VSPERF Design

Release draft (7c91fcd)

OPNFV

April 20, 2016

CHAPTER

ONE

VSPERF DESIGN DOCUMENT

1.1 Intended Audience

This document is intended to aid those who want to modify the vsperf code. Or to extend it - for example to add support for new traffic generators, deployment scenarios and so on.

1.2 Usage

1.2.1 Example Connectivity to DUT

Establish connectivity to the VSPERF DUT Linux host, such as the DUT in Pod 3, by following the steps in Testbed POD3

The steps cover booking the DUT and establishing the VSPERF environment.

1.2.2 Example Command Lines

List all the cli options:

```
$ ./vsperf -h
```

Run all tests that have `tput` in their name - `p2p_tput`, `pvp_tput` etc.:

```
$ ./vsperf --tests 'tput'
```

As above but override default configuration with settings in '10_custom.conf'. This is useful as modifying configuration directly in the configuration files in `conf/NN_*.py` shows up as changes under git source control:

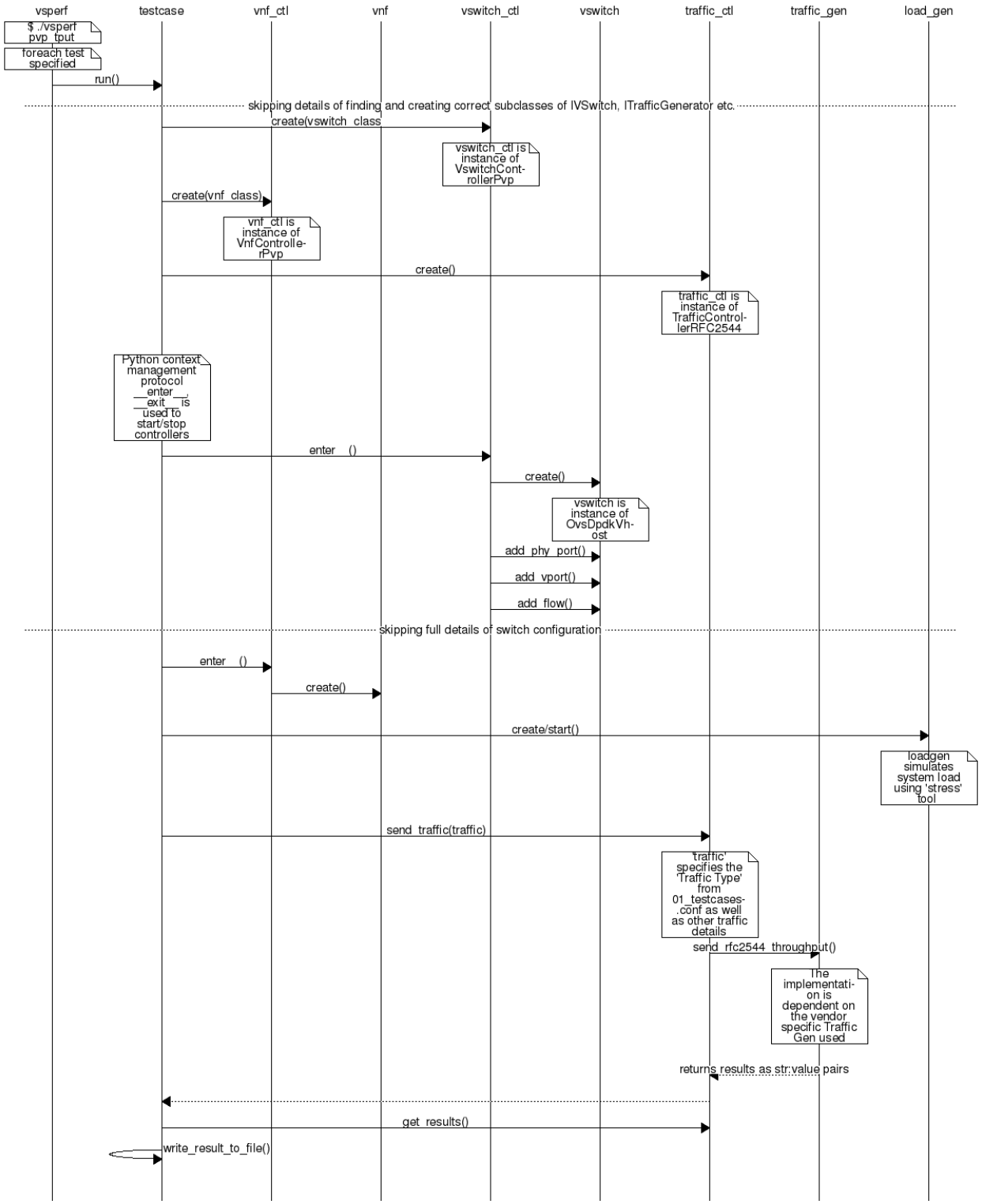
```
$ ./vsperf --conf-file=<path_to_custom_conf>/10_custom.conf --tests 'tput'
```

