VSPERF User Guide

Release colorado.1.0 (1ea5c52)

OPNFV

August 23, 2016

1 vSwitchPerf test suites userguide **1**
1.1 General 1
1.2 VSPERF Installation 1
1.3 Traffic Generator Setup 1
1.4 Cloning and building src dependencies 1
1.5 Configure the `./conf/10_custom.conf` file 2
1.6 Using a custom settings file 2
1.7 vloop_vnf 2
1.8 l2fwd Kernel Module 3
1.9 Executing tests 3
1.10 Executing Vanilla OVS tests 3
1.11 Executing PVP and PVVP tests 4
1.12 Executing PVP tests using Vanilla OVS 4
1.13 Using vfio_pci with DPDK 5
1.14 Using SRIOV support 5
1.15 Using QEMU with PCI passthrough support 6
1.16 Selection of loopback application for PVP and PVVP tests 6
1.17 Multi-Queue Configuration 6
1.18 Executing Packet Forwarding tests 7
1.19 VSPERF modes of operation 8
1.20 Code change verification by pylint 8
1.21 GOTCHAs: 9
1.22 More information 9

2 Integration tests **11**
2.1 Executing Integration Tests 11
2.2 Test Steps 11
2.3 Test Macros 11
2.4 HelloWorld and other basic Testcases 12
2.5 Executing Tunnel encapsulation tests 17
2.6 Executing VXLAN decapsulation tests 19
2.7 Executing GRE decapsulation tests 19
2.8 Executing GENEVE decapsulation tests 20
2.9 Executing Native/Vanilla OVS VXLAN decapsulation tests 21
2.10 Executing Native/Vanilla OVS GRE decapsulation tests 21
2.11 Executing Native/Vanilla OVS GENEVE decapsulation tests 22
2.12 Executing Tunnel encapsulation+decapsulation tests 23

3 Execution of vswitchperf testcases by Yardstick **25**
3.1 General 25
3.2 Yardstick Installation 25

3.3 Vswitchperf VM image preparation 25
3.4 Testcase customization 26
3.5 Testcase execution 28

CHAPTER

ONE

VSWITCHPERF TEST SUITES USERGUIDE

1.1 General

VSPERF requires a traffic generators to run tests, automated traffic gen support in VSPERF includes:

- IXIA traffic generator (IxNetwork hardware) and a machine that runs the IXIA client software.
- Spirent traffic generator (TestCenter hardware chassis or TestCenter virtual in a VM) and a VM to run the Spirent Virtual Deployment Service image, formerly known as "Spirent LabServer".
- Xena Network traffic generator (Xena hardware chassis) that houses the Xena Traffic generator modules.
- Moongen software traffic generator. Requires a separate machine running moongen to execute packet generation.

If you want to use another traffic generator, please select the Dummy generator option as shown in Traffic generator instructions

1.2 VSPERF Installation

To see the supported Operating Systems, vSwitches and system requirements, please follow the installation instructions to install.

1.3 Traffic Generator Setup

Follow the Traffic generator instructions to install and configure a suitable traffic generator.

1.4 Cloning and building src dependencies

In order to run VSPERF, you will need to download DPDK and OVS. You can do this manually and build them in a preferred location, OR you could use vswitchperf/src. The vswitchperf/src directory contains makefiles that will allow you to clone and build the libraries that VSPERF depends on, such as DPDK and OVS. To clone and build simply:

```
$ cd src
$ make
```

VSPERF can be used with stock OVS (without DPDK support). When build is finished, the libraries are stored in src_vanilla directory.

The 'make' builds all options in src:

- Vanilla OVS
- OVS with vhost_user as the guest access method (with DPDK support)

The vhost_user build will reside in src/ovs/ The Vanilla OVS build will reside in vswitchperf/src_vanilla

To delete a src subdirectory and its contents to allow you to re-clone simply use:

```
$ make clobber
```

1.5 Configure the `./conf/10_custom.conf` file

The `10_custom.conf` file is the configuration file that overrides default configurations in all the other configuration files in `./conf` The supplied `10_custom.conf` file **MUST** be modified, as it contains configuration items for which there are no reasonable default values.

The configuration items that can be added is not limited to the initial contents. Any configuration item mentioned in any .conf file in `./conf` directory can be added and that item will be overridden by the custom configuration value.

1.6 Using a custom settings file

If your `10_custom.conf` doesn't reside in the `./conf` directory of if you want to use an alternative configuration file, the file can be passed to `vsperf` via the `--conf-file` argument.

```
$ ./vsperf --conf-file <path_to_custom_conf> ...
```

Note that configuration passed in via the environment (`--load-env`) or via another command line argument will override both the default and your custom configuration files. This "priority hierarchy" can be described like so (1 = max priority):

1. Command line arguments
2. Environment variables
3. Configuration file(s)

1.7 vloop_vnf

vsperf uses a VM called vloop_vnf for looping traffic in the PVP and PVVP deployment scenarios. The image can be downloaded from http://artifacts.opnfv.org/.

```
$ wget http://artifacts.opnfv.org/vswitchperf/vloop-vnf-ubuntu-14.04_20151216.qcow2
```

Newer vloop_vnf images are available. Please reference the installation instructions for information on these images installation instructions

vloop_vnf forwards traffic through a VM using one of: * DPDK testpmd * Linux Bridge * l2fwd kernel Module.

Alternatively you can use your own QEMU image.

1.8 l2fwd Kernel Module

A Kernel Module that provides OSI Layer 2 Ipv4 termination or forwarding with support for Destination Network Address Translation (DNAT) for both the MAC and IP addresses. l2fwd can be found in <vswitchperf_dir>/src/l2fwd

1.9 Executing tests

Before running any tests make sure you have root permissions by adding the following line to /etc/sudoers:

```
username ALL=(ALL)       NOPASSWD: ALL
```

username in the example above should be replaced with a real username.

To list the available tests:

```
$ ./vsperf --list
```

To run a single test:

```
$ ./vsperf $TESTNAME
```

Where $TESTNAME is the name of the vsperf test you would like to run.

To run a group of tests, for example all tests with a name containing 'RFC2544':

```
$ ./vsperf --conf-file=<path_to_custom_conf>/10_custom.conf --tests="RFC2544"
```

To run all tests:

```
$ ./vsperf --conf-file=<path_to_custom_conf>/10_custom.conf
```

Some tests allow for configurable parameters, including test duration (in seconds) as well as packet sizes (in bytes).

```
$ ./vsperf --conf-file user_settings.py
    --tests RFC2544Tput
    --test-params "duration=10;pkt_sizes=128"
```

For all available options, check out the help dialog:

```
$ ./vsperf --help
```

1.10 Executing Vanilla OVS tests

1. If needed, recompile src for all OVS variants

```
$ cd src
$ make distclean
$ make
```

2. Update your "10_custom.conf" file to use the appropriate variables for Vanilla OVS:

```
VSWITCH = 'OvsVanilla'
```

Where $PORT1 and $PORT2 are the Linux interfaces you'd like to bind to the vswitch.

