, make sure the container is granted with privilege. The script will create files by default in `/tmp/workspace/yardstick` and the files will be owned by root.

The created image can be added to OpenStack using the OpenStack client or via the OpenStack Dashboard:

```
openstack image create --disk-format qcow2 --container-format bare \
   --public --file /tmp/workspace/yardstick/yardstick-image.img \
    yardstick-image
```

Some Yardstick test cases use a Cirros 0.3.5 image and/or a Ubuntu 16.04 image. Add Cirros and Ubuntu images to OpenStack:

```
openstack image create --disk-format qcow2 --container-format bare \
   --public --file $cirros_image_file cirros-0.3.5
openstack image create --disk-format qcow2 --container-format bare \
   --file $ubuntu_image_file Ubuntu-16.04
```

Automatic initialization of OpenStack (third way)

Similar to the second way, the first step is also to *Export OpenStack environment variables*. Then the following steps should be done.

Automatic creation of Yardstick flavor and guest images

Yardstick has a script for automatically creating Yardstick flavor and building Yardstick guest images. This script is mainly used for CI and can be also used in the local environment:

```
source $YARDSTICK_REPO_DIR/tests/ci/load_images.sh
```

The Yardstick container GUI

In Euphrates release, Yardstick implemented a GUI for Yardstick Docker container. After booting up Yardstick container, you can visit the GUI at `<container_host_ip>:8888/gui/index.html`.

For usage of Yardstick GUI, please watch our demo video at Yardstick GUI demo.

Note: The Yardstick GUI is still in development, the GUI layout and features may change.

Delete the Yardstick container

If you want to uninstall Yardstick, just delete the Yardstick container:

```
sudo docker stop yardstick && docker rm yardstick
```

2.4.3 Install Yardstick directly in Ubuntu (second option)

Alternatively you can install Yardstick framework directly in Ubuntu or in an Ubuntu Docker image. No matter which way you choose to install Yardstick, the following installation steps are identical.

If you choose to use the Ubuntu Docker image, you can pull the Ubuntu Docker image from Docker hub:

```
sudo -EH docker pull ubuntu:16.04
```

Install Yardstick

Prerequisite preparation:

```
sudo -EH apt-get update && sudo -EH apt-get install -y \
   git python-setuptools python-pip
sudo -EH easy_install -U setuptools==30.0.0
sudo -EH pip install appdirs==1.4.0
sudo -EH pip install virtualenv
```

Download the source code and install Yardstick from it:

```
git clone https://gerrit.opnfv.org/gerrit/yardstick
export YARDSTICK_REPO_DIR=~/yardstick
cd ~/yardstick
sudo -EH ./install.sh
```

If the host is ever restarted, nginx and uwsgi need to be restarted:

```
service nginx restart
uwsgi -i /etc/yardstick/yardstick.ini
```

Configure the Yardstick environment (Todo)

For installing Yardstick directly in Ubuntu, the `yardstick env` command is not available. You need to prepare OpenStack environment variables and create Yardstick flavor and guest images manually.

Uninstall Yardstick

For uninstalling Yardstick, just delete the virtual environment:

```
rm -rf ~/yardstick_venv
```

2.4.4 Install Yardstick directly in OpenSUSE

You can install Yardstick framework directly in OpenSUSE.

Install Yardstick

Prerequisite preparation:

```
sudo -EH zypper -n install -y gcc \
   wget \
   git \
   sshpass \
   qemu-tools \
   kpartx \
   libffi-devel \
   libopenssl-devel \
   python \
   python-devel \
   python-virtualenv \
   libxml2-devel \
   libxslt-devel \
   python-setuptools-git
```

Create a virtual environment:

```
virtualenv ~/yardstick_venv
export YARDSTICK_VENV=~/yardstick_venv
source ~/yardstick_venv/bin/activate
sudo -EH easy_install -U setuptools
```

Download the source code and install Yardstick from it:

```
git clone https://gerrit.opnfv.org/gerrit/yardstick
export YARDSTICK_REPO_DIR=~/yardstick
cd yardstick
sudo -EH python setup.py install
sudo -EH pip install -r requirements.txt
```

Install missing python modules:

```
sudo -EH pip install pyyaml \
   oslo_utils \
   oslo_serialization \
   oslo_config \
   paramiko \
   python.heatclient \
   python.novaclient \
   python.glanceclient \
   python.neutronclient \
   scp \
   jinja2
```

Configure the Yardstick environment

Source the OpenStack environment variables:

```
source DEVSTACK_DIRECTORY/openrc
```

Export the Openstack external network. The default installation of Devstack names the external network public:

```
export EXTERNAL_NETWORK=public
export OS_USERNAME=demo
```

Change the API version used by Yardstick to v2.0 (the devstack openrc sets it to v3):

```
export OS_AUTH_URL=http://PUBLIC_IP_ADDRESS:5000/v2.0
```

Uninstall Yardstick

For unistalling Yardstick, just delete the virtual environment:

```
rm -rf ~/yardstick_venv
```

2.4.5 Verify the installation

It is recommended to verify that Yardstick was installed successfully by executing some simple commands and test samples. Before executing Yardstick test cases make sure `yardstick-flavor` and `yardstick-image` can be found in OpenStack and the `openrc` file is sourced. Below is an example invocation of Yardstick `help` command and `ping.py` test sample:

```
yardstick -h
yardstick task start samples/ping.yaml
```

Note: The above commands could be run in both the Yardstick container and the Ubuntu directly.

Each testing tool supported by Yardstick has a sample configuration file. These configuration files can be found in the `samples` directory.

Default location for the output is `/tmp/yardstick.out`.

2.4.6 Automatic installation of Yardstick

Automatic installation can be used as an alternative to the manual by providing parameters for ansible script `install.yaml` in a `nsb_setup.sh` file. Yardstick can be installed on the bare metal and to the container. Yardstick container can be either pulled or built.

Bare metal installation

Modify `nsb_setup.sh` file `install.yaml` parameters to install Yardstick on Ubuntu server:

```
ansible-playbook -i install-inventory.ini install.yaml \
-e IMAGE_PROPERTY='none' \
-e YARDSTICK_DIR=<path to Yardstick folder>
```

Note: By default `INSTALLATION_MODE` is `baremetal`.

Note: No modification in `install-inventory.ini` is needed for Yardstick installation.

Note: To install Yardstick in virtual environment pass parameter `-e VIRTUAL_ENVIRONMENT=True`.

Container installation

Modify `install.yaml` parameters in `nsb_setup.sh` file to pull or build Yardstick container. To pull Yardstick image and start container run:

```
ansible-playbook -i install-inventory.ini install.yaml \
-e IMAGE_PROPERTY='none' \
-e INSTALLATION_MODE=container_pull
```

Note: Yardstick docker image is available for both Ubuntu 16.04 and Ubuntu 18.04. By default Ubuntu 16.04 based docker image is used. To use Ubuntu 18.04 based docker image pass `-i opnfv/yardstick-ubuntu-18.04` parameter to `nsb_setup.sh`.

To build Yardstick image modify Dockerfile as per comments in it and run:

```
cd yardstick
docker build -f docker/Dockerfile -t opnfv/yardstick:<tag> .
```

Note: Yardstick docker image based on Ubuntu 16.04 will be built. Pass `-f docker/Dockerfile_ubuntu18` to build Yardstick docker image based on Ubuntu 18.04.

Note: Add `--build-arg http_proxy=http://<proxy_host>:<proxy_port>` to build docker image if server is behind the proxy.

Parameters for `install.yaml`

Description of the parameters used with `install.yaml`:

Parameters	Detail
-i install-inventory.ini	Installs package dependency to remote servers and localhost Mandatory parameter By default no remote servers are provided
-e YARDSTICK_DIR	Path to Yardstick folder Mandatory parameter for Yardstick bare metal installation
-e INSTALLATION_MODE	baremetal: Yardstick is installed to the bare metal Default parameter
	container: Yardstick is installed in container Container is built from Dockerfile
	container_pull: Yardstick is installed in container Container is pulled from docker hub
-e OS_RELEASE	xenial or bionic: Ubuntu version to be used for VM image (nsb or normal) Default is Ubuntu 16.04, xenial
-e IMAGE_PROPERTY	nsb: Build Yardstick NSB VM image Used to run Yardstick NSB tests on sample VNF Default parameter
	normal: Build VM image to run ping test in OpenStack
	none: don't build a VM image.
-e VIRTUAL_ENVIRONMENT	False or True: Whether install in virtualenv Default is False
-e YARD_IMAGE_ARCH	CPU architecture on servers Default is 'amd64'

2.4.7 Deploy InfluxDB and Grafana using Docker

Without InfluxDB, Yardstick stores results for running test case in the file `/tmp/yardstick.out`. However, it's inconvenient to retrieve and display test results. So we will show how to use InfluxDB to store data and use Grafana to display data in the following sections.

Automatic deployment of InfluxDB and Grafana containers (recommended)

1. Enter the Yardstick container:

```
sudo -EH docker exec -it yardstick /bin/bash
```

2. Create InfluxDB container and configure with the following command:

```
yardstick env influxdb
```

3. Create and configure Grafana container:

```
yardstick env grafana
```

Then you can run a test case and visit http://host_ip:1948 (`admin`/`admin`) to see the results.

Note: Executing `yardstick env` command to deploy InfluxDB and Grafana requires Jumphost's docker API version => 1.24. Run the following command to check the docker API version on the Jumphost:

```
docker version
```

Manual deployment of InfluxDB and Grafana containers

You can also deploy influxDB and Grafana containers manually on the Jumphost. The following sections show how to do.

Pull docker images:

```
sudo -EH docker pull tutum/influxdb
sudo -EH docker pull grafana/grafana
```

Run influxDB:

```
sudo -EH docker run -d --name influxdb \
   -p 8083:8083 -p 8086:8086 --expose 8090 --expose 8099 \
   tutum/influxdb
```

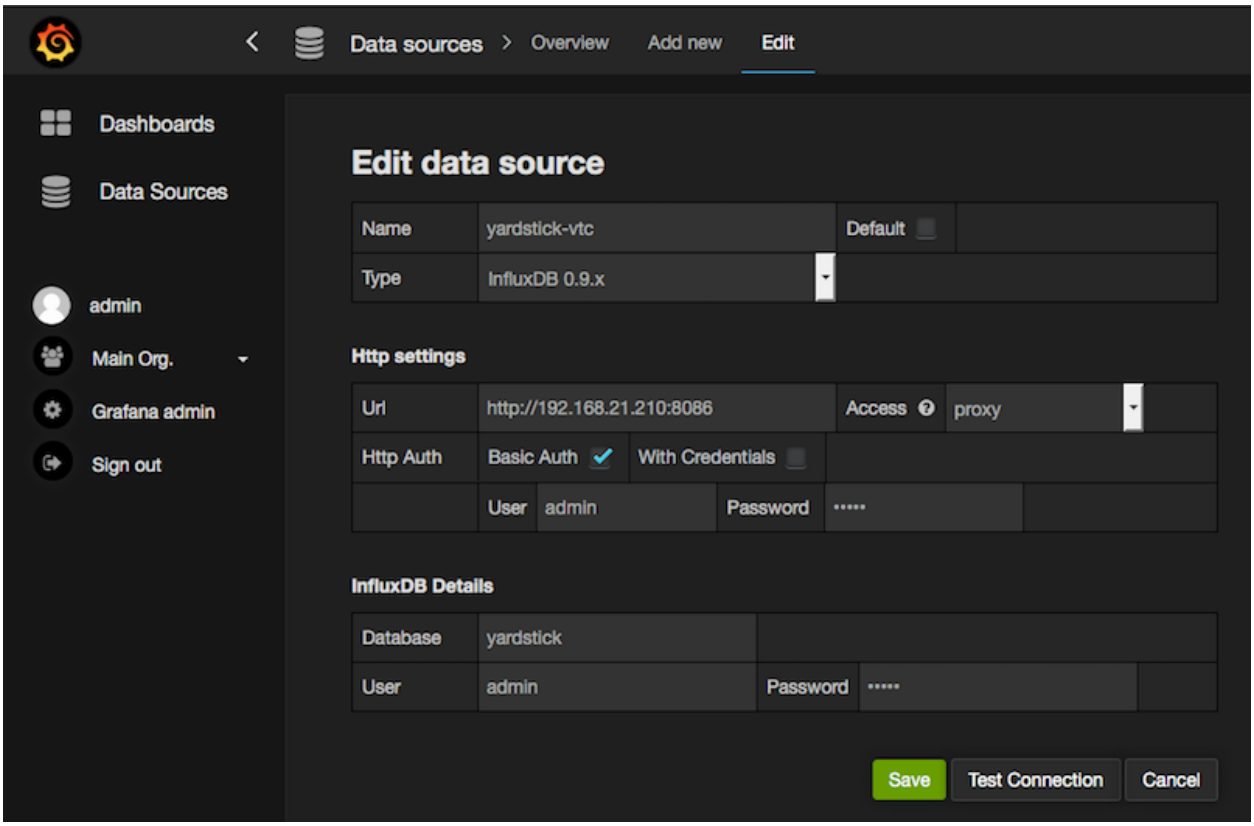
Configure influxDB:

```
docker exec -it influxdb influx
   > CREATE USER root WITH PASSWORD 'root' WITH ALL PRIVILEGES
   > CREATE DATABASE yardstick;
   > use yardstick;
   > show MEASUREMENTS;
   > exit
```

Run Grafana:

```
sudo -EH docker run -d --name grafana -p 1948:3000 grafana/grafana
```

Log on to `http://{YOUR_IP_HERE}:1948` using `admin/admin` and configure database resource to be `{YOUR_IP_HERE}:8086`.



Configure `yardstick.conf`:

```
sudo -EH docker exec -it yardstick /bin/bash
sudo cp etc/yardstick/yardstick.conf.sample /etc/yardstick/yardstick.conf
sudo vi /etc/yardstick/yardstick.conf
```

Modify `yardstick.conf` to add the `influxdb` dispatcher:

```
[DEFAULT]
debug = True
dispatcher = influxdb

[dispatcher_influxdb]
timeout = 5
target = http://{YOUR_IP_HERE}:8086
db_name = yardstick
username = root
password = root
```

Now Yardstick will store results in InfluxDB when you run a testcase.

2.4.8 Deploy InfluxDB and Grafana directly in Ubuntu (Todo)

2.4.9 Proxy Support

To configure the Jumphost to access Internet through a proxy its necessary to export several variables to the environment, contained in the following script:

```
#!/bin/sh
_proxy=<proxy_address>
_proxyport=<proxy_port>
_ip=$(hostname -I | awk '{print $1}')

export ftp_proxy=http://$_proxy:$_proxyport
export FTP_PROXY=http://$_proxy:$_proxyport
export http_proxy=http://$_proxy:$_proxyport
export HTTP_PROXY=http://$_proxy:$_proxyport
export https_proxy=http://$_proxy:$_proxyport
export HTTPS_PROXY=http://$_proxy:$_proxyport
export no_proxy=127.0.0.1,localhost,$_ip,$(hostname),<.localdomain>
export NO_PROXY=127.0.0.1,localhost,$_ip,$(hostname),<.localdomain>
```

To enable Internet access from a container using `docker`, depends on the OS version. On Ubuntu 14.04 LTS, which uses SysVinit, `/etc/default/docker` must be modified:

```
.......
# If you need Docker to use an HTTP proxy, it can also be specified here.
export http_proxy="http://<proxy_address>:<proxy_port>/"
export https_proxy="https://<proxy_address>:<proxy_port>/"
```

Then its necessary to restart the `docker` service:

```
sudo -EH service docker restart
```

In Ubuntu 16.04 LTS, which uses Systemd, its necessary to create a drop-in directory:

```
sudo mkdir /etc/systemd/system/docker.service.d
```

Then, the proxy configuration will be stored in the following file:

```
# cat /etc/systemd/system/docker.service.d/http-proxy.conf
[Service]
Environment="HTTP_PROXY=https://<proxy_address>:<proxy_port>/"
Environment="HTTPS_PROXY=https://<proxy_address>:<proxy_port>/"
Environment="NO_PROXY=localhost,127.0.0.1,<localaddress>,<.localdomain>"
```

The changes need to be flushed and the `docker` service restarted:

```
sudo systemctl daemon-reload
sudo systemctl restart docker
```

Any container is already created won't contain these modifications. If needed, stop and delete the container:

```
sudo docker stop yardstick
sudo docker rm yardstick
```

Warning: Be careful, the above `rm` command will delete the container completely. Everything on this container will be lost.

Then follow the previous instructions *Prepare the Yardstick container* to rebuild the Yardstick container.

2.4.10 References

2.5 Yardstick Usage

Once you have yardstick installed, you can start using it to run testcases immediately, through the CLI. You can also define and run new testcases and test suites. This chapter details basic usage (running testcases), as well as more advanced usage (creating your own testcases).

2.5.1 Yardstick common CLI

List test cases

`yardstick testcase list`: This command line would list all test cases in Yardstick. It would show like below:

```
+------------------------------------------------------------------------------------------
↪--
| Testcase Name         | Description
+------------------------------------------------------------------------------------------
↪--
| opnfv_yardstick_tc001 | Measure network throughput using pktgen
| opnfv_yardstick_tc002 | measure network latency using ping
| opnfv_yardstick_tc005 | Measure Storage IOPS, throughput and latency using fio.
...
+------------------------------------------------------------------------------------------
↪--
```

Show a test case config file

Take opnfv_yardstick_tc002 for an example. This test case measure network latency. You just need to type in `yardstick testcase show opnfv_yardstick_tc002`, and the console would show the config yaml of this test case:

```
---

schema: "yardstick:task:0.1"
description: >
    Yardstick TC002 config file;
    measure network latency using ping;

{% set image = image or "cirros-0.3.5" %}

{% set provider = provider or none %}
{% set physical_network = physical_network or 'physnet1' %}
{% set segmentation_id = segmentation_id or none %}
{% set packetsize = packetsize or 100 %}
```

(continues on next page)

(continued from previous page)

```
scenarios:
{% for i in range(2) %}
-
  type: Ping
  options:
    packetsize: {{packetsize}}
  host: athena.demo
  target: ares.demo

  runner:
    type: Duration
    duration: 60
    interval: 10

  sla:
    max_rtt: 10
    action: monitor
{% endfor %}

context:
  name: demo
  image: {{image}}
  flavor: yardstick-flavor
  user: cirros

  placement_groups:
    pgrp1:
      policy: "availability"

  servers:
    athena:
      floating_ip: true
      placement: "pgrp1"
    ares:
      placement: "pgrp1"

  networks:
    test:
      cidr: '10.0.1.0/24'
      {% if provider == "vlan" or provider == "sriov" %}
      provider: {{provider}}
      physical_network: {{physical_network}}
        {% if segmentation_id %}
      segmentation_id: {{segmentation_id}}
        {% endif %}
      {% endif %}
```

Run a Yardstick test case

If you want run a test case, then you need to use `yardstick task start <test_case_path>` this command support some parameters as below:

Parameters	Detail
-d	show debug log of yardstick running
–task-args	If you want to customize test case parameters, use "–task-args" to pass the value. The format is a json string with parameter key-value pair.
–task-args-file	If you want to use yardstick env prepare command(or related API) to load the
–parse-only	
–output-file OUTPUT_FILE_PATH	Specify where to output the log. if not pass, the default value is "/tmp/yardstick/yardstick.log"
–suite TEST_SUITE_PATH	run a test suite, TEST_SUITE_PATH specify where the test suite locates

2.5.2 Run Yardstick in a local environment

We also have a guide about How to run Yardstick in a local environment. This work is contributed by Tapio Tallgren.

2.5.3 Create a new testcase for Yardstick

As a user, you may want to define a new testcase in addition to the ones already available in Yardstick. This section will show you how to do this.

Each testcase consists of two sections:

- `scenarios` describes what will be done by the test
- `context` describes the environment in which the test will be run.

Defining the testcase scenarios

TODO

Defining the testcase context(s)

Each testcase consists of one or more contexts, which describe the environment in which the testcase will be run. Current available contexts are:

- `Dummy`: this is a no-op context, and is used when there is no environment to set up e.g. when testing whether OpenStack services are available
- `Node`: this context is used to perform operations on baremetal servers
- `Heat`: uses OpenStack to provision the required hosts, networks, etc.
- `Kubernetes`: uses Kubernetes to provision the resources required for the test.

Regardless of the context type, the `context` section of the testcase will consist of the following:

```
context:
  name: demo
  type: Dummy|Node|Heat|Kubernetes
```

The content of the `context` section will vary based on the context type.

Dummy Context

No additional information is required for the Dummy context:

```
context:
  name: my_context
  type: Dummy
```

Node Context

TODO

Heat Context

In addition to `name` and `type`, a Heat context requires the following arguments:

- `image`: the image to be used to boot VMs
- `flavor`: the flavor to be used for VMs in the context
- `user`: the username for connecting into the VMs
- `networks`: The networks to be created, networks are identified by name
  - `name`: network name (required)
  - (TODO) Any optional attributes
- `servers`: The servers to be created
  - `name`: server name
  - (TODO) Any optional attributes

In addition to the required arguments, the following optional arguments can be passed to the Heat context:

- `placement_groups`:
  - `name`: the name of the placement group to be created
  - `policy`: either `affinity` or `availability`
- `server_groups`:
  - `name`: the name of the server group
  - `policy`: either `affinity` or `anti-affinity`

Combining these elements together, a sample Heat context config looks like:

```
# Sample Heat context config with Dummy context

schema: "yardstick:task:0.1"

scenarios:
-
  type: Dummy

  runner:
    type: Duration
```

(continues on next page)

(continued from previous page)

```
    duration: 5
    interval: 1

context:
  name: {{ context_name }}
  image: yardstick-image
  flavor: yardstick-flavor
  user: ubuntu

  servers:
    athena:
      name: athena
    ares:
      name: ares

  networks:
    test:
      name: test
```

Using exisiting HOT Templates

TODO

Kubernetes Context

TODO

Using multiple contexts in a testcase

When using multiple contexts in a testcase, the `context` section is replaced by a `contexts` section, and each context is separated with a − line:

```
contexts:
-
  name: context1
  type: Heat
  ...
-
  name: context2
  type: Node
  ...
```

Reusing a context

Typically, a context is torn down after a testcase is run, however, the user may wish to keep an context intact after a testcase is complete.

Note: This feature has been implemented for the Heat context only

To keep or reuse a context, the `flags` option must be specified:

- **`no_setup`: skip the deploy stage, and fetch the details of a deployed** context/Heat stack.
- **`no_teardown`: skip the undeploy stage, thus keeping the stack intact for** the next test

If either of these `flags` are `True`, the context information must still be given. By default, these flags are disabled:

```
context:
  name: mycontext
  type: Heat
  flags:
    no_setup: True
    no_teardown: True
  ...
```

2.5.4 Create a test suite for Yardstick

A test suite in Yardstick is a .yaml file which includes one or more test cases. Yardstick is able to support running test suite task, so you can customize your own test suite and run it in one task.

`tests/opnfv/test_suites` is the folder where Yardstick puts CI test suite. A typical test suite is like below (the `fuel_test_suite.yaml` example):

```
---
# Fuel integration test task suite

schema: "yardstick:suite:0.1"

name: "fuel_test_suite"
test_cases_dir: "samples/"
test_cases:
-
  file_name: ping.yaml
-
  file_name: iperf3.yaml
```

As you can see, there are two test cases in the `fuel_test_suite.yaml`. The `schema` and the `name` must be specified. The test cases should be listed via the tag `test_cases` and their relative path is also marked via the tag `test_cases_dir`.

Yardstick test suite also supports constraints and task args for each test case. Here is another sample (the `os-nosdn-nofeature-ha.yaml` example) to show this, which is digested from one big test suite:

```
---

schema: "yardstick:suite:0.1"

name: "os-nosdn-nofeature-ha"
test_cases_dir: "tests/opnfv/test_cases/"
test_cases:
-
  file_name: opnfv_yardstick_tc002.yaml
-
  file_name: opnfv_yardstick_tc005.yaml
-
  file_name: opnfv_yardstick_tc043.yaml
     constraint:
```

(continues on next page)

(continued from previous page)

```
        installer: compass
        pod: huawei-pod1
    task_args:
        huawei-pod1: '{"pod_info": "etc/yardstick/.../pod.yaml",
        "host": "node4.LF","target": "node5.LF"}'
```

As you can see in test case `opnfv_yardstick_tc043.yaml`, there are two tags, `constraint` and `task_args`. `constraint` is to specify which installer or pod it can be run in the CI environment. `task_args` is to specify the task arguments for each pod.

All in all, to create a test suite in Yardstick, you just need to create a yaml file and add test cases, constraint or task arguments if necessary.

2.5.5 References

2.6 Installing a plug-in into Yardstick

2.6.1 Abstract

Yardstick provides a `plugin` CLI command to support integration with other OPNFV testing projects. Below is an example invocation of Yardstick plugin command and Storperf plug-in sample.

2.6.2 Installing Storperf into Yardstick

Storperf is delivered as a Docker container from https://hub.docker.com/r/opnfv/storperf/tags/.

There are two possible methods for installation in your environment:

- Run container on Jump Host
- Run container in a VM

In this introduction we will install Storperf on Jump Host.

Step 0: Environment preparation

Running Storperf on Jump Host Requirements:

- Docker must be installed
- Jump Host must have access to the OpenStack Controller API
- Jump Host must have internet connectivity for downloading docker image
- Enough floating IPs must be available to match your agent count

Before installing Storperf into yardstick you need to check your openstack environment and other dependencies:

1. Make sure docker is installed.
2. Make sure Keystone, Nova, Neutron, Glance, Heat are installed correctly.
3. Make sure Jump Host have access to the OpenStack Controller API.
4. Make sure Jump Host must have internet connectivity for downloading docker image.

5. You need to know where to get basic openstack Keystone authorization info, such as OS_PASSWORD, OS_PROJECT_NAME, OS_AUTH_URL, OS_USERNAME.
6. To run a Storperf container, you need to have OpenStack Controller environment variables defined and passed to Storperf container. The best way to do this is to put environment variables in a "storperf_admin-rc" file. The storperf_admin-rc should include credential environment variables at least:
   - OS_AUTH_URL
   - OS_USERNAME
   - OS_PASSWORD
   - OS_PROJECT_NAME
   - OS_PROJECT_ID
   - OS_USER_DOMAIN_ID

Yardstick has a `prepare_storperf_admin-rc.sh` script which can be used to generate the `storperf_admin-rc` file, this script is located at `test/ci/prepare_storperf_admin-rc.sh`

```
#!/bin/bash
# Prepare storperf_admin-rc for StorPerf.
AUTH_URL=${OS_AUTH_URL}
USERNAME=${OS_USERNAME:-admin}
PASSWORD=${OS_PASSWORD:-console}

# OS_TENANT_NAME is still present to keep backward compatibility with legacy
# deployments, but should be replaced by OS_PROJECT_NAME.
TENANT_NAME=${OS_TENANT_NAME:-admin}
PROJECT_NAME=${OS_PROJECT_NAME:-$TENANT_NAME}
PROJECT_ID=`openstack project show admin|grep '\bid\b' |awk -F '|' '{print $3}'|sed -
↪e 's/^[[:space:]]*//'`
USER_DOMAIN_ID=${OS_USER_DOMAIN_ID:-default}

rm -f ~/storperf_admin-rc
touch ~/storperf_admin-rc

echo "OS_AUTH_URL="$AUTH_URL >> ~/storperf_admin-rc
echo "OS_USERNAME="$USERNAME >> ~/storperf_admin-rc
echo "OS_PASSWORD="$PASSWORD >> ~/storperf_admin-rc
echo "OS_PROJECT_NAME="$PROJECT_NAME >> ~/storperf_admin-rc
echo "OS_PROJECT_ID="$PROJECT_ID >> ~/storperf_admin-rc
echo "OS_USER_DOMAIN_ID="$USER_DOMAIN_ID >> ~/storperf_admin-rc
```

The generated `storperf_admin-rc` file will be stored in the root directory. If you installed *Yardstick* using Docker, this file will be located in the container. You may need to copy it to the root directory of the Storperf deployed host.

Step 1: Plug-in configuration file preparation

To install a plug-in, first you need to prepare a plug-in configuration file in YAML format and store it in the "plugin" directory. The plugin configration file work as the input of yardstick "plugin" command. Below is the Storperf plug-in configuration file sample:

```
---
# StorPerf plugin configuration file
# Used for integration StorPerf into Yardstick as a plugin
schema: "yardstick:plugin:0.1"
```

(continues on next page)

(continued from previous page)

```
plugins:
  name: storperf
deployment:
  ip: 192.168.23.2
  user: root
  password: root
```

In the plug-in configuration file, you need to specify the plug-in name and the plug-in deployment info, including node ip, node login username and password. Here the Storperf will be installed on IP 192.168.23.2 which is the Jump Host in my local environment.

Step 2: Plug-in install/remove scripts preparation

In `yardstick/resource/scripts` directory, there are two folders: an `install` folder and a `remove` folder. You need to store the plug-in install/remove scripts in these two folders respectively.

The detailed installation or remove operation should de defined in these two scripts. The name of both install and remove scripts should match the plugin-in name that you specified in the plug-in configuration file.

For example, the install and remove scripts for Storperf are both named `storperf.bash`.

Step 3: Install and remove Storperf

To install Storperf, simply execute the following command:

```
# Install Storperf
yardstick plugin install plugin/storperf.yaml
```

Removing Storperf from Yardstick

To remove Storperf, simply execute the following command:

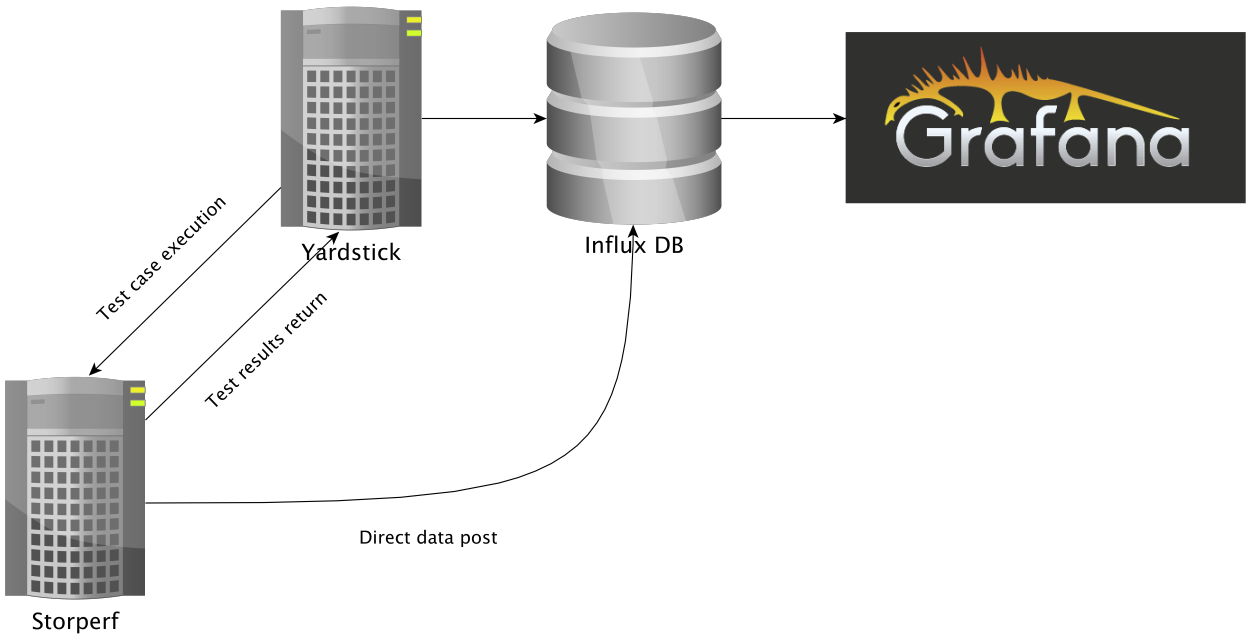
```
# Remove Storperf
yardstick plugin remove plugin/storperf.yaml
```

What yardstick plugin command does is using the username and password to log into the deployment target and then execute the corresponding install or remove script.

2.7 Store Other Project's Test Results in InfluxDB

2.7.1 Abstract

This chapter illustrates how to run plug-in test cases and store test results into community's InfluxDB. The framework is shown in Framework.



2.7.2 Store Storperf Test Results into Community's InfluxDB

As shown in Framework, there are two ways to store Storperf test results into community's InfluxDB:

1. Yardstick executes Storperf test case (TC074), posting test job to Storperf container via ReST API. After the test job is completed, Yardstick reads test results via ReST API from Storperf and posts test data to the influxDB.
2. Additionally, Storperf can run tests by itself and post the test result directly to the InfluxDB. The method for posting data directly to influxDB will be supported in the future.

Our plan is to support rest-api in D release so that other testing projects can call the rest-api to use yardstick dispatcher service to push data to Yardstick's InfluxDB database.

For now, InfluxDB only supports line protocol, and the json protocol is deprecated.

Take ping test case for example, the `raw_result` is json format like this:

```
  "benchmark": {
      "timestamp": 1470315409.868095,
      "errors": "",
      "data": {
        "rtt": {
        "ares": 1.125
        }
      },
    "sequence": 1
    },
  "runner_id": 2625
}
```

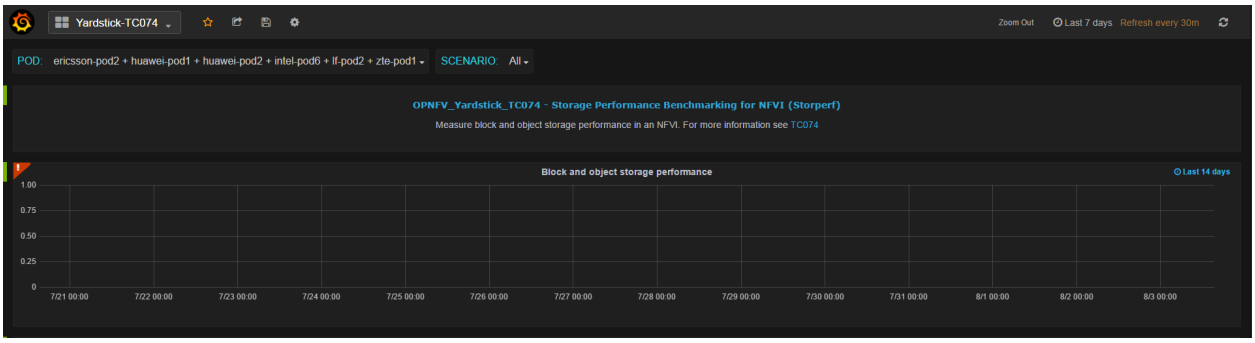
With the help of "influxdb_line_protocol", the json is transform to like below as a line string:

```
'ping,deploy_scenario=unknown,host=athena.demo,installer=unknown,pod_name=unknown,
  runner_id=2625,scenarios=Ping,target=ares.demo,task_id=77755f38-1f6a-4667-a7f3-
    301c99963656,version=unknown rtt.ares=1.125 1470315409868094976'
```

So, for data output of json format, you just need to transform json into line format and call influxdb api to post the data into the database. All this function has been implemented in Influxdb. If you need support on this, please contact Mingjiang.

```
curl -i -XPOST 'http://104.197.68.199:8086/write?db=yardstick' --
  data-binary 'ping,deploy_scenario=unknown,host=athena.demo,installer=unknown, ...'
```

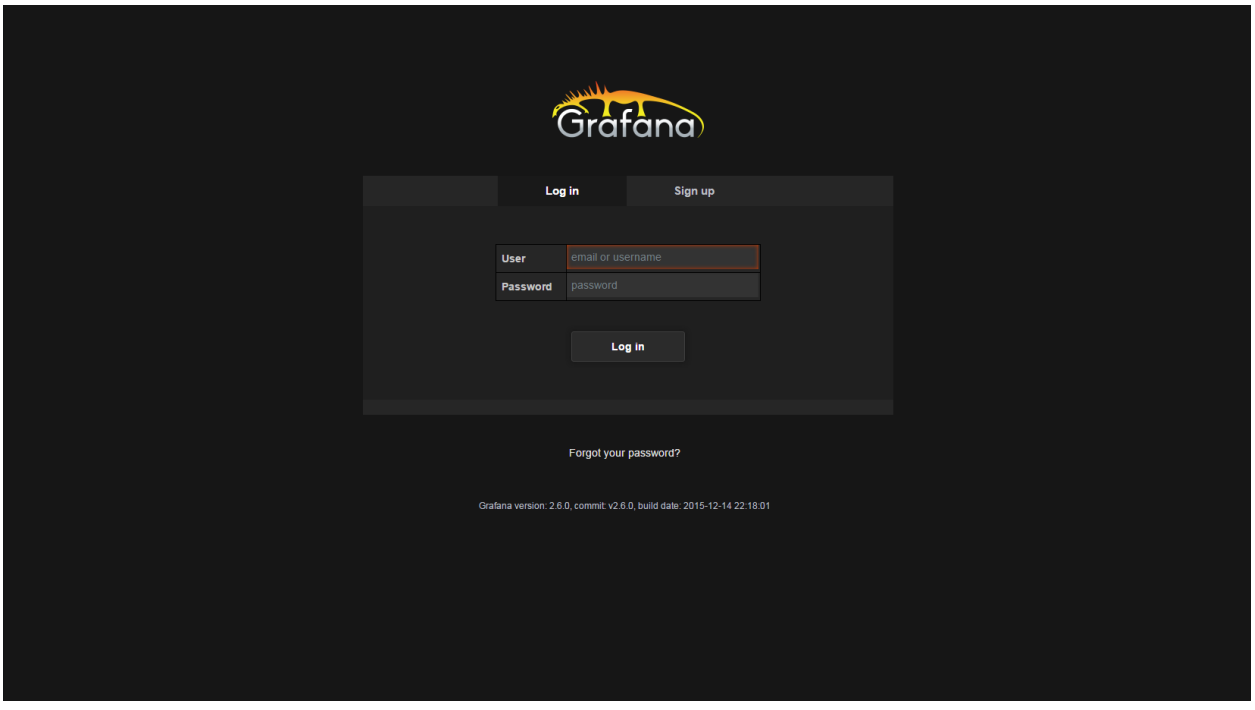
Grafana will be used for visualizing the collected test data, which is shown in Visual. Grafana can be accessed by Login.



2.8 Grafana dashboard

2.8.1 Abstract

This chapter describes the Yardstick grafana dashboard. The Yardstick grafana dashboard can be found here: http://testresults.opnfv.org/grafana/

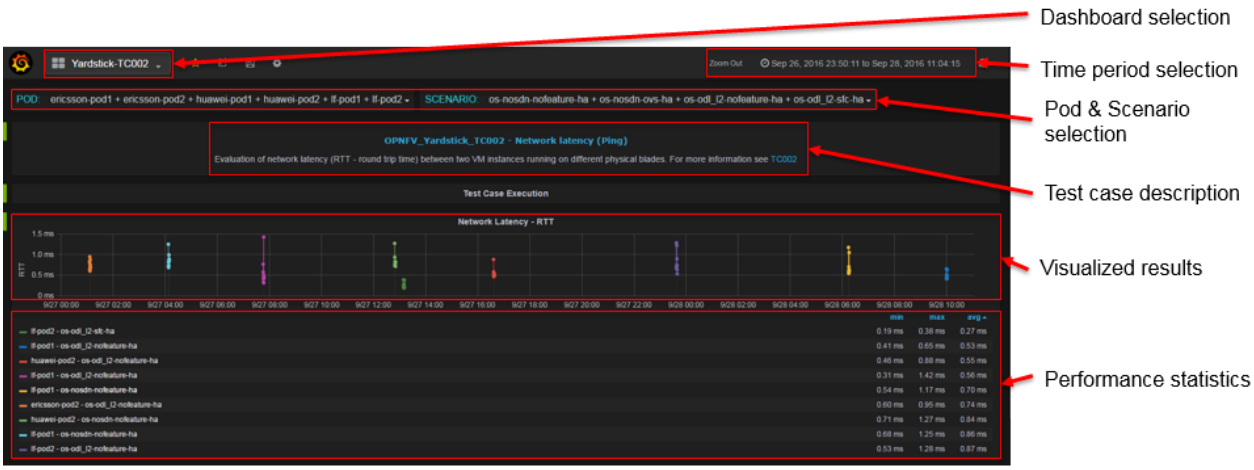


2.8.2 Public access

Yardstick provids a public account for accessing to the dashboard. The username and password are both set to 'opnfv'.

2.8.3 Testcase dashboard

For each test case, there is a dedicated dashboard. Shown here is the dashboard of TC002.



For each test case dashboard. On the top left, we have a dashboard selection, you can switch to different test cases using this pull-down menu.

Underneath, we have a pod and scenario selection. All the pods and scenarios that have ever published test data to the InfluxDB will be shown here.

You can check multiple pods or scenarios.

For each test case, we have a short description and a link to detailed test case information in Yardstick user guide.

Underneath, it is the result presentation section. You can use the time period selection on the top right corner to zoom in or zoom out the chart.

2.8.4 Administration access

For a user with administration rights it is easy to update and save any dashboard configuration. Saved updates immediately take effect and become live. This may cause issues like:

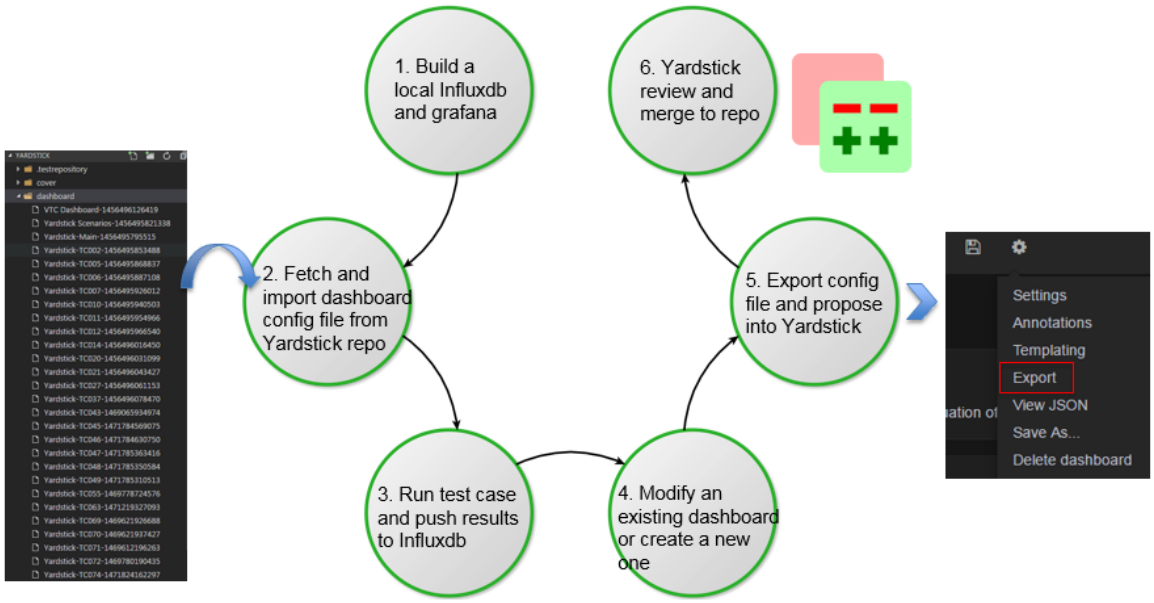
- Changes and updates made to the live configuration in Grafana can compromise existing Grafana content in an unwanted, unpredicted or incompatible way. Grafana as such is not version controlled, there exists one single Grafana configuration per dashboard.
- There is a risk several people can disturb each other when doing updates to the same Grafana dashboard at the same time.

Any change made by administrator should be careful.

2.8.5 Add a dashboard into Yardstick Grafana

Due to security concern, users that using the public opnfv account are not able to edit the yardstick grafana directly. It takes a few more steps for a non-yardstick user to add a custom dashboard into yardstick grafana.

There are 6 steps to go.



1. You need to build a local influxdb and grafana, so you can do the work locally. You can refer to How to deploy InfluxDB and Grafana locally wiki page about how to do this.
2. Once step one is done, you can fetch the existing grafana dashboard configuration file from the yardstick repository and import it to your local grafana. After import is done, you grafana dashboard will be ready to use just like the community's dashboard.
3. The third step is running some test cases to generate test results and publishing it to your local influxdb.
4. Now you have some data to visualize in your dashboard. In the fourth step, it is time to create your own dashboard. You can either modify an existing dashboard or try to create a new one from scratch. If you choose to modify an existing dashboard then in the curtain menu of the existing dashboard do a "Save As…" into a new dashboard copy instance, and then continue doing all updates and saves within the dashboard copy.
5. When finished with all Grafana configuration changes in this temporary dashboard then chose "export" of the updated dashboard copy into a JSON file and put it up for review in Gerrit, in file `/yardstick/dashboard/Yardstick-TCxxx-yyyyyyyyyyyyy`. For instance a typical default name of the file would be `Yardstick-TC001 Copy-1234567891234`.
6. Once you finish your dashboard, the next step is exporting the configuration file and propose a patch into Yardstick. Yardstick team will review and merge it into Yardstick repository. After approved review Yardstick team will do an "import" of the JSON file and also a "save dashboard" as soon as possible to replace the old live dashboard configuration.

2.9 Yardstick Restful API

2.9.1 Abstract

Yardstick support restful API since Danube.

2.9.2 Available API

/yardstick/env/action

Description: This API is used to prepare Yardstick test environment. For Euphrates, it supports:

1. Prepare yardstick test environment, including setting the `EXTERNAL_NETWORK` environment variable, load Yardstick VM images and create flavors;
2. Start an InfluxDB Docker container and config Yardstick output to InfluxDB;
3. Start a Grafana Docker container and config it with the InfluxDB.

Which API to call will depend on the parameters.

Method: POST

Prepare Yardstick test environment Example:

```
{
    'action': 'prepare_env'
}
```

This is an asynchronous API. You need to call `/yardstick/asynctask` API to get the task result.

Start and config an InfluxDB docker container Example:

```
{
    'action': 'create_influxdb'
}
```

This is an asynchronous API. You need to call `/yardstick/asynctask` API to get the task result.

Start and config a Grafana docker container Example:

```
{
    'action': 'create_grafana'
}
```

This is an asynchronous API. You need to call `/yardstick/asynctask` API to get the task result.

/yardstick/asynctask

Description: This API is used to get the status of asynchronous tasks

Method: GET

Get the status of asynchronous tasks Example:

```
http://<SERVER IP>:<PORT>/yardstick/asynctask?task_id=3f3f5e03-972a-4847-a5f8-
↪154f1b31db8c
```

The returned status will be 0(running), 1(finished) and 2(failed).

NOTE:

```
<SERVER IP>: The ip of the host where you start your yardstick container
<PORT>: The outside port of port mapping which set when you start start yardstick␣
↪container
```

/yardstick/testcases

Description: This API is used to list all released Yardstick test cases.

Method: GET

Get a list of released test cases Example:

```
http://<SERVER IP>:<PORT>/yardstick/testcases
```

/yardstick/testcases/release/action

Description: This API is used to run a Yardstick released test case.

Method: POST

Run a released test case Example:

```
{
    'action': 'run_test_case',
    'args': {
        'opts': {},
        'testcase': 'opnfv_yardstick_tc002'
    }
}
```

This is an asynchronous API. You need to call `/yardstick/results` to get the result.

/yardstick/testcases/samples/action

Description: This API is used to run a Yardstick sample test case.

Method: POST

Run a sample test case Example:

```
{
    'action': 'run_test_case',
    'args': {
        'opts': {},
        'testcase': 'ping'
    }
}
```

This is an asynchronous API. You need to call `/yardstick/results` to get the result.

/yardstick/testcases/<testcase_name>/docs

Description: This API is used to the documentation of a certain released test case.

Method: GET

Get the documentation of a certain test case Example:

```
http://<SERVER IP>:<PORT>/yardstick/taskcases/opnfv_yardstick_tc002/docs
```

/yardstick/testsuites/action

Description: This API is used to run a Yardstick test suite.

Method: POST

Run a test suite Example:

```
{
    'action': 'run_test_suite',
    'args': {
        'opts': {},
        'testsuite': 'opnfv_smoke'
    }
}
```

This is an asynchronous API. You need to call /yardstick/results to get the result.

/yardstick/tasks/<task_id>/log

Description: This API is used to get the real time log of test case execution.

Method: GET

Get real time of test case execution Example:

```
http://<SERVER IP>:<PORT>/yardstick/tasks/14795be8-f144-4f54-81ce-43f4e3eab33f/log?
↪index=0
```

/yardstick/results

Description: This API is used to get the test results of tasks. If you call /yardstick/testcases/samples/action API, it will return a task id. You can use the returned task id to get the results by using this API.

Method: GET

Get test results of one task Example:

```
http://<SERVER IP>:<PORT>/yardstick/results?task_id=3f3f5e03-972a-4847-a5f8-
↪154f1b31db8c
```

This API will return a list of test case result

/api/v2/yardstick/openrcs

Description: This API provides functionality of handling OpenStack credential file (openrc). For Euphrates, it supports:

1. Upload an openrc file for an OpenStack environment;
2. Update an openrc;
3. Get openrc file information;
4. Delete an openrc file.

Which API to call will depend on the parameters.

METHOD: POST

Upload an openrc file for an OpenStack environment Example:

```
{
    'action': 'upload_openrc',
    'args': {
        'file': file,
        'environment_id': environment_id
    }
}
```

METHOD: POST

Update an openrc file Example:

```
{
    'action': 'update_openrc',
    'args': {
        'openrc': {
            "EXTERNAL_NETWORK": "ext-net",
            "OS_AUTH_URL": "http://192.168.23.51:5000/v3",
            "OS_IDENTITY_API_VERSION": "3",
            "OS_IMAGE_API_VERSION": "2",
            "OS_PASSWORD": "console",
            "OS_PROJECT_DOMAIN_NAME": "default",
            "OS_PROJECT_NAME": "admin",
            "OS_USERNAME": "admin",
            "OS_USER_DOMAIN_NAME": "default"
        },
        'environment_id': environment_id
    }
}
```

/api/v2/yardstick/openrcs/<openrc_id>

Description: This API provides functionality of handling OpenStack credential file (openrc). For Euphrates, it supports:

1. Get openrc file information;
2. Delete an openrc file.

METHOD: GET

Get openrc file information Example:

```
http://<SERVER IP>:<PORT>/api/v2/yardstick/openrcs/5g6g3e02-155a-4847-a5f8-
↪154f1b31db8c
```

METHOD: DELETE

Delete openrc file Example:

```
http://<SERVER IP>:<PORT>/api/v2/yardstick/openrcs/5g6g3e02-155a-4847-a5f8-
↪154f1b31db8c
```

/api/v2/yardstick/pods

Description: This API provides functionality of handling Yardstick pod file (pod.yaml). For Euphrates, it supports:

1. Upload a pod file;

Which API to call will depend on the parameters.

METHOD: POST

Upload a pod.yaml file Example:

```
{
    'action': 'upload_pod_file',
    'args': {
        'file': file,
        'environment_id': environment_id
    }
}
```

/api/v2/yardstick/pods/<pod_id>

Description: This API provides functionality of handling Yardstick pod file (pod.yaml). For Euphrates, it supports:

1. Get pod file information;
2. Delete an openrc file.

METHOD: GET

Get pod file information Example:

```
http://<SERVER IP>:<PORT>/api/v2/yardstick/pods/5g6g3e02-155a-4847-a5f8-154f1b31db8c
```

METHOD: DELETE

Delete openrc file Example:

```
http://<SERVER IP>:<PORT>/api/v2/yardstick/pods/5g6g3e02-155a-4847-a5f8-154f1b31db8c
```

/api/v2/yardstick/images

Description: This API is used to do some work related to Yardstick VM images. For Euphrates, it supports:

1. Load Yardstick VM images;

Which API to call will depend on the parameters.

METHOD: POST

Load VM images Example:

```
{
    'action': 'load_image',
    'args': {
        'name': 'yardstick-image'
    }
}
```

/api/v2/yardstick/images/<image_id>

Description: This API is used to do some work related to Yardstick VM images. For Euphrates, it supports:

1. Get image's information;
2. Delete images

METHOD: GET

Get image information Example:

```
http://<SERVER IP>:<PORT>/api/v2/yardstick/images/5g6g3e02-155a-4847-a5f8-154f1b31db8c
```

METHOD: DELETE

Delete images Example:

```
http://<SERVER IP>:<PORT>/api/v2/yardstick/images/5g6g3e02-155a-4847-a5f8-154f1b31db8c
```

/api/v2/yardstick/tasks

Description: This API is used to do some work related to yardstick tasks. For Euphrates, it supports:

1. Create a Yardstick task;

Which API to call will depend on the parameters.

METHOD: POST

Create a Yardstick task Example:

```
{
    'action': 'create_task',
        'args': {
            'name': 'task1',
            'project_id': project_id
        }
}
```

/api/v2/yardstick/tasks/<task_id>

Description: This API is used to do some work related to yardstick tasks. For Euphrates, it supports:

1. Add a environment to a task
2. Add a test case to a task;
3. Add a test suite to a task;
4. run a Yardstick task;
5. Get a tasks' information;
6. Delete a task.

METHOD: PUT

Add a environment to a task

Example:

```
{
    'action': 'add_environment',
    'args': {
        'environment_id': 'e3cadbbb-0419-4fed-96f1-a232daa0422a'
    }
}
```

METHOD: PUT

Add a test case to a task Example:

```
{
    'action': 'add_case',
    'args': {
        'case_name': 'opnfv_yardstick_tc002',
        'case_content': case_content
    }
}
```

METHOD: PUT

Add a test suite to a task Example:

```
{
    'action': 'add_suite',
    'args': {
        'suite_name': 'opnfv_smoke',
        'suite_content': suite_content
    }
}
```

METHOD: PUT

Run a task

Example:

```
{
    'action': 'run'
}
```

METHOD: GET

Get a task's information Example:

```
http://<SERVER IP>:<PORT>/api/v2/yardstick/tasks/5g6g3e02-155a-4847-a5f8-154f1b31db8c
```

METHOD: DELETE

Delete a task

Example:

```
http://<SERVER IP>:<PORT>/api/v2/yardstick/tasks/5g6g3e02-155a-4847-a5f8-154f1b31db8c
```

/api/v2/yardstick/testcases

Description: This API is used to do some work related to Yardstick testcases. For Euphrates, it supports:

1. Upload a test case;
2. Get all released test cases' information;

Which API to call will depend on the parameters.

METHOD: POST

Upload a test case Example:

```
{
    'action': 'upload_case',
    'args': {
        'file': file
    }
}
```

METHOD: GET

Get all released test cases' information Example:

```
http://<SERVER IP>:<PORT>/api/v2/yardstick/testcases
```

/api/v2/yardstick/testcases/<case_name>

Description: This API is used to do some work related to yardstick testcases. For Euphrates, it supports:

1. Get certain released test case's information;
2. Delete a test case.

METHOD: GET

Get certain released test case's information Example:

```
http://<SERVER IP>:<PORT>/api/v2/yardstick/testcases/opnfv_yardstick_tc002
```

METHOD: DELETE

Delete a certain test case Example:

```
http://<SERVER IP>:<PORT>/api/v2/yardstick/testcases/opnfv_yardstick_tc002
```

/api/v2/yardstick/testsuites

Description: This API is used to do some work related to yardstick test suites. For Euphrates, it supports:

1. Create a test suite;
2. Get all test suites;

Which API to call will depend on the parameters.

METHOD: POST

Create a test suite Example:

```
{
    'action': 'create_suite',
    'args': {
        'name': <suite_name>,
        'testcases': [
            'opnfv_yardstick_tc002'
        ]
    }
}
```

METHOD: GET

Get all test suite Example:

```
http://<SERVER IP>:<PORT>/api/v2/yardstick/testsuites
```

/api/v2/yardstick/testsuites

Description: This API is used to do some work related to yardstick test suites. For Euphrates, it supports:

1. Get certain test suite's information;
2. Delete a test case.

METHOD: GET

Get certain test suite's information Example:

```
http://<SERVER IP>:<PORT>/api/v2/yardstick/testsuites/<suite_name>
```

METHOD: DELETE

Delete a certain test suite Example:

```
http://<SERVER IP>:<PORT>/api/v2/yardstick/testsuites/<suite_name>
```

/api/v2/yardstick/projects

Description: This API is used to do some work related to Yardstick test projects. For Euphrates, it supports:

1. Create a Yardstick project;
2. Get all projects;

Which API to call will depend on the parameters.

METHOD: POST

Create a Yardstick project Example:

```
{
    'action': 'create_project',
    'args': {
        'name': 'project1'
    }
}
```

METHOD: GET

Get all projects' information Example:

```
http://<SERVER IP>:<PORT>/api/v2/yardstick/projects
```

/api/v2/yardstick/projects

Description: This API is used to do some work related to yardstick test projects. For Euphrates, it supports:

1. Get certain project's information;
2. Delete a project.

METHOD: GET

Get certain project's information Example:

```
http://<SERVER IP>:<PORT>/api/v2/yardstick/projects/<project_id>
```

METHOD: DELETE

Delete a certain project Example:

```
http://<SERVER IP>:<PORT>/api/v2/yardstick/projects/<project_id>
```

/api/v2/yardstick/containers

Description: This API is used to do some work related to Docker containers. For Euphrates, it supports:

1. Create a Grafana Docker container;
2. Create an InfluxDB Docker container;

Which API to call will depend on the parameters.

