TRex Stateless support

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Audience

This document assumes basic knowledge of TRex, and assumes that TRex is installed and configured. For information, see the manual, especially the material up to the Basic Usage section.

Chapter 2

Stateless support

2.1 High level functionality

- Large scale - Supports about 10-22 million packets per second (mpps) per core, scalable with the number of cores
- Support for 1, 10, 25, 40, and 100 Gb/sec interfaces
- Support for multiple traffic profiles per interface
- Profile can support multiple streams, scalable to 10K parallel streams
- Supported for each stream:
  - Packet template - ability to build any packet (including malformed) using Scapy (example: MPLS/IPv4/Ipv6/GRE/VXLAN/NSH)
  - Field Engine program
    - Ability to change any field inside the packet (example: src_ip = 10.0.0.1-10.0.0.255)
    - Ability to change the packet size (example: random packet size 64-9K)
  - Mode - Continuous/Burst/Multi-burst support
  - Rate can be specified as:
    - Packets per second (example: 14MPPS)
    - L1 bandwidth (example: 500Mb/sec)
    - L2 bandwidth (example: 500Mb/sec)
    - Interface link percentage (example: 10%)
  - Support for HLTAPI-like profile definition
  - Action - stream can trigger a stream
- Interactive support - Fast Console, GUI
- Statistics per interface
- Statistics per stream done in hardware
- Latency and jitter per stream
- Blazingly fast automation support
  - Python 2.7/3.0 Client API
  - Python HLTAPI Client API
- Multi-user support - multiple users can interact with the same TRex instance simultaneously
- Routing protocols support — RIP/BGP/OSPF using BIRD

2.1.1 Traffic profile example

The following example shows three streams configured for Continuous, Burst, and Multi-burst traffic.

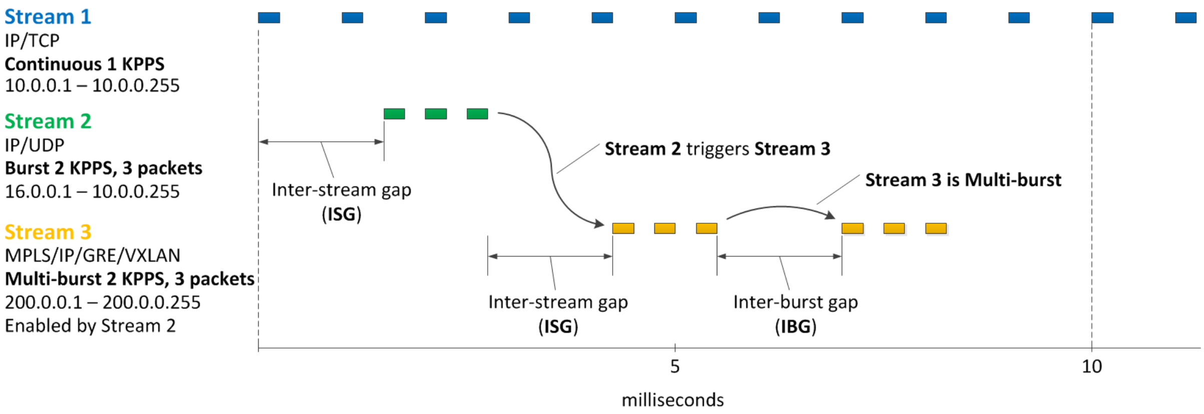


Figure 2.1: Example of multiple streams

2.2 IXIA IXExplorer vs TRex

TRex has limited functionality compared to IXIA, but has some advantages. The following table summarizes the differences:

Table 2.1: TRex vs IXExplorer

Feature	IXExplorer	TRex	Description
Line rate	Yes	10-24MPPS/core, depends on the use case	
Multi stream	255	**Software limited to ~20K**	
Packet build flexibility	Limited	**Scapy - Unlimited**	Example: GRE/VXLAN/NSH is supported. Can be extended to future protocols.
Packet Field Engine	Limited	**Unlimited**	
Tx Mode	Continuous/Burst/Multi-burst	Continuous/Burst/Multi-burst	
ARP/IPv6 ND Emulation	Yes	Yes	
DHCP Client Emulation	Yes	Yes	
Extendable Emulation framework	No	Yes	
Automation	TCL/Python wrapper to TCL	**Native Python/Scapy**	

Table 2.1: (continued)

Feature	IXExplorer	TRex	Description
Automation speed sec	30 sec	**1 msec**	Test of load/start/stop/get counters
HLTAPI	Full support. 2000 pages of documentation	Limited. 20 pages of documentation	
Per Stream statistics	255 streams with 4 global masks	128 rules for XL710/X710 hardware and software impl for 82599/I350/X550	Some packet type restrictions apply to XL710/X710. Software mode can be extended to 32K rules.
Latency Jitter	Yes. Nanosecond resolution (hardware-based)	Yes. Microsecond resolution (software-based)	
Multi-user support	Yes	Yes	
GUI	Very good	WIP, packet builder, Field Engine, global port statistics, latency, per stream statistics. Differs from IXIA GUI - for details, see: trex-stateless-gui	
Cisco pyATS support	Yes	Yes - Python 2.7/Python 3.4	
Routing Emulation	Yes	Yes	

2.3 RPC Architecture

A JSON-RPC2 thread in the TRex control plane core provides support for interactive mode.

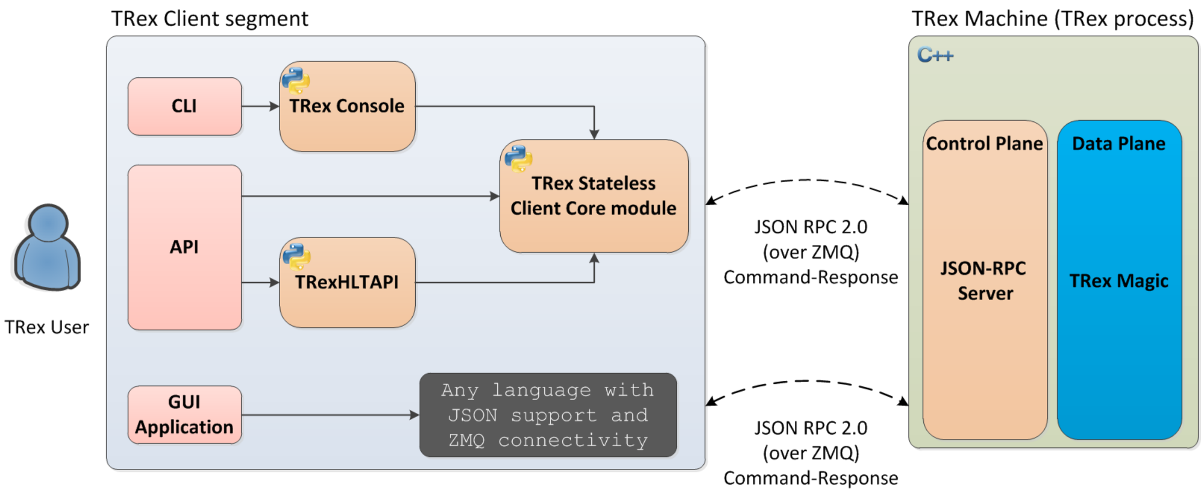


Figure 2.2: RPC server components

Layers

- Control transport protocol: ZMQ working in REQ/RES mode.
- RPC protocol on top of the control transport protocol: JSON-RPC2.
- Asynchronous transport: ZMQ working in SUB/PUB mode (used for asynchronous events such as interface change mode, counters, and so on).

Interfaces

- Automation API: Python is the first client to implement the Python automation API.
- User interface: The console uses the Python API to implement a user interface for TRex.
- GUI : The GUI works on top of the JSON-RPC2 layer.

Control of TRex interfaces

- Numerous users can control a single TRex server together, from different interfaces.
- Users acquire individual TRex interfaces exclusively. **Example**: Two users control a 4-port TRex server. User A acquires interfaces 0 and 1; User B acquires interfaces 3 and 4.
- Only one user interface (console or GUI) can have read/write control of a specific interface. This enables caching the TRex server interface information in the client core. **Example**: User A, with two acquired interfaces, can have only one read/write control session at a time.
- A user can set up numerous read-only clients on a single interface - for example, for monitoring traffic statistics on the interface.
- A client in read-write mode can acquire a statistic in real time (with ASYNC ZMQ). This enables viewing statistics through numerous user interfaces (console and GUI) simultaneously.

Synchronization

- A client syncs with the TRex server to get the state in connection time, and caches the server information locally after the state has changed.
- If a client crashes or exits, it syncs again after reconnecting.

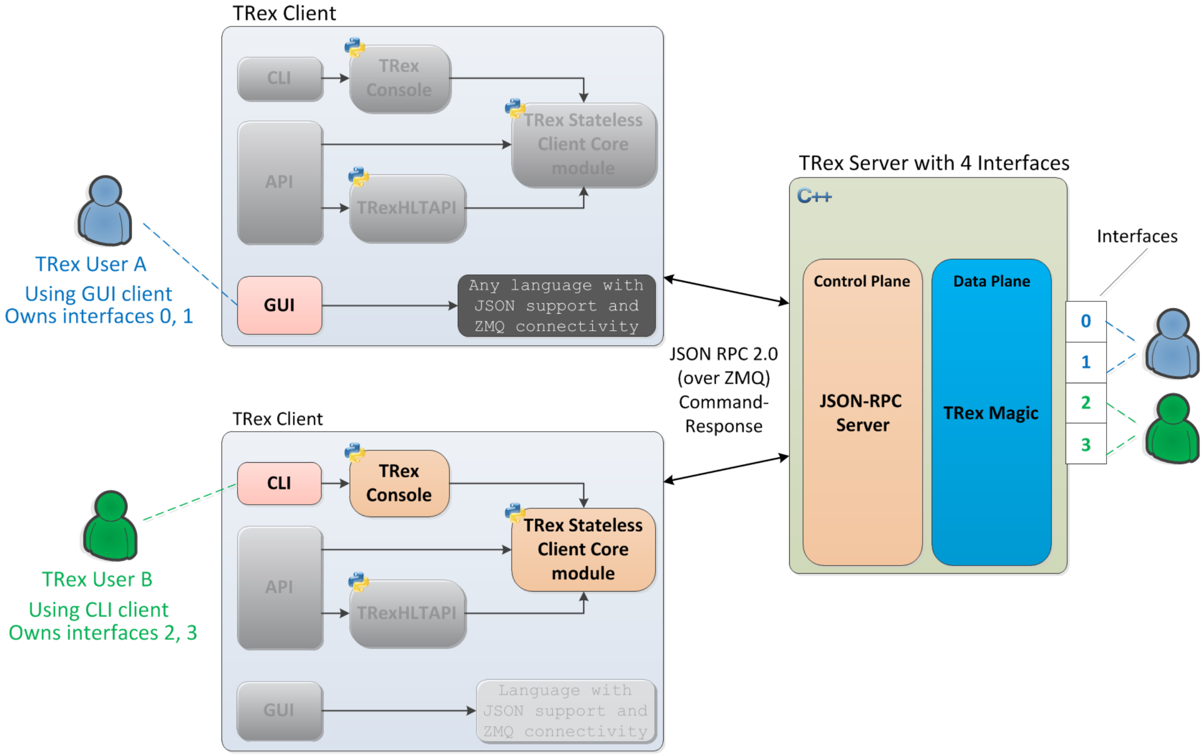


Figure 2.3: Multiple users, per interface

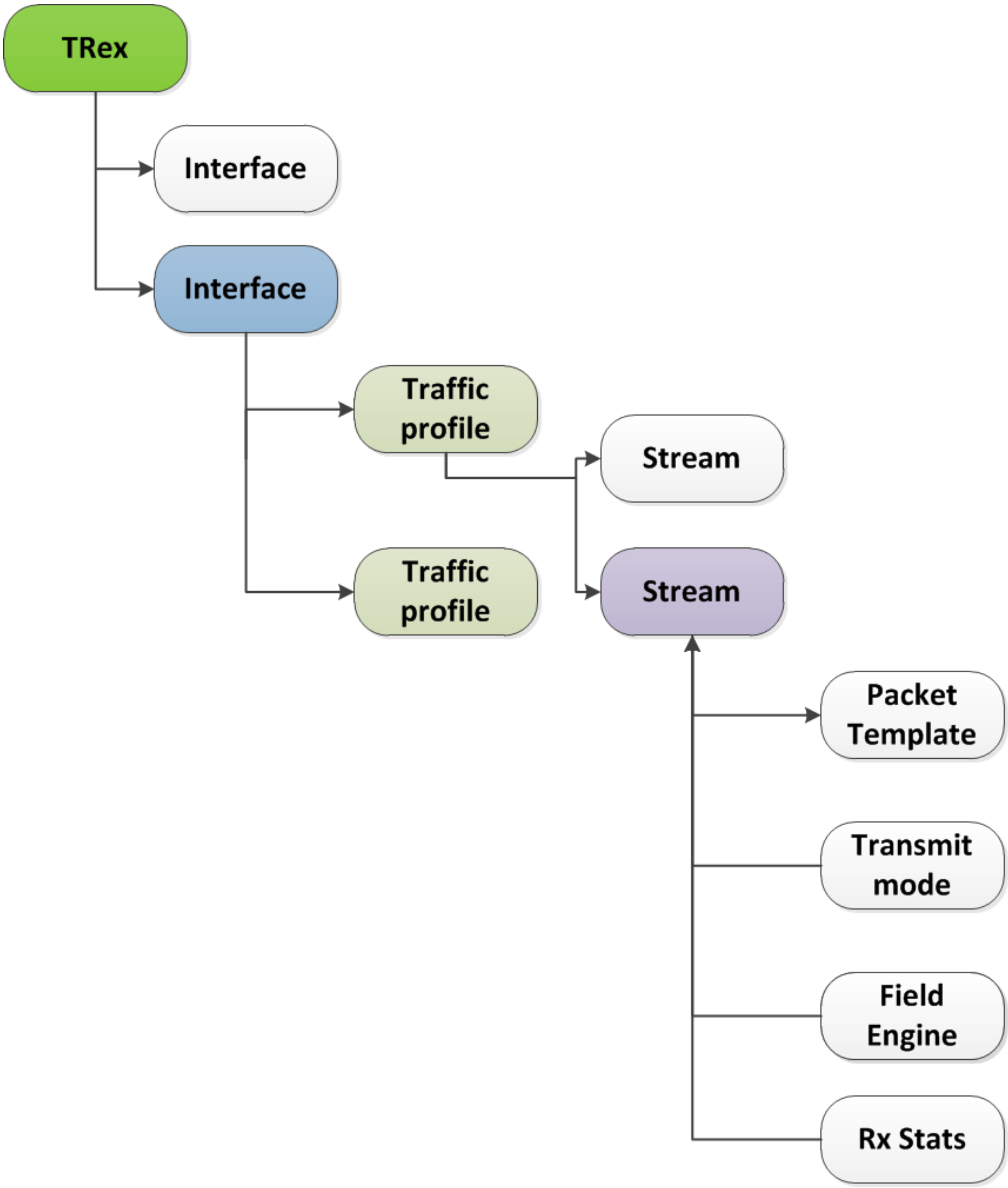
For details about the TRex RPC server, see the RPC specification.

2.3.1 RPC architecture highlights

The RPC architecture provides the following advantages:

- Fast interaction with TRex server. Loading, starting, and stopping a profile for an interface is very fast - about 2000 cycles/sec.
- Leverages Python/Scapy for building a packet/Field Engine.
- HLTAPI compiler complexity is handled in Python.

2.4 TRex Objects



- **TRex**: Each TRex instance supports numerous interfaces.
- **Interface**: Each interface supports one or more traffic profiles.
- **Traffic profile**: Each traffic profile supports one or more streams.
- **Stream**: Each stream includes:
  - **Packet**: Packet template up to 9 KB
  - **Field Engine**: Determines field to change and whether to change packet size
  - **Mode**: Specifies how to send packets: Continuous/Burst/Multi-burst

- **Rx Stats**: Statistics to collect for each stream
- **Rate**: Rate (packets per second or bandwidth)
- **Action**: Specifies stream to follow when the current stream is complete (valid for Continuous or Burst modes)

2.5 Stateful vs Stateless

TRex Stateless support enables basic L2/L3 testing, relevant mostly for a switch or router. In Stateless mode it is possible to define a stream with a **one** packet template, define a program to change any fields in the packet, and run the stream in one of the following modes:

- Continuous
- Burst
- Multi-burst

Stateless mode does **not** support learning NAT translation, as there is no context of flow/client/server.

- In Stateful mode, the basic building block is a flow/application (composed of many packets).
- Stateless mode is much more flexible, enabling you to define any type of packet, and build a simple program.

Table 2.2: Features: Stateful vs Stateless

Feature	Stateful	Stateless
Per flow state	Yes	No
NAT	Yes	No
Tunnel	Some are supported	Yes
L7 App emulation	Yes	No
Any type of packet	No	Yes
Latency Jitter	Global/Per flow	Per Stream

2.5.1 Using Stateless mode to mimic Stateful mode

Stateless mode can mimic some, but not all functionality of Stateful mode. For example, you can load a PCAP with the number of packets as a link of streams:
a→b→c→d→ back to a
You can then create a program for each stream to change:
src_ip=10.0.0.1-10.0.0.254
This creates traffic similar to that of Stateful mode, but with a completely different basis.

If you are confused you probably need Stateless. :-)

2.6 TRex package folders

Location	Description
/	t-rex-64/dpdk_set_ports/stl-sim

Location	Description
/stl	Stateless native (py) profiles
/stl/hlt	Stateless HLT profiles
/ko	Kernel modules for DPDK
/external_libs	Python external libs used by server/clients
/exp	Golden PCAP file for unit-tests
/cfg	Examples of config files
/cap2	Stateful profiles
/avl	Stateful profiles - SFR profile
/automation	Python client/server code for both Stateful and Stateless
/automation/regression	Regression for Stateless and Stateful
/automation/config	Regression setups config files
/automation/trex_control_plane/interactive/trex	Stateless lib and Console
/automation/trex_control_plane/interactive/trex/stl	Stateless lib
/automation/trex_control_plane/interactive/trex/examples/stl	Stateless examples

2.7 Getting Started Tutorials

The tutorials in this section demonstrate basic TRex **stateless** use cases. Examples include common and moderately advanced TRex concepts.

2.7.1 Tutorial: Prepare TRex configuration file

Goal
Define the TRex physical or virtual ports and create configuration file.

Follow this chapter first time configuration

2.7.2 Tutorial: Load TRex server, Simple IPv4 UDP

Goal
Send simple UDP packets from all ports of a TRex server.

Traffic profile
The following profile defines one stream, with an IP/UDP packet template with 10 bytes of *x*(0x78) of payload. For more examples of defining packets using Scapy, see the Scapy documentation.

File
stl/udp_1pkt_simple.py

```
from trex_stl_lib.api import *

class STLS1(object):

    def create_stream (self):

        return STLStream(
            packet =
                    STLPktBuilder(
                        pkt = Ether()/IP(src="16.0.0.1",dst="48.0.0.1")/
                                UDP(dport=12,sport=1025)/(10*'x')          ❶
                    ),
             mode = STLTXCont())                                           ❷
```

```
    def get_streams (self, direction = 0, **kwargs):       ❸
        # create 1 stream
        return [ self.create_stream() ]


# dynamic load - used for TRex console or simulator
def register():                                             ❹
    return STLS1()
```

❶ Defines the packet. In this case, the packet is IP/UDP with 10 bytes of *x*. For more information, see the Scapy documentation.

❷ Mode: Continuous. Rate: 1 PPS (default rate is 1 PPS)

❸ The `get_streams` function is mandatory.

❹ Each traffic profile module requires a `register` function.

Note
The SRC/DST MAC addresses are taken from /etc/trex_cfg.yaml. To change them, add Ether(dst="00:00:dd:dd:00:01") with the desired destination.

Start TRex as a server

Note
The TRex package includes all required packages. It is not necessary to install any Python packages (including Scapy).

```
[bash]>sudo ./t-rex-64 -i
```

- Wait until the server is up and running.
- (Optional) Use `-c` to add more cores.
- (Optional) Use `--cfg` to specify a different configuration file. The default is /etc/trex_cfg.yaml.

Connect with console
On the same machine, in a new terminal window (open a new window using `xterm`, or `ssh` again), connect to TRex using `trex-console`.

```
[bash]>trex-console                                                #❶

Connecting to RPC server on localhost:4501                  [SUCCESS]
connecting to publisher server on localhost:4500            [SUCCESS]
Acquiring ports [0, 1, 2, 3]:                               [SUCCESS]

125.69 [ms]

trex>start -f stl/udp_1pkt_simple.py -m 10mbps -a                  #❷

Removing all streams from port(s) [0, 1, 2, 3]:             [SUCCESS]
Attaching 1 streams to port(s) [0, 1, 2, 3]:                [SUCCESS]
```

```
Starting traffic on port(s) [0, 1, 2, 3]:                        [SUCCESS]

# pause  the traffic on all port
>pause -a                                                                    #3

# resume  the traffic on all port
>resume -a                                                                   #4

# stop traffic on all port
>stop -a                                                                     #5

# show dynamic statistic
>tui
```

1. Connects to the TRex server from the local machine.
2. Start the traffic on all ports at 10 mbps. Can also specify as MPPS. Example: 14 MPPS (`-m 14mpps`).
3. Pauses the traffic.
4. Resumes.
5. Stops traffic on all ports.

Note
If you have a connection **error**, open the /etc/trex_cfg.yaml file and remove keywords such as `enable_zmq_pub :true` and `zmq_pub_port :4501` from the file.

Viewing streams

To display stream data for all ports, use `streams -a`.

Streams

```
trex>streams -a
Port 0:

    ID |      packet type      |  length  |         mode         |  rate      | next stream
  -----------------------------------------------------------------------------------
    1  | Ethernet:IP:UDP:Raw |        56 |     Continuous     | 1.00 pps |       -1

Port 1:

    ID |      packet type      |  length  |         mode         |  rate      | next stream
  -----------------------------------------------------------------------------------
    1  | Ethernet:IP:UDP:Raw |        56 |     Continuous     | 1.00 pps |       -1

Port 2:

    ID |      packet type      |  length  |         mode         |  rate      | next stream
  -----------------------------------------------------------------------------------
    1  | Ethernet:IP:UDP:Raw |        56 |     Continuous     | 1.00 pps |       -1

Port 3:

    ID |      packet type      |  length  |         mode         |  rate      | next stream
  -----------------------------------------------------------------------------------
    1  | Ethernet:IP:UDP:Raw |        56 |     Continuous     | 1.00 pps |       -1
```

Viewing command help

To view help for a command, use `<command> --help`.

Viewing general statistics

To view general statistics, open a "textual user interface" with `tui`.

```
TRex >tui
Global Statistics

Connection  : localhost, Port 4501
Version     : v1.93, UUID: N/A
Cpu Util    : 0.2%
            :
Total Tx L2 : 40.01 Mb/sec
Total Tx L1 : 52.51 Mb/sec
Total Rx    : 40.01 Mb/sec
Total Pps   : 78.14 Kpkt/sec
            :
Drop Rate   : 0.00 b/sec
Queue Full  : 0 pkts

Port Statistics

   port    |         0          |         1          |
 --------------------------------------------------------
 owner      |              hhaim |              hhaim |
 state      |             ACTIVE |             ACTIVE |
 --         |                    |                    |
 Tx bps L2  |        10.00 Mbps |        10.00 Mbps |
 Tx bps L1  |        13.13 Mbps |        13.13 Mbps |
 Tx pps     |        19.54 Kpps |        19.54 Kpps |
 Line Util. |            0.13 % |            0.13 % |
 ---        |                    |                    |
 Rx bps     |        10.00 Mbps |        10.00 Mbps |
 Rx pps     |        19.54 Kpps |        19.54 Kpps |
 ----       |                    |                    |
 opackets   |           1725794 |           1725794 |
 ipackets   |           1725794 |           1725794 |
 obytes     |         110450816 |         110450816 |
 ibytes     |         110450816 |         110450816 |
 tx-bytes   |         110.45 MB |         110.45 MB |
 rx-bytes   |         110.45 MB |         110.45 MB |
 tx-pkts    |        1.73 Mpkts |        1.73 Mpkts |
 rx-pkts    |        1.73 Mpkts |        1.73 Mpkts |
 -----      |                    |                    |
 oerrors    |                 0 |                 0 |
 ierrors    |                 0 |                 0 |

 status:  /

 browse:     'q' - quit, 'g' - dashboard, '0-3' - port display
 dashboard:  'p' - pause, 'c' - clear, '-' - low 5%, '+' - up 5%,
```

Discussion

In this example TRex sends the **same** packet from all ports. If your setup is connected with loopback, you will see Tx packets from port 0 in Rx port 1 and vice versa. If you have DUT with static route, you might see all packets going to a specific port.

Static route

```
interface TenGigabitEthernet0/0/0
 mtu 9000
 ip address 1.1.9.1 255.255.255.0
!
interface TenGigabitEthernet0/1/0
 mtu 9000
 ip address 1.1.10.1 255.255.255.0
!

ip route 16.0.0.0 255.0.0.0 1.1.9.2
ip route 48.0.0.0 255.0.0.0 1.1.10.2
```

In this example all packets are routed to the `TenGigabitEthernet0/1/0` port. The following example uses the `direction` flag to change this.

File
stl/udp_1pkt_simple_bdir.py

```
 class STLS1(object):

    def create_stream (self):
        return STLStream(
            packet =
                    STLPktBuilder(
                        pkt  = Ether()/IP(src="16.0.0.1",dst="48.0.0.1")/
                                UDP(dport=12,sport=1025)/(10*'x')
                    ),
             mode = STLTXCont())

    def get_streams (self, direction = 0, **kwargs):
        # create 1 stream
        if direction==0:                                                       (1)
            src_ip="16.0.0.1"
            dst_ip="48.0.0.1"
        else:
            src_ip="48.0.0.1"
            dst_ip="16.0.0.1"

        pkt   = STLPktBuilder(
                              pkt  = Ether()/IP(src=src_ip,dst=dst_ip)/
                              UDP(dport=12,sport=1025)/(10*'x') )

        return [ STLStream( packet = pkt,mode = STLTXCont()) ]
```

(1) This use of the `direction` flag causes a different packet to be sent for each direction.

2.7.3 Tutorial: Connect from a remote server

Goal
Connect by console from remote machine to a TRex server.

Check that TRex server is operational
Ensure that the TRex server is running. If not, run TRex in interactive mode.

```
[bash]>sudo ./t-rex-64 -i
```

Connect with Console
From a remote machine, use `trex-console` to connect. Include the `-s` flag, as shown below, to specify the server.

```
[bash]>trex-console -s csi-kiwi-02   #1
```

1 TRex server is csi-kiwi-02.

The TRex client requires Python versions 2.7.x or 3.4.x. To change the Python version, set the **PYTHON** environment variable as follows:

tcsh shell

```
[tcsh]>setenv PYTHON /bin/python     #tcsh
```

bash shell

```
[bash]>extern PYTHON=/bin/mypython    #bash
```

Note
The client machine should run Python 2.7.x or 3.4.x. Cisco CEL/ADS is supported. The TRex package includes the required client archive.

2.7.4 Tutorial: Source and Destination MAC addresses

Goal
Change the source/destination MAC address.

Each TRex port has a source and destination MAC (DUT) configured in the /etc/trex_cfg.yaml configuration file. The source MAC is not necessarily the hardware MAC address configured in EEPROM. By default, the hardware-specified MAC addresses (source and destination) are used. If a source or destination MAC address is configured explicitly, that address has priority over the hardware-specified default.