Override specific test parameters. Useful for shortening the duration of tests for development purposes:

```
$ ./vsperf --test-params 'duration=10;rfc2544_trials=1;pkt_sizes=64' --tests 'pvp_tput'
```

1.3 Typical Test Sequence

This is a typical flow of control for a test.


vsperf
testcase
vnf_ctl
vnf
vswitch_ctl
vswitch
traffic_ctl
traffic_gen
load_gen
$ ./vsperf pvp_tput
foreach test specified
run()
skipping details of finding and creating correct subclasses of IVSwitch, ITrafficGenerator etc.
create(vswitch_class
vswitch_ctl is instance of VswitchControllerPvp
create(vnf_class)
vnf_ctl is instance of VnfControllerPvp
create()
traffic_ctl is instance of TrafficControllerRFC2544
Python context management protocol __enter__, __exit__ is used to start/stop controllers
enter ()
create()
vswitch is instance of OvsDpdkVhost
add_phy_port()
add_vport()
add_flow()
skipping full details of switch configuration
enter ()
create()
create/start()
loadgen simulates system load using 'stress' tool
send_traffic(traffic)
'traffic' specifies the 'Traffic Type' from 01_testcases.conf as well as other traffic details
send_rfc2544_throughput()
The implementation is dependent on the vendor specific Traffic Gen used
returns results as str:value pairs
get_results()
write_result_to_file()

1.4 Configuration

The conf package contains the configuration files (`*.conf`) for all system components, it also provides a `settings` object that exposes all of these settings.

Settings are not passed from component to component. Rather they are available globally to all components once they import the conf package.

```
from conf import settings
...
log_file = settings.getValue('LOG_FILE_DEFAULT')
```

Settings files (`*.conf`) are valid python code so can be set to complex types such as lists and dictionaries as well as scalar types:

```
first_packet_size = settings.getValue('PACKET_SIZE_LIST')[0]
```

1.4.1 Configuration Procedure and Precedence

Configuration files follow a strict naming convention that allows them to be processed in a specific order. All the .conf files are named `NN_name.conf`, where NN is a decimal number. The files are processed in order from 00_name.conf to 99_name.conf so that if the name setting is given in both a lower and higher numbered conf file then the higher numbered file is the effective setting as it is processed after the setting in the lower numbered file.

The values in the file specified by `--conf-file` takes precedence over all the other configuration files and does not have to follow the naming convention.

1.4.2 Other Configuration

`conf.settings` also loads configuration from the command line and from the environment.

1.5 VM, vSwitch, Traffic Generator Independence

VSPERF supports different vSwithes, Traffic Generators, VNFs and Forwarding Applications by using standard object-oriented polymorphism:

- Support for vSwitches is implemented by a class inheriting from IVSwitch.
- Support for Traffic Generators is implemented by a class inheriting from ITrafficGenerator.
- Support for VNF is implemented by a class inheriting from IVNF.
- Support for Forwarding Applications is implemented by a class inheriting from IPktFwd.

By dealing only with the abstract interfaces the core framework can support many implementations of different vSwitches, Traffic Generators, VNFs and Forwarding Applications.