METHOD: POST

Create a Grafana Docker container Example:

```
{
    'action': 'create_grafana',
    'args': {
        'environment_id': <environment_id>
    }
}
```

METHOD: POST

Create an InfluxDB Docker container Example:

```
{
    'action': 'create_influxdb',
    'args': {
        'environment_id': <environment_id>
    }
}
```

/api/v2/yardstick/containers/<container_id>

Description: This API is used to do some work related to Docker containers. For Euphrates, it supports:

1. Get certain container's information;
2. Delete a container.

METHOD: GET

Get certain container's information Example:

```
http://<SERVER IP>:<PORT>/api/v2/yardstick/containers/<container_id>
```

METHOD: DELETE

Delete a certain container Example:

```
http://<SERVER IP>:<PORT>/api/v2/yardstick/containers/<container_id>
```

2.10 Yardstick User Interface

This chapter describes how to generate HTML reports, used to view, store, share or publish test results in table and graph formats.

The following layouts are available:

- The compact HTML report layout is suitable for testcases producing a few metrics over a short period of time. All metrics for all timestamps are displayed in the data table and on the graph.
- The dynamic HTML report layout consists of a wider data table, a graph, and a tree that allows selecting the metrics to be displayed. This layout is suitable for testcases, such as NSB ones, producing a lot of metrics over a longer period of time.

2.10.1 Commands

To generate the compact HTML report, run:

```
yardstick report generate <task-ID> <testcase-filename>
```

To generate the dynamic HTML report, run:

```
yardstick report generate-nsb <task-ID> <testcase-filename>
```

2.10.2 Description

1. When the command is triggered, the relevant values for the provided task-id and testcase name are retrieved from the database (InfluxDB in this particular case).
2. The values are then formatted and provided to the html template to be rendered using Jinja2.
3. Then the rendered template is written into a html file.

The graph is framed with Timestamp on x-axis and output values (differ from testcase to testcase) on y-axis with the help of Chart.js.

2.11 Network Services Benchmarking (NSB)

2.11.1 Abstract

This chapter provides an overview of the NSB, a contribution to OPNFV Yardstick from Intel.

2.11.2 Overview

Network Services Benchmarking (*NSB*) uses the Yardstick framework for performing *VNF* and *NFVI* characterisation in an *NFV* environment.

For VNF characterisation, NSB will onboard a VNF, source and sink traffic to it via traffic generators, and collect a variety of key performance indicators (*KPI*) during VNF execution. The stream of KPI data is stored in a database, and it is visualized in a performance-visualization dashboard.

For NFVI characterisation, a fixed test VNF, called *PROX* is used. PROX implements a suite of test cases and visualizes the output data of the test suite. The PROX test cases implement various execution kernels found in real-world VNFs, and the output of the test cases provides an indication of the fitness of the infrastructure for running NFV services, in addition to indicating potential performance optimizations for the NFVI.

NSB extends the Yardstick framework to do VNF characterization in three different execution environments - bare metal i.e. native Linux environment, standalone virtual environment and managed virtualized environment (e.g. OpenStack). It also brings in the capability to interact with external traffic generators, both hardware and software based, for triggering and validating the traffic according to user defined profiles.

NSB extension includes:

- Generic data models of Network Services, based on ETSI spec ETSI GS NFV-TST001
- Standalone *context* for VNF testing SRIOV, OVS-DPDK, etc
- Generic VNF configuration models and metrics implemented with Python classes
- Traffic generator features and traffic profiles
  - L1-L3 stateless traffic profiles
  - L4-L7 state-full traffic profiles
  - Tunneling protocol/network overlay support
- Scenarios that handle NSB test cases execution
  - NSPerf - scenario that handles generic NSB test case execution (setup and init tg/vnf, trigger traffic on tg, collect kpi)
  - NSPerf-RFC2544 - scenario that allows repeatable triggering of traffic on traffic generators until test case acceptance criteria is met (for example RFC2544 binary search)
- Test case samples
  - Ping
  - Trex
  - vPE, vCGNAT, vFirewall etc - ipv4 throughput, latency etc
- Traffic generators i.e. Trex, ab/nginx, ixia, iperf, etc
- KPIs for a given use case:
  - System agent support for collecting NFVi KPI. This includes:

    * CPU statistic
    * Memory BW
    * OVS-DPDK Stats
  - Network KPIs e.g. inpackets, outpackets, thoughput, latency
  - VNF KPIs e.g. packet_in, packet_drop, packet_fwd

2.11.3 Architecture

The Network Service (NS) defines a set of Virtual Network Functions (VNF) connected together using NFV infrastructure.

The Yardstick NSB extension can support multiple VNFs created by different vendors including traffic generators. Every VNF being tested has its own data model. The Network service defines a VNF modelling on base of performed network functionality. The part of the data model is a set of the configuration parameters, number of connection points used and flavor including core and memory amount.

ETSI defines a Network Service as a set of configurable VNFs working in some NFV Infrastructure connecting each other using Virtual Links available through Connection Points. The ETSI MANO specification defines a set of management entities called Network Service Descriptors (NSD) and VNF Descriptors (VNFD) that define real Network Service. The picture below makes an example how the real Network Operator use-case can map into ETSI Network service definition.

Network Service framework performs the necessary test steps. It may involve:

- Interacting with traffic generator and providing the inputs on traffic type / packet structure to generate the required traffic as per the test case. Traffic profiles will be used for this.
- Executing the commands required for the test procedure and analyses the command output for confirming whether the command got executed correctly or not e.g. as per the test case, run the traffic for the given time period and wait for the necessary time delay.
- Verify the test result.
- Validate the traffic flow from SUT.
- Fetch the data from SUT and verify the value as per the test case.
- Upload the logs from SUT onto the Test Harness server
- Retrieve the KPI's provided by particular VNF

Components of Network Service

- *Models for Network Service benchmarking*: The Network Service benchmarking requires the proper modelling approach. The NSB provides models using Python files and defining of NSDs and VNFDs.

The benchmark control application being a part of OPNFV Yardstick can call that Python models to instantiate and configure the VNFs. Depending on infrastructure type (bare-metal or fully virtualized) that calls could be made directly or using MANO system.

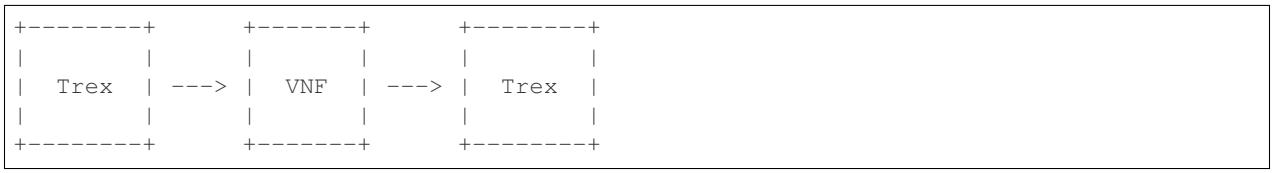
- *Traffic generators in NSB*: Any benchmark application requires a set of traffic generator and traffic profiles defining the method in which traffic is generated.

The Network Service benchmarking model extends the Network Service definition with a set of Traffic Generators (TG) that are treated same way as other VNFs being a part of benchmarked network service. Same as other VNFs the traffic generator are instantiated and terminated.

Every traffic generator has own configuration defined as a traffic profile and a set of KPIs supported. The python models for TG is extended by specific calls to listen and generate traffic.

- *The stateless TREX traffic generator*: The main traffic generator used as Network Service stimulus is open source TREX tool.

The TREX tool can generate any kind of stateless traffic.



Supported testcases scenarios:

- Correlated UDP traffic using TREX traffic generator and replay VNF.
  - Using different IMIX configuration like pure voice, pure video traffic etc
  - Using different number IP flows e.g. 1, 1K, 16K, 64K, 256K, 1M flows
  - Using different number of rules configured e.g. 1, 1K, 10K rules

For UDP correlated traffic following Key Performance Indicators are collected for every combination of test case parameters:

- RFC2544 throughput for various loss rate defined (1% is a default)

KPI Collection

KPI collection is the process of sampling KPIs at multiple intervals to allow for investigation into anomalies during runtime. Some KPI intervals are adjustable. KPIs are collected from traffic generators and NFVI for the SUT. There is already some reporting in NSB available, but NSB collects all KPIs for analytics to process.

Below is an example list of basic KPIs:

- Throughput
- Latency
- Packet delay variation
- Maximum establishment rate
- Maximum tear-down rate
- Maximum simultaneous number of sessions

Of course, there can be many other KPIs that will be relevant for a specific NFVI, but in most cases these KPIs are enough to give you a basic picture of the SUT. NSB also uses *collectd* in order to collect the KPIs. Currently the following collectd plug-ins are enabled for NSB testcases:

- Libvirt
- Interface stats
- OvS events
- vSwitch stats
- Huge Pages
- RAM

- CPU usage
- Intel® PMU
- Intel® RDT

2.11.4 Graphical Overview

NSB Testing with Yardstick framework facilitate performance testing of various VNFs provided.

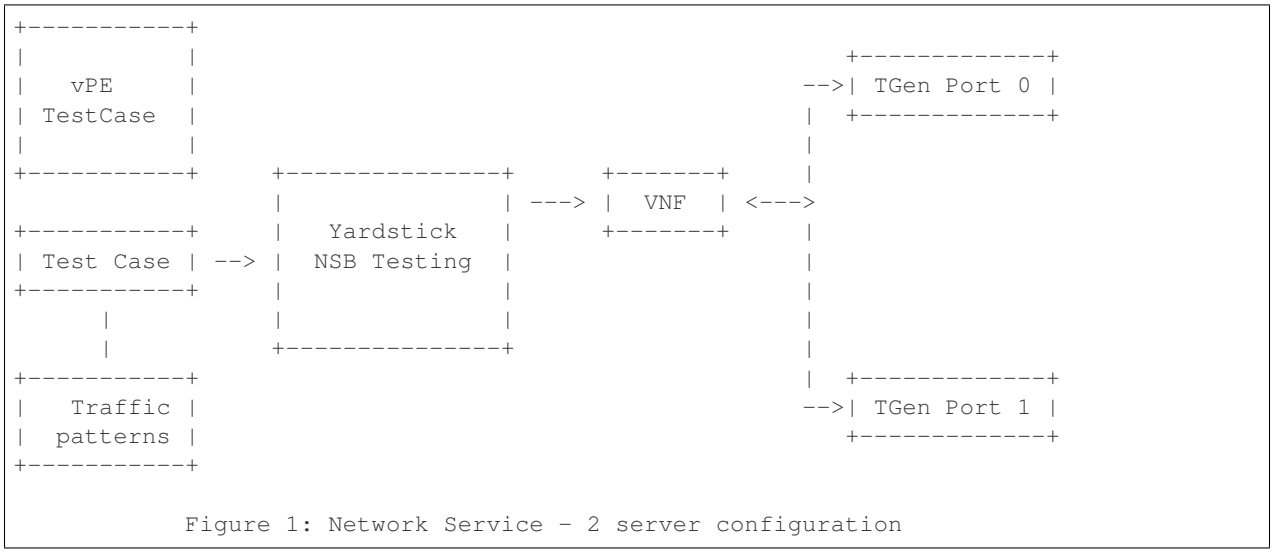


Figure 1: Network Service - 2 server configuration

VNFs supported for chracterization

1. CGNAPT - Carrier Grade Network Address and port Translation
2. vFW - Virtual Firewall
3. vACL - Access Control List
4. **PROX - Packet pROcessing eXecution engine:**
    - VNF can act as Drop, Basic Forwarding (no touch), L2 Forwarding (change MAC), GRE encap/decap, Load balance based on packet fields, Symmetric load balancing
    - QinQ encap/decap IPv4/IPv6, ARP, QoS, Routing, Unmpls, Policing, ACL
5. UDP_Replay

2.12 NSB Installation

2.12.1 Abstract

The steps needed to run Yardstick with NSB testing are:

- Install Yardstick (NSB Testing).
- Setup/reference `pod.yaml` describing Test topology.
- Create/reference the test configuration yaml file.

- Run the test case.

2.12.2 Prerequisites

Refer to *Yardstick Installation* for more information on Yardstick prerequisites.

Several prerequisites are needed for Yardstick (VNF testing):

- Python Modules: pyzmq, pika.
- flex
- bison
- build-essential
- automake
- libtool
- librabbitmq-dev
- rabbitmq-server
- collectd
- intel-cmt-cat

Hardware & Software Ingredients

SUT requirements:

Item	Description
Memory	Min 20GB
NICs	2 x 10G
OS	Ubuntu 16.04.3 LTS
kernel	4.4.0-34-generic
DPDK	17.02

Boot and BIOS settings:

Boot settings	default_hugepagesz=1G hugepagesz=1G hugepages=16 hugepagesz=2M hugepages=2048 isolcpus=1-11,22-33 nohz_full=1-11,22-33 rcu_nocbs=1-11,22-33 iommu=on iommu=pt intel_iommu=on Note: nohz_full and rcu_nocbs is to disable Linux kernel interrupts
BIOS	CPU Power and Performance Policy <Performance> CPU C-state Disabled CPU P-state Disabled Enhanced Intel® Speedstep® Tech Disabl Hyper-Threading Technology (If supported) Enabled Virtualization Techology Enabled Intel(R) VT for Direct I/O Enabled Coherency Enabled Turbo Boost Disabled

2.12.3 Install Yardstick (NSB Testing)

Yardstick with NSB can be installed using `nsb_setup.sh`. The `nsb_setup.sh` allows to:

1. Install Yardstick in specified mode: bare metal or container. Refer *Yardstick Installation*.

2. Install package dependencies on remote servers used as traffic generator or sample VNF. Install DPDK, sample VNFs, TREX, collectd. Add such servers to `install-inventory.ini` file to either `yardstick-standalone` or `yardstick-baremetal` server groups. It configures IOMMU, hugepages, open file limits, CPU isolation, etc.
3. Build VM image either nsb or normal. The nsb VM image is used to run Yardstick sample VNF tests, like vFW, vACL, vCGNAPT, etc. The normal VM image is used to run Yardstick ping tests in OpenStack context.
4. Add nsb or normal VM image to OpenStack together with OpenStack variables.

Firstly, configure the network proxy, either using the environment variables or setting the global environment file.

Set environment in the file:

```
http_proxy='http://proxy.company.com:port'
https_proxy='http://proxy.company.com:port'
```

Set environment variables:

```
export http_proxy='http://proxy.company.com:port'
export https_proxy='http://proxy.company.com:port'
```

Download the source code and check out the latest stable branch:

```
git clone https://gerrit.opnfv.org/gerrit/yardstick
cd yardstick
# Switch to latest stable branch
git checkout stable/gambia
```

Modify the Yardstick installation inventory used by Ansible:

```
cat ./ansible/install-inventory.ini
[jumphost]
localhost ansible_connection=local

# section below is only due backward compatibility.
# it will be removed later
[yardstick:children]
jumphost

[yardstick-baremetal]
baremetal ansible_host=192.168.2.51 ansible_connection=ssh

[yardstick-standalone]
standalone ansible_host=192.168.2.52 ansible_connection=ssh

[all:vars]
# Uncomment credentials below if needed
  ansible_user=root
  ansible_ssh_pass=root
# ansible_ssh_private_key_file=/root/.ssh/id_rsa
# When IMG_PROPERTY is passed neither normal nor nsb set
# "path_to_vm=/path/to/image" to add it to OpenStack
# path_to_img=/tmp/workspace/yardstick-image.img

# List of CPUs to be isolated (not used by default)
# Grub line will be extended with:
# "isolcpus=<ISOL_CPUS> nohz=on nohz_full=<ISOL_CPUS> rcu_nocbs=1<ISOL_CPUS>"
# ISOL_CPUS=2-27,30-55 # physical cpu's for all NUMA nodes, four cpu's reserved
```

Warning: Before running `nsb_setup.sh` make sure python is installed on servers added to `yardstick-standalone` and `yardstick-baremetal` groups.

Note: SSH access without password needs to be configured for all your nodes defined in `install-inventory.ini` file. If you want to use password authentication you need to install `sshpass`:

```
sudo -EH apt-get install sshpass
```

Note: A VM image built by other means than Yardstick can be added to OpenStack. Uncomment and set correct path to the VM image in the `install-inventory.ini` file:

```
path_to_img=/tmp/workspace/yardstick-image.img
```

Note: CPU isolation can be applied to the remote servers, like: ISOL_CPUS=2-27,30-55. Uncomment and modify accordingly in `install-inventory.ini` file.

By default `nsb_setup.sh` pulls Yardstick image based on Ubuntu 16.04 from docker hub and starts container, builds NSB VM image based on Ubuntu 16.04, installs packages to the servers given in `yardstick-standalone` and `yardstick-baremetal` host groups.

To pull Yardstick built based on Ubuntu 18 run:

```
./nsb_setup.sh -i opnfv/yardstick-ubuntu-18.04:latest
```

To change default behavior modify parameters for `install.yaml` in `nsb_setup.sh` file.

Refer chapter *Yardstick Installation* for more details on `install.yaml` parameters.

To execute an installation for a **BareMetal** or a **Standalone context**:

```
./nsb_setup.sh
```

To execute an installation for an **OpenStack** context:

```
./nsb_setup.sh <path to admin-openrc.sh>
```

Note: Yardstick may not be operational after distributive linux kernel update if it has been installed before. Run `nsb_setup.sh` again to resolve this.

Warning: The Yardstick VM image (NSB or normal) cannot be built inside a VM.

Warning: The `nsb_setup.sh` configures huge pages, CPU isolation, IOMMU on the grub. Reboot of the servers from `yardstick-standalone` or `yardstick-baremetal` groups in the file `install-inventory.ini` is required to apply those changes.

The above commands will set up Docker with the latest Yardstick code. To execute:

```
docker exec -it yardstick bash
```

Note: It may be needed to configure tty in docker container to extend commandline character length, for example:

stty size rows 58 cols 234

It will also automatically download all the packages needed for NSB Testing setup. Refer chapter *Yardstick Installation* for more on Docker: *Install Yardstick using Docker (first option) (recommended)*

Bare Metal context example

Let's assume there are three servers acting as TG, sample VNF DUT and jump host.

Perform following steps to install NSB:

1. Clone Yardstick repo to jump host.
2. Add TG and DUT servers to `yardstick-baremetal` group in `install-inventory.ini` file to install NSB and dependencies. Install python on servers.
3. Start deployment using docker image based on Ubuntu 16:

```
./nsb_setup.sh
```

4. Reboot bare metal servers.
5. Enter to yardstick container and modify pod yaml file and run tests.

Standalone context example for Ubuntu 18

Let's assume there are three servers acting as TG, sample VNF DUT and jump host. Ubuntu 18 is installed on all servers.

Perform following steps to install NSB:

1. Clone Yardstick repo to jump host.
2. Add TG server to `yardstick-baremetal` group in `install-inventory.ini` file to install NSB and dependencies. Add server where VM with sample VNF will be deployed to `yardstick-standalone` group in `install-inventory.ini` file. Target VM image named `yardstick-nsb-image.img` will be placed to `/var/lib/libvirt/images/`. Install python on servers.
3. Modify `nsb_setup.sh` on jump host:

```
ansible-playbook \
-e IMAGE_PROPERTY='nsb' \
-e OS_RELEASE='bionic' \
-e INSTALLATION_MODE='container_pull' \
-e YARD_IMAGE_ARCH='amd64' ${extra_args} \
-i install-inventory.ini install.yaml
```

4. Start deployment with Yardstick docker images based on Ubuntu 18:

```
./nsb_setup.sh -i opnfv/yardstick-ubuntu-18.04:latest -o <openrc_file>
```

5. Reboot servers.

6. Enter to yardstick container and modify pod yaml file and run tests.

2.12.4 System Topology

```
+----------+               +----------+
|          |               |          |
|          | (0)----->(0)  |          |
|    TG1   |               |    DUT   |
|          |               |          |
|          | (1)<-----(1)  |          |
+----------+               +----------+
trafficgen_0                    vnf
```

2.12.5 Environment parameters and credentials

Configure yardstick.conf

If you did not run `yardstick env influxdb` inside the container to generate `yardstick.conf`, then create the config file manually (run inside the container):

```
cp ./etc/yardstick/yardstick.conf.sample /etc/yardstick/yardstick.conf
vi /etc/yardstick/yardstick.conf
```

Add `trex_path`, `trex_client_lib` and `bin_path` to the `nsb` section:

```
[DEFAULT]
debug = True
dispatcher = influxdb

[dispatcher_influxdb]
timeout = 5
target = http://{YOUR_IP_HERE}:8086
db_name = yardstick
username = root
password = root

[nsb]
trex_path=/opt/nsb_bin/trex/scripts
bin_path=/opt/nsb_bin
trex_client_lib=/opt/nsb_bin/trex_client/stl
```

2.12.6 Run Yardstick - Network Service Testcases

NS testing - using yardstick CLI

> See *Yardstick Installation*

Connect to the Yardstick container:

```
docker exec -it yardstick /bin/bash
```

If you're running `heat` testcases and `nsb_setup.sh` was not used:

```
source /etc/yardstick/openstack.creds
```

In addition to the above, you need to set the `EXTERNAL_NETWORK` for OpenStack:

```
export EXTERNAL_NETWORK="<openstack public network>"
```

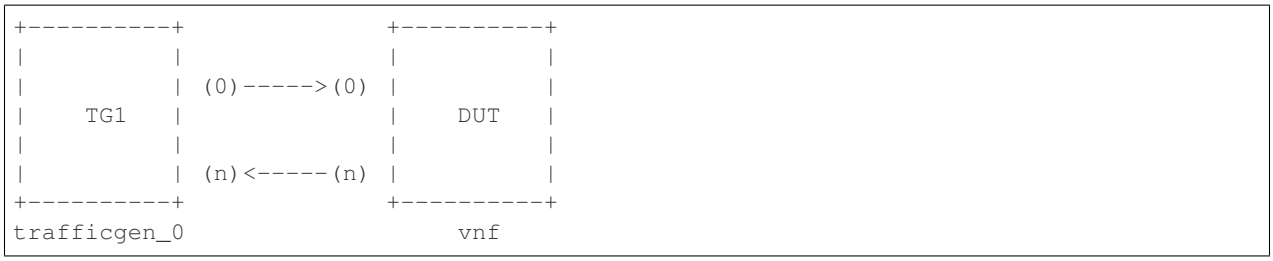
Finally, you should be able to run the testcase:

```
yardstick --debug task start yardstick/samples/vnf_samples/nsut/<vnf>/<test case>
```

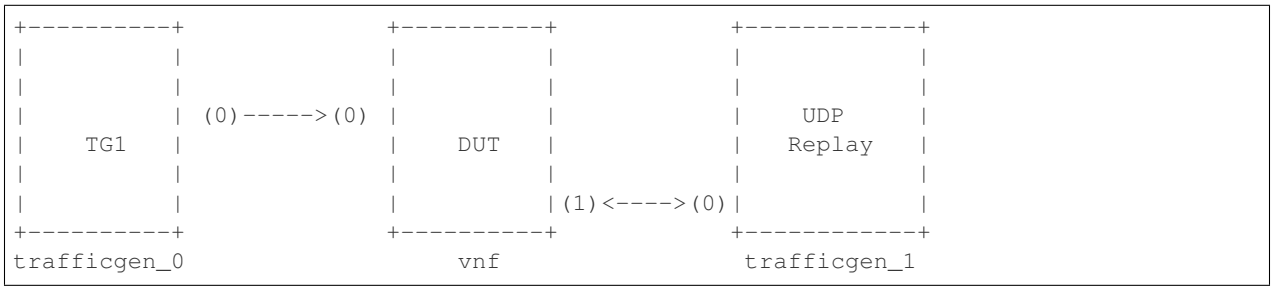
2.12.7 Network Service Benchmarking - Bare-Metal

Bare-Metal Config pod.yaml describing Topology

Bare-Metal 2-Node setup



Bare-Metal 3-Node setup - Correlated Traffic



Bare-Metal Config pod.yaml

Before executing Yardstick test cases, make sure that `pod.yaml` reflects the topology and update all the required fields.:

```
cp <yardstick>/etc/yardstick/nodes/pod.yaml.nsb.sample /etc/yardstick/nodes/pod.yaml
```

```
nodes:
-
    name: trafficgen_0
    role: TrafficGen
    ip: 1.1.1.1
    user: root
```

(continues on next page)

(continued from previous page)

```
    password: r00t
    interfaces:
        xe0:  # logical name from topology.yaml and vnfd.yaml
            vpci:      "0000:07:00.0"
            driver:    i40e # default kernel driver
            dpdk_port_num: 0
            local_ip: "152.16.100.20"
            netmask:   "255.255.255.0"
            local_mac: "00:00:00:00:00:01"
        xe1:  # logical name from topology.yaml and vnfd.yaml
            vpci:      "0000:07:00.1"
            driver:    i40e # default kernel driver
            dpdk_port_num: 1
            local_ip: "152.16.40.20"
            netmask:   "255.255.255.0"
            local_mac: "00:00:00:00:00:02"

-
    name: vnf
    role: vnf
    ip: 1.1.1.2
    user: root
    password: r00t
    host: 1.1.1.2 #BM - host == ip, virtualized env - Host - compute node
    interfaces:
        xe0:  # logical name from topology.yaml and vnfd.yaml
            vpci:      "0000:07:00.0"
            driver:    i40e # default kernel driver
            dpdk_port_num: 0
            local_ip: "152.16.100.19"
            netmask:   "255.255.255.0"
            local_mac: "00:00:00:00:00:03"

        xe1:  # logical name from topology.yaml and vnfd.yaml
            vpci:      "0000:07:00.1"
            driver:    i40e # default kernel driver
            dpdk_port_num: 1
            local_ip: "152.16.40.19"
            netmask:   "255.255.255.0"
            local_mac: "00:00:00:00:00:04"
    routing_table:
    - network: "152.16.100.20"
      netmask: "255.255.255.0"
      gateway: "152.16.100.20"
      if: "xe0"
    - network: "152.16.40.20"
      netmask: "255.255.255.0"
      gateway: "152.16.40.20"
      if: "xe1"
    nd_route_tbl:
    - network: "0064:ff9b:0:0:0:0:9810:6414"
      netmask: "112"
      gateway: "0064:ff9b:0:0:0:0:9810:6414"
      if: "xe0"
    - network: "0064:ff9b:0:0:0:0:9810:2814"
      netmask: "112"
      gateway: "0064:ff9b:0:0:0:0:9810:2814"
```

(continues on next page)

(continued from previous page)

```
      if: "xe1"
```

2.12.8 Standalone Virtualization

VM can be deployed manually or by Yardstick. If parameter *vm_deploy* is set to *True* VM will be deployed by Yardstick. Otherwise VM should be deployed manually. Test case example, context section:

```
contexts:
 ...
 vm_deploy: True
```

SR-IOV

SR-IOV Pre-requisites

On Host, where VM is created:

1. Create and configure a bridge named `br-int` for VM to connect to external network. Currently this can be done using VXLAN tunnel.

   Execute the following on host, where VM is created:

```
ip link add type vxlan remote <Jumphost IP> local <DUT IP> id <ID: 10>␣
↪dstport 4789
brctl addbr br-int
brctl addif br-int vxlan0
ip link set dev vxlan0 up
ip addr add <IP#1, like: 172.20.2.1/24> dev br-int
ip link set dev br-int up
```

   Note: You may need to add extra rules to iptable to forward traffic.

```
iptables -A FORWARD -i br-int -s <network ip address>/<netmask> -j ACCEPT
iptables -A FORWARD -o br-int -d <network ip address>/<netmask> -j ACCEPT
```

   Execute the following on a jump host:

```
ip link add type vxlan remote <DUT IP> local <Jumphost IP> id <ID: 10>␣
↪dstport 4789
ip addr add <IP#2, like: 172.20.2.2/24> dev vxlan0
ip link set dev vxlan0 up
```

   Note: Host and jump host are different baremetal servers.

2. Modify test case management CIDR. IP addresses IP#1, IP#2 and CIDR must be in the same network.

```
servers:
  vnf_0:
    network_ports:
```

(continues on next page)

(continued from previous page)

```
      mgmt:
        cidr: '1.1.1.7/24'
```

3. Build guest image for VNF to run. Most of the sample test cases in Yardstick are using a guest image called `yardstick-nsb-image` which deviates from an Ubuntu Cloud Server image Yardstick has a tool for building this custom image with SampleVNF. It is necessary to have `sudo` rights to use this tool.

   Also you may need to install several additional packages to use this tool, by following the commands below:

   ```
   sudo apt-get update && sudo apt-get install -y qemu-utils kpartx
   ```

   This image can be built using the following command in the directory where Yardstick is installed:

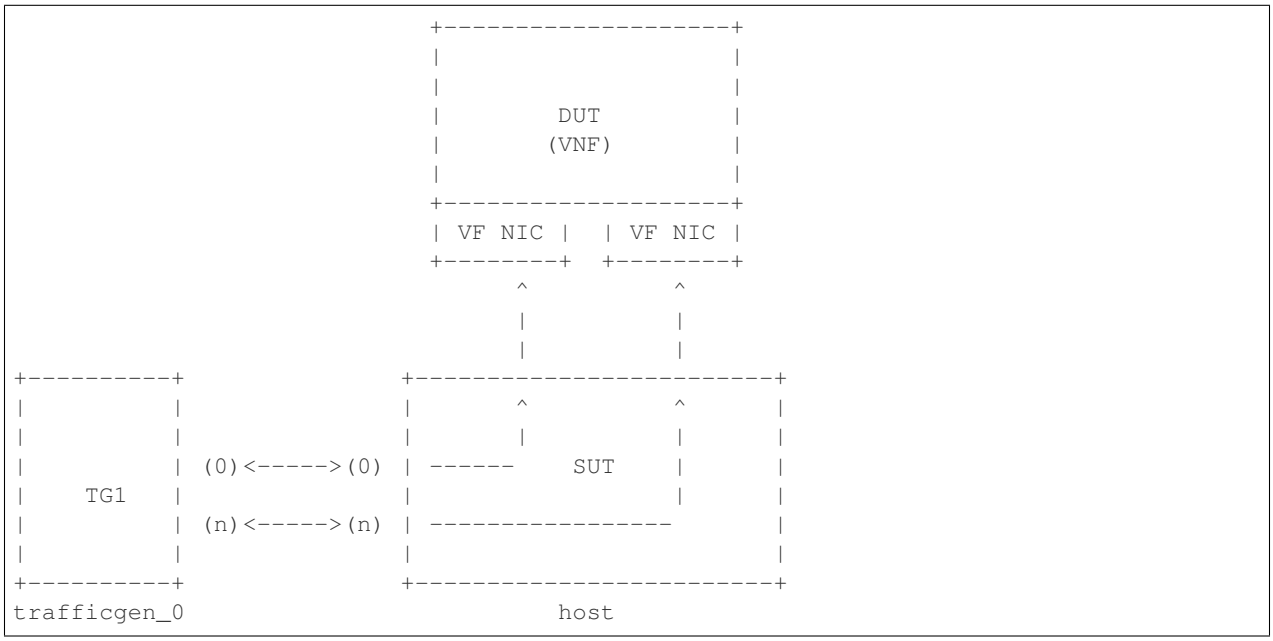
   ```
   export YARD_IMG_ARCH='amd64'
   sudo echo "Defaults env_keep += \'YARD_IMG_ARCH\'" >> /etc/sudoers
   ```

   For instructions on generating a cloud image using Ansible, refer to *Yardstick Installation*.

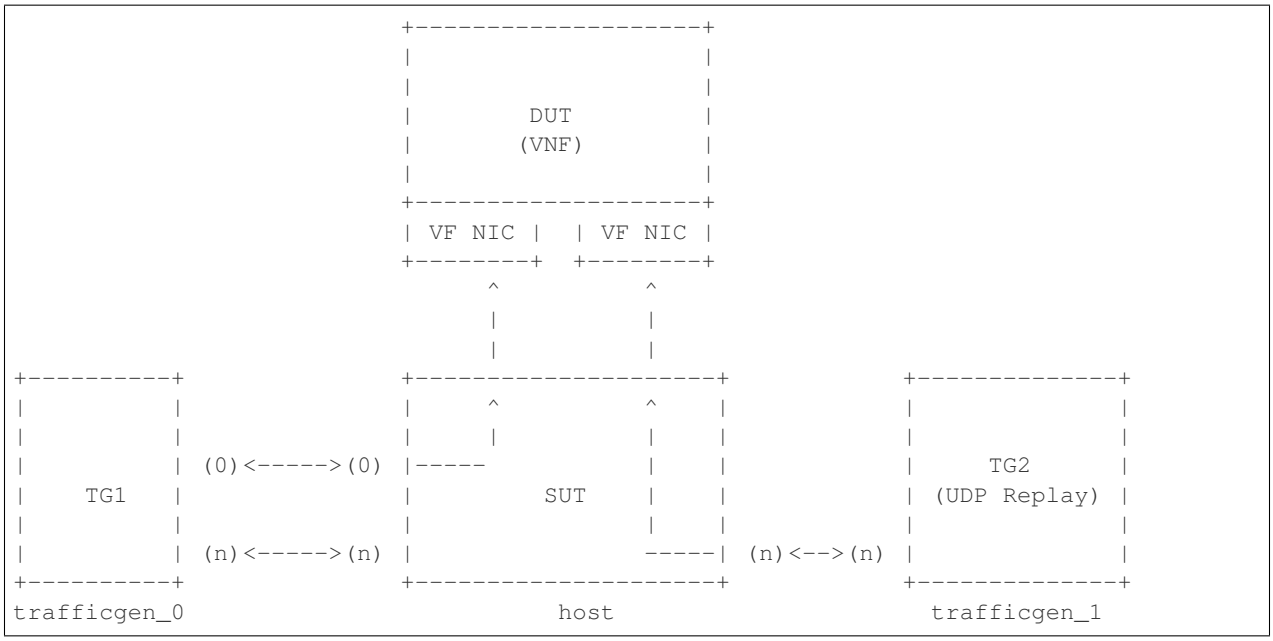
   Note: VM should be build with static IP and be accessible from the Yardstick host.

SR-IOV Config pod.yaml describing Topology

SR-IOV 2-Node setup



SR-IOV 3-Node setup - Correlated Traffic



Before executing Yardstick test cases, make sure that `pod.yaml` reflects the topology and update all the required fields.

```
cp <yardstick>/etc/yardstick/nodes/standalone/trex_bm.yaml.sample /etc/yardstick/
↪nodes/standalone/pod_trex.yaml
cp <yardstick>/etc/yardstick/nodes/standalone/host_sriov.yaml /etc/yardstick/nodes/
↪standalone/host_sriov.yaml
```

Note: Update all the required fields like ip, user, password, pcis, etc...

SR-IOV Config pod_trex.yaml

```
nodes:
-
    name: trafficgen_0
    role: TrafficGen
    ip: 1.1.1.1
    user: root
    password: r00t
    key_filename: /root/.ssh/id_rsa
    interfaces:
        xe0:  # logical name from topology.yaml and vnfd.yaml
            vpci:      "0000:07:00.0"
            driver:    i40e # default kernel driver
            dpdk_port_num: 0
            local_ip: "152.16.100.20"
            netmask:   "255.255.255.0"
            local_mac: "00:00:00:00:00:01"
        xe1:  # logical name from topology.yaml and vnfd.yaml
            vpci:      "0000:07:00.1"
            driver:    i40e # default kernel driver
```

(continues on next page)

(continued from previous page)

```
            dpdk_port_num: 1
            local_ip: "152.16.40.20"
            netmask:  "255.255.255.0"
            local_mac: "00:00:00:00:00:02"
```

SR-IOV Config host_sriov.yaml

```
nodes:
-
   name: sriov
   role: Sriov
   ip: 192.168.100.101
   user: ""
   password: ""
```

SR-IOV testcase update: `<yardstick>/samples/vnf_samples/nsut/vfw/tc_sriov_rfc2544_ipv4_1rule_1flow_64B_trex.yaml`

Update contexts section

```
contexts:
 - name: yardstick
   type: Node
   file: /etc/yardstick/nodes/standalone/pod_trex.yaml
 - type: StandaloneSriov
   file: /etc/yardstick/nodes/standalone/host_sriov.yaml
   name: yardstick
   vm_deploy: True
   flavor:
     images: "/var/lib/libvirt/images/ubuntu.qcow2"
     ram: 4096
     extra_specs:
       hw:cpu_sockets: 1
       hw:cpu_cores: 6
       hw:cpu_threads: 2
     user: "" # update VM username
     password: "" # update password
   servers:
     vnf_0:
       network_ports:
         mgmt:
           cidr: '1.1.1.61/24'  # Update VM IP address, if static, <ip>/<mask> or if
↪dynamic, <start of ip>/<mask>
         xe0:
           - uplink_0
         xe1:
           - downlink_0
   networks:
     uplink_0:
       phy_port: "0000:05:00.0"
       vpci: "0000:00:07.0"
       cidr: '152.16.100.10/24'
```

(continues on next page)

(continued from previous page)

```
    gateway_ip: '152.16.100.20'
  downlink_0:
    phy_port: "0000:05:00.1"
    vpci: "0000:00:08.0"
    cidr: '152.16.40.10/24'
    gateway_ip: '152.16.100.20'
```

SRIOV configuration options

The only configuration option available for SRIOV is *vpci*. It is used as base address for VFs that are created during SRIOV test case execution.

```
networks:
  uplink_0:
    phy_port: "0000:05:00.0"
    vpci: "0000:00:07.0"
    cidr: '152.16.100.10/24'
    gateway_ip: '152.16.100.20'
  downlink_0:
    phy_port: "0000:05:00.1"
    vpci: "0000:00:08.0"
    cidr: '152.16.40.10/24'
    gateway_ip: '152.16.100.20'
```

VM image properties

VM image properties example under *flavor* section:

```
flavor:
  images: <path>
  ram: 8192
  extra_specs:
     machine_type: 'pc-i440fx-xenial'
     hw:cpu_sockets: 1
     hw:cpu_cores: 6
     hw:cpu_threads: 2
     hw_socket: 0
     cputune: |
       <cputune>
         <vcpupin vcpu="0" cpuset="7"/>
         <vcpupin vcpu="1" cpuset="8"/>
         ...
         <vcpupin vcpu="11" cpuset="18"/>
         <emulatorpin cpuset="11"/>
       </cputune>
  user: ""
  password: ""
```

VM image properties description:

Parameters	Detail
images	Path to the VM image generated by `nsb_setup.sh` Default path is `/var/lib/libvirt/images/` Default file name `yardstick-nsb-image.img` or `yardstick-image.img`
ram	Amount of RAM to be used for VM Default is 4096 MB
hw:cpu_sockets	Number of sockets provided to the guest VM Default is 1
hw:cpu_cores	Number of cores provided to the guest VM Default is 2
hw:cpu_threads	Number of threads provided to the guest VM Default is 2
hw_socket	Generate vcpu cpuset from given HW socket Default is 0
cputune	Maps virtual cpu with logical cpu
machine_type	Machine type to be emulated in VM Default is 'pc-i440fx-xenial'
user	User name to access the VM Default value is 'root'
password	Password to access the VM

OVS-DPDK

OVS-DPDK Pre-requisites

On Host, where VM is created:

1. Create and configure a bridge named `br-int` for VM to connect to external network. Currently this can be done using VXLAN tunnel.

   Execute the following on host, where VM is created:

```
ip link add type vxlan remote <Jumphost IP> local <DUT IP> id <ID: 10>␣
↪dstport 4789
brctl addbr br-int
brctl addif br-int vxlan0
ip link set dev vxlan0 up
ip addr add <IP#1, like: 172.20.2.1/24> dev br-int
ip link set dev br-int up
```

   Note: May be needed to add extra rules to iptable to forward traffic.

```
iptables -A FORWARD -i br-int -s <network ip address>/<netmask> -j ACCEPT
iptables -A FORWARD -o br-int -d <network ip address>/<netmask> -j ACCEPT
```

   Execute the following on a jump host:

```
ip link add type vxlan remote <DUT IP> local <Jumphost IP> id <ID: 10>␣
↪dstport 4789
ip addr add <IP#2, like: 172.20.2.2/24> dev vxlan0
ip link set dev vxlan0 up
```

   Note: Host and jump host are different baremetal servers.

2. Modify test case management CIDR. IP addresses IP#1, IP#2 and CIDR must be in the same network.

```
servers:
  vnf_0:
    network_ports:
      mgmt:
        cidr: '1.1.1.7/24'
```

3. Build guest image for VNF to run. Most of the sample test cases in Yardstick are using a guest image called `yardstick-nsb-image` which deviates from an Ubuntu Cloud Server image Yardstick has a tool for building this custom image with SampleVNF. It is necessary to have `sudo` rights to use this tool.

   You may need to install several additional packages to use this tool, by following the commands below:

```
sudo apt-get update && sudo apt-get install -y qemu-utils kpartx
```

   This image can be built using the following command in the directory where Yardstick is installed:

```
export YARD_IMG_ARCH='amd64'
sudo echo "Defaults env_keep += \'YARD_IMG_ARCH\'" >> /etc/sudoers
sudo tools/yardstick-img-dpdk-modify tools/ubuntu-server-cloudimg-
↪samplevnf-modify.sh
```

for more details refer to chapter *Yardstick Installation*

Note: VM should be build with static IP and should be accessible from yardstick host.

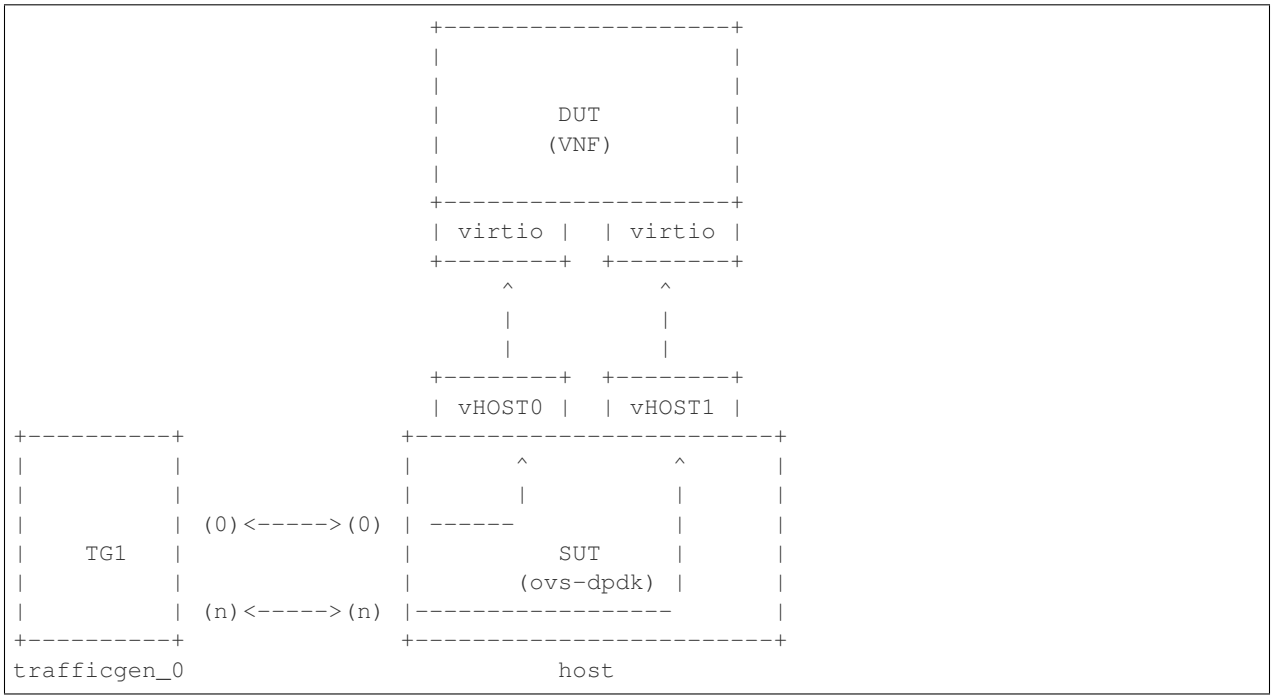
4. OVS & DPDK version:

- OVS 2.7 and DPDK 16.11.1 above version is supported

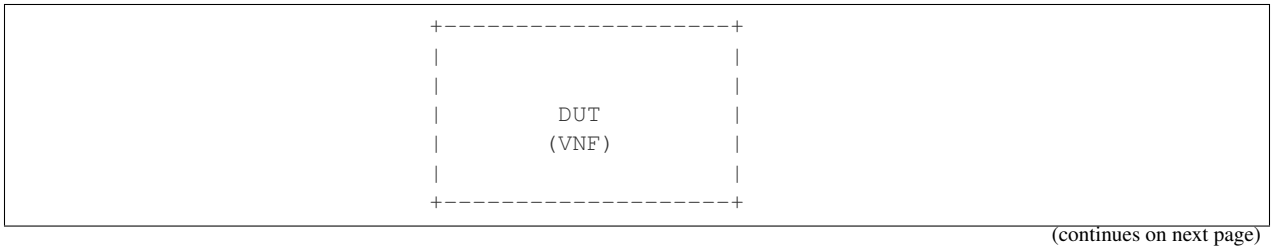
Refer setup instructions at OVS-DPDK on host.

OVS-DPDK Config pod.yaml describing Topology

OVS-DPDK 2-Node setup

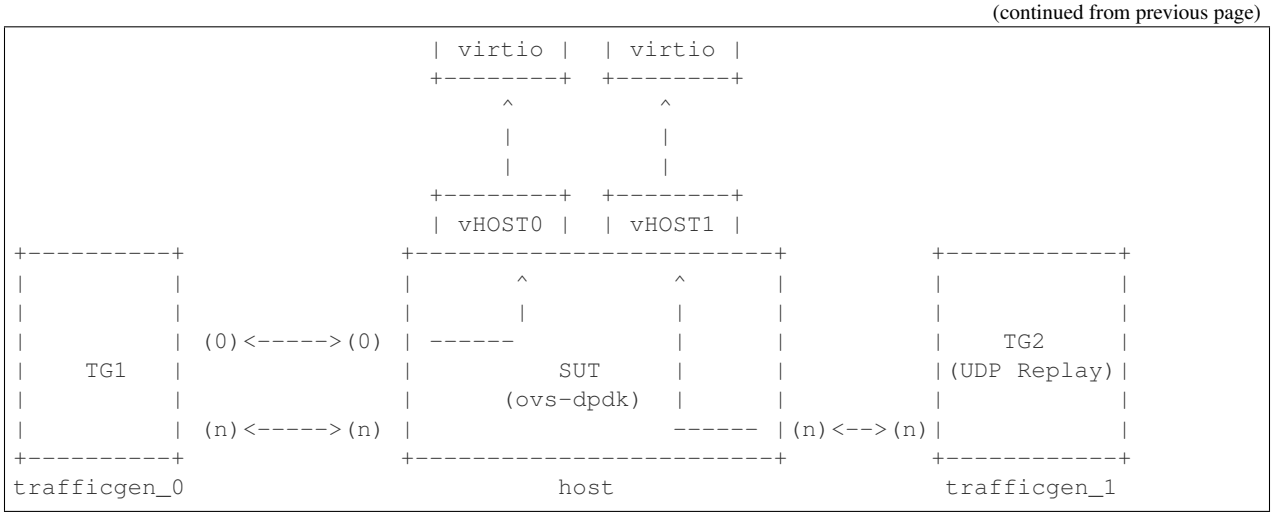


OVS-DPDK 3-Node setup - Correlated Traffic



(continues on next page)

(continued from previous page)



Before executing Yardstick test cases, make sure that the `pod.yaml` reflects the topology and update all the required fields:

```
cp <yardstick>/etc/yardstick/nodes/standalone/trex_bm.yaml.sample /etc/yardstick/
↪nodes/standalone/pod_trex.yaml
cp <yardstick>/etc/yardstick/nodes/standalone/host_ovs.yaml /etc/yardstick/nodes/
↪standalone/host_ovs.yaml
```

Note: Update all the required fields like ip, user, password, pcis, etc...

OVS-DPDK Config pod_trex.yaml

```
nodes:
-
  name: trafficgen_0
  role: TrafficGen
  ip: 1.1.1.1
  user: root
  password: r00t
  interfaces:
      xe0:  # logical name from topology.yaml and vnfd.yaml
          vpci:      "0000:07:00.0"
          driver:    i40e # default kernel driver
          dpdk_port_num: 0
          local_ip: "152.16.100.20"
          netmask:   "255.255.255.0"
          local_mac: "00:00:00:00:00:01"
      xe1:  # logical name from topology.yaml and vnfd.yaml
          vpci:      "0000:07:00.1"
          driver:    i40e # default kernel driver
          dpdk_port_num: 1
          local_ip: "152.16.40.20"
          netmask:   "255.255.255.0"
          local_mac: "00:00:00:00:00:02"
```

OVS-DPDK Config host_ovs.yaml

```
nodes:
-
   name: ovs_dpdk
   role: OvsDpdk
   ip: 192.168.100.101
   user: ""
   password: ""
```

ovs_dpdk testcase update: <yardstick>/samples/vnf_samples/nsut/vfw/tc_ovs_rfc2544_ipv4_1rule_1flow_64B_trex.yaml

Update contexts section

```
contexts:
 - name: yardstick
   type: Node
   file: /etc/yardstick/nodes/standalone/pod_trex.yaml
 - type: StandaloneOvsDpdk
   name: yardstick
   file: /etc/yardstick/nodes/standalone/pod_ovs.yaml
   vm_deploy: True
   ovs_properties:
     version:
       ovs: 2.7.0
       dpdk: 16.11.1
     pmd_threads: 2
     ram:
       socket_0: 2048
       socket_1: 2048
     queues: 4
     vpath: "/usr/local"

   flavor:
     images: "/var/lib/libvirt/images/ubuntu.qcow2"
     ram: 4096
     extra_specs:
       hw:cpu_sockets: 1
       hw:cpu_cores: 6
       hw:cpu_threads: 2
     user: "" # update VM username
     password: "" # update password
   servers:
     vnf_0:
       network_ports:
         mgmt:
           cidr: '1.1.1.61/24'  # Update VM IP address, if static, <ip>/<mask> or if␣
↪dynamic, <start of ip>/<mask>
         xe0:
           - uplink_0
         xe1:
           - downlink_0
   networks:
     uplink_0:
```

(continues on next page)

(continued from previous page)

```
      phy_port: "0000:05:00.0"
      vpci: "0000:00:07.0"
      cidr: '152.16.100.10/24'
      gateway_ip: '152.16.100.20'
    downlink_0:
      phy_port: "0000:05:00.1"
      vpci: "0000:00:08.0"
      cidr: '152.16.40.10/24'
      gateway_ip: '152.16.100.20'
```

OVS-DPDK configuration options

There are number of configuration options available for OVS-DPDK context in test case. Mostly they are used for performance tuning.

OVS-DPDK properties:

OVS-DPDK properties example under *ovs_properties* section:

```
ovs_properties:
  version:
    ovs: 2.8.1
    dpdk: 17.05.2
  pmd_threads: 4
  pmd_cpu_mask: "0x3c"
  ram:
   socket_0: 2048
   socket_1: 2048
  queues: 2
  vpath: "/usr/local"
  max_idle: 30000
  lcore_mask: 0x02
  dpdk_pmd-rxq-affinity:
    0: "0:2,1:2"
    1: "0:2,1:2"
    2: "0:3,1:3"
    3: "0:3,1:3"
  vhost_pmd-rxq-affinity:
    0: "0:3,1:3"
    1: "0:3,1:3"
    2: "0:4,1:4"
    3: "0:4,1:4"
```

OVS-DPDK properties description:

Parameters	Detail
version	Version of OVS and DPDK to be installed There is a relation between OVS and DPDK version which can be found at OVS-DPDK-versions By default OVS: 2.6.0, DPDK: 16.07.2
lcore_mask	Core bitmask used during DPDK initialization where the non-datapath OVS-DPDK threads such as handler and revalidator threads run
pmd_cpu_mask	Core bitmask that sets which cores are used by OVS-DPDK for datapath packet processing
pmd_threads	Number of PMD threads used by OVS-DPDK for datapath This core mask is evaluated in Yardstick It will be used if pmd_cpu_mask is not given Default is 2
ram	Amount of RAM to be used for each socket, MB Default is 2048 MB
queues	Number of RX queues used for DPDK physical interface
dpdk_pmd-rxq-affinity	RX queue assignment to PMD threads for DPDK e.g.: <port number> : <queue-id>:<core-id>
vhost_pmd-rxq-affinity	RX queue assignment to PMD threads for vhost e.g.: <port number> : <queue-id>:<core-id>
vpath	User path for openvswitch files Default is `/usr/local`
max_idle	The maximum time that idle flows will remain cached in the datapath, ms

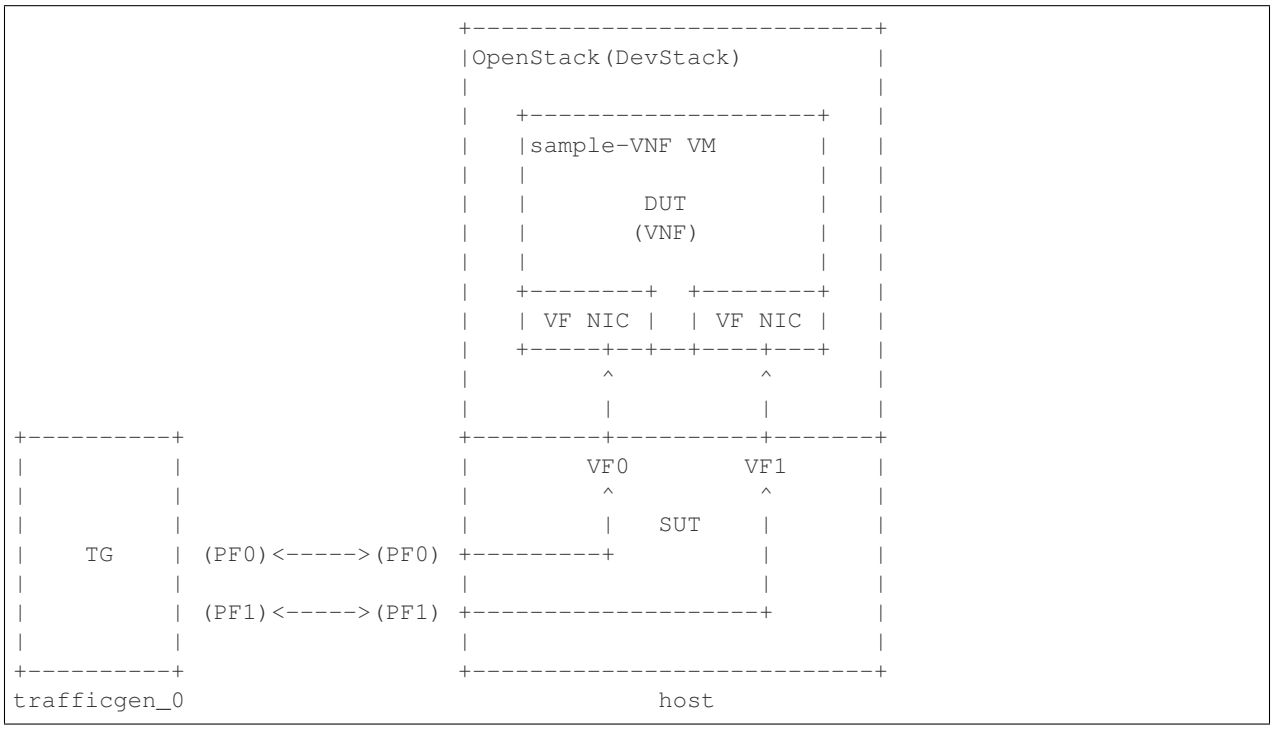
VM image properties

VM image properties are same as for SRIOV *VM image properties*.

2.12.9 OpenStack with SR-IOV support

This section describes how to run a Sample VNF test case, using Heat context, with SR-IOV. It also covers how to install OpenStack in Ubuntu 16.04, using DevStack, with SR-IOV support.

Single node OpenStack with external TG



Host pre-configuration

Warning: The following configuration requires sudo access to the system. Make sure that your user have the access.

Enable the Intel VT-d or AMD-Vi extension in the BIOS. Some system manufacturers disable this extension by default.

Activate the Intel VT-d or AMD-Vi extension in the kernel by modifying the GRUB config file `/etc/default/grub`.