3. Run test:

```
$ ./vsperf --conf-file=<path_to_custom_conf>
```

Please note if you don't want to configure Vanilla OVS through the configuration file, you can pass it as a CLI argument; BUT you must set the ports.

```
$ ./vsperf --vswitch OvsVanilla
```

1.11 Executing PVP and PVVP tests

To run tests using vhost-user as guest access method:

1. Set VHOST_METHOD and VNF of your settings file to:

```
VSWITCH = 'OvsDpdkVhost'
VNF = 'QemuDpdkVhost'
```

2. If needed, recompile src for all OVS variants

```
$ cd src
$ make distclean
$ make
```

3. Run test:

```
$ ./vsperf --conf-file=<path_to_custom_conf>/10_custom.conf
```

1.12 Executing PVP tests using Vanilla OVS

To run tests using Vanilla OVS:

1. Set the following variables:

```
VSWITCH = 'OvsVanilla'
VNF = 'QemuVirtioNet'

VANILLA_TGEN_PORT1_IP = n.n.n.n
VANILLA_TGEN_PORT1_MAC = nn:nn:nn:nn:nn:nn

VANILLA_TGEN_PORT2_IP = n.n.n.n
VANILLA_TGEN_PORT2_MAC = nn:nn:nn:nn:nn:nn

VANILLA_BRIDGE_IP = n.n.n.n

or use --test-param

$ ./vsperf --conf-file=<path_to_custom_conf>/10_custom.conf
           --test-params "vanilla_tgen_tx_ip=n.n.n.n;
                         vanilla_tgen_tx_mac=nn:nn:nn:nn:nn:nn"
```

2. If needed, recompile src for all OVS variants

```
$ cd src
$ make distclean
$ make
```

3. Run test:

```
$ ./vsperf --conf-file<path_to_custom_conf>/10_custom.conf
```

1.13 Using vfio_pci with DPDK

To use vfio with DPDK instead of igb_uio edit 'conf/02_vswitch.conf' with the following parameters:

```
DPDK_MODULES = [
 ('vfio-pci'),
]
SYS_MODULES = ['cuse']
```

NOTE: Please ensure that Intel VT-d is enabled in BIOS.

NOTE: Please ensure your boot/grub parameters include the following:

```
iommu=pt intel_iommu=on
```

To check that IOMMU is enabled on your platform:

```
$ dmesg | grep IOMMU
[    0.000000] Intel-IOMMU: enabled
[    0.139882] dmar: IOMMU 0: reg_base_addr fbffe000 ver 1:0 cap d2078c106f0466 ecap f020de
[    0.139888] dmar: IOMMU 1: reg_base_addr ebffc000 ver 1:0 cap d2078c106f0466 ecap f020de
[    0.139893] IOAPIC id 2 under DRHD base  0xfbffe000 IOMMU 0
[    0.139894] IOAPIC id 0 under DRHD base  0xebffc000 IOMMU 1
[    0.139895] IOAPIC id 1 under DRHD base  0xebffc000 IOMMU 1
[    3.335744] IOMMU: dmar0 using Queued invalidation
[    3.335746] IOMMU: dmar1 using Queued invalidation
....
```

1.14 Using SRIOV support

To use virtual functions of NIC with SRIOV support, use extended form of NIC PCI slot definition:

```
WHITELIST_NICS = ['0000:05:00.0|vf0', '0000:05:00.1|vf3']
```

Where 'vf' is an indication of virtual function usage and following number defines a VF to be used. In case that VF usage is detected, then vswitchperf will enable SRIOV support for given card and it will detect PCI slot numbers of selected VFs.

So in example above, one VF will be configured for NIC '0000:05:00.0' and four VFs will be configured for NIC '0000:05:00.1'. Vswitchperf will detect PCI addresses of selected VFs and it will use them during test execution.

At the end of vswitchperf execution, SRIOV support will be disabled.

SRIOV support is generic and it can be used in different testing scenarios. For example:

- vSwitch tests with DPDK or without DPDK support to verify impact of VF usage on vSwitch performance
- tests without vSwitch, where traffic is forwarded directly between VF interfaces by packet forwarder (e.g. testpmd application)
- tests without vSwitch, where VM accesses VF interfaces directly by *PCI-passthrough* to measure raw VM throughput performance.

1.15 Using QEMU with PCI passthrough support

Raw virtual machine throughput performance can be measured by execution of PVP test with direct access to NICs by PCI passthrough. To execute VM with direct access to PCI devices, enable *vfio-pci*. In order to use virtual functions, *SRIOV-support* must be enabled.

Execution of test with PCI passthrough with vswitch disabled:

```
$ ./vsperf --conf-file=<path_to_custom_conf>/10_custom.conf
           --vswitch none --vnf QemuPciPassthrough pvp_tput
```

Any of supported *guest-loopback-application* can be used inside VM with PCI passthrough support.

Note: Qemu with PCI passthrough support can be used only with PVP test deployment.

1.16 Selection of loopback application for PVP and PVVP tests

To select loopback application, which will perform traffic forwarding inside VM, following configuration parameter should be configured:

```
GUEST_LOOPBACK = ['testpmd', 'testpmd']
```

or use –test-param

```
$ ./vsperf --conf-file=<path_to_custom_conf>/10_custom.conf
      --test-params "guest_loopback=testpmd"
```

Supported loopback applications are:

```
'testpmd'       - testpmd from dpdk will be built and used
'l2fwd'         - l2fwd module provided by Huawei will be built and used
'linux_bridge'  - linux bridge will be configured
'buildin'       - nothing will be configured by vsperf; VM image must
                  ensure traffic forwarding between its interfaces
```

Guest loopback application must be configured, otherwise traffic will not be forwarded by VM and testcases with PVP and PVVP deployments will fail. Guest loopback application is set to 'testpmd' by default.

1.17 Multi-Queue Configuration

VSPerf currently supports multi-queue with the following limitations:

1. Execution of pvp/pvvp tests require testpmd as the loopback if multi-queue is enabled at the guest.
2. Requires QemuDpdkVhostUser as the vnf.
3. Requires switch to be set to OvsDpdkVhost.
4. Requires QEMU 2.5 or greater and any OVS version higher than 2.5. The default upstream package versions installed by VSPerf satisfy this requirement.
5. If using OVS versions 2.5.0 or less enable old style multi-queue as shown in the "02_vswitch.conf" file.

```
        OVS_OLD_STYLE_MQ = True
```

To enable multi-queue modify the "02_vswitch.conf" file to enable multi-queue on the switch.

```
VSWITCH_MULTI_QUEUES = 2
```

NOTE: you should consider using the switch affinity to set a pmd cpu mask that can optimize your performance. Consider the numa of the NIC in use if this applies by checking /sys/class/net/<eth_name>/device/numa_node and setting an appropriate mask to create PMD threads on the same numa node.

When multi-queue is enabled, each dpdk or dpdkvhostuser port that is created on the switch will set the option for multiple queues. If old style multi queue has been enabled a global option for multi queue will be used instead of the port by port option.

To enable multi-queue on the guest modify the "04_vnf.conf" file.

```
GUEST_NIC_QUEUES = 2
```

Enabling multi-queue at the guest will add multiple queues to each NIC port when qemu launches the guest.

Testpmd should be configured to take advantage of multi-queue on the guest. This can be done by modifying the "04_vnf.conf" file.

```
GUEST_TESTPMD_CPU_MASK = '-l 0,1,2,3,4'

GUEST_TESTPMD_NB_CORES = 4
GUEST_TESTPMD_TXQ = 2
GUEST_TESTPMD_RXQ = 2
```

NOTE: The guest SMP cores must be configured to allow for testpmd to use the optimal number of cores to take advantage of the multiple guest queues.

NOTE: For optimal performance guest SMPs should be on the same numa as the NIC in use if possible/applicable. Testpmd should be assigned at least (nb_cores +1) total cores with the cpu mask.