1.5.1 IVSwitch

```
class IVSwitch:
  start(self)
  stop(self)
  add_switch(switch_name)
```

```
  del_switch(switch_name)
  add_phy_port(switch_name)
  add_vport(switch_name)
  get_ports(switch_name)
  del_port(switch_name, port_name)
  add_flow(switch_name, flow)
  del_flow(switch_name, flow=None)
```

1.5.2 ITrafficGenerator

```
class ITrafficGenerator:
  connect()
  disconnect()

  send_burst_traffic(traffic, numpkts, time, framerate)

  send_cont_traffic(traffic, time, framerate)
  start_cont_traffic(traffic, time, framerate)
  stop_cont_traffic(self):

  send_rfc2544_throughput(traffic, trials, duration, lossrate)
  start_rfc2544_throughput(traffic, trials, duration, lossrate)
  wait_rfc2544_throughput(self)

  send_rfc2544_back2back(traffic, trials, duration, lossrate)
  start_rfc2544_back2back(traffic, , trials, duration, lossrate)
  wait_rfc2544_back2back()
```

Note `send_xxx()` blocks whereas `start_xxx()` does not and must be followed by a subsequent call to `wait_xxx()`.

1.5.3 IVnf

```
class IVnf:
  start(memory, cpus,
        monitor_path, shared_path_host,
        shared_path_guest, guest_prompt)
  stop()
  execute(command)
  wait(guest_prompt)
  execute_and_wait (command)
```

1.5.4 IPktFwd

```
    class IPktFwd:
        start()
        stop()
```

1.5.5 Controllers

Controllers are used in conjunction with abstract interfaces as way of decoupling the control of vSwtiches, VNFs, TrafficGenerators and Forwarding Applications from other components.

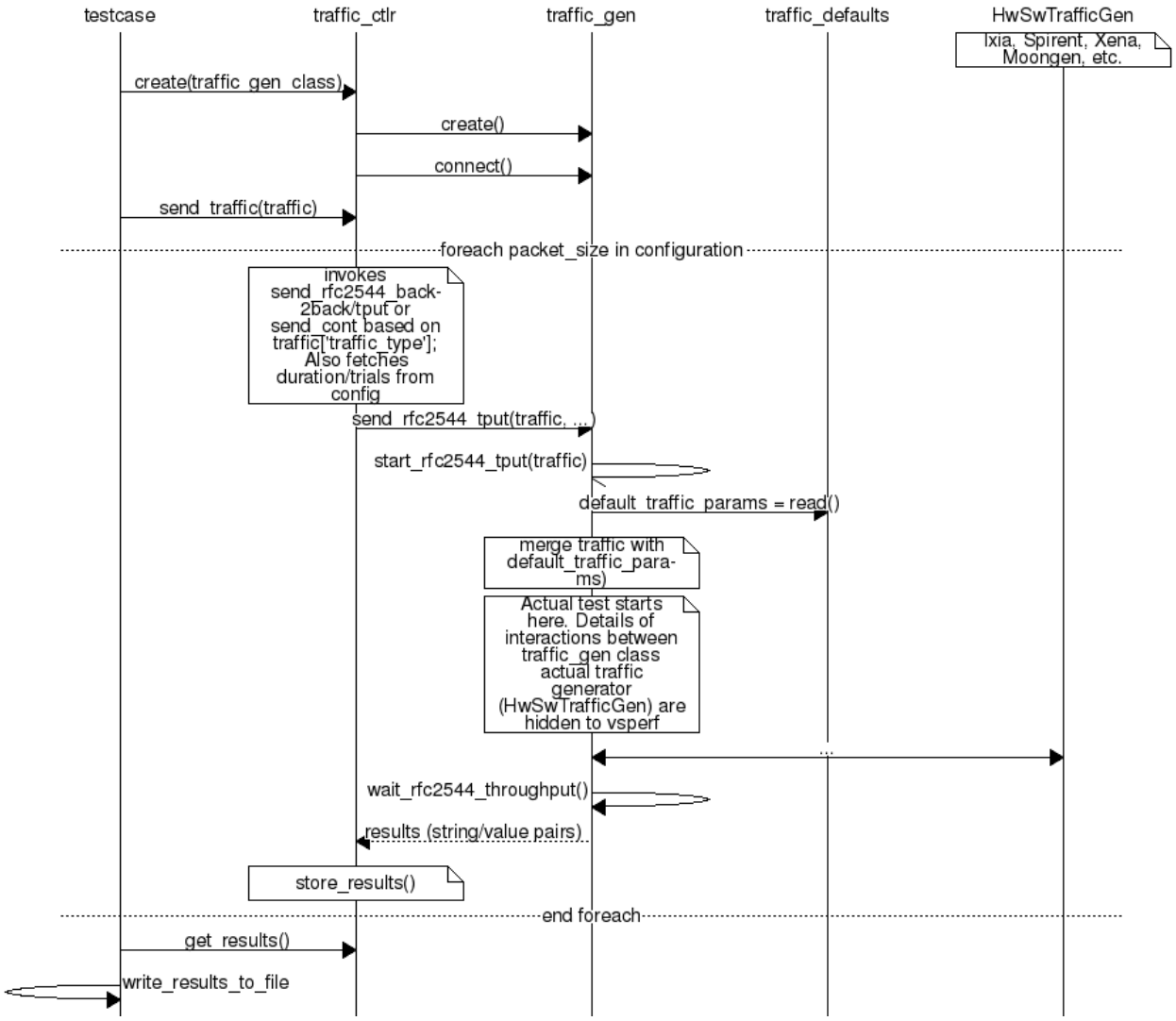
The controlled classes provide basic primitive operations. The Controllers sequence and co-ordinate these primitive operation in to useful actions. For instance the vswitch_controller_PVP can be used to bring any vSwitch (that implements the primitives defined in IVSwitch) into the configuration required by the Phy-to-Phy Deployment Scenario.