For the Intel platform:

```
...
GRUB_CMDLINE_LINUX_DEFAULT="intel_iommu=on"
...
```

For the AMD platform:

```
...
GRUB_CMDLINE_LINUX_DEFAULT="amd_iommu=on"
...
```

Update the grub configuration file and restart the system:

Warning: The following command will reboot the system.

```
sudo update-grub
sudo reboot
```

Make sure the extension has been enabled:

```
sudo journalctl -b 0 | grep -e IOMMU -e DMAR

Feb 06 14:50:14 hostname kernel: ACPI: DMAR 0x000000006C406000 0001E0 (v01 INTEL
↪S2600WF  00000001 INTL 20091013)
Feb 06 14:50:14 hostname kernel: DMAR: IOMMU enabled
Feb 06 14:50:14 hostname kernel: DMAR: Host address width 46
Feb 06 14:50:14 hostname kernel: DMAR: DRHD base: 0x000000d37fc000 flags: 0x0
Feb 06 14:50:14 hostname kernel: DMAR: dmar0: reg_base_addr d37fc000 ver 1:0 cap
↪8d2078c106f0466 ecap f020de
Feb 06 14:50:14 hostname kernel: DMAR: DRHD base: 0x000000e0ffc000 flags: 0x0
Feb 06 14:50:14 hostname kernel: DMAR: dmar1: reg_base_addr e0ffc000 ver 1:0 cap
↪8d2078c106f0466 ecap f020de
Feb 06 14:50:14 hostname kernel: DMAR: DRHD base: 0x000000ee7fc000 flags: 0x0
```

Setup system proxy (if needed). Add the following configuration into the `/etc/environment` file:

Note: The proxy server name/port and IPs should be changed according to actual/current proxy configuration in the lab.

```
export http_proxy=http://proxy.company.com:port
export https_proxy=http://proxy.company.com:port
export ftp_proxy=http://proxy.company.com:port
export no_proxy=localhost,127.0.0.1,company.com,<IP-OF-HOST1>,<IP-OF-HOST2>,...
export NO_PROXY=localhost,127.0.0.1,company.com,<IP-OF-HOST1>,<IP-OF-HOST2>,...
```

Upgrade the system:

```
sudo -EH apt-get update
sudo -EH apt-get upgrade
sudo -EH apt-get dist-upgrade
```

Install dependencies needed for DevStack

```
sudo -EH apt-get install python python-dev python-pip
```

Setup SR-IOV ports on the host:

Note: The `enp24s0f0`, `enp24s0f1` are physical function (PF) interfaces on a host and `enp24s0f3` is a public interface used in OpenStack, so the interface names should be changed according to the HW environment used for

testing.

```
sudo ip link set dev enp24s0f0 up
sudo ip link set dev enp24s0f1 up
sudo ip link set dev enp24s0f3 up

# Create VFs on PF
echo 2 | sudo tee /sys/class/net/enp24s0f0/device/sriov_numvfs
echo 2 | sudo tee /sys/class/net/enp24s0f1/device/sriov_numvfs
```

DevStack installation

If you want to try out NSB, but don't have OpenStack set-up, you can use Devstack to install OpenStack on a host. Please note, that the `stable/pike` branch of devstack repo should be used during the installation. The required `local.conf` configuration file is described below.

DevStack configuration file:

Note: Update the devstack configuration file by replacing angluar brackets with a short description inside.

Note: Use `lspci | grep Ether` & `lspci -n | grep <PCI ADDRESS>` commands to get device and vendor id of the virtual function (VF).

```
[[local|localrc]]
HOST_IP=<HOST_IP_ADDRESS>
ADMIN_PASSWORD=password
MYSQL_PASSWORD=$ADMIN_PASSWORD
DATABASE_PASSWORD=$ADMIN_PASSWORD
RABBIT_PASSWORD=$ADMIN_PASSWORD
SERVICE_PASSWORD=$ADMIN_PASSWORD
HORIZON_PASSWORD=$ADMIN_PASSWORD

# Internet access.
RECLONE=False
PIP_UPGRADE=True
IP_VERSION=4

# Services
disable_service n-net
ENABLED_SERVICES+=,q-svc,q-dhcp,q-meta,q-agt,q-sriov-agt

# Heat
enable_plugin heat https://git.openstack.org/openstack/heat stable/pike

# Neutron
enable_plugin neutron https://git.openstack.org/openstack/neutron.git stable/pike

# Neutron Options
FLOATING_RANGE=<RANGE_IN_THE_PUBLIC_INTERFACE_NETWORK>
Q_FLOATING_ALLOCATION_POOL=start=<START_IP_ADDRESS>,end=<END_IP_ADDRESS>
PUBLIC_NETWORK_GATEWAY=<PUBLIC_NETWORK_GATEWAY>
PUBLIC_INTERFACE=<PUBLIC INTERFACE>
```

(continues on next page)

(continued from previous page)

```
# ML2 Configuration
Q_PLUGIN=ml2
Q_ML2_PLUGIN_MECHANISM_DRIVERS=openvswitch,sriovnicswitch
Q_ML2_PLUGIN_TYPE_DRIVERS=vlan,flat,local,vxlan,gre,geneve

# Open vSwitch provider networking configuration
Q_USE_PROVIDERNET_FOR_PUBLIC=True
OVS_PHYSICAL_BRIDGE=br-ex
OVS_BRIDGE_MAPPINGS=public:br-ex
PHYSICAL_DEVICE_MAPPINGS=physnet1:<PF0_IFNAME>,physnet2:<PF1_IFNAME>
PHYSICAL_NETWORK=physnet1,physnet2


[[post-config|$NOVA_CONF]]
[DEFAULT]
scheduler_default_filters=RamFilter,ComputeFilter,AvailabilityZoneFilter,
↪ComputeCapabilitiesFilter,ImagePropertiesFilter,PciPassthroughFilter
# Whitelist PCI devices
pci_passthrough_whitelist = {\\"devname\\": \\"<PF0_IFNAME>\\", \\"physical_network\\
↪": \\"physnet1\\" }
pci_passthrough_whitelist = {\\"devname\\": \\"<PF1_IFNAME>\\", \\"physical_network\\
↪": \\"physnet2\\" }

[filter_scheduler]
enabled_filters = RetryFilter,AvailabilityZoneFilter,RamFilter,DiskFilter,
↪ComputeFilter,ComputeCapabilitiesFilter,ImagePropertiesFilter,
↪ServerGroupAntiAffinityFilter,ServerGroupAffinityFilter,SameHostFilter

[libvirt]
cpu_mode = host-model


# ML2 plugin bits for SR-IOV enablement of Intel Corporation XL710/X710 Virtual␣
↪Function
[[post-config|/$Q_PLUGIN_CONF_FILE]]
[ml2_sriov]
agent_required = True
supported_pci_vendor_devs = <VF_DEV_ID:VF_VEN_ID>
```

Start the devstack installation on a host.

TG host configuration

Yardstick automatically installs and configures Trex traffic generator on TG host based on provided POD file (see below). Anyway, it's recommended to check the compatibility of the installed NIC on the TG server with software Trex using the manual.

Run the Sample VNF test case

There is an example of Sample VNF test case ready to be executed in an OpenStack environment with SR-IOV support: `samples/vnf_samples/nsut/vfw/tc_heat_sriov_external_rfc2544_ipv4_1rule_1flow_trex.yaml`.

Install Yardstick using *Install Yardstick (NSB Testing)* steps for OpenStack context.

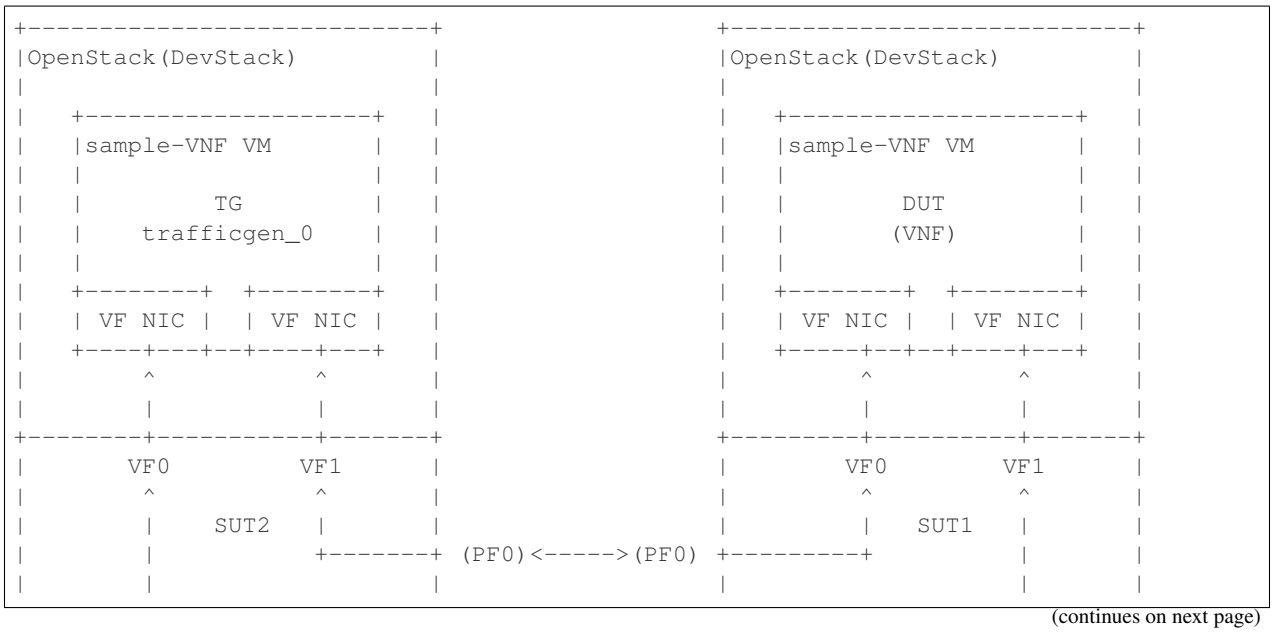
Create pod file for TG in the yardstick repo folder located in the yardstick container:

Note: The `ip`, `user`, `password` and `vpci` fields show be changed according to HW environment used for the testing. Use `lshw -c network -businfo` command to get the PF PCI address for `vpci` field.

```
nodes:
-
    name: trafficgen_1
    role: tg__0
    ip: <TG-HOST-IP>
    user: <TG-USER>
    password: <TG-PASS>
    interfaces:
        xe0:  # logical name from topology.yaml and vnfd.yaml
            vpci:      "0000:18:00.0"
            driver:    i40e # default kernel driver
            dpdk_port_num: 0
            local_ip: "10.1.1.150"
            netmask:   "255.255.255.0"
            local_mac: "00:00:00:00:00:01"
        xe1:  # logical name from topology.yaml and vnfd.yaml
            vpci:      "0000:18:00.1"
            driver:    i40e # default kernel driver
            dpdk_port_num: 1
            local_ip: "10.1.1.151"
            netmask:   "255.255.255.0"
            local_mac: "00:00:00:00:00:02"
```

Run the Sample vFW RFC2544 SR-IOV TC (`samples/vnf_samples/nsut/vfw/tc_heat_sriov_external_rfc2544_ipv4_1rule_1flow_64B_trex.yaml`) in the heat context using steps described in *NS testing - using yardstick CLI* section.

Multi node OpenStack TG and VNF setup (two nodes)



(continues on next page)

(continued from previous page)

```
|         +--------------------+ (PF1)<----->(PF1) +---------------------+         |
|                              |                   |                               |
+------------------------------+                   +-------------------------------+
          host2 (compute)                                 host1 (controller)
```

Controller/Compute pre-configuration

Pre-configuration of the controller and compute hosts are the same as described in *Host pre-configuration* section.

DevStack configuration

A reference `local.conf` for deploying OpenStack in a multi-host environment using Devstack is shown in this section. The `stable/pike` branch of devstack repo should be used during the installation.

Note: Update the devstack configuration files by replacing angluar brackets with a short description inside.

Note: Use `lspci | grep Ether` & `lspci -n | grep <PCI ADDRESS>` commands to get device and vendor id of the virtual function (VF).

DevStack configuration file for controller host:

```
[[local|localrc]]
HOST_IP=<HOST_IP_ADDRESS>
ADMIN_PASSWORD=password
MYSQL_PASSWORD=$ADMIN_PASSWORD
DATABASE_PASSWORD=$ADMIN_PASSWORD
RABBIT_PASSWORD=$ADMIN_PASSWORD
SERVICE_PASSWORD=$ADMIN_PASSWORD
HORIZON_PASSWORD=$ADMIN_PASSWORD
# Controller node
SERVICE_HOST=$HOST_IP
MYSQL_HOST=$SERVICE_HOST
RABBIT_HOST=$SERVICE_HOST
GLANCE_HOSTPORT=$SERVICE_HOST:9292

# Internet access.
RECLONE=False
PIP_UPGRADE=True
IP_VERSION=4

# Services
disable_service n-net
ENABLED_SERVICES+=,q-svc,q-dhcp,q-meta,q-agt,q-sriov-agt

# Heat
enable_plugin heat https://git.openstack.org/openstack/heat stable/pike

# Neutron
enable_plugin neutron https://git.openstack.org/openstack/neutron.git stable/pike
```

(continues on next page)

(continued from previous page)

```
# Neutron Options
FLOATING_RANGE=<RANGE_IN_THE_PUBLIC_INTERFACE_NETWORK>
Q_FLOATING_ALLOCATION_POOL=start=<START_IP_ADDRESS>,end=<END_IP_ADDRESS>
PUBLIC_NETWORK_GATEWAY=<PUBLIC_NETWORK_GATEWAY>
PUBLIC_INTERFACE=<PUBLIC INTERFACE>

# ML2 Configuration
Q_PLUGIN=ml2
Q_ML2_PLUGIN_MECHANISM_DRIVERS=openvswitch,sriovnicswitch
Q_ML2_PLUGIN_TYPE_DRIVERS=vlan,flat,local,vxlan,gre,geneve

# Open vSwitch provider networking configuration
Q_USE_PROVIDERNET_FOR_PUBLIC=True
OVS_PHYSICAL_BRIDGE=br-ex
OVS_BRIDGE_MAPPINGS=public:br-ex
PHYSICAL_DEVICE_MAPPINGS=physnet1:<PF0_IFNAME>,physnet2:<PF1_IFNAME>
PHYSICAL_NETWORK=physnet1,physnet2


[[post-config|$NOVA_CONF]]
[DEFAULT]
scheduler_default_filters=RamFilter,ComputeFilter,AvailabilityZoneFilter,
↪ComputeCapabilitiesFilter,ImagePropertiesFilter,PciPassthroughFilter
# Whitelist PCI devices
pci_passthrough_whitelist = {\\"devname\\": \\"<PF0_IFNAME>\\", \\"physical_network\\
↪": \\"physnet1\\" }
pci_passthrough_whitelist = {\\"devname\\": \\"<PF1_IFNAME>\\", \\"physical_network\\
↪": \\"physnet2\\" }

[libvirt]
cpu_mode = host-model


# ML2 plugin bits for SR-IOV enablement of Intel Corporation XL710/X710 Virtual␣
↪Function
[[post-config|/$Q_PLUGIN_CONF_FILE]]
[ml2_sriov]
agent_required = True
supported_pci_vendor_devs = <VF_DEV_ID:VF_VEN_ID>
```

DevStack configuration file for compute host:

```
[[local|localrc]]
HOST_IP=<HOST_IP_ADDRESS>
MYSQL_PASSWORD=password
DATABASE_PASSWORD=password
RABBIT_PASSWORD=password
ADMIN_PASSWORD=password
SERVICE_PASSWORD=password
HORIZON_PASSWORD=password
# Controller node
SERVICE_HOST=<CONTROLLER_IP_ADDRESS>
MYSQL_HOST=$SERVICE_HOST
RABBIT_HOST=$SERVICE_HOST
GLANCE_HOSTPORT=$SERVICE_HOST:9292

# Internet access.
```

(continues on next page)

(continued from previous page)

```
RECLONE=False
PIP_UPGRADE=True
IP_VERSION=4

# Neutron
enable_plugin neutron https://git.openstack.org/openstack/neutron.git stable/pike

# Services
ENABLED_SERVICES=n-cpu,rabbit,q-agt,placement-api,q-sriov-agt

# Neutron Options
PUBLIC_INTERFACE=<PUBLIC INTERFACE>

# ML2 Configuration
Q_PLUGIN=ml2
Q_ML2_PLUGIN_MECHANISM_DRIVERS=openvswitch,sriovnicswitch
Q_ML2_PLUGIN_TYPE_DRIVERS=vlan,flat,local,vxlan,gre,geneve

# Open vSwitch provider networking configuration
PHYSICAL_DEVICE_MAPPINGS=physnet1:<PF0_IFNAME>,physnet2:<PF1_IFNAME>


[[post-config|$NOVA_CONF]]
[DEFAULT]
scheduler_default_filters=RamFilter,ComputeFilter,AvailabilityZoneFilter,
↪ComputeCapabilitiesFilter,ImagePropertiesFilter,PciPassthroughFilter
# Whitelist PCI devices
pci_passthrough_whitelist = {\\"devname\\": \\"<PF0_IFNAME>\\", \\"physical_network\\
↪": \\"physnet1\\" }
pci_passthrough_whitelist = {\\"devname\\": \\"<PF1_IFNAME>\\", \\"physical_network\\
↪": \\"physnet2\\" }

[libvirt]
cpu_mode = host-model


# ML2 plugin bits for SR-IOV enablement of Intel Corporation XL710/X710 Virtual␣
↪Function
[[post-config|/$Q_PLUGIN_CONF_FILE]]
[ml2_sriov]
agent_required = True
supported_pci_vendor_devs = <VF_DEV_ID:VF_VEN_ID>
```

Start the devstack installation on the controller and compute hosts.

Run the sample vFW TC

Install Yardstick using *Install Yardstick (NSB Testing)* steps for OpenStack context.

Run the sample vFW RFC2544 SR-IOV test case (`samples/vnf_samples/nsut/vfw/tc_heat_rfc2544_ipv4_1rule_1flow_64B_trex.yaml`) in the heat context using steps described in *NS testing - using yardstick CLI* section and the following Yardstick command line arguments:

```
yardstick -d task start --task-args='{"provider": "sriov"}' \
samples/vnf_samples/nsut/vfw/tc_heat_rfc2544_ipv4_1rule_1flow_64B_trex.yaml
```

2.12.10 Enabling other Traffic generators

IxLoad

1. Software needed: IxLoadAPI `<IxLoadTclApi verson>Linux64.bin.tgz` and `<IxOS version>Linux64.bin.tar.gz` (Download from ixia support site) Install - `<IxLoadTclApi verson>Linux64.bin.tgz` and `<IxOS version>Linux64.bin.tar.gz` If the installation was not done inside the container, after installing the IXIA client, check `/opt/ixia/ixload/<ver>/bin/ixloadpython` and make sure you can run this cmd inside the yardstick container. Usually user is required to copy or link `/opt/ixia/python/<ver>/bin/ixiapython` to `/usr/bin/ixiapython<ver>` inside the container.

2. Update `pod_ixia.yaml` file with ixia details.

```
cp <repo>/etc/yardstick/nodes/pod.yaml.nsb.sample.ixia \
  etc/yardstick/nodes/pod_ixia.yaml
```

Config `pod_ixia.yaml`

```
nodes:
-
    name: trafficgen_1
    role: IxNet
    ip: 1.2.1.1 #ixia machine ip
    user: user
    password: r00t
    key_filename: /root/.ssh/id_rsa
    tg_config:
        ixchassis: "1.2.1.7" #ixia chassis ip
        tcl_port: "8009" # tcl server port
        lib_path: "/opt/ixia/ixos-api/8.01.0.2/lib/ixTcl1.0"
        root_dir: "/opt/ixia/ixos-api/8.01.0.2/"
        py_bin_path: "/opt/ixia/ixload/8.01.106.3/bin/"
        dut_result_dir: "/mnt/ixia"
        version: 8.1
    interfaces:
        xe0:  # logical name from topology.yaml and vnfd.yaml
            vpci: "2:5" # Card:port
            driver:    "none"
            dpdk_port_num: 0
            local_ip: "152.16.100.20"
            netmask:   "255.255.0.0"
            local_mac: "00:98:10:64:14:00"
        xe1:  # logical name from topology.yaml and vnfd.yaml
            vpci: "2:6" # [(Card, port)]
            driver:    "none"
            dpdk_port_num: 1
            local_ip: "152.40.40.20"
            netmask:   "255.255.0.0"
            local_mac: "00:98:28:28:14:00"
```

for sriov/ovs_dpdk pod files, please refer to *Standalone Virtualization* for ovs-dpdk/sriov configuration

3. Start IxOS TCL Server (Install 'Ixia IxExplorer IxOS <version>') You will also need to configure the IxLoad machine to start the IXIA IxosTclServer. This can be started like so:

    - Connect to the IxLoad machine using RDP

- Go to: `Start->Programs->Ixia->IxOS->IxOS 8.01-GA-Patch1->Ixia Tcl Server IxOS 8.01-GA-Patch1` or `C:\Program Files (x86)\Ixia\IxOS\8.01-GA-Patch1\ixTclServer.exe`

4. Create a folder `Results` in c:and share the folder on the network.
5. Execute testcase in samplevnf folder e.g. `<repo>/samples/vnf_samples/nsut/vfw/tc_baremetal_http_ixload_1b_Requests-65000_Concurrency.yaml`

IxNetwork

IxNetwork testcases use IxNetwork API Python Bindings module, which is installed as part of the requirements of the project.

1. Update `pod_ixia.yaml` file with ixia details.

```
cp <repo>/etc/yardstick/nodes/pod.yaml.nsb.sample.ixia \
etc/yardstick/nodes/pod_ixia.yaml
```

Configure `pod_ixia.yaml`

```
nodes:
-
    name: trafficgen_1
    role: IxNet
    ip: 1.2.1.1 #ixia machine ip
    user: user
    password: r00t
    key_filename: /root/.ssh/id_rsa
    tg_config:
        ixchassis: "1.2.1.7" #ixia chassis ip
        tcl_port: "8009" # tcl server port
        lib_path: "/opt/ixia/ixos-api/8.01.0.2/lib/ixTcl1.0"
        root_dir: "/opt/ixia/ixos-api/8.01.0.2/"
        py_bin_path: "/opt/ixia/ixload/8.01.106.3/bin/"
        dut_result_dir: "/mnt/ixia"
        version: 8.1
    interfaces:
        xe0:  # logical name from topology.yaml and vnfd.yaml
            vpci: "2:5" # Card:port
            driver:    "none"
            dpdk_port_num: 0
            local_ip: "152.16.100.20"
            netmask:   "255.255.0.0"
            local_mac: "00:98:10:64:14:00"
        xe1:  # logical name from topology.yaml and vnfd.yaml
            vpci: "2:6" # [(Card, port)]
            driver:    "none"
            dpdk_port_num: 1
            local_ip: "152.40.40.20"
            netmask:   "255.255.0.0"
            local_mac: "00:98:28:28:14:00"
```

for sriov/ovs_dpdk pod files, please refer to above *Standalone Virtualization* for ovs-dpdk/sriov configuration

2. Start IxNetwork TCL Server You will also need to configure the IxNetwork machine to start the IXIA IxNetworkTclServer. This can be started like so:

- Connect to the IxNetwork machine using RDP
- Go to: `Start->Programs->Ixia->IxNetwork->IxNetwork 7.21.893.14 GA->IxNetworkTclServer` (or `IxNetworkApiServer`)

3. Execute testcase in samplevnf folder e.g. `<repo>/samples/vnf_samples/nsut/vfw/tc_baremetal_rfc2544_ipv4_1rule_1flow_64B_ixia.yaml`

2.12.11 Spirent Landslide

In order to use Spirent Landslide for vEPC testcases, some dependencies have to be preinstalled and properly configured.

- Java

  32-bit Java installation is required for the Spirent Landslide TCL API.

  ```
  $ sudo apt-get install openjdk-8-jdk:i386
  ```

  Important: Make sure `LD_LIBRARY_PATH` is pointing to 32-bit JRE. For more details check *Linux Troubleshooting <http://TAS_HOST_IP/tclapiinstall.html#trouble>* section of installation instructions.

- LsApi (Tcl API module)

  Follow Landslide documentation for detailed instructions on Linux installation of Tcl API and its dependencies `http://TAS_HOST_IP/tclapiinstall.html`. For working with LsApi Python wrapper only steps 1-5 are required.

  Note: After installation make sure your API home path is included in `PYTHONPATH` environment variable.

  Important: The current version of LsApi module has an issue with reading LD_LIBRARY_PATH. For LsApi module to initialize correctly following lines (184-186) in lsapi.py

  ```
  ldpath = os.environ.get('LD_LIBRARY_PATH', '')
  if ldpath == '':
   environ['LD_LIBRARY_PATH'] = environ['LD_LIBRARY_PATH'] + ':' + ldpath
  ```

  should be changed to:

  ```
  ldpath = os.environ.get('LD_LIBRARY_PATH', '')
  if not ldpath == '':
        environ['LD_LIBRARY_PATH'] = environ['LD_LIBRARY_PATH'] + ':' +␣
  ↪ldpath
  ```

Note: The Spirent landslide TCL software package needs to be updated in case the user upgrades to a new version of Spirent landslide software.

2.13 Yardstick - NSB Testing - Operation

2.13.1 Abstract

NSB test configuration and OpenStack setup requirements

2.13.2 OpenStack Network Configuration

NSB requires certain OpenStack deployment configurations. For optimal VNF characterization using external traffic generators NSB requires provider/external networks.

Provider networks

The VNFs require a clear L2 connect to the external network in order to generate realistic traffic from multiple address ranges and ports.

In order to prevent Neutron from filtering traffic we have to disable Neutron Port Security. We also disable DHCP on the data ports because we are binding the ports to DPDK and do not need DHCP addresses. We also disable gateways because multiple default gateways can prevent SSH access to the VNF from the floating IP. We only want a gateway on the mgmt network

```
uplink_0:
  cidr: '10.1.0.0/24'
  gateway_ip: 'null'
  port_security_enabled: False
  enable_dhcp: 'false'
```

Heat Topologies

By default Heat will attach every node to every Neutron network that is created. For scale-out tests we do not want to attach every node to every network.

For each node you can specify which ports are on which network using the network_ports dictionary.

In this example we have `TRex xe0 <-> xe0 VNF xe1 <-> xe0 UDP_Replay`

```
vnf_0:
  floating_ip: true
  placement: "pgrp1"
  network_ports:
    mgmt:
      - mgmt
    uplink_0:
      - xe0
    downlink_0:
      - xe1
tg_0:
  floating_ip: true
  placement: "pgrp1"
  network_ports:
    mgmt:
      - mgmt
    uplink_0:
```

(continues on next page)

(continued from previous page)

```
      - xe0
    # Trex always needs two ports
    uplink_1:
      - xe1
tg_1:
  floating_ip: true
  placement: "pgrp1"
  network_ports:
    mgmt:
     - mgmt
    downlink_0:
     - xe0
```

Availability zone

The configuration of the availability zone is requred in cases where location of exact compute host/group of compute hosts needs to be specified for *SampleVNF* or traffic generator in the heat test case. If this is the case, please follow the instructions below.

1. Create a host aggregate in the OpenStack and add the available compute hosts into the aggregate group.

   Note: Change the `<AZ_NAME>` (availability zone name), `<AGG_NAME>` (host aggregate name) and `<HOST>` (host name of one of the compute) in the commands below.

   ```
   # create host aggregate
   openstack aggregate create --zone <AZ_NAME> \
     --property availability_zone=<AZ_NAME> <AGG_NAME>
   # show available hosts
   openstack compute service list --service nova-compute
   # add selected host into the host aggregate
   openstack aggregate add host <AGG_NAME> <HOST>
   ```

2. To specify the OpenStack location (the exact compute host or group of the hosts) of SampleVNF or traffic generator in the heat test case, the `availability_zone` server configuration option should be used. For example:

   Note: The `<AZ_NAME>` (availability zone name) should be changed according to the name used during the host aggregate creation steps above.

   ```
   context:
     name: yardstick
     image: yardstick-samplevnfs
     ...
     servers:
       vnf_0:
         ...
         availability_zone: <AZ_NAME>
         ...
       tg__0:
         ...
         availability_zone: <AZ_NAME>
   ```

(continues on next page)

(continued from previous page)

```
    ...
networks:
  ...
```

There are two example of SampleVNF scale out test case which use the `availability zone` feature to specify the exact location of scaled VNFs and traffic generators.

Those are:

```
<repo>/samples/vnf_samples/nsut/prox/tc_prox_heat_context_l2fwd_multiflow-2-scale-out.
↪yaml
<repo>/samples/vnf_samples/nsut/vfw/tc_heat_rfc2544_ipv4_1rule_1flow_64B_trex_scale_
↪out.yaml
```

Note: This section describes the PROX scale-out testcase, but the same procedure is used for the vFW test case.

1. Before running the scale-out test case, make sure the host aggregates are configured in the OpenStack environment. To check this, run the following command:

```
# show configured host aggregates (example)
openstack aggregate list
+----+------+-------------------+
| ID | Name | Availability Zone |
+----+------+-------------------+
|  4 | agg0 | AZ_NAME_0         |
|  5 | agg1 | AZ_NAME_1         |
+----+------+-------------------+
```

2. If no host aggregates are configured, please follow the instructions to *Create a host aggregate*
3. Run the SampleVNF PROX scale-out test case, specifying the `availability zone` of each VNF and traffic generator as task arguments.

   Note: The `az_0` and `az_1` should be changed according to the host aggregates created in the OpenStack.

```
yardstick -d task start \
<repo>/samples/vnf_samples/nsut/prox/tc_prox_heat_context_l2fwd_multiflow-2-scale-
↪out.yaml\
  --task-args='{
    "num_vnfs": 4, "availability_zone": {
      "vnf_0": "az_0", "tg_0": "az_1",
      "vnf_1": "az_0", "tg_1": "az_1",
      "vnf_2": "az_0", "tg_2": "az_1",
      "vnf_3": "az_0", "tg_3": "az_1"
    }
  }'
```

   `num_vnfs` specifies how many VNFs are going to be deployed in the `heat` contexts. `vnf_X` and `tg_X` arguments configure the availability zone where the VNF and traffic generator is going to be deployed.

2.13.3 Collectd KPIs

NSB can collect KPIs from collected. We have support for various plugins enabled by the *Barometer* project.

The default yardstick-samplevnf has collectd installed. This allows for collecting KPIs from the VNF.

Collecting KPIs from the NFVi is more complicated and requires manual setup. We assume that collectd is not installed on the compute nodes.

To collectd KPIs from the NFVi compute nodes:

- install_collectd on the compute nodes
- create pod.yaml for the compute nodes
- enable specific plugins depending on the vswitch and DPDK

example `pod.yaml` section for Compute node running collectd.

```
-
  name: "compute-1"
  role: Compute
  ip: "10.1.2.3"
  user: "root"
  ssh_port: "22"
  password: ""
  collectd:
    interval: 5
    plugins:
      # for libvirtd stats
      virt: {}
      intel_pmu: {}
      ovs_stats:
        # path to OVS socket
        ovs_socket_path: /var/run/openvswitch/db.sock
      intel_rdt: {}
```

2.13.4 Scale-Up

VNFs performance data with scale-up

- Helps to figure out optimal number of cores specification in the Virtual Machine template creation or VNF
- Helps in comparison between different VNF vendor offerings
- Better the scale-up index, indicates the performance scalability of a particular solution

Heat

For VNF scale-up tests we increase the number for VNF worker threads. In the case of VNFs we also need to increase the number of VCPUs and memory allocated to the VNF.

An example scale-up Heat testcase is:

```
# Copyright (c) 2016-2019 Intel Corporation
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
```

(continues on next page)

(continued from previous page)

```
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
{% set framesize = framesize or "{64B: 100}" %}
{% set mem = mem or 20480 %}
{% set vcpus = vcpus or 10 %}
{% set vports = vports or 2 %}
---
schema: yardstick:task:0.1
scenarios:
- type: NSPerf
  traffic_profile: ../../traffic_profiles/ipv4_throughput-scale-up.yaml
  extra_args:
    vports: {{ vports }}
  topology: vfw-tg-topology-scale-up.yaml
  nodes:
    tg__0: trafficgen_0.yardstick
    vnf__0: vnf_0.yardstick
  options:
    framesize:
      uplink: {{ framesize }}
      downlink: {{ framesize }}
    flow:
      src_ip: [
{% for vport in range(0,vports,2|int) %}
       {'tg__0': 'xe{{vport}}'},
{% endfor %}  ]
      dst_ip: [
{% for vport in range(1,vports,2|int) %}
      {'tg__0': 'xe{{vport}}'},
{% endfor %}  ]
      count: 1
    traffic_type: 4
    rfc2544:
      allowed_drop_rate: 0.0001 - 0.0001
    vnf__0:
      rules: acl_1rule.yaml
      vnf_config: {lb_config: 'SW', file: vfw_vnf_pipeline_cores_{{vcpus}}_ports_{
↪{vports}}_lb_1_sw.conf }
  runner:
    type: Iteration
    iterations: 10
    interval: 35
context:
  # put node context first, so we don't HEAT deploy if node has errors
  name: yardstick
  image: yardstick-samplevnfs
  flavor:
    vcpus: {{ vcpus }}
    ram: {{ mem }}
    disk: 6
    extra_specs:
      hw:cpu_sockets: 1
      hw:cpu_cores: {{ vcpus }}
      hw:cpu_threads: 1
  user: ubuntu
```

(continues on next page)

(continued from previous page)

```
  placement_groups:
    pgrp1:
      policy: "availability"
  servers:
    trafficgen_0:
      floating_ip: true
      placement: "pgrp1"
    vnf_0:
      floating_ip: true
      placement: "pgrp1"
  networks:
    mgmt:
      cidr: '10.0.1.0/24'
{% for vport in range(1,vports,2|int) %}
    uplink_{{loop.index0}}:
      cidr: '10.1.{{vport}}.0/24'
      gateway_ip: 'null'
      port_security_enabled: False
      enable_dhcp: 'false'
    downlink_{{loop.index0}}:
      cidr: '10.1.{{vport+1}}.0/24'
      gateway_ip: 'null'
      port_security_enabled: False
      enable_dhcp: 'false'
{% endfor %}
```

This testcase template requires specifying the number of VCPUs, Memory and Ports. We set the VCPUs and memory using the `--task-args` options

```
yardstick task start --task-args='{"mem": 10480, "vcpus": 4, "vports": 2}' \
samples/vnf_samples/nsut/vfw/tc_heat_rfc2544_ipv4_1rule_1flow_64B_trex_scale-up.yaml
```

In order to support ports scale-up, traffic and topology templates need to be used in testcase.

A example topology template is:

```
# Copyright (c) 2016-2018 Intel Corporation
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
---
{% set vports = get(extra_args, 'vports', '2') %}
nsd:nsd-catalog:
    nsd:
    -   id: 3tg-topology
        name: 3tg-topology
        short-name: 3tg-topology
        description: 3tg-topology
```

(continues on next page)

(continued from previous page)

```
        constituent-vnfd:
        -   member-vnf-index: '1'
            vnfd-id-ref: tg__0
            VNF model: ../../vnf_descriptors/tg_rfc2544_tpl.yaml      #VNF type
        -   member-vnf-index: '2'
            vnfd-id-ref: vnf__0
            VNF model: ../../vnf_descriptors/vfw_vnf.yaml      #VNF type

        vld:
{% for vport in range(0,vports,2|int) %}
        -   id: uplink_{{loop.index0}}
            name: tg__0 to vnf__0 link {{vport + 1}}
            type: ELAN
            vnfd-connection-point-ref:
            -   member-vnf-index-ref: '1'
                vnfd-connection-point-ref: xe{{vport}}
                vnfd-id-ref: tg__0
            -   member-vnf-index-ref: '2'
                vnfd-connection-point-ref: xe{{vport}}
                vnfd-id-ref: vnf__0
        -   id: downlink_{{loop.index0}}
            name: vnf__0 to tg__0 link {{vport + 2}}
            type: ELAN
            vnfd-connection-point-ref:
            -   member-vnf-index-ref: '2'
                vnfd-connection-point-ref: xe{{vport+1}}
                vnfd-id-ref: vnf__0
            -   member-vnf-index-ref: '1'
                vnfd-connection-point-ref: xe{{vport+1}}
                vnfd-id-ref: tg__0
{% endfor %}
```

This template has `vports` as an argument. To pass this argument it needs to be configured in `extra_args` scenario definition. Please note that more argument can be defined in that section. All of them will be passed to topology and traffic profile templates

For example:

```
schema: yardstick:task:0.1
scenarios:
- type: NSPerf
  traffic_profile: ../../traffic_profiles/ipv4_throughput-scale-up.yaml
  extra_args:
    vports: {{ vports }}
  topology: vfw-tg-topology-scale-up.yaml
```

A example traffic profile template is:

```
# Copyright (c) 2016-2019 Intel Corporation
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
```

(continues on next page)

(continued from previous page)

```
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# flow definition for ACL tests - 1K flows - ipv4 only
#
# the number of flows defines the widest range of parameters
# for example if srcip_range=1.0.0.1-1.0.0.255 and dst_ip_range=10.0.0.1-10.0.1.255
# and it should define only 16 flows
#
# there is assumption that packets generated will have a random sequences of␣
↪following addresses pairs
# in the packets
# 1. src=1.x.x.x(x.x.x =random from 1..255) dst=10.x.x.x (random from 1..512)
# 2. src=1.x.x.x(x.x.x =random from 1..255) dst=10.x.x.x (random from 1..512)
# ...
# 512. src=1.x.x.x(x.x.x =random from 1..255) dst=10.x.x.x (random from 1..512)
#
# not all combination should be filled
# Any other field with random range will be added to flow definition
#
# the example.yaml provides all possibilities for traffic generation
#
# the profile defines a public and private side to make limited traffic correlation
# between private and public side same way as it is made by IXIA solution.
#
{% set vports = get(extra_args, 'vports', 2) %}
---
schema: "nsb:traffic_profile:0.1"

# This file is a template, it will be filled with values from tc.yaml before passing␣
↪to the traffic generator

name: rfc2544
description: Traffic profile to run RFC2544 latency
traffic_profile:
  traffic_type: RFC2544Profile # defines traffic behavior - constant or look for␣
↪highest possible throughput
  frame_rate: 100  # pc of linerate
  duration: {{ duration }}

{% for vport in range((vports / 2)|int) %}
uplink_{{vport}}:
  ipv4:
    id: {{ (vport * 2) + 1 }}
    outer_l2:
      framesize:
        64B: "{{ get(imix, 'imix.uplink.64B', '0') }}"
        128B: "{{ get(imix, 'imix.uplink.128B', '0') }}"
        256B: "{{ get(imix, 'imix.uplink.256B', '0') }}"
        373b: "{{ get(imix, 'imix.uplink.373B', '0') }}"
        512B: "{{ get(imix, 'imix.uplink.512B', '0') }}"
        570B: "{{ get(imix, 'imix.uplink.570B', '0') }}"
        1024B: "{{get(imix, 'imix.uplink.1024B', '0') }}"
        1400B: "{{ get(imix, 'imix.uplink.1400B', '0') }}"
        1500B: "{{ get(imix, 'imix.uplink.1500B', '0') }}"
```

(continues on next page)

(continued from previous page)

```
        1518B: "{{ get(imix, 'imix.uplink.1518B', '0') }}"
    outer_l3v4:
      proto: "udp"
      srcip4: {{ get(flow, 'flow.src_ip_%s'| format(vport), '1.%s.1.1-1.%s.255.255'|␣
↪format(vport, vport)) }}
      dstip4: {{ get(flow, 'flow.dst_ip_%s'| format(vport), '90.%s.1.1-90.%s.255.255
↪'| format(vport, vport)) }}
      count: {{ get(flow, 'flow.count', '1') }}
      ttl: 32
      dscp: 0
    outer_l4:
      srcport: {{ get(flow, 'flow.src_port_%s'| format(vport), '1234-4321') }}
      dstport: {{ get(flow, 'flow.dst_port_%s'| format(vport), '2001-4001') }}
      count: {{ get(flow, 'flow.count', '1') }}
downlink_{{vport}}:
  ipv4:
    id: {{ (vport * 2) + 2}}
    outer_l2:
      framesize:
        64B: "{{ get(imix, 'imix.downlink.64B', '0') }}"
        128B: "{{ get(imix, 'imix.downlink.128B', '0') }}"
        256B: "{{ get(imix, 'imix.downlink.256B', '0') }}"
        373b: "{{ get(imix, 'imix.downlink.373B', '0') }}"
        512B: "{{ get(imix, 'imix.downlink.512B', '0') }}"
        570B: "{{ get(imix, 'imix.downlink.570B', '0') }}"
        1024B: "{{get(imix, 'imix.downlink.1024B', '0') }}"
        1400B: "{{ get(imix, 'imix.downlink.1400B', '0') }}"
        1500B: "{{ get(imix, 'imix.downlink.1500B', '0') }}"
        1518B: "{{ get(imix, 'imix.downlink.1518B', '0') }}"

    outer_l3v4:
      proto: "udp"
      srcip4: {{ get(flow, 'flow.dst_ip_%s'| format(vport), '90.%s.1.1-90.%s.255.255
↪'| format(vport, vport)) }}
      dstip4: {{ get(flow, 'flow.src_ip_%s'| format(vport), '1.%s.1.1-1.%s.255.255'|␣
↪format(vport, vport)) }}
      count: {{ get(flow, 'flow.count', '1') }}
      ttl: 32
      dscp: 0
    outer_l4:
      srcport: {{ get(flow, 'flow.dst_port_%s'| format(vport), '1234-4321') }}
      dstport: {{ get(flow, 'flow.src_port_%s'| format(vport), '2001-4001') }}
      count: {{ get(flow, 'flow.count', '1') }}
{% endfor %}
```

There is an option to provide predefined config for SampleVNFs. Path to config file may by specified in `vnf_config` scenario section.

```
vnf__0:
   rules: acl_1rule.yaml
   vnf_config: {lb_config: 'SW', file: vfw_vnf_pipeline_cores_4_ports_2_lb_1_sw.conf }
```

Baremetal

1. Follow above traffic generator section to setup.

2. Edit num of threads in `<repo>/samples/vnf_samples/nsut/vfw/tc_baremetal_rfc2544_ipv4_trex_scale_up.yaml` e.g, 6 Threads for given VNF

```
schema: yardstick:task:0.1
scenarios:
{% for worker_thread in [1, 2 ,3 , 4, 5, 6] %}
- type: NSPerf
  traffic_profile: ../../traffic_profiles/ipv4_throughput.yaml
  topology: vfw-tg-topology.yaml
  nodes:
    tg__0: trafficgen_0.yardstick
    vnf__0: vnf_0.yardstick
  options:
    framesize:
      uplink: {64B: 100}
      downlink: {64B: 100}
    flow:
      src_ip: [{'tg__0': 'xe0'}]
      dst_ip: [{'tg__0': 'xe1'}]
      count: 1
    traffic_type: 4
    rfc2544:
      allowed_drop_rate: 0.0001 - 0.0001
    vnf__0:
      rules: acl_1rule.yaml
      vnf_config: {lb_config: 'HW', lb_count: 1, worker_config: '1C/1T', worker_
↪threads: {{worker_thread}}}
      nfvi_enable: True
  runner:
    type: Iteration
    iterations: 10
    interval: 35
{% endfor %}
context:
  type: Node
  name: yardstick
  nfvi_type: baremetal
  file: /etc/yardstick/nodes/pod.yaml
```

2.13.5 Scale-Out

VNFs performance data with scale-out helps

- capacity planning to meet the given network node requirements
- comparison between different VNF vendor offerings
- better the scale-out index, provides the flexibility in meeting future capacity requirements

Standalone

Scale-out not supported on Baremetal.

1. Follow above traffic generator section to setup.
2. Generate testcase for standalone virtualization using ansible scripts

```
cd <repo>/ansible
trex: standalone_ovs_scale_out_test.yaml or standalone_sriov_scale_out_test.
↪yaml
ixia: standalone_ovs_scale_out_ixia_test.yaml or standalone_sriov_scale_out_
↪ixia_test.yaml
ixia_correlated: standalone_ovs_scale_out_ixia_correlated_test.yaml or␣
↪standalone_sriov_scale_out_ixia_correlated_test.yaml
```

update the ovs_dpdk or sriov above Ansible scripts reflect the setup

3. run the test

```
<repo>/samples/vnf_samples/nsut/tc_sriov_vfw_udp_ixia_correlated_scale_out-1.
↪yaml
<repo>/samples/vnf_samples/nsut/tc_sriov_vfw_udp_ixia_correlated_scale_out-2.
↪yaml
```

Heat

There are sample scale-out all-VM Heat tests. These tests only use VMs and don't use external traffic.

The tests use UDP_Replay and correlated traffic.

```
<repo>/samples/vnf_samples/nsut/cgnapt/tc_heat_rfc2544_ipv4_1flow_64B_trex_correlated_
↪scale_4.yaml
```

To run the test you need to increase OpenStack CPU, Memory and Port quotas.

2.13.6 Traffic Generator tuning

The TRex traffic generator can be setup to use multiple threads per core, this is for multiqueue testing.

TRex does not automatically enable multiple threads because we currently cannot detect the number of queues on a device.

To enable multiple queue set the `queues_per_port` value in the TG VNF options section.

```
scenarios:
  - type: NSPerf
    nodes:
      tg__0: trafficgen_0.yardstick

    options:
      tg_0:
        queues_per_port: 2
```

2.13.7 Standalone configuration

NSB supports certain Standalone deployment configurations. Standalone supports provisioning a VM in a standalone visualised environment using kvm/qemu. There two types of Standalone contexts available: OVS-DPDK and SRIOV. OVS-DPDK uses OVS network with DPDK drivers. SRIOV enables network traffic to bypass the software switch layer of the Hyper-V stack.

Emulated machine type

For better performance test results of emulated VM spawned by Yardstick SA context (OvS-DPDK/SRIOV), it may be important to control the emulated machine type used by QEMU emulator. This attribute can be configured via TC definition in `contexts` section under `extra_specs` configuration.

For example:

```
contexts:
   ...
   - type: StandaloneSriov
     ...
     flavor:
       ...
       extra_specs:
         ...
         machine_type: pc-i440fx-bionic
```

Where, `machine_type` can be set to one of the emulated machine type supported by QEMU running on SUT platform. To get full list of supported emulated machine types, the following command can be used on the target SUT host.

```
# qemu-system-x86_64 -machine ?
```

By default, the `machine_type` option is set to `pc-i440fx-xenial` which is suitable for running Ubuntu 16.04 VM image. So, if this type is not supported by the target platform or another VM image is used for stand alone (SA) context VM (e.g.: `bionic` image for Ubuntu 18.04), this configuration should be changed accordingly.

Standalone with OVS-DPDK

SampleVNF image is spawned in a VM on a baremetal server. OVS with DPDK is installed on the baremetal server.

Note: Ubuntu 17.10 requires DPDK v.17.05 and higher, DPDK v.17.05 requires OVS v.2.8.0.

Default values for OVS-DPDK:

- queues: 4
- lcore_mask: “”
- pmd_cpu_mask: “0x6”

Sample test case file

1. Prepare SampleVNF image and copy it to `flavor/images`.
2. Prepare context files for TREX and SampleVNF under `contexts/file`.
3. Add bridge named `br-int` to the baremetal where SampleVNF image is deployed.
4. Modify `networks/phy_port` accordingly to the baremetal setup.
5. Run test from:

```
# Copyright (c) 2016-2019 Intel Corporation
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
{% set framesize = framesize or "{64B: 100}" %}
---
schema: yardstick:task:0.1
scenarios:
- type: NSPerf
  traffic_profile: ../../traffic_profiles/ipv4_throughput.yaml
  topology: acl-tg-topology.yaml
  nodes:
    tg__0: trafficgen_0.yardstick
    vnf__0: vnf_0.yardstick
  options:
    framesize:
      uplink: {{ framesize }}
      downlink: {{ framesize }}
    flow:
      src_ip: [{'tg__0': 'xe0'}]
      dst_ip: [{'tg__0': 'xe1'}]
      count: 1
    traffic_type: 4
    rfc2544:
      allowed_drop_rate: 0.0001 - 0.0001
    vnf__0:
      rules: acl_1rule.yaml
      vnf_config: {lb_config: 'SW', lb_count: 1, worker_config: '1C/1T', worker_
↪threads: 1}
  runner:
    type: Iteration
    iterations: 10
    interval: 35
contexts:
   - name: yardstick
     type: Node
     file: /etc/yardstick/nodes/standalone/trex_bm.yaml
   - type: StandaloneOvsDpdk
     name: yardstick
     file: /etc/yardstick/nodes/standalone/host_ovs.yaml
     vm_deploy: True
     ovs_properties:
       version:
         ovs: 2.7.0
         dpdk: 16.11.1
       pmd_threads: 2
       ram:
         socket_0: 2048
```

(continues on next page)

(continued from previous page)

```
        socket_1: 2048
      queues: 4
      lcore_mask: ""
      pmd_cpu_mask: "0x6"
      vpath: "/usr/local"

    flavor:
      images: "/var/lib/libvirt/images/yardstick-nsb-image.img"
      ram: 16384
      extra_specs:
        hw:cpu_sockets: 1
        hw:cpu_cores: 6
        hw:cpu_threads: 2
      user: ""
      password: ""
    servers:
      vnf_0:
        network_ports:
          mgmt:
            cidr: '1.1.1.7/24'
          xe0:
            - uplink_0
          xe1:
            - downlink_0
    networks:
      uplink_0:
        port_num: 0
        phy_port: "0000:05:00.0"
        vpci: "0000:00:07.0"
        cidr: '152.16.100.10/24'
        gateway_ip: '152.16.100.20'
      downlink_0:
        port_num: 1
        phy_port: "0000:05:00.1"
        vpci: "0000:00:08.0"
        cidr: '152.16.40.10/24'
        gateway_ip: '152.16.100.20'
```

2.13.8 Preparing test run of vEPC test case

Provided vEPC test cases are examples of emulation of vEPC infrastructure components, such as UE, eNodeB, MME, SGW, PGW.

Location of vEPC test cases: `samples/vnf_samples/nsut/vepc/`.

Before running a specific vEPC test case using NSB, some preconfiguration needs to be done.

2.14 Update Spirent Landslide TG configuration in pod file

Examples of `pod.yaml` files could be found in `etc/yardstick/nodes/standalone`. The name of related pod file could be checked in the context section of NSB test case.

The `pod.yaml` related to vEPC test case uses some sub-structures that hold the details of accessing the Spirent Landslide traffic generator. These subsections and the changes to be done in provided example pod file are described

below.

1. `tas_manager`: data under this key holds the information required to access Landslide TAS (Test Administration Server) and perform needed configurations on it.

- `ip`: IP address of TAS Manager node; should be updated according to test setup used
- `super_user`: superuser name; could be retrieved from Landslide documentation
- `super_user_password`: superuser password; could be retrieved from Landslide documentation
- `cfguser_password`: password of predefined user named 'cfguser'; default password could be retrieved from Landslide documentation
- `test_user`: username to be used during test run as a Landslide library name; to be defined by test run operator
- `test_user_password`: password of test user; to be defined by test run operator
- `proto`: *http* or *https*; to be defined by test run operator
- `license`: Landslide license number installed on TAS

2. The `config` section holds information about test servers (TSs) and systems under test (SUTs). Data is represented as a list of entries. Each such entry contains:

- `test_server`: this subsection represents data related to test server configuration, such as:
  - `name`: test server name; unique custom name to be defined by test operator
  - `role`: this value is used as a key to bind specific Test Server and TestCase; should be set to one of test types supported by TAS license
  - `ip`: Test Server IP address
  - `thread_model`: parameter related to Test Server performance mode. The value should be one of the following: "Legacy" | "Max" | "Fireball". Refer to Landslide documentation for details.
  - `phySubnets`: a structure used to specify IP ranges reservations on specific network interfaces of related Test Server. Structure fields are:
  - `base`: start of IP address range
  - `mask`: IP range mask in CIDR format
  - `name`: network interface name, e.g. *eth1*
  - `numIps`: size of IP address range
- `preResolvedArpAddress`: a structure used to specify the range of IP addresses for which the ARP responses will be emulated
  - `StartingAddress`: IP address specifying the start of IP address range
  - `NumNodes`: size of the IP address range
- `suts`: a structure that contains definitions of each specific SUT (represents a vEPC component). SUT structure contains following key/value pairs:
  - `name`: unique custom string specifying SUT name
  - `role`: string value corresponding with an SUT role specified in the session profile (test session template) file
  - `managementIp`: SUT management IP adress
  - `phy`: network interface name, e.g. *eth1*
  - `ip`: vEPC component IP address used in test case topology

- `nextHop`: next hop IP address, to allow for vEPC inter-node communication

2.15 Update NSB test case definitions

NSB test case file designated for vEPC testing contains an example of specific test scenario configuration. Test operator may change these definitions as required for the use case that requires testing. Specifically, following subsections of the vEPC test case (section **scenarios**) may be changed.

1. Subsection `options`: contains custom parameters used for vEPC testing

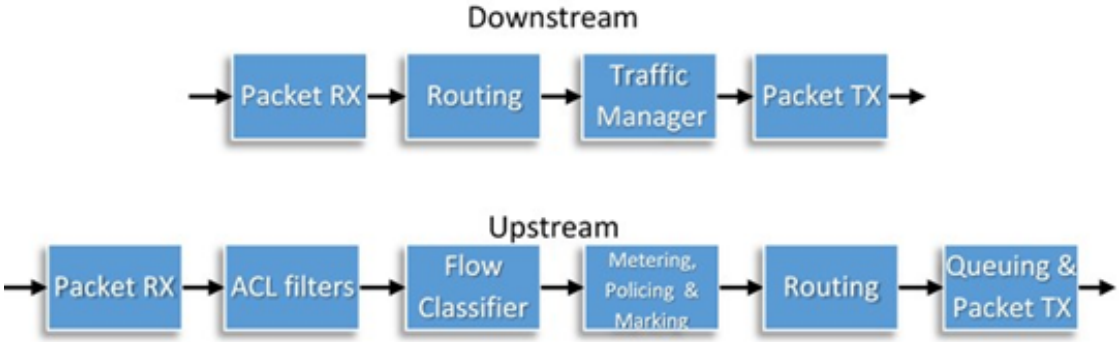
- subsection `dmf`: may contain one or more parameters specified in `traffic_profile` template file
- subsection `test_cases`: contains re-definitions of parameters specified in `session_profile` template file

> **Note:** All parameters in `session_profile`, value of which is a placeholder, needs to be re-defined to construct a valid test session.

2. Subsection `runner`: specifies the test duration and the interval of TG and VNF side KPIs polling. For more details, refer to *Architecture*.

2.15.1 Preparing test run of vPE test case

The vPE (Provider Edge Router) is a :term: *VNF* approximation serving as an Edge Router. The vPE is approximated using the `ip_pipeline` dpdk application.



The `vpe_config` file must be passed as it is not auto generated. The `vpe_script` defines the rules applied to each of the pipelines. This can be auto generated or a file can be passed using the `script_file` option in `vnf_config` as shown below. The `full_tm_profile_file` option must be used if a traffic manager is defined in `vpe_config`.

```
vnf_config: { file: './vpe_config/vpe_config_2_ports',
              action_bulk_file: './vpe_config/action_bulk_512.txt',
              full_tm_profile_file: './vpe_config/full_tm_profile_10G.cfg',
              script_file: './vpe_config/vpe_script_sample' }
```

Testcases for vPE can be found in the `vnf_samples/nsut/vpe` directory. A testcase can be started with the following command as an example:

```
yardstick task start /yardstick/samples/vnf_samples/nsut/vpe/tc_baremetal_rfc2544_
↪ipv4_1flow_64B_ixia.yaml
```

2.15.2 Preparing test run of vIPSEC test case

Location of vIPSEC test cases: `samples/vnf_samples/nsut/ipsec/`.

Before running a specific vIPSEC test case using NSB, some dependencies have to be preinstalled and properly configured. - VPP

```
export UBUNTU="xenial"
export RELEASE=".stable.1810"
sudo rm /etc/apt/sources.list.d/99fd.io.list
echo "deb [trusted=yes] https://nexus.fd.io/content/repositories/fd.io$RELEASE.ubuntu.
↪$UBUNTU.main/ ./" | sudo tee -a /etc/apt/sources.list.d/99fd.io.list
sudo apt-get update
sudo apt-get install vpp vpp-lib vpp-plugin vpp-dbg vpp-dev vpp-api-java vpp-api-
↪python vpp-api-lua
```

- VAT templates

  VAT templates is required for the VPP API.

```
mkdir -p /opt/nsb_bin/vpp/templates/
echo 'exec trace add dpdk-input 50' > /opt/nsb_bin/vpp/templates/enable_dpdk_traces.
↪vat
echo 'exec trace add vhost-user-input 50' > /opt/nsb_bin/vpp/templates/enable_vhost_
↪user_traces.vat
echo 'exec trace add memif-input 50' > /opt/nsb_bin/vpp/templates/enable_memif_traces.
↪vat
cat > /opt/nsb_bin/vpp/templates/dump_interfaces.vat << EOL
sw_interface_dump
dump_interface_table
quit
EOL
```

2.15.3 Preparing test run of vCMTS test case

Location of vCMTS test cases: `samples/vnf_samples/nsut/cmts/`.

Before running a specific vIPSEC test case using NSB, some changes must be made to the original vCMTS package.