1.18 Executing Packet Forwarding tests

To select application, which will perform packet forwarding, following configuration parameter should be configured:

```
VSWITCH = 'none'
PKTFWD = 'TestPMD'

or use --vswitch and --fwdapp

$ ./vsperf --conf-file user_settings.py
         --vswitch none
         --fwdapp TestPMD
```

Supported Packet Forwarding applications are:

```
'testpmd'       - testpmd from dpdk
```

1. Update your "10_custom.conf" file to use the appropriate variables for selected Packet Forwarder:

```
# testpmd configuration
TESTPMD_ARGS = []
# packet forwarding mode supported by testpmd; Please see DPDK documentation
# for comprehensive list of modes supported by your version.
# e.g. io|mac|mac_retry|macswap|flowgen|rxonly|txonly|csum|icmpecho|...
# Note: Option "mac_retry" has been changed to "mac retry" since DPDK v16.07
TESTPMD_FWD_MODE = 'csum'
# checksum calculation layer: ip|udp|tcp|sctp|outer-ip
```

```
    TESTPMD_CSUM_LAYER = 'ip'
    # checksum calculation place: hw (hardware) | sw (software)
    TESTPMD_CSUM_CALC = 'sw'
    # recognize tunnel headers: on|off
    TESTPMD_CSUM_PARSE_TUNNEL = 'off'
```

2. Run test:

```
    $ ./vsperf --conf-file <path_to_settings_py>
```

1.19 VSPERF modes of operation

VSPERF can be run in different modes. By default it will configure vSwitch, traffic generator and VNF. However it can be used just for configuration and execution of traffic generator. Another option is execution of all components except traffic generator itself.

Mode of operation is driven by configuration parameter -m or –mode

```
-m MODE, --mode MODE  vsperf mode of operation;
    Values:
        "normal" - execute vSwitch, VNF and traffic generator
        "trafficgen" - execute only traffic generator
        "trafficgen-off" - execute vSwitch and VNF
        "trafficgen-pause" - execute vSwitch and VNF but wait before traffic transmission
```

In case, that VSPERF is executed in "trafficgen" mode, then configuration of traffic generator should be configured through –test-params option. Supported CLI options useful for traffic generator configuration are:

```
'traffic_type'  - One of the supported traffic types. E.g. rfc2544,
                  back2back or continuous
                  Default value is "rfc2544".
'bidirectional' - Specifies if generated traffic will be full-duplex (true)
                  or half-duplex (false)
                  Default value is "false".
'iload'         - Defines desired percentage of frame rate used during
                  continuous stream tests.
                  Default value is 100.
'multistream'   - Defines number of flows simulated by traffic generator.
                  Value 0 disables MultiStream feature
                  Default value is 0.
'stream_type'   - Stream Type is an extension of the "MultiStream" feature.
                  If MultiStream is disabled, then Stream Type will be
                  ignored. Stream Type defines ISO OSI network layer used
                  for simulation of multiple streams.
                  Default value is "L4".
```

Example of execution of VSPERF in "trafficgen" mode:

```
$ ./vsperf -m trafficgen --trafficgen IxNet --conf-file vsperf.conf
    --test-params "traffic_type=continuous;bidirectional=True;iload=60"
```

1.20 Code change verification by pylint

Every developer participating in VSPERF project should run pylint before his python code is submitted for review. Project specific configuration for pylint is available at 'pylint.rc'.

Example of manual pylint invocation:

```
$ pylint --rcfile ./pylintrc ./vsperf
```

1.21 GOTCHAs:

1.21.1 OVS with DPDK and QEMU

If you encounter the following error: "before (last 100 chars): '-path=/dev/hugepages,share=on: unable to map backing store for hugepages: Cannot allocate memoryrnrn" with the PVP or PVVP deployment scenario, check the amount of hugepages on your system:

```
$ cat /proc/meminfo | grep HugePages
```

By default the vswitchd is launched with 1Gb of memory, to change this, modify –socket-mem parameter in conf/02_vswitch.conf to allocate an appropriate amount of memory:

```
VSWITCHD_DPDK_ARGS = ['-c', '0x4', '-n', '4', '--socket-mem 1024,0']
VSWITCHD_DPDK_CONFIG = {
    'dpdk-init' : 'true',
    'dpdk-lcore-mask' : '0x4',
    'dpdk-socket-mem' : '1024,0',
}
```

Note: Option VSWITCHD_DPDK_ARGS is used for vswitchd, which supports –dpdk parameter. In recent vswitchd versions, option VSWITCHD_DPDK_CONFIG will be used to configure vswitchd via ovs-vsctl calls.

1.22 More information

For more information and details refer to the vSwitchPerf user guide at: http://artifacts.opnfv.org/vswitchperf/docs/userguide/index.html

CHAPTER TWO

INTEGRATION TESTS

VSPERF includes a set of integration tests defined in conf/integration. These tests can be run by specifying –integration as a parameter to vsperf. Current tests in conf/integration include switch functionality and Overlay tests.

Tests in the conf/integration can be used to test scaling of different switch configurations by adding steps into the test case.

For the overlay tests VSPERF supports VXLAN, GRE and GENEVE tunneling protocols. Testing of these protocols is limited to unidirectional traffic and P2P (Physical to Physical scenarios).

NOTE: The configuration for overlay tests provided in this guide is for unidirectional traffic only.

2.1 Executing Integration Tests

To execute integration tests VSPERF is run with the integration parameter. To view the current test list simply execute the following command:

```
./vsperf --integration --list
```

The standard tests included are defined inside the `conf/integration/01_testcases.conf` file.

2.2 Test Steps

Execution of integration tests are done on a step by step work flow starting with step 0 as defined inside the test case. Each step of the test increments the step number by one which is indicated in the log.

```
(testcases.integration) - Step 1 - 'vswitch add_switch ['int_br1']' ... OK
```

Each step in the test case is validated. If a step does not pass validation the test will fail and terminate. The test will continue until a failure is detected or all steps pass. A csv report file is generated after a test completes with an OK or FAIL result.

2.3 Test Macros

Test profiles can include macros as part of the test step. Each step in the profile may return a value such as a port name. Recall macros use #STEP to indicate the recalled value inside the return structure. If the method the test step calls returns a value it can be later recalled, for example:

```
{
    "Name": "vswitch_add_del_vport",
    "Deployment": "clean",
    "Description": "vSwitch - add and delete virtual port",
    "TestSteps": [
            ['vswitch', 'add_switch', 'int_br0'],               # STEP 0
            ['vswitch', 'add_vport', 'int_br0'],                # STEP 1
            ['vswitch', 'del_port', 'int_br0', '#STEP[1][0]'],  # STEP 2
            ['vswitch', 'del_switch', 'int_br0'],               # STEP 3
         ]
}
```

This test profile uses the vswitch add_vport method which returns a string value of the port added. This is later called by the del_port method using the name from step 1.

Also commonly used steps can be created as a separate profile.

```
STEP_VSWITCH_PVP_INIT = [
    ['vswitch', 'add_switch', 'int_br0'],           # STEP 0
    ['vswitch', 'add_phy_port', 'int_br0'],         # STEP 1
    ['vswitch', 'add_phy_port', 'int_br0'],         # STEP 2
    ['vswitch', 'add_vport', 'int_br0'],            # STEP 3
    ['vswitch', 'add_vport', 'int_br0'],            # STEP 4
]
```

This profile can then be used inside other testcases

```
{
    "Name": "vswitch_pvp",
    "Deployment": "clean",
    "Description": "vSwitch - configure switch and one vnf",
    "TestSteps": STEP_VSWITCH_PVP_INIT +
                 [
                    ['vnf', 'start'],
                    ['vnf', 'stop'],
                 ] +
                 STEP_VSWITCH_PVP_FINIT
}
```

2.4 HelloWorld and other basic Testcases

The following examples are for demonstration purposes. You can run them by copying and pasting into the conf/integration/01_testcases.conf file. A command-line instruction is shown at the end of each example.