Table 2.3: MAC address

Scapy	Source MAC	Destination MAC
Ether()	trex_cfg (src)	trex_cfg(dst)
Ether(src="00:bb:12:34:56:01")	00:bb:12:34:56:01	trex_cfg(dst)
Ether(dst="00:bb:12:34:56:01")	trex_cfg(src)	00:bb:12:34:56:01

File
stl/udp_1pkt_1mac_override.py

```
    def create_stream (self):

        base_pkt =  Ether(src="00:bb:12:34:56:01")/        1
                    IP(src="16.0.0.1",dst="48.0.0.1")/
                    UDP(dport=12,sport=1025)
```

1 Specifying the source interface MAC replaces the default specified in the configuration YAML file.

Important
A TRex port receives a packet only if the packet's destination MAC matches the HW Src MAC defined for that port in the `/etc/trex_cfg.yaml` configuration file. Alternatively, a port can be put into promiscuous mode, allowing the port to receive all packets on the line. The port can be configured to promiscuous mode by API or by the following command at the console: `portattr -a --prom`.

To set ports to promiscuous mode and show the port status:

```
trex>portattr -a --prom on                                                  #1
trex>stats --ps                                                             #2
Port Status

     port       |          0           |          1           |
  -------------------------------------------------------------------
driver          |    rte_ixgbe_pmd     |    rte_ixgbe_pmd     |
maximum         |       10 Gb/s        |       10 Gb/s        |
status          |         IDLE         |         IDLE         |
promiscuous     |          on          |          on          |         #3
  --            |                      |                      |
HW src mac      |  90:e2:ba:36:33:c0   |  90:e2:ba:36:33:c1   |
SW src mac      |  00:00:00:01:00:00   |  00:00:00:01:00:00   |
SW dst mac      |  00:00:00:01:00:00   |  00:00:00:01:00:00   |
  ---           |                      |                      |
PCI Address     |     0000:03:00.0     |     0000:03:00.1     |
NUMA Node       |          0           |          0           |
```

1. Configures all ports to promiscuous mode.
2. Show port status.
3. "on" indicates port promiscuous mode.

To change ports to promiscuous mode by Python API:

Python API to change ports to promiscuous mode

```
        c = STLClient(verbose_level ="error")

        c.connect()

        my_ports=[0,1]

        # prepare our ports
        c.reset(ports = my_ports)

        # port info, mac-addr info, speed
        print c.get_port_info(my_ports)                               1

        c.set_port_attr(my_ports, promiscuous = True)                 2
```

1. Get port info for all ports.
2. Change the port attribute to `promiscuous =True`.

For more information see the Python Client API.

Note
Interfaces are not set to promiscuous mode by default. Typically, after changing the port to promiscuous mode for a specific test, it is advisable to change it back to non-promiscuous mode.

2.7.5 Tutorial: Python automation

Goal

Simple automation test using Python from a local or remote machine.

Directories

Python API examples: `automation/trex_control_plane/interactive/trex/examples/stl`

Python API library: `automation/trex_control_plane/interactive/trex/stl`

The TRex console uses the Python API library to interact with the TRex server using the JSON-RPC2 protocol over ZMQ.

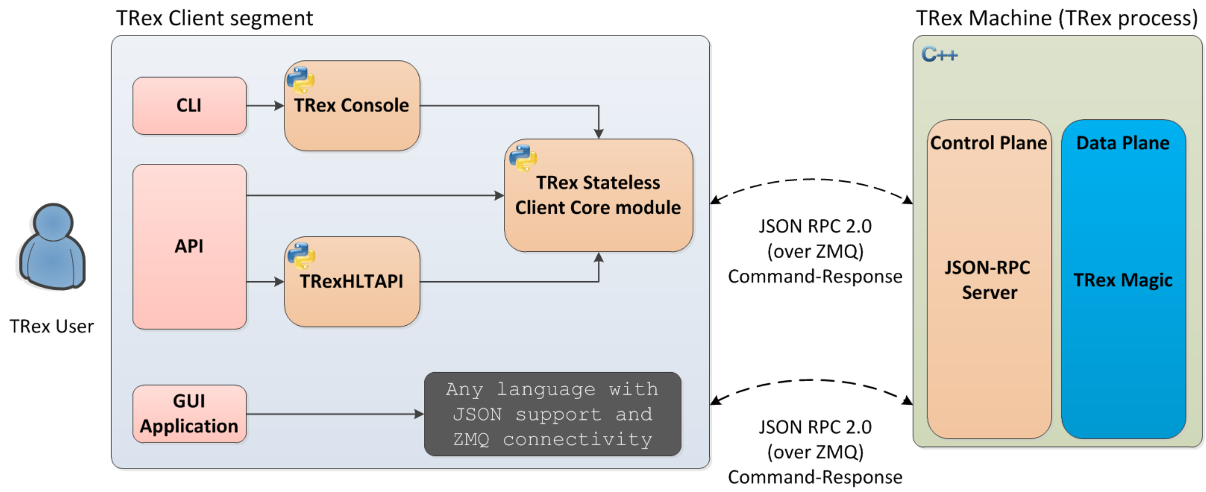


Figure 2.4: RPC server components

File

stl_bi_dir_flows.py

```
import stl_path                                                    ❶
from trex_stl_lib.api import *                                     ❷

import time
import json

# simple packet creation                                           ❸
def create_pkt (size, direction):

    ip_range = {'src': {'start': "10.0.0.1", 'end': "10.0.0.254"},
                'dst': {'start': "8.0.0.1",  'end': "8.0.0.254"}}

    if (direction == 0):
        src = ip_range['src']
        dst = ip_range['dst']
    else:
        src = ip_range['dst']
        dst = ip_range['src']

    vm = [
        # src                                                      ❹
```

```
        STLVmFlowVar(name="src",
                     min_value=src['start'],
                     max_value=src['end'],
                     size=4,op="inc"),
        STLVmWrFlowVar(fv_name="src",pkt_offset= "IP.src"),

        # dst
        STLVmFlowVar(name="dst",
                     min_value=dst['start'],
                     max_value=dst['end'],
                     size=4,op="inc"),
        STLVmWrFlowVar(fv_name="dst",pkt_offset= "IP.dst"),

        # checksum
        STLVmFixIpv4(offset = "IP")
        ]


    base = Ether()/IP()/UDP()
    pad = max(0, len(base)) * 'x'

    return STLPktBuilder(pkt = base/pad,
                         vm  = vm)


def simple_burst ():

    # create client
    c = STLClient()
                    # username/server can be changed those are the default
                    # username = common.get_current_user(),
                    # server = "localhost"
                    # STLClient(server = "my_server",username ="trex_client") for example
    passed = True

    try:
        # turn this on for some information
        #c.set_verbose("high")

        # create two streams
        s1 = STLStream(packet = create_pkt(200, 0),
                       mode = STLTXCont(pps = 100))

        # second stream with a phase of 1ms (inter stream gap)
        s2 = STLStream(packet = create_pkt(200, 1),
                       isg = 1000,
                       mode = STLTXCont(pps = 100))


        # connect to server
        c.connect()                                                           ❺

        # prepare our ports (my machine has 0 <--> 1 with static route)
        c.reset(ports = [0, 1]) #  Acquire port 0,1 for $USER                 ❻

        # add both streams to ports
        c.add_streams(s1, ports = [0])
        c.add_streams(s2, ports = [1])

        # clear the stats before injecting
        c.clear_stats()
```

```
        # choose rate and start traffic for 10 seconds on 5 mpps
        print "Running 5 Mpps on ports 0, 1 for 10 seconds..."
        c.start(ports = [0, 1], mult = "5mpps", duration = 10)                    7

        # block until done
        c.wait_on_traffic(ports = [0, 1])                                         8

        # read the stats after the test
        stats = c.get_stats()                                                     9

        print json.dumps(stats[0], indent = 4, separators=(',', ': '), sort_keys = True)
        print json.dumps(stats[1], indent = 4, separators=(',', ': '), sort_keys = True)

        lost_a = stats[0]["opackets"] - stats[1]["ipackets"]
        lost_b = stats[1]["opackets"] - stats[0]["ipackets"]

        print "\npackets lost from 0 --> 1:   {0} pkts".format(lost_a)
        print "packets lost from 1 --> 0:   {0} pkts".format(lost_b)

        if (lost_a == 0) and (lost_b == 0):
            passed = True
        else:
            passed = False

    except STLError as e:
        passed = False
        print e

    finally:
        c.disconnect()                                                            10

    if passed:
        print "\nTest has passed :-)\n"
    else:
        print "\nTest has failed :-(\n"


# run the tests
simple_burst()
```

1. Imports the stl_path. The path here is specific to this example. When configuring, provide the path to your stl_trex library.
2. Imports TRex Stateless library. When configuring, provide the path to your TRex Stateless library.
3. Creates packet per direction using Scapy.
4. See the Field Engine section for information.
5. Connects to the local TRex. Username and server can be added.
6. Acquires the ports.
7. Loads the traffic profile and start generating traffic.
8. Waits for the traffic to be finished. There is a polling function so you can test do something while waiting.
9. Get port statistics.
10. Disconnects.

See TRex Stateless Python API for details about using the Python APIs.

2.7.6 Tutorial: HLT Python API

HLT Python API is a layer on top of the native layer. It supports the standard Cisco traffic generator API. For more information, see Cisco/IXIA/Spirent documentation.
TRex supports a limited number of HLTAPI arguments. It is recommended to use the native API for simplicity and flexibility.

Supported HLT Python API classes:

- Device Control
  - connect
  - cleanup_session
  - device_info
  - info
- Interface
  - interface_config
  - interface_stats
- Traffic
  - traffic_config - not all arguments are supported
  - traffic_control
  - traffic_stats

For details, see: Appendix

File
hlt_udp_simple.py

```
import sys
import argparse
import stl_path
from trex_stl_lib.api import *                                          ❶
from trex_stl_lib.trex_stl_hltapi import *                              ❷


if __name__ == "__main__":
    parser = argparse.ArgumentParser(usage="""
    Connect to TRex and send burst of packets

    examples

     hlt_udp_simple.py -s 9000 -d 30

     hlt_udp_simple.py -s 9000 -d 30 -rate_percent 10

     hlt_udp_simple.py -s 300 -d 30 -rate_pps 5000000

     hlt_udp_simple.py -s 800 -d 30 -rate_bps 500000000 --debug

     then run the simulator on the output
       ./stl-sim -f example.py -o a.pcap  ==> a.pcap include the packet

    """,
    description="Example for TRex HLTAPI",
    epilog=" based on hhaim's stl_run_udp_simple example")
```

```
    parser.add_argument("--ip",
                        dest="ip",
                        help='Remote trex ip',
                        default="127.0.0.1",
                        type = str)

    parser.add_argument("-s", "--frame-size",
                        dest="frame_size",
                        help='L2 frame size in bytes without FCS',
                        default=60,
                        type = int,)

    parser.add_argument('-d','--duration',
                        dest='duration',
                        help='duration in second ',
                        default=10,
                        type = int,)

    parser.add_argument('--rate-pps',
                        dest='rate_pps',
                        help='speed in pps',
                        default="100")

    parser.add_argument('--src',
                        dest='src_mac',
                        help='src MAC',
                        default='00:50:56:b9:de:75')

    parser.add_argument('--dst',
                        dest='dst_mac',
                        help='dst MAC',
                        default='00:50:56:b9:34:f3')

    args = parser.parse_args()

    hltapi = CTRexHltApi()
    print 'Connecting to TRex'
    res = hltapi.connect(device = args.ip, port_list = [0, 1], reset = True, break_locks =  ↩
        True)
    check_res(res)
    ports = res['port_handle']
    if len(ports) < 2:
        error('Should have at least 2 ports for this test')
    print 'Connected, acquired ports: %s' % ports

    print 'Creating traffic'

    res = hltapi.traffic_config(mode = 'create', bidirectional = True,
                                port_handle = ports[0], port_handle2 = ports[1],
                                frame_size = args.frame_size,
                                mac_src = args.src_mac, mac_dst = args.dst_mac,
                                mac_src2 = args.dst_mac, mac_dst2 = args.src_mac,
                                l3_protocol = 'ipv4',
                                ip_src_addr = '10.0.0.1', ip_src_mode = 'increment',  ↩
                                    ip_src_count = 254,
                                ip_dst_addr = '8.0.0.1', ip_dst_mode = 'increment',  ↩
                                    ip_dst_count = 254,
                                l4_protocol = 'udp',
                                udp_dst_port = 12, udp_src_port = 1025,
                                stream_id = 1, # temporary workaround, add_stream does not  ↩
                                    return stream_id
```

```
                                  rate_pps = args.rate_pps,
                                  )
    check_res(res)

    print 'Starting traffic'
    res = hltapi.traffic_control(action = 'run', port_handle = ports[:2])
    check_res(res)
    wait_with_progress(args.duration)

    print 'Stopping traffic'
    res = hltapi.traffic_control(action = 'stop', port_handle = ports[:2])
    check_res(res)

    res = hltapi.traffic_stats(mode = 'aggregate', port_handle = ports[:2])
    check_res(res)
    print_brief_stats(res)

    res = hltapi.cleanup_session(port_handle = 'all')
    check_res(res)

    print 'Done'
```

1. Imports the native TRex API.
2. Imports the HLT API.

2.7.7 Tutorial: Simple IPv4/UDP packet simulator

Goal
Use the TRex Stateless simulator.

Demonstrates the most basic use case for the TRex simulator.

The TRex package includes a simulator tool, `stl-sim`. The simulator operates as a Python script that calls an executable. The platform requirements for the simulator tool are the same as for TRex.

The TRex simulator can:

- Test your traffic profiles before running them on TRex.
- Generate an output PCAP file.
- Simulate a number of threads.
- Convert from one type of profile to another.
- Convert any profile to JSON (API). See: TRex stream specification

Example traffic profile:

File
stl/udp_1pkt_simple.py

```
from trex_stl_lib.api import *

class STLS1(object):

    def create_stream (self):
```

```
        return STLStream(
            packet =
                    STLPktBuilder(
                        pkt = Ether()/IP(src="16.0.0.1",dst="48.0.0.1")/
                                UDP(dport=12,sport=1025)/(10*'x')          ❶
                    ),
             mode = STLTXCont())                                           ❷


    def get_streams (self, direction = 0, **kwargs):
        # create 1 stream
        return [ self.create_stream() ]


# dynamic load - used for TRex console or simulator
def register():                                                            ❸
    return STLS1()
```

❶ Defines the packet - in this case, IP/UDP with 10 bytes of *x*.

❷ Mode is Continuous, with a rate of 1 PPS. (Default rate: 1 PPS)

❸ Each traffic profile module requires a `register` function.

The following runs the traffic profile through the TRex simulator, limiting the number of packets to 10, and storing the output in a PCAP file.

```
[bash]>./stl-sim -f stl/udp_1pkt_simple.py -o b.pcap -l 10
  executing command: 'bp-sim-64-debug --pcap --sl --cores 1 --limit 5000 -f /tmp/tmpq94Tfx ↩
      -o b.pcap'

  General info:
  ------------

  image type:               debug
  I/O output:               b.pcap
  packet limit:             10
  core recording:           merge all

  Configuration info:
  -------------------

  ports:                    2
  cores:                    1

  Port Config:
  ------------

  stream count:             1
  max PPS     :             1.00  pps
  max BPS L1  :             672.00  bps
  max BPS L2  :             512.00  bps
  line util.  :             0.00  %


  Starting simulation...


  Simulation summary:
  -------------------
```

```
simulated 10 packets
written 10 packets to 'b.pcap'
```

Contents of the output PCAP file produced by the simulator in the previous step:

1	0.000000	16.0.0.1	48.0.0.1	UDP	60	Source port: 1025 Destination port: 12
2	1.000000	16.0.0.1	48.0.0.1	UDP	60	Source port: 1025 Destination port: 12
3	2.000000	16.0.0.1	48.0.0.1	UDP	60	Source port: 1025 Destination port: 12
4	3.000000	16.0.0.1	48.0.0.1	UDP	60	Source port: 1025 Destination port: 12
5	4.000000	16.0.0.1	48.0.0.1	UDP	60	Source port: 1025 Destination port: 12
6	5.000000	16.0.0.1	48.0.0.1	UDP	60	Source port: 1025 Destination port: 12
7	6.000000	16.0.0.1	48.0.0.1	UDP	60	Source port: 1025 Destination port: 12
8	7.000000	16.0.0.1	48.0.0.1	UDP	60	Source port: 1025 Destination port: 12
9	8.000000	16.0.0.1	48.0.0.1	UDP	60	Source port: 1025 Destination port: 12
10	9.000000	16.0.0.1	48.0.0.1	UDP	60	Source port: 1025 Destination port: 12
11	10.000000	16.0.0.1	48.0.0.1	UDP	60	Source port: 1025 Destination port: 12
12	11.000000	16.0.0.1	48.0.0.1	UDP	60	Source port: 1025 Destination port: 12
13	12.000000	16.0.0.1	48.0.0.1	UDP	60	Source port: 1025 Destination port: 12
14	13.000000	16.0.0.1	48.0.0.1	UDP	60	Source port: 1025 Destination port: 12
15	14.000000	16.0.0.1	48.0.0.1	UDP	60	Source port: 1025 Destination port: 12
16	15.000000	16.0.0.1	48.0.0.1	UDP	60	Source port: 1025 Destination port: 12
17	16.000000	16.0.0.1	48.0.0.1	UDP	60	Source port: 1025 Destination port: 12
18	17.000000	16.0.0.1	48.0.0.1	UDP	60	Source port: 1025 Destination port: 12

Figure 2.5: TRex simulator output stored in PCAP file

Adding `--json` displays the details of the JSON command for adding a stream:

```
[bash]>./stl-sim -f stl/udp_1pkt_simple.py --json
[
    {
        "id": 1,
        "jsonrpc": "2.0",
        "method": "add_stream",
        "params": {
            "handler": 0,
            "port_id": 0,
            "stream": {
                "action_count": 0,
                "enabled": true,
                "flags": 0,
                "isg": 0.0,
                "mode": {
                    "rate": {
                        "type": "pps",
                        "value": 1.0
                    },
                    "type": "continuous"
                },
                "next_stream_id": -1,
                "packet": {
                    "binary": "AAAAAQAAAAAAAgAACABFAAAmAA",
                    "meta": ""
                },
                "rx_stats": {
                    "enabled": false
                },
                "self_start": true,
                "vm": {
                    "instructions": [],
                    "split_by_var": ""
                }
            },
```

```
                "stream_id": 1
            }
        },
        {
            "id": 1,
            "jsonrpc": "2.0",
            "method": "start_traffic",
            "params": {
                "duration": -1,
                "force": true,
                "handler": 0,
                "mul": {
                    "op": "abs",
                    "type": "raw",
                    "value": 1.0
                },
                "port_id": 0
            }
        }
]
```


For more information about stream definition, see the RPC specification.

To convert the profile to YAML format:

```
$./stl-sim -f stl/udp_1pkt_simple.py --yaml
- stream:
    action_count: 0
    enabled: true
    flags: 0
    isg: 0.0
    mode:
      pps: 1.0
      type: continuous
    packet:
      binary: AAAAAQAAAAAAAgAACABFAAAmAAEAAEARO
      meta: ''
    rx_stats:
      enabled: false
    self_start: true
    vm:
      instructions: []
      split_by_var: ''
```

To display packet details, use the `--pkt` option (using Scapy).

```
[bash]>./stl-sim -f stl/udp_1pkt_simple.py --pkt
 =======================
 Stream 0
 =======================
###[ Ethernet ]###
  dst       = 00:00:00:01:00:00
  src       = 00:00:00:02:00:00
  type      = IPv4
###[ IP ]###
     version   = 4L
     ihl       = 5L
     tos       = 0x0
     len       = 38
     id        = 1
     flags     =
     frag      = 0L
```

```
     ttl       = 64
     proto     = udp
     chksum    = 0x3ac5
     src       = 16.0.0.1
     dst       = 48.0.0.1
     \options   \
###[ UDP ]###
        sport     = blackjack
        dport     = 12
        len       = 18
        chksum    = 0x6161
###[ Raw ]###
           load      = 'xxxxxxxxxx'
0000   00 00 00 01 00 00 00 00  00 02 00 00 08 00 45 00   ..............E.
0010   00 26 00 01 00 00 40 11  3A C5 10 00 00 01 30 00   .&....@.:.....0.
0020   00 01 04 01 00 0C 00 12  61 61 78 78 78 78 78 78   ........aaxxxxxx
0030   78 78 78 78                                        xxxx
```

To convert any profile type to native again, use the `--native` option, as shown in the following example, which includes the input file, the command to convert it to native, and the output:

Input YAML format

```
- name: udp_64B
  stream:
    self_start: True
    packet:
       binary: ↩
           AAAAAAAAAAAAAAAACABFAAAuBNIAAH8R9usQAAABMAAAAQQBBAEAEgAAAAAAAAAAAAAAAAAAAAAAAA
    mode:
      type: continuous
      pps: 100
```

Command to convert to native:

```
[bash]>./stl-sim -f my_yaml.yaml --native
```

The output:

Output Native

```
# !!! Auto-generated code !!!
from trex_stl_lib.api import *

class STLS1(object):
    def get_streams(self):
        streams = []

        packet = (Ether(src='00:de:01:0a:01:00', dst='00:50:56:80:0d:28', type=2048) /
                  IP(src='101.0.0.1', proto=17, dst='102.0.0.1', chksum=28605, len=46,  ↩
                      flags=2L, ihl=5L, id=0) /
                  UDP(dport=2001, sport=2001, len=26, chksum=1176) /
                  Raw(load='\xde\xad\xbe\xef\x00\x01\x06\x07\x08\x09\x0a\x0b\x00\x9b\xe7\ ↩
                      xdb\x82M'))
        vm = STLScVmRaw([], split_by_field = '')
        stream = STLStream(packet = CScapyTRexPktBuilder(pkt = packet, vm = vm),
                           name = 'udp_64B',
                           mac_src_override_by_pkt = 0,
                           mac_dst_override_mode = 0,
                           mode = STLTXCont(pps = 100))
        streams.append(stream)

        return streams
```

```
def register():
    return STLS1()
```

Discussion

The following are the main traffic profile formats. Native is the preferred format. There is a separation between how the traffic is defined and how to control/activate it. The API/Console/GUI can load a traffic profile and start/stop/get a statistic. Due to this separation it is possible to share traffic profiles.

Table 2.4: Traffic profile formats

Profile Type	Format	Description
Native	Python	Most flexible. Any format can be converted to native using the `stl-sim` command with the `--native` option.
HLT	Python	Uses HLT arguments.
YAML/JSON	YAML/JSON	The common denominator traffic profile. Information is shared between console, GUI, and simulator in YAML format. This format is difficult to use for defining packets; primarily for machine use. YAML can be converted to native using the `stl-sim` command with the `--native` option.

2.7.8 Tutorial: Port layer mode configuration

Goal

Configure TRex port with either IPv4 or MAC address.

TRex ports can operate in two different mutually exclusive modes:

- **Layer 2 mode** - MAC level configuration
- **Layer 3 mode** - IPv4/IPv6 configuration

Table 2.5: Port layer modes

Mode	Port configuration requirements	Notes
Layer 2 mode	When configuring a port for L2 mode, must provide the destination MAC address for the port (Legacy mode previous to v2.12 version).	-
Layer 3 mode	When configuring a port for L3, must provide both source IPv4/IPv6 address and a IPv4/IPv6 destination address.	As an intergral part of configuring L3, the client will try to ARP resolve the destination address and automatically configure the correct destination MAC, instead of sending an ARP request when starting traffic. **Note**: While in L3 mode, TRex server will generate **gratuitous ARP** packets to make sure that no ARP timeout on the DUT/router will result in a faliure of the test.

Example of configuring L2 mode Console

```
trex>service

trex>l2 --help
usage: port [-h] --port PORT --dst DST_MAC

Configures a port in L2 mode

optional arguments:
  -h, --help            show this help message and exit
  --port PORT, -p PORT  source port for the action
  --dst DST_MAC         Configure destination MAC address


trex(service)>l2 -p 0 --dst 6A:A7:B5:3A:4E:FF

Setting port 0 in L2 mode:                                [SUCCESS]

trex>service --off
```

Example of configuring L2 mode- Python API

```
  client.set_service_mode(port = 0, enabled = True)

  client.set_l2_mode(port = 0, dst_mac = "6A:A7:B5:3A:4E:FF")

  client.set_service_mode(port = 0, enabled = False)
```

Example of configuring L3 mode- Console

```
trex>service


trex(service)>l3 --help
usage: port [-h] --port PORT --src SRC_IPV4 --dst DST_IPV4

Configures a port in L3 mode

optional arguments:
  -h, --help            show this help message and exit
  --port PORT, -p PORT  source port for the action
  --src SRC_IPV4        Configure source IPv4 address
  --dst DST_IPV4        Configure destination IPv4 address

trex(service)>l3 -p 0 --src 1.1.1.2 --dst 1.1.1.1

Setting port 0 in L3 mode:                                [SUCCESS]


ARP resolving address '1.1.1.1':                          [SUCCESS]

trex>service --off
```

Example of configuring L3 mode - Python API

```
client.set_service_mode(port = 0, enabled = True)

client.set_l3_mode(port = 0, src_ipv4 = '1.1.1.2', dst_ipv4 = '1.1.1.1')

client.set_service_mode(port = 0, enabled = False)
```

2.8 Port service mode

In *normal operation mode*, to preserve high speed processing of packets, TRex ignores most of the Rx traffic, with the exception of counting/statistic and handling latency flows.

The **modes**:

1. **On** : All the packets are forward to rx to be processed by Client or capture
2. **Off** - Only latency packets are forward. before v2.66 non TCP UDP were forward to rx in software mode after v2.66 (including) only in filter mode those packets are forward for more flexibility
3. **Filter** [bgp,no_tcp_udp] - In this case specific packets are forward to rx for example BGP packets for BIRD. It is relevant only for software mode `--software`` Using filter mode you would be able to run TCP/UDP traffic in high rate while keeping the routing protocols function. The performance would be as good as "off" , the only penalty comes from the software mode that requires all packets to be processed in the rx side.

Note
Filter mode is new from version v2.66

The following illustrates how Rx packets are handled. Only a portion are forwarded to the Rx handling module and none are forwarded back to the Python client.

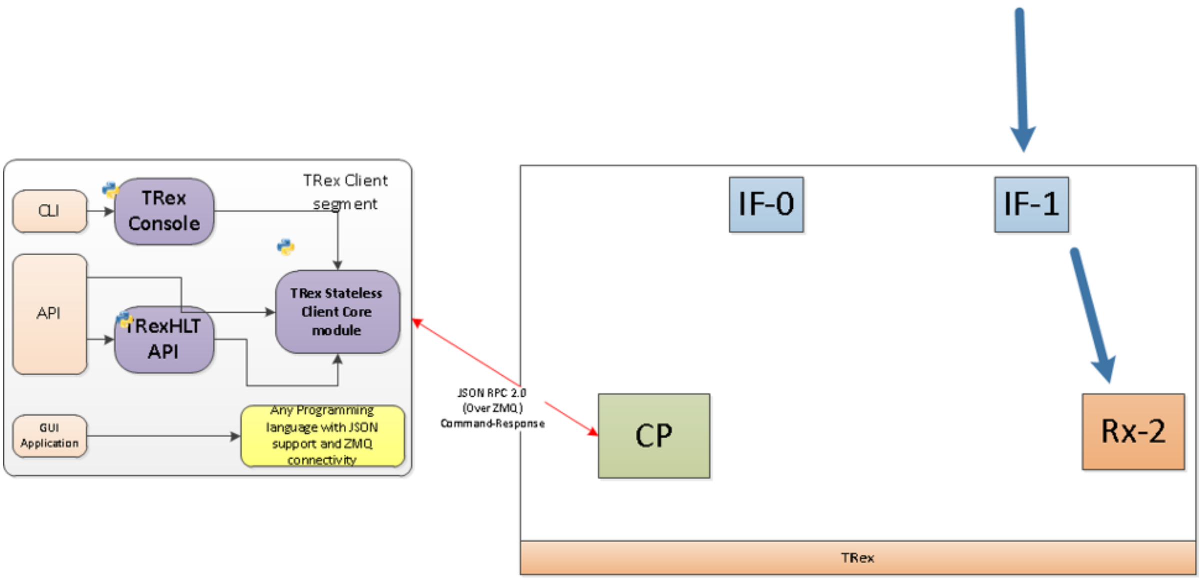


Figure 2.6: Port Under Normal Mode

In **service mode**, a port responds to ping and ARP requests, and also enables forwarding packets to the Python control plane for applying full duplex protocols (DCHP, IPv6 neighboring, and so on).