Allow SSH access to the docker images

Follow the documentation at `https://docs.docker.com/engine/examples/running_ssh_service/` to allow SSH access to the Pktgen/vcmts-d containers located at:

- `$VCMTS_ROOT/pktgen/docker/docker-image-pktgen/Dockerfile` and
- `$VCMTS_ROOT/vcmtsd/docker/docker-image-vcmtsd/Dockerfile`

Deploy the ConfigMaps for Pktgen and vCMTSd

```
cd $VCMTS_ROOT/kubernetes/helm/pktgen
helm template . -x templates/pktgen-configmap.yaml > configmap.yaml
kubectl create -f configmap.yaml

cd $VCMTS_ROOT/kubernetes/helm/vcmtsd
```

(continues on next page)

(continued from previous page)

```
helm template . -x templates/vcmts-configmap.yaml > configmap.yaml
kubectl create -f configmap.yaml
```

2.16 Yardstick Test Cases

2.16.1 Abstract

This chapter lists available Yardstick test cases. Yardstick test cases are divided in two main categories:

- *Generic NFVI Test Cases* - Test Cases developed to realize the methodology described in *Methodology*
- *OPNFV Feature Test Cases* - Test Cases developed to verify one or more aspect of a feature delivered by an OPNFV Project.

2.16.2 Generic NFVI Test Case Descriptions

Yardstick Test Case Description TC001

Network Performance	
test case id	OPNFV_YARDSTICK_TC001_NETWORK PERFORMANCE
metric	Number of flows and throughput
test purpose	The purpose of TC001 is to evaluate the IaaS network performance with regards to flows and throughput, such as if and how different amounts of flows matter for the throughput between hosts on different compute blades. Typically e.g. the performance of a vSwitch depends on the number of flows running through it. Also performance of other equipment or entities can depend on the number of flows or the packet sizes used. The purpose is also to be able to spot the trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
test tool	pktgen Linux packet generator is a tool to generate packets at very high speed in the kernel. pktgen is mainly used to drive and LAN equipment test network. pktgen supports multi threading. To generate random MAC address, IP address, port number UDP packets, pktgen uses multiple CPU processors in the different PCI bus (PCI, PCIe bus) with Gigabit Ethernet tested (pktgen performance depends on the CPU processing speed, memory delay, PCI bus speed hardware parameters), Transmit data rate can be even larger than 10GBit/s. Visible can satisfy most card test requirements. (Pktgen is not always part of a Linux distribution, hence it needs to be installed. It is part of the Yardstick Docker image. As an example see the /yardstick/tools/ directory for how to generate a Linux image with pktgen included.)
test description	This test case uses Pktgen to generate packet flow between two hosts for simulating network workloads on the SUT.
traffic profile	An IP table is setup on server to monitor for received packets.
configuration	file: opnfv_yardstick_tc001.yaml Packet size is set to 60 bytes. Number of ports: 10, 50, 100, 500 and 1000, where each runs for 20 seconds. The whole sequence is run twice The client and server are distributed on different hardware. For SLA max_ppm is set to 1000. The amount of configured ports map to between 110 up to 1001000 flows, respectively.
applicability	Test can be configured with different: • packet sizes; • amount of flows; • test duration. Default values exist. SLA (optional): max_ppm: The number of packets per million packets sent that are acceptable to loose, not received.
usability	This test case is used for generating high network throughput to simulate certain workloads on the SUT. Hence it should work with other test cases.

Yardstick Test Case Description TC002

Network Latency	
test case id	OPNFV_YARDSTICK_TC002_NETWORK LATENCY
metric	RTT (Round Trip Time)
test purpose	The purpose of TC002 is to do a basic verification that network latency is within acceptable boundaries when packets travel between hosts located on same or different compute blades. The purpose is also to be able to spot the trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
test tool	ping Ping is a computer network administration software utility used to test the reachability of a host on an Internet Protocol (IP) network. It measures the round-trip time for packet sent from the originating host to a destination computer that are echoed back to the source. Ping is normally part of any Linux distribution, hence it doesn't need to be installed. It is also part of the Yardstick Docker image. (For example also a Cirros image can be downloaded from cirros-image, it includes ping)
test topology	Ping packets (ICMP protocol's mandatory ECHO_REQUEST datagram) are sent from host VM to target VM(s) to elicit ICMP ECHO_RESPONSE. For one host VM there can be multiple target VMs. Host VM and target VM(s) can be on same or different compute blades.
configuration	file: opnfv_yardstick_tc002.yaml Packet size 100 bytes. Test duration 60 seconds. One ping each 10 seconds. Test is iterated two times. SLA RTT is set to maximum 10 ms.
applicability	This test case can be configured with different: • packet sizes; • burst sizes; • ping intervals; • test durations; • test iterations. Default values exist. SLA is optional. The SLA in this test case serves as an example. Considerably lower RTT is expected, and also normal to achieve in balanced L2 environments. However, to cover most configurations, both bare metal and fully virtualized ones, this value should be possible to achieve and acceptable for black box testing. Many real time applications start to suffer badly if the RTT time is higher than this. Some may suffer bad also close to this RTT, while others may not suffer at all. It is a compromise that may have to be tuned for different configuration purposes.
usability	This test case is one of Yardstick's generic test. Thus it is runnable on most of the scenarios.
references	Ping ETSI-NFV-TST001
pre-test conditions	The test case image (cirros-image) needs to be installed into Glance with ping included in it. No POD specific requirements have been identified.

Yardstick Test Case Description TC004

Cache Utilization	
test case id	OPNFV_YARDSTICK_TC004_CACHE Utilization
metric	cache hit, cache miss, hit/miss ratio, buffer size and page cache size
test purpose	The purpose of TC004 is to evaluate the IaaS compute capability with regards to cache utilization.This test case should be run in parallel with other Yardstick test cases and not run as a stand-alone test case. This test case measures cache usage statistics, including cache hit, cache miss, hit ratio, buffer cache size and page cache size, with some wokloads runing on the infrastructure. Both average and maximun values are collected. The purpose is also to be able to spot the trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
test tool	cachestat cachestat is a tool using Linux ftrace capabilities for showing Linux page cache hit/miss statistics. (cachestat is not always part of a Linux distribution, hence it needs to be installed. As an example see the /yardstick/tools/ directory for how to generate a Linux image with cachestat included.)
test description	cachestat test is invoked in a host VM on a compute blade, cachestat test requires some other test cases running in the host to stimulate workload.
configuration	File: cachestat.yaml (in the 'samples' directory) Interval is set 1. Test repeat, pausing every 1 seconds in-between. Test durarion is set to 60 seconds. SLA is not available in this test case.
applicability	Test can be configured with different: • interval; • runner Duration. Default values exist.
usability	This test case is one of Yardstick's generic test. Thus it is runnable on most of the scenarios.
references	cachestat ETSI-NFV-TST001
pre-test conditions	The test case image needs to be installed into Glance with cachestat included in the image. No POD specific requirements have been identified.
test sequence	description and expected result
step 1	A host VM with cachestat installed is booted.
step 2	Yardstick is connected with the host VM by using ssh. 'cache_stat' bash script is copyied from Jump Host to the server VM via the ssh tunnel.
step 3	'cache_stat' script is invoked. Raw cache usage statistics are collected and filtrated. Average and maximum values are calculated and recorded. Logs are produced and stored. Result: Logs are stored.
step 4	The host VM is deleted.
test verdict	None. Cache utilization results are collected and stored.

Yardstick Test Case Description TC005

Storage Performance	
test case id	OPNFV_YARDSTICK_TC005_STORAGE PERFORMANCE
metric	IOPS (Average IOs performed per second), Throughput (Average disk read/write bandwidth rate), Latency (Average disk read/write latency)
test purpose	The purpose of TC005 is to evaluate the IaaS storage performance with regards to IOPS, throughput and latency. The purpose is also to be able to spot the trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
test tool	fio fio is an I/O tool meant to be used both for benchmark and stress/hardware verification. It has support for 19 different types of I/O engines (sync, mmap, libaio, posixaio, SG v3, splice, null, network, syslet, guasi, solarisaio, and more), I/O priorities (for newer Linux kernels), rate I/O, forked or threaded jobs, and much more. (fio is not always part of a Linux distribution, hence it needs to be installed. As an example see the /yardstick/tools/ directory for how to generate a Linux image with fio included.)
test description	fio test is invoked in a host VM on a compute blade, a job file as well as parameters are passed to fio and fio will start doing what the job file tells it to do.
configuration	file: opnfv_yardstick_tc005.yaml IO types is set to read, write, randwrite, randread, rw. IO block size is set to 4KB, 64KB, 1024KB. fio is run for each IO type and IO block size scheme, each iteration runs for 30 seconds (10 for ramp time, 20 for runtime). For SLA, minimum read/write iops is set to 100, minimum read/write throughput is set to 400 KB/s, and maximum read/write latency is set to 20000 usec.
applicability	This test case can be configured with different: • IO types; • IO block size; • IO depth; • ramp time; • test duration. Default values exist. SLA is optional. The SLA in this test case serves as an example. Considerably higher throughput and lower latency are expected. However, to cover most configurations, both baremetal and fully virtualized ones, this value should be possible to achieve and acceptable for black box testing. Many heavy IO applications start to suffer badly if the read/write bandwidths are lower than this.
usability	This test case is one of Yardstick's generic test. Thus it is runnable on most of the scenarios.
references	fio ETSI-NFV-TST001
pre-test conditions	The test case image needs to be installed into Glance with fio included in it. No POD specific requirements have been identified.

Yardstick Test Case Description TC006

Volume storage Performance	
test case id	OPNFV_YARDSTICK_TC006_VOLUME STORAGE PERFORMANCE
metric	IOPS (Average IOs performed per second), Throughput (Average disk read/write bandwidth rate), Latency (Average disk read/write latency)
test purpose	The purpose of TC006 is to evaluate the IaaS volume storage performance with regards to IOPS, throughput and latency. The purpose is also to be able to spot the trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
test tool	fio fio is an I/O tool meant to be used both for benchmark and stress/hardware verification. It has support for 19 different types of I/O engines (sync, mmap, libaio, posixaio, SG v3, splice, null, network, syslet, guasi, solarisaio, and more), I/O priorities (for newer Linux kernels), rate I/O, forked or threaded jobs, and much more. (fio is not always part of a Linux distribution, hence it needs to be installed. As an example see the /yardstick/tools/ directory for how to generate a Linux image with fio included.)
test description	fio test is invoked in a host VM with a volume attached on a compute blade, a job file as well as parameters are passed to fio and fio will start doing what the job file tells it to do.
configuration	file: opnfv_yardstick_tc006.yaml Fio job file is provided to define the benchmark process Target volume is mounted at /FIO_Test directory For SLA, minimum read/write iops is set to 100, minimum read/write throughput is set to 400 KB/s, and maximum read/write latency is set to 20000 usec.
applicability	This test case can be configured with different: • Job file; • Volume mount directory. SLA is optional. The SLA in this test case serves as an example. Considerably higher throughput and lower latency are expected. However, to cover most configurations, both baremetal and fully virtualized ones, this value should be possible to achieve and acceptable for black box testing. Many heavy IO applications start to suffer badly if the read/write bandwidths are lower than this.
usability	This test case is one of Yardstick's generic test. Thus it is runnable on most of the scenarios.
references	fio ETSI-NFV-TST001
pre-test conditions	The test case image needs to be installed into Glance with fio included in it. No POD specific requirements have been identified.
~~test sequence~~	~~description and expected result~~
step 1	A host VM with fio installed is booted. A 200G volume is attached to the host VM
step 2	Yardstick is connected with the host VM by using ssh. 'job_file.ini' is copyied from Jump Host to the host VM

Yardstick Test Case Description TC008

Packet Loss Extended Test	
test case id	OPNFV_YARDSTICK_TC008_NW PERF, Packet loss Extended Test
metric	Number of flows, packet size and throughput
test purpose	To evaluate the IaaS network performance with regards to flows and throughput, such as if and how different amounts of packet sizes and flows matter for the throughput between VMs on different compute blades. Typically e.g. the performance of a vSwitch depends on the number of flows running through it. Also performance of other equipment or entities can depend on the number of flows or the packet sizes used. The purpose is also to be able to spot trends. Test results, graphs ans similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
configuration	file: opnfv_yardstick_tc008.yaml Packet size: 64, 128, 256, 512, 1024, 1280 and 1518 bytes. Number of ports: 1, 10, 50, 100, 500 and 1000. The amount of configured ports map from 2 up to 1001000 flows, respectively. Each packet_size/port_amount combination is run ten times, for 20 seconds each. Then the next packet_size/port_amount combination is run, and so on. The client and server are distributed on different HW. For SLA max_ppm is set to 1000.
test tool	pktgen (Pktgen is not always part of a Linux distribution, hence it needs to be installed. It is part of the Yardstick Docker image. As an example see the /yardstick/tools/ directory for how to generate a Linux image with pktgen included.)
references	pktgen ETSI-NFV-TST001
applicability	Test can be configured with different packet sizes, amount of flows and test duration. Default values exist. SLA (optional): max_ppm: The number of packets per million packets sent that are acceptable to loose, not received.
pre-test conditions	The test case image needs to be installed into Glance with pktgen included in it. No POD specific requirements have been identified.
test sequence	description and expected result
step 1	The hosts are installed, as server and client. pktgen is invoked and logs are produced and stored. Result: Logs are stored.
test verdict	Fails only if SLA is not passed, or if there is a test case execution problem.

Yardstick Test Case Description TC009

Packet Loss	
test case id	OPNFV_YARDSTICK_TC009_NW PERF, Packet loss
metric	Number of flows, packets lost and throughput
test purpose	To evaluate the IaaS network performance with regards to flows and throughput, such as if and how different amounts of flows matter for the throughput between VMs on different compute blades. Typically e.g. the performance of a vSwitch depends on the number of flows running through it. Also performance of other equipment or entities can depend on the number of flows or the packet sizes used. The purpose is also to be able to spot trends. Test results, graphs ans similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
configuration	file: opnfv_yardstick_tc009.yaml Packet size: 64 bytes Number of ports: 1, 10, 50, 100, 500 and 1000. The amount of configured ports map from 2 up to 1001000 flows, respectively. Each port amount is run ten times, for 20 seconds each. Then the next port_amount is run, and so on. The client and server are distributed on different HW. For SLA max_ppm is set to 1000.
test tool	pktgen (Pktgen is not always part of a Linux distribution, hence it needs to be installed. It is part of the Yardstick Docker image. As an example see the /yardstick/tools/ directory for how to generate a Linux image with pktgen included.)
references	pktgen ETSI-NFV-TST001
applicability	Test can be configured with different packet sizes, amount of flows and test duration. Default values exist. SLA (optional): max_ppm: The number of packets per million packets sent that are acceptable to loose, not received.
pre-test conditions	The test case image needs to be installed into Glance with pktgen included in it. No POD specific requirements have been identified.
test sequence	description and expected result
step 1	The hosts are installed, as server and client. pktgen is invoked and logs are produced and stored. Result: logs are stored.
test verdict	Fails only if SLA is not passed, or if there is a test case execution problem.

Yardstick Test Case Description TC010

Memory Latency	
test case id	OPNFV_YARDSTICK_TC010_MEMORY LATENCY
metric	Memory read latency (nanoseconds)
test purpose	The purpose of TC010 is to evaluate the IaaS compute performance with regards to memory read latency. It measures the memory read latency for varying memory sizes and strides. Whole memory hierarchy is measured. The purpose is also to be able to spot the trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
test tool	Lmbench Lmbench is a suite of operating system microbenchmarks. This test uses lat_mem_rd tool from that suite including: • Context switching • Networking: connection establishment, pipe, TCP, UDP, and RPC hot potato • File system creates and deletes • Process creation • Signal handling • System call overhead • Memory read latency (LMbench is not always part of a Linux distribution, hence it needs to be installed. As an example see the /yardstick/tools/ directory for how to generate a Linux image with LMbench included.)
test description	LMbench lat_mem_rd benchmark measures memory read latency for varying memory sizes and strides. The benchmark runs as two nested loops. The outer loop is the stride size. The inner loop is the array size. For each array size, the benchmark creates a ring of pointers that point backward one stride. Traversing the array is done by: `p = (char **)*p;` in a for loop (the over head of the for loop is not significant; the loop is an unrolled loop 100 loads long). The size of the array varies from 512 bytes to (typically) eight megabytes. For the small sizes, the cache will have an effect, and the loads will be much faster. This becomes much more apparent when the data is plotted. Only data accesses are measured; the instruction cache is not measured. The results are reported in nanoseconds per load and have been verified accurate to within a few nanoseconds on an SGI Indy.
configuration	File: opnfv_yardstick_tc010.yaml • SLA (max_latency): 30 nanoseconds • Stride - 128 bytes • Stop size - 64 megabytes • Iterations: 10 - test is run 10 times iteratively. • Interval: 1 - there is 1 second delay between each iteration. SLA is optional. The SLA in this test case serves as an example. Considerably lower read latency is expected. However, to cover most configurations, both baremetal

Yardstick Test Case Description TC011

Packet delay variation between VMs	
test case id	OPNFV_YARDSTICK_TC011_PACKET DELAY VARIATION BETWEEN VMs
metric	jitter: packet delay variation (ms)
test purpose	The purpose of TC011 is to evaluate the IaaS network performance with regards to network jitter (packet delay variation). It measures the packet delay variation sending the packets from one VM to the other. The purpose is also to be able to spot the trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
test tool	iperf3 iPerf3 is a tool for active measurements of the maximum achievable bandwidth on IP networks. It supports tuning of various parameters related to timing, buffers and protocols. The UDP protocols can be used to measure jitter delay. (iperf3 is not always part of a Linux distribution, hence it needs to be installed. It is part of the Yardstick Docker image. As an example see the /yardstick/tools/ directory for how to generate a Linux image with pktgen included.)
test description	iperf3 test is invoked between a host VM and a target VM. Jitter calculations are continuously computed by the server, as specified by RTP in RFC 1889. The client records a 64 bit second/microsecond timestamp in the packet. The server computes the relative transit time as (server's receive time - client's send time). The client's and server's clocks do not need to be synchronized; any difference is subtracted outin the jitter calculation. Jitter is the smoothed mean of differences between consecutive transit times.
configuration	File: opnfv_yardstick_tc011.yaml • options: protocol: udp # The protocol used by iperf3 tools # Send the given number of packets without pausing: bandwidth: 20m • runner: duration: 30 # Total test duration 30 seconds. • SLA (optional): jitter: 10 (ms) # The maximum amount of jitter that is accepted.
applicability	Test can be configured with different: • **bandwidth: Test case can be configured with different** bandwidth. • duration: The test duration can be configured. • **jitter: SLA is optional. The SLA in this test case** serves as an example.
usability	This test case is one of Yardstick's generic test. Thus it is runnable on most of the scenarios.
references	iperf3 ETSI-NFV-TST001
pre-test conditions	The test case image needs to be installed into Glance with iperf3 included in the image. No POD specific requirements have been identified.

Yardstick Test Case Description TC012

Memory Bandwidth	
test case id	OPNFV_YARDSTICK_TC012_MEMORY BAND-WIDTH
metric	Memory read/write bandwidth (MBps)
test purpose	The purpose of TC012 is to evaluate the IaaS compute performance with regards to memory throughput. It measures the rate at which data can be read from and written to the memory (this includes all levels of memory). The purpose is also to be able to spot the trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
test tool	LMbench LMbench is a suite of operating system microbenchmarks. This test uses bw_mem tool from that suite including: • Cached file read • Memory copy (bcopy) • Memory read • Memory write • Pipe • TCP (LMbench is not always part of a Linux distribution, hence it needs to be installed. As an example see the /yardstick/tools/ directory for how to generate a Linux image with LMbench included.)
test description	LMbench bw_mem benchmark allocates twice the specified amount of memory, zeros it, and then times the copying of the first half to the second half. The benchmark is invoked in a host VM on a compute blade. Results are reported in megabytes moved per second.
configuration	File: opnfv_yardstick_tc012.yaml • SLA (optional): 15000 (MBps) min_bw: The minimum amount of memory bandwidth that is accepted. • Size: 10 240 kB - test allocates twice that size (20 480kB) zeros it and then measures the time it takes to copy from one side to another. • Benchmark: rdwr - measures the time to read data into memory and then write data to the same location. • Warmup: 0 - the number of iterations to perform before taking actual measurements. • Iterations: 10 - test is run 10 times iteratively. • Interval: 1 - there is 1 second delay between each iteration. SLA is optional. The SLA in this test case serves as an example. Considerably higher bandwidth is expected. However, to cover most configurations, both baremetal and fully virtualized ones, this value should be possible to achieve and acceptable for black box testing. Many heavy IO applications start to suffer badly if the read/write bandwidths are lower than this.
applicability	Test can be configured with different: • memory sizes; • memory operations (such as rd, wr, rdwr, cp, frd,

Yardstick Test Case Description TC014

Processing speed	
test case id	OPNFV_YARDSTICK_TC014_PROCESSING SPEED
metric	score of single cpu running, score of parallel running
test purpose	The purpose of TC014 is to evaluate the IaaS compute performance with regards to CPU processing speed. It measures score of single cpu running and parallel running. The purpose is also to be able to spot the trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
test tool	UnixBench Unixbench is the most used CPU benchmarking software tool. It can measure the performance of bash scripts, CPUs in multithreading and single threading. It can also measure the performance for parallel taks. Also, specific disk IO for small and large files are performed. You can use it to measure either linux dedicated servers and linux vps servers, running CentOS, Debian, Ubuntu, Fedora and other distros. (UnixBench is not always part of a Linux distribution, hence it needs to be installed. As an example see the /yardstick/tools/ directory for how to generate a Linux image with UnixBench included.)
test description	The UnixBench runs system benchmarks in a host VM on a compute blade, getting information on the CPUs in the system. If the system has more than one CPU, the tests will be run twice – once with a single copy of each test running at once, and once with N copies, where N is the number of CPUs. UnixBench will processs a set of results from a single test by averaging the individal pass results into a single final value.
configuration	file: opnfv_yardstick_tc014.yaml run_mode: Run unixbench in quiet mode or verbose mode test_type: dhry2reg, whetstone and so on For SLA with single_score and parallel_score, both can be set by user, default is NA.
applicability	Test can be configured with different: • test types; • dhry2reg; • whetstone. Default values exist. SLA (optional) : min_score: The minimun UnixBench score that is accepted.
usability	This test case is one of Yardstick's generic test. Thus it is runnable on most of the scenarios.
references	unixbench ETSI-NFV-TST001
pre-test conditions	The test case image needs to be installed into Glance with unixbench included in it. No POD specific requirements have been identified.
test sequence	description and expected result
step 1	A host VM with UnixBench installed is booted.
step 2	Yardstick is connected with the host VM by using ssh. "unixbench_benchmark" bash script is copied from

Yardstick Test Case Description TC015

Processing speed with impact on energy consumption and CPU load	
test case id	OPNFV_YARDSTICK_TC015_PROCESSING SPEED
metric	score of single cpu running, score of parallel running, energy consumption cpu load
test purpose	The purpose of TC015 is to evaluate the IaaS compute performance with regards to CPU processing speed with its impact on the energy consumption It measures score of single cpu running and parallel running. Energy consumption and cpu load are monitored while the cpu test is running. The purpose is also to be able to spot the trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations, different server types.
test tool	UnixBench Unixbench is the most used CPU benchmarking software tool. It can measure the performance of bash scripts, CPUs in multithreading and single threading. It can also measure the performance for parallel tasks. Also, specific disk IO for small and large files are performed. You can use it to measure either linux dedicated servers and linux vps servers, running CentOS, Debian, Ubuntu, Fedora and other distros. (UnixBench is not always part of a Linux distribution, hence it needs to be installed. As an example see the /yardstick/tools/ directory for how to generate a Linux image with UnixBench included.) Redfish API This HTTPS interface is provided by BMC of every telco grade server. Is is a standard interface.
test description	The UnixBench runs system benchmarks on a compute, getting information on the CPUs in the system. If the system has more than one CPU, the tests will be run twice – once with a single copy of each test running at once, and once with N N copies, where N is the number of CPUs. UnixBench will process a set of results from a single test by averaging the individual pass results into a single final value. While the cpu test is running Energy scenario run in background to monitor the number of watt consumed by the compute server on the fly. The same is done using Cpuload scenario to monitor the overall percentage of CPU used on the fly. This enables to balance the CPU score with its impact on energy consumption. Synchronized measurements enables to look at any relation between CPU load and energy consumption.
configuration	file: opnfv_yardstick_tc015.yaml **run_mode:** Run Energy and Cpuload in background Run unixbench in quiet mode or verbose mode test_type: dhry2reg, whetstone and so on Duration and Interval are set globally for Energy and Cpuload, aligned with duration of UnixBench test. SLA can be set for each scenario type. Default is NA. For SLA with single_score and parallel_score, both can be set by user, default is NA.

Yardstick Test Case Description TC024

CPU Load	
test case id	OPNFV_YARDSTICK_TC024_CPU Load
metric	CPU load
test purpose	To evaluate the CPU load performance of the IaaS. This test case should be run in parallel to other Yardstick test cases and not run as a stand-alone test case. Average, minimum and maximun values are obtained. The purpose is also to be able to spot trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
configuration	file: cpuload.yaml (in the 'samples' directory) • interval: 1 - repeat, pausing every 1 seconds in-between. • count: 10 - display statistics 10 times, then exit.
test tool	mpstat (mpstat is not always part of a Linux distribution, hence it needs to be installed. It is part of the Yardstick Glance image. However, if mpstat is not present the TC instead uses /proc/stats as source to produce "mpstat" output.
references	man-pages
applicability	Test can be configured with different: • interval; • count; • runner Iteration and intervals. There are default values for each above-mentioned option. Run in background with other test cases.
pre-test conditions	The test case image needs to be installed into Glance with mpstat included in it. No POD specific requirements have been identified.
test sequence	description and expected result
step 1	The host is installed. The related TC, or TCs, is invoked and mpstat logs are produced and stored. Result: Stored logs
test verdict	None. CPU load results are fetched and stored.

Yardstick Test Case Description TC037

Latency, CPU Load, Throughput, Packet Loss	
test case id	OPNFV_YARDSTICK_TC037_LATENCY,CPU LOAD,THROUGHPUT, PACKET LOSS
metric	Number of flows, latency, throughput, packet loss CPU utilization percentage, CPU interrupt per second
test purpose	The purpose of TC037 is to evaluate the IaaS compute capacity and network performance with regards to CPU utilization, packet flows and network throughput, such as if and how different amounts of flows matter for the throughput between hosts on different compute blades, and the CPU load variation. Typically e.g. the performance of a vSwitch depends on the number of flows running through it. Also performance of other equipment or entities can depend on the number of flows or the packet sizes used The purpose is also to be able to spot the trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
test tool	Ping, Pktgen, mpstat Ping is a computer network administration software utility used to test the reachability of a host on an Internet Protocol (IP) network. It measures the round-trip time for packet sent from the originating host to a destination computer that are echoed back to the source. Linux packet generator is a tool to generate packets at very high speed in the kernel. pktgen is mainly used to drive and LAN equipment test network. pktgen supports multi threading. To generate random MAC address, IP address, port number UDP packets, pktgen uses multiple CPU processors in the different PCI bus (PCI, PCIe bus) with Gigabit Ethernet tested (pktgen performance depends on the CPU processing speed, memory delay, PCI bus speed hardware parameters), Transmit data rate can be even larger than 10GBit/s. Visible can satisfy most card test requirements. The mpstat command writes to standard output activities for each available processor, processor 0 being the first one. Global average activities among all processors are also reported. The mpstat command can be used both on SMP and UP machines, but in the latter, only global average activities will be printed. (Ping is normally part of any Linux distribution, hence it doesn't need to be installed. It is also part of the Yardstick Docker image. For example also a Cirros image can be downloaded from cirros-image, it includes ping. Pktgen and mpstat are not always part of a Linux distribution, hence it needs to be installed. It is part of the Yardstick Docker image. As an example see the /yardstick/tools/ directory for how to generate a Linux image with pktgen and mpstat included.)
test description	This test case uses Pktgen to generate packet flow between two hosts for simulating network workloads on the SUT. Ping packets (ICMP protocol's mandatory ECHO_REQUEST datagram) are sent from a host VM to the target VM(s) to elicit ICMP ECHO_RESPONSE, meanwhile CPU activities are monitored by mpstat.

Yardstick Test Case Description TC038

Latency, CPU Load, Throughput, Packet Loss (Extended measurements)	
test case id	OPNFV_YARDSTICK_TC038_Latency,CPU Load,Throughput,Packet Loss
metric	Number of flows, latency, throughput, CPU load, packet loss
test purpose	To evaluate the IaaS network performance with regards to flows and throughput, such as if and how different amounts of flows matter for the throughput between hosts on different compute blades. Typically e.g. the performance of a vSwitch depends on the number of flows running through it. Also performance of other equipment or entities can depend on the number of flows or the packet sizes used. The purpose is also to be able to spot trends. Test results, graphs ans similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
configuration	file: opnfv_yardstick_tc038.yaml Packet size: 64 bytes Number of ports: 1, 10, 50, 100, 300, 500, 750 and 1000. The amount configured ports map from 2 up to 1001000 flows, respectively. Each port amount is run ten times, for 20 seconds each. Then the next port_amount is run, and so on. During the test CPU load on both client and server, and the network latency between the client and server are measured. The client and server are distributed on different HW. For SLA max_ppm is set to 1000.
test tool	pktgen (Pktgen is not always part of a Linux distribution, hence it needs to be installed. It is part of the Yardstick Glance image. As an example see the /yardstick/tools/ directory for how to generate a Linux image with pktgen included.) ping Ping is normally part of any Linux distribution, hence it doesn't need to be installed. It is also part of the Yardstick Glance image. (For example also a cirros image can be downloaded, it includes ping) mpstat (Mpstat is not always part of a Linux distribution, hence it needs to be installed. It is part of the Yardstick Glance image.
references	Ping and Mpstat man pages pktgen ETSI-NFV-TST001
applicability	Test can be configured with different packet sizes, amount of flows and test duration. Default values exist. SLA (optional): max_ppm: The number of packets per million packets sent that are acceptable to loose, not received.
pre-test conditions	The test case image needs to be installed into Glance with pktgen included in it. No POD specific requirements have been identified.
test sequence	description and expected result
step 1	The hosts are installed, as server and client. pktgen is invoked and logs are produced and stored. Result: Logs are stored.
test verdict	Fails only if SLA is not passed, or if there is a test case execution problem.

Yardstick Test Case Description TC042

Network Performance	
test case id	OPNFV_YARDSTICK_TC042_DPDK pktgen latency measurements
metric	L2 Network Latency
test purpose	Measure L2 network latency when DPDK is enabled between hosts on different compute blades.
configuration	file: opnfv_yardstick_tc042.yaml • Packet size: 64 bytes • SLA(max_latency): 100usec
test tool	DPDK Pktgen-dpdk (DPDK and Pktgen-dpdk are not part of a Linux distribution, hence they needs to be installed. As an example see the /yardstick/tools/ directory for how to generate a Linux image with DPDK and pktgen-dpdk included.)
references	DPDK Pktgen-dpdk ETSI-NFV-TST001
applicability	Test can be configured with different packet sizes. Default values exist.
pre-test conditions	The test case image needs to be installed into Glance with DPDK and pktgen-dpdk included in it. The NICs of compute nodes must support DPDK on POD. And at least compute nodes setup hugepage. If you want to achievement a hight performance result, it is recommend to use NUAM, CPU pin, OVS and so on.
test sequence	description and expected result
step 1	The hosts are installed on different blades, as server and client. Both server and client have three interfaces. The first one is management such as ssh. The other two are used by DPDK.
step 2	Testpmd is invoked with configurations to forward packets from one DPDK port to the other on server.
step 3	Pktgen-dpdk is invoked with configurations as a traffic generator and logs are produced and stored on client. Result: Logs are stored.
test verdict	Fails only if SLA is not passed, or if there is a test case execution problem.

Yardstick Test Case Description TC043

Network Latency Between NFVI Nodes	
test case id	OPNFV_YARDSTICK_TC043_LATENCY_BETWEEN_NFVI_NODES
metric	RTT (Round Trip Time)
test purpose	The purpose of TC043 is to do a basic verification that network latency is within acceptable boundaries when packets travel between different NFVI nodes. The purpose is also to be able to spot the trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
test tool	ping Ping is a computer network administration software utility used to test the reachability of a host on an Internet Protocol (IP) network. It measures the round-trip time for packet sent from the originating host to a destination computer that are echoed back to the source.
test topology	Ping packets (ICMP protocol's mandatory ECHO_REQUEST datagram) are sent from host node to target node to elicit ICMP ECHO_RESPONSE.
configuration	file: opnfv_yardstick_tc043.yaml Packet size 100 bytes. Total test duration 600 seconds. One ping each 10 seconds. SLA RTT is set to maximum 10 ms.
applicability	This test case can be configured with different: • packet sizes; • burst sizes; • ping intervals; • test durations; • test iterations. Default values exist. SLA is optional. The SLA in this test case serves as an example. Considerably lower RTT is expected, and also normal to achieve in balanced L2 environments. However, to cover most configurations, both bare metal and fully virtualized ones, this value should be possible to achieve and acceptable for black box testing. Many real time applications start to suffer badly if the RTT time is higher than this. Some may suffer bad also close to this RTT, while others may not suffer at all. It is a compromise that may have to be tuned for different configuration purposes.
references	Ping ETSI-NFV-TST001
pre_test conditions	Each pod node must have ping included in it.
test sequence	description and expected result
step 1	Yardstick is connected with the NFVI node by using ssh. 'ping_benchmark' bash script is copyied from Jump Host to the NFVI node via the ssh tunnel.
step 2	Ping is invoked. Ping packets are sent from server node to client node. RTT results are calculated and checked against the SLA. Logs are produced and stored. Result: Logs are stored.
test verdict	Test should not PASS if any RTT is above the optional SLA value, or if there is a test case execution problem.

Yardstick Test Case Description TC044

Memory Utilization	
test case id	OPNFV_YARDSTICK_TC044_Memory Utilization
metric	Memory utilization
test purpose	To evaluate the IaaS compute capability with regards to memory utilization.This test case should be run in parallel to other Yardstick test cases and not run as a stand-alone test case. Measure the memory usage statistics including used memory, free memory, buffer, cache and shared memory. Both average and maximun values are obtained. The purpose is also to be able to spot trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
configuration	File: memload.yaml (in the 'samples' directory) • interval: 1 - repeat, pausing every 1 seconds in-between. • count: 10 - display statistics 10 times, then exit.
test tool	free free provides information about unused and used memory and swap space on any computer running Linux or another Unix-like operating system. free is normally part of a Linux distribution, hence it doesn't needs to be installed.
references	man-pages ETSI-NFV-TST001
applicability	Test can be configured with different: • interval; • count; • runner Iteration and intervals. There are default values for each above-mentioned option. Run in background with other test cases.
pre-test conditions	The test case image needs to be installed into Glance with free included in the image. No POD specific requirements have been identified.
test sequence	description and expected result
step 1	The host is installed as client. The related TC, or TCs, is invoked and free logs are produced and stored. Result: logs are stored.
test verdict	None. Memory utilization results are fetched and stored.

Yardstick Test Case Description TC055

Compute Capacity	
test case id	OPNFV_YARDSTICK_TC055_Compute Capacity
metric	Number of cpus, number of cores, number of threads, available memory size and total cache size.
test purpose	To evaluate the IaaS compute capacity with regards to hardware specification, including number of cpus, number of cores, number of threads, available memory size and total cache size. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
configuration	file: opnfv_yardstick_tc055.yaml There is are no additional configurations to be set for this TC.
test tool	/proc/cpuinfo this TC uses /proc/cpuinfo as source to produce compute capacity output.
references	/proc/cpuinfo ETSI-NFV-TST001
applicability	None.
pre-test conditions	No POD specific requirements have been identified.
test sequence	description and expected result
step 1	The hosts are installed, TC is invoked and logs are produced and stored. Result: Logs are stored.
test verdict	None. Hardware specification are fetched and stored.

Yardstick Test Case Description TC061

Network Utilization	
test case id	OPNFV_YARDSTICK_TC061_Network Utilization
metric	Network utilization
test purpose	To evaluate the IaaS network capability with regards to network utilization, including Total number of packets received per second, Total number of packets transmitted per second, Total number of kilobytes received per second, Total number of kilobytes transmitted per second, Number of compressed packets received per second (for cslip etc.), Number of compressed packets transmitted per second, Number of multicast packets received per second, Utilization percentage of the network interface. This test case should be run in parallel to other Yardstick test cases and not run as a standalone test case. Measure the network usage statistics from the network devices Average, minimum and maximun values are obtained. The purpose is also to be able to spot trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
configuration	File: netutilization.yaml (in the 'samples' directory) • interval: 1 - repeat, pausing every 1 seconds in-between. • count: 1 - display statistics 1 times, then exit.
test tool	sar The sar command writes to standard output the contents of selected cumulative activity counters in the operating system. sar is normally part of a Linux distribution, hence it doesn't needs to be installed.
references	man-pages ETSI-NFV-TST001
applicability	Test can be configured with different: • interval; • count; • runner Iteration and intervals. There are default values for each above-mentioned option. Run in background with other test cases.
pre-test conditions	The test case image needs to be installed into Glance with sar included in the image. No POD specific requirements have been identified.
test sequence	description and expected result.
step 1	The host is installed as client. The related TC, or TCs, is invoked and sar logs are produced and stored. Result: logs are stored.
test verdict	None. Network utilization results are fetched and stored.

Yardstick Test Case Description TC063

Storage Capacity	
test case id	OPNFV_YARDSTICK_TC063_Storage Capacity
metric	Storage/disk size, block size Disk Utilization
test purpose	This test case will check the parameters which could decide several models and each model has its specified task to measure. The test purposes are to measure disk size, block size and disk utilization. With the test results, we could evaluate the storage capacity of the host.
configuration	file: opnfv_yardstick_tc063.yaml • test_type: "disk_size" • **runner:** type: Iteration iterations: 1 - test is run 1 time iteratively.
test tool	fdisk A command-line utility that provides disk partitioning functions iostat This is a computer system monitor tool used to collect and show operating system storage input and output statistics.
references	iostat fdisk ETSI-NFV-TST001
applicability	Test can be configured with different: • test_type: "disk size", "block size", "disk utilization" • **interval: 1 - how ofter to stat disk utilization** type: int unit: seconds • **count: 15 - how many times to stat disk utilization** type: int unit: na There are default values for each above-mentioned option. Run in background with other test cases.
pre-test conditions	The test case image needs to be installed into Glance No POD specific requirements have been identified.
test sequence	Output the specific storage capacity of disk information as the sequence into file.
step 1	The pod is available and the hosts are installed. Node5 is used and logs are produced and stored. Result: Logs are stored.
test verdict	None.

Yardstick Test Case Description TC069

Memory Bandwidth	
test case id	OPNFV_YARDSTICK_TC069_Memory Bandwidth
metric	Megabyte per second (MBps)
test purpose	To evaluate the IaaS compute performance with regards to memory bandwidth. Measure the maximum possible cache and memory performance while reading and writing certain blocks of data (starting from 1Kb and further in power of 2) continuously through ALU and FPU respectively. Measure different aspects of memory performance via synthetic simulations. Each simulation consists of four performances (Copy, Scale, Add, Triad). Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.

configuration	File: opnfv_yardstick_tc069.yaml
	• SLA (optional): 7000 (MBps) min_bandwidth: The minimum amount of memory bandwidth that is accepted.
	• type_id: 1 - runs a specified benchmark (by an ID number):
	`1 -- INTmark [writing] 4 -- FLOATmark [writing]`
	`2 -- INTmark [reading] 5 -- FLOATmark [reading]`
	`3 -- INTmem 6 -- FLOATmem`
	• **block_size: 64 Megabytes - the maximum block** size per array.
	• load: 32 Gigabytes - the amount of data load per pass.
	• iterations: 5 - test is run 5 times iteratively.
	• interval: 1 - there is 1 second delay between each iteration.

test tool	RAMspeed RAMspeed is a free open source command line utility to measure cache and memory performance of computer systems. RAMspeed is not always part of a Linux distribution, hence it needs to be installed in the test image.
references	RAMspeed ETSI-NFV-TST001
applicability	Test can be configured with different: • benchmark operations (such as INTmark [writing], INTmark [reading], FLOATmark [writing], FLOATmark [reading], INTmem, FLOATmem); • block size per array; • load per pass; • number of batch run iterations; • iterations and intervals. There are default values for each above-mentioned option.
pre-test conditions	The test case image needs to be installed into Glance with RAmspeed included in the image. No POD specific requirements have been identified.
test sequence	description and expected result
step 1	The host is installed as client. RAMspeed is invoked

Yardstick Test Case Description TC070

Latency, Memory Utilization, Throughput, Packet Loss	
test case id	OPNFV_YARDSTICK_TC070_Latency, Memory Utilization, Throughput,Packet Loss
metric	Number of flows, latency, throughput, Memory Utilization, packet loss
test purpose	To evaluate the IaaS network performance with regards to flows and throughput, such as if and how different amounts of flows matter for the throughput between hosts on different compute blades. Typically e.g. the performance of a vSwitch depends on the number of flows running through it. Also performance of other equipment or entities can depend on the number of flows or the packet sizes used. The purpose is also to be able to spot trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
configuration	file: opnfv_yardstick_tc070.yaml Packet size: 64 bytes Number of ports: 1, 10, 50, 100, 300, 500, 750 and 1000. The amount configured ports map from 2 up to 1001000 flows, respectively. Each port amount is run two times, for 20 seconds each. Then the next port_amount is run, and so on. During the test Memory Utilization on both client and server, and the network latency between the client and server are measured. The client and server are distributed on different HW. For SLA max_ppm is set to 1000.
test tool	pktgen Pktgen is not always part of a Linux distribution, hence it needs to be installed. It is part of the Yardstick Glance image. (As an example see the /yardstick/tools/ directory for how to generate a Linux image with pktgen included.) ping Ping is normally part of any Linux distribution, hence it doesn't need to be installed. It is also part of the Yardstick Glance image. (For example also a cirros image can be downloaded, it includes ping) free free provides information about unused and used memory and swap space on any computer running Linux or another Unix-like operating system. free is normally part of a Linux distribution, hence it doesn't needs to be installed.
references	Ping and free man pages pktgen ETSI-NFV-TST001
applicability	Test can be configured with different packet sizes, amount of flows and test duration. Default values exist. SLA (optional): max_ppm: The number of packets per million packets sent that are acceptable to lose, not received.
pre-test conditions	The test case image needs to be installed into Glance with pktgen included in it. No POD specific requirements have been identified.
test sequence	description and expected result
step 1	The hosts are installed, as server and client. pktgen is invoked and logs are produced and stored. Result: Logs are stored.
test verdict	Fails only if SLA is not passed, or if there is a test case execution problem.

Yardstick Test Case Description TC071

Latency, Cache Utilization, Throughput, Packet Loss	
test case id	OPNFV_YARDSTICK_TC071_Latency, Cache Utilization, Throughput,Packet Loss
metric	Number of flows, latency, throughput, Cache Utilization, packet loss
test purpose	To evaluate the IaaS network performance with regards to flows and throughput, such as if and how different amounts of flows matter for the throughput between hosts on different compute blades. Typically e.g. the performance of a vSwitch depends on the number of flows running through it. Also performance of other equipment or entities can depend on the number of flows or the packet sizes used. The purpose is also to be able to spot trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
configuration	file: opnfv_yardstick_tc071.yaml Packet size: 64 bytes Number of ports: 1, 10, 50, 100, 300, 500, 750 and 1000. The amount configured ports map from 2 up to 1001000 flows, respectively. Each port amount is run two times, for 20 seconds each. Then the next port_amount is run, and so on. During the test Cache Utilization on both client and server, and the network latency between the client and server are measured. The client and server are distributed on different HW. For SLA max_ppm is set to 1000.
test tool	pktgen Pktgen is not always part of a Linux distribution, hence it needs to be installed. It is part of the Yardstick Glance image. (As an example see the /yardstick/tools/ directory for how to generate a Linux image with pktgen included.) ping Ping is normally part of any Linux distribution, hence it doesn't need to be installed. It is also part of the Yardstick Glance image. (For example also a cirros image can be downloaded, it includes ping) cachestat cachestat is not always part of a Linux distribution, hence it needs to be installed.
references	Ping man pages pktgen cachestat ETSI-NFV-TST001
applicability	Test can be configured with different packet sizes, amount of flows and test duration. Default values exist. SLA (optional): max_ppm: The number of packets per million packets sent that are acceptable to lose, not received.
pre-test conditions	The test case image needs to be installed into Glance with pktgen included in it. No POD specific requirements have been identified.
test sequence	description and expected result
step 1	The hosts are installed, as server and client. pktgen is invoked and logs are produced and stored. Result: Logs are stored.
test verdict	Fails only if SLA is not passed, or if there is a test case execution problem.

Yardstick Test Case Description TC072

Latency, Network Utilization, Throughput, Packet Loss	
test case id	OPNFV_YARDSTICK_TC072_Latency, Network Utilization, Throughput,Packet Loss
metric	Number of flows, latency, throughput, Network Utilization, packet loss
test purpose	To evaluate the IaaS network performance with regards to flows and throughput, such as if and how different amounts of flows matter for the throughput between hosts on different compute blades. Typically e.g. the performance of a vSwitch depends on the number of flows running through it. Also performance of other equipment or entities can depend on the number of flows or the packet sizes used. The purpose is also to be able to spot trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
configuration	file: opnfv_yardstick_tc072.yaml Packet size: 64 bytes Number of ports: 1, 10, 50, 100, 300, 500, 750 and 1000. The amount configured ports map from 2 up to 1001000 flows, respectively. Each port amount is run two times, for 20 seconds each. Then the next port_amount is run, and so on. During the test Network Utilization on both client and server, and the network latency between the client and server are measured. The client and server are distributed on different HW. For SLA max_ppm is set to 1000.
test tool	pktgen Pktgen is not always part of a Linux distribution, hence it needs to be installed. It is part of the Yardstick Glance image. (As an example see the /yardstick/tools/ directory for how to generate a Linux image with pktgen included.) ping Ping is normally part of any Linux distribution, hence it doesn't need to be installed. It is also part of the Yardstick Glance image. (For example also a cirros image can be downloaded, it includes ping) sar The sar command writes to standard output the contents of selected cumulative activity counters in the operating system. sar is normally part of a Linux distribution, hence it doesn't needs to be installed.
references	Ping and sar man pages pktgen ETSI-NFV-TST001
applicability	Test can be configured with different packet sizes, amount of flows and test duration. Default values exist. SLA (optional): max_ppm: The number of packets per million packets sent that are acceptable to lose, not received.
pre-test conditions	The test case image needs to be installed into Glance with pktgen included in it. No POD specific requirements have been identified.
test sequence	description and expected result
step 1	The hosts are installed, as server and client. pktgen is invoked and logs are produced and stored. Result: Logs are stored.
test verdict	Fails only if SLA is not passed, or if there is a test case execution problem.

Yardstick Test Case Description TC073

Throughput per NFVI node test	
test case id	OPNFV_YARDSTICK_TC073_Network latency and throughput between nodes
metric	Network latency and throughput
test purpose	To evaluate the IaaS network performance with regards to flows and throughput, such as if and how different amounts of packet sizes and flows matter for the throughput between nodes in one pod.
configuration	file: opnfv_yardstick_tc073.yaml Packet size: default 1024 bytes. Test length: default 20 seconds. The client and server are distributed on different nodes. For SLA max_mean_latency is set to 100.
test tool	netperf Netperf is a software application that provides network bandwidth testing between two hosts on a network. It supports Unix domain sockets, TCP, SCTP, DLPI and UDP via BSD Sockets. Netperf provides a number of predefined tests e.g. to measure bulk (unidirectional) data transfer or request response performance. (netperf is not always part of a Linux distribution, hence it needs to be installed.)
references	netperf Man pages ETSI-NFV-TST001
applicability	Test can be configured with different packet sizes and test duration. Default values exist. SLA (optional): max_mean_latency
pre-test conditions	The POD can be reached by external ip and logged on via ssh
test sequence	description and expected result
step 1	Install netperf tool on each specified node, one is as the server, and the other as the client.
step 2	Log on to the client node and use the netperf command to execute the network performance test
step 3	The throughput results stored.
test verdict	Fails only if SLA is not passed, or if there is a test case execution problem.

Yardstick Test Case Description TC074

Storperf	
test case id	OPNFV_YARDSTICK_TC074_Storperf
metric	Storage performance
test purpose	To evaluate and report on the Cinder volume performance. This testcase integrates with OPNFV StorPerf to measure block performance of the underlying Cinder drivers. Many options are supported, and even the root disk (Glance ephemeral storage can be profiled. The fundamental concept of the test case is to first fill the volumes with random data to ensure reported metrics are indicative of continued usage and not skewed by transitional performance while the underlying storage driver allocates blocks. The metrics for filling the volumes with random data are not reported in the final results. The test also ensures the volumes are performing at a consistent level of performance by measuring metrics every minute, and comparing the trend of the metrics over the run. By evaluating the min and max values, as well as the slope of the trend, it can make the determination that the metrics are stable, and not fluctuating beyond industry standard norms.
configuration	file: opnfv_yardstick_tc074.yaml • agent_count: 1 - the number of VMs to be created • agent_image: "Ubuntu-14.04" - image used for creating VMs • public_network: "ext-net" - name of public network • volume_size: 2 - cinder volume size • block_sizes: "4096" - data block size • queue_depths: "4" - the number of simultaneous I/Os to perform at all times • StorPerf_ip: "192.168.200.2" • query_interval: 10 - state query interval • timeout: 600 - maximum allowed job time
test tool	Storperf StorPerf is a tool to measure block and object storage performance in an NFVI. StorPerf is delivered as a Docker container from https://hub.docker.com/r/opnfv/storperf-master/tags/. The underlying tool used is FIO, and StorPerf supports any FIO option in order to tailor the test to the exact workload needed.
references	Storperf ETSI-NFV-TST001
applicability	Test can be configured with different: • agent_count • volume_size • block_sizes • queue_depths • query_interval • timeout • target=[device or path] The path to either an attached storage device (/dev/vdb, etc) or a directory path (/opt/storperf) that will be used to execute the performance test. In the case of a device,

Yardstick Test Case Description TC075

Network Capacity and Scale Testing	
test case id	OPNFV_YARDSTICK_TC075_Network_Capacity_and_Scale_testing
metric	Number of connections, Number of frames sent/received
test purpose	To evaluate the network capacity and scale with regards to connections and frmaes.
configuration	file: opnfv_yardstick_tc075.yaml There is no additional configuration to be set for this TC.
test tool	netstar Netstat is normally part of any Linux distribution, hence it doesn't need to be installed.
references	Netstat man page ETSI-NFV-TST001
applicability	This test case is mainly for evaluating network performance.
pre_test conditions	Each pod node must have netstat included in it.
test sequence	description and expected result
step 1	The pod is available. Netstat is invoked and logs are produced and stored. Result: Logs are stored.
test verdict	None. Number of connections and frames are fetched and stored.

Yardstick Test Case Description TC076

Monitor Network Metrics	
test case id	OPNFV_YARDSTICK_TC076_Monitor_Network_Metrics
metric	IP datagram error rate, ICMP message error rate, TCP segment error rate and UDP datagram error rate
test purpose	The purpose of TC076 is to evaluate the IaaS network reliability with regards to IP datagram error rate, ICMP message error rate, TCP segment error rate and UDP datagram error rate. TC076 monitors network metrics provided by the Linux kernel in a host and calculates IP datagram error rate, ICMP message error rate, TCP segment error rate and UDP datagram error rate. The purpose is also to be able to spot the trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
test tool	nstat nstat is a simple tool to monitor kernel snmp counters and network interface statistics. (nstat is not always part of a Linux distribution, hence it needs to be installed. nstat is provided by the iproute2 collection, which is usually also the name of the package in many Linux distributions.As an example see the /yardstick/tools/ directory for how to generate a Linux image with iproute2 included.)
test description	Ping packets (ICMP protocol's mandatory ECHO_REQUEST datagram) are sent from host VM to target VM(s) to elicit ICMP ECHO_RESPONSE. nstat is invoked on the target vm to monitors network metrics provided by the Linux kernel.
configuration	file: opnfv_yardstick_tc076.yaml There is no additional configuration to be set for this TC.
references	nstat man page ETSI-NFV-TST001
applicability	This test case is mainly for monitoring network metrics.
pre_test conditions	The test case image needs to be installed into Glance with fio included in it. No POD specific requirements have been identified.
test sequence	description and expected result
step 1	Two host VMs are booted, as server and client.
step 2	Yardstick is connected with the server VM by using ssh. 'ping_benchmark' bash script is copyied from Jump Host to the server VM via the ssh tunnel.
step 3	Ping is invoked. Ping packets are sent from server VM to client VM. RTT results are calculated and checked against the SLA. nstat is invoked on the client vm to monitors network metrics provided by the Linux kernel. IP datagram error rate, ICMP message error rate, TCP segment error rate and UDP datagram error rate are calculated. Logs are produced and stored. Result: Logs are stored.
step 4	Two host VMs are deleted.
test verdict	None.

Yardstick Test Case Description TC078

Compute Performance	
test case id	OPNFV_YARDSTICK_TC078_SPEC CPU 2006
metric	compute-intensive performance
test purpose	The purpose of TC078 is to evaluate the IaaS compute performance by using SPEC CPU 2006 benchmark. The SPEC CPU 2006 benchmark has several different ways to measure computer performance. One way is to measure how fast the computer completes a single task; this is called a speed measurement. Another way is to measure how many tasks computer can accomplish in a certain amount of time; this is called a throughput, capacity or rate measurement.
test tool	SPEC CPU 2006 The SPEC CPU 2006 benchmark is SPEC's industry-standardized, CPU-intensive benchmark suite, stressing a system's processor, memory subsystem and compiler. This benchmark suite includes the SPECint benchmarks and the SPECfp benchmarks. The SPECint 2006 benchmark contains 12 different enchmark tests and the SPECfp 2006 benchmark contains 19 different benchmark tests. SPEC CPU 2006 is not always part of a Linux distribution. SPEC requires that users purchase a license and agree with their terms and conditions. For this test case, users must manually download cpu2006-1.2.iso from the SPEC website and save it under the yardstick/resources folder (e.g. /home/opnfv/repos/yardstick/yardstick/resources/cpu2006-1.2.iso) SPEC CPU® 2006 benchmark is available for purchase via the SPEC order form (https://www.spec.org/order.html).
test description	This test case uses SPEC CPU 2006 benchmark to measure compute-intensive performance of hosts.
configuration	file: spec_cpu.yaml (in the 'samples' directory) benchmark_subset is set to int. SLA is not available in this test case.
applicability	Test can be configured with different: • benchmark_subset - a subset of SPEC CPU2006 benchmarks to run; • SPECint_benchmark - a SPECint benchmark to run; • SPECint_benchmark - a SPECfp benchmark to run; • output_format - desired report format; • runspec_config - SPEC CPU2006 config file provided to the runspec binary; • runspec_iterations - the number of benchmark iterations to execute. For a reportable run, must be 3; • runspec_tune - tuning to use (base, peak, or all). For a reportable run, must be either base or all. Reportable runs do base first, then (optionally) peak; • runspec_size - size of input data to run (test, train, or ref). Reportable runs ensure that your binaries can produce correct results with the test and train workloads

Yardstick Test Case Description TC079

Storage Performance	
test case id	OPNFV_YARDSTICK_TC079_Bonnie++
metric	Sequential Input/Output and Sequential/Random Create speed and CPU useage.
test purpose	The purpose of TC078 is to evaluate the IaaS storage performance with regards to Sequential Input/Output and Sequential/Random Create speed and CPU useage statistics.
test tool	Bonnie++ Bonnie++ is a disk and file system benchmarking tool for measuring I/O performance. With Bonnie++ you can quickly and easily produce a meaningful value to represent your current file system performance. Bonnie++ is not always part of a Linux distribution, hence it needs to be installed in the test image.
test description	**This test case uses Bonnie++ to perform the tests below:** • Create files in sequential order • Stat files in sequential order • Delete files in sequential order • Create files in random order • Stat files in random order • Delete files in random order
configuration	file: bonnie++.yaml (in the 'samples' directory) file_size is set to 1024; ram_size is set to 512; test_dir is set to '/tmp'; concurrency is set to 1. SLA is not available in this test case.
applicability	Test can be configured with different: • file_size - size fo the test file in MB. File size should be double RAM for good results; • ram_size - specify RAM size in MB to use, this is used to reduce testing time; • test_dir - this directory is where bonnie++ will create the benchmark operations; • test_user - the user who should perform the test. This is not required if you are not running as root; • concurrency - number of thread to perform test;
usability	This test case is used for executing Bonnie++ benchmark in VMs.
references	bonnie++_ ETSI-NFV-TST001
pre-test conditions	The Bonnie++ distribution includes a 'bon_csv2html' Perl script, which takes the comma-separated values reported by Bonnie++ and generates an HTML page displaying them. To use this feature, bonnie++ is required to be install with yardstick (e.g. in yardstick docker).
test sequence	description and expected result
step 1	A host VM with fio installed is booted.
step 2	Yardstick is connected with the host VM by using ssh.
step 3	Bonnie++ benchmark is invoked. Simulated IO operations are started. Logs are produced and stored. Result: Logs are stored.
step 4	An HTML report is generated using bonnie++ benchmark results and stored under /tmp/bonnie.html.