In order to support a new vSwitch only a new implementation of IVSwitch needs be created for the new vSwitch to be capable of fulfilling all the Deployment Scenarios provided for by existing or future vSwitch Controllers.

Similarly if a new Deployment Scenario is required it only needs to be written once as a new vSwitch Controller and it will immediately be capable of controlling all existing and future vSwitches in to that Deployment Scenario.

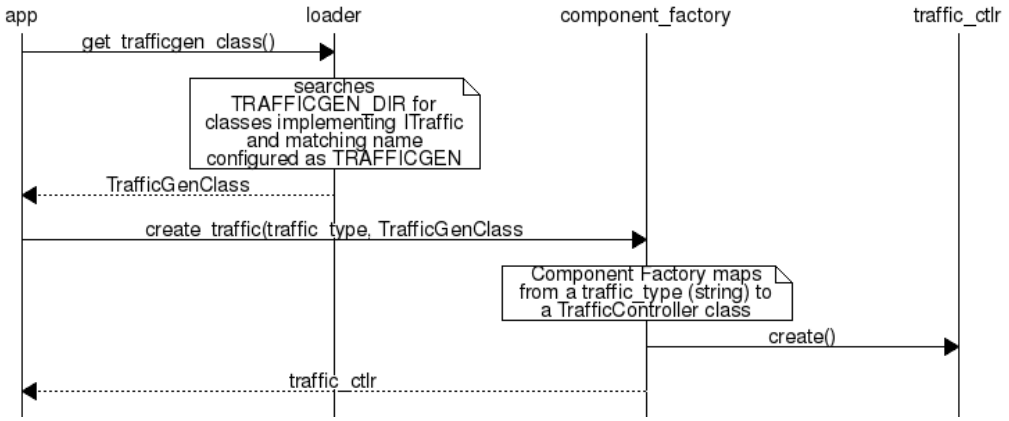
Similarly the Traffic Controllers can be used to co-ordinate basic operations provided by implementers of ITraffic-Generator to provide useful tests. Though traffic generators generally already implement full test cases i.e. they both generate suitable traffic and analyse returned traffic in order to implement a test which has typically been predefined in an RFC document. However the Traffic Controller class allows for the possibility of further enhancement - such as iterating over tests for various packet sizes or creating new tests.

1.5.6 Traffic Controller's Role



1.5.7 Loader & Component Factory

The working of the Loader package (which is responsible for *finding* arbitrary classes based on configuration data) and the Component Factory which is responsible for *choosing* the correct class for a particular situation - e.g. Deployment Scenario can be seen in this diagram.