The following illustrates how packets can be forwarded back to the Python client.

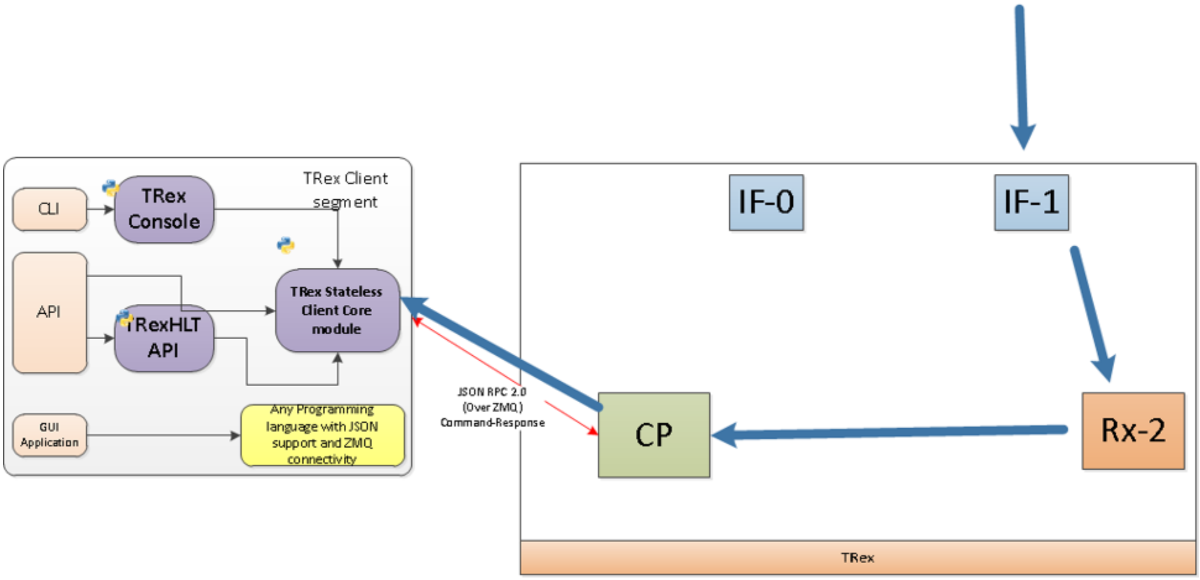


Figure 2.7: Port Under Service Mode

Service mode can be useful when writing Python plugins for emulation (example: IPV6 ND/DHCP) to prepare the setup. Then you can move to normal mode for high speed testing.

Example of switching between Service and Normal modes

```
trex(service)>service --help
usage: service [-h] [--port PORTS [PORTS ...] | -a] [--off]

Configures port for service mode. In service mode ports will reply to ARP,
PING and etc.

optional arguments:
  -h, --help            show this help message and exit
  --port PORTS [PORTS ...], -p PORTS [PORTS ...]
                        A list of ports on which to apply the command
  -a                    Set this flag to apply the command on all available
                        ports
  --off                 Deactivates services on port(s)

 --all                  Allow filtering bgp and no tcp udp

 --bgp                  filter mode with bgp packets forward to rx

 --no-tcp-udp           filter mode with no_tcp_udp packets forward to rx


trex>service

Enabling service mode on port(s) [0, 1]:                    [SUCCESS]

trex(service)>service --off

Disabling service mode on port(s) [0, 1]:                   [SUCCESS]
```

Example Of switching between Service and Normal modes: API

```
  client.set_service_mode(ports = [0, 1], enabled = True)
```

```
client.set_service_mode(ports = [0, 1], enabled = False)
```

2.8.1 ARP / ICMP response

Important
Only when in service mode, ports will reply to ICMP echo requests and ARP requests.

2.9 Packet capturing

Important
This section is relevant only for service mode.

In service mode, TRex provides a few ways to examine and manipulate both Rx and Tx packets.

Packet capturing is implemented by allocating one more more fast, in-memory queues on the server side that copy-and-store the packet buffer.

Each queue can be defined with the following attributes:

- Storage
- Which ports on either Tx/Rx it should capture
- Whether it should be *cyclic* or *fixed*

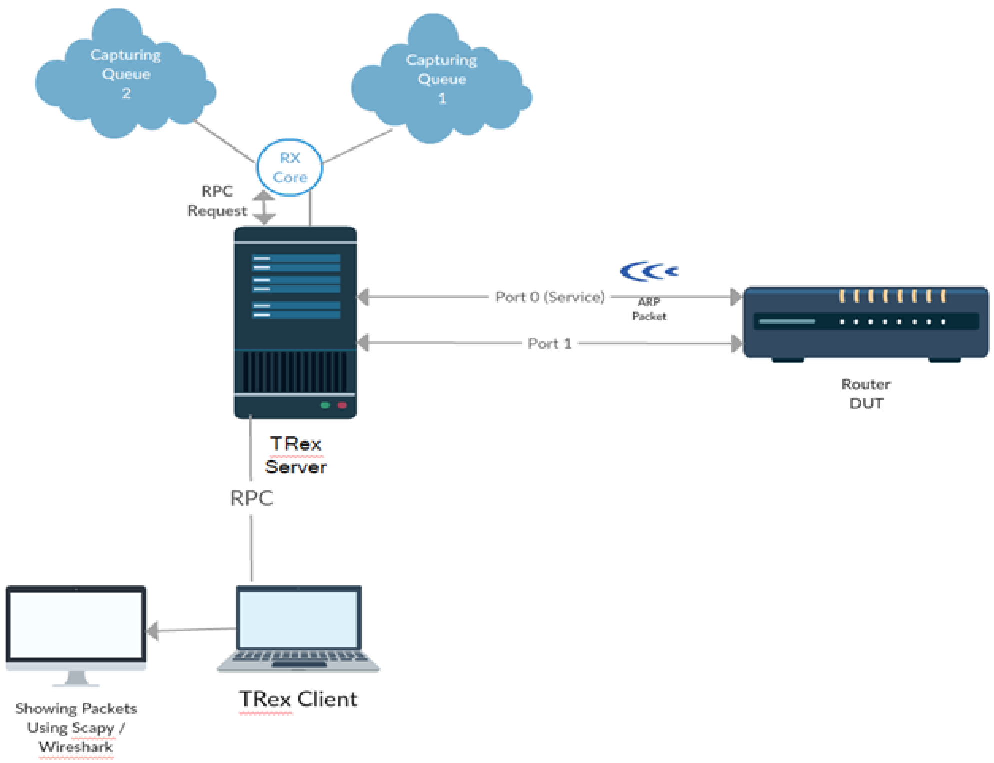


Figure 2.8: Packet Captruing Architecture

The above architecture implies that we can *capture* at high speed for a short amount of time.

For example, a queue of 1 million packets can be allocated as a cyclic queue and be active with a rate of couple of MPPS. This effectively provides a sampling of the last 1 million packets seen by the server with the given filters.

2.9.1 BPF Filtering

Before demonstrating how to use **Packet Capturing**, it is helpful to review how **filtering** is done.

Each packet capture is assigned a filter (by default, a filter that matches any packet). Any filter that follows the syntax rules of **The Berekely Packet Filter (BPF)** can be assigned.

BPF filters are widely used by the Linux kernel, TCP dump and others. Basically any *tcpdump* filtering tutorial can be used to define a filter for TRex.

Some simple examples using **BPF**:

- All *ARP* or *ICMP* packets:

```
'arp or icmp'
```

- All *UDP* packets with destination port 53:

```
'udp and dst 53'
```

- All packets *VLAN* tagged *200* and *TCP SYN*:

```
'vlan 200 and tcp[tcpflags] == tcp-syn'
```

For more examples, refer to BPF and tcpdump examples available online.

2.9.2 BPFJIT

TRex server uses **BPF JIT**, a compiled version of BPF to native code, to allow very fast filtering. So *high speed filtering* is very much possible in TRex.

Before

The following is a snapshot of a XL710 with Intel® Xeon® CPU E5-2667 v3 @ 3.20GHz handling 15.72 mpps **before** applying a BPF filter.

```
Global Statistics

connection   : localhost, Port 4501                  total_tx_L2  : 8.18 Gb/sec
version      : v2.28                                 total_tx_L1  : 10.73 Gb/sec
cpu_util.    : 3.31% @ 14 cores (7 per port)         total_rx     : 8.18 Gb/sec
rx_cpu_util. : 82.0% / 15.72 Mpkt/sec ❶             total_pps    : 15.97 Mpkt/sec
async_util.  : 0.19% / 1.76 KB/sec                   drop_rate    : 0.00 b/sec
                                                     queue_full   : 0 pkts
```

After

With a *non-hitting* filter to measure the effect of using the BPF filter:

```
Global Statistics

connection   : localhost, Port 4501                  total_tx_L2  : 8.21 Gb/sec
version      : v2.28                                 total_tx_L1  : 10.77 Gb/sec
cpu_util.    : 3.37% @ 14 cores (7 per port)         total_rx     : 8.21 Gb/sec
rx_cpu_util. : 86.4% / 15.63 Mpkt/sec ❶             total_pps    : 16.03 Mpkt/sec
async_util.  : 0.21% / 1.64 KB/sec                   drop_rate    : 0.00 b/sec
                                                     queue_full   : 0 pkts
```

There is almost zero impact (<5%) on CPU utilization for negative filtering.

Of course, a hitting filter will have impact but usually on a very small portion of the traffic.

2.9.3 API usage

Using the Python API is fairly simple:

Python API:

```
# move port 1 to service mode as we want to capture traffic on it
client.set_service_mode(ports = 1)

# start a capture on port 1 Rx side with a limit, a mode and a *BPF* filter for any UDP ↩
    with dst port 53
capture = client.start_capture(rx_ports = 1, limit = 100, mode = 'fixed', bpf_filter = 'udp ↩
     and dst 53')

# execute your code here

# save the packets to a file or to a list (see the Python API docs)
client.stop_capture(capture['id'], '/home/mydir/port_0_rx.pcap')

# exit service mode on port 1
client.set_service_mode(ports = 1, enabled = False)
```

2.9.4 Console usage

The console provides couple of flexible ways to handle packet capturing

- **Capture Monitoring**
- **Capture Recording**

2.9.4.1 Capture monitoring

Capture monitoring is a non-persistent method for capturing and showing packets from either Tx / Rx of one or more ports.

Monitoring has 3 modes:

- **Low Verbose** - A short line per packet will be displayed
- **High Verbose** - Full Scapy show will be displayed per packet
- **Wireshark Pipe** - Launches Wireshark with a pipe connected to the traffic being captured

The first two options display packet information **on the console**. This is ideal if a moderate amount of traffic is being monitored. However, if a large amount of traffic is being monitored, consider **Wireshark Pipe** or the **Capture Recording** method.

Example of capturing traffic using the console with verbose on

```
trex>service                                                                  ❶

Enabling service mode on port(s) [0, 1, 2, 3]:              [SUCCESS]

trex(service)>capture monitor start --rx 3 -v                                 ❷

Starting stdout capture monitor - verbose: 'high'           [SUCCESS]


*** use 'capture monitor stop' to abort capturing... ***

trex(service)>arp -p 3                                                        ❸

Resolving destination on port(s) [3]:                       [SUCCESS]

Port 3 - Recieved ARP reply from: 1.1.1.1, hw: 90:e2:ba:ae:88:b8              ❹
38.14 [ms]

trex(service)>

#1 Port: 3 -- Rx

    Type: ARP, Size: 60 B, TS: 16.98 [sec]

    ###[ Ethernet ]###
      dst       = 90:e2:ba:af:13:89
      src       = 90:e2:ba:ae:88:b8
      type      = 0x806
    ###[ ARP ]###
         hwtype    = 0x1
         ptype     = 0x800
         hwlen     = 6
         plen      = 4
         op        = is-at                                                    ❺
         hwsrc     = 90:e2:ba:ae:88:b8
         psrc      = 1.1.1.1
```

```
         hwdst      = 90:e2:ba:af:13:89
         pdst       = 4.4.4.4
    ###[ Padding ]###
            load      = '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\ ↩
                x00\x00'


trex(service)>
```

1, 1, 1 Move to **service mode** to allow capturing.

2 Activate a capture monitor on port **3** Rx side with **verbose** on.

3 Send an ARP request on port **3**.

4 The console shows the returning packet.

5 **is-at** ARP response was captured.

Example of capturing traffic using Wireshark pipe

```
trex(service)>capture monitor start --rx 3 -f udp -p                                  1

Starting pipe capture monitor                               [SUCCESS]


Trying to locate Wireshark                                  [SUCCESS]


Checking permissions on '/usr/bin/dumpcap'                  [SUCCESS]


Launching '/usr/bin/wireshark -k -i /tmp/tmputa4jf3c'       [SUCCESS]                 2


Waiting for Wireshark pipe connection                       [SUCCESS]                 3


*** Capture monitoring started ***                                                    4

trex(service)>arp                                                                     5

Resolving destination on port(s) [0, 1, 2, 3]:              [SUCCESS]

Port 0 - Recieved ARP reply from: 4.4.4.4, hw: 90:e2:ba:af:13:89
Port 1 - Recieved ARP reply from: 3.3.3.3, hw: 90:e2:ba:af:13:88
Port 2 - Recieved ARP reply from: 2.2.2.2, hw: 90:e2:ba:ae:88:b9
Port 3 - Recieved ARP reply from: 1.1.1.1, hw: 90:e2:ba:ae:88:b8
```

1 Activate a monitor using a Wireshark pipe and a UDP filter (BPF).

2 Attempts to launch Wireshark with a connection to the pipe.

3 Console is blocked until connection is established.

4 Monitor is active.

5 Sends ARP request.

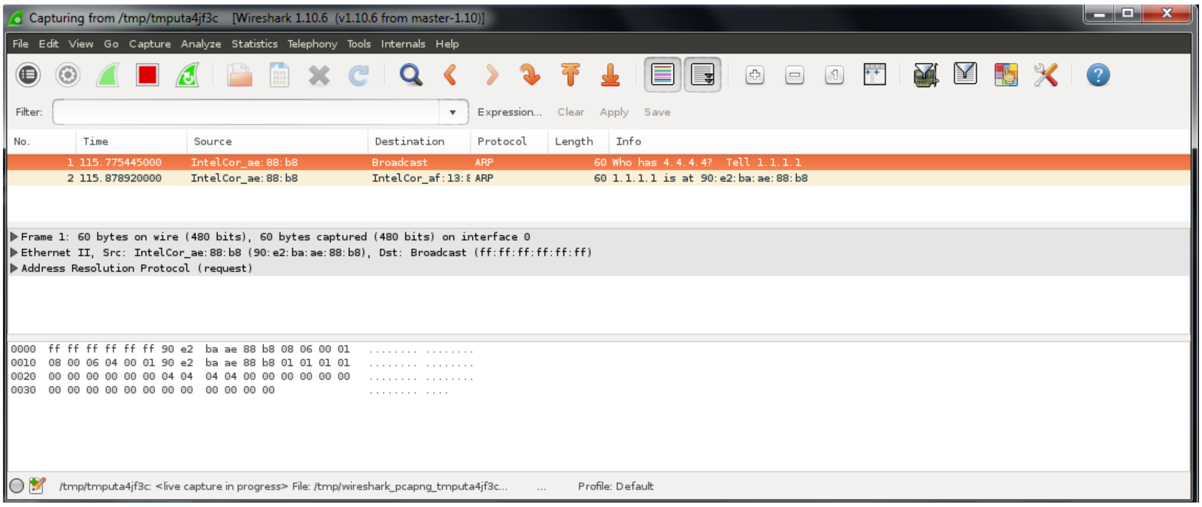


Figure 2.9: Wireshark Pipe

2.9.4.2 Capture recording

In addition to monitoring, the console allows a simple recording as well. Recording enables you to define a fixed-size queue which then can be saved to a PCAP file.

Example of capturing a traffic to a fixed size queue

```
trex(service)>capture record start --rx 3 --limit 200                        ❶

Starting packet capturing up to 200 packets                 [SUCCESS]

*** Capturing ID is set to '4' ***                                           ❷
*** Please call 'capture record stop --id 4 -o <out.pcap>' when done ***

trex(service)>capture                                                        ❸

Active Recorders

      ID          |      Status        |      Packets       |       Bytes        |     TX Ports       | ↩
           RX Ports
 -----------------------------------------------------------------------------------------------------

       4          |      ACTIVE        |      [0/200]       |        0 B         |         -          | ↩
                3



trex(service)>start -f stl/imix.py -m 1kpps -p 0 --force                     ❹

Removing all streams from port(s) [0]:                      [SUCCESS]


Attaching 3 streams to port(s) [0]:                         [SUCCESS]


Starting traffic on port(s) [0]:                            [SUCCESS]

20.42 [ms]

trex(service)>capture                                                        ❺
```

```
Active Recorders

      ID         |      Status       |      Packets      |       Bytes       |      TX Ports      |  ↩
            RX Ports
 ------------------------------------------------------------------------------------------------------

       4         |      ACTIVE       |     [200/200]     |     74.62 KB      |         -          |  ↩
                 3


trex(service)>capture record stop --id 4 -o /tmp/rx_3.pcap                       ❻

Stopping packet capture 4                                      [SUCCESS]


Writing 200 packets to '/tmp/rx_3.pcap'                        [SUCCESS]


Removing PCAP capture 4 from server                            [SUCCESS]

trex(service)>
```

❶ Start a packet record on port **3** Rx side with a limit of **200** packets.

❷ A new capture is created with an ID **4**.

❸ Show the capture status - currently empty.

❹ Start traffic on port **0**, which is connected to port **3**.

❺ Show the capture status - full.

❻ Save 200 packets to an output file: **/tmp/rx_3.pcap**

2.9.5 Using capture as a counter

Another use of packet capturing is *counting*. Instead of fetching the packets, you can simply count packets that hit the BPF filter.

For example, to count any packet that is *UDP* with source port of *5000*, you can simply attach an *empty* capture with the correct BPF filter and examine the *matched* field:

```
trex(service)>capture record start --rx 3 --limit 0 -f udp and src 5000

Starting packet capturing up to 0 packets                      [SUCCESS]

*** Capturing ID is set to '14' ***
*** Please call 'capture record stop --id 14 -o <out.pcap>' when done ***

trex(service)>capture

Active Recorders

    ID      |   Status     |  Matched   |     Packets      |    Bytes     |     RX Ports      |    ↩
       BPF Filter
 ------------------------------------------------------------------------------------------------------

    14      |   ACTIVE     |     0      |      [0/0]       |     0 B      |         3         | udp ↩
         and src 5000

trex(service)>
```

The **Matched** field indicates how many packets matched the filter.

2.9.6 Using capture port for faster packet capture / packet injection

Using the packet capture mechanism to inspect packets on TRex client python side typically yields to a transfer rate of about 5000 packets/sec due to the polling nature and the overhead of the RPC protocol.

In order to do faster packet transfer between TRex server and TRex client, as well as improving the injection of packets, the capture port feature can be used. It typically runs about 4x faster than the TRex capture.

This feature enables oneTRex server to connect to an already-opened ZeroMQ socket that will solely used to send / receive raw packets for a given TRex port. Pushing some data on this socket will translate to a new packet sent on the TRex port, while the packets received on the port will sent over the ZeroMQ socket as well.

Optionally, a BPF filter can be also specified to restrict the packets sent from TRex server to the TRex client side.

Here is a usage example that will first send one packet on TRex port 0, and then blocks until one IP packet is received back:

```
import zmq
# Bind our ZeroMQ socket so that the TRex server can connect to it
capture_port = "ipc:///tmp/trex_capture_port"
zmq_context = zmq.Context()
zmq_socket = zmq_context.socket(zmq.PAIR)
zmq_socket.bind(capture_port)

# move port 0 to service mode as we want to start capture port on it
client.set_service_mode(ports = 0)

# start a trex capture port on port 0 with *BPF* filter for any IP packets
client.start_capture_port(port = 0, endpoint = capture_port, bpf_filter = 'ip')

# Send one packet (using scapy here)
zmq_socket.send(bytes(Ether()/IP()/IP()/UDP()))

# Wait until we get an IP packet on TRex port 0 and display it parsed using Scapy
received_packet = zmq_socket.recv()
Ether(received_packet).show2()

# Stop capture port
client.stop_capture_port(port = 0)

# exit service mode on port 0
client.set_service_mode(ports = 0, enabled = False)
```

2.9.7 Video tutorials

This tutorial demonstrates the new packet capture ability.

2.10 Neighboring protocols

To preserve high speed traffic generation, TRex handles neighboring protocols in the pre-test phase.

A test that requires running a neighboring protocol should first move to *service mode*, execute the required steps in Python, switch back to *normal mode*, and start the actual test.

2.10.1 ARP

A basic neighboring protocol that is provided as part of TRex is ARP.

Example setup:

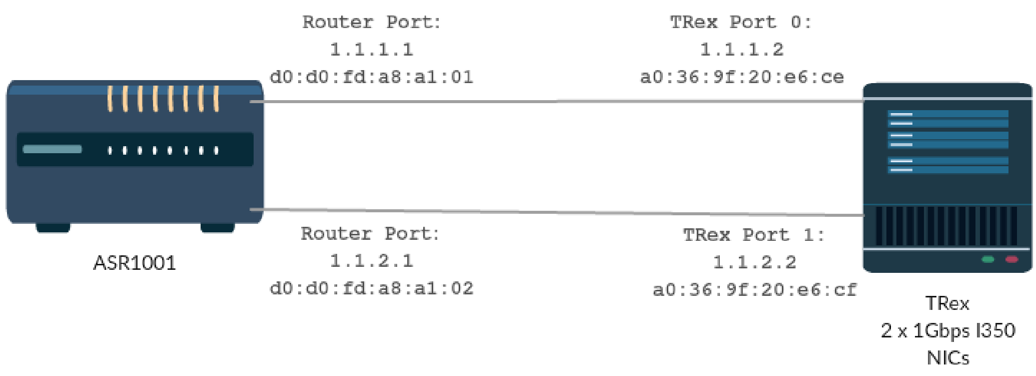


Figure 2.10: Router ARP

```
trex>service                                                         #❶

Enabling service mode on port(s) [0, 1]:                       [SUCCESS]

trex(service)>portattr  --port 0

             port       |          0            |
        -------------------------------------------
        driver          |    rte_ixgbe_pmd      |
        description     |  82599EB 10-Gigabit   |
        link status     |          UP           |
        link speed      |       10 Gb/s         |
        port status     |         IDLE          |
        promiscuous     |         off           |
        flow ctrl       |         none          |
        --              |                       |
        src IPv4        |          -            |
        src MAC         |  00:00:00:01:00:00    |
        ---             |                       |
        Destination     |  00:00:00:01:00:00    |
        ARP Resolution  |          -            |
        ----            |                       |
        PCI Address     |     0000:03:00.0      |
        NUMA Node       |          0            |
        -----           |                       |
        RX Filter Mode  |    hardware match     |
        RX Queueing     |         off           |
        RX sniffer      |         off           |
        Grat ARP        |         off           |


trex(service)>l3 -p -s 1.1.1.1 -d 1.1.1.2                            #❷

trex(service)>arp -p 0 1                                             #❸

Resolving destination on port(s) [0, 1]:                       [SUCCESS]


Port 0 - Recieved ARP reply from: 1.1.1.1, hw: d0:d0:fd:a8:a1:01
Port 1 - Recieved ARP reply from: 1.1.2.1, hw: d0:d0:fd:a8:a1:02
```

```
trex(service)>service --off                                      #4
```

1. Enable service mode.
2. Set IPv4/default gateway. It will resolve the ARP.
3. Repeat ARP resolution.
4. Exit from service mode.

To revert back to MAC address mode (without ARP resolution):

Disable L3 mode

```
trex>l2 -p 0 --dst 00:00:00:01:00:00       #1

trex>portattr  --port 0

             port       |          0           |
        ------------------------------------------
        driver          |    rte_ixgbe_pmd     |
        description     |  82599EB 10-Gigabit  |
        link status     |          UP          |
        link speed      |       10 Gb/s        |
        port status     |         IDLE         |
        promiscuous     |         off          |
        flow ctrl       |         none         |
        --              |                      |
        src IPv4        |          -           |
        src MAC         |  00:00:00:01:00:00   |
        ---             |                      |
        Destination     |  00:00:00:01:00:00   |
        ARP Resolution  |          -           |
        ----            |                      |
        PCI Address     |     0000:03:00.0     |
        NUMA Node       |          0           |
        -----           |                      |
        RX Filter Mode  |    hardware match    |
        RX Queueing     |         off          |
        RX sniffer      |         off          |
        Grat ARP        |         off          |
```

1. Disable service mode.

Python API:

```
client.set_service_mode(ports = [0, 1], enabled = True)                    1

# configure port 0, 1 to Layer 3 mode
client.set_l3_mode(port = 0, src_ipv4 = '1.1.1.2', dst_ipv4 = '1.1.1.2')  2
client.set_l3_mode(port = 1, src_ipv4 = '1.1.2.2', dst_ipv4 = '1.1.2.1')

# ARP resolve ports 0, 1
c.resolve(ports = [0, 1])

client.set_service_mode(ports = [0, 1], enabled = False)                   3
```

1. Enable service mode.
2. Configure IPv4 and default gateway.
3. Disable service mode.

2.10.2 ICMP

Another basic protocol provided with TRex is ICMP. It is possible, under service mode, to ping the DUT or even a TRex port from the console / API.

TRex Console

```
trex(service)>ping --help
usage: ping [-h] --port PORT -d PING_IPV4 [-s PKT_SIZE] [-n COUNT]

pings the server / specific IP

optional arguments:
  -h, --help            show this help message and exit
  --port PORT, -p PORT  source port for the action
  -d PING_IPV4          which IPv4 to ping
  -s PKT_SIZE           packet size to use
  -n COUNT, --count COUNT
                        How many times to ping [default is 5]

trex(service)>ping -p 0 -d 1.1.2.2

Pinging 1.1.2.2 from port 0 with 64 bytes of data:
Reply from 1.1.2.2: bytes=64, time=27.72ms, TTL=127
Reply from 1.1.2.2: bytes=64, time=1.40ms, TTL=127
Reply from 1.1.2.2: bytes=64, time=1.31ms, TTL=127
Reply from 1.1.2.2: bytes=64, time=1.78ms, TTL=127
Reply from 1.1.2.2: bytes=64, time=1.95ms, TTL=127
```

Python API

```
# move to service mode
client.set_service_mode(ports = ports, enabled = True)

# configure port 0, 1 to Layer 3 mode
client.set_l3_mode(port = 0, src_ipv4 = '1.1.1.2', dst_ipv4 = '1.1.1.1')
client.set_l3_mode(port = 1, src_ipv4 = '1.1.2.2', dst_ipv4 = '1.1.2.1')

# ping port 1 from port 0 through the router
client.ping_ip(src_port = 0, dst_ipv4 = '1.1.2.2', pkt_size = 64)        ❶

# disable service mode
client.set_service_mode(enabled = False)
```

❶ Check connectivity.