Yardstick Test Case Description TC080

Network Latency	
test case id	OPNFV_YARDSTICK_TC080_NETWORK_LATENCY_BETWEEN_CO
metric	RTT (Round Trip Time)
test purpose	The purpose of TC080 is to do a basic verification that network latency is within acceptable boundaries when packets travel between containers located in two different Kubernetes pods. The purpose is also to be able to spot the trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
test tool	ping Ping is a computer network administration software utility used to test the reachability of a host on an Internet Protocol (IP) network. It measures the round-trip time for packet sent from the originating host to a destination computer that are echoed back to the source. Ping is normally part of any Linux distribution, hence it doesn't need to be installed. It is also part of the Yardstick Docker image.
test topology	Ping packets (ICMP protocol's mandatory ECHO_REQUEST datagram) are sent from host container to target container to elicit ICMP ECHO_RESPONSE.
configuration	file: opnfv_yardstick_tc080.yaml Packet size 200 bytes. Test duration 60 seconds. SLA RTT is set to maximum 10 ms.
applicability	This test case can be configured with different: • packet sizes; • burst sizes; • ping intervals; • test durations; • test iterations. Default values exist. SLA is optional. The SLA in this test case serves as an example. Considerably lower RTT is expected, and also normal to achieve in balanced L2 environments. However, to cover most configurations, both bare metal and fully virtualized ones, this value should be possible to achieve and acceptable for black box testing. Many real time applications start to suffer badly if the RTT time is higher than this. Some may suffer bad also close to this RTT, while others may not suffer at all. It is a compromise that may have to be tuned for different configuration purposes.
usability	This test case should be run in Kunernetes environment.
references	Ping ETSI-NFV-TST001
pre-test conditions	The test case Docker image (openretriever/yardstick) needs to be pulled into Kubernetes environment. No further requirements have been identified.
test sequence	description and expected result
step 1	Two containers are booted, as server and client.
step 2	Yardstick is connected with the server container by using ssh. 'ping_benchmark' bash script is copied from Jump Host to the server container via the ssh tunnel.
step 3	Ping is invoked. Ping packets are sent from server con-

Yardstick Test Case Description TC081

Network Latency	
test case id	OPNFV_YARDSTICK_TC081_NETWORK_LATENCY_BETWEEN_CC _VM
metric	RTT (Round Trip Time)
test purpose	The purpose of TC081 is to do a basic verification that network latency is within acceptable boundaries when packets travel between a containers and a VM. The purpose is also to be able to spot the trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
test tool	ping Ping is a computer network administration software utility used to test the reachability of a host on an Internet Protocol (IP) network. It measures the round-trip time for packet sent from the originating host to a destination computer that are echoed back to the source. Ping is normally part of any Linux distribution, hence it doesn't need to be installed. It is also part of the Yardstick Docker image. (For example also a Cirros image can be downloaded from cirros-image, it includes ping)
test topology	Ping packets (ICMP protocol's mandatory ECHO_REQUEST datagram) are sent from host container to target vm to elicit ICMP ECHO_RESPONSE.
configuration	file: opnfv_yardstick_tc081.yaml Packet size 200 bytes. Test duration 60 seconds. SLA RTT is set to maximum 10 ms.
applicability	This test case can be configured with different: • packet sizes; • burst sizes; • ping intervals; • test durations; • test iterations. Default values exist. SLA is optional. The SLA in this test case serves as an example. Considerably lower RTT is expected, and also normal to achieve in balanced L2 environments. However, to cover most configurations, both bare metal and fully virtualized ones, this value should be possible to achieve and acceptable for black box testing. Many real time applications start to suffer badly if the RTT time is higher than this. Some may suffer bad also close to this RTT, while others may not suffer at all. It is a compromise that may have to be tuned for different configuration purposes.
usability	This test case should be run in Kunernetes environment.
references	Ping ETSI-NFV-TST001
pre-test conditions	The test case Docker image (openretriever/yardstick) needs to be pulled into Kubernetes environment. The VM image (cirros-image) needs to be installed into Glance with ping included in it. No further requirements have been identified.
test sequence	description and expected result
step 1	A containers is booted, as server and a VM is booted as client.
step 2	Yardstick is connected with the server container by us-

Yardstick Test Case Description TC083

Throughput per VM test	
test case id	OPNFV_YARDSTICK_TC083_Network latency and throughput between VMs
metric	Network latency and throughput
test purpose	To evaluate the IaaS network performance with regards to flows and throughput, such as if and how different amounts of packet sizes and flows matter for the throughput between 2 VMs in one pod.
configuration	file: opnfv_yardstick_tc083.yaml Packet size: default 1024 bytes. Test length: default 20 seconds. The client and server are distributed on different nodes. For SLA max_mean_latency is set to 100.
test tool	netperf Netperf is a software application that provides network bandwidth testing between two hosts on a network. It supports Unix domain sockets, TCP, SCTP, DLPI and UDP via BSD Sockets. Netperf provides a number of predefined tests e.g. to measure bulk (unidirectional) data transfer or request response performance. (netperf is not always part of a Linux distribution, hence it needs to be installed.)
references	netperf Man pages ETSI-NFV-TST001
applicability	Test can be configured with different packet sizes and test duration. Default values exist. SLA (optional): max_mean_latency
pre-test conditions	The POD can be reached by external ip and logged on via ssh
test sequence	description and expected result
step 1	Install netperf tool on each specified node, one is as the server, and the other as the client.
step 2	Log on to the client node and use the netperf command to execute the network performance test
step 3	The throughput results stored.
test verdict	Fails only if SLA is not passed, or if there is a test case execution problem.

Yardstick Test Case Description TC084

Compute Performance	
test case id	OPNFV_YARDSTICK_TC084_SPEC CPU 2006 FOR VM
metric	compute-intensive performance
test purpose	The purpose of TC084 is to evaluate the IaaS compute performance by using SPEC CPU 2006 benchmark. The SPEC CPU 2006 benchmark has several different ways to measure computer performance. One way is to measure how fast the computer completes a single task; this is called a speed measurement. Another way is to measure how many tasks computer can accomplish in a certain amount of time; this is called a throughput, capacity or rate measurement.
test tool	SPEC CPU 2006 The SPEC CPU 2006 benchmark is SPEC's industry-standardized, CPU-intensive benchmark suite, stressing a system's processor, memory subsystem and compiler. This benchmark suite includes the SPECint benchmarks and the SPECfp benchmarks. The SPECint 2006 benchmark contains 12 different benchmark tests and the SPECfp 2006 benchmark contains 19 different benchmark tests. SPEC CPU 2006 is not always part of a Linux distribution. SPEC requires that users purchase a license and agree with their terms and conditions. For this test case, users must manually download cpu2006-1.2.iso from the SPEC website and save it under the yardstick/resources folder (e.g. /home/opnfv/repos/yardstick/yardstick/resources/cpu2006-1.2.iso) SPEC CPU® 2006 benchmark is available for purchase via the SPEC order form (https://www.spec.org/order.html).
test description	This test case uses SPEC CPU 2006 benchmark to measure compute-intensive performance of VMs.
configuration	file: opnfv_yardstick_tc084.yaml benchmark_subset is set to int. SLA is not available in this test case.
applicability	Test can be configured with different: • benchmark_subset - a subset of SPEC CPU 2006 benchmarks to run; • SPECint_benchmark - a SPECint benchmark to run; • SPECint_benchmark - a SPECfp benchmark to run; • output_format - desired report format; • runspec_config - SPEC CPU 2006 config file provided to the runspec binary; • runspec_iterations - the number of benchmark iterations to execute. For a reportable run, must be 3; • runspec_tune - tuning to use (base, peak, or all). For a reportable run, must be either base or all. Reportable runs do base first, then (optionally) peak; • runspec_size - size of input data to run (test, train, or ref). Reportable runs ensure that your binaries can produce correct results with the test and train

2.16.3 OPNFV Feature Test Cases

H A

Yardstick Test Case Description TC019

Control Node Openstack Service High Availability	
test case id	OPNFV_YARDSTICK_TC019_HA: Control node Openstack service down
test purpose	This test case will verify the high availability of the service provided by OpenStack (like nova-api, neutro-server) on control node.
test method	This test case kills the processes of a specific Openstack service on a selected control node, then checks whether the request of the related Openstack command is OK and the killed processes are recovered.
attackers	In this test case, an attacker called "kill-process" is needed. This attacker includes three parameters: 1) fault_type: which is used for finding the attacker's scripts. It should be always set to "kill-process" in this test case. 2) process_name: which is the process name of the specified OpenStack service. If there are multiple processes use the same name on the host, all of them are killed by this attacker. 3) host: which is the name of a control node being attacked. e.g. -fault_type: "kill-process" -process_name: "nova-api" -host: node1
monitors	In this test case, two kinds of monitor are needed: 1. the "openstack-cmd" monitor constantly request a specific Openstack command, which needs two parameters: 1. monitor_type: which is used for finding the monitor class and related scritps. It should be always set to "openstack-cmd" for this monitor. 2. command_name: which is the command name used for request 2. the "process" monitor check whether a process is running on a specific node, which needs three parameters: 1. monitor_type: which used for finding the monitor class and related scritps. It should be always set to "process" for this monitor. 2. process_name: which is the process name for monitor 3. host: which is the name of the node runing the process e.g. monitor1: -monitor_type: "openstack-cmd" -command_name: "openstack server list" monitor2: -monitor_type: "process" -process_name: "nova-api" -host: node1
metrics	In this test case, there are two metrics: 1)service_outage_time: which indicates the maximum outage time (seconds) of the specified Openstack command request. 2)process_recover_time: which indicates the maximun time (seconds) from the process being killed to recovered
test tool	Developed by the project. Please see folder: "yardstick/benchmark/scenarios/availability/ha_tools"
references	ETSI NFV REL001
configuration	This test case needs two configuration files: 1) test case file: opnfv_yardstick_tc019.yaml -Attackers: see above "attackers" discription -waiting_time: which is the time

Yardstick Test Case Description TC025

OpenStack Controller Node abnormally shutdown High Availability	
test case id	OPNFV_YARDSTICK_TC025_HA: OpenStack Controller Node abnormally shutdown
test purpose	This test case will verify the high availability of controller node. When one of the controller node abnormally shutdown, the service provided by it should be OK.
test method	This test case shutdowns a specified controller node with some fault injection tools, then checks whether all services provided by the controller node are OK with some monitor tools.
attackers	In this test case, an attacker called "host-shutdown" is needed. This attacker includes two parameters: 1) fault_type: which is used for finding the attacker's scripts. It should be always set to "host-shutdown" in this test case. 2) host: the name of a controller node being attacked. e.g. -fault_type: "host-shutdown" -host: node1
monitors	In this test case, one kind of monitor are needed: 1. the "openstack-cmd" monitor constantly request a specific Openstack command, which needs two parameters 1) monitor_type: which is used for finding the monitor class and related scripts. It should be always set to "openstack-cmd" for this monitor. 2) command_name: which is the command name used for request There are four instance of the "openstack-cmd" monitor: monitor1: -monitor_type: "openstack-cmd" -api_name: "nova image-list" monitor2: -monitor_type: "openstack-cmd" -api_name: "neutron router-list" monitor3: -monitor_type: "openstack-cmd" -api_name: "heat stack-list" monitor4: -monitor_type: "openstack-cmd" -api_name: "cinder list"
metrics	In this test case, there is one metric: 1)service_outage_time: which indicates the maximum outage time (seconds) of the specified Openstack command request.
test tool	Developed by the project. Please see folder: "yardstick/benchmark/scenarios/availability/ha_tools"
references	ETSI NFV REL001
configuration	This test case needs two configuration files: 1) test case file: opnfv_yardstick_tc019.yaml -Attackers: see above "attackers" discription -waiting_time: which is the time (seconds) from the process being killed to stoping monitors the monitors -Monitors: see above "monitors" discription -SLA: see above "metrics" discription 2)POD file: pod.yaml The POD configuration should record on pod.yaml first. the "host" item in this test case will use the node name in the pod.yaml.
test sequence	description and expected result
step 1	start monitors: each monitor will run with independently process Result: The monitor info will be collected.
step 2	do attacker: connect the host through SSH, and then execute shutdown script on the host

Yardstick Test Case Description TC045

Control Node Openstack Service High Availability - Neutron Server	
test case id	OPNFV_YARDSTICK_TC045: Control node Openstack service down - neutron server
test purpose	This test case will verify the high availability of the network service provided by OpenStack (neutro-server) on control node.
test method	This test case kills the processes of neutron-server service on a selected control node, then checks whether the request of the related Openstack command is OK and the killed processes are recovered.
attackers	In this test case, an attacker called "kill-process" is needed. This attacker includes three parameters: 1) fault_type: which is used for finding the attacker's scripts. It should be always set to "kill-process" in this test case. 2) process_name: which is the process name of the specified OpenStack service. If there are multiple processes use the same name on the host, all of them are killed by this attacker. In this case. This parameter should always set to "neutron- server". 3) host: which is the name of a control node being attacked. e.g. -fault_type: "kill-process" -process_name: "neutron-server" -host: node1
monitors	In this test case, two kinds of monitor are needed: 1. the "openstack-cmd" monitor constantly request a specific Openstack command, which needs two parameters: 1) monitor_type: which is used for finding the monitor class and related scritps. It should be always set to "openstack-cmd" for this monitor. 2) command_name: which is the command name used for request. In this case, the command name should be neutron related commands. 2. the "process" monitor check whether a process is running on a specific node, which needs three parameters: 1) monitor_type: which used for finding the monitor class and related scritps. It should be always set to "process" for this monitor. 2) process_name: which is the process name for monitor 3) host: which is the name of the node runing the process e.g. monitor1: -monitor_type: "openstack-cmd" -command_name: "neutron agent-list" monitor2: -monitor_type: "process" -process_name: "neutron-server" -host: node1
metrics	In this test case, there are two metrics: 1)service_outage_time: which indicates the maximum outage time (seconds) of the specified Openstack command request. 2)process_recover_time: which indicates the maximun time (seconds) from the process being killed to recovered
test tool	Developed by the project. Please see folder: "yardstick/benchmark/scenarios/availability/ha_tools"
references	ETSI NFV REL001
configuration	This test case needs two configuration files: 1) test case file: opnfv_yardstick_tc045.yaml -Attackers: see above "attackers" discription -waiting_time: which is the time (seconds) from the process being killed to stoping monitors the monitors -Monitors: see above "monitors" discription -SLA: see above "metrics" discription 2)POD file: pod.yaml The POD configuration should record on pod.yaml first. the "host" item in this test case will use the node name in the pod.yaml.
test sequence	description and expected result
step 1	start monitors: each monitor will run with independently process Result: The monitor info will be collected.
step 2	do attacker: connect the host through SSH, and then execute the kill process script with param value specified by "process_name" Result: Process will be killed.
step 3	stop monitors after a period of time specified by "waiting_time" Result: The monitor info will be aggregated.
step 4	verify the SLA Result: The test case is passed or not.
post-action	It is the action when the test cases exist. It will check the status of the specified process on the host, and restart the process if it is not running for next test cases. Notice: This post-action uses 'lsb_release' command to check the host linux distribution and determine the OpenStack service name to restart the process. Lack of 'lsb_release' on the host may cause failure to restart

Yardstick Test Case Description TC046

Control Node Openstack Service High Availability - Keystone	
test case id	OPNFV_YARDSTICK_TC046: Control node Openstack service down - keystone
test purpose	This test case will verify the high availability of the user service provided by OpenStack (keystone) on control node.
test method	This test case kills the processes of keystone service on a selected control node, then checks whether the request of the related Openstack command is OK and the killed processes are recovered.
attackers	In this test case, an attacker called "kill-process" is needed. This attacker includes three parameters: 1) fault_type: which is used for finding the attacker's scripts. It should be always set to "kill-process" in this test case. 2) process_name: which is the process name of the specified OpenStack service. If there are multiple processes use the same name on the host, all of them are killed by this attacker. In this case. This parameter should always set to "keystone" 3) host: which is the name of a control node being attacked. e.g. -fault_type: "kill-process" -process_name: "keystone" -host: node1
monitors	In this test case, two kinds of monitor are needed: 1. the "openstack-cmd" monitor constantly request a specific Openstack command, which needs two parameters: 1) monitor_type: which is used for finding the monitor class and related scritps. It should be always set to "openstack-cmd" for this monitor. 2) command_name: which is the command name used for request. In this case, the command name should be keystone related commands. 2. the "process" monitor check whether a process is running on a specific node, which needs three parameters: 1) monitor_type: which used for finding the monitor class and related scritps. It should be always set to "process" for this monitor. 2) process_name: which is the process name for monitor 3) host: which is the name of the node runing the process e.g. monitor1: -monitor_type: "openstack-cmd" -command_name: "keystone user-list" monitor2: -monitor_type: "process" -process_name: "keystone" -host: node1
metrics	In this test case, there are two metrics: 1)service_outage_time: which indicates the maximum outage time (seconds) of the specified Openstack command request. 2)process_recover_time: which indicates the maximun time (seconds) from the process being killed to recovered
test tool	Developed by the project. Please see folder: "yardstick/benchmark/scenarios/availability/ha_tools"
references	ETSI NFV REL001
configuration	This test case needs two configuration files: 1) test case file: opnfv_yardstick_tc046.yaml -Attackers: see above "attackers" discription -waiting_time: which is the time (seconds) from the process being killed to stoping monitors the monitors -Monitors: see above "monitors" discription -SLA: see above "metrics" discription 2)POD file: pod.yaml The POD configuration should record on pod.yaml first. the "host" item in this test case will use the node name in the pod.yaml.
test sequence	description and expected result
step 1	start monitors: each monitor will run with independently process Result: The monitor info will be collected.
step 2	do attacker: connect the host through SSH, and then execute the kill process script with param value specified by "process_name" Result: Process will be killed.
step 3	stop monitors after a period of time specified by "waiting_time" Result: The monitor info will be aggregated.
step 4	verify the SLA Result: The test case is passed or not.
post-action	It is the action when the test cases exist. It will check the status of the specified process on the host, and restart the process if it is not running for next test cases. Notice: This post-action uses 'lsb_release' command to check the host linux distribution and determine the OpenStack service name to restart the process. Lack of 'lsb_release' on the host may cause failure to restart the process.

Yardstick Test Case Description TC047

Control Node Openstack Service High Availability - Glance Api	
test case id	OPNFV_YARDSTICK_TC047: Control node Openstack service down - glance api
test purpose	This test case will verify the high availability of the image service provided by OpenStack (glance-api) on control node.
test method	This test case kills the processes of glance-api service on a selected control node, then checks whether the request of the related Openstack command is OK and the killed processes are recovered.
attackers	In this test case, an attacker called "kill-process" is needed. This attacker includes three parameters: 1) fault_type: which is used for finding the attacker's scripts. It should be always set to "kill-process" in this test case. 2) process_name: which is the process name of the specified OpenStack service. If there are multiple processes use the same name on the host, all of them are killed by this attacker. In this case. This parameter should always set to "glance- api". 3) host: which is the name of a control node being attacked. e.g. -fault_type: "kill-process" -process_name: "glance-api" -host: node1
monitors	In this test case, two kinds of monitor are needed: 1. the "openstack-cmd" monitor constantly request a specific Openstack command, which needs two parameters: 1) monitor_type: which is used for finding the monitor class and related scritps. It should be always set to "openstack-cmd" for this monitor. 2) command_name: which is the command name used for request. In this case, the command name should be glance related commands. 2. the "process" monitor check whether a process is running on a specific node, which needs three parameters: 1) monitor_type: which used for finding the monitor class and related scritps. It should be always set to "process" for this monitor. 2) process_name: which is the process name for monitor 3) host: which is the name of the node runing the process e.g. monitor1: -monitor_type: "openstack-cmd" -command_name: "glance image-list" monitor2: -monitor_type: "process" -process_name: "glance-api" -host: node1
metrics	In this test case, there are two metrics: 1)service_outage_time: which indicates the maximum outage time (seconds) of the specified Openstack command request. 2)process_recover_time: which indicates the maximun time (seconds) from the process being killed to recovered
test tool	Developed by the project. Please see folder: "yardstick/benchmark/scenarios/availability/ha_tools"
references	ETSI NFV REL001
configuration	This test case needs two configuration files: 1) test case file: opnfv_yardstick_tc047.yaml -Attackers: see above "attackers" discription -waiting_time: which is the time (seconds) from the process being killed to stoping monitors the monitors -Monitors: see above "monitors" discription -SLA: see above "metrics" discription 2)POD file: pod.yaml The POD configuration should record on pod.yaml first. the "host" item in this test case will use the node name in the pod.yaml.
test sequence	description and expected result
step 1	start monitors: each monitor will run with independently process Result: The monitor info will be collected.
step 2	do attacker: connect the host through SSH, and then execute the kill process script with param value specified by "process_name" Result: Process will be killed.
step 3	stop monitors after a period of time specified by "waiting_time" Result: The monitor info will be aggregated.
step 4	verify the SLA Result: The test case is passed or not.
post-action	It is the action when the test cases exist. It will check the status of the specified process on the host, and restart the process if it is not running for next test cases. Notice: This post-action uses 'lsb_release' command to check the host linux distribution and determine the OpenStack service name to restart the process. Lack of 'lsb_release' on the host may cause failure to restart the process.

Yardstick Test Case Description TC048

Control Node Openstack Service High Availability - Cinder Api	
test case id	OPNFV_YARDSTICK_TC048: Control node Openstack service down - cinder api
test purpose	This test case will verify the high availability of the volume service provided by OpenStack (cinder-api) on control node.
test method	This test case kills the processes of cinder-api service on a selected control node, then checks whether the request of the related Openstack command is OK and the killed processes are recovered.
attackers	In this test case, an attacker called "kill-process" is needed. This attacker includes three parameters: 1) fault_type: which is used for finding the attacker's scripts. It should be always set to "kill-process" in this test case. 2) process_name: which is the process name of the specified OpenStack service. If there are multiple processes use the same name on the host, all of them are killed by this attacker. In this case. This parameter should always set to "cinder- api". 3) host: which is the name of a control node being attacked. e.g. -fault_type: "kill-process" -process_name: "cinder-api" -host: node1
monitors	In this test case, two kinds of monitor are needed: 1. the "openstack-cmd" monitor constantly request a specific Openstack command, which needs two parameters: 1) monitor_type: which is used for finding the monitor class and related scritps. It should be always set to "openstack-cmd" for this monitor. 2) command_name: which is the command name used for request. In this case, the command name should be cinder related commands. 2. the "process" monitor check whether a process is running on a specific node, which needs three parameters: 1) monitor_type: which used for finding the monitor class and related scritps. It should be always set to "process" for this monitor. 2) process_name: which is the process name for monitor 3) host: which is the name of the node runing the process e.g. monitor1: -monitor_type: "openstack-cmd" -command_name: "cinder list" monitor2: -monitor_type: "process" -process_name: "cinder-api" -host: node1
metrics	In this test case, there are two metrics: 1)service_outage_time: which indicates the maximum outage time (seconds) of the specified Openstack command request. 2)process_recover_time: which indicates the maximun time (seconds) from the process being killed to recovered
test tool	Developed by the project. Please see folder: "yardstick/benchmark/scenarios/availability/ha_tools"
references	ETSI NFV REL001
configuration	This test case needs two configuration files: 1) test case file: opnfv_yardstick_tc048.yaml -Attackers: see above "attackers" discription -waiting_time: which is the time (seconds) from the process being killed to stoping monitors the monitors -Monitors: see above "monitors" discription -SLA: see above "metrics" discription 2)POD file: pod.yaml The POD configuration should record on pod.yaml first. the "host" item in this test case will use the node name in the pod.yaml.
test sequence	description and expected result
step 1	start monitors: each monitor will run with independently process Result: The monitor info will be collected.
step 2	do attacker: connect the host through SSH, and then execute the kill process script with param value specified by "process_name" Result: Process will be killed.
step 3	stop monitors after a period of time specified by "waiting_time" Result: The monitor info will be aggregated.
step 4	verify the SLA Result: The test case is passed or not.
post-action	It is the action when the test cases exist. It will check the status of the specified process on the host, and restart the process if it is not running for next test case Notice: This post-action uses 'lsb_release' command to check the host linux distribution and determine the OpenStack service name to restart the process. Lack of 'lsb_release' on the host may cause failure to restart the process.

Yardstick Test Case Description TC049

Control Node Openstack Service High Availability - Swift Proxy	
test case id	OPNFV_YARDSTICK_TC049: Control node Openstack service down - swift proxy
test purpose	This test case will verify the high availability of the storage service provided by OpenStack (swift-proxy) on control node.
test method	This test case kills the processes of swift-proxy service on a selected control node, then checks whether the request of the related Openstack command is OK and the killed processes are recovered.
attackers	In this test case, an attacker called "kill-process" is needed. This attacker includes three parameters: 1) fault_type: which is used for finding the attacker's scripts. It should be always set to "kill-process" in this test case. 2) process_name: which is the process name of the specified OpenStack service. If there are multiple processes use the same name on the host, all of them are killed by this attacker. In this case. This parameter should always set to "swift- proxy". 3) host: which is the name of a control node being attacked. e.g. -fault_type: "kill-process" -process_name: "swift-proxy" -host: node1
monitors	In this test case, two kinds of monitor are needed: 1. the "openstack-cmd" monitor constantly request a specific Openstack command, which needs two parameters: 1) monitor_type: which is used for finding the monitor class and related scritps. It should be always set to "openstack-cmd" for this monitor. 2) command_name: which is the command name used for request. In this case, the command name should be swift related commands. 2. the "process" monitor check whether a process is running on a specific node, which needs three parameters: 1) monitor_type: which used for finding the monitor class and related scritps. It should be always set to "process" for this monitor. 2) process_name: which is the process name for monitor 3) host: which is the name of the node runing the process e.g. monitor1: -monitor_type: "openstack-cmd" -command_name: "swift stat" monitor2: -monitor_type: "process" -process_name: "swift-proxy" -host: node1
metrics	In this test case, there are two metrics: 1)service_outage_time: which indicates the maximum outage time (seconds) of the specified Openstack command request. 2)process_recover_time: which indicates the maximun time (seconds) from the process being killed to recovered
test tool	Developed by the project. Please see folder: "yardstick/benchmark/scenarios/availability/ha_tools"
references	ETSI NFV REL001
configuration	This test case needs two configuration files: 1) test case file: opnfv_yardstick_tc049.yaml -Attackers: see above "attackers" discription -waiting_time: which is the time (seconds) from the process being killed to stoping monitors the monitors -Monitors: see above "monitors" discription -SLA: see above "metrics" discription 2)POD file: pod.yaml The POD configuration should record on pod.yaml first. the "host" item in this test case will use the node name in the pod.yaml.
test sequence	description and expected result
step 1	start monitors: each monitor will run with independently process Result: The monitor info will be collected.
step 2	do attacker: connect the host through SSH, and then execute the kill process script with param value specified by "process_name" Result: Process will be killed.
step 3	stop monitors after a period of time specified by "waiting_time" Result: The monitor info will be aggregated.
step 4	verify the SLA Result: The test case is passed or not.
post-action	It is the action when the test cases exist. It will check the status of the specified process on the host, and restart the process if it is not running for next test cases. Notice: This post-action uses 'lsb_release' command to check the host linux distribution and determine the OpenStack service name to restart the process. Lack of 'lsb_release' on the host may cause failure to restart the process.

Yardstick Test Case Description TC050

OpenStack Controller Node Network High Availability	
test case id	OPNFV_YARDSTICK_TC050: OpenStack Controller Node Network High Availability
test purpose	This test case will verify the high availability of control node. When one of the controller failed to connect the network, which breaks down the Openstack services on this node. These Openstack service should able to be accessed by other controller nodes, and the services on failed controller node should be isolated.
test method	This test case turns off the network interfaces of a specified control node, then checks whether all services provided by the control node are OK with some monitor tools.

attackers	In this test case, an attacker called "close-interface" is needed. This attacker includes three parameters: 1) fault_type: which is used for finding the attacker's scripts. It should be always set to "close-interface" in this test case. 2) host: which is the name of a control node being attacked. 3) interface: the network interface to be turned off.
	The interface to be closed by the attacker can be set by the variable of "{{ interface_name }}":
	`attackers:`
	`  -`
	`    fault_type: "general-attacker"`
	`    host: {{ attack_host }}`
	`    key: "close-br-public"`
	`    attack_key: "close-interface"`
	`    action_parameter:`
	`      interface: {{ interface_name }}`
	`    rollback_parameter:`
	`      interface: {{ interface_name }}`

monitors	In this test case, the monitor named "openstack-cmd" is needed. The monitor needs needs two parameters: 1) monitor_type: which is used for finding the monitor class and related scritps. It should be always set to "openstack-cmd" for this monitor. 2) command_name: which is the command name used for request
	There are four instance of the "openstack-cmd" monitor:
	`monitor1:`
	`    - monitor_type: "openstack-cmd"`
	`    - command_name: "nova image-list"`
	`monitor2:`
	`    - monitor_type: "openstack-cmd"`
	`    - command_name: "neutron router-list"`
	`monitor3:`
	`    - monitor_type: "openstack-cmd"`
	`    - command_name: "heat stack-list"`
	`monitor4:`
	`    - monitor_type: "openstack-cmd"`
	`    - command_name: "cinder list"`

metrics	In this test case, there is one metric: User-vice_outage_time: which indicates the maximum outage time (seconds) of the specified Openstack command request.
test tool	Developed by the project. Please see folder: "yard-

Yardstick Test Case Description TC051

OpenStack Controller Node CPU Overload High Availability	
test case id	OPNFV_YARDSTICK_TC051: OpenStack Controller Node CPU Overload High Availability
test purpose	This test case will verify the high availability of control node. When the CPU usage of a specified controller node is stressed to 100%, which breaks down the Openstack services on this node. These Openstack service should able to be accessed by other controller nodes, and the services on failed controller node should be isolated.
test method	This test case stresses the CPU uasge of a specified control node to 100%, then checks whether all services provided by the environment are OK with some monitor tools.
attackers	In this test case, an attacker called "stress-cpu" is needed. This attacker includes two parameters: 1) fault_type: which is used for finding the attacker's scripts. It should be always set to "stress-cpu" in this test case. 2) host: which is the name of a control node being attacked. e.g. -fault_type: "stress-cpu" -host: node1
monitors	In this test case, the monitor named "openstack-cmd" is needed. The monitor needs needs two parameters: 1) monitor_type: which is used for finding the monitor class and related scritps. It should be always set to "openstack-cmd" for this monitor. 2) command_name: which is the command name used for request There are four instance of the "openstack-cmd" monitor: monitor1: -monitor_type: "openstack-cmd" -command_name: "nova image-list" monitor2: -monitor_type: "openstack-cmd" -command_name: "neutron router-list" monitor3: -monitor_type: "openstack-cmd" -command_name: "heat stack-list" monitor4: -monitor_type: "openstack-cmd" -command_name: "cinder list"
metrics	In this test case, there is one metric: 1)service_outage_time: which indicates the maximum outage time (seconds) of the specified Openstack command request.
test tool	Developed by the project. Please see folder: "yardstick/benchmark/scenarios/availability/ha_tools"
references	ETSI NFV REL001
configuration	This test case needs two configuration files: 1) test case file: opnfv_yardstick_tc051.yaml -Attackers: see above "attackers" discription -waiting_time: which is the time (seconds) from the process being killed to stoping monitors the monitors -Monitors: see above "monitors" discription -SLA: see above "metrics" discription 2)POD file: pod.yaml The POD configuration should record on pod.yaml first. the "host" item in this test case will use the node name in the pod.yaml.
test sequence	description and expected result
step 1	start monitors: each monitor will run with independently process Result: The monitor info will be collected.
step 2	do attacker: connect the host through SSH, and then execute the stress cpu script on the host. Result: The CPU usage of the host will be stressed to 100%.
step 3	stop monitors after a period of time specified by "waiting_time" Result: The monitor info will be aggregated.
step 4	verify the SLA Result: The test case is passed or not.
post-action	It is the action when the test cases exist. It kills the process that stresses the CPU usage.
test verdict	Fails only if SLA is not passed, or if there is a test case execution problem.

Yardstick Test Case Description TC052

OpenStack Controller Node Disk I/O Block High Availability	
test case id	OPNFV_YARDSTICK_TC052: OpenStack Controller Node Disk I/O Block High Availability
test purpose	This test case will verify the high availability of control node. When the disk I/O of a specified disk is blocked, which breaks down the Openstack services on this node. Read and write services should still be accessed by other controller nodes, and the services on failed controller node should be isolated.
test method	This test case blocks the disk I/O of a specified control node, then checks whether the services that need to read or wirte the disk of the control node are OK with some monitor tools.
attackers	In this test case, an attacker called "disk-block" is needed. This attacker includes two parameters: 1) fault_type: which is used for finding the attacker's scripts. It should be always set to "disk-block" in this test case. 2) host: which is the name of a control node being attacked. e.g. -fault_type: "disk-block" -host: node1

monitors	In this test case, two kinds of monitor are needed: 1. the "openstack-cmd" monitor constantly request a specific Openstack command, which needs two parameters: 1) monitor_type: which is used for finding the monitor class and related scripts. It should be always set to "openstack-cmd" for this monitor. 2) command_name: which is the command name used for request. e.g. -monitor_type: "openstack-cmd" -command_name: "nova flavor-list" 2. the second monitor verifies the read and write function by a "operation" and a "result checker". the "operation" have two parameters: 1) operation_type: which is used for finding the operation class and related scripts. 2) action_parameter: parameters for the operation. the "result checker" have three parameters: 1) checker_type: which is used for finding the reuslt checker class and realted scripts. 2) expectedValue: the expected value for the output of the checker script. 3) condition: whether the expected value is in the output of checker script or is totally same with the output. In this case, the "operation" adds a flavor and the "result checker" checks whether ths flavor is created. Their parameters show as follows:
	`operation:`
	`-operation_type: "nova-create-flavor"`
	`-action_parameter:`
	`flavorconfig: "test-001 test-001 100 1 1"`
	`result checker:`
	`-checker_type: "check-flavor"`
	`-expectedValue: "test-001"`
	`-condition: "in"`

metrics	In this test case, there is one metric: 1)service_outage_time: which indicates the maximum outage time (seconds) of the specified Openstack command request.
test tool	Developed by the project. Please see folder: "yard-

Yardstick Test Case Description TC053

OpenStack Controller Load Balance Service High Availability	
test case id	OPNFV_YARDSTICK_TC053: OpenStack Controller Load Balance Service High Availability
test purpose	This test case will verify the high availability of the load balance service(current is HAProxy) that supports OpenStack on controller node. When the load balance service of a specified controller node is killed, whether other load balancers on other controller nodes will work, and whether the controller node will restart the load balancer are checked.
test method	This test case kills the processes of load balance service on a selected control node, then checks whether the request of the related Openstack command is OK and the killed processes are recovered.
attackers	In this test case, an attacker called "kill-process" is needed. This attacker includes three parameters: 1) fault_type: which is used for finding the attacker's scripts. It should be always set to "kill-process" in this test case. 2) process_name: which is the process name of the specified OpenStack service. If there are multiple processes use the same name on the host, all of them are killed by this attacker. In this case. This parameter should always set to "swift- proxy". 3) host: which is the name of a control node being attacked. e.g. -fault_type: "kill-process" -process_name: "haproxy" -host: node1
monitors	In this test case, two kinds of monitor are needed: 1. the "openstack-cmd" monitor constantly request a specific Openstack command, which needs two parameters: 1) monitor_type: which is used for finding the monitor class and related scritps. It should be always set to "openstack-cmd" for this monitor. 2) command_name: which is the command name used for request. 2. the "process" monitor check whether a process is running on a specific node, which needs three parameters: 1) monitor_type: which used for finding the monitor class and related scripts. It should be always set to "process" for this monitor. 2) process_name: which is the process name for monitor 3) host: which is the name of the node runing the process In this case, the command_name of monitor1 should be services that is supported by load balancer and the process- name of monitor2 should be "haproxy", for example: e.g. monitor1: -monitor_type: "openstack-cmd" -command_name: "nova image-list" monitor2: -monitor_type: "process" -process_name: "haproxy" -host: node1
metrics	In this test case, there are two metrics: 1)service_outage_time: which indicates the maximum outage time (seconds) of the specified Openstack command request. 2)process_recover_time: which indicates the maximun time (seconds) from the process being killed to recovered
test tool	Developed by the project. Please see folder: "yardstick/benchmark/scenarios/availability/ha_tools"
references	ETSI NFV REL001
configuration	This test case needs two configuration files: 1) test case file: opnfv_yardstick_tc053.yaml -Attackers: see above "attackers" discription -waiting_time: which is the time (seconds) from the process being killed to stoping monitors the monitors -Monitors: see above "monitors" discription -SLA: see above "metrics" discription 2)POD file: pod.yaml The POD configuration should record on pod.yaml first. the "host" item in this test case will use the node name in the pod.yaml.
test sequence	description and expected result
step 1	start monitors: each monitor will run with independently process Result: The monitor info will be collected.
step 2	do attacker: connect the host through SSH, and then execute the kill process script with param value specified by "process_name" Result: Process will be killed.
step 3	stop monitors after a period of time specified by "waiting_time" Result: The monitor info will be aggregated.
step 4	verify the SLA Result: The test case is passed or not.
post-action	It is the action when the test cases exist. It will check the status of the specified process on the host, and restart the process if it is not running for next test cases. Notice: This post-action uses 'lsb_release' command to check the host linux distribution and determine the OpenStack service name to restart the process. Lack of 'lsb_release' on the host may cause failure to restart

Yardstick Test Case Description TC054

OpenStack Virtual IP High Availability	
test case id	OPNFV_YARDSTICK_TC054: OpenStack Virtual IP High Availability
test purpose	This test case will verify the high availability for virtual ip in the environment. When master node of virtual ip is abnormally shutdown, connection to virtual ip and the services binded to the virtual IP it should be OK.
test method	This test case shutdowns the virtual IP master node with some fault injection tools, then checks whether virtual ips can be pinged and services binded to virtual ip are OK with some monitor tools.
attackers	In this test case, an attacker called "control-shutdown" is needed. This attacker includes two parameters: 1) fault_type: which is used for finding the attacker's scripts. It should be always set to "control-shutdown" in this test case. 2) host: which is the name of a control node being attacked. In this case the host should be the virtual ip master node, that means the host ip is the virtual ip, for exapmle: -fault_type: "control-shutdown" -host: node1(the VIP Master node)
monitors	In this test case, two kinds of monitor are needed: 1. the "ip_status" monitor that pings a specific ip to check the connectivity of this ip, which needs two parameters: 1) monitor_type: which is used for finding the monitor class and related scripts. It should be always set to "ip_status" for this monitor. 2) ip_address: The ip to be pinged. In this case, ip_address should be the virtual IP. 2. the "openstack-cmd" monitor constantly request a specific Openstack command, which needs two parameters: 1) monitor_type: which is used for finding the monitor class and related scripts. It should be always set to "openstack-cmd" for this monitor. 2) command_name: which is the command name used for request. e.g. monitor1: -monitor_type: "ip_status" -host: 192.168.0.2 monitor2: -monitor_type: "openstack-cmd" -command_name: "nova image-list"
metrics	In this test case, there are two metrics: 1) ping_outage_time: which-indicates the maximum outage time to ping the specified host. 2)service_outage_time: which indicates the maximum outage time (seconds) of the specified Openstack command request.
test tool	Developed by the project. Please see folder: "yardstick/benchmark/scenarios/availability/ha_tools"
references	ETSI NFV REL001
configuration	This test case needs two configuration files: 1) test case file: opnfv_yardstick_tc054.yaml -Attackers: see above "attackers" discription -waiting_time: which is the time (seconds) from the process being killed to stoping monitors the monitors -Monitors: see above "monitors" discription -SLA: see above "metrics" discription 2)POD file: pod.yaml The POD configuration should record on pod.yaml first. the "host" item in this test case will use the node name in the pod.yaml.
test sequence	description and expected result
step 1	start monitors: each monitor will run with independently process Result: The monitor info will be collected.
step 2	do attacker: connect the host through SSH, and then execute the shutdown script on the VIP master node. Result: VIP master node will be shutdown
step 3	stop monitors after a period of time specified by "waiting_time" Result: The monitor info will be aggregated.
step 4	verify the SLA Result: The test case is passed or not.
post-action	It is the action when the test cases exist. It restarts the original VIP master node if it is not restarted.
test verdict	Fails only if SLA is not passed, or if there is a test case execution problem.

Yardstick Test Case Description TC056

OpenStack Controller Messaging Queue Service High Availability	
test case id	OPNFV_YARDSTICK_TC056:OpenStack Controller Messaging Queue Service High Availability
test purpose	This test case will verify the high availability of the messaging queue service(RabbitMQ) that supports OpenStack on controller node. When messaging queue service(which is active) of a specified controller node is killed, the test case will check whether messaging queue services(which are standby) on other controller nodes will be switched active, and whether the cluster manager on attacked the controller node will restart the stopped messaging queue.
test method	This test case kills the processes of messaging queue service on a selected controller node, then checks whether the request of the related Openstack command is OK and the killed processes are recovered.
attackers	In this test case, an attacker called "kill-process" is needed. This attacker includes three parameters: 1) fault_type: which is used for finding the attacker's scripts. It should be always set to "kill-process" in this test case. 2) process_name: which is the process name of the specified OpenStack service. If there are multiple processes use the same name on the host, all of them are killed by this attacker. In this case, this parameter should always set to "rabbitmq". 3) host: which is the name of a control node being attacked. e.g. -fault_type: "kill-process" -process_name: "rabbitmq-server" -host: node1
monitors	In this test case, two kinds of monitor are needed: 1. the "openstack-cmd" monitor constantly request a specific Openstack command, which needs two parameters: 1) monitor_type: which is used for finding the monitor class and related scritps. It should be always set to "openstack-cmd" for this monitor. 2) command_name: which is the command name used for request. 2. the "process" monitor check whether a process is running on a specific node, which needs three parameters: 1) monitor_type: which used for finding the monitor class and related scripts. It should be always set to "process" for this monitor. 2) process_name: which is the process name for monitor 3) host: which is the name of the node runing the process In this case, the command_name of monitor1 should be services that will use the messaging queue(current nova, neutron, cinder ,heat and ceilometer are using RabbitMQ) , and the process-name of monitor2 should be "rabbitmq", for example: e.g. monitor1-1: -monitor_type: "openstack-cmd" -command_name: "openstack image list" monitor1-2: -monitor_type: "openstack-cmd" -command_name: "openstack network list" monitor1-3: -monitor_type: "openstack-cmd" -command_name: "openstack volume list" monitor2: -monitor_type: "process" -process_name: "rabbitmq" -host: node1
metrics	In this test case, there are two metrics: 1)service_outage_time: which indicates the maximum outage time (seconds) of the specified Openstack command request. 2)process_recover_time: which indicates the maximum time (seconds) from the process being killed to recovered
test tool	Developed by the project. Please see folder: "yardstick/benchmark/scenarios/availability/ha_tools"
references	ETSI NFV REL001
configuration	This test case needs two configuration files: 1) test case file: opnfv_yardstick_tc056.yaml -Attackers: see above "attackers" description -waiting_time: which is the time (seconds) from the process being killed to stoping monitors the monitors -Monitors: see above "monitors" description -SLA: see above "metrics" description 2)POD file: pod.yaml The POD configuration should record on pod.yaml first. the "host" item in this test case will use the node name in the pod.yaml.
test sequence	description and expected result
step 1	start monitors: each monitor will run with independently process Result: The monitor info will be collected.
step 2	do attacker: connect the host through SSH, and then execute the kill process script with param value specified by "process_name" Result: Process will be killed.
step 3	stop monitors after a period of time specified by "waiting_time" Result: The monitor info will be aggregated.
step 4	verify the SLA Result: The test case is passed or not.

Yardstick Test Case Description TC057

OpenStack Controller Cluster Management Service High Availability	
test case id	OPNFV_YARDSTICK_TC057_HA: OpenStack Controller Cluster Management Service High Availability
test purpose	This test case will verify the quorum configuration of the cluster manager(pacemaker) on controller nodes. When a controller node , which holds all active application resources, failed to communicate with other cluster nodes (via corosync), the test case will check whether the standby application resources will take place of those active application resources which should be regarded to be down in the cluster manager.
test method	This test case kills the processes of cluster messaging service(corosync) on a selected controller node(the node holds the active application resources), then checks whether active application resources are switched to other controller nodes and whether the Openstack commands are OK.
attackers	In this test case, an attacker called "kill-process" is needed. This attacker includes three parameters: 1) fault_type: which is used for finding the attacker's scripts. It should be always set to "kill-process" in this test case. 2) process_name: which is the process name of the load balance service. If there are multiple processes use the same name on the host, all of them are killed by this attacker. 3) host: which is the name of a control node being attacked. In this case, this process name should set to "corosync" , for example -fault_type: "kill-process" -process_name: "corosync" -host: node1
monitors	In this test case, a kind of monitor is needed: 1. the "openstack-cmd" monitor constantly request a specific Openstack command, which needs two parameters: 1. monitor_type: which is used for finding the monitor class and related scripts. It should be always set to "openstack-cmd" for this monitor. 2. command_name: which is the command name used for request In this case, the command_name of monitor1 should be services that are managed by the cluster manager. (Since rabbitmq and haproxy are managed by pacemaker, most Openstack Services can be used to check high availability in this case) (e.g.) monitor1: -monitor_type: "openstack-cmd" -command_name: "nova image-list" monitor2: -monitor_type: "openstack-cmd" -command_name: "neutron router-list" monitor3: -monitor_type: "openstack-cmd" -command_name: "heat stack-list" monitor4: -monitor_type: "openstack-cmd" -command_name: "cinder list"
checkers	In this test case, a checker is needed, the checker will the status of application resources in pacemaker and the checker have three parameters: 1) checker_type: which is used for finding the result checker class and related scripts. In this case the checker type will be "pacemaker-check-resource" 2) resource_name: the application resource name 3) resource_status: the expected status of the resource 4) expectedValue: the expected value for the output of the checker script, in the case the expected value will be the identifier in the cluster manager 3) condition: whether the expected value is in the output of checker script or is totally same with the output. (note: pcs is required to installed on controller node in order to run this checker) (e.g.) checker1: -checker_type: "pacemaker-check-resource" -resource_name: "p_rabbitmq-server" -resource_status: "Stopped" -expectedValue: "node-1" -condition: "in" checker2: -checker_type: "pacemaker-check-resource" -resource_name: "p_rabbitmq-server" -resource_status: "Master" -expectedValue: "node-2" -condition: "in"
metrics	In this test case, there are two metrics: 1)service_outage_time: which indicates the maximum outage time (seconds) of the specified Openstack command request.
test tool	None. Self-developed.
references	ETSI NFV REL001
configuration	This test case needs two configuration files: 1) test case file: opnfv_yardstick_tc057.yaml -Attackers: see above "attackers" description -Monitors: see above "monitors" description -Checkers: see above "checkers" description -Steps: the test case execution step, see "test sequence" description below 2)POD file: pod.yaml The POD configuration should record on pod.yaml first. the "host" item in this test case will use the node name in the pod.yaml.

Yardstick Test Case Description TC058

OpenStack Controller Virtual Router Service High Availability	
test case id	OPNFV_YARDSTICK_TC058: OpenStack Controller Virtual Router Service High Availability
test purpose	This test case will verify the high availability of virtual routers(L3 agent) on controller node. When a virtual router service on a specified controller node is shut down, this test case will check whether the network of virtual machines will be affected, and whether the attacked virtual router service will be recovered.
test method	This test case kills the processes of virtual router service (l3-agent) on a selected controller node(the node holds the active l3-agent), then checks whether the network routing of virtual machines is OK and whether the killed service will be recovered.
attackers	In this test case, an attacker called "kill-process" is needed. This attacker includes three parameters: 1) fault_type: which is used for finding the attacker's scripts. It should be always set to "kill-process" in this test case. 2) process_name: which is the process name of the load balance service. If there are multiple processes use the same name on the host, all of them are killed by this attacker. 3) host: which is the name of a control node being attacked. In this case, this process name should set to "l3agent" , for example -fault_type: "kill-process" -process_name: "l3agent" -host: node1
monitors	In this test case, two kinds of monitor are needed: 1. the "ip_status" monitor that pings a specific ip to check the connectivity of this ip, which needs two parameters: 1) monitor_type: which is used for finding the monitor class and related scripts. It should be always set to "ip_status" for this monitor. 2) ip_address: The ip to be pinged. In this case, ip_address will be either an ip address of external network or an ip address of a virtual machine. 3) host: The node on which ping will be executed, in this case the host will be a virtual machine. 2. the "process" monitor check whether a process is running on a specific node, which needs three parameters: 1) monitor_type: which used for finding the monitor class and related scripts. It should be always set to "process" for this monitor. 2) process_name: which is the process name for monitor. In this case, the process-name of monitor2 should be "l3agent" 3) host: which is the name of the node running the process e.g. monitor1-1: -monitor_type: "ip_status" -host: 172.16.0.11 -ip_address: 172.16.1.11 monitor1-2: -monitor_type: "ip_status" -host: 172.16.0.11 -ip_address: 8.8.8.8 monitor2: -monitor_type: "process" -process_name: "l3agent" -host: node1
metrics	In this test case, there are two metrics: 1)service_outage_time: which indicates the maximum outage time (seconds) of the specified Openstack command request. 2)process_recover_time: which indicates the maximum time (seconds) from the process being killed to recovered
test tool	None. Self-developed.
references	ETSI NFV REL001
configuration	This test case needs two configuration files: 1) test case file: opnfv_yardstick_tc058.yaml -Attackers: see above "attackers" description -Monitors: see above "monitors" description -Steps: the test case execution step, see "test sequence" description below 2)POD file: pod.yaml The POD configuration should record on pod.yaml first. the "host" item in this test case will use the node name in the pod.yaml.
test sequence	description and expected result
pre-test conditions	The test case image needs to be installed into Glance with cachestat included in the image.
step 1	Two host VMs are booted, these two hosts are in two different networks, the networks are connected by a virtual router.
step 1	start monitors: each monitor will run with independently process Result: The monitor info will be collected.
step 2	do attacker: connect the host through SSH, and then execute the kill process script with param value specified by "process_name"

Yardstick Test Case Description TC087

SDN Controller resilience in non-HA configuration	
test case id	OPNFV_YARDSTICK_TC087: SDN controller resilience in non-HA configuration
test purpose	This test validates that network data plane services are highly available in the event of an SDN Controller failure, even if the SDN controller is deployed in a non-HA configuration. Specifically, the test verifies that existing data plane connectivity is not impacted, i.e. all configured network services such as DHCP, ARP, L2, L3 Security Groups should continue to operate between the existing VMs while the SDN controller is offline or rebooting. The test also validates that new network service operations (creating a new VM in the existing L2/L3 network or in a new network, etc.) are operational after the SDN controller has recovered from a failure.
test method	This test case fails the SDN controller service running on the OpenStack controller node, then checks if already configured DHCP/ARP/L2/L3/SNAT connectivity is not impacted between VMs and the system is able to execute new virtual network operations once the SDN controller is restarted and has fully recovered
attackers	In this test case, an attacker called "kill-process" is needed. This attacker includes three parameters: 1. fault_type: which is used for finding the attacker's scripts. It should be set to 'kill-process' in this test 2. process_name: should be set to the name of the SDN controller process 3. host: which is the name of a control node where the SDN controller process is running **e.g. -fault_type: "kill-process"** -process_name: "opendaylight" -host: node1
monitors	This test case utilizes two monitors of type "ip-status" and one monitor of type "process" to track the following conditions: 1. "ping_same_network_l2": monitor ICMP traffic between VMs in the same Neutron network 2. "ping_external_snat": monitor ICMP traffic from VMs to an external host on the Internet to verify SNAT functionality. 3. "SDN controller process monitor": a monitor checking the state of a specified SDN controller process. It measures the recovery time of the given process. Monitors of type "ip-status" use the "ping" utility to verify reachability of a given target IP.
operations	In this test case, the following operations are needed: 1. "nova-create-instance-in_network": create a VM instance in one of the existing Neutron network.
metrics	In this test case, there are two metrics: 1. process_recover_time: which indicates the maximun time (seconds) from the process being killed to recovered 2. packet_drop: measure the packets that have been

Yardstick Test Case Description TC088

Control Node Openstack Service High Availability - Nova Scheduler	
test case id	OPNFV_YARDSTICK_TC088: Control node Openstack service down - nova scheduler
test purpose	This test case will verify the high availability of the compute scheduler service provided by OpenStack (nova- scheduler) on control node.
test method	This test case kills the processes of nova-scheduler service on a selected control node, then checks whether the request of the related OpenStack command is OK and the killed processes are recovered.
attackers	In this test case, an attacker called "kill-process" is needed. This attacker includes three parameters: 1) fault_type: which is used for finding the attacker's scripts. It should be always set to "kill-process" in this test case. 2) process_name: which is the process name of the specified OpenStack service. If there are multiple processes use the same name on the host, all of them are killed by this attacker. In this case. This parameter should always set to "nova- scheduler". 3) host: which is the name of a control node being attacked. e.g. -fault_type: "kill-process" -process_name: "nova-scheduler" -host: node1
monitors	In this test case, one kind of monitor is needed: 1. the "process" monitor check whether a process is running on a specific node, which needs three parameters: 1) monitor_type: which used for finding the monitor class and related scripts. It should be always set to "process" for this monitor. 2) process_name: which is the process name for monitor 3) host: which is the name of the node running the process e.g. monitor: -monitor_type: "process" -process_name: "nova-scheduler" -host: node1
operations	In this test case, the following operations are needed: 1. "nova-create-instance": create a VM instance to check whether the nova-scheduler works normally.
metrics	In this test case, there are one metric: 1)process_recover_time: which indicates the maximum time (seconds) from the process being killed to recovered
test tool	Developed by the project. Please see folder: "yardstick/benchmark/scenarios/availability/ha_tools"
references	ETSI NFV REL001
configuration	This test case needs two configuration files: 1) test case file: opnfv_yardstick_tc088.yaml -Attackers: see above "attackers" description -waiting_time: which is the time (seconds) from the process being killed to stopping monitors the monitors -Monitors: see above "monitors" description -SLA: see above "metrics" description 2)POD file: pod.yaml The POD configuration should record on pod.yaml first. the "host" item in this test case will use the node name in the pod.yaml.
test sequence	description and expected result
step 1	do attacker: connect the host through SSH, and then execute the kill process script with param value specified by "process_name" Result: Process will be killed.
step 2	start monitors: each monitor will run with independently process Result: The monitor info will be collected.
step 3	create a new instance to check whether the nova scheduler works normally.
step 4	stop the monitor after a period of time specified by "waiting_time" Result: The monitor info will be aggregated.
post-action	It is the action when the test cases exist. It will check the status of the specified process on the host, and restart the process if it is not running for next test cases
test verdict	Fails only if SLA is not passed, or if there is a test case execution problem.

Yardstick Test Case Description TC089

Control Node Openstack Service High Availability - Nova Conductor	
test case id	OPNFV_YARDSTICK_TC089: Control node Openstack service down - nova conductor
test purpose	This test case will verify the high availability of the compute database proxy service provided by OpenStack (nova- conductor) on control node.
test method	This test case kills the processes of nova-conductor service on a selected control node, then checks whether the request of the related OpenStack command is OK and the killed processes are recovered.
attackers	In this test case, an attacker called "kill-process" is needed. This attacker includes three parameters: 1) fault_type: which is used for finding the attacker's scripts. It should be always set to "kill-process" in this test case. 2) process_name: which is the process name of the specified OpenStack service. If there are multiple processes use the same name on the host, all of them are killed by this attacker. In this case. This parameter should always set to "nova- conductor". 3) host: which is the name of a control node being attacked. e.g. -fault_type: "kill-process" -process_name: "nova-conductor" -host: node1
monitors	In this test case, one kind of monitor is needed: 1. the "process" monitor check whether a process is running on a specific node, which needs three parameters: 1) monitor_type: which used for finding the monitor class and related scripts. It should be always set to "process" for this monitor. 2) process_name: which is the process name for monitor 3) host: which is the name of the node running the process e.g. monitor: -monitor_type: "process" -process_name: "nova-conductor" -host: node1
operations	In this test case, the following operations are needed: 1. "nova-create-instance": create a VM instance to check whether the nova-conductor works normally.
metrics	In this test case, there are one metric: 1)process_recover_time: which indicates the maximum time (seconds) from the process being killed to recovered
test tool	Developed by the project. Please see folder: "yardstick/benchmark/scenarios/availability/ha_tools"
references	ETSI NFV REL001
configuration	This test case needs two configuration files: 1) test case file: opnfv_yardstick_tc089.yaml -Attackers: see above "attackers" description -waiting_time: which is the time (seconds) from the process being killed to stopping monitors the monitors -Monitors: see above "monitors" description -SLA: see above "metrics" description 2)POD file: pod.yaml The POD configuration should record on pod.yaml first. the "host" item in this test case will use the node name in the pod.yaml.
test sequence	description and expected result
step 1	do attacker: connect the host through SSH, and then execute the kill process script with param value specified by "process_name" Result: Process will be killed.
step 2	start monitors: each monitor will run with independently process Result: The monitor info will be collected.
step 3	create a new instance to check whether the nova conductor works normally.
step 4	stop the monitor after a period of time specified by "waiting_time" Result: The monitor info will be aggregated.
post-action	It is the action when the test cases exist. It will check the status of the specified process on the host, and restart the process if it is not running for next test cases
test verdict	Fails only if SLA is not passed, or if there is a test case execution problem.