2.4.1 HelloWorld

The first example is a HelloWorld testcase. It simply creates a bridge with 2 physical ports, then sets up a flow to drop incoming packets from the port that was instantiated at the STEP #1. There's no interaction with the traffic generator. Then the flow, the 2 ports and the bridge are deleted. 'add_phy_port' method creates a 'dpdk' type interface that will manage the physical port. The string value returned is the port name that will be referred by 'del_port' later on.

```
{
    "Name": "HelloWorld",
    "Description": "My first testcase",
    "Deployment": "clean",
    "TestSteps": [
        ['vswitch', 'add_switch', 'int_br0'],   # STEP 0
        ['vswitch', 'add_phy_port', 'int_br0'], # STEP 1
        ['vswitch', 'add_phy_port', 'int_br0'], # STEP 2
        ['vswitch', 'add_flow', 'int_br0', {'in_port': '#STEP[1][1]', \
            'actions': ['drop'], 'idle_timeout': '0'}],
        ['vswitch', 'del_flow', 'int_br0'],
        ['vswitch', 'del_port', 'int_br0', '#STEP[1][0]'],
        ['vswitch', 'del_port', 'int_br0', '#STEP[2][0]'],
        ['vswitch', 'del_switch', 'int_br0'],
    ]

}
```

To run HelloWorld test:

```
     ./vsperf --conf-file user_settings.py --integration HelloWorld
```

2.4.2 Specify a Flow by the IP address

The next example shows how to explicitly set up a flow by specifying a destination IP address. All packets received from the port created at STEP #1 that have a destination IP address = 90.90.90.90 will be forwarded to the port created at the STEP #2.

```
{
    "Name": "p2p_rule_l3da",
    "Description": "Phy2Phy with rule on L3 Dest Addr",
    "Deployment": "clean",
    "biDirectional": "False",
    "TestSteps": [
        ['vswitch', 'add_switch', 'int_br0'],   # STEP 0
        ['vswitch', 'add_phy_port', 'int_br0'], # STEP 1
        ['vswitch', 'add_phy_port', 'int_br0'], # STEP 2
        ['vswitch', 'add_flow', 'int_br0', {'in_port': '#STEP[1][1]', \
            'dl_type': '0x0800', 'nw_dst': '90.90.90.90', \
            'actions': ['output:#STEP[2][1]'], 'idle_timeout': '0'}],
        ['trafficgen', 'send_traffic', {'traffic_type' : 'continuous'}],
        ['vswitch', 'dump_flows', 'int_br0'],   # STEP 5
        ['vswitch', 'del_flow', 'int_br0'],     # STEP 7 == del-flows
        ['vswitch', 'del_port', 'int_br0', '#STEP[1][0]'],
        ['vswitch', 'del_port', 'int_br0', '#STEP[2][0]'],
        ['vswitch', 'del_switch', 'int_br0'],
    ]
},
```

To run the test:

```
     ./vsperf --conf-file user_settings.py --integration p2p_rule_l3da
```

2.4.3 Multistream feature

The next testcase uses the multistream feature. The traffic generator will send packets with different UDP ports. That is accomplished by using “Stream Type” and “MultiStream” keywords. 4 different flows are set to forward all incoming packets.

```
{
    "Name": "multistream_l4",
    "Description": "Multistream on UDP ports",
    "Deployment": "clean",
    "Stream Type": "L4",
    "MultiStream": 4,
    "TestSteps": [
        ['vswitch', 'add_switch', 'int_br0'],   # STEP 0
        ['vswitch', 'add_phy_port', 'int_br0'], # STEP 1
        ['vswitch', 'add_phy_port', 'int_br0'], # STEP 2
        # Setup Flows
        ['vswitch', 'add_flow', 'int_br0', {'in_port': '#STEP[1][1]', \
            'dl_type': '0x0800', 'nw_proto': '17', 'udp_dst': '0', \
            'actions': ['output:#STEP[2][1]'], 'idle_timeout': '0'}],
        ['vswitch', 'add_flow', 'int_br0', {'in_port': '#STEP[1][1]', \
            'dl_type': '0x0800', 'nw_proto': '17', 'udp_dst': '1', \
            'actions': ['output:#STEP[2][1]'], 'idle_timeout': '0'}],
        ['vswitch', 'add_flow', 'int_br0', {'in_port': '#STEP[1][1]', \
            'dl_type': '0x0800', 'nw_proto': '17', 'udp_dst': '2', \
            'actions': ['output:#STEP[2][1]'], 'idle_timeout': '0'}],
        ['vswitch', 'add_flow', 'int_br0', {'in_port': '#STEP[1][1]', \
            'dl_type': '0x0800', 'nw_proto': '17', 'udp_dst': '3', \
            'actions': ['output:#STEP[2][1]'], 'idle_timeout': '0'}],
        # Send mono-dir traffic
        ['trafficgen', 'send_traffic', {'traffic_type' : 'continuous', \
            'bidir' : 'False'}],
        # Clean up
        ['vswitch', 'del_flow', 'int_br0'],
        ['vswitch', 'del_port', 'int_br0', '#STEP[1][0]'],
        ['vswitch', 'del_port', 'int_br0', '#STEP[2][0]'],
        ['vswitch', 'del_switch', 'int_br0'],
     ]
},
```

To run the test:

```
     ./vsperf --conf-file user_settings.py --integration multistream_l4
```