2.10.3 IPv6 ND client

Current: TRex supports scanning of network for IPv6-enabled neighbors, and pinging nearby devices from the console.
Plans for future phase: Add support at the CPP server.
The advantage of those methods is that they can be easily extended to simulate a large number of clients in automation.

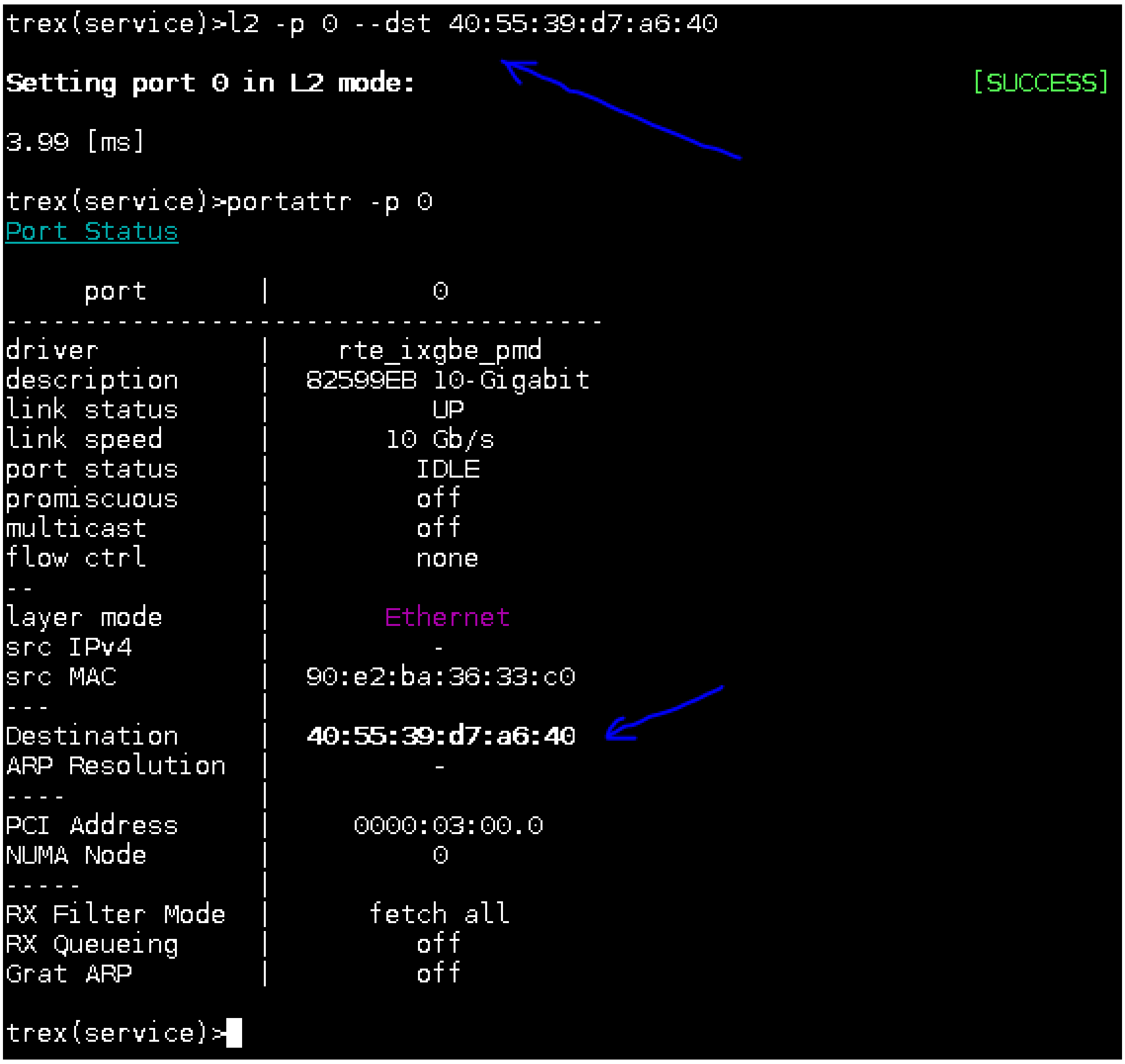
Scanning example:

```
trex(service)>scan6 -p 0

Scanning network for IPv6 nodes on port(s) [0]:                  [SUCCESS]

Port 0 - IPv6 search result:
 Device   |          MAC          |          IPv6 address
------------------------------------------------------------
 Router   |  40:55:39:d7:a6:40  | fe80::4255:39ff:fed7:a640

5.01 [sec]

trex(service)>_
```

Ping example:

```
trex(service)>ping -p 0 -d fe80::4255:39ff:fed7:a640

Pinging fe80::4255:39ff:fed7:a640 from port 0 with 64 bytes of data:
Reply from fe80::4255:39ff:fed7:a640: bytes=64, time=2.27ms, hlim=64
Reply from fe80::4255:39ff:fed7:a640: bytes=64, time=1.49ms, hlim=64
Reply from fe80::4255:39ff:fed7:a640: bytes=64, time=2.08ms, hlim=64
Reply from fe80::4255:39ff:fed7:a640: bytes=64, time=1.84ms, hlim=64
Reply from fe80::4255:39ff:fed7:a640: bytes=64, time=0.74ms, hlim=64
trex(service)>
```

Those utilities (available from API as well) can help user to configure next hop. From the console, one could set "l2" destination MAC taken from the scan6 result:



```
trex(service)>l2 -p 0 --dst 40:55:39:d7:a6:40

Setting port 0 in L2 mode:                                          [SUCCESS]

3.99 [ms]

trex(service)>portattr -p 0
Port Status

     port       |          0
--------------------------------------
driver          |    rte_ixgbe_pmd
description     |  82599EB 10-Gigabit
link status     |          UP
link speed      |       10 Gb/s
port status     |         IDLE
promiscuous     |         off
multicast       |         off
flow ctrl       |         none
--              |
layer mode      |       Ethernet
src IPv4        |          -
src MAC         |  90:e2:ba:36:33:c0
---             |
Destination     |  40:55:39:d7:a6:40
ARP Resolution  |          -
----            |
PCI Address     |     0000:03:00.0
NUMA Node       |          0
-----           |
RX Filter Mode  |      fetch all
RX Queueing     |         off
Grat ARP        |         off

trex(service)>
```

For setting own IPv6, we use local address as described in RFC 3513.
For scanning of network, we ping the multicast address ff02::1 and establish connection via NS/ND conversations.

Additional links on scanning network:

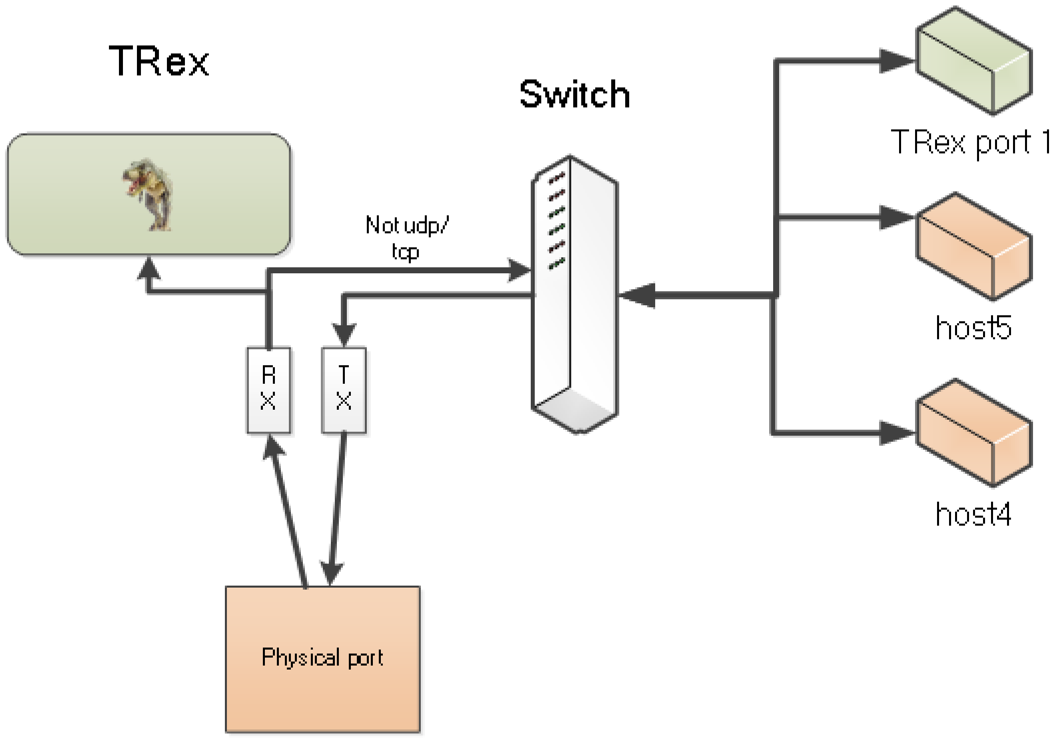
- RFC draft of scanning
- Scanning of network in Ubuntu: scan6

Example of using IPv6 methods in automation:

- stl_ipv6_tools.py

2.10.4 Linux network namespace

From version v2.50 it is possible to attach a few Linux Network Namespaces to TRex physical interface. Each host in a namespace a.k.a Node simulating a real client. Using this method, it is possible to simulate a network with many Linux network devices, each with its own separate network stack (e.g. different ipv4/ipv6/QinQ/Dot1Q/routing/arp tables etc.)



The above figure shows two hosts, host4 and host5 attached to a virtual switch (implemented inside TRex). TRex port itself has its own namespace with its own IPv4 and IPv6 configuration (using the old API). Each host when created has a distinct mac address as a key and network configuration (IPv4,IPv6, Dot1Q,QinQ etc). The protocols are implemented by the Linux kernel (timeout,arp table,ipv6,routing per namespace).

Let's take an example of ingress and egress packets

DUT ARP request packet comes from DUT to host4 (whois host4 -broadcast). This packet reachs the TRex Rx core (due to service mode configuration in STL and by default in ASTF). Rx core will broadcast it to all nodes (host4,host5,trex-port1) host4 will answer with unicast ARP response tat will reach the DUT (by TRex switch implementation)

IPv6 MLD/Broadcast packets generated by a node (multicast/boradcast packet) will be forward **only** to DUT. DUT broadcast/-multicast packets will be forward to all nodes.

Note

When a new node is created. It will not send gratuitous arp Multicast/Promiscuous mode should be enabled in port level else unitcast packets won't reach the nodes

---

The API capabilities are

1. Create a new node and associated it to a physical port (e.g. port 0).
2. Configure node with IPv4 and/or IPv6.
3. Remove a Node from a port.
4. Get statistics/status

One downside of this method is that it is a bit slow to add/remove new nodes. It could take ~100msec (due to kernel interaction) to create one. However, once the nodes were created TRex will be able to handle many of them without a problem as the traffic rate is not high (multicast/broadcast packets only) TRex can handle bursts of multicast (DUT→ many nodes) by splitting this operation to many small operations.

The scale is limited by the kernel memory and creation time.

Beacuse it is a very slow operation the API is a bit different than what we have today.

differences

1. **reset** API will **not** remove all the nodes
2. Once client disconnected/connected there is no sync with the nodes information
3. "Service mode" shoule be enabled in STL (for ASTF there is no need)
4. Need to enable multicast/promiscuous mode
5. Add `stack:linux_based` in trex_cfg.yaml see Linux Stack

linux_based stack configuration

```
  - version: 2
    interfaces: ['82:00.0', '82:00.1']
    stack: linux_based
...
```

full API could be found namespace API

2.10.4.1 Types of nodes

All nodes (node==veth) are identified by a unique MAC address, however there are some differences. There are 3 types of nodes: **Normal**, **Shared Namespace** and **Bird nodes**.

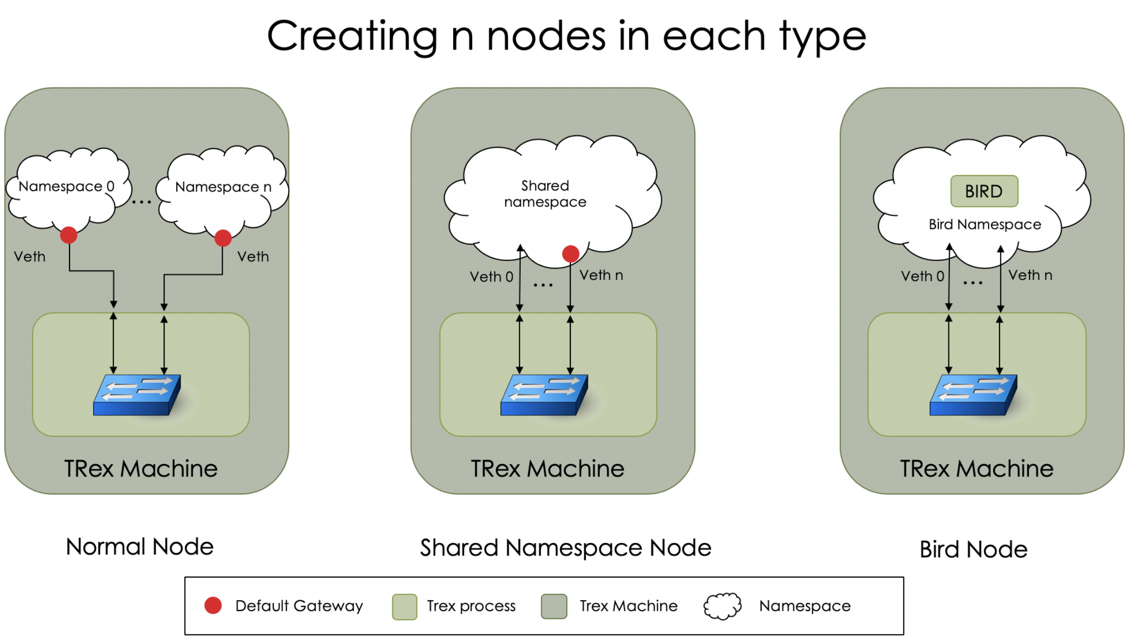


Figure 2.11: Three types of nodes

2.10.4.2 Normal Node

These are the default type of nodes, creating 1 namespace and 1 veth pair. Adding another new node will create another namespace with 1 veth pair. These nodes are configured by ipv4 and a default gateway (all traffic are forward to the default single veth)

Example of usage

```
cmds = NSCmds()
MAC = "00:01:02:03:04:05"
cmds.add_node(MAC)                                                    ❶
c.set_namespace(0, method='set_ipv4', mac=MAC, ipv4="1.1.1.3", dg="1.1.1.2")    ❷
...
```

❶ defualt node is normal node, specifying only mac address

❷ configuring ipv4 with ip address and a default gateway

More examples can be found here

2.10.4.3 Shared Namespace Node

These are nodes(veth) attached to a pre-created namespace. This way you can create multiple veths attached to the same namespace. These nodes are configured by ipv4 and a subnet mask and also can set default gateway for the whole namespace (set to one of the veth) The configuration of Dot1Q/QinQ is per veth and not per namespace

Example of usage

```
cmds = NSCmds()
cmds.add_node(mac = "00:01:02:03:04:05", shared_ns = "my-ns")        ❶
cmds.add_node(mac = "00:01:02:03:04:06", shared_ns = "my-ns")        ❷
c.set_namespace(0, method='set_ipv4', mac="00:01:02:03:04:05",
                                       ipv4 = "1.1.1.3",
                                       subnet = 24,
                                       shared_ns = True)             ❸

c.set_namespace(0, method='set_ipv4', mac="00:01:02:03:04:06",
                                       ipv4 = "1.1.2.3",
                                       subnet = 24,
                                       shared_ns = True)
...
```

❶ "my-ns" is the name of the pre-created namespace

❷ creating a second veth pair in the same namespace

❸ must specify shared_ns parameter

2.10.4.4 Bird Node

Bird node is a specific case of shared namespace node. Bird namespace is created once running TRex with "--bird-server" flag, in that way they are similar to shared namespace.

To learn more on the usage of bird with TRex see:

- Bird Integration
- Stateless Python API - Bird Node

Table 2.6: (continued)

2.10.4.5 Summary of node types differences

Table 2.6: Types of nodes differences

Node Type	Node : veth ratio	Namespace creation	Subnet mask	Default Gateway
Normal	1 : 1	Created on each node	Not in use, using default gateway	1 for the whole node
Shared Namespace	1 : Many	Created by "add_shared_ns" command, name determined by TRex	Configured at "set_ipv4/6"	1 for the whole namespace
Bird	1 : Many	Pre-created , name determined by TRex	Configured at "set_ipv4/6"	Not in use, using subnet mask

Note
Notice namespaces cannot communicate with each other since they are all passing packets through TRex ports to the DUT.

2.10.4.6 Add one node using batch API

In this example, one node with MAC: "00:01:02:03:04:05" and ipv4="1.1.1.3" default_gateway ="1.1.1.2" and IPv6 enabled will be added. The API takes a batch of operations. Those operation could work in parallel to other API (e.g. traffic) to verify the response need to call `wait_for_async_results` API

```
c = STLClient(verbose_level = 'error')
c.connect()

my_ports=[0,1]
c.reset(ports = my_ports)                                                    (1)

# move to service mode
c.set_service_mode (ports = my_ports, enabled = True)                        (2)


cmds=NSCmds()                                                                (3)
MAC="00:01:02:03:04:05"
cmds.add_node(MAC)  # add namespace                                          (4)
cmds.set_vlan(MAC,[123,123]) # add valn + QinQ tags
cmds.set_ipv4(MAC,"1.1.1.3","1.1.1.2") # configure ipv4 and default gateway  (5)
cmds.set_ipv6(MAC,True) # enable ipv6 (auto mode, get src addrees from the router)  (6)


# start the batch
c.set_namespace_start( 0, cmds)                                              (7)
# wait for the results
res = c.wait_for_async_results(0)                                            (8)

# print the results
print(res);
```

(1) reset API won't remove old nodes

2. move to service mode
3. define a NSCmds object,it will be filled with async commands
4. Add a new node
5. configure ipv4
6. enable ipv6
7. provide the objects with all the commands to set_namespace_start
8. wait for the operation to finished

2.10.4.7 Blocking API

```
c = STLClient(verbose_level = 'error')
c.connect()

my_ports=[0,1]
c.reset(ports = my_ports)

# move to service mode
c.set_service_mode (ports = my_ports, enabled = True)

# remove all old name spaces from all the ports
c.namespace_remove_all()

# utility function
MAC="00:01:02:03:04:05"

# each function will block
c.set_namespace(0,method='add_node',mac=MAC)   (1)
c.set_namespace(0,method='set_vlan',vlans=[123,123])
c.set_namespace(0,method='set_ipv4', mac=MAC, ipv4="1.1.1.3", dg="1.1.1.2")
c.set_namespace(0,method='set_ipv6', mac=MAC, enable= True)
```

1. There is no need to fill the object NSCmds. Call set_namespace API with the namespace requested API

2.10.4.8 Statistics query

```
c = STLClient(verbose_level = 'error')
c.connect()
my_ports=[0,1]
c.reset(ports = my_ports)

# get all active nodes on port 0
r=c.set_namespace(0, method='get_nodes')                        (1)
c.set_namespace (0, method='get_nodes_info', macs_list=r)       (2)

r=c.set_namespace(0, method='counters_get_meta')                (3)
r=c.set_namespace(0, method='counters_get_values', zeros=True)  (4)
```

1. Get the active nodes (mac for each node)
2. Get full information per node
3. Get stats counters (names/type)
4. Get counters values

2.10.4.9 Ping to a new node

In this setup there are two ports 1.1.1.1 (port0) <→ 1.1.1.2 (port1)

This is the configuration before:

promiscuous is off

```
        trex>portattr -a
        Port Status

             port       |          0           |          1
        ----------------+----------------------+---------------------
        driver          |     net_vmxnet3      |     net_vmxnet3
        description     |  VMXNET3 Ethernet C  |  VMXNET3 Ethernet C
        link status     |          UP          |          UP
        link speed      |       10 Gb/s        |       10 Gb/s
        port status     |         IDLE         |         IDLE
        promiscuous     |         off          |         off
        multicast       |          on          |          on
        flow ctrl       |         N/A          |         N/A
        vxlan fs        |         N/A          |         N/A
        --              |                      |
        layer mode      |         IPv4         |         IPv4
        src IPv4        |       1.1.1.1        |       1.1.1.2
        IPv6            |         off          |         off
        src MAC         |  00:0c:29:b4:e7:e9   |  00:0c:29:b4:e7:11
        ---             |                      |
        Destination     |       1.1.1.2        |       1.1.1.1
        ARP Resolution  |  00:0c:29:b4:e7:11   |  00:0c:29:b4:e7:e9
        ----            |                      |
        VLAN            |          -           |          -
        -----           |                      |
        PCI Address     |     0000:0b:00.0     |     0000:0c:00.0
        NUMA Node       |          0           |          0
        RX Filter Mode  |    hardware match    |    hardware match
        RX Queueing     |         off          |         off
        Grat ARP        |         off          |         off
        ------          |                      |
```

Let's make sure we are in promiscuous mode and service mode

```
  trex>portattr --mul on
  trex>portattr --prom on
  trex>service
```

```
        trex>portattr -a
        Port Status

             port       |          0           |          1
        ----------------+----------------------+---------------------
        driver          |     net_vmxnet3      |     net_vmxnet3
        description     |  VMXNET3 Ethernet C  |  VMXNET3 Ethernet C
        link status     |          UP          |          UP
        link speed      |       10 Gb/s        |       10 Gb/s
        port status     |         IDLE         |         IDLE
        promiscuous     |          on          |          on
        multicast       |          on          |          on
        flow ctrl       |         N/A          |         N/A
        vxlan fs        |         N/A          |         N/A
        --              |                      |
        layer mode      |         IPv4         |         IPv4
        src IPv4        |       1.1.1.1        |       1.1.1.2
```

```
        IPv6            |         off          |         off
        src MAC         |  00:0c:29:b4:e7:e9   |  00:0c:29:b4:e7:11
        ---             |                      |
        Destination     |       1.1.1.2        |       1.1.1.1
        ARP Resolution  |  00:0c:29:b4:e7:11   |  00:0c:29:b4:e7:e9
        ----            |                      |
        VLAN            |          -           |          -
        -----           |                      |
        PCI Address     |     0000:0b:00.0     |     0000:0c:00.0
        NUMA Node       |          0           |          0
        RX Filter Mode  |    hardware match    |    hardware match
        RX Queueing     |         off          |         off
        Grat ARP        |         off          |         off
        ------
```

Let's call this script to add a new node on port 0

```
    cmds=NSCmds()
    MAC="00:01:02:03:04:05"
    cmds.add_node(MAC)
    cmds.set_ipv4(MAC,"1.1.1.3","1.1.1.2")
    cmds.set_ipv6(MAC,True)

    res = c.set_namespace_start( 0, cmds)
    res = c.wait_for_async_results(0)
```

Now we can ping to the new node

```
  trex(service)>l3 -p 1 --src 1.1.1.2 --dst 1.1.1.3
  trex(service)>ping -p 1 -d 1.1.1.3

   Pinging 1.1.1.3 from port 1 with 64 bytes of data:
   Reply from 1.1.1.3: bytes=64, time=32.29ms, TTL=64
   Reply from 1.1.1.3: bytes=64, time=3.73ms, TTL=64
   Reply from 1.1.1.3: bytes=64, time=3.89ms, TTL=64
   Reply from 1.1.1.3: bytes=64, time=2.85ms, TTL=64
   Reply from 1.1.1.3: bytes=64, time=2.82ms, TTL=64
```

to debug it from Linux shell you can do this

```
$sudo ip netns show
```

This will shows al the network namespace

for port 0 the name is **trex-a-0-x** where X is the number of the namespace

to look into the information

```
$sudo ip netns exec trex-a-0-1 ifconfig
trex-a-0-1-L: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 9280
        inet 1.1.1.3  netmask 255.255.255.255  broadcast 0.0.0.0
        inet6 fe80::201:2ff:fe03:405  prefixlen 64  scopeid 0x20<link>
        ether 00:01:02:03:04:05  txqueuelen 1000  (Ethernet)
        RX packets 1  bytes 60 (60.0 B)
        RX errors 0  dropped 1  overruns 0  frame 0
        TX packets 8  bytes 648 (648.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Note
It is possible to monitor and capture the traffic from/to the namespace using the usual capture capability

Note
You won't be able to ping 1.1.1.1 (original port ip) as TRex ping has one default gateway

2.10.4.10 Console commands example

The Console commands objective is for low scale operation add/removing one node at a time The folowing example present that

```
trex(service)>ns add -p 0 --mac 00:01:02:03:04:05 --src 1.1.1.3 --dst 1.1.1.2 --ipv6

trex(service)>ns show-node -p 0 --mac 00:01:02:03:04:05

 {'nodes': [{'ether': {'src': '00:01:02:03:04:05'},
          'ipv4': {'dst': '1.1.1.2', 'src': '1.1.1.3'},
          'ipv6': {'enabled': True, 'src': ''},
          'linux-ns': 'trex-a-0-1',
          'linux-veth-external': 'trex-a-0-1-T',
          'linux-veth-internal': 'trex-a-0-1-L',
          'vlan': {'tags': []}}]}


trex(service)>ns show-nodes -p 0

     Setting port 0 in with namespace configuration               [SUCCESS]


      wait_for_async_results                                       [SUCCESS]

      ns nods

       node-id   |        mac
      -----------+------------------
          0      | 00:01:02:03:04:05


trex(service)>ns show-counters -p 0

      Setting port 0 in with namespace configuration               [SUCCESS]


      wait_for_async_results                                       [SUCCESS]

      ns stats

              name         |       value       |         help
      ---------------------+-------------------+---------------------
      rx_unicast_bytes     |        60         | rx unicast bytes
      tx_multicast_bytes   |        648        | tx multicast bytes
      rx_unicast_pkts      |         1         | rx unicast pkts
      tx_multicast_pkts    |         8         | tx multicast pkts
      rx_multicast_bytes   |        480        | rx multicast bytes
      rx_multicast_pkts    |         8         | rx multicast pkts
```

2.11 Traffic profile tutorials

2.11.1 Tutorial: Simple interleaving streams

Goal
Demonstrate interleaving of multiple streams.

The following example demonstrates 3 streams with different rates (10, 20, 40 PPS) and different start times, based on an inter-stream gap (ISG) of 0, 25 msec, or 50 msec.

File
stl/simple_3pkt.py

Interleaving multiple streams

```
    def create_stream (self):

        # create a base packet and pad it to size
        size = self.fsize - 4 # no FCS
        base_pkt =  Ether()/IP(src="16.0.0.1",dst="48.0.0.1")/UDP(dport=12,sport=1025)      1
        base_pkt1 =  Ether()/IP(src="16.0.0.2",dst="48.0.0.1")/UDP(dport=12,sport=1025)
        base_pkt2 =  Ether()/IP(src="16.0.0.3",dst="48.0.0.1")/UDP(dport=12,sport=1025)
        pad = max(0, size - len(base_pkt)) * 'x'


        return STLProfile( [ STLStream( isg = 0.0,
                                        packet = STLPktBuilder(pkt = base_pkt/pad),
                                        mode = STLTXCont( pps = 10),                         2
                                        ),

                             STLStream( isg = 25000.0, #defined in usec, 25 msec
                                        packet  = STLPktBuilder(pkt = base_pkt1/pad),
                                        mode    = STLTXCont( pps = 20),                      3
                                        ),

                             STLStream(  isg = 50000.0,#defined in usec, 50 msec
                                         packet = STLPktBuilder(pkt = base_pkt2/pad),
                                         mode    = STLTXCont( pps = 40)                      4

                                        )
                           ]).get_streams()
```

1. Defines template packets using Scapy.
2. Defines streams with rate of 10 PPS.
3. Defines streams with rate of 20 PPS.
4. Defines streams with rate of 40 PPS.

Output
The following figure presents the output.

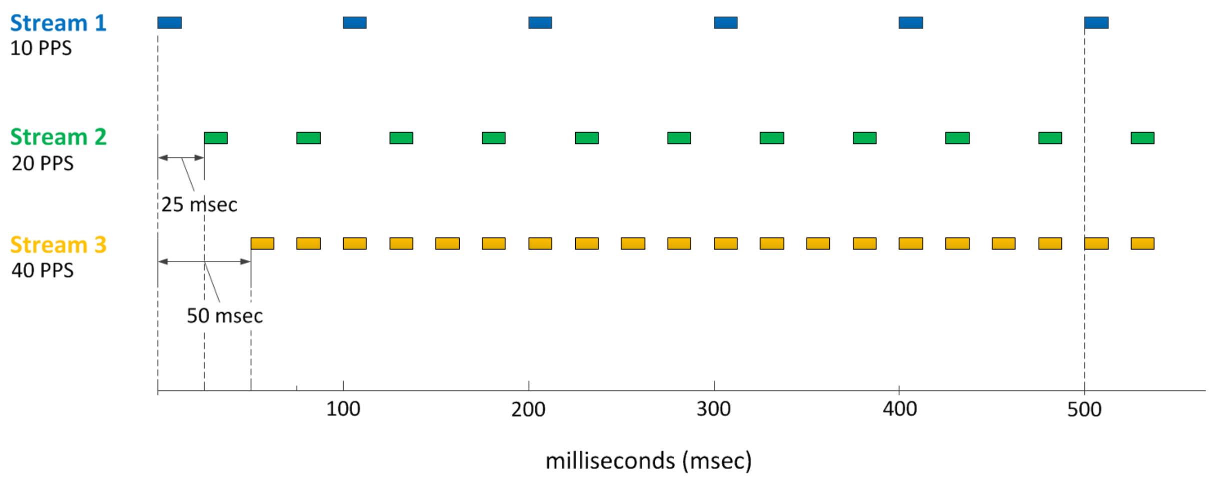


Figure 2.12: Interleaving of streams

Discussion

- Stream #1
  - Schedules a packet each 100 msec
- Stream #2
  - Schedules a packet each 50 msec
  - Starts 25 msec after stream #1
- Stream #3
  - Schedules a packet each 25 msec
  - Starts 50 msec after stream #1

You can run the traffic profile in the TRex simulator and view the details in the PCAP file containing the simulation output.