Yardstick Test Case Description TC090

Control Node OpenStack Service High Availability - Database Instances	
test case id	OPNFV_YARDSTICK_TC090: Control node OpenStack service down - database instances
test purpose	This test case will verify the high availability of the data base instances used by OpenStack (mysql) on control node.
test method	This test case kills the processes of database service on a selected control node, then checks whether the request of the related OpenStack command is OK and the killed processes are recovered.
attackers	In this test case, an attacker called "kill-process" is needed. This attacker includes three parameters: 1) fault_type: which is used for finding the attacker's scripts. It should be always set to "kill-process" in this test case. 2) process_name: which is the process name of the specified OpenStack service. If there are multiple processes use the same name on the host, all of them are killed by this attacker. In this case. This parameter should always set to the name of the database service of OpenStack. 3) host: which is the name of a control node being attacked. e.g. -fault_type: "kill-process" -process_name: "mysql" -host: node1
monitors	In this test case, two kinds of monitor are needed: 1. the "openstack-cmd" monitor constantly request a specific Openstack command, which needs two parameters: 1) monitor_type: which is used for finding the monitor class and related scritps. It should be always set to "openstack-cmd" for this monitor. 2) command_name: which is the command name used for request. In this case, the command name should be neutron related commands. 2. the "process" monitor check whether a process is running on a specific node, which needs three parameters: 1) monitor_type: which used for finding the monitor class and related scripts. It should be always set to "process" for this monitor. 2) process_name: which is the process name for monitor 3) host: which is the name of the node running the process The examples of monitors show as follows, there are four instance of the "openstack-cmd" monitor, in order to check the database connection of different OpenStack components. monitor1: -monitor_type: "openstack-cmd" -api_name: "openstack image list" monitor2: -monitor_type: "openstack-cmd" -api_name: "openstack router list" monitor3: -monitor_type: "openstack-cmd" -api_name: "openstack stack list" monitor4: -monitor_type: "openstack-cmd" -api_name: "openstack volume list" monitor5: -monitor_type: "process" -process_name: "mysql" -host: node1
metrics	In this test case, there are two metrics: 1)service_outage_time: which indicates the maximum outage time (seconds) of the specified OpenStack command request. 2)process_recover_time: which indicates the maximum time (seconds) from the process being killed to recovered
test tool	Developed by the project. Please see folder: "yardstick/benchmark/scenarios/availability/ha_tools"
references	ETSI NFV REL001
configuration	This test case needs two configuration files: 1) test case file: opnfv_yardstick_tc090.yaml -Attackers: see above "attackers" description -waiting_time: which is the time (seconds) from the process being killed to stopping monitors the monitors -Monitors: see above "monitors" description -SLA: see above "metrics" description 2)POD file: pod.yaml The POD configuration should record on pod.yaml first. the "host" item in this test case will use the node name in the pod.yaml.
test sequence	description and expected result
step 1	start monitors: each monitor will run with independently process Result: The monitor info will be collected.
step 2	do attacker: connect the host through SSH, and then execute the kill process script with param value specified by "process_name" Result: Process will be killed.
step 3	stop monitors after a period of time specified by "waiting_time" Result: The monitor info will be aggregated.
step 4	verify the SLA Result: The test case is passed or not.

Yardstick Test Case Description TC091

Control Node Openstack Service High Availability - Heat Api	
test case id	OPNFV_YARDSTICK_TC091: Control node OpenStack service down - heat api
test purpose	This test case will verify the high availability of the orchestration service provided by OpenStack (heat-api) on control node.
test method	This test case kills the processes of heat-api service on a selected control node, then checks whether the request of the related OpenStack command is OK and the killed processes are recovered.
attackers	In this test case, an attacker called "kill-process" is needed. This attacker includes three parameters: 1) fault_type: which is used for finding the attacker's scripts. It should be always set to "kill-process" in this test case. 2) process_name: which is the process name of the specified OpenStack service. If there are multiple processes use the same name on the host, all of them are killed by this attacker. In this case. This parameter should always set to "heat-api". 3) host: which is the name of a control node being attacked. e.g. -fault_type: "kill-process" -process_name: "heat-api" -host: node1
monitors	In this test case, two kinds of monitor are needed: 1. the "openstack-cmd" monitor constantly request a specific OpenStack command, which needs two parameters: 1) monitor_type: which is used for finding the monitor class and related scripts. It should be always set to "openstack-cmd" for this monitor. 2) command_name: which is the command name used for request. In this case, the command name should be neutron related commands. 2. the "process" monitor check whether a process is running on a specific node, which needs three parameters: 1) monitor_type: which used for finding the monitor class and related scripts. It should be always set to "process" for this monitor. 2) process_name: which is the process name for monitor 3) host: which is the name of the node running the process e.g. monitor1: -monitor_type: "openstack-cmd" -command_name: "heat stack list" monitor2: -monitor_type: "process" -process_name: "heat-api" -host: node1
metrics	In this test case, there are two metrics: 1)service_outage_time: which indicates the maximum outage time (seconds) of the specified OpenStack command request. 2)process_recover_time: which indicates the maximum time (seconds) from the process being killed to recovered
test tool	Developed by the project. Please see folder: "yardstick/benchmark/scenarios/availability/ha_tools"
references	ETSI NFV REL001
configuration	This test case needs two configuration files: 1) test case file: opnfv_yardstick_tc091.yaml -Attackers: see above "attackers" description -waiting_time: which is the time (seconds) from the process being killed to the monitor stopped -Monitors: see above "monitors" description -SLA: see above "metrics" description 2)POD file: pod.yaml The POD configuration should record on pod.yaml first. the "host" item in this test case will use the node name in the pod.yaml.
test sequence	description and expected result
step 1	start monitors: each monitor will run with independently process Result: The monitor info will be collected.
step 2	do attacker: connect the host through SSH, and then execute the kill process script with param value specified by "process_name" Result: Process will be killed.
step 3	stop monitors after a period of time specified by "waiting_time" Result: The monitor info will be aggregated.
step 4	verify the SLA Result: The test case is passed or not.
post-action	It is the action when the test cases exist. It will check the status of the specified process on the host, and restart the process if it is not running for next test cases
test verdict	Fails only if SLA is not passed, or if there is a test case execution problem.

Yardstick Test Case Description TC092

SDN Controller resilience in HA configuration	
test case id	OPNFV_YARDSTICK_TC092: SDN controller resilience and high availability HA configuration
test purpose	This test validates SDN controller node high availability by verifying there is no impact on the data plane connectivity when one SDN controller fails in a HA configuration, i.e. all existing configured network services DHCP, ARP, L2, L3VPN, Security Groups should continue to operate between the existing VMs while one SDN controller instance is offline and rebooting. The test also validates that network service operations such as creating a new VM in an existing or new L2 network network remain operational while one instance of the SDN controller is offline and recovers from the failure.
test method	**This test case:** 1. fails one instance of a SDN controller cluster running in a HA configuration on the OpenStack controller node 2. checks if already configured L2 connectivity between existing VMs is not impacted 3. verifies that the system never loses the ability to execute virtual network operations, even when the failed SDN Controller is still recovering
attackers	In this test case, an attacker called "kill-process" is needed. This attacker includes three parameters: 1. `fault_type`: which is used for finding the attacker's scripts. It should be set to 'kill-process' in this test 2. `process_name`: should be set to sdn controller process 3. `host`: which is the name of a control node where opendaylight process is running **example:** • `fault_type`: "kill-process" • `process_name`: "opendaylight-karaf" (TBD) • `host`: node1
monitors	**In this test case, the following monitors are needed** 1. `ping_same_network_l2`: monitor pinging traffic between the VMs in same neutron network 2. `ping_external_snat`: monitor ping traffic from VMs to external destinations (e.g. google.com) 3. `SDN controller process monitor`: a monitor checking the state of a specified SDN controller process. It measures the recovery time of the given process.
operations	**In this test case, the following operations are needed:**

Yardstick Test Case Description TC093

SDN Vswitch resilience in non-HA or HA configuration	
test case id	OPNFV_YARDSTICK_TC093: SDN Vswitch resilience in non-HA or HA configuration
test purpose	This test validates that network data plane services are resilient in the event of Virtual Switch failure in compute nodes. Specifically, the test verifies that existing data plane connectivity is not permanently impacted i.e. all configured network services such as DHCP, ARP, L2, L3 Security Groups continue to operate between the existing VMs eventually after the Virtual Switches have finished rebooting. The test also validates that new network service operations (creating a new VM in the existing L2/L3 network or in a new network, etc.) are operational after the Virtual Switches have recovered from a failure.
test method	This testcase first checks if the already configured DHCP/ARP/L2/L3/SNAT connectivity is proper. After it fails and restarts again the VSwitch services which are running on both OpenStack compute nodes, and then checks if already configured DHCP/ARP/L2/L3/SNAT connectivity is not permanently impacted (even if there are some packet loss events) between VMs and the system is able to execute new virtual network operations once the Vswitch services are restarted and have been fully recovered
attackers	In this test case, two attackers called "kill-process" are needed. These attackers include three parameters: 1. fault_type: which is used for finding the attacker's scripts. It should be set to 'kill-process' in this test 2. process_name: should be set to the name of the Vswitch process 3. host: which is the name of the compute node where the Vswitch process is running **e.g. -fault_type: "kill-process"** -process_name: "openvswitch" -host: node1
monitors	This test case utilizes two monitors of type "ip-status" and one monitor of type "process" to track the following conditions: 1. "ping_same_network_l2": monitor ICMP traffic between VMs in the same Neutron network 2. "ping_external_snat": monitor ICMP traffic from VMs to an external host on the Internet to verify SNAT functionality. 3. "Vswitch process monitor": a monitor checking the state of the specified Vswitch process. It measures the recovery time of the given process. Monitors of type "ip-status" use the "ping" utility to verify reachability of a given target IP.
operations	**In this test case, the following operations are needed:** 1. "nova-create-instance-in_network": create a VM instance in one of the existing Neutron network.

IPv6

Yardstick Test Case Description TC027

IPv6 connectivity between nodes on the tenant network	
test case id	OPNFV_YARDSTICK_TC027_IPv6 connectivity
metric	RTT, Round Trip Time
test purpose	To do a basic verification that IPv6 connectivity is within acceptable boundaries when ipv6 packets travel between hosts located on same or different compute blades. The purpose is also to be able to spot trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
configuration	file: opnfv_yardstick_tc027.yaml Packet size 56 bytes. SLA RTT is set to maximum 30 ms. ipv6 test case can be configured as three independent modules (setup, run, teardown). if you only want to setup ipv6 testing environment, do some tests as you want, "run_step" of task yaml file should be configured as "setup". if you want to setup and run ping6 testing automatically, "run_step" should be configured as "setup, run". and if you have had a environment which has been setup, you only wan to verify the connectivity of ipv6 network, "run_step" should be "run". Of course, default is that three modules run sequentially.
test tool	ping6 Ping6 is normally part of Linux distribution, hence it doesn't need to be installed.
references	ipv6 ETSI-NFV-TST001
applicability	Test case can be configured with different run step you can run setup, run benchmark, teardown independently SLA is optional. The SLA in this test case serves as an example. Considerably lower RTT is expected.
pre-test conditions	The test case image needs to be installed into Glance with ping6 included in it. For Brahmaputra, a compass_os_nosdn_ha deploy scenario is need. more installer and more sdn deploy scenario will be supported soon
test sequence	description and expected result
step 1	To setup IPV6 testing environment: 1. disable security group 2. create (ipv6, ipv4) router, network and subnet 3. create VRouter, VM1, VM2
step 2	To run ping6 to verify IPV6 connectivity : 1. ssh to VM1 2. Ping6 to ipv6 router from VM1 3. Get the result(RTT) and logs are stored
step 3	To teardown IPV6 testing environment 1. delete VRouter, VM1, VM2 2. delete (ipv6, ipv4) router, network and subnet 3. enable security group
test verdict	Test should not PASS if any RTT is above the optional SLA value, or if there is a test case execution problem.

KVM

Yardstick Test Case Description TC028

KVM Latency measurements	
test case id	OPNFV_YARDSTICK_TC028_KVM Latency measurements
metric	min, avg and max latency
test purpose	To evaluate the IaaS KVM virtualization capability with regards to min, avg and max latency. The purpose is also to be able to spot trends. Test results, graphs and similar shall be stored for comparison reasons and product evolution understanding between different OPNFV versions and/or configurations.
configuration	file: samples/cyclictest-node-context.yaml
test tool	Cyclictest (Cyclictest is not always part of a Linux distribution, hence it needs to be installed. As an example see the /yardstick/tools/ directory for how to generate a Linux image with cyclictest included.)
references	Cyclictest
applicability	This test case is mainly for kvm4nfv project CI verify. Upgrade host linux kernel, boot a gust vm update it's linux kernel, and then run the cyclictest to test the new kernel is work well.
pre-test conditions	The test kernel rpm, test sequence scripts and test guest image need put the right folders as specified in the test case yaml file. The test guest image needs with cyclictest included in it. No POD specific requirements have been identified.
test sequence	description and expected result
step 1	The host and guest os kernel is upgraded. Cyclictest is invoked and logs are produced and stored. Result: Logs are stored.
test verdict	Fails only if SLA is not passed, or if there is a test case execution problem.

Parser

Yardstick Test Case Description TC040

Verify Parser Yang-to-Tosca	
test case id	OPNFV_YARDSTICK_TC040 Verify Parser Yang-to-Tosca
metric	1. tosca file which is converted from yang file by Parser 2. result whether the output is same with expected outcome
test purpose	To verify the function of Yang-to-Tosca in Parser.
configuration	file: opnfv_yardstick_tc040.yaml yangfile: the path of the yangfile which you want to convert toscafile: the path of the toscafile which is your expected outcome.
test tool	Parser (Parser is not part of a Linux distribution, hence it needs to be installed. As an example see the /yardstick/benchmark/scenarios/parser/parser_setup.sh for how to install it manual. Of course, it will be installed and uninstalled automatically when you run this test case by yardstick)
references	Parser
applicability	Test can be configured with different path of yangfile and toscafile to fit your real environment to verify Parser
pre-test conditions	No POD specific requirements have been identified. it can be run without VM
test sequence	description and expected result
step 1	parser is installed without VM, running Yang-to-Tosca module to convert yang file to tosca file, validating output against expected outcome. Result: Logs are stored.
test verdict	Fails only if output is different with expected outcome or if there is a test case execution problem.

StorPerf

2.16.4 Templates

Yardstick Test Case Description TCXXX

test case slogan e.g. Network Latency	
test case id	e.g. OPNFV_YARDSTICK_TC001_NW Latency
metric	what will be measured, e.g. latency
test purpose	describe what is the purpose of the test case
configuration	what .yaml file to use, state SLA if applicable, state test duration, list and describe the scenario options used in this TC and also list the options using default values.
test tool	e.g. ping
references	e.g. RFCxxx, ETSI-NFVyyy
applicability	describe variations of the test case which can be performend, e.g. run the test for different packet sizes
pre-test conditions	describe configuration in the tool(s) used to perform the measurements (e.g. fio, pktgen), POD-specific configuration required to enable running the test
test sequence	description and expected result
step 1	use this to describe tests that require sveveral steps e.g collect logs. Result: what happens in this step e.g. logs collected
step 2	remove interface Result: interface down.
step N	what is done in step N Result: what happens
test verdict	expected behavior, or SLA, pass/fail criteria

Task Template Syntax

Basic template syntax

A nice feature of the input task format used in Yardstick is that it supports the template syntax based on Jinja2. This turns out to be extremely useful when, say, you have a fixed structure of your task but you want to parameterize this task in some way. For example, imagine your input task file (task.yaml) runs a set of Ping scenarios:

```
# Sample benchmark task config file
# measure network latency using ping
schema: "yardstick:task:0.1"

scenarios:
-
  type: Ping
  options:
    packetsize: 200
  host: athena.demo
  target: ares.demo

  runner:
    type: Duration
    duration: 60
    interval: 1

  sla:
    max_rtt: 10
```

(continues on next page)

(continued from previous page)

```
    action: monitor

context:
    ...
```

Let's say you want to run the same set of scenarios with the same runner/ context/sla, but you want to try another packetsize to compare the performance. The most elegant solution is then to turn the packetsize name into a template variable:

```
# Sample benchmark task config file
# measure network latency using ping

schema: "yardstick:task:0.1"
scenarios:
-
  type: Ping
  options:
    packetsize: {{packetsize}}
  host: athena.demo
  target: ares.demo

  runner:
    type: Duration
    duration: 60
    interval: 1

  sla:
    max_rtt: 10
    action: monitor

context:
    ...
```

and then pass the argument value for {{packetsize}} when starting a task with this configuration file. Yardstick provides you with different ways to do that:

1.Pass the argument values directly in the command-line interface (with either a JSON or YAML dictionary):

```
yardstick task start samples/ping-template.yaml
--task-args'{"packetsize":"200"}'
```

2.Refer to a file that specifies the argument values (JSON/YAML):

```
yardstick task start samples/ping-template.yaml --task-args-file args.yaml
```

Using the default values

Note that the Jinja2 template syntax allows you to set the default values for your parameters. With default values set, your task file will work even if you don't parameterize it explicitly while starting a task. The default values should be set using the {% set ... %} clause (task.yaml). For example:

```
# Sample benchmark task config file
# measure network latency using ping
schema: "yardstick:task:0.1"
{% set packetsize = packetsize or "100" %}
```

(continues on next page)

(continued from previous page)

```
scenarios:
-
  type: Ping
  options:
  packetsize: {{packetsize}}
  host: athena.demo
  target: ares.demo

  runner:
    type: Duration
    duration: 60
    interval: 1
  ...
```

If you don't pass the value for {{packetsize}} while starting a task, the default one will be used.

Advanced templates

Yardstick makes it possible to use all the power of Jinja2 template syntax, including the mechanism of built-in functions. As an example, let us make up a task file that will do a block storage performance test. The input task file (fio-template.yaml) below uses the Jinja2 for-endfor construct to accomplish that:

```
#Test block sizes of 4KB, 8KB, 64KB, 1MB
#Test 5 workloads: read, write, randwrite, randread, rw
schema: "yardstick:task:0.1"

 scenarios:
{% for bs in ['4k', '8k', '64k', '1024k' ] %}
  {% for rw in ['read', 'write', 'randwrite', 'randread', 'rw' ] %}
-
  type: Fio
  options:
    filename: /home/ubuntu/data.raw
    bs: {{bs}}
    rw: {{rw}}
    ramp_time: 10
  host: fio.demo
  runner:
    type: Duration
    duration: 60
    interval: 60

  {% endfor %}
{% endfor %}
context
    ...
```

2.17 NSB Sample Test Cases

2.17.1 Abstract

This chapter lists available NSB test cases.

2.17.2 NSB PROX Test Case Descriptions

Yardstick Test Case Description: NSB PROX ACL

NSB PROX test for NFVI characterization	
test case id	tc_prox_{context}_acl-{port_num} • context = baremetal or heat_context; • port_num = 2 or 4;
metric	• Network Throughput; • TG Packets Out; • TG Packets In; • VNF Packets Out; • VNF Packets In; • Dropped packets;
test purpose	This test allows to measure how well the SUT can exploit structures in the list of ACL rules. The ACL rules are matched against a 7-tuple of the input packet: the regular 5-tuple and two VLAN tags. The rules in the rule set allow the packet to be forwarded and the rule set contains a default "match all" rule. The KPI is measured with the rule set that has a moderate number of rules with moderate similarity between the rules & the fraction of rules that were used. The ACL test cases are implemented to run in baremetal and heat context for 2 port and 4 port configuration.
configuration	The ACL test cases are listed below: • tc_prox_baremetal_acl-2.yaml • tc_prox_baremetal_acl-4.yaml • tc_prox_heat_context_acl-2.yaml • tc_prox_heat_context_acl-4.yaml Test duration is set as 300sec for each test. Packet size set as 64 bytes in traffic profile. These can be configured
test tool	PROX PROX is a DPDK application that can simulate VNF workloads and can generate traffic and used for NFVI characterization
applicability	This PROX ACL test cases can be configured with different: • packet sizes; • test durations; • tolerated loss; Default values exist.
pre-test conditions	For Openstack test case image (yardstick-samplevnfs) needs to be installed into Glance with Prox and Dpdk included in it. The test need multi-queue enabled in Glance image. For Baremetal tests cases Prox and Dpdk must be installed in the hosts where the test is executed. The pod.yaml file must have the necessary system and NIC information
test sequence	description and expected result
step 1	For Baremetal test: The TG and VNF are started on the hosts based on the pod file. For Heat test: Two host VMs are booted, as Traffic generator and VNF(ACL workload) based on the test flavor.
step 2	Yardstick is connected with the TG and VNF by using ssh. The test will resolve the topology and instantiate the VNF and TG and collect the KPI's/metrics.
step 3	The TG will send packets to the VNF. If the number of dropped packets is more than the tolerated loss the

Yardstick Test Case Description: NSB PROX BNG

NSB PROX test for NFVI characterization	
test case id	tc_prox_{context}_bng-{port_num} • context = baremetal or heat_context; • port_num = 4;
metric	• Network Throughput; • TG Packets Out; • TG Packets In; • VNF Packets Out; • VNF Packets In; • Dropped packets;
test purpose	The BNG workload converts packets from QinQ to GRE tunnels, handles routing and adds/removes MPLS tags. This use case simulates a realistic and complex application. The number of users is 32K per port and the number of routes is 8K. The BNG test cases are implemented to run in baremetal and heat context an require 4 port topology to run the default configuration.
configuration	The BNG test cases are listed below: • tc_prox_baremetal_bng-2.yaml • tc_prox_baremetal_bng-4.yaml • tc_prox_heat_context_bng-2.yaml • tc_prox_heat_context_bng-4.yaml Test duration is set as 300sec for each test. The minimum packet size for BNG test is 78 bytes. This is set in the BNG traffic profile and can be configured to use a higher packet size for the test.
test tool	PROX PROX is a DPDK application that can simulate VNF workloads and can generate traffic and used for NFVI characterization
applicability	The PROX BNG test cases can be configured with different: • packet sizes; • test durations; • tolerated loss; Default values exist.
pre-test conditions	For Openstack test case image (yardstick-samplevnfs) needs to be installed into Glance with Prox and Dpdk included in it. The test need multi-queue enabled in Glance image. For Baremetal tests cases Prox and Dpdk must be installed in the hosts where the test is executed. The pod.yaml file must have the necessary system and NIC information
test sequence	description and expected result
step 1	For Baremetal test: The TG and VNF are started on the hosts based on the pod file. For Heat test: Two host VMs are booted, as Traffic generator and VNF(BNG workload) based on the test flavor.
step 2	Yardstick is connected with the TG and VNF by using ssh. The test will resolve the topology and instantiate the VNF and TG and collect the KPI's/metrics.
step 3	The TG will send packets to the VNF. If the number of dropped packets is more than the tolerated loss the

Yardstick Test Case Description: NSB PROX BNG_QoS

NSB PROX test for NFVI characterization	
test case id	tc_prox_{context}_bng_qos-{port_num} • context = baremetal or heat_context; • port_num = 4;
metric	• Network Throughput; • TG Packets Out; • TG Packets In; • VNF Packets Out; • VNF Packets In; • Dropped packets;
test purpose	The BNG+QoS workload converts packets from QinQ to GRE tunnels, handles routing and adds/removes MPLS tags and performs a QoS. This use case simulates a realistic and complex application. The number of users is 32K per port and the number of routes is 8K. The BNG_QoS test cases are implemented to run in baremetal and heat context an require 4 port topology to run the default configuration.
configuration	The BNG_QoS test cases are listed below: • tc_prox_baremetal_bng_qos-2.yaml • tc_prox_baremetal_bng_qos-4.yaml • tc_prox_heat_context_bng_qos-2.yaml • tc_prox_heat_context_bng_qos-4.yaml Test duration is set as 300sec for each test. The minumum packet size for BNG_QoS test is 78 bytes. This is set in the bng_qos traffic profile and can be configured to use a higher packet size for the test.
test tool	PROX PROX is a DPDK application that can simulate VNF workloads and can generate traffic and used for NFVI characterization
applicability	This PROX BNG_QoS test cases can be configured with different: • packet sizes; • test durations; • tolerated loss; Default values exist.
pre-test conditions	For Openstack test case image (yardstick-samplevnfs) needs to be installed into Glance with Prox and Dpdk included in it. The test need multi-queue enabled in Glance image. For Baremetal tests cases Prox and Dpdk must be installed in the hosts where the test is executed. The pod.yaml file must have the necessary system and NIC information
test sequence	description and expected result
step 1	For Baremetal test: The TG and VNF are started on the hosts based on the pod file. For Heat test: Two host VMs are booted, as Traffic generator and VNF(BNG_QoS workload) based on the test flavor.
step 2	Yardstick is connected with the TG and VNF by using ssh. The test will resolve the topology and instantiate the VNF and TG and collect the KPI's/metrics.
step 3	The TG will send packets to the VNF. If the number of dropped packets is more than the tolerated loss the

Yardstick Test Case Description: NSB PROX L2FWD

NSB PROX test for NFVI characterization	
test case id	tc_prox_{context}_l2fwd-{port_num} • context = baremetal or heat_context; • port_num = 2 or 4;
metric	• Network Throughput; • TG Packets Out; • TG Packets In; • VNF Packets Out; • VNF Packets In; • Dropped packets;
test purpose	The PROX L2FWD test has 3 types of test cases: L2FWD: The application will take packets in from one port and forward them unmodified to another port L2FWD_Packet_Touch: The application will take packets in from one port, update src and dst MACs and forward them to another port. L2FWD_Multi_Flow: The application will take packets in from one port, update src and dst MACs and forward them to another port. This test case exercises the softswitch with 200k flows. The above test cases are implemented for baremetal and heat context for 2 port and 4 port configuration.
configuration	The L2FWD test cases are listed below: • tc_prox_baremetal_l2fwd-2.yaml • tc_prox_baremetal_l2fwd-4.yaml • tc_prox_baremetal_l2fwd_pktTouch-2.yaml • tc_prox_baremetal_l2fwd_pktTouch-4.yaml • tc_prox_baremetal_l2fwd_multiflow-2.yaml • tc_prox_baremetal_l2fwd_multiflow-4.yaml • tc_prox_heat_context_l2fwd-2.yaml • tc_prox_heat_context_l2fwd-4.yaml • tc_prox_heat_context_l2fwd_pktTouch-2.yaml • tc_prox_heat_context_l2fwd_pktTouch-4.yaml • tc_prox_heat_context_l2fwd_multiflow-2.yaml • tc_prox_heat_context_l2fwd_multiflow-4.yaml Test duration is set as 300sec for each test. Packet size set as 64 bytes in traffic profile These can be configured
test tool	PROX PROX is a DPDK application that can simulate VNF workloads and can generate traffic and used for NFVI characterization
applicability	The PROX L2FWD test cases can be configured with different: • packet sizes; • test durations; • tolerated loss; Default values exist.
pre-test conditions	For Openstack test case image (yardstick-samplevnfs) needs to be installed into Glance with Prox and Dpdk included in it. For Baremetal tests cases Prox and Dpdk must be installed in the hosts where the test is executed. The pod.yaml file must have the necessary system and NIC information
test sequence	description and expected result
step 1	For Baremetal test: The TG and VNF are started on the hosts based on the pod file

Yardstick Test Case Description: NSB PROX L3FWD

NSB PROX test for NFVI characterization	
test case id	tc_prox_{context}_l3fwd-{port_num} • context = baremetal or heat_context; • port_num = 2 or 4;
metric	• Network Throughput; • TG Packets Out; • TG Packets In; • VNF Packets Out; • VNF Packets In; • Dropped packets;
test purpose	The PROX L3FWD application performs basic routing of packets with LPM based look-up method. The L3FWD test cases are implemented for baremetal and heat context for 2 port and 4 port configuration.
configuration	The L3FWD test cases are listed below: • tc_prox_baremetal_l3fwd-2.yaml • tc_prox_baremetal_l3fwd-4.yaml • tc_prox_heat_context_l3fwd-2.yaml • tc_prox_heat_context_l3fwd-4.yaml Test duration is set as 300sec for each test. The minimum packet size for L3FWD test is 64 bytes. This is set in the traffic profile and can be configured to use a higher packet size for the test.
test tool	PROX PROX is a DPDK application that can simulate VNF workloads and can generate traffic and used for NFVI characterization
applicability	This PROX L3FWD test cases can be configured with different: • packet sizes; • test durations; • tolerated loss; Default values exist.
pre-test conditions	For Openstack test case image (yardstick-samplevnfs) needs to be installed into Glance with Prox and Dpdk included in it. The test need multi-queue enabled in Glance image. For Baremetal tests cases Prox and Dpdk must be installed in the hosts where the test is executed. The pod.yaml file must have the necessary system and NIC information
test sequence	description and expected result
step 1	For Baremetal test: The TG and VNF are started on the hosts based on the pod file. For Heat test: Two host VMs are booted, as Traffic generator and VNF(L3FWD workload) based on the test flavor.
step 2	Yardstick is connected with the TG and VNF by using ssh. The test will resolve the topology and instantiate the VNF and TG and collect the KPI's/metrics.
step 3	The TG will send packet to the VNF. If the number of dropped packets is more than the tolerated loss the line rate or throughput is halved. This is done until the dropped packets are within an acceptable tolerated loss. The KPI is the number of packets per second for 64 byte packets with an accepted minimal packet loss for the

Yardstick Test Case Description: NSB PROX MPLS Tagging

NSB PROX test for NFVI characterization	
test case id	tc_prox_{context}_mpls_tagging-{port_num} • context = baremetal or heat_context; • port_num = 2 or 4;
metric	• Network Throughput; • TG Packets Out; • TG Packets In; • VNF Packets Out; • VNF Packets In; • Dropped packets;
test purpose	The PROX MPLS Tagging test will take packets in from one port add an MPLS tag and forward them to another port. While forwarding packets in other direction MPLS tags will be removed. The MPLS test cases are implemented to run in baremetal and heat context an require 4 port topology to run the default configuration.
configuration	The MPLS Tagging test cases are listed below: • tc_prox_baremetal_mpls_tagging-2.yaml • tc_prox_baremetal_mpls_tagging-4.yaml • tc_prox_heat_context_mpls_tagging-2.yaml • tc_prox_heat_context_mpls_tagging-4.yaml Test duration is set as 300sec for each test. The minimum packet size for MPLS test is 68 bytes. This is set in the traffic profile and can be configured to use higher packet sizes.
test tool	PROX PROX is a DPDK application that can simulate VNF workloads and can generate traffic and used for NFVI characterization
applicability	The PROX MPLS Tagging test cases can be configured with different: • packet sizes; • test durations; • tolerated loss; Default values exist.
pre-test conditions	For Openstack test case image (yardstick-samplevnfs) needs to be installed into Glance with Prox and Dpdk included in it. For Baremetal tests cases Prox and Dpdk must be installed in the hosts where the test is executed. The pod.yaml file must have the necessary system and NIC information
test sequence	description and expected result
step 1	For Baremetal test: The TG and VNF are started on the hosts based on the pod file. For Heat test: Two host VMs are booted, as Traffic generator and VNF(MPLS workload) based on the test flavor.
step 2	Yardstick is connected with the TG and VNF by using ssh. The test will resolve the topology and instantiate the VNF and TG and collect the KPI's/metrics.
step 3	The TG will send packets to the VNF. If the number of dropped packets is more than the tolerated loss the line rate or throughput is halved. This is done until the

Yardstick Test Case Description: NSB PROX Packet Buffering

NSB PROX test for NFVI characterization	
test case id	tc_prox_{context}_buffering-{port_num} • context = baremetal or heat_context • port_num = 1
metric	• Network Throughput; • TG Packets Out; • TG Packets In; • VNF Packets Out; • VNF Packets In; • Dropped packets;
test purpose	This test measures the impact of the condition when packets get buffered, thus they stay in memory for the extended period of time, 125ms in this case. The Packet Buffering test cases are implemented to run in baremetal and heat context. The test runs only on the first port of the SUT.
configuration	The Packet Buffering test cases are listed below: • tc_prox_baremetal_buffering-1.yaml • tc_prox_heat_context_buffering-1.yaml Test duration is set as 300sec for each test. The minimum packet size for Buffering test is 64 bytes. This is set in the traffic profile and can be configured to use a higher packet size for the test.
test tool	PROX PROX is a DPDK application that can simulate VNF workloads and can generate traffic and used for NFVI characterization
applicability	**The PROX Packet Buffering test cases can be configured with** different: • packet sizes; • test durations; • tolerated loss; Default values exist.
pre-test conditions	For Openstack test case image (yardstick-samplevnfs) needs to be installed into Glance with Prox and Dpdk included in it. The test need multi-queue enabled in Glance image. For Baremetal tests cases Prox and Dpdk must be installed in the hosts where the test is executed. The pod.yaml file must have the necessary system and NIC information
test sequence	description and expected result
step 1	For Baremetal test: The TG and VNF are started on the hosts based on the pod file. For Heat test: Two host VMs are booted, as Traffic generator and VNF(Packet Buffering workload) based on the test flavor.
step 2	Yardstick is connected with the TG and VNF by using ssh. The test will resolve the topology and instantiate the VNF and TG and collect the KPI's/metrics.
step 3	The TG will send packets to the VNF. If the number of dropped packets is more than the tolerated loss the line rate or throughput is halved. This is done until the dropped packets are within an acceptable tolerated loss. The KPI in this test is the maximum number of pack-

Yardstick Test Case Description: NSB PROX Load Balancer

NSB PROX test for NFVI characterization	
test case id	tc_prox_{context}_lb-{port_num} • context = baremetal or heat_context • port_num = 4
metric	• Network Throughput; • TG Packets Out; • TG Packets In; • VNF Packets Out; • VNF Packets In; • Dropped packets;
test purpose	The applciation transmits packets on one port and recieves them on 4 ports. The conventional 5-tuple is used in this test as it requires some extraction steps and allows defining enough distinct values to find the performance limits. The load is increased (adding more ports if needed) while packets are load balanced using a hash table of 8M entries The number of packets per second that can be forwarded determines the KPI. The default packet size is 64 bytes.
configuration	The Load Balancer test cases are listed below: • tc_prox_baremetal_lb-4.yaml • tc_prox_heat_context_lb-4.yaml Test duration is set as 300sec for each test. Packet size set as 64 bytes in traffic profile. These can be configured
test tool	PROX PROX is a DPDK application that can simulate VNF workloads and can generate traffic and used for NFVI characterization
applicability	**The PROX Load Balancer test cases can be configured with** different: • packet sizes; • test durations; • tolerated loss; Default values exist.
pre-test conditions	For Openstack test case image (yardstick-samplevnfs) needs to be installed into Glance with Prox and Dpdk included in it. The test need multi-queue enabled in Glance image. For Baremetal tests cases Prox and Dpdk must be installed in the hosts where the test is executed. The pod.yaml file must have the necessary system and NIC information
test sequence	description and expected result
step 1	For Baremetal test: The TG and VNF are started on the hosts based on the pod file. For Heat test: Two host VMs are booted, as Traffic generator and VNF(Load Balancer workload) based on the test flavor.
step 2	Yardstick is connected with the TG and VNF by using ssh. The test will resolve the topology and instantiate the VNF and TG and collect the KPI's/metrics.
step 3	The TG will send packets to the VNF. If the number of dropped packets is more than the tolerated loss the line rate or throughput is halved. This is done until the

Yardstick Test Case Description: NSB PROX VPE

NSB PROX test for NFVI characterization	
test case id	tc_prox_{context}_vpe-{port_num} • context = baremetal or heat_context; • port_num = 4;
metric	• Network Throughput; • TG Packets Out; • TG Packets In; • VNF Packets Out; • VNF Packets In; • Dropped packets;
test purpose	The PROX VPE test handles packet processing, routing, QinQ encapsulation, flows, ACL rules, adds/removes MPLS tagging and performs QoS before forwarding packet to another port. The reverse applies to forwarded packets in the other direction. The VPE test cases are implemented to run in baremetal and heat context an require 4 port topology to run the default configuration.
configuration	The VPE test cases are listed below: • tc_prox_baremetal_vpe-4.yaml • tc_prox_heat_context_vpe-4.yaml Test duration is set as 300sec for each test. The minimum packet size for VPE test is 68 bytes. This is set in the traffic profile and can be configured to use higher packet sizes.
test tool	PROX PROX is a DPDK application that can simulate VNF workloads and can generate traffic and used for NFVI characterization
applicability	The PROX VPE test cases can be configured with different: • packet sizes; • test durations; • tolerated loss; Default values exist.
pre-test conditions	For Openstack test case image (yardstick-samplevnfs) needs to be installed into Glance with Prox and Dpdk included in it. For Baremetal tests cases Prox and Dpdk must be installed in the hosts where the test is executed. The pod.yaml file must have the necessary system and NIC information
test sequence	description and expected result
step 1	For Baremetal test: The TG and VNF are started on the hosts based on the pod file. For Heat test: Two host VMs are booted, as Traffic generator and VNF(VPE workload) based on the test flavor.
step 2	Yardstick is connected with the TG and VNF by using ssh. The test will resolve the topology and instantiate the VNF and TG and collect the KPI's/metrics.
step 3	The TG will send packets to the VNF. If the number of dropped packets is more than the tolerated loss the line rate or throughput is halved. This is done until the dropped packets are within an acceptable tolerated loss. The KPI is the number of packets per second for 68 bytes packet size with an accepted minimal packet loss

Yardstick Test Case Description: NSB PROX LwAFTR

NSB PROX test for NFVI characterization	
test case id	tc_prox_{context}_lw_aftr-{port_num} • context = baremetal or heat_context; • port_num = 4;
metric	• Network Throughput; • TG Packets Out; • TG Packets In; • VNF Packets Out; • VNF Packets In; • Dropped packets;
test purpose	The PROX LW_AFTR test will take packets in from one port and remove the ipv6 encapsulation and forward them to another port. While forwarded packets in other direction will be encapsulated in an ipv6 header. The lw_aftr test cases are implemented to run in baremetal and heat context an require 4 port topology to run the default configuration.
configuration	The LW_AFTR test cases are listed below: • tc_prox_baremetal_lw_aftr-4.yaml • tc_prox_heat_context_lw_aftr-4.yaml Test duration is set as 300sec for each test. The minimum packet size for MPLS test is 68 bytes. This is set in the traffic profile and can be configured to use higher packet sizes.
test tool	PROX PROX is a DPDK application that can simulate VNF workloads and can generate traffic and used for NFVI characterization
applicability	The PROX lwAFTR test cases can be configured with different: • packet sizes; • test durations; • tolerated loss; Default values exist.
pre-test conditions	For Openstack test case image (yardstick-samplevnfs) needs to be installed into Glance with Prox and Dpdk included in it. For Baremetal tests cases Prox and Dpdk must be installed in the hosts where the test is executed. The pod.yaml file must have the necessary system and NIC information
test sequence	description and expected result
step 1	For Baremetal test: The TG and VNF are started on the hosts based on the pod file. For Heat test: Two host VMs are booted, as Traffic generator and VNF(LW_AFTR workload) based on the test flavor.
step 2	Yardstick is connected with the TG and VNF by using ssh. The test will resolve the topology and instantiate the VNF and TG and collect the KPI's/metrics.
step 3	The TG will send packets to the VNF. If the number of dropped packets is more than the tolerated loss the line rate or throughput is halved. This is done until the dropped packets are within an acceptable tolerated loss. The KPI is the number of packets per second for 86 bytes packet size with an accepted minimal packet loss

Yardstick Test Case Description: NSB EPC DEFAULT BEARER

NSB EPC default bearer test case	
test case id	tc_epc_default_bearer_landslide_{dmf_setup} • dmf_setup: single or multi dmf test session setup;
metric	All metrics provided by Spirent Landslide traffic generator
test purpose	The Spirent Landslide product provides one box solution which allows to fully emulate all EPC network nodes including mobile users, network host and generate control and data plane traffic. This test allows to check processing capability of EPC under different levels of load (number of subscriber, generated traffic throughput) for case when only one default bearer is using for transferring traffic from UE to Network. It's easy to replace emulated node or multiple nodes in test topology with real node or corresponding vEPC VNF as DUT and check it's processing capabilities under specific test case load conditions.
configuration	The EPC default bearer test cases are listed below: • tc_epc_default_bearer_create_landslide.yaml • tc_epc_default_bearer_create_landslide_multi_dmf.yaml Test duration: • is set as 60sec (specified in test session profile); Traffic type: • UDP - for single DMF test case; • UDP and TCP - for multi DMF test case; Packet sizes: • 512 bytes for UDP packets; • 1518 bytes for TCP packets; Traffic transaction rate: • 5 trans/s.; Number of mobile subscribers: • 20000; Number of default bearers per subscriber: • 1. The above fields and values are the main options used for the test case. Other configurable options could be found in test session profile yaml file. All these options have default values which can be overwritten in test case file.
test tool	Spirent Landslide The Spirent Landslide is a tool for functional & performance testing of different types of mobile networks. It emulates real-world control and data traffic of mobile subscribers moving through virtualized EPC network. Detailed description of Spirent Landslide product could be found here: https://www.spirent.com/Products/Landslide
applicability	This EPC DEFAULT BEARER test cases can be configured with different: • packet sizes; • traffic transaction rate; • number of subscribers sessions; • number of default bearers per subscriber; • subscribers connection rate; • subscribers disconnection rate;

Yardstick Test Case Description: NSB EPC DEDICATED BEARER

NSB EPC dedicated bearer test case	
test case id	tc_epc_{initiator}_dedicated_bearer_landslide • initiator: dedicated bearer creation initiator side could be UE (ue) or Network (network).
metric	All metrics provided by Spirent Landslide traffic generator
test purpose	The Spirent Landslide product provides one box solution which allows to fully emulate all EPC network nodes including mobile users, network host and generate control and data plane traffic. This test allows to check processing capability under different levels of load (number of subscriber, generated traffic throughput, etc.) for case when default and dedicated bearers are creating and using for traffic transferring. It's easy to replace emulated node or multiple nodes in test topology with real node or corresponding vEPC VNF as DUT and check it's processing capabilities under specific test case load conditions.
configuration	The EPC dedicated bearer test cases are listed below: • tc_epc_ue_dedicated_bearer_create_landslide.yaml • tc_epc_network_dedicated_bearer_create_landslide.yaml Test duration: • is set as 60sec (specified in test session profile); Traffic type: • UDP; Packet sizes: • 512 bytes; Traffic transaction rate: • 5 trans/s.; Number of mobile subscribers: • 20000; Number of default bearers per subscriber: • 1; Number of dedicated bearers per default bearer: • 1. The above fields and values are the main options used for the test case. Other configurable options could be found in test session profile yaml file. All these options have default values which can be overwritten in test case file.
test tool	Spirent Landslide The Spirent Landslide is a tool for functional and performance testing of different types of mobile networks. It emulates real-world control and data traffic of mobile subscribers moving through virtualized EPC network. Detailed description of Spirent Landslide product could be found here: https://www.spirent.com/Products/Landslide
applicability	This EPC DEDICATED BEARER test cases can be configured with different: • packet sizes; • traffic transaction rate; • number of subscribers sessions; • number of default bearers per subscriber; • number of dedicated bearers per default;

Yardstick Test Case Description: NSB EPC SAEGW RELOCATION

NSB EPC SAEGW throughput with relocation test case	
test case id	tc_epc_saegw_tput_relocation_landslide
metric	All metrics provided by Spirent Landslide traffic generator
test purpose	The Spirent Landslide product provides one box solution which allows to fully emulate all EPC network nodes including mobile users, network host and generate control and data plane traffic. This test allows to check processing capability of EPC handling large amount of subscribers X2 handovers between different eNBs while UEs are sending traffic. It's easy to replace emulated node or multiple nodes in test topology with real node or corresponding vEPC VNF as DUT and check it's processing capabilities under specific test case load conditions.
configuration	The EPC SAEGW throughput with relocation tests are listed below: • tc_epc_saegw_tput_relocation_landslide.yaml Test duration: • is set as 60sec (specified in test session profile); Traffic type: • UDP; Packet sizes: • 512 bytes; Traffic transaction rate: • 5 trans/s.; Number of mobile subscribers: • 20000; Number of default bearers per subscriber: • 1; Handover type: • X2 handover; Mobility time (timeout after sessions were established after which handover will start): • 10000ms; Handover start type: • When all sessions started; Mobility mode: • Single handoff; Mobility Rate: • 120 subscribers/s. The above fields and values are the main options used for the test case. Other configurable options could be found in test session profile yaml file. All these options have default values which can be overwritten in test case file.
test tool	Spirent Landslide The Spirent Landslide is a tool for functional & performance testing of different types of mobile networks. It emulates real-world control and data traffic of mobile subscribers moving through virtualized EPC network. Detailed description of Spirent Landslide product could be found here: https://www.spirent.com/Products/Landslide
applicability	This EPC UE SERVICE REQUEST test cases can be configured with different: • packet sizes;

Yardstick Test Case Description: NSB EPC NETWORK SERVICE REQUEST

NSB EPC network service request test case	
test case id	tc_epc_network_service_request_landslide • initiator: service request initiator side could be UE (ue) or Network (network).
metric	All metrics provided by Spirent Landslide traffic generator
test purpose	The Spirent Landslide product provides one box solution which allows to fully emulate all EPC network nodes including mobile users, network host and generate control and data plane traffic. This test covers case of network initiated service request & allows to check processing capabilities of EPC handling high amount of continuous Downlink Data Notification messages from network to UEs which are in Idle state. It's easy to replace emulated node or multiple nodes in test topology with real node or corresponding vEPC VNF as DUT and check it's processing capabilities under specific test case load conditions.
configuration	The EPC network service request test cases are listed below: • tc_epc_network_service_request_landslide.yaml Test duration: • is set as 60sec (specified in test session profile); Traffic type: • UDP; Packet sizes: • 512 bytes; Traffic transaction rate: • 0.1 trans/s.; Number of mobile subscribers: • 20000; Number of default bearers per subscriber: • 1; Idle entry time (timeout after which UE goes to Idle state): • 5s; Traffic start delay: • 1000ms. The above fields and values are the main options used for the test case. Other configurable options could be found in test session profile yaml file. All these options have default values which can be overwritten in test case file.
test tool	Spirent Landslide The Spirent Landslide is a tool for functional & performance testing of different types of mobile networks. It emulates real-world control and data traffic of mobile subscribers moving through virtualized EPC network. Detailed description of Spirent Landslide product could be found here: https://www.spirent.com/Products/Landslide
applicability	This EPC NETWORK SERVICE REQUEST test case can be configured with different: • packet sizes; • traffic transaction rate;

Yardstick Test Case Description: NSB EPC UE SERVICE REQUEST

NSB EPC UE service request test case	
test case id	tc_epc_{initiator}_service_request_landslide • initiator: service request initiator side could be UE (ue) or Network (nw).
metric	All metrics provided by Spirent Landslide traffic generator
test purpose	The Spirent Landslide product provides one box solution which allows to fully emulate all EPC network nodes including mobile users, network host and generate control and data plane traffic. This test allows to check processing capabilities of EPC under high user connections rate and traffic load for case when UEs initiates service request (UE initiates bearer modification request to provide dedicated bearer for new type of traffic) It's easy to replace emulated node or multiple nodes in test topology with real node or corresponding vEPC VNF as DUT and check it's processing capabilities under specific test case load conditions.
configuration	The EPC ue service request test cases are listed below: • tc_epc_ue_service_request_landslide.yaml Test duration: • is set as 60sec (specified in test session profile); Traffic type: • UDP; Packet sizes: • 512 bytes; Traffic transaction rate: • 5 trans/s.; Number of mobile subscribers: • 20000; Number of default bearers per subscriber: • 1; Number of dedicated bearers per default bearer: • 1. TFT settings for dedicated bearers: • TFT configured to filter TCP traffic (Protocol ID 6) Modified TFT settings: • Create new TFT to filter UDP traffic (Protocol ID 17) from 2002 local port and 2003 remote port; Modified QoS settings: • Set QCI 5 for dedicated bearers; The above fields and values are the main options used for the test case. Other configurable options could be found in test session profile yaml file. All these options have default values which can be overwritten in test case file.
test tool	Spirent Landslide The Spirent Landslide is a tool for functional & performance testing of different types of mobile networks. It emulates real-world control and data traffic of mobile subscribers moving through virtualized EPC network. Detailed description of Spirent Landslide product could be found here: https://www.spirent.com/Products/Landslide

Yardstick Test Case Description: NSB vFW RFC2544

NSB vFW test for VNF characterization	
test case id	tc_{context}_rfc2544_ipv4_1rule_1flow_{pkt_size}_{tg_type} • **context = baremetal, heat, heat_external, ovs, sriov** heat_sriov_external contexts; • **tg_type = ixia (context != heat,heat_sriov_external),** trex; • **pkt_size = 64B - all contexts;** 128B, 256B, 512B, 1024B, 1280B, 1518B - (context = heat, tg_type = ixia)
metric	• Network Throughput; • TG Packets Out; • TG Packets In; • TG Latency; • VNF Packets Out; • VNF Packets In; • VNF Packets Fwd; • Dropped packets;
test purpose	The VFW RFC2544 tests measure performance characteristics of the SUT (multiple ports) and sends UDP bidirectional traffic from all TG ports to SampleVNF vFW application. The application forwards received traffic based on rules provided by the user in the TC configuration and default rules created by vFW to send traffic from uplink ports to downlink and voice versa.
configuration	The 2 ports RFC2544 test cases are listed below: • tc_baremetal_rfc2544_ipv4_1rule_1flow_64B_ixia.yaml • tc_baremetal_rfc2544_ipv4_1rule_1flow_64B_trex.yaml • tc_heat_external_rfc2544_ipv4_1rule_1flow_1024B_ixia.yaml • tc_heat_external_rfc2544_ipv4_1rule_1flow_1280B_ixia.yaml • tc_heat_external_rfc2544_ipv4_1rule_1flow_128B_ixia.yaml • tc_heat_external_rfc2544_ipv4_1rule_1flow_1518B_ixia.yaml • tc_heat_external_rfc2544_ipv4_1rule_1flow_256B_ixia.yaml • tc_heat_external_rfc2544_ipv4_1rule_1flow_512B_ixia.yaml • tc_heat_external_rfc2544_ipv4_1rule_1flow_64B_ixia.yaml • tc_heat_external_rfc2544_ipv4_1rule_1flow_64B_trex.yaml • tc_heat_sriov_external_rfc2544_ipv4_1rule_1flow_64B_trex.yaml • tc_heat_rfc2544_ipv4_1rule_1flow_64B_trex.yaml • tc_ovs_rfc2544_ipv4_1rule_1flow_64B_ixia.yaml • tc_ovs_rfc2544_ipv4_1rule_1flow_64B_trex.yaml • tc_sriov_rfc2544_ipv4_1rule_1flow_64B_ixia.yaml • tc_sriov_rfc2544_ipv4_1rule_1flow_64B_trex.yaml The 4 ports RFC2544 test cases are listed below: • tc_baremetal_rfc2544_ipv4_1rule_1flow_64B_ixia_4port.yaml • tc_tc_baremetal_rfc2544_ipv4_1rule_1flow_64B_trex_4port.yaml • tc_tc_heat_external_rfc2544_ipv4_1rule_1flow_64B_trex_4port.yaml • tc_tc_heat_rfc2544_ipv4_1rule_1flow_64B_trex_4port.yaml The scale-up RFC2544 test cases are listed below: • tc_tc_heat_rfc2544_ipv4_1rule_1flow_64B_trex_scale-up.yaml The scale-out RFC2544 test cases are listed below: • tc_heat_rfc2544_ipv4_1rule_1flow_64B_trex_scale_out.yaml Test duration is set as 30 sec for each test and default

Yardstick Test Case Description: NSB vFW RFC2544 (correlated)

NSB vFW test for VNF characterization using correlated traffic	
test case id	tc_{context}_rfc2544_ipv4_1rule_1flow_64B_trex_corelated • context = baremetal, heat
metric	• Network Throughput; • TG Packets Out; • TG Packets In; • TG Latency; • VNF Packets Out; • VNF Packets In; • VNF Packets Fwd; • Dropped packets; NOTE: For correlated TCs the TG metrics are available on uplink ports.
test purpose	The VFW RFC2544 correlated tests measure performance characteristics of the SUT (multiple ports) and sends UDP traffic from uplink TG ports to SampleVNF vFW application. The application forwards received traffic from uplink ports to downlink ports based on rules provided by the user in the TC configuration and default rules created by vFW. The VNF downlink traffic is received by another UDPReplay VNF and it is mirrored back to the VNF on the same port. Finally, the traffic is received back to the TG uplink port.
configuration	The 2 ports RFC2544 correlated test cases are listed below: • tc_baremetal_rfc2544_ipv4_1rule_1flow_64B_trex_corelated_traffic.yaml Multiple VNF (2, 4, 10) RFC2544 correlated test cases are listed below: • tc_heat_rfc2544_ipv4_1rule_1flow_64B_trex_correlated_scale_10.yaml • tc_heat_rfc2544_ipv4_1rule_1flow_64B_trex_correlated_scale_2.yaml • tc_heat_rfc2544_ipv4_1rule_1flow_64B_trex_correlated_scale_4.yaml The scale-out RFC2544 test cases are listed below: • tc_heat_rfc2544_ipv4_1rule_1flow_64B_trex_correlated_scale_out.yaml Test duration is set as 30 sec for each test and default number of rules are applied. These can be configured
test tool	The vFW is a DPDK application that performs basic filtering for malformed packets and dynamic packet filtering of incoming packets using the connection tracker library.
applicability	The vFW RFC2544 test cases can be configured with different: • packet sizes; • test duration; • tolerated loss; • traffic flows; • rules;
	Default values exist.
pre-test conditions	For OpenStack test case image (yardstick-samplevnf) needs to be installed into Glance with vFW and DPDK included in it (NSB install). For Baremetal tests cases vFW and DPDK must be in-

Yardstick Test Case Description: NSB vFW RFC3511 (HTTP)

NSB vFW test for VNF characterization based on RFC3511 and IXIA	
test case id	tc_{context}_http_ixload_{http_size}_Requests-65000_{type} • context = baremetal, heat_external • http_size = 1b, 4k, 64k, 256k, 512k, 1024k payload size • type = Concurrency, Connections, Throughput
metric	• HTTP Total Throughput (Kbps); • HTTP Simulated Users; • HTTP Concurrent Connections; • HTTP Connection Rate; • HTTP Transaction Rate
test purpose	The vFW RFC3511 tests measure performance characteristics of the SUT by sending the HTTP traffic from uplink to downlink TG ports through vFW VNF. The application forwards received traffic based on rules provided by the user in the TC configuration and default rules created by vFW to send traffic from uplink ports to downlink and voice versa.
configuration	The 2 ports RFC3511 test cases are listed below: • tc_baremetal_http_ixload_1024k_Requests-65000 _Concurrency.yaml • tc_baremetal_http_ixload_1b_Requests-65000 _Concurrency.yaml • tc_baremetal_http_ixload_256k_Requests-65000 _Concurrency.yaml • tc_baremetal_http_ixload_4k_Requests-65000 _Concurrency.yaml • tc_baremetal_http_ixload_512k_Requests-65000 _Concurrency.yaml • tc_baremetal_http_ixload_64k_Requests-65000 _Concurrency.yaml • tc_heat_external_http_ixload_1b_Requests-10Gbps _Throughput.yaml • tc_heat_external_http_ixload_1b_Requests-65000 _Concurrency.yaml • tc_heat_external_http_ixload_1b_Requests-65000 _Connections.yaml The 4 ports RFC3511 test cases are listed below: • tc_baremetal_http_ixload_1b_Requests-65000 _Concurrency_4port.yaml
test tool	The vFW is a DPDK application that performs basic filtering for malformed packets and dynamic packet filtering of incoming packets using the connection tracker library.
applicability	The vFW RFC3511 test cases can be configured with different: • http payload sizes; • traffic flows; • rules; Default values exist.
pre-test conditions	For OpenStack test case image (yardstick-samplevnf) needs to be installed into Glance with vFW and DPDK included in it (NSB install)

Yardstick Test Case Description: NSB VPP IPSEC

NSB VPP test for vIPSEC characterization	
test case id	tc_baremetal_rfc2544_ipv4_{crypto_dev}_{crypto_alg} • crypto_dev = HW_cryptodev or SW_cryptodev; • crypto_alg = aes-gcm or cbc-sha1;
metric	• Network Throughput NDR or PDR; • Connections Per Second (CPS); • Latency; • Number of tunnels; • TG Packets Out; • TG Packets In; • VNF Packets Out; • VNF Packets In; • Dropped packets;
test purpose	IPv4 IPsec tunnel mode performance test: • Finds and reports throughput NDR (Non Drop Rate) with zero packet loss tolerance or throughput PDR (Partial Drop Rate) with non-zero packet loss tolerance (LT) expressed in number of packets transmitted. • The IPSEC test cases are implemented to run in baremetal
configuration	The IPSEC test cases are listed below: • tc_baremetal_rfc2544_ipv4_hw_aesgcm_IMIX_trex.yaml • tc_baremetal_rfc2544_ipv4_hw_aesgcm_trex.yaml • tc_baremetal_rfc2544_ipv4_hw_cbcsha1_IMIX_trex.yaml • tc_baremetal_rfc2544_ipv4_hw_cbcsha1_trex.yaml • tc_baremetal_rfc2544_ipv4_sw_aesgcm_IMIX_trex.yaml • tc_baremetal_rfc2544_ipv4_sw_aesgcm_trex.yaml • tc_baremetal_rfc2544_ipv4_sw_cbcsha1_IMIX_trex.yaml • tc_baremetal_rfc2544_ipv4_sw_cbcsha1_trex.yaml Test duration is set as 500sec for each test. Packet size set as 64 bytes or higher. Number of tunnels set as 1 or higher. Number of connections set as 1 or higher These can be configured
test tool	Vector Packet Processing (VPP) The VPP platform is an extensible framework that provides out-of-the-box production quality switch/router functionality. Its high performance, proven technology, its modularity and, flexibility and rich feature set
applicability	This VPP IPSEC test cases can be configured with different: • packet sizes; • test durations; • tolerated loss; • crypto device type; • number of physical cores; • number of tunnels; • number of connections; • encryption algorithms - integrity algorithm;
pre-test conditions	Default values exist. For Baremetal tests cases VPP and DPDK must be installed in the hosts where the test is executed. The pod.yaml file must have the necessary system and NIC information

Yardstick Test Case Description: NSB VIMS

NSB VIMS test for vIMS characterization	
test case id	tc_vims_{context}_sipp • context = baremetal or heat;
metric	• Successful registrations per second; • Total number of active registrations per server; • Successful de-registrations per second; • Successful session establishments per second; • Total number of active sessions per server; • Mean session setup time; • Successful re-registrations per second;
test purpose	The vIMS test handles registration rate, call rate, round trip delay, and message statistics of vIMS system. The vIMS test cases are implemented to run in baremetal and heat context default configuration.
configuration	The vIMS test cases are listed below: • tc_vims_baremetal_sipp.yaml • tc_vims_heat_sipp.yaml Each test runs one time and collects all the KPIs. The configuration of vIMS and SIPp can be changed in each test.
test tool	SIPp SIPp is an application that can simulate SIP scenarios, can generate RTP traffic and used for vIMS characterization.
applicability	The SIPp test cases can be configured with different: • number of accounts; • the call per second (cps) of SIP test; • the holding time; • RTP configuratioin;
pre-test conditions	For Openstack test case, only vIMS is deployed by external heat template, SIPp needs pod.yaml file with the necessary system and NIC information For Baremetal tests cases SIPp and vIMS must be installed in the hosts where the test is executed. The pod.yaml file must have the necessary system and NIC information
test sequence	description and expected result
step 1	For Baremetal test: The TG and VNF are started on the hosts based on the pod file. For Heat test: One host VM for vIMS is booted, based on the test flavor. Another host for SIPp is booted as traffic generator, based on pod.yaml file
step 2	Yardstick is connected with the vIMS and SIPp via ssh. The test will resolve the topology, instantiate the vIMS and SIPp and collect the KPIs/metrics.
step 3	The SIPp will run scenario tests with parameters configured in test case files (tc_vims_baremetal_sipp.yaml and tc_vims_heat_sipp.yaml files). This is done until the KPIs of SIPp are within an acceptable threshold.
step 4	In Baremetal test: The test quits the application. In Heat test: The host VM of vIMS is deleted on test completion.
test verdict	The test case will collect the KPIs and plot on Grafana.