2.4.4 PVP with a VM Replacement

This example launches a 1st VM in a PVP topology, then the VM is replaced by another VM. When VNF setup parameter in ./conf/04_vnf.conf is “QemuDpdkVhostUser” ‘add_vport’ method creates a ‘dpdkvhostuser’ type port to connect a VM.

```
{
    "Name": "ex_replace_vm",
    "Description": "PVP with VM replacement",
    "Deployment": "clean",
    "TestSteps": [
        ['vswitch', 'add_switch', 'int_br0'],       # STEP 0
        ['vswitch', 'add_phy_port', 'int_br0'],     # STEP 1
        ['vswitch', 'add_phy_port', 'int_br0'],     # STEP 2
        ['vswitch', 'add_vport', 'int_br0'],        # STEP 3    vm1
        ['vswitch', 'add_vport', 'int_br0'],        # STEP 4

        # Setup Flows
        ['vswitch', 'add_flow', 'int_br0', {'in_port': '#STEP[1][1]', \
            'actions': ['output:#STEP[3][1]'], 'idle_timeout': '0'}],
        ['vswitch', 'add_flow', 'int_br0', {'in_port': '#STEP[4][1]', \
            'actions': ['output:#STEP[2][1]'], 'idle_timeout': '0'}],
        ['vswitch', 'add_flow', 'int_br0', {'in_port': '#STEP[2][1]', \
            'actions': ['output:#STEP[4][1]'], 'idle_timeout': '0'}],
        ['vswitch', 'add_flow', 'int_br0', {'in_port': '#STEP[3][1]', \
            'actions': ['output:#STEP[1][1]'], 'idle_timeout': '0'}],

        # Start VM 1
        ['vnf1', 'start'],
        # Now we want to replace VM 1 with another VM
        ['vnf1', 'stop'],

        ['vswitch', 'add_vport', 'int_br0'],         # STEP 11     vm2
        ['vswitch', 'add_vport', 'int_br0'],         # STEP 12
        ['vswitch', 'del_flow', 'int_br0'],
        ['vswitch', 'add_flow', 'int_br0', {'in_port': '#STEP[1][1]', \
            'actions': ['output:#STEP[11][1]'], 'idle_timeout': '0'}],
        ['vswitch', 'add_flow', 'int_br0', {'in_port': '#STEP[12][1]', \
            'actions': ['output:#STEP[2][1]'], 'idle_timeout': '0'}],

        # Start VM 2
        ['vnf2', 'start'],
        ['vnf2', 'stop'],
        ['vswitch', 'dump_flows', 'int_br0'],

        # Clean up
        ['vswitch', 'del_flow', 'int_br0'],
        ['vswitch', 'del_port', 'int_br0', '#STEP[1][0]'],
        ['vswitch', 'del_port', 'int_br0', '#STEP[2][0]'],
        ['vswitch', 'del_port', 'int_br0', '#STEP[3][0]'],     # vm1
        ['vswitch', 'del_port', 'int_br0', '#STEP[4][0]'],
        ['vswitch', 'del_port', 'int_br0', '#STEP[11][0]'],    # vm2
        ['vswitch', 'del_port', 'int_br0', '#STEP[12][0]'],
        ['vswitch', 'del_switch', 'int_br0'],
    ]
},
```

To run the test:

```
    ./vsperf --conf-file user_settings.py --integration ex_replace_vm
```

2.4.5 VM with a Linux bridge

In this example a command-line parameter allows to set up a Linux bridge into the guest VM. That's one of the available ways to specify the guest application. Packets matching the flow will be forwarded to the VM.

```
{
    "Name": "ex_pvp_rule_l3da",
    "Description": "PVP with flow on L3 Dest Addr",
    "Deployment": "clean",
    "TestSteps": [
        ['vswitch', 'add_switch', 'int_br0'],       # STEP 0
        ['vswitch', 'add_phy_port', 'int_br0'],     # STEP 1
        ['vswitch', 'add_phy_port', 'int_br0'],     # STEP 2
        ['vswitch', 'add_vport', 'int_br0'],        # STEP 3    vm1
        ['vswitch', 'add_vport', 'int_br0'],        # STEP 4
        # Setup Flows
        ['vswitch', 'add_flow', 'int_br0', {'in_port': '#STEP[1][1]', \
            'dl_type': '0x0800', 'nw_dst': '90.90.90.90', \
            'actions': ['output:#STEP[3][1]'], 'idle_timeout': '0'}],
        # Each pkt from the VM is forwarded to the 2nd dpdk port
        ['vswitch', 'add_flow', 'int_br0', {'in_port': '#STEP[4][1]', \
            'actions': ['output:#STEP[2][1]'], 'idle_timeout': '0'}],
        # Start VMs
        ['vnf1', 'start'],
        ['trafficgen', 'send_traffic', {'traffic_type' : 'continuous', \
            'bidir' : 'False'}],
        ['vnf1', 'stop'],
        # Clean up
        ['vswitch', 'dump_flows', 'int_br0'],       # STEP 10
        ['vswitch', 'del_flow', 'int_br0'],         # STEP 11
        ['vswitch', 'del_port', 'int_br0', '#STEP[1][0]'],
        ['vswitch', 'del_port', 'int_br0', '#STEP[2][0]'],
        ['vswitch', 'del_port', 'int_br0', '#STEP[3][0]'],  # vm1 ports
        ['vswitch', 'del_port', 'int_br0', '#STEP[4][0]'],
        ['vswitch', 'del_switch', 'int_br0'],
    ]
},
```

To run the test:

```
    ./vsperf --conf-file user_settings.py --test-params
            "guest_loopback=linux_bridge" --integration ex_pvp_rule_l3da
```

2.4.6 Forward packets based on UDP port

This examples launches 2 VMs connected in parallel. Incoming packets will be forwarded to one specific VM depending on the destination UDP port.

```
{
    "Name": "ex_2pvp_rule_l4dp",
    "Description": "2 PVP with flows on L4 Dest Port",
    "Deployment": "clean",
    "Stream Type": "L4",     # loop UDP ports
    "MultiStream": 2,
    "TestSteps": [
        ['vswitch', 'add_switch', 'int_br0'],       # STEP 0
        ['vswitch', 'add_phy_port', 'int_br0'],     # STEP 1
        ['vswitch', 'add_phy_port', 'int_br0'],     # STEP 2
        ['vswitch', 'add_vport', 'int_br0'],        # STEP 3    vm1
        ['vswitch', 'add_vport', 'int_br0'],        # STEP 4
        ['vswitch', 'add_vport', 'int_br0'],        # STEP 5    vm2
        ['vswitch', 'add_vport', 'int_br0'],        # STEP 6
        # Setup Flows to reply ICMPv6 and similar packets, so to
        # avoid flooding internal port with their re-transmissions
        ['vswitch', 'add_flow', 'int_br0', \
            {'priority': '1', 'dl_src': '00:00:00:00:00:01', \
            'actions': ['output:#STEP[3][1]'], 'idle_timeout': '0'}],
        ['vswitch', 'add_flow', 'int_br0', \
            {'priority': '1', 'dl_src': '00:00:00:00:00:02', \
            'actions': ['output:#STEP[4][1]'], 'idle_timeout': '0'}],
        ['vswitch', 'add_flow', 'int_br0', \
            {'priority': '1', 'dl_src': '00:00:00:00:00:03', \
            'actions': ['output:#STEP[5][1]'], 'idle_timeout': '0'}],
        ['vswitch', 'add_flow', 'int_br0', \
            {'priority': '1', 'dl_src': '00:00:00:00:00:04', \
            'actions': ['output:#STEP[6][1]'], 'idle_timeout': '0'}],
        # Forward UDP packets depending on dest port
        ['vswitch', 'add_flow', 'int_br0', {'in_port': '#STEP[1][1]', \
            'dl_type': '0x0800', 'nw_proto': '17', 'udp_dst': '0', \
            'actions': ['output:#STEP[3][1]'], 'idle_timeout': '0'}],
        ['vswitch', 'add_flow', 'int_br0', {'in_port': '#STEP[1][1]', \
            'dl_type': '0x0800', 'nw_proto': '17', 'udp_dst': '1', \
            'actions': ['output:#STEP[5][1]'], 'idle_timeout': '0'}],
        # Send VM output to phy port #2
        ['vswitch', 'add_flow', 'int_br0', {'in_port': '#STEP[4][1]', \
            'actions': ['output:#STEP[2][1]'], 'idle_timeout': '0'}],
        ['vswitch', 'add_flow', 'int_br0', {'in_port': '#STEP[6][1]', \
            'actions': ['output:#STEP[2][1]'], 'idle_timeout': '0'}],
        # Start VMs
        ['vnf1', 'start'],                          # STEP 16
        ['vnf2', 'start'],                          # STEP 17
        ['trafficgen', 'send_traffic', {'traffic_type' : 'continuous', \
            'bidir' : 'False'}],
        ['vnf1', 'stop'],
        ['vnf2', 'stop'],
        ['vswitch', 'dump_flows', 'int_br0'],
        # Clean up
        ['vswitch', 'del_flow', 'int_br0'],
        ['vswitch', 'del_port', 'int_br0', '#STEP[1][0]'],
        ['vswitch', 'del_port', 'int_br0', '#STEP[2][0]'],
        ['vswitch', 'del_port', 'int_br0', '#STEP[3][0]'],  # vm1 ports
        ['vswitch', 'del_port', 'int_br0', '#STEP[4][0]'],
        ['vswitch', 'del_port', 'int_br0', '#STEP[5][0]'],  # vm2 ports
        ['vswitch', 'del_port', 'int_br0', '#STEP[6][0]'],
        ['vswitch', 'del_switch', 'int_br0'],
    ]
},
```

To run the test:

```
    ./vsperf --conf-file user_settings.py --integration ex_2pvp_rule_l4dp
```

2.5 Executing Tunnel encapsulation tests

The VXLAN OVS DPDK encapsulation tests requires IPs, MAC addresses, bridge names and WHITELIST_NICS for DPDK.