```
[bash]>./stl-sim -f stl/simple_3pkt.py -o b.pcap -l 200
```

To run the traffic profile from console in TRex, use the following command.

```
trex>start -f stl/simple_3pkt.py -m 10mbps
```

2.11.2 Tutorial: Multi burst streams - action next stream

Goal
Create a profile with a stream that trigger another stream

The following example demonstrates:

1. More than one stream
2. Burst of 10 packets
3. One stream activating another stream (see `self_start=False` in the traffic profile)

File
stl/burst_3pkt_60pkt.py

```
    def create_stream (self):

        # create a base packet and pad it to size
        size = self.fsize - 4 # no FCS
        base_pkt =  Ether()/IP(src="16.0.0.1",dst="48.0.0.1")/UDP(dport=12,sport=1025)
        base_pkt1 =  Ether()/IP(src="16.0.0.2",dst="48.0.0.1")/UDP(dport=12,sport=1025)
        base_pkt2 =  Ether()/IP(src="16.0.0.3",dst="48.0.0.1")/UDP(dport=12,sport=1025)
        pad = max(0, size - len(base_pkt)) * 'x'


        return STLProfile( [ STLStream( isg = 10.0, # star in delay
                                        name    ='S0',
                                        packet = STLPktBuilder(pkt = base_pkt/pad),
                                        mode = STLTXSingleBurst( pps = 10, total_pkts = 10) ↩
                                            ,  ❶
                                        next = 'S1'), # point to next stream

                             STLStream( self_start = False, # stream is  disabled enable ↩
                                 trow S0  ❷
                                        name    ='S1',
                                        packet  = STLPktBuilder(pkt = base_pkt1/pad),
                                        mode    = STLTXSingleBurst( pps = 10, total_pkts = ↩
                                            20),
                                        next    = 'S2' ),

                             STLStream(  self_start = False, # stream is  disabled enable ↩
                                 trow S0 ❸
                                         name    ='S2',
                                         packet = STLPktBuilder(pkt = base_pkt2/pad),
                                         mode = STLTXSingleBurst( pps = 10, total_pkts = 30 ↩
                                              )
                                        )
                            ]).get_streams()
```

1. Stream S0 is configured to `self_start=True`, starts after 10 sec.
2. S1 is configured to `self_start=False`, activated by stream S0.
3. S2 is activated by S1.

To run the simulation, use this command.

```
[bash]>./stl-sim -f stl/stl/burst_3pkt_60pkt.py -o b.pcap
```

The generated PCAP file has 60 packets. The first 10 packets have src_ip=16.0.0.1. The next 20 packets has src_ip=16.0.0.2. The next 30 packets has src_ip=16.0.0.3.

This run the profile from console use this command.

```
TRex>start -f stl/stl/burst_3pkt_60pkt.py --port 0
```

2.11.3 Tutorial: Multi-burst mode

Goal : Use Multi-burst transmit mode

File
: stl/multi_burst_2st_1000pkt.py

```
    def create_stream (self):

        # create a base packet and pad it to size
        size = self.fsize - 4 # no FCS
        base_pkt =  Ether()/IP(src="16.0.0.1",dst="48.0.0.1")/UDP(dport=12,sport=1025)
        base_pkt1 =  Ether()/IP(src="16.0.0.2",dst="48.0.0.1")/UDP(dport=12,sport=1025)
        pad = max(0, size - len(base_pkt)) * 'x'


        return STLProfile( [ STLStream( isg = 10.0, # start in delay ↩
                                        (1)
                                        name    ='S0',
                                        packet = STLPktBuilder(pkt = base_pkt/pad),
                                        mode = STLTXSingleBurst( pps = 10, total_pkts =  ↩
                                            self.burst_size),
                                        next = 'S1'), # point to next stream

                             STLStream( self_start = False, # stream is disabled. Enabled  ↩
                                 by S0          (2)
                                        name    ='S1',
                                        packet  = STLPktBuilder(pkt = base_pkt1/pad),
                                        mode    = STLTXMultiBurst( pps = 1000,
                                                                   pkts_per_burst = 4,
                                                                   ibg = 1000000.0,
                                                                   count = 5)
                                        )

                           ]).get_streams()
```

1. Stream S0 waits 10 usec (inter-stream gap, ISG) and then sends a burst of self.burst_size packets at 10 PPS.
2. Multi-burst of 5 bursts of 4 packets with an inter-burst gap of 1 second.

The following illustration does not fully match the Python example cited above. It has been simplified, such as using a 0.5 second ISG, for illustration purposes.

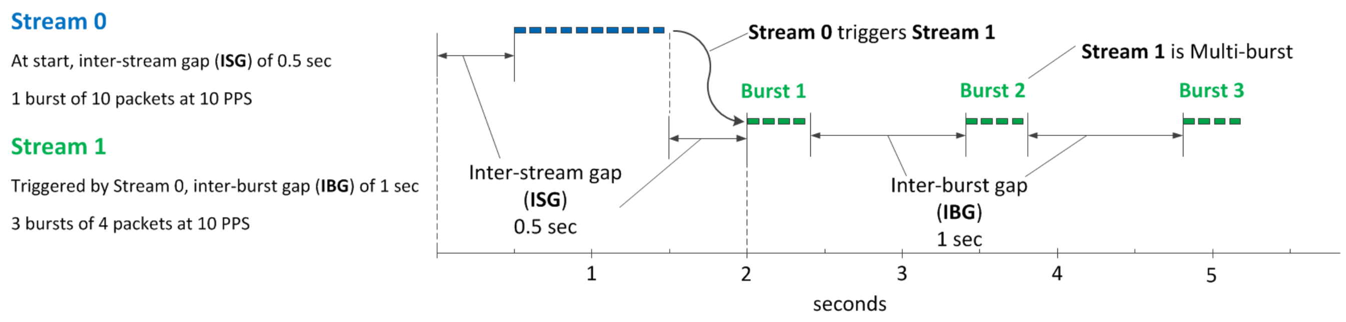


Figure 2.13: Example of multiple streams

2.11.4 Tutorial: Loops of streams

Goal: Demonstrate a limited loop of streams.

File

stl/burst_3st_loop_x_times.py

```
    def create_stream (self):

        # create a base packet and pad it to size
        size = self.fsize - 4 # no FCS
        base_pkt =  Ether()/IP(src="16.0.0.1",dst="48.0.0.1")/UDP(dport=12,sport=1025)
        base_pkt1 =  Ether()/IP(src="16.0.0.2",dst="48.0.0.1")/UDP(dport=12,sport=1025)
        base_pkt2 =  Ether()/IP(src="16.0.0.3",dst="48.0.0.1")/UDP(dport=12,sport=1025)
        pad = max(0, size - len(base_pkt)) * 'x'


        return STLProfile( [ STLStream( isg = 10.0, # start in delay
                                        name    ='S0',
                                        packet = STLPktBuilder(pkt = base_pkt/pad),
                                        mode = STLTXSingleBurst( pps = 10, total_pkts = 1),
                                        next = 'S1'), # point to next stream

                             STLStream( self_start = False, # stream is disabled. Enabled ↩
                                 by S0
                                        name    ='S1',
                                        packet  = STLPktBuilder(pkt = base_pkt1/pad),
                                        mode    = STLTXSingleBurst( pps = 10, total_pkts = ↩
                                            2),
                                        next    = 'S2' ),

                             STLStream(  self_start = False, # stream is disabled. Enabled ↩
                                 by S1
                                         name    ='S2',
                                         packet = STLPktBuilder(pkt = base_pkt2/pad),
                                         mode = STLTXSingleBurst( pps = 10, total_pkts = 3 ↩
                                             ),
                                         action_count = 2, # loop 2 times  ↩
                                                                (1)
                                         next     = 'S0' # loop back to S0
                                        )
                            ]).get_streams()
```

(1) Go back to S0, but limit it to 2 loops.

2.11.5 Tutorial: IMIX with UDP packets, bi-directional

Goal: Demonstrate how to create an IMIX traffic profile.

This profile defines 3 streams, with packets of different sizes. The rate is different for each stream/size. See the Wikipedia article on Internet Mix.

File
stl/imix.py

```
    def __init__ (self):
        # default IP range
        self.ip_range = {'src': {'start': "10.0.0.1", 'end': "10.0.0.254"},
                         'dst': {'start': "8.0.0.1",  'end': "8.0.0.254"}}

        # default IMIX properties
        self.imix_table = [ {'size': 60,   'pps': 28,  'isg':0 },
                            {'size': 590,  'pps': 16,  'isg':0.1 },
                            {'size': 1514, 'pps': 4,   'isg':0.2 } ]
```

```
def create_stream (self, size, pps, isg, vm ):
    # create a base packet and pad it to size
    base_pkt = Ether()/IP()/UDP()
    pad = max(0, size - len(base_pkt)) * 'x'

    pkt = STLPktBuilder(pkt = base_pkt/pad,
                        vm = vm)

    return STLStream(isg = isg,
                     packet = pkt,
                     mode = STLTXCont(pps = pps))


def get_streams (self, direction = 0, **kwargs):                    ❶

    if direction == 0:                                              ❷
        src = self.ip_range['src']
        dst = self.ip_range['dst']
    else:
        src = self.ip_range['dst']
        dst = self.ip_range['src']

    # construct the base packet for the profile

    vm =[                                                           ❸
        # src
        STLVmFlowVar(name="src",
                     min_value=src['start'],
                     max_value=src['end'],
                     size=4,op="inc"),
        STLVmWrFlowVar(fv_name="src",pkt_offset= "IP.src"),

        # dst
        STLVmFlowVar(name="dst",
                     min_value=dst['start'],
                     max_value=dst['end'],
                     size=4,
                     op="inc"),
        STLVmWrFlowVar(fv_name="dst",pkt_offset= "IP.dst"),

        # checksum
        STLVmFixIpv4(offset = "IP")

        ]

    # create imix streams
    return [self.create_stream(x['size'], x['pps'],x['isg'] , vm) for x in self. ↩
        imix_table]
```

❶ Constructs a diffrent stream for each direction (replaces src and dest).

❷ Even port id has direction==0 and odd has direction==1.

❸ Field Engine program to change fields within the packets.

2.11.6 Tutorial: Field Engine, syn attack

The following example demonstrates changing packet fields. The Field Engine (FE) has a limited number of instructions/operation, which support most use cases.

The FE can:

- Allocate stream variables in a stream context
- Write a stream variable to a packet offset
- Change packet size
- and more...
- **Plan for future version**: Add LuaJIT to be more flexible at the cost of performance.

Examples:

- Change ipv4.tos value (1 to 10)
- Change packet size to a random value in the range 64 to 9K
- Create a range of flows (change src_ip, dest_ip, src_port, dest_port)
- Update the IPv4 checksum

For more information, see: TRex RPC Server

The following example demonstrates creating a SYN attack from many src addresses to one server.

File
stl/syn_attack.py

```
    def create_stream (self):

        # TCP SYN
        base_pkt  = Ether()/IP(dst="48.0.0.1")/TCP(dport=80,flags="S")        ❶


        # vm
        vm = STLScVmRaw( [ STLVmFlowVar(name="ip_src",
                                        min_value="16.0.0.0",
                                        max_value="18.0.0.254",
                                        size=4, op="random"),            ❷

                           STLVmFlowVar(name="src_port",
                                        min_value=1025,
                                        max_value=65000,
                                        size=2, op="random"),            ❸

                           STLVmWrFlowVar(fv_name="ip_src", pkt_offset= "IP.src" ), ❹

                           STLVmFixIpv4(offset = "IP"), # fix checksum          ❺

                           STLVmWrFlowVar(fv_name="src_port",                   ❻
                                          pkt_offset= "TCP.sport") # U

                          ]
                        )

        pkt = STLPktBuilder(pkt = base_pkt,
                            vm = vm)

        return STLStream(packet = pkt,
                         random_seed = 0x1234,# can be removed. will give the same random ↩
                             value any run
                         mode = STLTXCont())
```

1. Creates SYN packet using Scapy .
2. Defines a stream variable `name=ip_src`, size 4 bytes, for IPv4.
3. Defines a stream variable `name=src_port`, size 2 bytes, for port.
4. Writes `ip_src` stream var into `IP.src` packet offset. Scapy calculates the offset. Can specify `IP:1.src` for a second IP header in the packet.
5. Fixes IPv4 checksum. Provides the header name `IP`. Can specify `IP:1` for a second IP.
6. Writes `src_port` stream var into `TCP.sport` packet offset. TCP checksum is not updated here.

Warning
Original Scapy cannot calculate offset for a header/field by name. This offset capability will not work for all cases. In some complex cases, Scapy may rebuild the header. In such cases, specify the offset as a number.

Output PCAP file:

Table 2.7: Output - PCAP file

pkt	Client IPv4	Client Port
1	17.152.71.218	5814
2	17.7.6.30	26810
3	17.3.32.200	1810
4	17.135.236.168	55810
5	17.46.240.12	1078
6	16.133.91.247	2323

2.11.7 Tutorial: Field Engine, tuple generator

The following example creates multiple flows from the same packet template. The Tuple Generator instructions are used to create two stream variables for IP and port. See: TRex RPC Server

File
stl/udp_1pkt_tuple_gen.py

```
base_pkt = Ether()/IP(src="16.0.0.1",dst="48.0.0.1")/UDP(dport=12,sport=1025)

pad = max(0, size - len(base_pkt)) * 'x'

vm = STLScVmRaw( [   STLVmTupleGen ( ip_min="16.0.0.1",                    ❶
                                     ip_max="16.0.0.2",
                                     port_min=1025,
                                     port_max=65535,
                                     name="tuple"), # define tuple gen

                     STLVmWrFlowVar (fv_name="tuple.ip", pkt_offset= "IP.src" ),  ❷
                     STLVmFixIpv4(offset = "IP"),
                     STLVmWrFlowVar (fv_name="tuple.port", pkt_offset= "UDP.sport" ↵
                         ) ❸
                       ]
                 )
```

```
        pkt = STLPktBuilder(pkt = base_pkt/pad,
                            vm = vm)
```

1. Defines a struct with two dependent variables: tuple.ip, tuple.port
2. Writes the tuple.ip variable to `IPv4.src` field offset.
3. Writes the tuple.port variable to `UDP.sport` field offset. Set `UDP.checksum` to 0.

Table 2.8: Output - PCAP file

pkt	Client IPv4	Client Port
1	16.0.0.1	1025
2	16.0.0.2	1025
3	16.0.0.1	1026
4	16.0.0.2	1026
5	16.0.0.1	1027
6	16.0.0.2	1027

- Number of clients: 2: 16.0.0.1 and 16.0.0.2
- Number of flows is limited to 129020: (2 * (65535-1025))
- The stream variable size should match the size of the FlowVarWr instruction.

2.11.8 Tutorial: Field Engine, write to a bit-field packet

The following example writes a stream variable to a bit field packet variable. In this example, an MPLS label field is changed.

Table 2.9: MPLS header

Label																				TC			S	TTL							
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1

File
stl/udp_1pkt_mpls_vm.py

```
    def create_stream (self):
        # 2 MPLS label the internal with  s=1 (last one)
        pkt =  Ether()/
               MPLS(label=17,cos=1,s=0,ttl=255)/
               MPLS(label=0,cos=1,s=1,ttl=12)/
               IP(src="16.0.0.1",dst="48.0.0.1")/
               UDP(dport=12,sport=1025)/('x'*20)

        vm = STLScVmRaw( [ STLVmFlowVar(name="mlabel",                    ❶
                                        min_value=1,
                                        max_value=2000,
                                        size=2, op="inc"), # 2 bytes var  ❷
```

```
                                  STLVmWrMaskFlowVar(fv_name="mlabel",
                                                     pkt_offset= "MPLS:1.label",          ❸
                                                     pkt_cast_size=4,
                                                     mask=0xFFFFF000,shift=12) # write to 20bit ←
                                                         MSB
                                 ]
                             )

        # burst of 100 packets
        return STLStream(packet = STLPktBuilder(pkt = pkt ,vm = vm),
                         mode = STLTXSingleBurst( pps = 1, total_pkts = 100) )
```

❶ Defines a variable size of 2 bytes.

❷ Writes the stream variable label with a shift of 12 bits, with a 20-bit MSB mask. Cast the stream variables of 2 bytes to 4 bytes.

❸ Change the second MPLS header.

2.11.9 Tutorial: Field Engine, random packet size

The following example demonstrates varies the packet size randomly, as follows:

1. Defines the template packet with maximum size.
2. Trims the packet to the size you want.
3. Updates the packet fields according to the new size.

File
stl/udp_rand_len_9k.py

```
    def create_stream (self):
        # pkt
        p_l2  = Ether()
        p_l3  = IP(src="16.0.0.1",dst="48.0.0.1")
        p_l4  = UDP(dport=12,sport=1025)
        pyld_size = max(0, self.max_pkt_size_l3 - len(p_l3/p_l4))
        base_pkt = p_l2/p_l3/p_l4/('\x55'*(pyld_size))

        l3_len_fix =-(len(p_l2))
        l4_len_fix =-(len(p_l2/p_l3))


        # vm
        vm = STLScVmRaw( [ STLVmFlowVar(name="fv_rand",                          ❶
                                        min_value=64,
                                        max_value=len(base_pkt),
                                        size=2,
                                        op="random"),

                           STLVmTrimPktSize("fv_rand"), # total packet size      ❷

                           STLVmWrFlowVar(fv_name="fv_rand",                      ❸
                                          pkt_offset= "IP.len",
                                          add_val=l3_len_fix), # fix ip len

                           STLVmFixIpv4(offset = "IP"),

                           STLVmWrFlowVar(fv_name="fv_rand",                      ❹
```

```
                                                    pkt_offset= "UDP.len",
                                                    add_val=l4_len_fix) # fix udp len
                                ]
                             )
```

1. Defines a random stream variable with the maximum size of the packet.
2. Trims the packet size to the `fv_rand` value.
3. Fixes ip.len to reflect the packet size.
4. Fixes udp.len to reflect the packet size.

2.11.10 Tutorial: Field Engine: Pre-caching to improve performance

The following example demonstrates how to significantly improve Field Engine performance, if necessary.

The Field Engine has a cost in CPU resources: CPU cycles and CPU memory bandwidth (bandwidth available for CPU to read from or write to memory). It is possible to significantly improve performance by caching the packets and running the Field Engine offline (before sending the packets). Typically, this is done with small packets (example: 64 bytes) where performance is an issue. This method can also improve a large-packet senario with a complex Field Engine program.

Limitations of the pre-caching method:

1. Only a limited number of packets can be cached. The total number of cached packets for all the streams on all ports is limited by the memory pool (range: approximately 10 - 40K).
2. Pre-caching packets is not appropriate for some traffic requirements. Examples: A program that steps in prime numbers or uses a random variable.

An example of a scenario that cannot use this method, due to the packet limitation, is a program that changes the src_ip randomly. Pre-caching a limited number of packets would not be compatible with continuously varying the src_ip randomly.

File
stl/udp_1pkt_src_ip_split.py

```
    def create_stream (self):
        # create a base packet and pad it to size
        size = self.fsize - 4; # no FCS

        base_pkt = Ether()/IP(src="16.0.0.1",dst="48.0.0.1")/UDP(dport=12,sport=1025)

        pad = max(0, size - len(base_pkt)) * 'x'

        vm = STLScVmRaw( [   STLVmFlowVar ( "ip_src",
                                            min_value="10.0.0.1",
                                            max_value="10.0.0.255",
                                            size=4, step=1,op="inc"),

                             STLVmWrFlowVar (fv_name="ip_src",
                                             pkt_offset= "IP.src" ),

                             STLVmFixIpv4(offset = "IP")
                         ],
                         split_by_field = "ip_src",
                         cache_size =255 # the cache size                    (1)
                        );

        pkt = STLPktBuilder(pkt = base_pkt/pad,
```

```
                            vm = vm)

        stream = STLStream(packet = pkt,
                           mode = STLTXCont())
        return stream
```

1 Cache 255 packets. The range is the same as the ip_src stream variable.

This example FE program is fully compatible with pre-caching - the traffic output is exactly the same as when running without pre-caching. Caching the packets enables the program to run **2 to 5 times faster**.

2.11.11 Tutorial: New Scapy header

The following example uses a header that is not supported by Scapy by default. The example demonstrates VXLAN support.

File
stl/udp_1pkt_vxlan.py

```
# Adding header that does not exists yet in Scapy
# This was taken from pull request of Scapy
#


# RFC 7348 - Virtual eXtensible Local Area Network (VXLAN):  ↩
                                                   ❶
# A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks
# http://tools.ietf.org/html/rfc7348
_VXLAN_FLAGS = ['R' for i in range(0, 24)] + ['R', 'R', 'R', 'I', 'R', 'R', 'R', 'R', 'R']

class VXLAN(Packet):
    name = "VXLAN"
    fields_desc = [FlagsField("flags", 0x08000000, 32, _VXLAN_FLAGS),
                   ThreeBytesField("vni", 0),
                   XByteField("reserved", 0x00)]

    def mysummary(self):
        return self.sprintf("VXLAN (vni=%VXLAN.vni%)")

bind_layers(UDP, VXLAN, dport=4789)
bind_layers(VXLAN, Ether)


class STLS1(object):

    def __init__ (self):
        pass

    def create_stream (self):
        pkt =  Ether()/IP()/UDP(sport=1337,dport=4789)/VXLAN(vni=42)/Ether()/IP()/('x'*20)  ↩
                  ❷
        #pkt.show2()
        #hexdump(pkt)

        # burst of 17 packets
        return STLStream(packet = STLPktBuilder(pkt = pkt ,vm = []),
                         mode = STLTXSingleBurst( pps = 1, total_pkts = 17) )
```

1. Downloads and adds a Scapy header from the specified location. An alternative is to write a Scapy header.
2. Apply the header.

For more information how to define headers see Adding new protocols in the Scapy documentation.

2.11.12 Tutorial: Field Engine, Multiple Clients

The following example generates traffic from many clients with different IP/MAC addresses to one server.

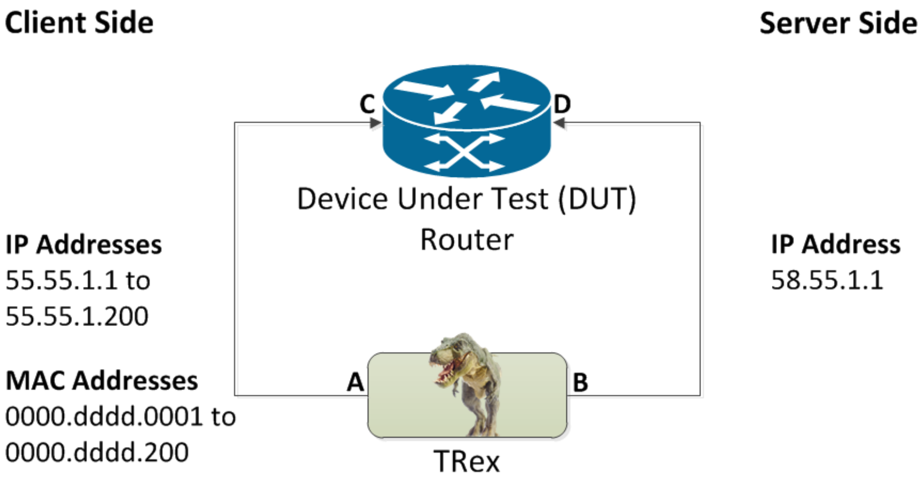


Figure 2.14: Multiple clients, single server

1. Send a gratuitous ARP from B→D with server IP/MAC (58.55.1.1).
2. DUT learns the ARP of server IP/MAC (58.55.1.1).
3. Send traffic from A→C with many client IP/MAC addresses.

Example:

- Base source IPv4: 55.55.1.1
- Destination IPv4: 58.55.1.1

Increment src ipt portion starting at 55.55.1.1 for n number of clients (55.55.1.1, 55.55.1.2)
Src MAC: Start with 0000.dddd.0001, increment MAC in steps of 1
Dst MAC: Fixed 58.55.1.1

The following sends a gratuitous ARP from the TRex server port for this server (58.0.0.1).