1.6 Routing Tables

Vsperf uses a standard set of routing tables in order to allow tests to easily mix and match Deployment Scenarios (PVP, P2P topology), Tuple Matching and Frame Modification requirements.

```
+--------------+
|              |
| Table 0      |  table#0 - Match table. Flows designed to force 5 & 10
|              |  tuple matches go here.
|              |
+--------------+
       |
       |
       v
+--------------+  table#1 - Routing table. Flow entries to forward
|              |  packets between ports goes here.
| Table 1      |  The chosen port is communicated to subsequent tables by
|              |  setting the metadata value to the egress port number.
|              |  Generally this table is set-up by by the
+--------------+  vSwitchController.
       |
       |
       v
+--------------+  table#2 - Frame modification table. Frame modification
|              |  flow rules are isolated in this table so that they can
| Table 2      |  be turned on or off without affecting the routing or
|              |  tuple-matching flow rules. This allows the frame
|              |  modification and tuple matching required by the tests
|              |  in the VSWITCH PERFORMANCE FOR TELCO NFV test
+--------------+  specification to be independent of the Deployment
       |          Scenario set up by the vSwitchController.
       |
       v
+--------------+
|              |
| Table 3      |  table#3 - Egress table. Egress packets on the ports
|              |  setup in Table 1.
+--------------+
```

CHAPTER

TWO

TRAFFIC GENERATOR INTEGRATION GUIDE

2.1 Intended Audience

This document is intended to aid those who want to integrate new traffic generator into the vsperf code. It is expected, that reader has already read generic part of VSPERF Design Document.

Let us create a sample traffic generator called **sample_tg**, step by step.

2.2 Step 1 - create a directory

Implementation of trafficgens is located at tools/pkt_gen/ directory, where every implementation has its dedicated sub-directory. It is required to create a new directory for new traffic generator implementations.

E.g.

```
$ mkdir tools/pkt_gen/sample_tg
```

2.3 Step 2 - create a trafficgen module

Every trafficgen class must inherit from generic **ITrafficGenerator** interface class. VSPERF during its initialization scans content of pkt_gen directory for all python modules, that inherit from **ITrafficGenerator**. These modules are automatically added into the list of supported traffic generators.

Example:

Let us create a draft of tools/pkt_gen/sample_tg/sample_tg.py module.

```
from tools.pkt_gen import trafficgen

class SampleTG(trafficgen.ITrafficGenerator):
    """
    A sample traffic generator implementation
    """
    pass
```

VSPERF is immediately aware of the new class:

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

Output should look like:

```
Classes derived from: ITrafficGenerator
======

* Ixia:             A wrapper around the IXIA traffic generator.

* IxNet:            A wrapper around IXIA IxNetwork applications.

* Dummy:            A dummy traffic generator whose data is generated by the user.

* SampleTG:         A sample traffic generator implementation

* TestCenter:       Spirent TestCenter
```

2.4 Step 3 - configuration

All configuration values, required for correct traffic generator function, are passed from VSPERF to the traffic generator in a dictionary. Default values shared among all traffic generators are defined in **tools/pkt_gen/trafficgen/trafficgenhelper.py** as **TRAFFIC_DEFAULTS** dictionary. Default values are loaded by **ITrafficGenerator** interface class automatically, so it is not needed to load them explicitly. In case that there are any traffic generator specific default values, then they should be set within class specific **__init__** function.

VSPERF passes test specific configuration within **traffic** dictionary to every start and send function. So implementation of these functions must ensure, that default values are updated with the testcase specific values. Proper merge of values is assured by call of **merge_spec** function from **trafficgenhelper** module.

Example of **merge_spec** usage in **tools/pkt_gen/sample_tg/sample_tg.py** module:

```
from tools.pkt_gen.trafficgen.trafficgenhelper import merge_spec

def start_rfc2544_throughput(self, traffic=None, duration=30):
    self._params = {}
    self._params['traffic'] = self.traffic_defaults.copy()
    if traffic:
        self._params['traffic'] = trafficgen.merge_spec(
            self._params['traffic'], traffic)
```

2.5 Step 4 - generic functions

There are some generic functions, which every traffic generator should provide. Although these functions are mainly optional, at least empty implementation must be provided. This is required, so that developer is explicitly aware of these functions.