NOTE: Only Ixia traffic generators currently support the execution of the tunnel encapsulation tests. Support for other traffic generators may come in a future release.

Default values are already provided. To customize for your environment, override the following variables in you user_settings.py file:

```
    # Variables defined in conf/integration/02_vswitch.conf
    # Tunnel endpoint for Overlay P2P deployment scenario
    # used for br0
    VTEP_IP1 = '192.168.0.1/24'

    # Used as remote_ip in adding OVS tunnel port and
    # to set ARP entry in OVS (e.g. tnl/arp/set br-ext 192.168.240.10 02:00:00:00:00:02
    VTEP_IP2 = '192.168.240.10'

    # Network to use when adding a route for inner frame data
    VTEP_IP2_SUBNET = '192.168.240.0/24'

    # Bridge names
    TUNNEL_INTEGRATION_BRIDGE = 'br0'
    TUNNEL_EXTERNAL_BRIDGE = 'br-ext'

    # IP of br-ext
    TUNNEL_EXTERNAL_BRIDGE_IP = '192.168.240.1/24'

    # vxlan|gre|geneve
    TUNNEL_TYPE = 'vxlan'

    # Variables defined conf/integration/03_traffic.conf
    # For OP2P deployment scenario
    TRAFFICGEN_PORT1_MAC = '02:00:00:00:00:01'
    TRAFFICGEN_PORT2_MAC = '02:00:00:00:00:02'
    TRAFFICGEN_PORT1_IP = '1.1.1.1'
    TRAFFICGEN_PORT2_IP = '192.168.240.10'
```

To run VXLAN encapsulation tests:

```
./vsperf --conf-file user_settings.py --integration
         --test-params 'tunnel_type=vxlan' overlay_p2p_tput
```

To run GRE encapsulation tests:

```
./vsperf --conf-file user_settings.py --integration
         --test-params 'tunnel_type=gre' overlay_p2p_tput
```

To run GENEVE encapsulation tests:

```
./vsperf --conf-file user_settings.py --integration
         --test-params 'tunnel_type=geneve' overlay_p2p_tput
```

To run OVS NATIVE tunnel tests (VXLAN/GRE/GENEVE):

1. Install the OVS kernel modules

```
cd src/ovs/ovs
sudo -E make modules_install
```

2. Set the following variables:

```
VSWITCH = 'OvsVanilla'
# Specify vport_* kernel module to test.
VSWITCH_VANILLA_KERNEL_MODULES = ['vport_vxlan',
                                  'vport_gre',
                                  'vport_geneve',
                                  os.path.join(OVS_DIR_VANILLA,
                                  'datapath/linux/openvswitch.ko')]
```

3. Run tests:

```
./vsperf --conf-file user_settings.py --integration
         --test-params 'tunnel_type=vxlan' overlay_p2p_tput
```

2.6 Executing VXLAN decapsulation tests

To run VXLAN decapsulation tests:

1. Set the variables used in "Executing Tunnel encapsulation tests"
2. Set dstmac of DUT_NIC2_MAC to the MAC adddress of the 2nd NIC of your DUT

```
DUT_NIC2_MAC = '<DUT NIC2 MAC>'
```

3. Run test:

```
./vsperf --conf-file user_settings.py --integration overlay_p2p_decap_cont
```

If you want to use different values for your VXLAN frame, you may set:

```
VXLAN_FRAME_L3 = {'proto': 'udp',
                  'packetsize': 64,
                  'srcip': TRAFFICGEN_PORT1_IP,
                  'dstip': '192.168.240.1',
                 }
VXLAN_FRAME_L4 = {'srcport': 4789,
                  'dstport': 4789,
                  'vni': VXLAN_VNI,
                  'inner_srcmac': '01:02:03:04:05:06',
                  'inner_dstmac': '06:05:04:03:02:01',
                  'inner_srcip': '192.168.0.10',
                  'inner_dstip': '192.168.240.9',
                  'inner_proto': 'udp',
                  'inner_srcport': 3000,
                  'inner_dstport': 3001,
                 }
```

2.7 Executing GRE decapsulation tests

To run GRE decapsulation tests:

1. Set the variables used in "Executing Tunnel encapsulation tests"
2. Set dstmac of DUT_NIC2_MAC to the MAC adddress of the 2nd NIC of your DUT

```
DUT_NIC2_MAC = '<DUT NIC2 MAC>'
```

3. Run test:

```
./vsperf --conf-file user_settings.py --test-params 'tunnel_type=gre'
         --integration overlay_p2p_decap_cont
```

If you want to use different values for your GRE frame, you may set:

```
GRE_FRAME_L3 = {'proto': 'gre',
                'packetsize': 64,
                'srcip': TRAFFICGEN_PORT1_IP,
                'dstip': '192.168.240.1',
               }

GRE_FRAME_L4 = {'srcport': 0,
                'dstport': 0
```

```
                         'inner_srcmac': '01:02:03:04:05:06',
                         'inner_dstmac': '06:05:04:03:02:01',
                         'inner_srcip': '192.168.0.10',
                         'inner_dstip': '192.168.240.9',
                         'inner_proto': 'udp',
                         'inner_srcport': 3000,
                         'inner_dstport': 3001,
                        }
```

2.8 Executing GENEVE decapsulation tests

IxNet 7.3X does not have native support of GENEVE protocol. The template, GeneveIxNetTemplate.xml_ClearText.xml, should be imported into IxNET for this testcase to work.

To import the template do:

1. Run the IxNetwork TCL Server
2. Click on the Traffic menu
3. Click on the Traffic actions and click Edit Packet Templates
4. On the Template editor window, click Import. Select the template tools/pkt_gen/ixnet/GeneveIxNetTemplate.xml_ClearText.xml and click import.
5. Restart the TCL Server.