```
    def create_stream (self):
        # create a base packet and pad it to size
        base_pkt =  Ether(src="00:00:dd:dd:01:01",
                          dst="ff:ff:ff:ff:ff:ff")/
                    ARP(psrc="58.55.1.1",
                        hwsrc="00:00:dd:dd:01:01",
                        hwdst="00:00:dd:dd:01:01",
                        pdst="58.55.1.1")
```

Then traffic can be sent from client side: A→C

File
stl/udp_1pkt_range_clients_split.py

```
class STLS1(object):

    def __init__ (self):
        self.num_clients  =30000 # max is 16bit
        self.fsize        =64

    def create_stream (self):

        # create a base packet and pad it to size
        size = self.fsize - 4 # no FCS
        base_pkt =  Ether(src="00:00:dd:dd:00:01")/
                        IP(src="55.55.1.1",dst="58.55.1.1")/UDP(dport=12,sport=1025)
        pad = max(0, size - len(base_pkt)) * 'x'

        vm = STLScVmRaw( [ STLVmFlowVar(name="mac_src",
                                        min_value=1,
                                        max_value=self.num_clients,
                                        size=2, op="inc"), # 1 byte varible, range 1-10

                            STLVmWrFlowVar(fv_name="mac_src", pkt_offset= 10),          ❶
                            STLVmWrFlowVar(fv_name="mac_src" ,
                                           pkt_offset="IP.src",
                                           offset_fixup=2),                             ❷
                            STLVmFixIpv4(offset = "IP")
                           ]
                          ,split_by_field = "mac_src"  # split
                        )

        return STLStream(packet = STLPktBuilder(pkt = base_pkt/pad,vm = vm),
                         mode = STLTXCont( pps=10 ))
```

❶ Writes the stream variable `mac_src` with an offset of 10 (last 2 bytes of `src_mac` field). The offset is specified explicitly as 10 bytes from the beginning of the packet.

❷ Writes the stream variable `mac_src` with an offset determined by the offset of `IP.src` plus the `offset_fixup` of 2.

2.11.13 Tutorial: Field Engine, many clients with ARP

In the following example, there are two switches: SW1 and SW2. TRex port 0 is connected to SW1 and TRex port 1 is connected to SW2. There are 253 hosts connected to SW1 and SW2 with two network ports.

Table 2.10: Client side the network of the hosts

Name	Description
TRex port 0 MAC	00:00:01:00:00:01
TRex port 0 IPv4	16.0.0.1
IPv4 host client side range	16.0.0.2-16.0.0.254
MAC host client side range	00:00:01:00:00:02- 00:00:01:00:00:FE

Table 2.11: Server side the network of the hosts

Name	Description
TRex port 1 MAC	00:00:02:00:00:01
TRex port 1 IPv4	48.0.0.1
IPv4 host server side range	48.0.0.2-48.0.0.254
MAC host server side range	00:00:02:00:00:02-00:00:02:00:00:FE

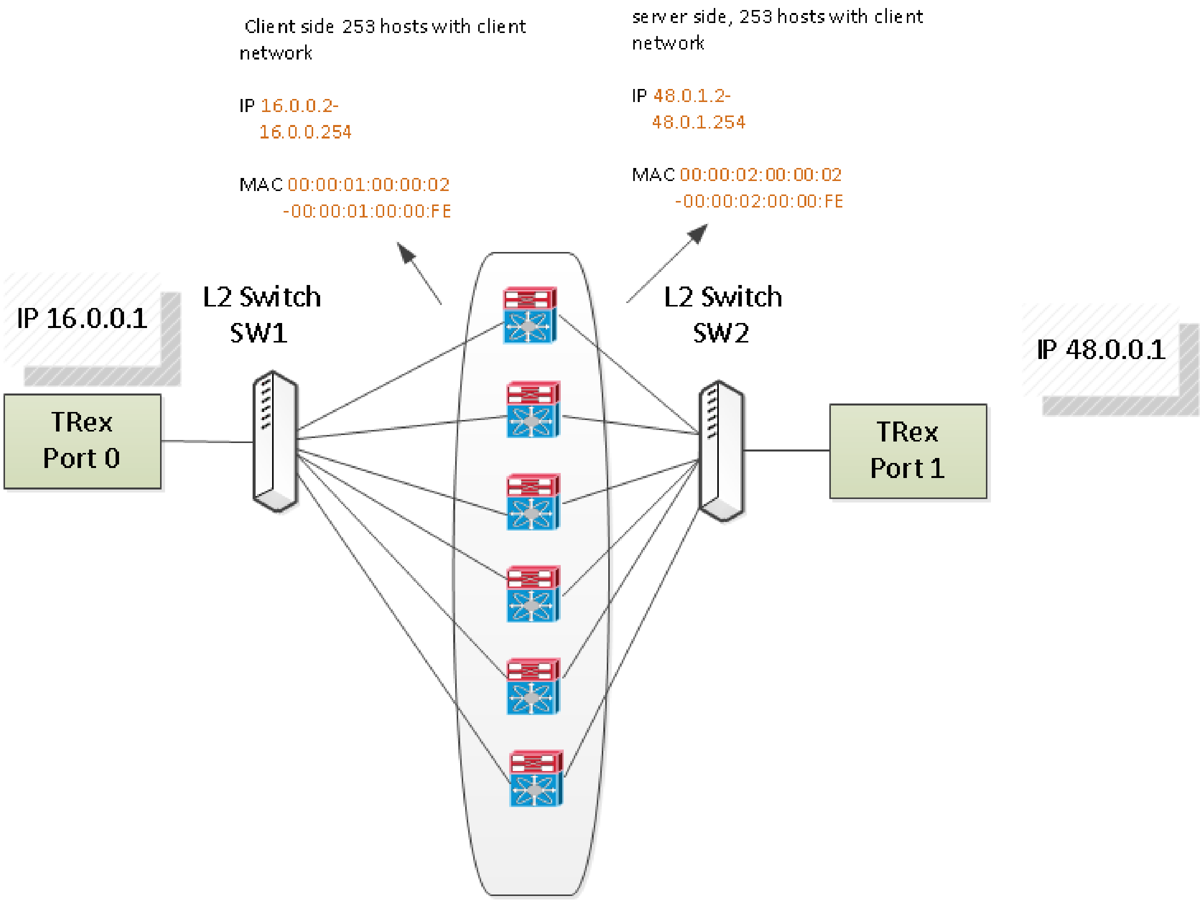


Figure 2.15: arp/nd

In the following example, there are two switches: SW1 and SW2. TRex port 0 is connected to SW1 and TRex port 1 is connected to SW2. In this example, because there are many hosts connected to the same network using SW1 and not as a next hop, we would like to teach SW1 the MAC addresses of the hosts and not to send the traffic directly to the hosts MAC (as it is unknown). For that we would send an ARP to all the hosts (16.0.0.2-16.0.0.254) from TRex port 0 and gratuitous ARP from server side (48.0.0.1) TRex port 1 as the first stage of the test.

Procedure:

1. Send a gratuitous ARP from TRex port 1 with server IP/MAC (48.0.0.1). Now SW2 will know that 48.0.0.1 is located after this port of SW2.
2. Send ARP request for all hosts from port 0 with a range of 16.0.0.2-16.0.0.254. Now all switch ports will learn the PORT/MAC locations. Without this stage, the first packets from TRex port 0 would be flooded to all switch ports.

3. Send traffic from TRex0→clients, port 1→servers.

ARP traffic profile

```
base_pkt =  Ether(dst="ff:ff:ff:ff:ff:ff")/
           ARP(psrc="16.0.0.1",hwsrc="00:00:01:00:00:01", pdst="16.0.0.2")  ↩
                                        ❶


vm = STLScVmRaw( [ STLVmFlowVar(name="mac_src", min_value=2, max_value=254, size=2, op=" ↩
    inc"),  ❷
                   STLVmWrFlowVar(fv_name="mac_src" ,pkt_offset="ARP.pdst",offset_fixup=2) ↩
                       ,
                 ]
                ,split_by_field = "mac_src"  # split
               )
```

❶ ARP packet with TRex port 0 MAC and IP and pdst as variable.

❷ Write it to `ARP.pdst`.

Gratuitous ARP traffic profile

```
       base_pkt =  Ether(src="00:00:02:00:00:01",dst="ff:ff:ff:ff:ff:ff")/
                   ARP(psrc="48.0.0.1",hwsrc="00:00:02:00:00:01",
                       hwdst="00:00:02:00:00:01", pdst="48.0.0.1") ❶
```

❶ Gratuitous ARP packet with TRex port 1 MAC and IP. No VM is needed.

> **Note**
> This can be also be done for IPv6. ARP could be replaced with Neighbor Solicitation IPv6 packet.

2.11.14 Tutorial: Field Engine, null stream

The following example creates a stream with no packets. The example uses the inter-stream gap (ISG) of the null stream, and then starts a new stream. Essentially, this uses one property of the stream (ISG) without actually including packets in the stream.

This method can create loops like the following:

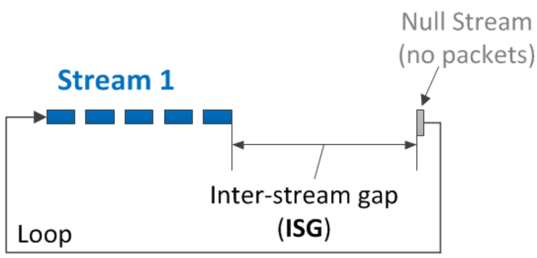


Figure 2.16: Null stream

File

null_stream.py

```
    def create_stream(self):
        size = self.fsize - 4  # no FCS
        base_pkt = Ether()/IP(src = "16.0.0.1", dst = "48.0.0.1")/UDP(dport = 12, sport = ↩
            1025)
        pad = max(0, size - len(base_pkt)) * 'x'

        return STLProfile([ STLStream( name = 'S1',     ❶
                                       packet = STLPktBuilder(pkt = base_pkt/pad),
                                       mode = STLTXSingleBurst(pps = 10, total_pkts = 5),
                                       next = 'NULL'),
                            STLStream( self_start = False,     ❷
                                       isg = 1000000.0,
                                       name = 'NULL',
                                       mode = STLTXSingleBurst(),
                                       dummy_stream = True,
                                       next = 'S1')
                          ]).get_streams()
```

❶ S1 - Sends a burst of packets, then proceed to stream NULL.

❷ NULL - Waits the inter-stream gap (ISG) time, then proceeds to S1.

Null stream configuration requirements:

1. Mode: Single Burst
2. dummy_stream = True

2.11.15 Tutorial: Field Engine, stream barrier (split)

(Future feature - not yet implemented)

In some situations, it is necessary to split streams into threads in such a way that specific streams will continue only after all the threads have passed the same path. In the figure below, a barrier ensures that stream S3 starts only after all threads of S2 are complete.

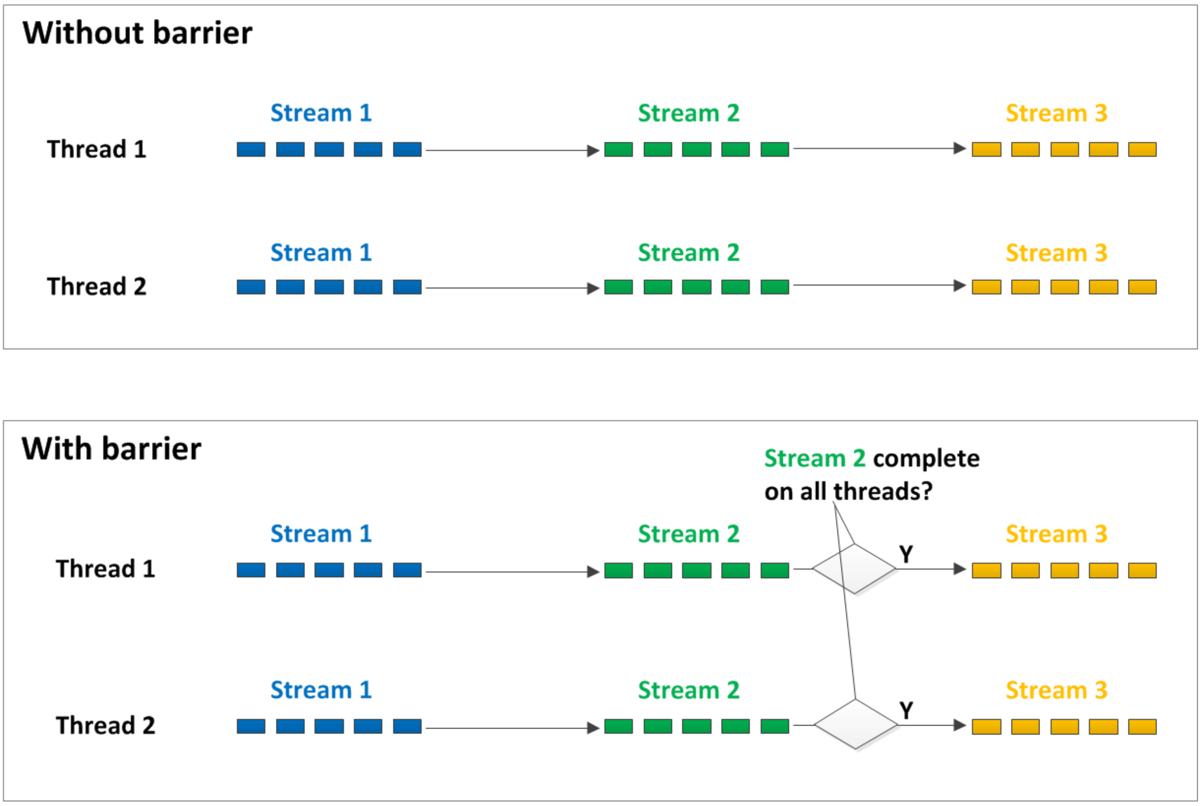


Figure 2.17: Stream Barrier

2.11.16 Tutorial: PCAP file to one stream

Goal
Load a stream template packet from a PCAP file instead of Scapy.

Assumption: The PCAP file contains only one packet. If the PCAP file contains more than one packet, this procedure loads only the first packet.

File
stl/udp_1pkt_pcap.py

```
    def get_streams (self, direction = 0, **kwargs):
        return [STLStream(packet =
                          STLPktBuilder(pkt ="stl/udp_64B_no_crc.pcap"), # path relative to ↩
                              pwd    ❶
                           mode = STLTXCont(pps=10)) ]
```

❶ Takes the packet from the specified PCAP file. The file location is relative to the directory in which you are running the script.

File
udp_1pkt_pcap_relative_path.py

```
    def get_streams (self, direction = 0, **kwargs):
        return [STLStream(packet = STLPktBuilder(pkt ="udp_64B_no_crc.pcap",
                                                 path_relative_to_profile = True), ❶
                         mode = STLTXCont(pps=10)) ]
```

1. Takes the packet from the PCAP file, relative to the directory of the **profile** file location.

2.11.17 Tutorial: Teredo tunnel (IPv6 over IPv4)

The following example demonstrates creating an IPv6 packet within an IPv4 packet, and creating a range of IP addresses.

File
stl/udp_1pkt_ipv6_in_ipv4.py

```
    def create_stream (self):
        # Teredo Ipv6 over Ipv4
        pkt =  Ether()/IP(src="16.0.0.1",dst="48.0.0.1")/
              UDP(dport=3797,sport=3544)/
              IPv6(dst="2001:0:4137:9350:8000:f12a:b9c8:2815",
                   src="2001:4860:0:2001::68")/
              UDP(dport=12,sport=1025)/ICMPv6Unknown()

        vm = STLScVmRaw( [
                            # tuple gen for inner Ipv6
                            STLVmTupleGen ( ip_min="16.0.0.1", ip_max="16.0.0.2",
                                            port_min=1025, port_max=65535,
                                            name="tuple"),                    (1)

                             STLVmWrFlowVar (fv_name="tuple.ip",
                                             pkt_offset= "IPv6.src",
                                             offset_fixup=12 ),               (2)
                             STLVmWrFlowVar (fv_name="tuple.port",
                                             pkt_offset= "UDP:1.sport" )      (3)
                          ]
                       )
```

1. Defines a stream struct called tuple with the following variables: `tuple.ip`, `tuple.port`
2. Writes a stream `tuple.ip` variable with an offset determined by the `IPv6.src` offset plus the `offset_fixup` of 12 bytes (only 4 LSB).
3. Writes a stream `tuple.port` variable into the second UDP header.

2.11.18 Tutorial: Mask instruction

STLVmWrMaskFlowVar is single-instruction-multiple-data Field Engine instruction. The pseudocode is as follows:

Pseudocode

```
        uint32_t val=(cast_to_size)rd_from_variable("name") # read flow-var
        val+=m_add_value                                    # add value

        if (m_shift>0) {                                    # shift
            val=val<<m_shift
        }else{
            if (m_shift<0) {
                val=val>>(-m_shift)
            }
        }

        pkt_val=rd_from_pkt(pkt_offset)                     # RMW
        pkt_val = (pkt_val & ~m_mask) | (val & m_mask)
        wr_to_pkt(pkt_offset,pkt_val)
```

Example 1

In this example, STLVmWrMaskFlowVar casts a stream variable with 2 bytes to be 1 byte.

```
vm = STLScVmRaw( [ STLVmFlowVar(name="mac_src",
                                min_value=1,
                                max_value=30,
                                size=2, op="dec",step=1),
                   STLVmWrMaskFlowVar(fv_name="mac_src",
                                      pkt_offset= 11,
                                      pkt_cast_size=1,
                                      mask=0xff) # mask command ->write it as one ↩
                                          byte
                 ]
               )
```

Example 2

In this example, STLVmWrMaskFlowVar shifts a variable by 8, which effectively multiplies by 256.

```
vm = STLScVmRaw( [ STLVmFlowVar(name="mac_src",
                                min_value=1,
                                max_value=30,
                                size=2, op="dec",step=1),
                   STLVmWrMaskFlowVar(fv_name="mac_src",
                                      pkt_offset= 10,
                                      pkt_cast_size=2,
                                      mask=0xff00,
                                      shift=8) # take the var shift it 8 (x256) ↩
                                          write only to LSB
                 ]
               )
```

Table 2.12: Output

value
0x0100
0x0200
0x0300

Example 3

In this example, STLVmWrMaskFlowVar generates the values shown in the table below as offset values for `pkt_offset`.

```
vm = STLScVmRaw( [ STLVmFlowVar(name="mac_src",
                                min_value=1,
                                max_value=30,
                                size=2,
                                op="dec",step=1),
                   STLVmWrMaskFlowVar(fv_name="mac_src",
                                      pkt_offset= 10,
                                      pkt_cast_size=1,
                                      mask=0x1,
                                      shift=-1)          ❶
                 ]
               )
```

1. Divides the value of `mac_src` by 2, and writes the LSB. For every two packets, the value written is changed.

Table 2.13: Output

value
0x00
0x00
0x01
0x01
0x00
0x00
0x01
0x01

2.11.19 Tutorial: Advanced traffic profile

Goal

- Define a different profile to operate in each traffic direction.
- Define a different profile for each port.
- Tune a profile tune by the arguments of tunables.

Every traffic profile must define the following function:

```
def get_streams (self, direction = 0, **kwargs)
```

`direction` is a mandatory field, required for any profile being loaded.

A profile can have any key-value pairs. Key-value pairs are called "tunables" and can be used to customize the profile.

The profile defines which tunables can be input to customize output.

Usage notes for defining parameters

- All parameters require default values.
- A profile must be loadable with no parameters specified.
- Every tunable must be expressed as key-value pair with a default value.
- All automatically provided values that are not tunables are defined by kwargs (Python keyword arguments).

For example, for the profile below, *pcap_with_vm.py*:

- The profile receives *direction* as a tunable and mandatory field.
- The profile defines 4 additional tunables.
- Automatic values such as *port_id*, which are not tunables, will be provided on kwargs.

File

stl/pcap_with_vm.py

```
def get_streams (self,
                 direction = 0,
                 ipg_usec = 10.0,
                 loop_count = 5,
                 ip_src_range = None,
                 ip_dst_range = {'start' : '10.0.0.1', 'end': '10.0.0.254'},
                 **kwargs)
```

Direction

: `direction` is a tunable that is always provided by the API/console when loading a profile, but it can be overridden by the user. It is used to make the traffic profile more usable - for example, as a bi-directional profile. However, the profile can ignore this parameter.

By default, `direction` is defined as the remainder of port_id divided by 2 (0 for even port_id, 1 for odd port_id).

```
def get_streams (self, direction = 0,**kwargs):
    if direction = 0:
        rate =100                                               (1)
    else:
        rate =200
    return [STLHltStream(tcp_src_port_mode = 'decrement',
                         tcp_src_port_count = 10,
                         tcp_src_port = 1234,
                         tcp_dst_port_mode = 'increment',
                         tcp_dst_port_count = 10,
                         tcp_dst_port = 1234,
                         name = 'test_tcp_ranges',
                         direction = direction,
                         rate_pps = rate,
                         ),
           ]
```

(1) Specifies different rates (100 and 200), based on direction.

```
$start -f ex1.py -a
```

For 4 interfaces:

- Interfaces 0 and 2: direction 0
- Interfaces 1 and 3: direction 1

The rate changes accordingly.

Customizing Profiles Using *'port_id'*

Keyword arguments (kwargs) provide default values that are passed along to the profile.

In the following, *port_id* (port ID for the profile) is a **kwarg. Using port_id, you can define a complex profile based on different ID of ports, providing a different profile for each port.

```
def create_streams (self, direction = 0, **args):

    port_id = args.get('port_id')

    if port_id == 0:
```

```
        return [STLHltStream(tcp_src_port_mode = 'decrement',
                             tcp_src_port_count = 10,
                             tcp_src_port = 1234,
                             tcp_dst_port_mode = 'increment',
                             tcp_dst_port_count = 10,
                             tcp_dst_port = 1234,
                             name = 'test_tcp_ranges',
                             direction = direction,
                             rate_pps = rate,
                             ),
               ]

    if port_id == 1:
        return STLHltStream(
                #enable_auto_detect_instrumentation = '1', # not supported yet
                ip_dst_addr = '192.168.1.3',
                ip_dst_count = '1',
                ip_dst_mode = 'increment',
                ip_dst_step = '0.0.0.1',
                ip_src_addr = '192.168.0.3',
                ip_src_count = '1',
                ip_src_mode = 'increment',
                ip_src_step = '0.0.0.1',
                l3_imix1_ratio = 7,
                l3_imix1_size = 70,
                l3_imix2_ratio = 4,
                l3_imix2_size = 570,
                l3_imix3_ratio = 1,
                l3_imix3_size = 1518,
                l3_protocol = 'ipv4',
                length_mode = 'imix',
                #mac_dst_mode = 'discovery', # not supported yet
                mac_src = '00.00.c0.a8.00.03',
                mac_src2 = '00.00.c0.a8.01.03',
                pkts_per_burst = '200000',
                rate_percent = '0.4',
                transmit_mode = 'continuous',
                vlan_id = '1',
                direction = direction,
                )

    if port_id = 3:
          ..
```

Full example using the TRex Console

The following command displays information about tunables for the pcap_with_vm.py traffic profile.

```
-=TRex Console v1.1=-

Type 'help' or '?' for supported actions

trex>profile -f stl/pcap_with_vm.py

Profile Information:


General Information:
Filename:         stl/pcap_with_vm.py
Stream count:          5
```

```
Specific Information:
Type:               Python Module
Tunables:           ['direction = 0', 'ip_src_range = None', 'loop_count = 5', 'ipg_usec = ↩
    10.0',
                     "ip_dst_range = {'start': '10.0.0.1', 'end': '10.0.0.254'}"]

trex>
```

You can specify values for tunables. The following command changes two values (ipg_usec and loop_count):

```
trex>start -f stl/pcap_with_vm.py -t ipg_usec=15.0,loop_count=25

Removing all streams from port(s) [0, 1, 2, 3]:             [SUCCESS]


Attaching 5 streams to port(s) [0]:                         [SUCCESS]


Attaching 5 streams to port(s) [1]:                         [SUCCESS]


Attaching 5 streams to port(s) [2]:                         [SUCCESS]


Attaching 5 streams to port(s) [3]:                         [SUCCESS]


Starting traffic on port(s) [0, 1, 2, 3]:                   [SUCCESS]

61.10 [ms]

trex>
```

The following command customizes these to different ports:

```
trex>start -f stl/pcap_with_vm.py --port 0 1 -t ipg_usec=15.0,loop_count=25#ipg_usec=100, ↩
    loop_count=300

Removing all streams from port(s) [0, 1]:                   [SUCCESS]


Attaching 5 streams to port(s) [0]:                         [SUCCESS]


Attaching 5 streams to port(s) [1]:                         [SUCCESS]


Starting traffic on port(s) [0, 1]:                         [SUCCESS]

51.00 [ms]

trex>
```

2.11.20 Tutorial: Per stream statistics

- Per stream statistics are implemented using hardware assist when possible (examples: Intel X710/XL710 NIC flow director rules).
- With other NICs (examples: Intel I350, 82599), per stream statistics are implemented in software.

- Implementation:
  - User chooses 32-bit packet group ID (pg_id) for each stream that requires statistics reporting. The same pg_id can be used for more than one stream. In this case, statistics for all streams with the same pg_id will be combined.
  - The IPv4 identification (or IPv6 flow label for IPv6 packet) field of the stream is changed to a value within the reserved range 0xff00 to 0xffff (0xff00 to 0xfffff for IPv6). Note that if a stream for which no statistics are needed has an IPv4 Id (or IPv6 flow label) in the reserved range, it is changed (the left bit becomes 0).
  - Software implementation: Hardware rules are used to direct packets from relevant streams to rx threads, where they are counted.
  - Hardware implementation: Hardware rules are inserted to count packets from relevant streams.
- Summed up statistics (per stream, per port) is sent using a ZMQ channel to clients upon request.

Limitations

- The feature supports only following packet types.
  - IPv4 over Ethernet.
  - IPv4 with one VLAN tag (except 82599 which does not support this type of packet).
  - IPv6 over Ethernet (except 82599 which does not support this type of packet).
  - IPv6 with one VLAN tag (except 82599 which does not support this type of packet).
  - Beginning with version 2.21, QinQ (two vlan tags) is supported if using "- - software" command line argument (details).
- Maximum number of concurrent streams (with different pg_id) on which statistics may be collected is described in the following table.

Table 2.14: Maximum concurrent streams for collecting flow statistics

NIC type	Max streams supported using HW filters	Using "- - software" (from version 2.23)
i350	255	1023
x710	127	1023
xl710	255	1023
82599	127	1023
Mellanox	127	127
virtio/vmxnet/other virtual NICs (Always working implicitly in "- - software" mode)	1023	1023

- On x710/xl710 cards, rx bytes counters (rx-bps, rx-bps-L1, ...) are not supported. This is because we use hardware counters which support only packets count on these cards.
  Starting from version 2.21, you can specify the "--no-hw-flow-stat" command line argument to make x710 behave like other cards, and count statistics in software. This enables RX byte count support, but limits the total rate of streams that can be counted.

Two examples follow, one using the console and the other using the Python API.

Console
The following simple traffic profile defines 2 streams and configures them with 2 different PG IDs.

File
stl/flow_stats.py

```
class STLS1(object):

    def get_streams (self, direction = 0):
        return [STLStream(packet = STLPktBuilder(pkt ="stl/udp_64B_no_crc.pcap"),
                          mode = STLTXCont(pps = 1000),
                          flow_stats = STLFlowStats(pg_id = 7)), ❶

                STLStream(packet = STLPktBuilder(pkt ="stl/udp_594B_no_crc.pcap"),
                          mode = STLTXCont(pps = 5000),
                          flow_stats = STLFlowStats(pg_id = 12)) ❷
               ]
```

❶ Assigned to PG ID 7.

❷ Assigned to PG ID 12.

The following command injects this to the console and uses the textual user interface (TUI) to display the TRex activity:

```
trex>start -f stl/flow_stats.py --port 0

Removing all streams from port(s) [0]:                         [SUCCESS]


Attaching 2 streams to port(s) [0]:                            [SUCCESS]