Yardstick Test Case Description: NSB vCMTS

NSB Pktgen test for vCMTS characterization	
test case id	tc_vcmts_k8s_pktgen
metric	• Upstream Processing (Per Service Group); • Downstream Processing (Per Service Group); • Upstream Throughput; • Downstream Throughput; • Platform Metrics; • Power Consumption; • Upstream Throughput Time Series; • Downstream Throughput Time Series; • System Summary;
test purpose	• The vCMTS test handles service groups and packet generation containers setup, and metrics collection. • The vCMTS test case is implemented to run in Kubernetes environment with vCMTS pre-installed.
configuration	The vCMTS test case configurable values are listed below • **num_sg: Number of service groups (Upstream/Downstream** container pairs). • num_tg: Number of Pktgen containers. • vcmtsd_image: vCMTS container image (feat/perf). • qat_on: QAT status (true/false). num_sg and num_tg values should be configured in the test case file and in the topology file.
test tool	Intel vCMTS Reference Dataplane Reference implementation of a DPDK-based vCMTS (DOCSIS MAC) dataplane in a Kubernetes-orchestrated Linux Container environment.
applicability	This test cases can be configured with different: • Number of service groups • Number of Pktgen instances • QAT offloading • Feat/Perf Images for performance or features (more data collection) Default values exist.
pre-test conditions	Intel vCMTS Reference Dataplane should be installed and runnable on 2 nodes Kubernetes environment with modifications to the containers to allow yardstick ssh access, and the ConfigMaps from the original vCMTS package deployed.
test sequence	description and expected result
step 1	Yardstick is connected to the Kubernetes Master node using the configuration file in /etc/kubernetes/admin.yaml
step 2	The TG containers are created and started on the traffic generator server (Master node), While the VNF containers are created and started on the data plan server.
step 3	Yardstick is connected with the TG and VNF by using ssh. to start vCMTS-d, and Pktgen.
step 4	Yardstick connects to the running Pktgen instances to start generating traffic using the configurations from:

2.18 Glossary

API Application Programming Interface

Barometer OPNFV NFVi Service Assurance project. Barometer upstreams changes to collectd, OpenStack, etc to improve features related to NFVi monitoring and service assurance. More info on: https://opnfv-barometer.readthedocs.io/en/latest/

collectd collectd is a system statistics collection daemon. More info on: https://collectd.org/

context A context describes the environment in which a yardstick testcase will be run. It can refer to a pre-provisioned environment, or an environment that will be set up using OpenStack or Kubernetes.

Docker Docker provisions and manages containers. Yardstick and many other OPNFV projects are deployed in containers. Docker is required to launch the containerized versions of these projects.

DPDK Data Plane Development Kit

DPI Deep Packet Inspection

DSCP Differentiated Services Code Point

flavor A specification of virtual resources used by OpenStack in the creation of a VM instance.

Grafana A visualization tool, used in Yardstick to retrieve test data from InfluxDB and display it. Grafana works by defining dashboards, which are combinations of visualization panes (e.g. line charts and gauges) and forms that assist the user in formulating SQL-like queries for InfluxDB. More info on: https://grafana.com/

IGMP Internet Group Management Protocol

InfluxDB One of the Dispatchers supported by Yardstick, it allows test results to be reported to a time-series database. More info on: https://www.influxdata.com/

IOPS Input/Output Operations Per Second A performance measurement used to benchmark storage devices.

KPI Key Performance Indicator

Kubernetes k8s Kubernetes is an open-source container-orchestration system for automating deployment, scaling and management of containerized applications. It is one of the contexts supported in Yardstick.

MPLS Multiprotocol Label Switching

NFV Network Function Virtualization NFV is an initiative to take network services which were traditionally run on proprietary, dedicated hardware, and virtualize them to run on general purpose hardware.

NFVI Network Function Virtualization Infrastructure The servers, routers, switches, etc on which the NFV system runs.

NIC Network Interface Controller

NSB Network Services Benchmarking. A subset of Yardstick features concerned with NFVI and VNF characterization.

OpenStack OpenStack is a cloud operating system that controls pools of compute, storage, and networking resources. OpenStack is an open source project licensed under the Apache License 2.0.

PBFS Packet Based per Flow State

PROX Packet pROcessing eXecution engine

QoS Quality of Service The ability to guarantee certain network or storage requirements to satisfy a Service Level Agreement (SLA) between an application provider and end users. Typically includes performance requirements like networking bandwidth, latency, jitter correction, and reliability as well as storage performance in Input/Output Operations Per Second (IOPS), throttling agreements, and performance expectations at peak load

runner The part of a Yardstick testcase that determines how the test will be run (e.g. for x iterations, y seconds or until state z is reached). The runner also determines when the metrics are collected/reported.

SampleVNF OPNFV project providing a repository of reference VNFs. More info on: https://opnfv-samplevnf.readthedocs.io/en/latest/

scenario The part of a Yardstick testcase that describes each test step.

SLA Service Level Agreement An SLA is an agreement between a service provider and a customer to provide a certain level of service/performance.

SR-IOV Single Root IO Virtualization A specification that, when implemented by a physical PCIe device, enables it to appear as multiple separate PCIe devices. This enables multiple virtualized guests to share direct access to the physical device.

SUT System Under Test

testcase A task in Yardstick; the yaml file that is read by Yardstick to determine how to run a test.

ToS Type of Service

VLAN Virtual LAN (Local Area Network)

VM Virtual Machine An operating system instance that runs on top of a hypervisor. Multiple VMs can run at the same time on the same physical host.

VNF Virtual Network Function

VNFC Virtual Network Function Component

2.19 References

2.19.1 OPNFV

- Parser wiki: https://wiki.opnfv.org/display/parser
- Pharos wiki: https://wiki.opnfv.org/display/pharos
- Yardstick CI: https://build.opnfv.org/ci/view/yardstick/
- Yardstick and ETSI TST001 presentation: https://wiki.opnfv.org/display/yardstick/Yardstick?preview=%2F2925202%2F2925205%2Fopnfv_summit_-_bridging_opnfv_and_etsi.pdf
- Yardstick Project presentation: https://wiki.opnfv.org/display/yardstick/Yardstick?preview=%2F2925202%2F2925208%2Fopnfv_summit_-_yardstick_project.pdf
- Yardstick wiki: https://wiki.opnfv.org/display/yardstick

2.19.2 References used in Test Cases

- cachestat: https://github.com/brendangregg/perf-tools/tree/master/fs
- cirros-image: https://download.cirros-cloud.net
- cyclictest: https://rt.wiki.kernel.org/index.php/Cyclictest
- DPDKpktgen: https://github.com/Pktgen/Pktgen-DPDK/
- DPDK supported NICs: http://core.dpdk.org/supported/
- fdisk: http://www.tldp.org/HOWTO/Partition/fdisk_partitioning.html

- fio: https://bluestop.org/files/fio/HOWTO.txt
- free: http://manpages.ubuntu.com/manpages/trusty/en/man1/free.1.html
- iperf3: https://iperf.fr/
- iostat: https://linux.die.net/man/1/iostat
- Lmbench man-pages: http://manpages.ubuntu.com/manpages/trusty/lat_mem_rd.8.html
- Memory bandwidth man-pages: http://manpages.ubuntu.com/manpages/trusty/bw_mem.8.html
- mpstat man-pages: http://manpages.ubuntu.com/manpages/trusty/man1/mpstat.1.html
- netperf: https://hewlettpackard.github.io/netperf/
- pktgen: https://www.kernel.org/doc/Documentation/networking/pktgen.txt
- RAMspeed: http://alasir.com/software/ramspeed/
- sar: https://linux.die.net/man/1/sar
- SR-IOV: https://wiki.openstack.org/wiki/SR-IOV-Passthrough-For-Networking
- Storperf: https://wiki.opnfv.org/display/storperf/Storperf
- unixbench: https://github.com/kdlucas/byte-unixbench/tree/master/UnixBench

2.19.3 Research

- NCSRD: http://www.demokritos.gr/?lang=en
- T-NOVA: http://www.t-nova.eu/
- T-NOVA Results: http://www.t-nova.eu/results/

2.19.4 Standards

- ETSI NFV: https://www.etsi.org/technologies-clusters/technologies/nfv
- ETSI GS-NFV TST 001: https://www.etsi.org/deliver/etsi_gs/NFV-TST/001_099/001/01.01.01_60/gs_NFV-TST001v010101p.pdf
- RFC2544: https://www.ietf.org/rfc/rfc2544.txt

CHAPTER

THREE

YARDSTICK DEVELOPER GUIDE

3.1 Introduction

Yardstick is a project dealing with performance testing. Yardstick produces its own test cases but can also be considered as a framework to support feature project testing.

Yardstick developed a test API that can be used by any OPNFV project. Therefore there are many ways to contribute to Yardstick.

You can:

- Develop new test cases
- Review codes
- Develop Yardstick API / framework
- Develop Yardstick grafana dashboards and Yardstick reporting page
- Write Yardstick documentation

This developer guide describes how to interact with the Yardstick project. The first section details the main working areas of the project. The Second part is a list of "How to" to help you to join the Yardstick family whatever your field of interest is.

3.1.1 Where can I find some help to start?

This guide is made for you. You can have a look at the user guide. There are also references on documentation, video tutorials, tips in the project wiki page. You can also directly contact us by mail with `#yardstick` or `[yardstick]` prefix in the subject at `opnfv-tech-discuss@lists.opnfv.org` or on the IRC channel `#opnfv-yardstick`.

3.2 Yardstick developer areas

3.2.1 Yardstick framework

Yardstick can be considered as a framework. Yardstick is released as a docker file, including tools, scripts and a CLI to prepare the environement and run tests. It simplifies the integration of external test suites in CI pipelines and provides commodity tools to collect and display results.

Since Danube, test categories (also known as tiers) have been created to group similar tests, provide consistant sub-lists and at the end optimize test duration for CI (see How To section).

The definition of the tiers has been agreed by the testing working group.

The tiers are:

- smoke
- features
- components
- performance
- vnf

3.3 How Todos?

3.3.1 How Yardstick works?

The installation and configuration of the Yardstick is described in the user guide.

3.3.2 How to work with test cases?

Sample Test cases

Yardstick provides many sample test cases which are located at `samples` directory of repo.

Sample test cases are designed with the following goals:

1. Helping user better understand Yardstick features (including new feature and new test capacity).
2. Helping developer to debug a new feature and test case before it is offically released.
3. Helping other developers understand and verify the new patch before the patch is merged.

Developers should upload their sample test cases as well when they are uploading a new patch which is about the Yardstick new test case or new feature.

OPNFV Release Test cases

OPNFV Release test cases are located at `yardstick/tests/opnfv/test_cases`. These test cases are run by OPNFV CI jobs, which means these test cases should be more mature than sample test cases. OPNFV scenario owners can select related test cases and add them into the test suites which represent their scenario.

Test case Description File

This section will introduce the meaning of the Test case description file. we will use ping.yaml as a example to show you how to understand the test case description file. This `yaml` file consists of two sections. One is `scenarios`, the other is `context`.:

```
---
  # Sample benchmark task config file
  # measure network latency using ping

  schema: "yardstick:task:0.1"

```

(continues on next page)

(continued from previous page)

```
{% set provider = provider or none %}
{% set physical_network = physical_network or 'physnet1' %}
{% set segmentation_id = segmentation_id or none %}
scenarios:
-
  type: Ping
  options:
    packetsize: 200
  host: athena.demo
  target: ares.demo

  runner:
    type: Duration
    duration: 60
    interval: 1

  sla:
    max_rtt: 10
    action: monitor

context:
  name: demo
  image: yardstick-image
  flavor: yardstick-flavor
  user: ubuntu

  placement_groups:
    pgrp1:
      policy: "availability"

  servers:
    athena:
      floating_ip: true
      placement: "pgrp1"
    ares:
      placement: "pgrp1"

  networks:
    test:
      cidr: '10.0.1.0/24'
      {% if provider == "vlan" %}
      provider: {{provider}}
      physical_network: {{physical_network}}
        {% if segmentation_id %}
      segmentation_id: {{segmentation_id}}
        {% endif %}
     {% endif %}
```

The `contexts` section is the description of pre-condition of testing. As `ping.yaml` shows, you can configure the image, flavor, name, affinity and network of Test VM (servers), with this section, you will get a pre-condition env for Testing. Yardstick will automatically setup the stack which are described in this section. Yardstick converts this section to heat template and sets up the VMs with heat-client (Yardstick can also support to convert this section to Kubernetes template to setup containers).

In the examples above, two Test VMs (athena and ares) are configured by keyword `servers`. `flavor` will determine how many vCPU, how much memory for test VMs. As `yardstick-flavor` is a basic flavor which will be automatically created when you run command `yardstick env prepare`. `yardstick-flavor` is `1 vCPU 1G`

RAM, 3G Disk. image is the image name of test VMs. If you use cirros.3.5.0, you need fill the username of this image into user. The policy of placement of Test VMs have two values (affinity and availability). availability means anti-affinity. In the network section, you can configure which provider network and physical_network you want Test VMs to use. You may need to configure segmentation_id when your network is vlan.

Moreover, you can configure your specific flavor as below, Yardstick will setup the stack for you.

```
flavor:
  name: yardstick-new-flavor
  vcpus: 12
  ram: 1024
  disk: 2
```

Besides default Heat context, Yardstick also allows you to setup two other types of context. They are Node and Kubernetes.

```
context:
  type: Kubernetes
  name: k8s
```

and

```
context:
  type: Node
  name: LF
```

The scenarios section is the description of testing steps, you can orchestrate the complex testing step through scenarios.

Each scenario will do one testing step. In one scenario, you can configure the type of scenario (operation), runner type and sla of the scenario.

For TC002, We only have one step, which is Ping from host VM to target VM. In this step, we also have some detailed operations implemented (such as ssh to VM, ping from VM1 to VM2. Get the latency, verify the SLA, report the result).

If you want to get this implementation details implement, you can check with the scenario.py file. For Ping scenario, you can find it in Yardstick repo (yardstick/yardstick/benchmark/scenarios/networking/ping.py).

After you select the type of scenario (such as Ping), you will select one type of runner, there are 4 types of runner. Iteration and Duration are the most commonly used, and the default is Iteration.

For Iteration, you can specify the iteration number and interval of iteration.

```
runner:
  type: Iteration
  iterations: 10
  interval: 1
```

That means Yardstick will repeat the Ping test 10 times and the interval of each iteration is one second.

For Duration, you can specify the duration of this scenario and the interval of each ping test.

```
runner:
  type: Duration
  duration: 60
  interval: 10
```

That means Yardstick will run the ping test as loop until the total time of this scenario reaches 60s and the interval of each loop is ten seconds.

SLA is the criterion of this scenario. This depends on the scenario. Different scenarios can have different SLA metric.

How to write a new test case

Yardstick already provides a library of testing steps (i.e. different types of scenario).

Basically, what you need to do is to orchestrate the scenario from the library.

Here, we will show two cases. One is how to write a simple test case, the other is how to write a quite complex test case.

Write a new simple test case

First, you can image a basic test case description as below.

Storage Performance	
metric	IOPS (Average IOs performed per second), Throughput (Average disk read/write bandwidth rate), Latency (Average disk read/write latency)
test purpose	The purpose of TC005 is to evaluate the IaaS storage performance with regards to IOPS, throughput and latency.
test description	fio test is invoked in a host VM on a compute blade, a job file as well as parameters are passed to fio and fio will start doing what the job file tells it to do.
configuration	file: opnfv_yardstick_tc005.yaml IO types is set to read, write, randwrite, randread, rw. IO block size is set to 4KB, 64KB, 1024KB. fio is run for each IO type and IO block size scheme, each iteration runs for 30 seconds (10 for ramp time, 20 for runtime). For SLA, minimum read/write iops is set to 100, minimum read/write throughput is set to 400 KB/s, and maximum read/write latency is set to 20000 usec.
applicability	This test case can be configured with different: • IO types; • IO block size; • IO depth; • ramp time; • test duration. Default values exist. SLA is optional. The SLA in this test case serves as an example. Considerably higher throughput and lower latency are expected. However, to cover most configurations, both baremetal and fully virtualized ones, this value should be possible to achieve and acceptable for black box testing. Many heavy IO applications start to suffer badly if the read/write bandwidths are lower than this.
pre-test conditions	The test case image needs to be installed into Glance with fio included in it. No POD specific requirements have been identified.
test sequence	description and expected result
step 1	A host VM with fio installed is booted.
step 2	Yardstick is connected with the host VM by using ssh. 'fio_benchmark' bash script is copyied from Jump Host to the host VM via the ssh tunnel.
step 3	'fio_benchmark' script is invoked. Simulated IO operations are started. IOPS, disk read/write bandwidth and latency are recorded and checked against the SLA. Logs are produced and stored. Result: Logs are stored.
step 4	The host VM is deleted.
test verdict	Fails only if SLA is not passed, or if there is a test case execution problem.

TODO

3.3.3 How can I contribute to Yardstick?

If you are already a contributor of any OPNFV project, you can contribute to Yardstick. If you are totally new to OPNFV, you must first create your Linux Foundation account, then contact us in order to declare you in the repository database.

We distinguish 2 levels of contributors:

- the standard contributor can push patch and vote +1/0/-1 on any Yardstick patch
- The commitor can vote -2/-1/0/+1/+2 and merge

Yardstick commitors are promoted by the Yardstick contributors.

Gerrit & JIRA introduction

OPNFV uses Gerrit for web based code review and repository management for the Git Version Control System. You can access OPNFV Gerrit. Please note that you need to have Linux Foundation ID in order to use OPNFV Gerrit. You can get one from this link.

OPNFV uses JIRA for issue management. An important principle of change management is to have two-way traceability between issue management (i.e. JIRA) and the code repository (via Gerrit). In this way, individual commits can be traced to JIRA issues and we also know which commits were used to resolve a JIRA issue.

If you want to contribute to Yardstick, you can pick a issue from Yardstick's JIRA dashboard or you can create you own issue and submit it to JIRA.

Install Git and Git-reviews

Installing and configuring Git and Git-Review is necessary in order to submit code to Gerrit. The Getting to the code page will provide you with some help for that.

Verify your patch locally before submitting

Once you finish a patch, you can submit it to Gerrit for code review. A developer sends a new patch to Gerrit will trigger patch verify job on Jenkins CI. The yardstick patch verify job includes python pylint check, unit test and code coverage test. Before you submit your patch, it is recommended to run the patch verification in your local environment first.

Open a terminal window and set the project's directory to the working directory using the `cd` command. Assume that `YARDSTICK_REPO_DIR` is the path to the Yardstick project folder on your computer:

```
cd $YARDSTICK_REPO_DIR
```

Verify your patch:

```
tox
```

It is used in CI but also by the CLI.

For more details on `tox` and tests, please refer to the *Running tests* and *working with tox* sections below, which describe the different available environments.

Submit the code with Git

Tell Git which files you would like to take into account for the next commit. This is called 'staging' the files, by placing them into the staging area, using the `git add` command (or the synonym `git stage` command):

```
git add $YARDSTICK_REPO_DIR/samples/sample.yaml
```

Alternatively, you can choose to stage all files that have been modified (that is the files you have worked on) since the last time you generated a commit, by using the *-a* argument:

```
git add -a
```

Git won't let you push (upload) any code to Gerrit if you haven't pulled the latest changes first. So the next step is to pull (download) the latest changes made to the project by other collaborators using the `pull` command:

```
git pull
```

Now that you have the latest version of the project and you have staged the files you wish to push, it is time to actually commit your work to your local Git repository:

```
git commit --signoff -m "Title of change"

Test of change that describes in high level what was done. There is a lot of
documentation in code so you do not need to repeat it here.

JIRA: YARDSTICK-XXX
```

The message that is required for the commit should follow a specific set of rules. This practice allows to standardize the description messages attached to the commits, and eventually navigate among the latter more easily.

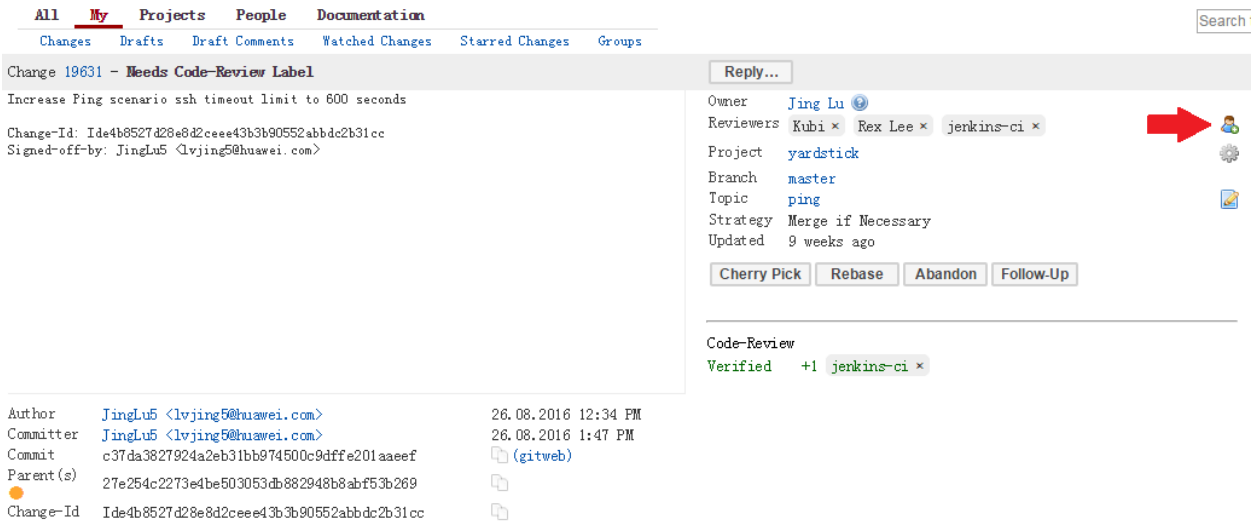
This document happened to be very clear and useful to get started with that.

Push the code to Gerrit for review

Now that the code has been comitted into your local Git repository the following step is to push it online to Gerrit for it to be reviewed. The command we will use is `git review`:

```
git review
```

This will automatically push your local commit into Gerrit. You can add Yardstick committers and contributors to review your codes.



You can find a list Yardstick people here, or use the `yardstick-reviewers` and `yardstick-committers` groups in gerrit.

Modify the code under review in Gerrit

At the same time the code is being reviewed in Gerrit, you may need to edit it to make some changes and then send it back for review. The following steps go through the procedure.

Once you have modified/edited your code files under your IDE, you will have to stage them. The `git status` command is very helpful at this point as it provides an overview of Git's current state:

```
git status
```

This command lists the files that have been modified since the last commit.

You can now stage the files that have been modified as part of the Gerrit code review addition/modification/improvement using `git add` command. It is now time to commit the newly modified files, but the objective here is not to create a new commit, we simply want to inject the new changes into the previous commit. You can achieve that with the '–amend' option on the `git commit` command:

```
git commit --amend
```

If the commit was successful, the `git status` command should not return the updated files as about to be commited.

The final step consists in pushing the newly modified commit to Gerrit:

```
git review
```

3.4 Backporting changes to stable branches

During the release cycle, when master and the `stable/<release>` branch have diverged, it may be necessary to backport (cherry-pick) changes top the `stable/<release>` branch once they have merged to master. These changes should be identified by the committers reviewing the patch. Changes should be backported **as soon as possible** after merging of the original code.

..note:: Besides the commit and review process below, the Jira tick must be updated to add dual release versions and indicate that the change is to be backported.

The process for backporting is as follows:

- Committer A merges a change to master (process for normal changes).
- Committer A cherry-picks the change to `stable/<release>` branch (if the bug has been identified for backporting).
- The original author should review the code and verify that it still works (and give a `+1`).
- Committer B reviews the change, gives a `+2` and merges to `stable/<release>`.

A backported change needs a `+1` and a `+2` from a committer who didn't propose the change (i.e. minimum 3 people involved).

3.5 Development guidelines

This section provides guidelines and best practices for feature development and bug fixing in Yardstick.

In general, bug fixes should be submitted as a single patch.

When developing larger features, all commits on the local topic branch can be submitted together, by running `git review` on the tip of the branch. This creates a chain of related patches in gerrit.

Each commit should contain one logical change and the author should aim for no more than 300 lines of code per commit. This helps to make the changes easier to review.

Each feature should have the following:

- Feature/bug fix code
- Unit tests (both positive and negative)
- Functional tests (optional)
- Sample testcases (if applicable)
- Documentation
- Update to release notes

3.5.1 Coding style

Please follow the OpenStack Style Guidelines for code contributions (the section on Internationalization (i18n) Strings is not applicable).

When writing commit message, the OPNFV coding guidelines on git commit message style should also be used.

3.5.2 Running tests

Once your patch has been submitted, a number of tests will be run by Jenkins CI to verify the patch. Before submitting your patch, you should run these tests locally. You can do this using `tox`, which has a number of different test environments defined in `tox.ini`. Calling `tox` without any additional arguments runs the default set of tests (unit tests, functional tests, coverage and pylint).

If some tests are failing, you can save time and select test environments individually, by passing one or more of the following command-line options to `tox`:

- `-e py27`: Unit tests using Python 2.7
- `-e py3`: Unit tests using Python 3

- `-e pep8`: Linter and style checks on updated files
- `-e functional`: Functional tests using Python 2.7
- `-e functional-py3`: Functional tests using Python 3
- `-e coverage`: Code coverage checks

Note: You need to stage your changes prior to running coverage for those changes to be checked.

In addition to the tests run by Jenkins (listed above), there are a number of other test environments defined.

- `-e pep8-full`: Linter and style checks are run on the whole repo (not just on updated files)
- `-e os-requirements`: Check that the requirements are compatible with OpenStack requirements.

Working with tox

`tox` uses virtualenv to create isolated Python environments to run the tests in. The test environments are located at `.tox/<environment_name>` e.g. `.tox/py27`.

If requirements are changed, you will need to recreate the tox test environment to make sure the new requirements are installed. This is done by passing the additional `-r` command-line option to `tox`:

```
tox -r -e ...
```

This can also be achieved by deleting the test environments manually before running `tox`:

```
rm -rf .tox/<environment_name>
rm -rf .tox/py27
```

3.5.3 Writing unit tests

For each change submitted, a set of unit tests should be submitted, which should include both positive and negative testing.

In order to help identify which tests are needed, follow the guidelines below.

- In general, there should be a separate test for each branching point, return value and input set.
- Negative tests should be written to make sure exceptions are raised and/or handled appropriately.

The following convention should be used for naming tests:

```
test_<method_name>_<some_comment>
```

The comment gives more information on the nature of the test, the side effect being checked, or the parameter being modified:

```
test_my_method_runtime_error
test_my_method_invalid_credentials
test_my_method_param1_none
```

Mocking

The `mock` library is used for unit testing to stub out external libraries.

The following conventions are used in Yardstick:

- Use `mock.patch.object` instead of `mock.patch`.
- When naming mocked classes/functions, use `mock_<class_and_function_name>` e.g. `mock_subprocess_call`
- Avoid decorating classes with mocks. Apply the mocking in `setUp()`:

```
@mock.patch.object(ssh, 'SSH')
class MyClassTestCase(unittest.TestCase):
```

should be:

```
class MyClassTestCase(unittest.TestCase):
    def setUp(self):
        self._mock_ssh = mock.patch.object(ssh, 'SSH')
        self.mock_ssh = self._mock_ssh.start()

        self.addCleanup(self._stop_mocks)

    def _stop_mocks(self):
        self._mock_ssh.stop()
```

3.6 Plugins

For information about Yardstick plugins, refer to the chapter **Installing a plug-in into Yardstick** in the user guide.

3.7 Introduction

This document describes the steps to create a new NSB PROX test based on existing PROX functionalities. NSB PROX provides is a simple approximation of an operation and can be used to develop best practices and TCO models for Telco customers, investigate the impact of new Intel compute, network and storage technologies, characterize performance, and develop optimal system architectures and configurations.

NSB PROX Supports Baremetal, Openstack and standalone configuration.

Contents

3.8 Prerequisites

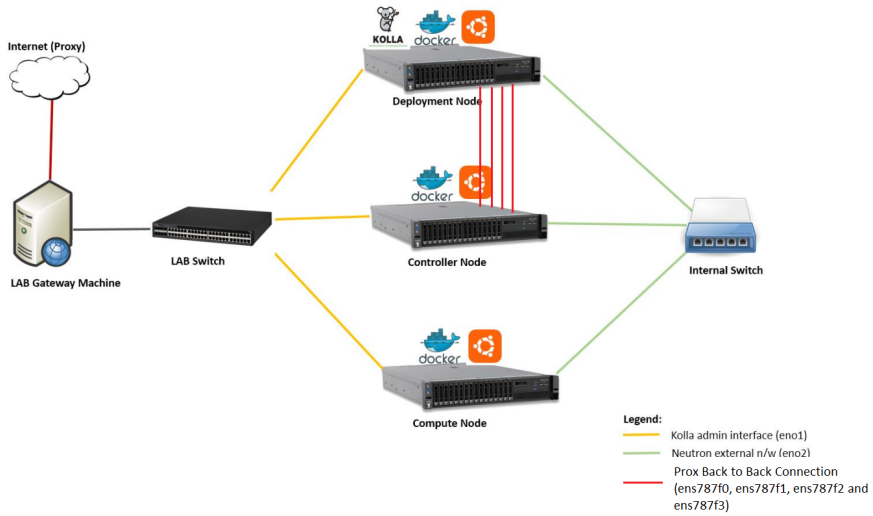
In order to integrate PROX tests into NSB, the following prerequisites are required.

- A working knowledge of Yardstick. See yardstick wiki page.
- A working knowledge of PROX. See Prox documentation.
- Knowledge of Openstack. See openstack wiki page.
- Knowledge of how to use Grafana. See grafana getting started.
- How to Deploy InfluxDB & Grafana. See grafana deployment.
- How to use Grafana in OPNFV/Yardstick. See opnfv grafana dashboard.
- How to install NSB. See NSB Installation

3.9 Sample Prox Test Hardware Architecture

The following is a diagram of a sample NSB PROX Hardware Architecture for both NSB PROX on Bare metal and on Openstack.

In this example when running yardstick on baremetal, yardstick will run on the deployment node, the generator will run on the deployment node and the SUT(SUT) will run on the Controller Node.



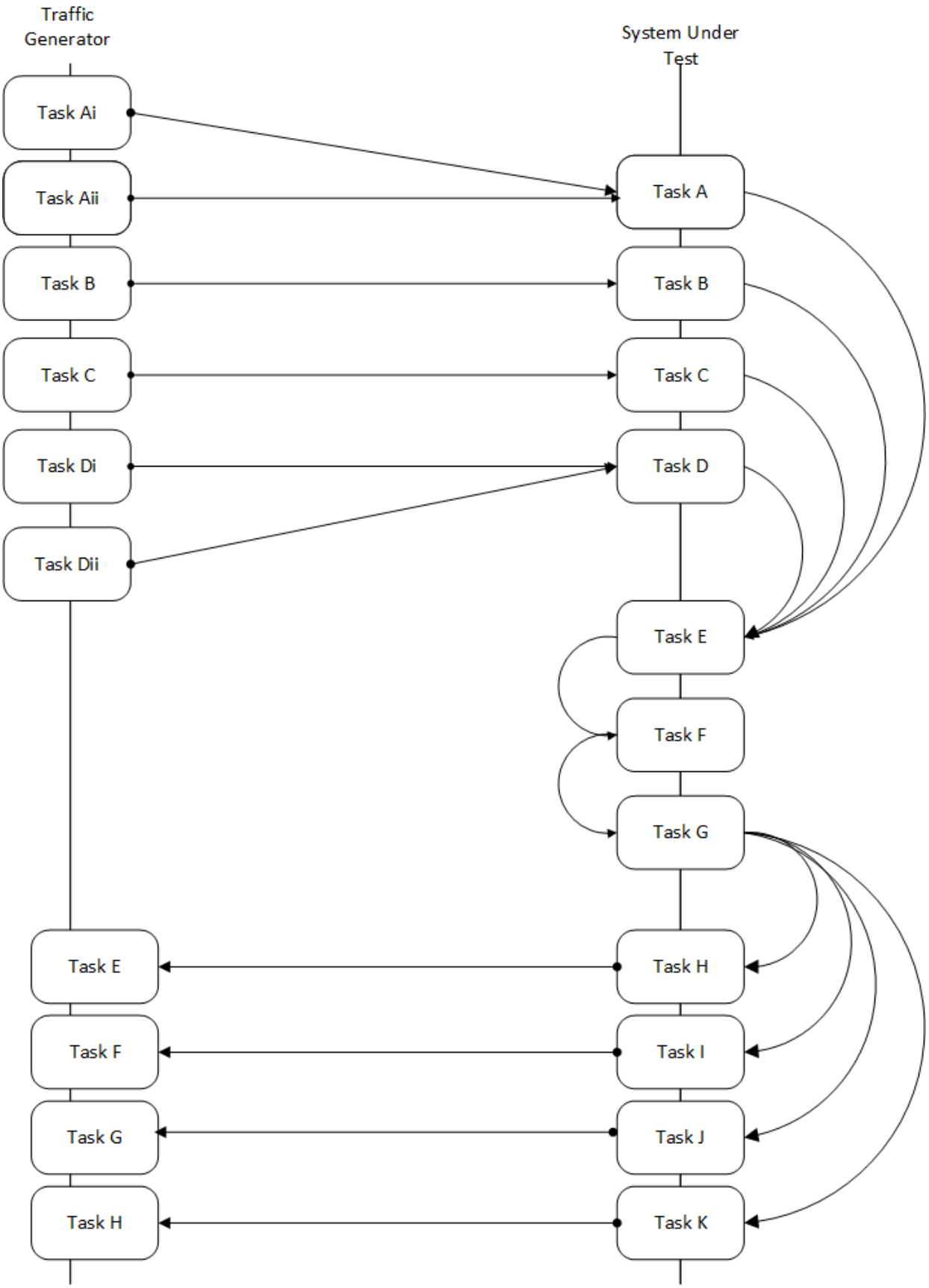
3.10 Prox Test Architecture

In order to create a new test, one must understand the architecture of the test.

A NSB Prox test architecture is composed of:

- A traffic generator. This provides blocks of data on 1 or more ports to the SUT. The traffic generator also consumes the result packets from the system under test.
- A SUT consumes the packets generated by the packet generator, and applies one or more tasks to the packets and return the modified packets to the traffic generator.

  This is an example of a sample NSB PROX test architecture.


Traffic Generator
System Under Test
Task Ai
Task Aii
Task B
Task C
Task Di
Task Dii
Task A
Task B
Task C
Task D
Task E
Task F
Task G
Task E
Task F
Task G
Task H
Task H
Task I
Task J
Task K

This diagram is of a sample NSB PROX test application.

- Traffic Generator
  - Generator Tasks - Composted of 1 or more tasks (It is possible to have multiple tasks sending packets to same port No. See Tasks Ai and Aii plus Di and Dii)
    - Task Ai - Generates Packets on Port 0 of Traffic Generator and send to Port 0 of SUT Port 0
    - Task Aii - Generates Packets on Port 0 of Traffic Generator and send to Port 0 of SUT Port 0
    - Task B - Generates Packets on Port 1 of Traffic Generator and send to Port 1 of SUT Port 1
    - Task C - Generates Packets on Port 2 of Traffic Generator and send to Port 2 of SUT Port 2
    - Task Di - Generates Packets on Port 3 of Traffic Generator and send to Port 3 of SUT Port 3
    - Task Dii - Generates Packets on Port 0 of Traffic Generator and send to Port 0 of SUT Port 0
  - Verifier Tasks - Composed of 1 or more tasks which receives packets from SUT
    - Task E - Receives packets on Port 0 of Traffic Generator sent from Port 0 of SUT Port 0
    - Task F - Receives packets on Port 1 of Traffic Generator sent from Port 1 of SUT Port 1
    - Task G - Receives packets on Port 2 of Traffic Generator sent from Port 2 of SUT Port 2
    - Task H - Receives packets on Port 3 of Traffic Generator sent from Port 3 of SUT Port 3
- SUT
  - Receiver Tasks - Receives packets from generator - Composed on 1 or more tasks which consume the packs sent from Traffic Generator
    - Task A - Receives Packets on Port 0 of System-Under-Test from Traffic Generator Port 0, and forwards packets to Task E
    - Task B - Receives Packets on Port 1 of System-Under-Test from Traffic Generator Port 1, and forwards packets to Task E
    - Task C - Receives Packets on Port 2 of System-Under-Test from Traffic Generator Port 2, and forwards packets to Task E
    - Task D - Receives Packets on Port 3 of System-Under-Test from Traffic Generator Port 3, and forwards packets to Task E
  - Processing Tasks - Composed of multiple tasks in series which carry out some processing on received packets before forwarding to the task.
    - Task E - This receives packets from the Receiver Tasks, carries out some operation on the data and forwards to result packets to the next task in the sequence - Task F
    - Task F - This receives packets from the previous Task - Task E, carries out some operation on the data and forwards to result packets to the next task in the sequence - Task G
    - Task G - This receives packets from the previous Task - Task F and distributes the result packages to the Transmitter tasks
  - Transmitter Tasks - Composed on 1 or more tasks which send the processed packets back to the Traffic Generator
    - Task H - Receives Packets from Task G of System-Under-Test and sends packets to Traffic Generator Port 0
    - Task I - Receives Packets from Task G of System-Under-Test and sends packets to Traffic Generator Port 1

* Task J - Receives Packets from Task G of System-Under-Test and sends packets to Traffic Generator Port 2
* Task K - Receives Packets From Task G of System-Under-Test and sends packets to Traffic Generator Port 3

3.11 NSB Prox Test

A NSB Prox test is composed of the following components :-

- Test Description File. Usually called `tc_prox_<context>_<test>-<ports>.yaml` where
  - <context> is either `baremetal` or `heat_context`
  - <test> is the a one or 2 word description of the test.
  - <ports> is the number of ports used

  Example tests `tc_prox_baremetal_l2fwd-2.yaml` or `tc_prox_heat_context_vpe-4.yaml`. This file describes the components of the test, in the case of openstack the network description and server descriptions, in the case of baremetal the hardware description location. It also contains the name of the Traffic Generator, the SUT config file and the traffic profile description, all described below. See *Test Description File*
- Traffic Profile file. Example `prox_binsearch.yaml`. This describes the packet size, tolerated loss, initial line rate to start traffic at, test interval etc See *Traffic Profile File*
- Traffic Generator Config file. Usually called `gen_<test>-<ports>.cfg`.

  This describes the activity of the traffic generator
  - What each core of the traffic generator does,
  - The packet of data sent by a core on a port of the traffic generator to the system under test
  - What core is used to wait on what port for data from the system under test.

  Example traffic generator config file `gen_l2fwd-4.cfg` See *Traffic Generator Config file*
- SUT Config file. Usually called `handle_<test>-<ports>.cfg`.

  This describes the activity of the SUTs
  - What each core of the does,
  - What cores receives packets from what ports
  - What cores perform operations on the packets and pass the packets onto another core
  - What cores receives packets from what cores and transmit the packets on the ports to the Traffic Verifier tasks of the Traffic Generator.

  Example traffic generator config file `handle_l2fwd-4.cfg` See *SUT Config File*
- NSB PROX Baremetal Configuration file. Usually called `prox-baremetal-<ports>.yaml`
  - <ports> is the number of ports used

  This is required for baremetal only. This describes hardware, NICs, IP addresses, Network drivers, usernames and passwords. See *Baremetal Configuration File*
- Grafana Dashboard. Usually called `Prox_<context>_<test>-<port>-<DateAndTime>.json` where
  - <context> Is `BM`,‘‘heat‘‘,‘‘ovs_dpdk‘‘ or `sriov`

- <test> Is the a one or 2 word description of the test.
- <port> is the number of ports used express as `2Port` or `4Port`
- <DateAndTime> is the Date and Time expressed as a string.

Example grafana dashboard `Prox_BM_L2FWD-4Port-1507804504588.json`

Other files may be required. These are test specific files and will be covered later.

3.11.1 Test Description File

Here we will discuss the test description for baremetal, openstack and standalone.

3.11.2 Test Description File for Baremetal

This section will introduce the meaning of the Test case description file. We will use `tc_prox_baremetal_l2fwd-2.yaml` as an example to show you how to understand the test description file.

```
schema: "yardstick:task:0.1"

scenarios:
-
  type: NSPerf
  traffic_profile: ../../traffic_profiles/prox_binsearch.yaml
  topology: prox-tg-topology-4.yaml

  nodes:
    tg__0: tg_0.yardstick
    vnf__0: vnf_0.yardstick

  options:
    interface_speed_gbps: 10

    vnf__0:
      collectd:
        interval: 1
      prox_path: /opt/nsb_bin/prox
      prox_config: "configs/handle_l2fwd-4.cfg"
      prox_args:
        "-t": ""

    tg__0:
      collectd:
        interval: 1
      prox_path: /opt/nsb_bin/prox
      prox_config: "configs/gen_l2fwd-4.cfg"
      prox_args:
        "-e": ""
        "-t": ""


  runner:
    type: ProxDuration
    # sampling interval
    interval: 1
    # sampled : yes OR sampled: no (DEFAULT yes)
    sampled: yes
    # we kill after duration, independent of test duration, so set this high
    duration: 3100
    # Confirmation attempts
    confirmation: 1

context:
  type: Node
  name: yardstick
  nfvi_type: baremetal
  file: prox-baremetal-4.yaml
```

Now let's examine the components of the file in detail

1. `traffic_profile` - This specifies the traffic profile for the test. In this case `prox_binsearch.yaml` is used. See *Traffic Profile File*
2. **`topology` - This is either `prox-tg-topology-1.yaml` or** `prox-tg-topology-2.yaml` or `prox-tg-topology-4.yaml` depending on number of ports required.

3. `nodes` - This names the Traffic Generator and the System under Test. Does not need to change.
4. `interface_speed_gbps` - This is an optional parameter. If not present the system defaults to 10Gbps. This defines the speed of the interfaces.
5. `collectd` - (Optional) This specifies we want to collect NFVI statistics like CPU Utilization,
6. `prox_path` - Location of the Prox executable on the traffic generator (Either baremetal or Openstack Virtual Machine)
7. `prox_config` - This is the `SUT Config File`. In this case it is `handle_l2fwd-2.cfg`

   A number of additional parameters can be added. This example is for VPE:

```
options:
  interface_speed_gbps: 10

  traffic_config:
    tolerated_loss: 0.01
    test_precision: 0.01
    packet_sizes: [64]
    duration: 30
    lower_bound: 0.0
    upper_bound: 100.0

  vnf__0:
    prox_path: /opt/nsb_bin/prox
    prox_config: ``configs/handle_vpe-4.cfg``
    prox_args:
      ``-t``: ````
    prox_files:
      ``configs/vpe_ipv4.lua`` : ````
      ``configs/vpe_dscp.lua`` : ````
      ``configs/vpe_cpe_table.lua`` : ````
      ``configs/vpe_user_table.lua`` : ````
      ``configs/vpe_rules.lua`` : ````
    prox_generate_parameter: True

``interface_speed_gbps`` - this specifies the speed of the interface
in Gigabits Per Second. This is used to calculate pps(packets per second).
If the interfaces are of different speeds, then this specifies the speed
of the slowest interface. This parameter is optional. If omitted the
interface speed defaults to 10Gbps.

``traffic_config`` - This allows the values here to override the values in
in the traffic_profile file. e.g. "prox_binsearch.yaml". Values provided
here override values provided in the "traffic_profile" section of the
traffic_profile file. Some, all or none of the values can be provided here.

The values describes the packet size, tolerated loss, initial line rate
to start traffic at, test interval etc See `Traffic Profile File`_

``prox_files`` - this specified that a number of addition files
need to be provided for the test to run correctly. This files
could provide routing information,hashing information or a
hashing algorithm and ip/mac information.

``prox_generate_parameter`` - this specifies that the NSB application
is required to provide information to the nsb Prox in the form
```

(continues on next page)

(continued from previous page)

```
of a file called ``parameters.lua``, which contains information
retrieved from either the hardware or the openstack configuration.
```

8. `prox_args` - this specifies the command line arguments to start prox. See prox command line.
9. `prox_config` - This specifies the Traffic Generator config file.
10. `runner` - This is set to `ProxDuration` - This specifies that the test runs for a set duration. Other runner types are available but it is recommend to use `ProxDuration`. The following parameters are supported

    `interval` - (optional) - This specifies the sampling interval. Default is 1 sec

    `sampled` - (optional) - This specifies if sampling information is required. Default `no`

    `duration` - This is the length of the test in seconds. Default is 60 seconds.

    `confirmation` - This specifies the number of confirmation retests to be made before deciding to increase or decrease line speed. Default 0.

11. `context` - This is `context` for a 2 port Baremetal configuration.

    If a 4 port configuration was required then file `prox-baremetal-4.yaml` would be used. This is the NSB Prox baremetal configuration file.

3.11.3 Traffic Profile File

This describes the details of the traffic flow. In this case `prox_binsearch.yaml` is used.

```
schema: "nsb:traffic_profile:0.1"

name:           prox_binsearch                                    <= 1
description:    Binary search for max no-drop throughput over given packet sizes

traffic_profile:
  traffic_type: ProxBinSearchProfile                              <= 2
  tolerated_loss: 0.001                                           <= 3
  test_precision: 0.1                                             <= 4
  packet_sizes: [64, 128, 256, 512, 1024, 1280, 1518]             <= 5
  duration: 10                                                    <= 6
  lower_bound: 0.0                                                <= 7
  upper_bound: 100.0                                              <= 8
```

1. `name` - The name of the traffic profile. This name should match the name specified in the `traffic_profile` field in the Test Description File.
2. `traffic_type` - This specifies the type of traffic pattern generated, This name matches class name of the traffic generator. See:

```
network_services/traffic_profile/prox_binsearch.py class␣
↪ProxBinSearchProfile(ProxProfile)
```

   In this case it lowers the traffic rate until the number of packets sent is equal to the number of packets received (plus a tolerated loss). Once it achieves this it increases the traffic rate in order to find the highest rate with no traffic loss.

   Custom traffic types can be created by creating a new traffic profile class.

3. `tolerated_loss` - This specifies the percentage of packets that can be lost/dropped before we declare success or failure. Success is Transmitted-Packets from Traffic Generator is greater than or equal to packets received by Traffic Generator plus tolerated loss.

4. `test_precision` - This specifies the precision of the test results. For some tests the success criteria may never be achieved because the test precision may be greater than the successful throughput. For finer results increase the precision by making this value smaller.
5. `packet_sizes` - This specifies the range of packets size this test is run for.
6. `duration` - This specifies the sample duration that the test uses to check for success or failure.
7. `lower_bound` - This specifies the test initial lower bound sample rate. On success this value is increased.
8. `upper_bound` - This specifies the test initial upper bound sample rate. On success this value is decreased.

Other traffic profiles exist eg prox_ACL.yaml which does not compare what is received with what is transmitted. It just sends packet at max rate.

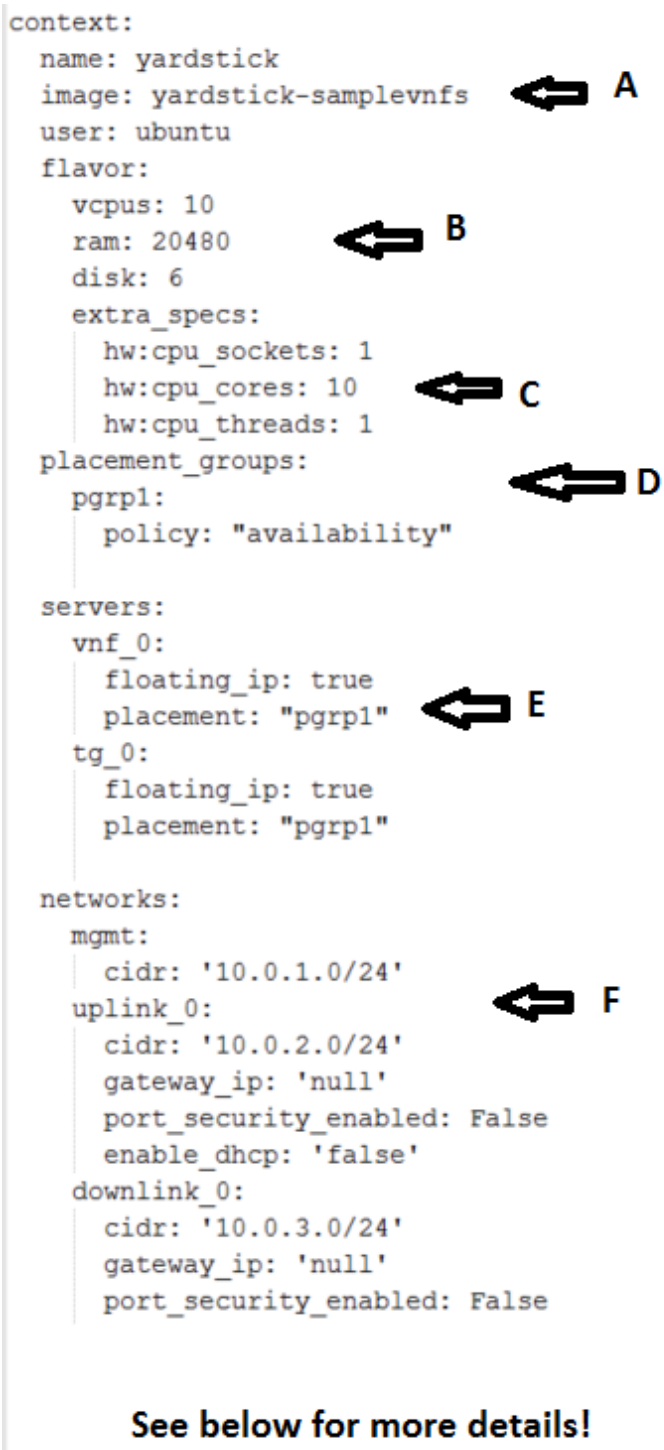
It is possible to create custom traffic profiles with by creating new file in the same folder as prox_binsearch.yaml. See this prox_vpe.yaml as example:

```
schema: ``nsb:traffic_profile:0.1``

name:            prox_vpe
description:     Prox vPE traffic profile

traffic_profile:
  traffic_type: ProxBinSearchProfile
  tolerated_loss: 100.0 #0.001
  test_precision: 0.01
# The minimum size of the Ethernet frame for the vPE test is 68 bytes.
  packet_sizes: [68]
  duration: 5
  lower_bound: 0.0
  upper_bound: 100.0
```

3.11.4 Test Description File for Openstack

We will use `tc_prox_heat_context_l2fwd-2.yaml` as a example to show you how to understand the test description file.

```
---
schema: "yardstick:task:0.1"

scenarios:
-
  type: NSPerf
  traffic_profile: ../../traffic_profiles/prox_binsearch.yaml <= 1
  topology: prox-tg-topology-4.yaml <= 2

  nodes:
    tg__0: tg_0.yardstick <= 3
    vnf__0: vnf_0.yardstick

  options:

    interface_speed_gbps: 10 <= 4

    vnf__0:
      prox_path: /opt/nsb_bin/prox <= 5
      prox_config: "configs/handle_l2fwd-4.cfg" <= 6
      prox_args:
        "-t": "" <= 7

    tg__0:
      prox_path: /opt/nsb_bin/prox <= 8
      prox_config: "configs/gen_l2fwd-4.cfg"
      prox_args:
        "-e": ""
        "-t": ""

  runner:
    type: ProxDuration <= 9
    # sampling interval
    interval: 1
    # sampled : yes OR sampled: no (DEFAULT yes)
    sampled: yes
    # we kill after duration, independent of test duration, so set this high
    duration: 3100
    # Confirmation attempts
    confirmation: 1
```


context:
name: yardstick
image: yardstick-samplevnfs
A
user: ubuntu
flavor:
vcpus: 10
ram: 20480
B
disk: 6
extra_specs:
hw:cpu_sockets: 1
hw:cpu_cores: 10
C
hw:cpu_threads: 1
placement_groups:
D
pgrp1:
policy: "availability"
servers:
vnf_0:
floating_ip: true
placement: "pgrp1"
E
tg_0:
floating_ip: true
placement: "pgrp1"
networks:
mgmt:
cidr: '10.0.1.0/24'
uplink_0:
F
cidr: '10.0.2.0/24'
gateway_ip: 'null'
port_security_enabled: False
enable_dhcp: 'false'
downlink_0:
cidr: '10.0.3.0/24'
gateway_ip: 'null'
port_security_enabled: False
See below for more details!

Now lets examine the components of the file in detail

Sections 1 to 9 are exactly the same in Baremetal and in Heat. Section `10` is replaced with sections A to F. Section 10 was for a baremetal configuration file. This has no place in a heat configuration.

A. `image` - yardstick-samplevnfs. This is the name of the image created during the installation of NSB. This is fixed.

B. `flavor` - The flavor is created dynamically. However we could use an already existing flavor if required. In that case the flavor would be named:

```
flavor: yardstick-flavor
```

C. `extra_specs` - This allows us to specify the number of cores sockets and hyperthreading assigned to it. In this case we have 1 socket with 10 codes and no hyperthreading enabled.