The **connect** function is called from the traffic generator controller from its **__enter__** method. This function should assure proper connection initialization between DUT and traffic generator. In case, that such implementation is not needed, empty implementation is required.

The **disconnect** function should perform clean up of any connection specific actions called from the **connect** function.

Example in **tools/pkt_gen/sample_tg/sample_tg.py** module:

```
def connect(self):
    pass

def disconnect(self):
    pass
```

2.6 Step 5 - supported traffic types

Currently VSPERF supports three different types of tests for traffic generators, these are identified in vsperf through the traffic type, which include:

- **RFC2544 throughput - Send fixed size packets at different rates, using** traffic configuration, until minimum rate at which no packet loss is detected is found. Methods with its implementation have suffix **_rfc2544_throughput**.
- **RFC2544 back2back - Send fixed size packets at a fixed rate, using traffic** configuration, for specified time interval. Methods with its implementation have suffix **_rfc2544_back2back**.
- **continuous flow - Send fixed size packets at given framerate, using traffic** configuration, for specified time interval. Methods with its implementation have suffix **_cont_traffic**.

In general, both synchronous and asynchronous interfaces must be implemented for each traffic type. Synchronous functions start with prefix **send_**. Asynchronous with prefixes **start_** and **wait_** in case of throughput and back2back and **start_** and **stop_** in case of continuous traffic type.

Example of synchronous interfaces:

```
def send_rfc2544_throughput(self, traffic=None, trials=3, duration=20,
                            lossrate=0.0):
def send_rfc2544_back2back(self, traffic=None, trials=1, duration=20,
                           lossrate=0.0):
def send_cont_traffic(self, traffic=None, duration=20):
```

Example of asynchronous interfaces:

```
def start_rfc2544_throughput(self, traffic=None, trials=3, duration=20,
                             lossrate=0.0):
def wait_rfc2544_throughput(self):

def start_rfc2544_back2back(self, traffic=None, trials=1, duration=20,
                            lossrate=0.0):
def wait_rfc2544_back2back(self):

def start_cont_traffic(self, traffic=None, duration=20):
def stop_cont_traffic(self):
```

Description of parameters used by **send**, **start**, **wait** and **stop** functions:

- param **traffic**: A dictionary with detailed definition of traffic pattern. It contains following parameters to be implemented by traffic generator.

  Note: Traffic dictionary has also virtual switch related parameters, which are not listed below.

  Note: There are parameters specific to testing of tunnelling protocols, which are discussed in detail at integration tests userguide

  - param **traffic_type**: One of the supported traffic types, e.g. **rfc2544**, **continuous** or **back2back**.
  - param **frame_rate**: Defines desired percentage of frame rate used during continuous stream tests. It can be set by test parameter iLoad or by CLI parameter iload.
  - param **bidir**: Specifies if generated traffic will be full-duplex (true) or half-duplex (false).
  - param **multistream**: Defines number of flows simulated by traffic generator. Value 0 disables MultiStream feature.
  - param **stream_type**: Stream Type defines ISO OSI network layer used for simulation of multiple streams. Supported values:

        * **L2** - iteration of destination MAC address
        * **L3** - iteration of destination IP address
        * **L4** - iteration of destination port of selected transport protocol
    - param **l2**: A dictionary with data link layer details, e.g. **srcmac**, **dstmac** and **framesize**.
    - param **l3**: A dictionary with network layer details, e.g. **srcip**, **dstip** and **proto**.
    - param **l3**: A dictionary with transport layer details, e.g. **srcport**, **dstport**.
    - param **vlan**: A dictionary with vlan specific parameters, e.g. **priority**, **cfi**, **id** and vlan on/off switch **enabled**.
- param **trials**: Number of trials to execute.
- param **duration**: Duration of continuous test or per iteration duration in case of RFC2544 throughput or back2back traffic types.
- param **lossrate**: Acceptable lossrate percentage.

2.7 Step 6 - passing back results

It is expected that methods **send**, **wait** and **stop** will return values measured by traffic generator within a dictionary. Dictionary keys are defined in **ResultsConstants** implemented in **core/results/results_constants.py**. Please check sections for RFC2544 Throughput & Continuous and for Back2Back. The same key names should be used by all traffic generator implementations.