Starting traffic on port(s) [0]:                               [SUCCESS]

155.81 [ms]

trex>tui

Streams Statistics

   PG ID    |        12          |          7
 --------------------------------------------------
 Tx pps     |         5.00 Kpps  |        999.29 pps    #❶
 Tx bps L2  |        23.60 Mbps  |       479.66 Kbps
 Tx bps L1  |        24.40 Mbps  |       639.55 Kbps
 ---        |                    |
 Rx pps     |         5.00 Kpps  |        999.29 pps    #❷
 Rx bps     |               N/A  |               N/A    #❸
 ----       |                    |
 opackets   |            222496  |             44500    #❹
 ipackets   |            222496  |             44500
 obytes     |         131272640  |           2670000
 ibytes     |               N/A  |               N/A    #❺
 -----      |                    |
 opackets   |      222.50 Kpkts  |       44.50 Kpkts
 ipackets   |      222.50 Kpkts  |       44.50 Kpkts
 obytes     |         131.27 MB  |           2.67 MB
 ibytes     |               N/A  |               N/A    #❻
```

❶ Tx bandwidth of the streams matches the configured values.

❷ Rx bandwidth (999.29 pps) matches the Tx bandwidth (999.29 pps), indicating that there were no drops.

❸, ❺, ❻ RX byte count is not supported on this platform (no hardware support for byte count), so TRex displays N/A. You can add “--no-hw-flow-stat” command line argument to count everything in software, but max rate of streams that can be tracked will be lower.

❹ opackets/ipackets/obytes/ibytes appear twice, first with accurate number, and second time formatted.

Flow Stats Using The Python API

The Python API example uses the following traffic profile:

```
def rx_example (tx_port, rx_port, burst_size):

    # create client
    c = STLClient()

    try:
        pkt = STLPktBuilder(pkt = Ether()/IP(src="16.0.0.1",dst="48.0.0.1")/
                                  UDP(dport=12,sport=1025)/IP()/'a_payload_example')

        s1 = STLStream(name = 'rx',
                       packet = pkt,
                       flow_stats = STLFlowStats(pg_id = 5),     ❶
                       mode = STLTXSingleBurst(total_pkts = 5000,
                                               percentage = 80
                                               ))

        # connect to server
        c.connect()

        # prepare our ports - TX/RX
        c.reset(ports = [tx_port, rx_port])

        # add the stream to the TX port
        c.add_streams([s1], ports = [tx_port])

        # start and wait for completion
        c.start(ports = [tx_port])
        c.wait_on_traffic(ports = [tx_port])

        # fetch stats for PG ID 5
        flow_stats = c.get_stats()['flow_stats'].get(5)     ❷

        tx_pkts  = flow_stats['tx_pkts'].get(tx_port, 0)    ❸
        tx_bytes = flow_stats['tx_bytes'].get(tx_port, 0)   ❹
        rx_pkts  = flow_stats['rx_pkts'].get(rx_port, 0)    ❺
```

❶ Configures the stream to use PG ID 5.

❷, ❸, ❹, ❺ The structure of the object *'flow_stats'* is described below.

2.11.21 Tutorial: flow_stats object structure

The flow_stats object is a dictionary whose keys are the configured PG IDs. The next level is a dictionary containing *tx_pkts*, *tx_bytes*, *rx_pkts*, and *rx_bytes* (on supported HW). Each of these keys contains a dictionary of per port values.

The following shows a flow_stats object for 3 PG IDs after a specific run:

```
{
 5: {'rx_pkts'  : {0: 0, 1: 0, 2: 500000, 3: 0, 'total': 500000},
     'tx_bytes' : {0: 0, 1: 39500000, 2: 0, 3: 0, 'total': 39500000},
     'tx_pkts'  : {0: 0, 1: 500000, 2: 0, 3: 0, 'total': 500000}},

 7: {'rx_pkts'  : {0: 0, 1: 0, 2: 0, 3: 288, 'total': 288},
     'tx_bytes' : {0: 17280, 1: 0, 2: 0, 3: 0, 'total': 17280},
```

```
     'tx_pkts'  : {0: 288, 1: 0, 2: 0, 3: 0, 'total': 288}},

 12: {'rx_pkts' : {0: 0, 1: 0, 2: 0, 3: 1439, 'total': 1439},
      'tx_bytes': {0: 849600, 1: 0, 2: 0, 3: 0, 'total': 849600},
      'tx_pkts' : {0: 1440, 1: 0, 2: 0, 3: 0, 'total': 1440}}
}
```

2.11.22 Tutorial: Per stream latency/jitter/packet errors

- Per stream latency/jitter is implemented by software. This is an extension of the per stream statistics. Whenever you choose to display latency info for a stream, the statistics described in the "Per stream statistics" section are also available.
- Implementation:
  - Select a 32-bit packet group ID (pg_id) for each stream that requires latency reporting. The pg_id should be unique per stream.
  - The IPv4 identification field (or IPv6 flow label in case of IPv6 packet) of the stream is changed to a defined constant value (in the reserved range described in the "per stream statistics" section), to signal to the hardware to pass the stream to the software.
  - The last 16 bytes of the packet payload are used to pass needed information: ID of the stream, packet sequence number (per stream), timestamp of packet transmission.
- Gathered info (per stream) is sent to clients over a ZeroMQ messaging channel upon request.

  Limitations

- The feature supports only the following packet types. (Exception: When using the “--software” command line arg, all packet types are supported.)
  - IPv4 over Ethernet
  - IPv4 with one VLAN tag (except 82599 which does not support this type of packet)
  - IPv6 over Ethernet (except 82599 which does not support this type of packet)
  - IPv6 with one VLAN tag (except 82599 which does not support this type of packet)
- Packets must contain at least 16 bytes of payload.
- Each stream must have unique pg_id number. This also means that a given "latency collecting" stream can’t be transmitted from two interfaces in parallel (internally it means that there are two streams).
- Maximum number of concurrent streams (with different pg_id) on which latency info may be collected: 128 (in addition to the streams which collect per stream statistics)
- Global multiplier does not apply to this type of stream. Latency streams are processed by software, so multiplying them might inadvertently overwhelm the RX core. Consequently, if you have profile with 1 latency stream and 1 non-latency stream, and you change the traffic multipler, the latency stream keeps the same rate. To change the rate of a latency stream, manually edit the profile file. Usually this is not necessary because normally you stress the system using non latency stream, and (in parallel) measure latency using a constant-rate latency stream.

Important
Latency streams (transmit or receive) are not supported at full line rate like normal streams. This is a design feature intended to keep latency measurement accurate while preserving CPU resources. Typically, it is sufficient to have a low-rate latency stream. For example, if the required latency resolution is 10 usec, it is not necessary to send a latency stream at a speed higher than 100 KPPS. Typically, queues are built over time, so it is not possible that one packet will have latency and another packet in the same path will not have the same latency. The non-latency streams can be at full line rate, to load the DUT, while the low speed latency streams measure the latency of this path. Do not make the total rate of latency streams higher than 5 MPPS.

Two examples follow, using console and Python API.

Console
The following simple traffic profile defines 2 streams and configures them with 2 different PG IDs.

File
stl/flow_stats_latency.py

```
class STLS1(object):

    def get_streams (self, direction = 0):
        return [STLStream(packet = STLPktBuilder(pkt ="stl/udp_64B_no_crc.pcap"),
                          mode = STLTXCont(pps = 1000),
                          flow_stats = STLFlowLatencyStats(pg_id = 7)), ❶

                STLStream(packet = STLPktBuilder(pkt ="stl/udp_594B_no_crc.pcap"),
                          mode = STLTXCont(pps = 5000),
                          flow_stats = STLFlowLatencyStats(pg_id = 12)) ❷
               ]
```

❶ Assigned to PG ID 7, PPS would be **1000** regardless of the multplier.

❷ Assigned to PG ID 12, PPS would be **5000** regardless of the multplier.

The following command injects this to the console and uses the textual user interface (TUI) to display the TRex activity:

```
trex>start -f stl/flow_stats.py --port 0

trex>tui

Latency Statistics (usec)

   PG ID     |       7        |       12
 ----------------------------------------------
 Max latency  |              0 |               0 #❶
 Avg latency  |              5 |               5 #❷
 -- Window -- |                |
 Last (max)   |              3 |               4 #❸
 Last-1       |              3 |               3
 Last-2       |              4 |               4
 Last-3       |              4 |               3
 Last-4       |              4 |               4
 Last-5       |              3 |               4
 Last-6       |              4 |               3
 Last-7       |              4 |               3
 Last-8       |              4 |               4
 Last-9       |              4 |               3
 ---          |                |
 Jitter       |              0 |               0 #❹
 ----         |                |
 Errors       |              0 |               0 #❺
```

❶ Maximum latency measured over the stream lifetime (in usec).

❷ Average latency over the stream lifetime (usec).

❸ Maximum latency measured between last two data reads from server (currently reads every 0.5 second). Numbers below are maximum latency for previous measuring intervals, so the output displays the latency history for the last few seconds.

❹ Jitter of latency measurements.

5 Indication of number of errors (sum of seq_too_high and seq_too_low). You can see description in Python API doc below). In the future it will be possible to *zoom in* to see specific counters. For now, if you need to see specific counters, use the Python API.

Example using Python API:

Example File
stl_flow_latency_stats.py

```
stats = c.get_stats()

flow_stats = stats['flow_stats'].get(5)
lat_stats = stats['latency'].get(5)                         1


tx_pkts  = flow_stats['tx_pkts'].get(tx_port, 0)
tx_bytes = flow_stats['tx_bytes'].get(tx_port, 0)
rx_pkts  = flow_stats['rx_pkts'].get(rx_port, 0)
drops = lat_stats['err_cntrs']['dropped']
ooo = lat_stats['err_cntrs']['out_of_order']
dup = lat_stats['err_cntrs']['dup']
sth = lat_stats['err_cntrs']['seq_too_high']
stl = lat_stats['err_cntrs']['seq_too_low']
lat = lat_stats['latency']
jitter = lat['jitter']
avg = lat['average']
tot_max = lat['total_max']
last_max = lat['last_max']
hist = lat ['histogram']

# lat_stats will be in this format

latency_stats ==  {
     'err_cntrs':{                  # error counters 2
        u'dup':0,                   # Same sequence number was received twice in a row
        u'out_of_order':0,          # Packets received with sequence number too low (We ↩
             assume it is reorder)
        u'dropped':0                # Estimate of number of packets that were dropped ( ↩
            using seq number)
        u'seq_too_high':0,          # seq number too high events
        u'seq_too_low':0,           # seq number too low events
     },
     'latency':{
        'jitter':0,                 # in usec
        'average':15.2,             # average latency (usec)
        'last_max':0,               # last 0.5 sec window maximum latency (usec)
        'total_max':44,             # maximum latency (usec)
        'histogram':[               # histogram of latency
           {
              u'key':20,            # bucket counting packets with latency in the range ↩
                   20 to 30 usec
              u'val':489342         # number of samples that hit this bucket's range
           },
           {
              u'key':30,
              u'val':10512
           },
           {
              u'key':40,
              u'val':143
           },
```

```
                {
                   'key':0,              # bucket counting packets with latency in the range ↩
                        0 to 10 usec
                   'val':3
                }
             ]
          }
       },
```

1. Get the Latency dictionary.
2. For calculating packet error events, we add sequence number to each packet's payload. We decide what went wrong only according to sequence number of last packet received and that of the previous packet. *seq_too_low* and *seq_too_high* count events we see. *dup*, *out_of_order* and *dropped* are heuristics we apply to try and understand what happened. They will be accurate in common error scenarios. We describe few scenarios below to help understand this.

Error counters scenarios

- Scenario 1: Receive a packet with seq num 10, and another packet with seq num 10. We increment *dup* and *seq_too_low* by 1.
- Scenario 2: Receive a packet with seq num 10 and then a packet with seq num 15. We assume 4 packets were dropped, and increment *dropped* by 4, and *seq_too_high* by 1. We expect next packet to arrive with sequence number 16.
- Continuation of Scenario 2: Receive a packet with seq num 11. We increment *seq_too_low* by 1. We increment *out_of_order* by 1. We **decrement** *dropped* by 1. (The assumption is that one of the packets that was considered as dropped before has actually arrived out of order.)

2.11.23 Tutorial: HLT profiles

HLTAPI traffic_config() function has a set of arguments for creating streams.
It is possible to define TRex traffic profile using those arguments, which are converted under the hood to native Scapy/Field Engine instructions.
For limitations see here [altapi-support].

File
stl/hlt/hlt_udp_inc_dec_len_9k.py

```
class STLS1(object):
    '''
    Create 2 Eth/IP/UDP streams with different packet size:
    First stream will start from 64 bytes (default) and will increase until max_size  ↩
        (9,216)
    Seconds stream will decrease the packet size in reverse way
    '''

    def create_streams (self):
        max_size = 9*1024
        return [STLHltStream(length_mode = 'increment',
                             frame_size_max = max_size,
                             l3_protocol = 'ipv4',
                             ip_src_addr = '16.0.0.1',
                             ip_dst_addr = '48.0.0.1',
                             l4_protocol = 'udp',
                             udp_src_port = 1025,
                             udp_dst_port = 12,
                             rate_pps = 1,
```

```
                          ),
                 STLHltStream(length_mode = 'decrement',
                              frame_size_max = max_size,
                              l3_protocol = 'ipv4',
                              ip_src_addr = '16.0.0.1',
                              ip_dst_addr = '48.0.0.1',
                              l4_protocol = 'udp',
                              udp_src_port = 1025,
                              udp_dst_port = 12,
                              rate_pps = 1,
                              )
                ]

    def get_streams (self, direction = 0, **kwargs):
        return self.create_streams()
```

The following command, executed within a bash window, runs the traffic profile with the simulator to generate a PCAP file.

```
[bash]>./stl-sim -f stl/hlt/hlt_udp_inc_dec_len_9k.py -o b.pcap -l 10
```

The following commands, executed within a bash window, convert to native JSON or YAML.

```
[bash]>./stl-sim -f stl/hlt/hlt_udp_inc_dec_len_9k.py --json
```

```
[bash]>./stl-sim -f stl/hlt/hlt_udp_inc_dec_len_9k.py --yaml
```

Alternatively, use the following command to convert to a native Python profile.

```
[bash]>./stl-sim -f stl/hlt/hlt_udp_inc_dec_len_9k.py --native
```

Auto-generated code

```
# !!! Auto-generated code !!!
from trex_stl_lib.api import *

class STLS1(object):
    def get_streams(self):
        streams = []

        packet = (Ether(src='00:00:01:00:00:01', dst='00:00:00:00:00:00', type=2048) /
                  IP(proto=17, chksum=5882, len=9202, ihl=5L, id=0) /
                  UDP(dport=12, sport=1025, len=9182, chksum=55174) /
                  Raw(load='!' * 9174))
        vm = STLScVmRaw([CTRexVmDescFlowVar(name='pkt_len', size=2, op='inc',
                          init_value=64, min_value=64, max_value=9216, step=1),
                         CTRexVmDescTrimPktSize(fv_name='pkt_len'),
                         CTRexVmDescWrFlowVar(fv_name='pkt_len',
                         pkt_offset=16, add_val=-14, is_big=True),
                         CTRexVmDescWrFlowVar(fv_name='pkt_len',
                         pkt_offset=38, add_val=-34, is_big=True),
                         CTRexVmDescFixIpv4(offset=14)], split_by_field = 'pkt_len')
        stream = STLStream(packet = CScapyTRexPktBuilder(pkt = packet, vm = vm),
                           mode = STLTXCont(pps = 1.0))
        streams.append(stream)

        packet = (Ether(src='00:00:01:00:00:01', dst='00:00:00:00:00:00', type=2048) /
                  IP(proto=17, chksum=5882, len=9202, ihl=5L, id=0) /
                  UDP(dport=12, sport=1025, len=9182, chksum=55174) /
                  Raw(load='!' * 9174))
        vm = STLScVmRaw([CTRexVmDescFlowVar(name='pkt_len', size=2, op='dec',
                         init_value=9216, min_value=64,
```

```
                            max_value=9216, step=1),
                            CTRexVmDescTrimPktSize(fv_name='pkt_len'),
                            CTRexVmDescWrFlowVar(fv_name='pkt_len', pkt_offset=16,
                            add_val=-14, is_big=True),
                            CTRexVmDescWrFlowVar(fv_name='pkt_len',
                            pkt_offset=38, add_val=-34, is_big=True),
                            CTRexVmDescFixIpv4(offset=14)], split_by_field = 'pkt_len')
        stream = STLStream(packet = CScapyTRexPktBuilder(pkt = packet, vm = vm),
                           mode = STLTXCont(pps = 1.0))
        streams.append(stream)

        return streams

def register():
    return STLS1()
```

Use the following command within the TRex console to run the profile.

```
TRex>start -f stl/hlt/hlt_udp_inc_dec_len_9k.py -m 10mbps -a
```

2.11.24 Tutorial: Core pinning

Goal

Demonstrate how to assign a stream to a specific core. Core pinning was developed to avoid possible out of order for packets of the same stream.

The following example demonstrates 2 continuous streams S0 and S1 which are pinned to cores 0 and 1 respectively.

File

core_pinning_tutorial.py

Core pinning

```
    def create_stream (self):

        base_pkt =  Ether()/IP(src="55.55.1.1",dst="58.0.0.1")/UDP(dport=12,sport=1025)

        return STLProfile ([STLStream(name = 'S0',
                                      packet = STLPktBuilder(pkt = base_pkt),
                                      mode = STLTXCont(),
                                      core_id = 0), ❶
                            STLStream(name = 'S1',
                                      packet = STLPktBuilder(pkt = base_pkt),
                                      mode = STLTXCont(),
                                      core_id = 1) ❷
                            ]).get_streams()
```

❶ Creates a continuous stream named S0 which runs on core 0.

❷ Creates a continuous stream named S1 which runs on core 1.

Output

The following figure presents the output.

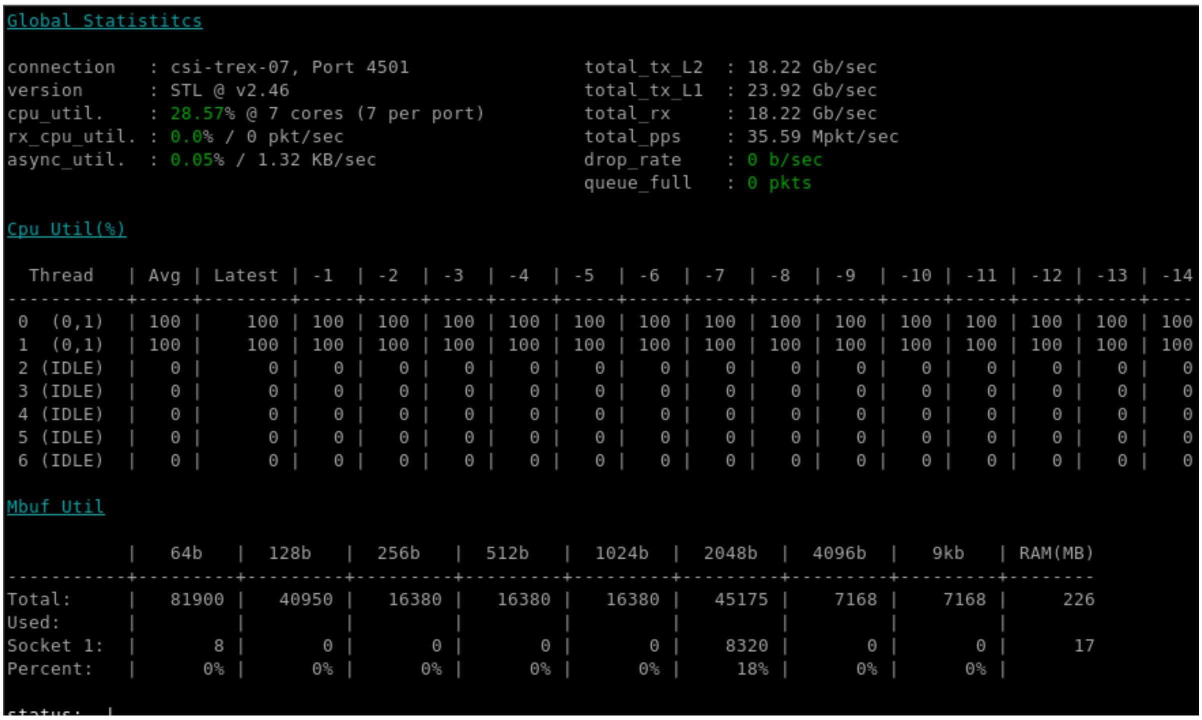


Figure 2.18: Core pinning

You need to run TRex with at least two cores for this tutorial to work.
You can define the number of cores with the -c flag:

```
[bash]>sudo ./t-rex-64 --no-sca -i -c 7 #❶
```

❶ Runs TRex with 7 cores.

To run the console use the following command:

```
[bash]>./trex-console -s <TRex hostname>
```

To run the traffic profile from console in TRex, use the following command:

```
trex>start -f automation/trex_control_plane/interactive/trex/examples/stl/ ↩
    core_pinning_tutorial.py -m 10gbps
```

To see the statistics use the following console command:

```
trex>stats -c
```

2.11.25 Tutorial: Field Engine variable split to cores

Goal
: Demonstrate how to split to cores a field engine variable. By default all the variables are split to cores and each core updates the variable by step * number of cores. In case we want each core to update the variable by step only, we set this parameter to false.

File
: split_var_to_cores.py

split_var_to_cores

```
def create_stream (self, direction):

    base_pkt =  Ether()/IP()/UDP()

    ip_range = {'src': {'start': "10.0.0.1", 'end': "10.0.0.254"},
                'dst': {'start': "8.0.0.1",  'end': "8.0.0.254"}}

    if (direction == 0):
        src = ip_range['src']
        dst = ip_range['dst']
    else:
        src = ip_range['dst']
        dst = ip_range['src']


    vm = STLVM()

    vm.var(name="src", min_value=src['start'], max_value=src['end'], size=4,
            op='inc', split_to_cores = False)                               #1
    vm.var(name="dst", min_value=dst['start'], max_value=dst['end'], size=4,
            op='inc')                                                       #2
    vm.repeatable_random_var(fv_name="src_port", size=2, min_value = 1024,
            max_value = 65535, limit=3, seed=0, split_to_cores = False)     #3
    vm.repeatable_random_var(fv_name="dst_port", size=2, min_value = 1024,
            max_value = 65535, limit=3, seed=0)
    vm.write(fv_name="src", pkt_offset='IP.src')
    vm.write(fv_name="dst", pkt_offset='IP.dst')
    vm.write(fv_name="src_port", pkt_offset="UDP.sport")
    vm.write(fv_name="dst_port", pkt_offset="UDP.dport")
    vm.fix_chksum(offset='IP')
    return STLStream(packet = STLPktBuilder(pkt = base_pkt, vm = vm),
                     mode = STLTXCont())
```

1. The source address variable is not split to cores, hence it will be incremented by step = 1 in each core.
2. By default the destination address variable is split to cores, as such it will be incremented by step * number of cores in each core.
3. Repeatable random variables can also be split to cores. The source port is such a variable and hence it will change once in number of cores.

You can simulate the profile using the STL simulator:

```
[bash]>./stl-sim -f stl/split_var_to_cores.py -o b.pcap -c 7
```

In the pcap we can see that each source address is repeated number of cores times, meanwhile the destination addresses are repeated once. The same holds for the source and destination port.

2.11.26 Tutorial: Field Engine dependent variables

Sometimes we would like to have a type of dependency between our field engine variables. For example, we want one variable to perform its op only if another variable finished wrapping around. A classical case would be getting all the combinations of two letters aa, ab, . . . , az, ba, . . . , bz, . . . , zz. We want the first letter to update only after we got all the possible values of the second letter. For such cases, we implemented the next_var feature.

Goal

Demonstrate the next_var feature and how to use it.

File
dependent_field_engine_vars.py

Dependent Field Engine Variables

```
def create_stream (self):

    base_pkt = Ether()/IP()/UDP(dport=12, sport=1025)/ (24 * 'x')

    vm = STLVM()
    vm.var(name='IP_src', min_value=None, max_value=None, size=4, op='dec', step=1,
           value_list = ['16.0.0.1', '10.0.0.1', '14.0.0.1'], next_var='var1')      ❶
    vm.write(fv_name='IP_src', pkt_offset='IP.src')
    vm.var(name='var1', min_value=ord('a'), max_value=ord('c'), size=1, step=1,
           op='inc', next_var='var2')                                              ❷
    vm.write(fv_name='var1', pkt_offset='Raw.load', offset_fixup=3)
    vm.repeatable_random_var(fv_name='var2', min_value=ord('a'), max_value=ord('z'),
                             size=1, limit=4, seed=0, next_var='var3')             ❸
    vm.write(fv_name='var2', pkt_offset='Raw.load', offset_fixup=2)
    vm.var(name='var3', min_value=ord('a'), max_value=ord('b'), size=1, step=1,
           op='inc', split_to_cores=False)                                         ❹
    vm.write(fv_name='var3', pkt_offset='Raw.load', offset_fixup=1)
    vm.fix_chksum()

    return STLStream(packet = STLPktBuilder(pkt = base_pkt, vm = vm),
                     mode = STLTXCont())
```

❶ The *next_var* can be used in any type of variable whose wrap around is defined, such an example is a value list. *var1* will increment only after a wrap around of *IP_src*.

❷ *var2* will increment only after *var1* performs a full wrap around.

❸ Repeatable random variables also support the *next_var* feature as the wrap around for such a variable is defined. In case you want to use the *next_var* with randomness, define a repeatable random and don't use *op='random'* for a regular variable.

❹ A variable that has some other variable pointing at it must run on a single core. This is implemented as the default action, hence no need to explicitly set *split_to_cores=False*.

You can simulate the profile using the STL simulator:

```
[bash]>./stl-sim -f stl/dependent_field_engine_vars.py -o b.pcap
```

2.12 Dynamic multiple profiles

TRex can support multiple profiles operations on the same port similar to virtual interfaces (from v2.57)

In the console, profile ids can be specified via -p argument.
The expression in TRex console allows -p <port number>.<profile id> instead of port number. This is done for backward compatibility reasons
For example, start imix.py profile on port *0.imix* and udp_1pkt.py profile on port *0.udp1pkt*.
In this way, *imix* and *udp1pkt* traffic can be running simultaneously and independently on port 0.

Example

```
# start(add) dynamic profile on port 0.imix(3 streams are running on IMIX)
trex> start -f stl/imix.py -p 0.imix

# start(add) dynamic profile on port 0.udp1pkt
trex> start -f stl/udp_1pkt.py -p 0.udp1pkt

# start(add) dynamic profile on port 1.imix(3 streams are running on IMIX)
trex> start -f stl/imix.py -p 1.imix

# start(add) dynamic profile on port 1.udp1pkt
trex> start -f stl/udp_1pkt.py -p 1.udp1pkt

# show all streams
trex> streams
Port 0:

    ID     |    profile      | length  |       rate        |
-----------+----------------+---------+------------------+-
    1      |     imix        |      64 |      28 pps       |
    2      |     imix        |     594 |      16 pps       |
    3      |     imix        |    1518 |       4 pps       |
    4      |    udp1pkt      |      64 |       1 pps       |

Port 1:

    ID     |    profile      |  length  |    rate        |
-----------+----------------+----------+---------------+--
    1      |     imix        |       64 |   28 pps       |
    2      |     imix        |      594 |   16 pps       |
    3      |     imix        |     1518 |    4 pps       |
    4      |    udp1pkt      |       64 |    1 pps       |


# change packet rate of udp1pkt during run-time
trex> update -p 0.udp1pkt -m 1gbps

# pause traffic on port 0.udp1pkt (traffic on 0.imix is still running
trex> pause -p 0.udp1pkt

# show all profiles
trex> profiles
Port 0:

  Profile ID    |   state    | stream ID
----------------+------------+-----------
    udp1pkt     |   PAUSE    |    [4]
     imix       |     TX     | [1, 2, 3]

Port 1:

  Profile ID    |   state    | stream ID
----------------+------------+-----------
    udp1pkt     |     TX     |    [4]
     imix       |     TX     | [1, 2, 3]

# resume every paused traffic on port 0 (including 0.udp1pkt)
trex> resume -p 0.*

Resume traffic on port(s) [0.udp1pkt]:
[SUCCESS]
```

```
# stop traffic on 0.imix (traffic on 0.udp1pkt is still running)
trex> stop -p 0.imix

# stop all traffic on port 0
trex> stop -p 0.*

# stop and remove all traffic on port 1
trex> stop -p 1.* --remove

# show all streams
trex>streams
Port 0:

    ID      |    profile      |  length  |        mode        |       rate        |
-----------+----------------+----------+-------------------+------------------+-
    1       |     imix        |       64 |    Continuous      |     28 pps        |
    2       |     imix        |      594 |    Continuous      |     16 pps        |
    3       |     imix        |     1518 |    Continuous      |      4 pps        |
    4       |    udp1pkt      |       64 |    Continuous      |      1 pps        |
```

From API perspective it is the same expect the port id that could include the profile id.

Some exceptions:

1. pgid should be unique, it is not possible to run flow-stats with the same id on different profiles
2. push pcap remote can't be run in parallel with another profile

Dynmaic profile API example

```
    def dynamic_profile (self):

        port_list = [self.tx_port, self.rx_port]
        profile_list = ["p1", "p2"]
        stream_pg_id = 0
        port = 0

        try:
                # start profiles 0.p1 0.p2 (two profiles on all ports)
                for profile in profile_list:

                    stream_pg_id = stream_pg_id + 1
                    s1 = STLStream(name = 'latency',
                                   packet = self.pkt,
                                   mode = STLTXCont(percentage = self.percentage),
                                   flow_stats = STLFlowLatencyStats(pg_id = stream_pg_id))

                    port_profile = str(port)  + "." + str(profile) #e.g 0.p1

                    self.c.add_streams([s1], ports = port_profile)
                    self.c.start(ports = port_profile)


                # stop all profiles on port 0 using 0.*
                self.c.stop(ports = str(port)+".*")

        except STLError as e:
            assert False , '{0}'.format(e)
```

2.13 Functional Tutorials

On functional tests we demonstrate a way to test certain cases which does not require high bandwidth but instead require more flexibility such as fetching all the packets on the RX side.

2.13.1 Tutorial: Testing Dot1Q VLAN tagging

Goal
Generate a Dot1Q packet with a vlan tag and verify the returned packet is on the same vlan.

File
stl_functional.py

The following example is presented here in a simplified form. See the file above for the full working example.