To run GENEVE decapsulation tests:

1. Set the variables used in "Executing Tunnel encapsulation tests"
2. Set dstmac of DUT_NIC2_MAC to the MAC adddress of the 2nd NIC of your DUT

```
DUT_NIC2_MAC = '<DUT NIC2 MAC>'
```

3. Run test:

```
./vsperf --conf-file user_settings.py --test-params 'tunnel_type=geneve'
          --integration overlay_p2p_decap_cont
```

If you want to use different values for your GENEVE frame, you may set:

```
GENEVE_FRAME_L3 = {'proto': 'udp',
                   'packetsize': 64,
                   'srcip': TRAFFICGEN_PORT1_IP,
                   'dstip': '192.168.240.1',
                  }

GENEVE_FRAME_L4 = {'srcport': 6081,
                   'dstport': 6081,
                   'geneve_vni': 0,
                   'inner_srcmac': '01:02:03:04:05:06',
                   'inner_dstmac': '06:05:04:03:02:01',
                   'inner_srcip': '192.168.0.10',
                   'inner_dstip': '192.168.240.9',
                   'inner_proto': 'udp',
                   'inner_srcport': 3000,
                   'inner_dstport': 3001,
                  }
```

2.9 Executing Native/Vanilla OVS VXLAN decapsulation tests

To run VXLAN decapsulation tests:

1. Set the following variables in your user_settings.py file:

```
VSWITCH_VANILLA_KERNEL_MODULES = ['vport_vxlan',
                                  os.path.join(OVS_DIR_VANILLA,
                                  'datapath/linux/openvswitch.ko')]

DUT_NIC1_MAC = '<DUT NIC1 MAC ADDRESS>'

TRAFFICGEN_PORT1_IP = '172.16.1.2'
TRAFFICGEN_PORT2_IP = '192.168.1.11'

VTEP_IP1 = '172.16.1.2/24'
VTEP_IP2 = '192.168.1.1'
VTEP_IP2_SUBNET = '192.168.1.0/24'
TUNNEL_EXTERNAL_BRIDGE_IP = '172.16.1.1/24'
TUNNEL_INT_BRIDGE_IP = '192.168.1.1'

VXLAN_FRAME_L2 = {'srcmac':
                  '01:02:03:04:05:06',
                  'dstmac': DUT_NIC1_MAC
                 }

VXLAN_FRAME_L3 = {'proto': 'udp',
                  'packetsize': 64,
                  'srcip': TRAFFICGEN_PORT1_IP,
                  'dstip': '172.16.1.1',
                 }

VXLAN_FRAME_L4 = {
                  'srcport': 4789,
                  'dstport': 4789,
                  'protocolpad': 'true',
                  'vni': 99,
                  'inner_srcmac': '01:02:03:04:05:06',
                  'inner_dstmac': '06:05:04:03:02:01',
                  'inner_srcip': '192.168.1.2',
                  'inner_dstip': TRAFFICGEN_PORT2_IP,
                  'inner_proto': 'udp',
                  'inner_srcport': 3000,
                  'inner_dstport': 3001,
                 }
```

2. Run test:

```
./vsperf --conf-file user_settings.py --integration
         --test-params 'tunnel_type=vxlan' overlay_p2p_decap_cont
```

2.10 Executing Native/Vanilla OVS GRE decapsulation tests

To run GRE decapsulation tests:

1. Set the following variables in your user_settings.py file:

```
VSWITCH_VANILLA_KERNEL_MODULES = ['vport_gre',
                                  os.path.join(OVS_DIR_VANILLA,
                                  'datapath/linux/openvswitch.ko')]

DUT_NIC1_MAC = '<DUT NIC1 MAC ADDRESS>'

TRAFFICGEN_PORT1_IP = '172.16.1.2'
TRAFFICGEN_PORT2_IP = '192.168.1.11'

VTEP_IP1 = '172.16.1.2/24'
VTEP_IP2 = '192.168.1.1'
VTEP_IP2_SUBNET = '192.168.1.0/24'
TUNNEL_EXTERNAL_BRIDGE_IP = '172.16.1.1/24'
TUNNEL_INT_BRIDGE_IP = '192.168.1.1'

GRE_FRAME_L2 = {'srcmac':
                 '01:02:03:04:05:06',
                 'dstmac': DUT_NIC1_MAC
                }

GRE_FRAME_L3 = {'proto': 'udp',
                 'packetsize': 64,
                 'srcip': TRAFFICGEN_PORT1_IP,
                 'dstip': '172.16.1.1',
                }

GRE_FRAME_L4 = {
                 'srcport': 4789,
                 'dstport': 4789,
                 'protocolpad': 'true',
                 'inner_srcmac': '01:02:03:04:05:06',
                 'inner_dstmac': '06:05:04:03:02:01',
                 'inner_srcip': '192.168.1.2',
                 'inner_dstip': TRAFFICGEN_PORT2_IP,
                 'inner_proto': 'udp',
                 'inner_srcport': 3000,
                 'inner_dstport': 3001,
                }
```

2. Run test:

```
./vsperf --conf-file user_settings.py --integration
         --test-params 'tunnel_type=gre' overlay_p2p_decap_cont
```

2.11 Executing Native/Vanilla OVS GENEVE decapsulation tests

To run GENEVE decapsulation tests:

1. Set the following variables in your user_settings.py file:

```
VSWITCH_VANILLA_KERNEL_MODULES = ['vport_geneve',
                                  os.path.join(OVS_DIR_VANILLA,
                                  'datapath/linux/openvswitch.ko')]

DUT_NIC1_MAC = '<DUT NIC1 MAC ADDRESS>'

TRAFFICGEN_PORT1_IP = '172.16.1.2'
```

```
TRAFFICGEN_PORT2_IP = '192.168.1.11'

VTEP_IP1 = '172.16.1.2/24'
VTEP_IP2 = '192.168.1.1'
VTEP_IP2_SUBNET = '192.168.1.0/24'
TUNNEL_EXTERNAL_BRIDGE_IP = '172.16.1.1/24'
TUNNEL_INT_BRIDGE_IP = '192.168.1.1'

GENEVE_FRAME_L2 = {'srcmac':
                   '01:02:03:04:05:06',
                   'dstmac': DUT_NIC1_MAC
                  }

GENEVE_FRAME_L3 = {'proto': 'udp',
                   'packetsize': 64,
                   'srcip': TRAFFICGEN_PORT1_IP,
                   'dstip': '172.16.1.1',
                  }

GENEVE_FRAME_L4 = {'srcport': 6081,
                   'dstport': 6081,
                   'protocolpad': 'true',
                   'geneve_vni': 0,
                   'inner_srcmac': '01:02:03:04:05:06',
                   'inner_dstmac': '06:05:04:03:02:01',
                   'inner_srcip': '192.168.1.2',
                   'inner_dstip': TRAFFICGEN_PORT2_IP,
                   'inner_proto': 'udp',
                   'inner_srcport': 3000,
                   'inner_dstport': 3001,
                  }
```

2. Run test:

```
./vsperf --conf-file user_settings.py --integration
         --test-params 'tunnel_type=geneve' overlay_p2p_decap_cont
```

2.12 Executing Tunnel encapsulation+decapsulation tests

The OVS DPDK encapsulation_decapsulation tests requires IPs, MAC addresses, bridge names and WHITELIST_NICS for DPDK.

The test cases can test the tunneling encap and decap without using any ingress overlay traffic as compared to above test cases. To achieve this the OVS is configured to perform encap and decap in a series on the same traffic stream as given below.