D. `placement_groups` - default. Do not change for NSB PROX.

E. `servers` - `tg_0` is the traffic generator and `vnf_0` is the system under test.

F. `networks` - is composed of a management network labeled `mgmt` and one uplink network labeled `uplink_0` and one downlink network labeled `downlink_0` for 2 ports. If this was a 4 port configuration there would be 2 extra downlink ports. See this example from a 4 port l2fwd test.:

```
networks:
  mgmt:
    cidr: '10.0.1.0/24'
  uplink_0:
    cidr: '10.0.2.0/24'
    gateway_ip: 'null'
    port_security_enabled: False
    enable_dhcp: 'false'
  downlink_0:
    cidr: '10.0.3.0/24'
    gateway_ip: 'null'
    port_security_enabled: False
    enable_dhcp: 'false'
  uplink_1:
    cidr: '10.0.4.0/24'
    gateway_ip: 'null'
    port_security_enabled: False
    enable_dhcp: 'false'
  downlink_1:
    cidr: '10.0.5.0/24'
    gateway_ip: 'null'
    port_security_enabled: False
    enable_dhcp: 'false'
```

3.11.5 Test Description File for Standalone

We will use `tc_prox_ovs-dpdk_l2fwd-2.yaml` as a example to show you how to understand the test description file.

```
---
schema: "yardstick:task:0.1"

scenarios:
-
  type: NSPerf
  traffic_profile: ../../traffic_profiles/prox_binsearch.yaml <= 1
  topology: prox-tg-topology-2.yaml  <= 2

  nodes:
    tg__0: tg_0.yardstick
    vnf__0: vnf_0.yardstick  <= 3

  options:
    interface_speed_gbps: 10  <= 4

    vnf__0:
      prox_path: /opt/nsb_bin/prox  <= 5
      prox_config: "configs/handle_l2fwd-2.cfg" <= 6
      prox_args:  <= 7
        "-t": ""

    tg__0:
      prox_path: /opt/nsb_bin/prox
      prox_config: "configs/gen_l2fwd-2.cfg" <= 8
      prox_args:
        "-e": ""
        "-t": ""

  runner:
    type: Duration  <= 9
    # we kill after duration, independent of test duration, so set this high
    duration: 300
```


contexts:
- name: yardstick
type: Node
file: prox_tg_bm.yaml
A
- name: yardstick
type: StandaloneOvsDpdk
B
file: /etc/yardstick/nodes/standalone/host_ovs.yaml
C
vm_deploy: True
D
ovs_properties:
E
version:
ovs: 2.8.0
dpdk: 17.05.2
pmd_threads: 2
ram:
socket_0: 2048
socket_1: 2048
queues: 4
vpath: "/usr/local"
flavor:
F
images: "/var/lib/libvirt/images/yardstick-nsb-image.img"
ram: 16384
extra_specs:
hw:cpu_sockets: 1
hw:cpu_cores: 10
hw:cpu_threads: 2
user: "root"
password: ""
servers:
vnf_0:
network_ports:
mgmt:
cidr: '172.20.2.7/24'
G
xe0:
- uplink_0
H
xe1:
- downlink_0
I
networks:
uplink_0:
port_num: 0
phy_port: "0000:af:00.0"
J
vpci: "0000:00:07.0"
cidr: '152.16.100.10/24'
gateway_ip: '152.16.100.20'
downlink_0:
port_num: 1
phy_port: "0000:af:00.1"
K
vpci: "0000:00:08.0"
cidr: '152.16.40.10/24'
gateway_ip: '152.16.100.20'

Now lets examine the components of the file in detail

Sections 1 to 9 are exactly the same in Baremetal and in Heat. Section `10` is replaced with sections A to F. Section 10 was for a baremetal configuration file. This has no place in a heat configuration.

A. `file` - Pod file for Baremetal Traffic Generator configuration: IP Address, User/Password & Interfaces

B. `type` - This defines the type of standalone configuration. Possible values are `StandaloneOvsDpdk` or `StandaloneSriov`

C. `file` - Pod file for Standalone host configuration: IP Address, User/Password & Interfaces

D. `vm_deploy` - Deploy a new VM or use an existing VM

E. `ovs_properties` - OVS Version, DPDK Version and configuration to use.

F. `flavor`- NSB image generated when installing NSB using ansible-playbook:

```
ram- Configurable RAM for SUT VM
extra_specs
  hw:cpu_sockets - Configurable number of Sockets for SUT VM
  hw:cpu_cores - Configurable number of Cores for SUT VM
  hw:cpu_threads- Configurable number of Threads for SUT VM
```

G. `mgmt` - Management port of the SUT VM. Preconfig needed on TG & SUT host machines. is the system under test.

H. `xe0` - Upline Network port

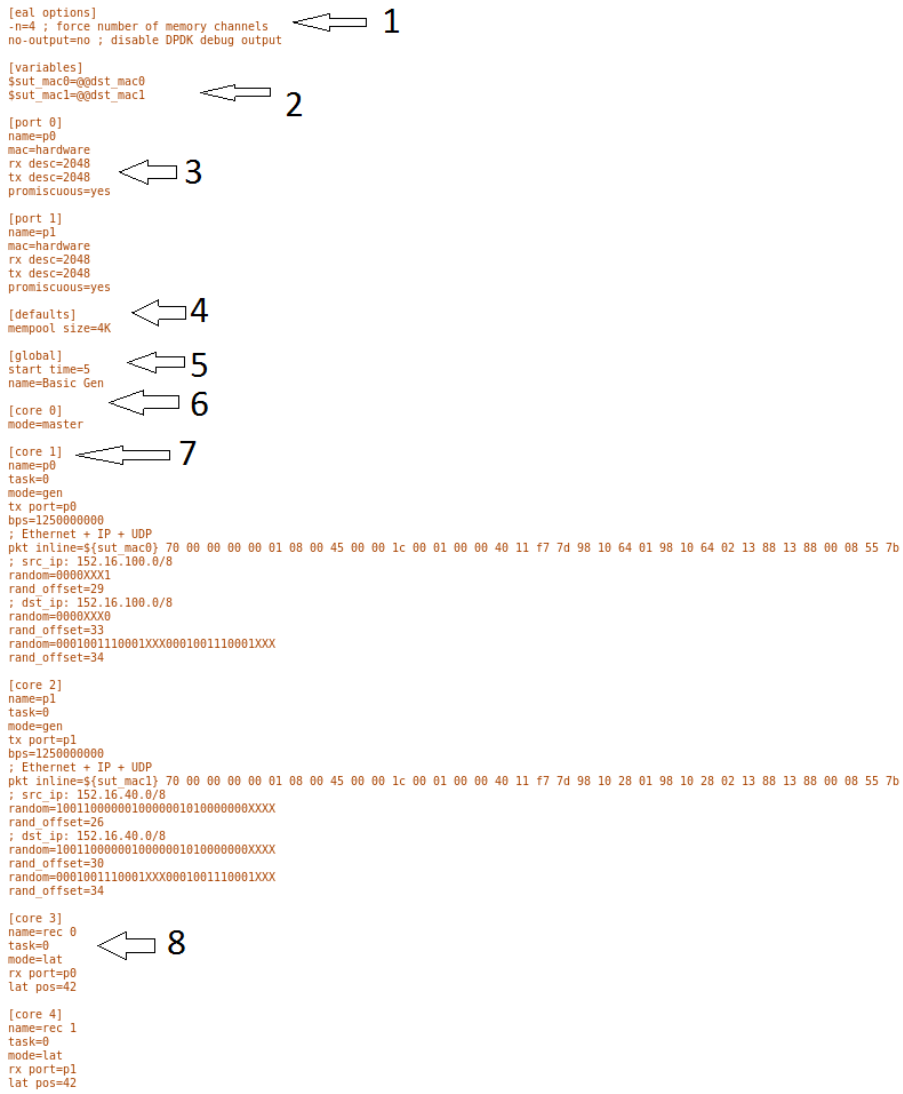
I. `xe1` - Downline Network port

J. `uplink_0` - Uplink Phy port of the NIC on the host. This will be used to create the Virtual Functions.

K. `downlink_0` - Downlink Phy port of the NIC on the host. This will be used to create the Virtual Functions.

3.11.6 Traffic Generator Config file

This section will describe the traffic generator config file. This is the same for both baremetal and heat. See this example of `gen_l2fwd_multiflow-2.cfg` to explain the options.



The configuration file is divided into multiple sections, each of which is used to define some parameters and options.:

```
[eal options]
[variables]
[port 0]
[port 1]
[port .]
[port Z]
[defaults]
[global]
[core 0]
[core 1]
[core 2]
[core .]
[core Z]
```

See prox options for details

Now let's examine the components of the file in detail

1. `[eal options]` - This specified the EAL (Environmental Abstraction Layer) options. These are default

values and are not changed. See dpdk wiki page.

2. `[variables]` - This section contains variables, as the name suggests. Variables for Core numbers, mac addresses, ip addresses etc. They are assigned as a `key = value` where the key is used in place of the value.

> **Caution:** A special case for valuables with a value beginning with `@@`. These values are dynamically updated by the NSB application at run time. Values like MAC address, IP Address etc.

3. `[port 0]` - This section describes the DPDK Port. The number following the keyword `port` usually refers to the DPDK Port Id. usually starting from `0`. Because you can have multiple ports this entry usually repeated. Eg. For a 2 port setup `[port0]` and `[port 1]` and for a 4 port setup `[port 0]`, `[port 1]`, `[port 2]` and `[port 3]`:

```
[port 0]
name=p0
mac=hardware
rx desc=2048
tx desc=2048
promiscuous=yes
```

   a. In this example `name = p0` assigned the name `p0` to the port. Any name can be assigned to a port.

   b. `mac=hardware` sets the MAC address assigned by the hardware to data from this port.

   c. `rx desc=2048` sets the number of available descriptors to allocate for receive packets. This can be changed and can effect performance.

   d. `tx desc=2048` sets the number of available descriptors to allocate for transmit packets. This can be changed and can effect performance.

   e. `promiscuous=yes` this enables promiscuous mode for this port.

4. `[defaults]` - Here default operations and settings can be over written. In this example `mempool size=4K` the number of mbufs per task is altered. Altering this value could effect performance. See prox options for details.

5. `[global]` - Here application wide setting are supported. Things like application name, start time, duration and memory configurations can be set here. In this example.:

```
  [global]
  start time=5
  name=Basic Gen

a. ``start time=5`` Time is seconds after which average
   stats will be started.
b. ``name=Basic Gen`` Name of the configuration.
```

6. `[core 0]` - This core is designated the master core. Every Prox application must have a master core. The master mode must be assigned to exactly one task, running alone on one core.:

```
[core 0]
mode=master
```

7. `[core 1]` - This describes the activity on core 1. Cores can be configured by means of a set of [core #] sections, where # represents either:

   a. an absolute core number: e.g. on a 10-core, dual socket system with hyper-threading, cores are numbered from 0 to 39.

b. PROX allows a core to be identified by a core number, the letter 's', and a socket number.

   It is possible to write a baremetal and an openstack test which use the same traffic generator config file and SUT config file. In this case it is advisable not to use physical core numbering.

   However it is also possible to write NSB Prox tests that have been optimized for a particular hardware configuration. In this case it is advisable to use the core numbering. It is up to the user to make sure that cores from the right sockets are used (i.e. from the socket on which the NIC is attached to), to ensure good performance (EPA).

Each core can be assigned with a set of tasks, each running one of the implemented packet processing modes.:

```
[core 1]
name=p0
task=0
mode=gen
tx port=p0
bps=1250000000
; Ethernet + IP + UDP
pkt inline=${sut_mac0} 70 00 00 00 00 01 08 00 45 00 00 1c 00 01 00 00 40 11 f7␣
↪7d 98 10 64 01 98 10 64 02 13 88 13 88 00 08 55 7b
; src_ip: 152.16.100.0/8
random=0000XXX1
rand_offset=29
; dst_ip: 152.16.100.0/8
random=0000XXX0
rand_offset=33
random=0001001110001XXX0001001110001XXX
rand_offset=34
```

a. `name=p0` - Name assigned to the core.

b. `task=0` - Each core can run a set of tasks. Starting with `0`. Task 1 can be defined later in this core or can be defined in another `[core 1]` section with `task=1` later in configuration file. Sometimes running multiple task related to the same packet on the same physical core improves performance, however sometimes it is optimal to move task to a separate core. This is best decided by checking performance.

c. `mode=gen` - Specifies the action carried out by this task on this core. Supported modes are: classify, drop, gen, lat, genl4, nop, l2fwd, gredecap, greencap, lbpos, lbnetwork, lbqinq, lb5tuple, ipv6_decap, ipv6_encap, qinqdecapv4, qinqencapv4, qos, routing, impair, mirror, unmpls, tagmpls, nat, decapnsh, encapnsh, police, acl Which are :-

   - Classify
   - Drop
   - Basic Forwarding (no touch)
   - L2 Forwarding (change MAC)
   - GRE encap/decap
   - Load balance based on packet fields
   - Symmetric load balancing
   - QinQ encap/decap IPv4/IPv6
   - ARP
   - QoS
   - Routing

- Unmpls
- Nsh encap/decap
- Policing
- ACL

In the traffic generator we expect a core to generate packets (`gen`) and to receive packets & calculate latency (`lat`) This core does `gen` . ie it is a traffic generator.

To understand what each of the modes support please see prox documentation.

d. `tx port=p0` - This specifies that the packets generated are transmitted to port `p0`

e. `bps=1250000000` - This indicates Bytes Per Second to generate packets.

f. `; Ethernet + IP + UDP` - This is a comment. Items starting with `;` are ignored.

g. `pkt inline=${sut_mac0} 70 00 00 00 ...` - Defines the packet format as a sequence of bytes (each expressed in hexadecimal notation). This defines the packet that is generated. This packets begins with the hexadecimal sequence assigned to `sut_mac` and the remainder of the bytes in the string. This packet could now be sent or modified by `random=..` described below before being sent to target.

h. `; src_ip:  152.16.100.0/8` - Comment

i. `random=0000XXX1` - This describes a field of the packet containing random data. This string can be 8,16,24 or 32 character long and represents 1,2,3 or 4 bytes of data. In this case it describes a byte of data. Each character in string can be 0,1 or `X`. 0 or 1 are fixed bit values in the data packet and `X` is a random bit. So random=0000XXX1 generates 00000001(1), 00000011(3), 00000101(5), 00000111(7), 00001001(9), 00001011(11), 00001101(13) and 00001111(15) combinations.

j. `rand_offset=29` - Defines where to place the previously defined random field.

k. `; dst_ip:  152.16.100.0/8` - Comment

l. `random=0000XXX0` - This is another random field which generates a byte of 00000000(0), 00000010(2), 00000100(4), 00000110(6), 00001000(8), 00001010(10), 00001100(12) and 00001110(14) combinations.

m. `rand_offset=33` - Defines where to place the previously defined random field.

n. `random=0001001110001XXX0001001110001XXX` - This is another random field which generates 4 bytes.

o. `rand_offset=34` - Defines where to place the previously defined 4 byte random field.

Core 2 executes same scenario as Core 1. The only difference in this case is that the packets are generated for Port 1.

8. `[core 3]` - This defines the activities on core 3. The purpose of `core 3` and `core 4` is to receive packets sent by the SUT.:

```
[core 3]
name=rec 0
task=0
mode=lat
rx port=p0
lat pos=42
```

a. `name=rec 0` - Name assigned to the core.

b. `task=0` - Each core can run a set of tasks. Starting with `0`. Task 1 can be defined later in this core or can be defined in another `[core 1]` section with `task=1` later in configuration file. Sometimes

running multiple task related to the same packet on the same physical core improves performance, however sometimes it is optimal to move task to a separate core. This is best decided by checking performance.

c. `mode=lat` - Specifies the action carried out by this task on this core. Supported modes are: `acl`, `classify`, `drop`, `gredecap`, `greencap`, `ipv6_decap`, `ipv6_encap`, `l2fwd`, `lbnetwork`, `lbpos`, `lbqinq`, `nop`, `police`, `qinqdecapv4`, `qinqencapv4`, `qos`, `routing`, `impair`, `lb5tuple`, `mirror`, `unmpls`, `tagmpls`, `nat`, `decapnsh`, `encapnsh`, `gen`, `genl4` and `lat`. This task(0) per core(3) receives packets on port.

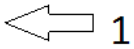
d. `rx port=p0` - The port to receive packets on `Port 0`. Core 4 will receive packets on `Port 1`.

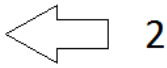
e. `lat pos=42` - Describes where to put a 4-byte timestamp in the packet. Note that the packet length should be longer than `lat pos` + 4 bytes to avoid truncation of the timestamp. It defines where the timestamp is to be read from. Note that the SUT workload might cause the position of the timestamp to change (i.e. due to encapsulation).

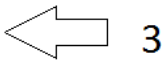
3.11.7 SUT Config File

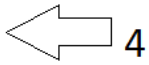
This section will describes the SUT(VNF) config file. This is the same for both baremetal and heat. See this example of `handle_l2fwd_multiflow-2.cfg` to explain the options.

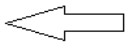
```
[eal options]
-n=4
no-output=no ; disable DPDK debug output

[port 0]
name=if0
mac=hardware
rx desc=2048
tx desc=2048
promiscuous=yes

[port 1]
name=if1
mac=hardware
rx desc=2048
tx desc=2048
promiscuous=yes

[defaults]
mempool size=8K
memcache size=512

[global]
start time=5
name=Handle L2FWD Multiflow (2x)

[core 0]
mode=master

[core 1]
name=none
task=0
mode=l2fwd
dst mac=@@tester_mac1
rx port=if0
tx port=if1
drop=no

[core 2]
name=none
task=0
mode=l2fwd
dst mac=@@tester_mac0
rx port=if1
tx port=if0
drop=no
```

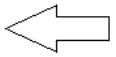












See prox options for details

Now let's examine the components of the file in detail

1. `[eal options]` - same as the Generator config file. This specified the EAL (Environmental Abstraction Layer) options. These are default values and are not changed. See dpdk wiki page.
2. `[port 0]` - This section describes the DPDK Port. The number following the keyword `port` usually refers to the DPDK Port Id. usually starting from `0`. Because you can have multiple ports this entry usually repeated. E.g. For a 2 port setup `[port0]` and `[port 1]` and for a 4 port setup `[port 0]`, `[port 1]`, `[port 2]` and `[port 3]`:

```
[port 0]
name=if0
mac=hardware
rx desc=2048
tx desc=2048
promiscuous=yes
```

   a. In this example `name =if0` assigned the name `if0` to the port. Any name can be assigned to a port.
   b. `mac=hardware` sets the MAC address assigned by the hardware to data from this port.
   c. `rx desc=2048` sets the number of available descriptors to allocate for receive packets. This can be changed and can effect performance.
   d. `tx desc=2048` sets the number of available descriptors to allocate for transmit packets. This can be changed and can effect performance.
   e. `promiscuous=yes` this enables promiscuous mode for this port.

3. `[defaults]` - Here default operations and settings can be over written.:

```
[defaults]
mempool size=8K
memcache size=512
```

   a. In this example `mempool size=8K` the number of mbufs per task is altered. Altering this value could effect performance. See prox options for details.
   b. `memcache size=512` - number of mbufs cached per core, default is 256 this is the cache_size. Altering this value could affect performance.

4. `[global]` - Here application wide setting are supported. Things like application name, start time, duration and memory configurations can be set here. In this example.:

```
  [global]
  start time=5
  name=Basic Gen

a. ``start time=5`` Time is seconds after which average stats will be
   started.
b. ``name=Handle L2FWD Multiflow (2x)`` Name of the configuration.
```

5. `[core 0]` - This core is designated the master core. Every Prox application must have a master core. The master mode must be assigned to exactly one task, running alone on one core.:

```
[core 0]
mode=master
```

6. `[core 1]` - This describes the activity on core 1. Cores can be configured by means of a set of [core #] sections, where # represents either:

   a. an absolute core number: e.g. on a 10-core, dual socket system with hyper-threading, cores are numbered from 0 to 39.

   b. PROX allows a core to be identified by a core number, the letter 's', and a socket number. However NSB PROX is hardware agnostic (physical and virtual configurations are the same) it is advisable no to use physical core numbering.

   Each core can be assigned with a set of tasks, each running one of the implemented packet processing modes.:

```
[core 1]
name=none
task=0
mode=l2fwd
dst mac=@@tester_mac1
rx port=if0
tx port=if1
```

   a. `name=none` - No name assigned to the core.

   b. `task=0` - Each core can run a set of tasks. Starting with `0`. Task 1 can be defined later in this core or can be defined in another `[core 1]` section with `task=1` later in configuration file. Sometimes running multiple task related to the same packet on the same physical core improves performance, however sometimes it is optimal to move task to a separate core. This is best decided by checking performance.

   c. `mode=l2fwd` - Specifies the action carried out by this task on this core. Supported modes are: `acl`, `classify`, `drop`, `gredecap`, `greencap`, `ipv6_decap`, `ipv6_encap`, `l2fwd`, `lbnetwork`, `lbpos`, `lbqinq`, `nop`, `police`, `qinqdecapv4`, `qinqencapv4`, `qos`, `routing`, `impair`, `lb5tuple`, `mirror`, `unmpls`, `tagmpls`, `nat`, `decapnsh`, `encapnsh`, `gen`, `genl4` and `lat`. This code does `l2fwd`. i.e. it does the L2FWD.

   d. `dst mac=@@tester_mac1` - The destination mac address of the packet will be set to the MAC address of `Port 1` of destination device. (The Traffic Generator/Verifier)

   e. `rx port=if0` - This specifies that the packets are received from `Port 0` called if0

   f. `tx port=if1` - This specifies that the packets are transmitted to `Port 1` called if1

   In this example we receive a packet on core on a port, carry out operation on the packet on the core and transmit it on on another port still using the same task on the same core.

   On some implementation you may wish to use multiple tasks, like this.:

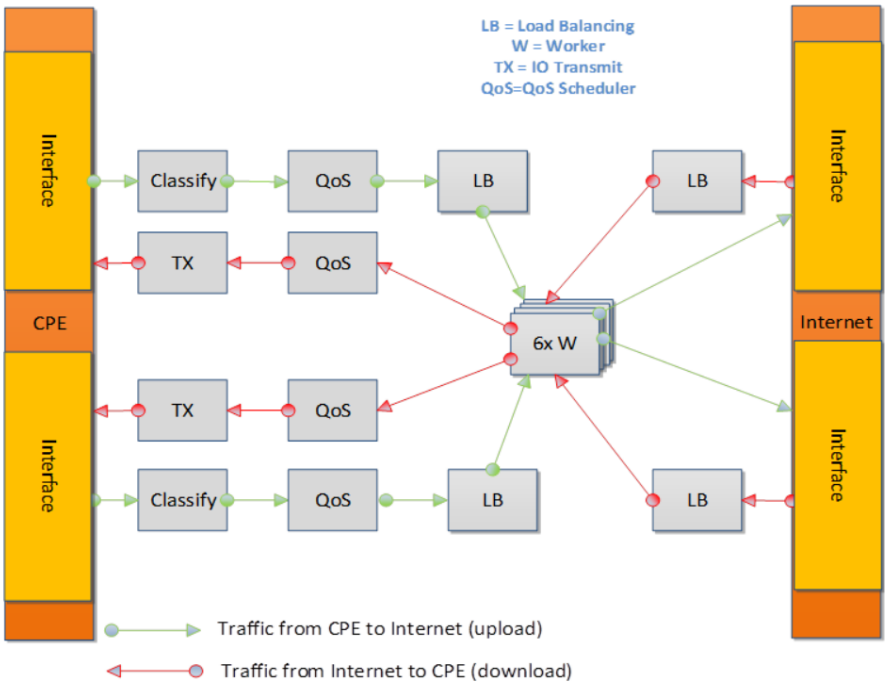
```
[core 1]
name=rx_task
task=0
mode=l2fwd
dst mac=@@tester_p0
rx port=if0
tx cores=1t1
drop=no

name=l2fwd_if0
task=1
mode=nop
rx ring=yes
tx port=if0
drop=no
```

In this example you can see Core 1/Task 0 called `rx_task` receives the packet from if0 and perform the l2fwd. However instead of sending the packet to a port it sends it to a core see `tx cores=1t1`. In this case it sends it to Core 1/Task 1.

Core 1/Task 1 called `l2fwd_if0`, receives the packet, not from a port but from the ring. See `rx ring=yes`. It does not perform any operation on the packet See `mode=none` and sends the packets to `if0` see `tx port=if0`.

It is also possible to implement more complex operations by chaining multiple operations in sequence and using rings to pass packets from one core to another.

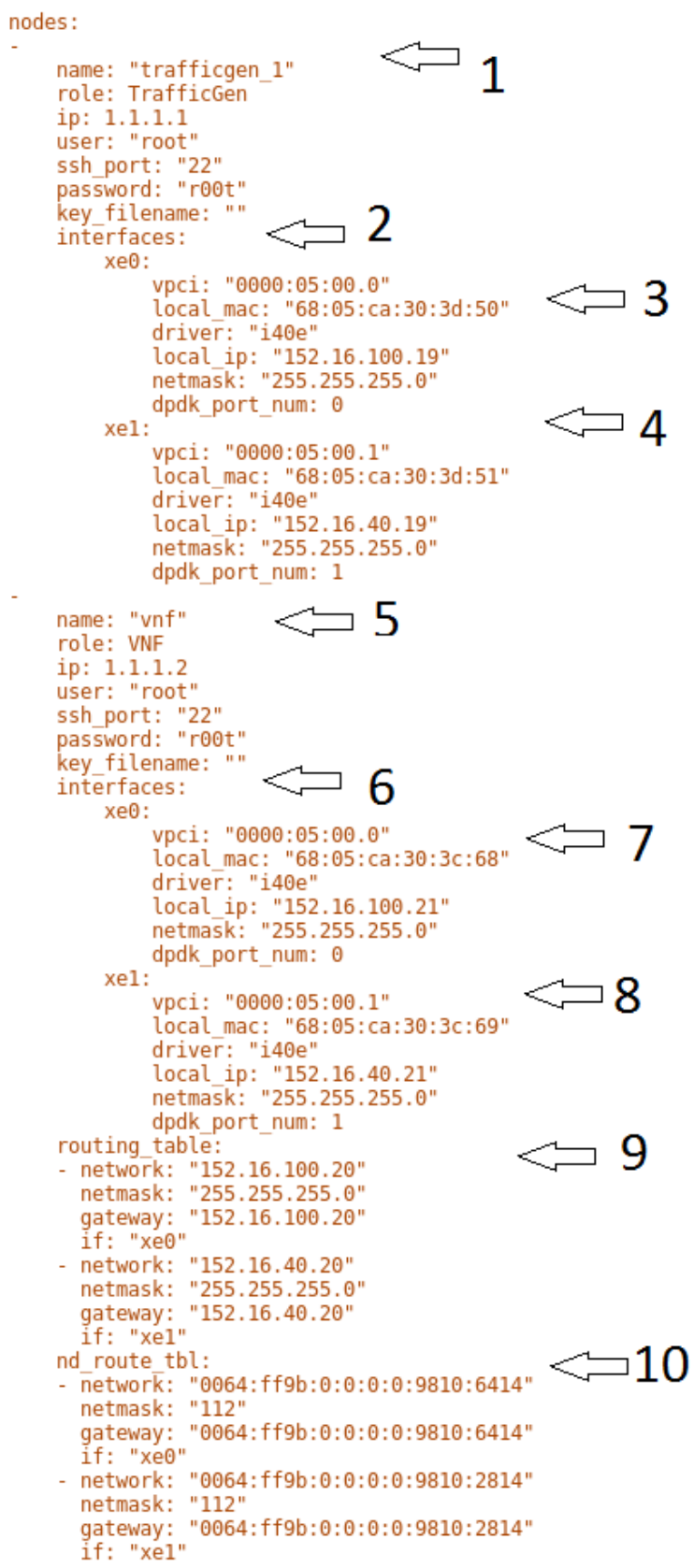
In this example, we show a Broadband Network Gateway (BNG) with Quality of Service (QoS). Communication from task to task is via rings.



3.11.8 Baremetal Configuration File

This is required for baremetal testing. It describes the IP address of the various ports, the Network devices drivers and MAC addresses and the network configuration.

In this example we will describe a 2 port configuration. This file is the same for all 2 port NSB Prox tests on the same platforms/configuration.


nodes:
-
name: "trafficgen_1"
role: TrafficGen
ip: 1.1.1.1
user: "root"
ssh_port: "22"
password: "r00t"
key_filename: ""
interfaces:
xe0:
vpci: "0000:05:00.0"
local_mac: "68:05:ca:30:3d:50"
driver: "i40e"
local_ip: "152.16.100.19"
netmask: "255.255.255.0"
dpdk_port_num: 0
xe1:
vpci: "0000:05:00.1"
local_mac: "68:05:ca:30:3d:51"
driver: "i40e"
local_ip: "152.16.40.19"
netmask: "255.255.255.0"
dpdk_port_num: 1
-
name: "vnf"
role: VNF
ip: 1.1.1.2
user: "root"
ssh_port: "22"
password: "r00t"
key_filename: ""
interfaces:
xe0:
vpci: "0000:05:00.0"
local_mac: "68:05:ca:30:3c:68"
driver: "i40e"
local_ip: "152.16.100.21"
netmask: "255.255.255.0"
dpdk_port_num: 0
xe1:
vpci: "0000:05:00.1"
local_mac: "68:05:ca:30:3c:69"
driver: "i40e"
local_ip: "152.16.40.21"
netmask: "255.255.255.0"
dpdk_port_num: 1
routing_table:
- network: "152.16.100.20"
netmask: "255.255.255.0"
gateway: "152.16.100.20"
if: "xe0"
- network: "152.16.40.20"
netmask: "255.255.255.0"
gateway: "152.16.40.20"
if: "xe1"
nd_route_tbl:
- network: "0064:ff9b:0:0:0:0:9810:6414"
netmask: "112"
gateway: "0064:ff9b:0:0:0:0:9810:6414"
if: "xe0"
- network: "0064:ff9b:0:0:0:0:9810:2814"
netmask: "112"
gateway: "0064:ff9b:0:0:0:0:9810:2814"
if: "xe1"
1
2
3
4
5
6
7
8
9
10

Now let's describe the sections of the file.

1. `TrafficGen` - This section describes the Traffic Generator node of the test configuration. The name of the node `trafficgen_1` must match the node name in the `Test Description File for Baremetal` mentioned earlier. The password attribute of the test needs to be configured. All other parameters can remain as default settings.
2. `interfaces` - This defines the DPDK interfaces on the Traffic Generator.
3. `xe0` is DPDK Port 0. `lspci` and `./dpdk-devbind.py -s` can be used to provide the interface information. `netmask` and `local_ip` should not be changed
4. `xe1` is DPDK Port 1. If more than 2 ports are required then `xe1` section needs to be repeated and modified accordingly.
5. `vnf` - This section describes the SUT of the test configuration. The name of the node `vnf` must match the node name in the `Test Description File for Baremetal` mentioned earlier. The password attribute of the test needs to be configured. All other parameters can remain as default settings
6. `interfaces` - This defines the DPDK interfaces on the SUT
7. `xe0` - Same as 3 but for the `SUT`.
8. `xe1` - Same as 4 but for the `SUT` also.
9. `routing_table` - All parameters should remain unchanged.
10. `nd_route_tbl` - All parameters should remain unchanged.

3.11.9 Grafana Dashboard

The grafana dashboard visually displays the results of the tests. The steps required to produce a grafana dashboard are described here.

a. Configure `yardstick` to use influxDB to store test results. See file `/etc/yardstick/yardstick.conf`.

```
[DEFAULT]
debug = False
# setup multiple dipatcher with comma deperted e.g. file,http
dispatcher = influxdb

[dispatcher_http]
timeout = 5
target = http://127.0.0.1:8000/results

[dispatcher_file]
file_path = /tmp/yardstick.out
max_bytes = 0
backup_count = 0

[dispatcher_influxdb]
timeout = 5
target = http://10.237.222.55:8086
db_name = yardstick
username = root
password = password

[nsb]
trex_path=/opt/nsb_bin/trex/scripts
bin_path=/opt/nsb_bin
trex_client_lib=/opt/nsb_bin/trex_client/stl
```

1. Specify the dispatcher to use influxDB to store results.

2. "target = .. " - Specify location of influxDB to store results. "db_name = yardstick" - name of database. Do not change "username = root" - username to use to store result. (Many tests are run as root) "password = . . . " - Please set to root user password

b. Deploy InfludDB & Grafana. See how to Deploy InfluxDB & Grafana. See grafana deployment.

c. Generate the test data. Run the tests as follows .:

```
yardstick --debug task start tc_prox_<context>_<test>-ports.yaml
```

eg.:

```
yardstick --debug task start tc_prox_heat_context_l2fwd-4.yaml
```

d. Now build the dashboard for the test you just ran. The easiest way to do this is to copy an existing dashboard and rename the test and the field names. The procedure to do so is described here. See opnfv grafana dashboard.

3.12 How to run NSB Prox Test on an baremetal environment

In order to run the NSB PROX test.

1. Install NSB on Traffic Generator node and Prox in SUT. See NSB Installation
2. To enter container:

```
docker exec -it yardstick /bin/bash
```

3. Install baremetal configuration file (POD files)
   a. Go to location of PROX tests in container

   ```
   cd /home/opnfv/repos/yardstick/samples/vnf_samples/nsut/prox
   ```

   b. Install prox-baremetal-2.yam and prox-baremetal-4.yaml for that topology into this directory as per *Baremetal Configuration File*

   c. Install and configure `yardstick.conf`

   ```
   cd /etc/yardstick/
   ```

   Modify /etc/yardstick/yardstick.conf as per *yardstick-config-label*

4. Execute the test. Eg.:

```
yardstick --debug task start ./tc_prox_baremetal_l2fwd-4.yaml
```

3.13 How to run NSB Prox Test on an Openstack environment

In order to run the NSB PROX test.

1. Install NSB on Openstack deployment node. See NSB Installation
2. To enter container:

```
docker exec -it yardstick /bin/bash
```

3. Install configuration file

a. Goto location of PROX tests in container

```
cd /home/opnfv/repos/yardstick/samples/vnf_samples/nsut/prox
```

b. Install and configure `yardstick.conf`

```
cd /etc/yardstick/
```

Modify /etc/yardstick/yardstick.conf as per *yardstick-config-label*

4. Execute the test. Eg.:

```
yardstick --debug task start ./tc_prox_heat_context_l2fwd-4.yaml
```

3.14 Frequently Asked Questions

Here is a list of frequently asked questions.

3.14.1 NSB Prox does not work on Baremetal, How do I resolve this?

If PROX NSB does not work on baremetal, problem is either in network configuration or test file.

1. Verify network configuration. Execute existing baremetal test.:

```
yardstick --debug task start ./tc_prox_baremetal_l2fwd-4.yaml
```

If test does not work then error in network configuration.

a. Check DPDK on Traffic Generator and SUT via:-

```
/root/dpdk-17./usertools/dpdk-devbind.py
```

b. Verify MAC addresses match `prox-baremetal-<ports>.yaml` via `ifconfig` and `dpdk-devbind`

c. Check your eth port is what you expect. You would not be the first person to think that the port your cable is plugged into is ethX when in fact it is ethY. Use ethtool to visually confirm that the eth is where you expect.:

```
ethtool -p ethX
```

A led should start blinking on port. (On both System-Under-Test and Traffic Generator)

d. Check cable.

Install Linux kernel network driver and ensure your ports are `bound` to the driver via `dpdk-devbind`. Bring up port on both SUT and Traffic Generator and check connection.

i) On SUT and on Traffic Generator:

```
ifconfig ethX/enoX up
```

ii) Check link

ethtool ethX/enoX

See `Link detected` if `yes` .... Cable is good. If `no` you have an issue with your cable/port.

2. If existing baremetal works then issue is with your test. Check the traffic generator gen_<test>-<ports>.cfg to ensure it is producing a valid packet.

3.14.2 How do I debug NSB Prox on Baremetal?

1. Execute the test as follows:

```
yardstick --debug task start ./tc_prox_baremetal_l2fwd-4.yaml
```

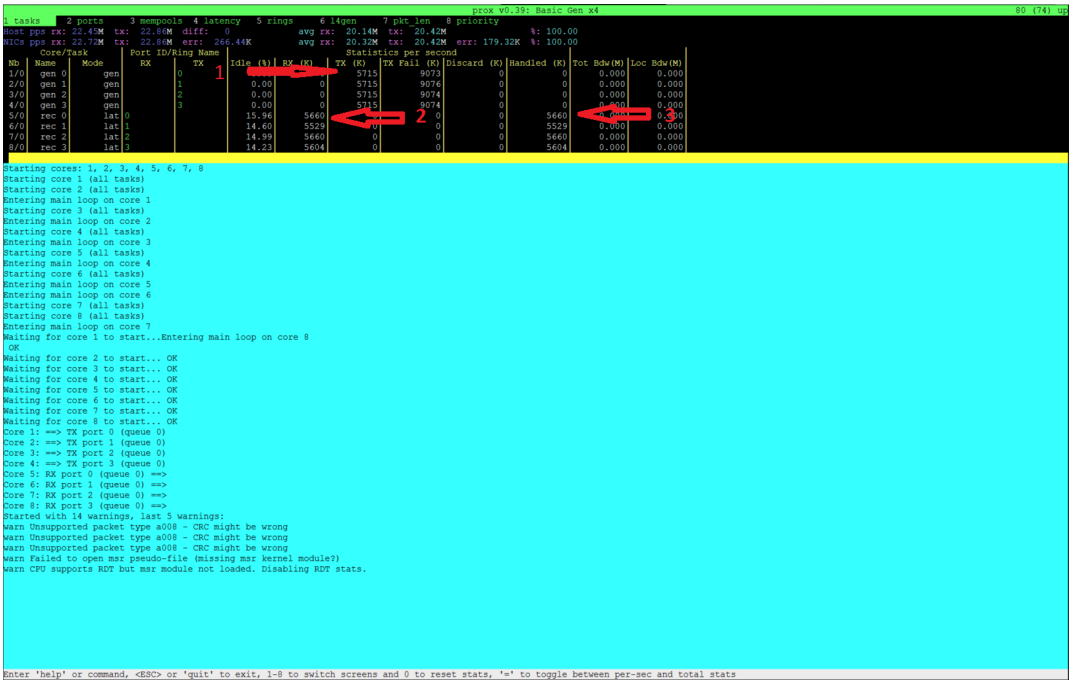
2. Login to Traffic Generator as `root`.:

```
cd
/opt/nsb_bin/prox -f /tmp/gen_<test>-<ports>.cfg
```

3. Login to SUT as `root`.:

```
cd
/opt/nsb_bin/prox -f /tmp/handle_<test>-<ports>.cfg
```

4. Now let's examine the Generator Output. In this case the output of `gen_l2fwd-4.cfg`.



Now let's examine the output

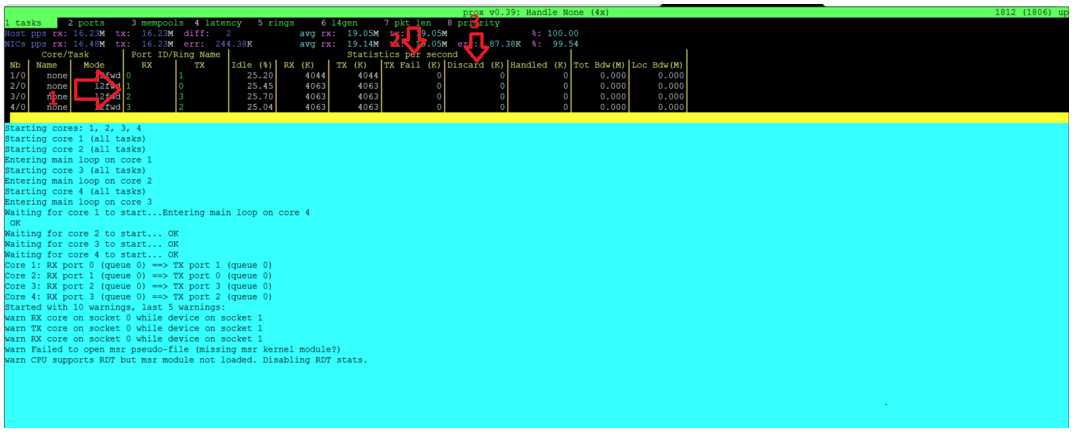
1. Indicates the amount of data successfully transmitted on Port 0
2. Indicates the amount of data successfully received on port 1
3. Indicates the amount of data successfully handled for port 1

It appears what is transmitted is received.

> **Caution:** The number of packets MAY not exactly match because the ports are read in sequence.

> **Caution:** What is transmitted on PORT X may not always be received on same port. Please check the Test scenario.

5. Now lets examine the SUT Output



Now lets examine the output

1. What is received on 0 is transmitted on 1, received on 1 transmitted on 0, received on 2 transmitted on 3 and received on 3 transmitted on 2.
2. No packets are Failed.
3. No packets are discarded.

We can also dump the packets being received or transmitted via the following commands.

```
dump                    Arguments: <core id> <task id> <nb packets>
                        Create a hex dump of <nb_packets> from <task_id> on
↪<core_id> showing how
                        packets have changed between RX and TX.
dump_rx                 Arguments: <core id> <task id> <nb packets>
                        Create a hex dump of <nb_packets> from <task_id> on
↪<core_id> at RX
dump_tx                 Arguments: <core id> <task id> <nb packets>
                        Create a hex dump of <nb_packets> from <task_id> on
↪<core_id> at TX
```

eg.:

```
dump_tx 1 0 1
```

3.14.3 NSB Prox works on Baremetal but not in Openstack. How do I resolve this?

NSB Prox on Baremetal is a lot more forgiving than NSB Prox on Openstack. A badly formed packed may still work with PROX on Baremetal. However on Openstack the packet must be correct and all fields of the header correct. E.g. A packet with an invalid Protocol ID would still work in Baremetal but this packet would be rejected by openstack.

1. Check the validity of the packet.
2. Use a known good packet in your test
3. If using `Random` fields in the traffic generator, disable them and retry.

3.14.4 How do I debug NSB Prox on Openstack?

1. Execute the test as follows:

```
yardstick --debug task start --keep-deploy ./tc_prox_heat_context_l2fwd-4.yaml
```

2. Access docker image if required via:

```
docker exec -it yardstick /bin/bash
```

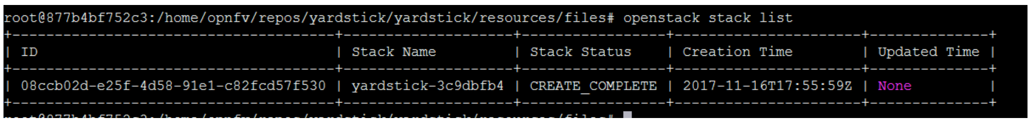
3. Install openstack credentials.

   Depending on your openstack deployment, the location of these credentials may vary. On this platform I do this via:

```
scp root@10.237.222.55:/etc/kolla/admin-openrc.sh .
source ./admin-openrc.sh
```

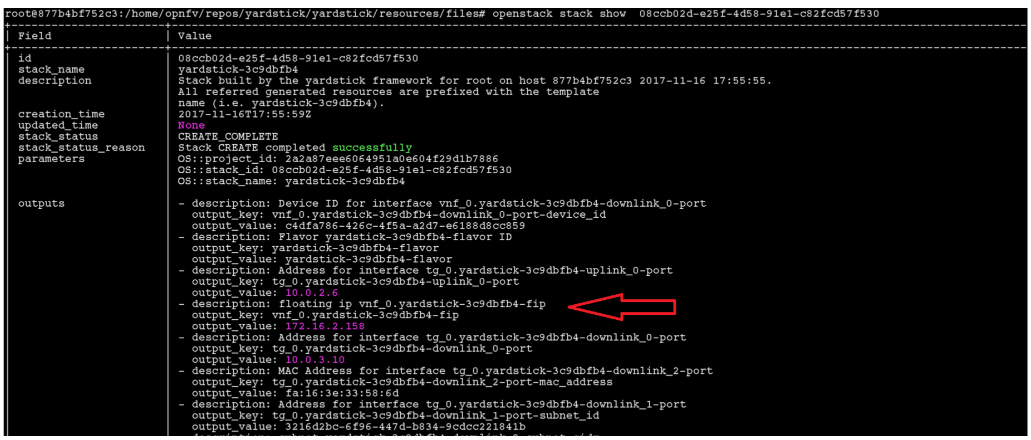
4. List Stack details

   a. Get the name of the Stack.



   b. Get the Floating IP of the Traffic Generator & SUT

   This generates a lot of information. Please note the floating IP of the VNF and the Traffic Generator.



   From here you can see the floating IP Address of the SUT / VNF

   From here you can see the floating IP Address of the Traffic Generator

c. Get ssh identity file

In the docker container locate the identity file.:

```
cd /home/opnfv/repos/yardstick/yardstick/resources/files
ls -lt
```

5. Login to SUT as Ubuntu.:

```
ssh -i ./yardstick_key-01029d1d ubuntu@172.16.2.158
```

Change to root:

```
 sudo su

Now continue as baremetal.
```

6. Login to SUT as Ubuntu.:

```
ssh -i ./yardstick_key-01029d1d ubuntu@172.16.2.156
```

Change to root:

```
 sudo su

Now continue as baremetal.
```

3.14.5 How do I resolve "Quota exceeded for resources"

This usually occurs due to 2 reasons when executing an openstack test.

1. One or more stacks already exists and are consuming all resources. To resolve

```
openstack stack list
```

Response:

```
+--------------------------------------+--------------------+-----------------+---
↪------------------+--------------+
| ID                                   | Stack Name         | Stack Status    |␣
↪Creation Time        | Updated Time |
+--------------------------------------+--------------------+-----------------+---
↪------------------+--------------+
| acb559d7-f575-4266-a2d4-67290b556f15 | yardstick-e05ba5a4 | CREATE_COMPLETE |␣
↪2017-12-06T15:00:05Z | None         |
| 7edf21ce-8824-4c86-8edb-f7e23801a01b | yardstick-08bda9e3 | CREATE_COMPLETE |␣
↪2017-12-06T14:56:43Z | None         |
+--------------------------------------+--------------------+-----------------+---
↪------------------+--------------+
```

In this case 2 stacks already exist.

To remove stack:

```
openstack stack delete yardstick-08bda9e3
Are you sure you want to delete this stack(s) [y/N]? y
```

2. The openstack configuration quotas are too small.

   The solution is to increase the quota. Use below to query existing quotas:

   ```
   openstack quota show
   ```

   And to set quota:

   ```
   openstack quota set <resource>
   ```

3.14.6 Openstack CLI fails or hangs. How do I resolve this?

If it fails due to

```
Missing value auth-url required for auth plugin password
```

Check your shell environment for Openstack variables. One of them should contain the authentication URL

```
OS_AUTH_URL=``https://192.168.72.41:5000/v3``
```

Or similar. Ensure that openstack configurations are exported.

```
cat  /etc/kolla/admin-openrc.sh
```

Result

```
export OS_PROJECT_DOMAIN_NAME=default
export OS_USER_DOMAIN_NAME=default
export OS_PROJECT_NAME=admin
export OS_TENANT_NAME=admin
export OS_USERNAME=admin
export OS_PASSWORD=BwwSEZqmUJA676klr9wa052PFjNkz99tOccS9sTc
export OS_AUTH_URL=http://193.168.72.41:35357/v3
export OS_INTERFACE=internal
export OS_IDENTITY_API_VERSION=3
export EXTERNAL_NETWORK=yardstick-public
```

and visible.

If the Openstack CLI appears to hang, then verify the proxys and `no_proxy` are set correctly. They should be similar to

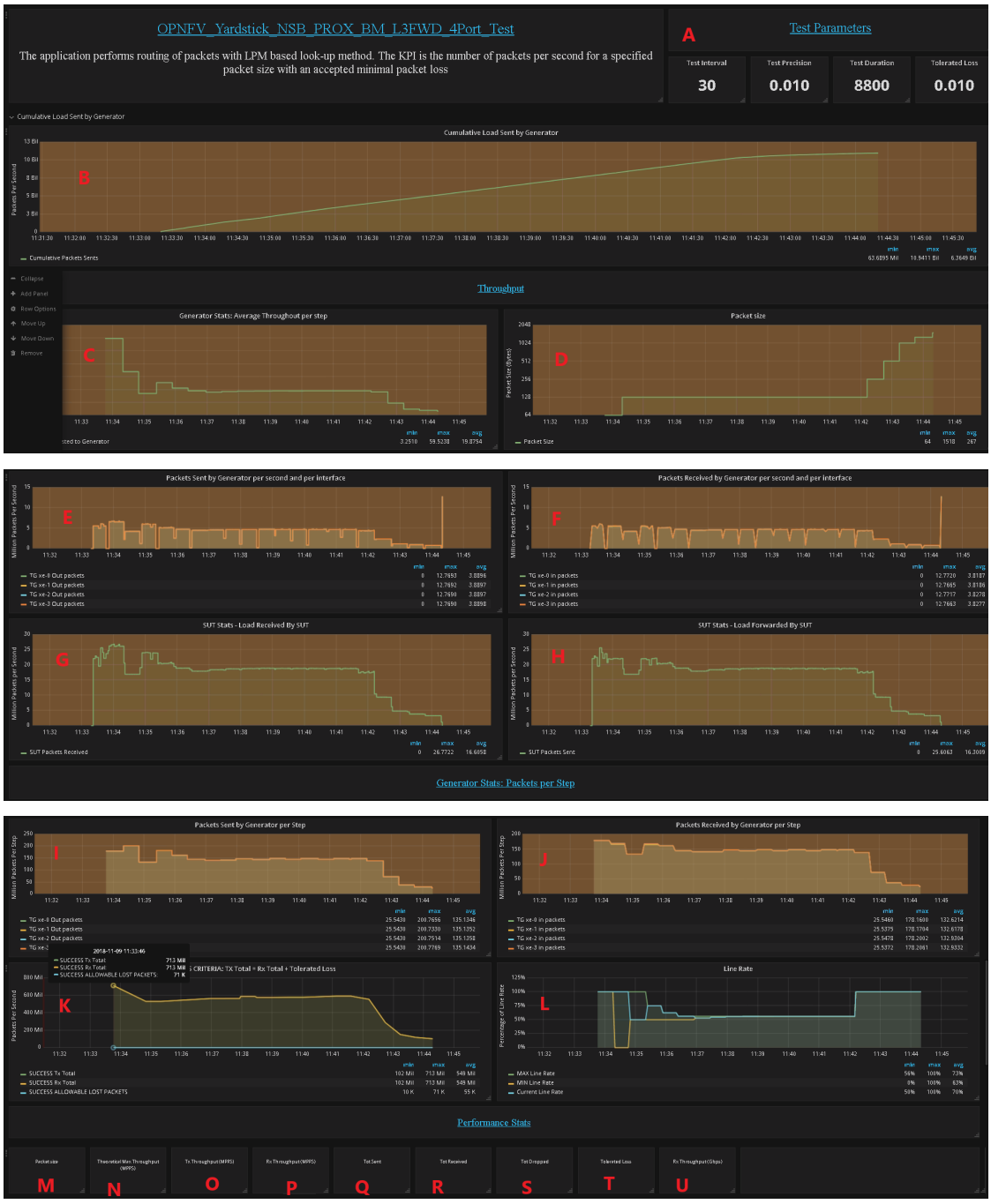
```
FTP_PROXY="http://<your_proxy>:<port>/"
HTTPS_PROXY="http://<your_proxy>:<port>/"
HTTP_PROXY="http://<your_proxy>:<port>/"
NO_PROXY="localhost,127.0.0.1,10.237.222.55,10.237.223.80,10.237.222.134,.ir.intel.com
↪"
ftp_proxy="http://<your_proxy>:<port>/"
http_proxy="http://<your_proxy>:<port>/"
https_proxy="http://<your_proxy>:<port>/"
no_proxy="localhost,127.0.0.1,10.237.222.55,10.237.223.80,10.237.222.134,.ir.intel.com
↪"
```

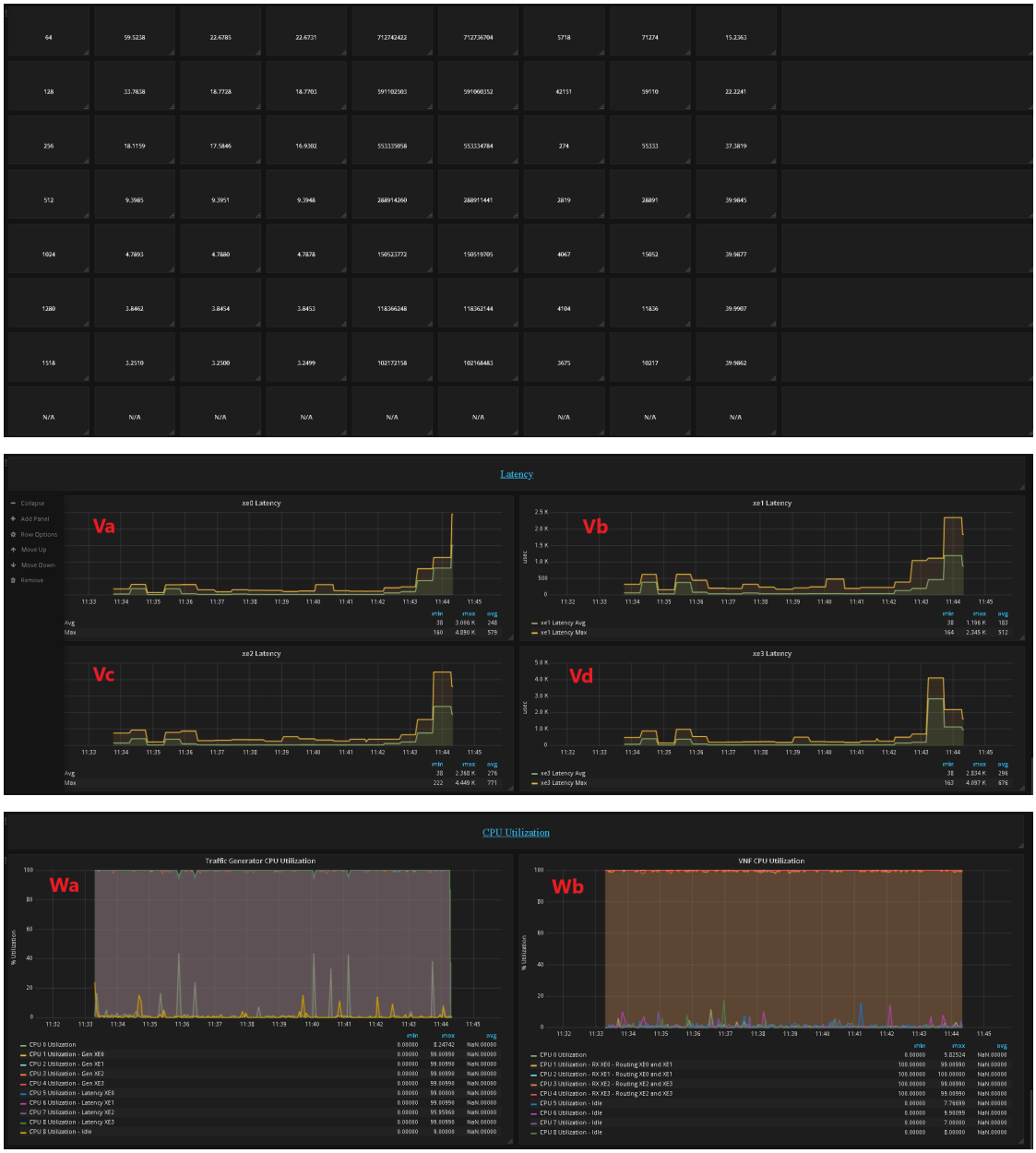
Where

1) 10.237.222.55 = IP Address of deployment node
2) 10.237.223.80 = IP Address of Controller node

3) 10.237.222.134 = IP Address of Compute Node

3.14.7 How to Understand the Grafana output?





A. Test Parameters - Test interval, Duration, Tolerated Loss and Test Precision

B. No. of packets send and received during test

C. Generator Stats - Average Throughput per step (Step Duration is specified by "Duration" field in A above)

D. Packet size

E. No. of packets sent by the generator per second per interface in millions of packets per second.

F. No. of packets recieved by the generator per second per interface in millions of packets per second.

G. No. of packets received by the SUT from the generator in millions of packets per second.

H. No. of packets sent by the the SUT to the generator in millions of packets per second.

I. No. of packets sent by the Generator to the SUT per step per interface in millions of packets per second.

J. No. of packets received by the Generator from the SUT per step per interface in millions of packets per second.

K. No. of packets sent and received by the generator and lost by the SUT that meet the success criteria

L. The change in the Percentage of Line Rate used over a test, The MAX and the MIN should converge to within the interval specified as the `test-precision`.

M. Packet size supported during test. If *N/A* appears in any field the result has not been decided.

N. The Theretical Maximum no. of packets per second that can be sent for this packet size.

O. No. of packets sent by the generator in MPPS

P. No. of packets received by the generator in MPPS

Q. No. of packets sent by SUT.

R. No. of packets received by the SUT

S. Total no. of dropped packets – Packets sent but not received back by the generator, these may be dropped by the SUT or the generator.

T. The tolerated no. of dropped packets.

U. Test throughput in Gbps

V. **Latencey per Port**

- Va - Port XE0
- Vb - Port XE1
- Vc - Port XE0
- Vd - Port XE0

W. **CPU Utilization**

- Wa - CPU Utilization of the Generator
- Wb - CPU Utilization of the SUT

INDEX