TRAFFIC-IN –> [ENCAP] –> [MOD-PKT] –> [DECAP] –> TRAFFIC-OUT

Default values are already provided. To customize for your environment, override the following variables in you user_settings.py file:

```
# Variables defined in conf/integration/02_vswitch.conf

# Bridge names
TUNNEL_EXTERNAL_BRIDGE1 = 'br-phy1'
TUNNEL_EXTERNAL_BRIDGE2 = 'br-phy2'
TUNNEL_MODIFY_BRIDGE1 = 'br-mod1'
```

```
TUNNEL_MODIFY_BRIDGE2 = 'br-mod2'

# IP of br-mod1
TUNNEL_MODIFY_BRIDGE_IP1 = '10.0.0.1/24'

# Mac of br-mod1
TUNNEL_MODIFY_BRIDGE_MAC1 = '00:00:10:00:00:01'

# IP of br-mod2
TUNNEL_MODIFY_BRIDGE_IP2 = '20.0.0.1/24'

#Mac of br-mod2
TUNNEL_MODIFY_BRIDGE_MAC2 = '00:00:20:00:00:01'

# vxlan|gre|geneve, Only VXLAN is supported for now.
TUNNEL_TYPE = 'vxlan'
```

To run VXLAN encapsulation+decapsulation tests:

```
./vsperf --conf-file user_settings.py --integration
         overlay_p2p_mod_tput
```

CHAPTER

THREE

EXECUTION OF VSWITCHPERF TESTCASES BY YARDSTICK

3.1 General

Yardstick is a generic framework for a test execution, which is used for validation of installation of OPNFV platform. In the future, Yardstick will support two options of vswitchperf testcase execution:

- plugin mode, which will execute native vswitchperf testcases; Tests will be executed natively by vsperf, and test results will be processed and reported by yardstick.
- traffic generator mode, which will run vswitchperf in **trafficgen** mode only; Yardstick framework will be used to launch VNFs and to configure flows to ensure, that traffic is properly routed. This mode will allow to test OVS performance in real world scenarios.

In Colorado release only the traffic generator mode is supported.

3.2 Yardstick Installation

In order to run Yardstick testcases, you will need to prepare your test environment. Please follow the installation instructions to install the yardstick.

Please note, that yardstick uses OpenStack for execution of testcases. OpenStack must be installed with Heat and Neutron services. Otherwise vswitchperf testcases cannot be executed.

3.3 Vswitchperf VM image preparation

In general, any Linux distribution supported by vswitchperf can be used as a base image for vswitchperf. One of the possibilities is to modify vloop-vnf image, which can be downloaded from http://artifacts.opnfv.org/.

```
$ wget http://artifacts.opnfv.org/vswitchperf/vloop-vnf-ubuntu-14.04_20151216.qcow2
```

Please follow the installation instructions to install vswitchperf inside vloop-vnf image. As vswitchperf will be run in trafficgen mode, it is possible to skip installation and compilation of OVS, QEMU and DPDK to keep image size smaller.

In case, that selected traffic generator requires installation of additional client software, please follow appropriate documentation. For example in case of IXIA, you would need to install IxOS and IxNetowrk TCL API.

Final image with vswitchperf must be uploaded into the glance service and vswitchperf specific flavor configured, e.g.:

```
$ glance --os-username admin --os-image-api-version 1 image-create --name
  vsperf --is-public true --disk-format qcow2 --container-format bare --file
  image.qcow2

$ nova --os-username admin flavor-create vsperf-flavor 100 2048 25 1
```

3.4 Testcase customization

Yardstick testcases are described by YAML files. vswitchperf specific testcases are part of the vswitchperf repository and their yaml files can be found at `yardstick/tests` directory. For detailed description of yaml file structure, please see yardstick documentation and testcase samples. Only vswitchperf specific parts will be discussed here.

Example of yaml file:

```
...
scenarios:
-
  type: Vsperf
  options:
    testname: 'rfc2544_p2p_tput'
    traffic_type: 'rfc2544'
    pkt_sizes: '64'
    bidirectional: 'True'
    iload: 100
    duration: 30
    trafficgen_port1: 'eth1'
    trafficgen_port2: 'eth3'
    external_bridge: 'br-ex'
    conf-file: '~/vsperf-yardstick.conf'

  host: vsperf.demo

  runner:
    type: Sequence
    scenario_option_name: pkt_sizes
    sequence:
    - 64
    - 128
    - 512
    - 1024
    - 1518
  sla:
    metrics: 'throughput_rx_fps'
    throughput_rx_fps: 500000
    action: monitor

context:
...
```

3.4.1 Section option

Section **option** defines details of vswitchperf test scenario. Lot of options are identical to the vswitchperf parameters passed through `--test-params` argument. Following options are supported:

- **traffic_type** - specifies the type of traffic executed by traffic generator; valid values are “rfc2544”, “continuous” and “back2back”; Default: ‘rfc2544’
- **pkt_sizes** - a packet size for which test should be executed; Multiple packet sizes can be tested by modification of Sequence runner section inside YAML definition. Default: ‘64’
- **duration** - sets duration for which traffic will be generated; Default: 30
- **bidirectional** - specifies if traffic will be uni (False) or bi-directional (True); Default: False
- **iload** - specifies frame rate; Default: 100
- **rfc2544_tests** - specifies the number of tests performed for each packet size
- **multistream** - specifies the number of simulated streams; Default: 0 (i.e. multistream feature is disabled)
- **stream_type** - specifies network layer used for multistream simulation the valid values are “L4”, “L3” and “L2”; Default: ‘L4’
- **conf-file** - sets path to the vswitchperf configuration file, which will be uploaded to VM; Default: ‘~/vsperf-yardstick.conf’
- **setup-script** - sets path to the setup script, which will be executed during setup and teardown phases
- **trafficgen_port1** - specifies device name of 1st interface connected to the trafficgen
- **trafficgen_port2** - specifies device name of 2nd interface connected to the trafficgen
- **external_bridge** - specifies name of external bridge configured in OVS; Default: ‘br-ex’

In case that **trafficgen_port1** and/or **trafficgen_port2** are defined, then these interfaces will be inserted into the **external_bridge** of OVS. It is expected, that OVS runs at the same node, where the testcase is executed. In case of more complex OpenStack installation or a need of additional OVS configuration, **setup-script** can be used.

Note: It is essential to prepare customized configuration file for the vsperf and to specify its name by **conf-file** option. Config file must specify, which traffic generator will be used and configure traffic generator specific options.

3.4.2 Section runner

Yardstick supports several runner types. In case of vswitchperf specific TCs, **Sequence** runner type can be used to execute the testcase for given list of packet sizes.

3.4.3 Section sla

In case that sla section is not defined, then testcase will be always considered as successful. On the other hand, it is possible to define a set of test metrics and their minimal values to evaluate test success. Any numeric value, reported by vswitchperf inside CSV result file, can be used. Multiple metrics can be defined as a coma separated list of items. Minimal value must be set separately for each metric.

e.g.:

```
sla:
    metrics: 'throughput_rx_fps,throughput_rx_mbps'
    throughput_rx_fps: 500000
    throughput_rx_mbps: 1000
```

In case that any of defined metrics will be lower than defined value, then testcase will be marked as failed. Based on `action` policy, yardstick will either stop test execution (value `assert`) or it will run next test (value `monitor`).

3.5 Testcase execution

After installation, yardstick is available as python package within yardstick specific virtual environment. It means, that before test execution yardstick environment must be enabled, e.g.:

```
source ~/yardstick_venv/bin/activate
```

Next step is configuration of OpenStack environment, e.g. in case of devstack:

```
source /opt/openstack/devstack/openrc
export EXTERNAL_NETWORK=public
```

Vswitchperf testcases executable by yardstick are located at vswitchperf repository inside `yardstick/tests` directory. Example of their download and execution follows:

```
git clone https://gerrit.opnfv.org/gerrit/vswitchperf
cd vswitchperf

yardstick -d task start yardstick/tests/p2p_cont.yaml
```

Note: Optional argument `-d` shows debug output.