```
#passed a connected client object and two ports
def test_dot1q (c, rx_port, tx_port):

    # activate service mode on RX code
    c.set_service_mode(ports = rx_port)

    # generate a simple Dot1Q
    pkt = Ether() / Dot1Q(vlan = 100) / IP()

    # start a capture
    capture = c.start_capture(rx_ports = rx_port)

    # push the Dot1Q packet to TX port... we need 'force' because this is under service ↩
       mode
    print('\nSending 1 Dot1Q packet(s) on port {}'.format(tx_port))

    c.push_packets(ports = tx_port, pkts = pkt, force = True)
    c.wait_on_traffic(ports = tx_port)

    rx_pkts = []
    c.stop_capture(capture_id = capture['id'], output = rx_pkts)

    print('\nRecived {} packets on port {}:\n'.format(len(rx_pkts), rx_port))

    c.set_service_mode(ports = rx_port, enabled = False)

    # got back one packet
    assert(len(rx_pkts) == 1)
    rx_scapy_pkt = Ether(rx_pkts[0]['binary'])

    # it's a Dot1Q with the same VLAN
    assert('Dot1Q' in rx_scapy_pkt)
    assert(rx_scapy_pkt.vlan == 100)


    rx_scapy_pkt.show2()
```

2.13.2 Tutorial: Testing IPv4 ping - echo request / echo reply

Goal
Generate a ICMP echo request from one interface to another one and validate the response.

File
    stl_functional.py

```
# test a echo request / echo reply
def test_ping (c, tx_port, rx_port):

    # activate service mode on RX code
    c.set_service_mode(ports = [tx_port, rx_port])

    # fetch the config
    tx_port_attr = c.get_port_attr(port = tx_port)
    rx_port_attr = c.get_port_attr(port = rx_port)

    assert(tx_port_attr['layer_mode'] == 'IPv4')
    assert(rx_port_attr['layer_mode'] == 'IPv4')

    pkt = Ether() / IP(src = tx_port_attr['src_ipv4'], dst = rx_port_attr['src_ipv4']) / ↩
        ICMP(type = 8)

    # start a capture on the sending port
    capture = c.start_capture(rx_ports = tx_port)

    print('\nSending ping request on port {}'.format(tx_port))

    # send the ping packet
    c.push_packets(ports = tx_port, pkts = pkt, force = True)
    c.wait_on_traffic(ports = tx_port)

    # fetch the packet
    rx_pkts = []
    c.stop_capture(capture_id = capture['id'], output = rx_pkts)

    print('\nRecived {} packets on port {}:\n'.format(len(rx_pkts), tx_port))

    c.set_service_mode(ports = rx_port, enabled = False)

    # got back one packet
    assert(len(rx_pkts) == 1)
    rx_scapy_pkt = Ether(rx_pkts[0]['binary'])

    # check for ICMP reply
    assert('ICMP' in rx_scapy_pkt)
    assert(rx_scapy_pkt['ICMP'].type == 0)

    rx_scapy_pkt.show2()
```

2.14 Services

Important
The following section relies on *service mode* - please refer to *service mode* section for more details

2.14.1 Overview

While under *service mode*, TRex provides the ability to run *services*.

A *service* is an instance of a *service type* that has a certain request / response state machine.

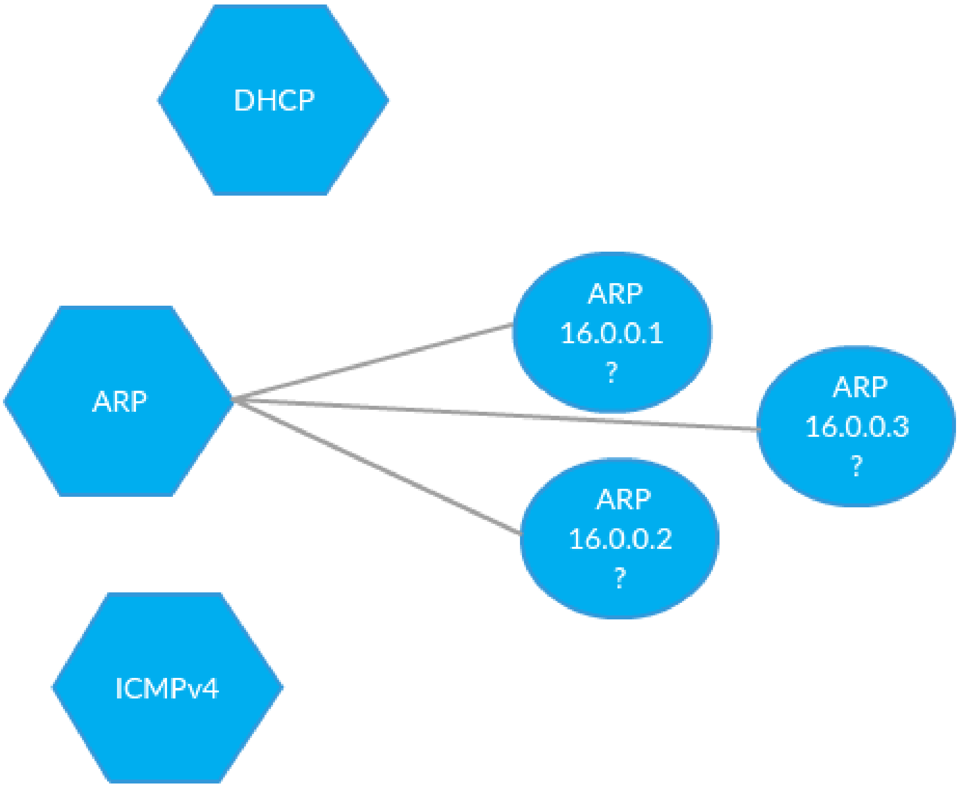


Figure 2.19: Services Instances

For example, the **ARP** service type provides a way to create ARP request instances that can be then executed by TRex in a **parallel** way supporting up to ~1000 requests in parallel.

The following diagram illustrates how *services* fit in the general flow:

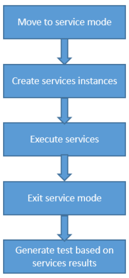


Figure 2.20: Services Execution Flow

Note
A simple example

The simplest example of a service execution:

```
# import the ARP service
from trex_stl_lib.services.trex_stl_service_arp import STLServiceARP

# create a service context on port 0
ctx  = client.create_service_ctx(port = 0)

# generate single ARP request from 1.1.1.5 to 1.1.1.1
arp = STLServiceARP(ctx, src_ip = '1.1.1.5', dst_ip = '1.1.1.1')

# move to service mode and execute service
client.set_service_mode(ports = 0)
try:
    ctx.run(arp)
finally:
    client.set_service_mode(ports = 0, enabled = False)

# show the ARP result
print(arp.get_record())
```

```
Recieved ARP reply from: 1.1.1.1, hw: b8:46:dd:63:21:e4"
```

There are two main usages for services:

- **Customizing Tests**
- **Control Plane Stress**

2.14.2 Customizing Tests

Services provides an easy way to customize tests:

executing services can be used to dynamically acquire data prior to the test and then generate a test based on the results.

> **Note**
> An example of using DHCP service to cutomize a test

Let's assume that our topology includes a DHCP server which will allow traffic from previously leased addreses only. Without services we will not be able to statically generate a test that will be accepted by the server.

However, with services we can generate clients using the DHCP service type and used the leased addresses to generate traffic.

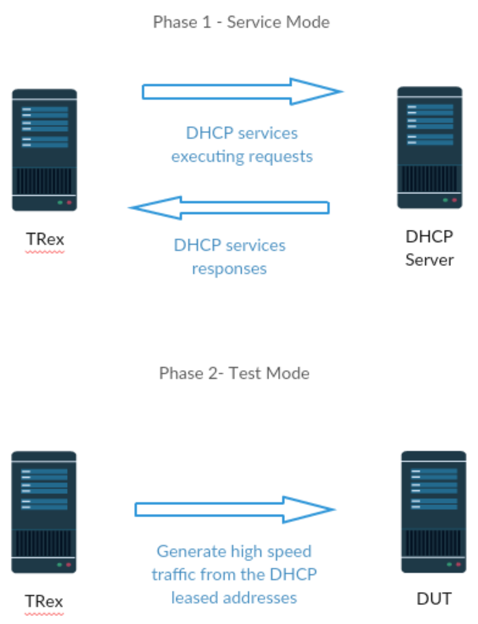


Figure 2.21: Services Two Phase Based Test

Let’s take a deep dive into how to use Python API to implement the above example:

```
# first we import the relevant service
from trex_stl_lib.services.trex_stl_service_dhcp import STLServiceDHCP

# next we generate a service context on the required port
# all services will be executed on the same port - there is no cross-port service execution
ctx  = client.create_service_ctx(port = 0)

# generate 100 clients from random MACs (random MAC function omitted)
# you can, of course, supply specific MAC addresses
dhcps = [STLServiceDHCP(mac = random_mac()) for _ in range(100)]

# now we execute the service context under service mode

client.set_service_mode(ports = 0)
try:
    ctx.run(dhcps)
finally:
    client.set_service_mode(ports = 0, enabled = False)


# inspect the DHCP execution result
for dhcp in dhcps:
    record = dhcp.get_record()
    print('client: MAC {0} - DHCP: {1}'.format(dhcp.get_mac(),record))

# let's filter all the DHCPs that successfuly moved to 'BOUND' state
# refer to the DHCP code reference to see all the available states
bounded_dhcps = [dhcp for dhcp in dhcps if dhcp.state == 'BOUND']
```

```
# we can use the above results to generate traffic from the leased addresses

streams = []
for bound_dhcp in bounded_dhcps:
    record = bound_dhcp.get_record()

    pkt = STLPktBuilder(pkt = Ether(src=record.client_mac)/
                              IP(src=record.client_ip,dst=record.server_ip)/
                              UDP)
    streams.append(STLStream(packet = pkt, mode = STLTXSingleBurst(total_pkts = 1000)))

# add streams and generate traffic
client.add_streams(ports = 0, streams = streams)
client.start(ports = 0, mult = '100%')
client.wait_on_traffic()
```

And here is how the output (partial) looks like:

```
client: MAC 3c:1d:08:91:7f:34 - DHCP: ip: 1.1.1.8,  server_ip: 1.1.1.1, subnet: ↩
    255.255.255.0
client: MAC 21:3c:a3:3f:cb:a7 - DHCP: ip: 1.1.1.5,  server_ip: 1.1.1.1, subnet: ↩
    255.255.255.0
client: MAC f9:ba:11:51:91:8b - DHCP: ip: 1.1.1.7,  server_ip: 1.1.1.1, subnet: ↩
    255.255.255.0
client: MAC b8:46:dd:63:21:e4 - DHCP: ip: 1.1.1.11, server_ip: 1.1.1.1, subnet: ↩
    255.255.255.0
client: MAC b8:38:f9:c7:1c:6e - DHCP: ip: 1.1.1.9,  server_ip: 1.1.1.1, subnet: ↩
    255.255.255.0
client: MAC 44:27:f1:f3:9a:bd - DHCP: ip: 1.1.1.10, server_ip: 1.1.1.1, subnet: ↩
    255.255.255.0
client: MAC cd:8d:c6:c9:5c:6a - DHCP: ip: 1.1.1.2,  server_ip: 1.1.1.1, subnet: ↩
    255.255.255.0
client: MAC 51:ee:33:d9:d8:9f - DHCP: ip: 1.1.1.3,  server_ip: 1.1.1.1, subnet: ↩
    255.255.255.0
client: MAC 75:f2:22:ce:86:47 - DHCP: ip: 1.1.1.4,  server_ip: 1.1.1.1, subnet: ↩
    255.255.255.0
client: MAC 19:bb:56:20:52:3b - DHCP: ip: 1.1.1.6,  server_ip: 1.1.1.1, subnet: ↩
    255.255.255.0
```

Note
An example of using IPv6 ND to establish IPv6 neighborships before running DP tests.

The IPv6 ND service has many options, most of them are not mandatory: mandatory parameters:

- ctx: the service context object
- dst_ip: the IPv6 neighbor address to be resolved
- src_ip: the IPv6 source address to be used

optional parameters:

- retries: number of retries in case of timeouts (default=1)
- src_mac: source mac address to be used in Ethernet packets (default taken from port in use)
- timeout: timeout in seconds to wait for neighbor advertisements in response to our neighbor solicitation packets (default 2)

- verify_timeout: timeout in seconds to wait for neighbor soliciation messages from a neighbor, after our NS was answered (Neighbor verification is not always performed, but depends on **our** state in the neighbors ND cache).
- vlan: vlan identifiers used for dot1q/dot1ad vlan headers (e.g. [200,2] uses outer vlan 200, inner vlan 2)
- fmt: encapsulation format used for vlan tagging (*Q*: dot1q, *D*: dot1ad). Double tagging can be formatted with "QQ" (double-dot1q) or "DQ" (dot1q in dot1ad), or *DD* or *QD* . . . .
- verbose_level: increase logging of IPv6 service instances (e.g service_level = STLServiceIPv6ND.ERROR)

```
#!/usr/bin/python

from stl_path import *
from trex_stl_lib.api import *
from trex_stl_lib.services.trex_stl_service_IPv6ND import STLServiceIPv6ND

c = STLClient()
c.connect()
c.acquire(force = True)
c.set_service_mode(ports = 0)

# create service context
ctx  = c.create_service_ctx(port = 0)

nd_service = STLServiceIPv6ND(  ctx,
                                src_ip = "2001:db8:10:22::15",
                                dst_ip = "2001:db8:10:22::1",
                                vlan = [ 500, 22],
                                timeout=2,
                                verify_timeout=6,
                                fmt = "QQ",
                                verbose_level = STLServiceIPv6ND.INFO

                                    )
ctx.run(nd_service)
print nd_service.get_record()
```

2.14.3 Control Plane Stress Tests

Another practical use-case of services is to simply use the first phase as the main phase and focus on generating many control plane requests.

For example, the same DHCP example can be used to stress out a DHCP server by generating many requests.

Now, even though service mode is **slower** that regular mode, and service context execution is even slower as we wait for response from the server there are still two major benefits:

- **Parallelism** - When generating many service instances, there will be minimum impact on the total run time as we execute services in parallel
- **Flexibility** - Putting aside performance, TRex services are written in Python and uses Scapy to generate traffic and thus are very easy to manipulate and custom fit

2.14.4 Currently Provided Services

Currently, the implemented services provided with TRex package are:

- **ARP** - provides an ARP resolution for an IPv4 address

- **ICMPv4** - provides Ping IPv4 for an IPv4 address
- **DHCP** - provides a DHCP bound/release lease address
- **IPv6ND** - provides IPv6 neighbor discovery

We are planning to add more and hope for contribution in this area

2.14.5 A Detailed DHCP Example

Full DHCP example can be found under the following GitHub link:

- stl_dhcp_example.py

2.14.6 Limitations

There is no limitation on the **types** of services that are being executed. It is possible to run *ARP* and *DHCP* in **parallel** if it is needed.

The only limitation is that *services* run under context which is bounded to a single port.

There is no way to forward response from another port to the context.

Also, the number of service instances per execution is currently limited to **1000**.

2.14.7 Console plugins

Another usage of services (or even mix of them) is plugins infrastructure in trex-console.
Plugins system is a way to dynamically import and run some code.

```
trex>plugins -h
usage: plugins [-h]  ...

Show / load / use plugins

optional arguments:
  -h, --help  show this help message and exit

command:

    show      Show / search for plugins
    load      Load (or implicitly reload) plugin by name
    unload    Unload plugin by name
```

Plugins are located in console/plugins directory, and their filename begins with "plugin_".
They can be searched via "show" command and loaded via "load" command:

```
trex>plugins load wlc

Loading plugin: wlc                                          [SUCCESS]

trex>plugins show
+-----------------------+------------------+
|      Plugin name      |      Loaded      |
+=======================+==================+
|        IPv6ND         |        No        |
+-----------------------+------------------+
|        hello          |        No        |
+-----------------------+------------------+
|          wlc          |        Yes       |
+-----------------------+------------------+
```

Now, loaded plugin can be seen in menu of plugins and used:

```
trex>plugins -h
usage: plugins [-h]  ...

Show / load / use plugins

optional arguments:
  -h, --help  show this help message and exit

command:

    show      Show / search for plugins
    load      Load (or implicitly reload) plugin by name
    unload    Unload plugin by name
    wlc       WLC testing related functionality

trex>plugins wlc -h
usage: plugins wlc [-h]
                   {add_client,base,close,create_ap,reconnect,show,start} ...

optional arguments:
  -h, --help            show this help message and exit

commands:
  {add_client,base,close,create_ap,reconnect,show,start}
    add_client          Add client(s) to AP(s)
    base                Set base values of MAC, IP etc. for created AP/Client.
                        Will be increased for each new device.
    close               Closes all wlc-related stuff
    create_ap           Create AP(s) on port
    reconnect           Reconnect disconnected AP(s) or Client(s).
    show                Show status of APs
    start               Start traffic on behalf on client(s).

trex>plugins wlc create_ap -p 0

Enabling service mode on port(s) [0]:                        [SUCCESS]


Discovering WLC                                              [SUCCESS]


Establishing DTLS connection                                 [SUCCESS]


Join WLC and get SSID                                        [SUCCESS]
```

Example of plugin (file console/plugins/plugin_hello.py):

```
#!/usr/bin/python

from console.plugins import *

'''
Example plugin
'''

class Hello_Plugin(ConsolePlugin):
    def plugin_description(self):
        return 'Simple example'
```

```
    # used to init stuff
    def plugin_load(self):
        # Adding arguments to be used at do_* functions
        self.add_argument(type = str,
                dest = 'username', # <----- variable name to be used
                help = 'Username to greet')

    # We build argparser from do_* functions, stripping the "do_" from name
    def do_greet(self, username): # <------ username was registered in plugin_load
        '''Greet some username'''
        self.trex_client.logger.log('Hello, %s!' % bold(username.capitalize())) # <---  ↩
            trex_client is set implicitly
```

Note
An plugin that uses the IPv6 service, that allows experimenting with IPv6 from the console.

```
trex(service)>plugins load IPv6ND

Loading plugin: IPv6ND                                      [SUCCESS]

trex(service)>plugins IPv6ND -h
usage: plugins IPv6ND [-h] {clear,resolve,status} ...

optional arguments:
  -h, --help            show this help message and exit

commands:
  {clear,resolve,status}
    clear               clear IPv6 ND requests/entries
    resolve             perform IPv6 neighbor discovery
    status              show status of generated ND requests



 trex(service)>plugins IPv6ND  resolve -h
 usage: IPv6ND resolve [-h] -p PORT -s SRC_IP [-m SRC_MAC] -d DST_IP [-v VLAN]
                       [-f FMT] [-t TIMEOUT] [-T VERIFY_TIMEOUT] [-c COUNT]
                       [-r RATE] [-R RETRIES] [-V]

 perform IPv6 neighbor discovery

 optional arguments:
   -h, --help            show this help message and exit
   -p PORT, --port PORT  trex port to use
   -s SRC_IP, --src-ip SRC_IP
                         src ip to use
   -m SRC_MAC, --src-mac SRC_MAC
                         src mac to use
   -d DST_IP, --dst-ip DST_IP
                         IPv6 dst ip to discover
   -v VLAN, --vlan VLAN  vlan(s) to use (comma separated)
   -f FMT, --format FMT  vlan encapsulation to use (QQ: qinq, DA: 802.1AD ->
                         802.1q)
   -t TIMEOUT, --timeout TIMEOUT
                         timeout to wait for NA
   -T VERIFY_TIMEOUT, --verify-timeout VERIFY_TIMEOUT
                         timeout to wait for neighbor verification NS
   -c COUNT, --count COUNT
                         nr of nd to perform (auto-scale src-addr to test
```

```
                        parallel NDs)
  -r RATE, --rate RATE  rate limiter value to pass to services framework
  -R RETRIES, --retries RETRIES
                        number of retries in case no answer was received
  -V, --verbose         verbose mode
```

Perform 3 parallel ND operations, where source IPv6 addresses are incremented automatically:

```
trex(service)>plugins IPv6ND  resolve -c 3 -v 500,22 -p 0 -s 2001:db8:10:22::70 -d 2001: ↩
    db8:10:22::1 -V --format QQ -T 6

performing ND for 3 addresses.
NA response timeout...........: 2s
Neighbor verification timeout..: 6s

ND: TX NS: 2001:db8:10:22::70,74:a0:2f:b4:97:49 -> 2001:db8:10:22::1 (retry 0)
ND: TX NS: 2001:db8:10:22::71,74:a0:2f:b4:97:49 -> 2001:db8:10:22::1 (retry 0)
ND: TX NS: 2001:db8:10:22::72,74:a0:2f:b4:97:49 -> 2001:db8:10:22::1 (retry 0)
ND: RX NA: 2001:db8:10:22::70 <- 2001:db8:10:22::1, 00:05:73:a0:00:01
ND: RX NA: 2001:db8:10:22::72 <- 2001:db8:10:22::1, 00:05:73:a0:00:01
ND: timeout for 2001:db8:10:22::71,74:a0:2f:b4:97:49 <-- 2001:db8:10:22::1 (retry 0)
ND: TX NS: 2001:db8:10:22::71,74:a0:2f:b4:97:49 -> 2001:db8:10:22::1 (retry 1)
ND: RX NA: 2001:db8:10:22::71 <- 2001:db8:10:22::1, 00:05:73:a0:00:01
ND: RX NS: 2001:db8:10:22::70 <-- 2001:db8:10:22::d,00:de:fb:1d:83:c4
ND: TX NA: 2001:db8:10:22::70,74:a0:2f:b4:97:49 -> 2001:db8:10:22::d,00:de:fb:1d:83:c4
ND: RX NS: 2001:db8:10:22::70 <-- 2001:db8:10:22::c,00:de:fb:1d:84:c5
ND: TX NA: 2001:db8:10:22::70,74:a0:2f:b4:97:49 -> 2001:db8:10:22::c,00:de:fb:1d:84:c5
ND: RX NS: 2001:db8:10:22::71 <-- 2001:db8:10:22::c,00:de:fb:1d:84:c5
ND: TX NA: 2001:db8:10:22::71,74:a0:2f:b4:97:49 -> 2001:db8:10:22::c,00:de:fb:1d:84:c5
ND: RX NS: 2001:db8:10:22::71 <-- 2001:db8:10:22::d,00:de:fb:1d:83:c4
ND: TX NA: 2001:db8:10:22::71,74:a0:2f:b4:97:49 -> 2001:db8:10:22::d,00:de:fb:1d:83:c4
ND: RX NS: 2001:db8:10:22::72 <-- 2001:db8:10:22::d,00:de:fb:1d:83:c4
ND: TX NA: 2001:db8:10:22::72,74:a0:2f:b4:97:49 -> 2001:db8:10:22::d,00:de:fb:1d:83:c4
ND: RX NS: 2001:db8:10:22::72 <-- 2001:db8:10:22::c,00:de:fb:1d:84:c5
ND: TX NA: 2001:db8:10:22::72,74:a0:2f:b4:97:49 -> 2001:db8:10:22::c,00:de:fb:1d:84:c5

trex(service)>
```

Show status of local IPv6 neighborships:

```
trex(service)>plugins IPv6ND  status


ND Status
---------

used vlan(s)...................: [500, 22]
used encapsulation.............: QQ
number of IPv6 source addresses: 3


    SRC MAC              SRC IPv6                 |       DST IPv6                   DST ↩
         MAC       STATE         VERIFIED
---------------------------------------------------------------------------------------------------

74:a0:2f:b4:97:49  2001:db8:10:22::70             | 2001:db8:10:22::1              ↩
    00:05:73:a0:00:01 REACHABLE        2
74:a0:2f:b4:97:49  2001:db8:10:22::71             | 2001:db8:10:22::1              ↩
    00:05:73:a0:00:01 REACHABLE        2
74:a0:2f:b4:97:49  2001:db8:10:22::72             | 2001:db8:10:22::1              ↩
    00:05:73:a0:00:01 REACHABLE        2
```

```
resolved..: 3
unresolved: 0
verified..: 3

trex(service)>
```

Clear local IPv6 neighborships:

```
trex(service)>plugins IPv6ND clear
trex(service)>plugins IPv6ND status


ND Status
---------

used vlan(s)...................: [500, 22]
used encapsulation.............: QQ
number of IPv6 source addresses: 3


    SRC MAC              SRC IPv6                 |       DST IPv6                     DST ←
         MAC        STATE      VERIFIED
--------------------------------------------------------------------------------------------------


resolved..: 0
unresolved: 0
verified..: 0

trex(service)>
```

2.15 PCAP Based Traffic Tutorials

2.15.1 PCAP Based Traffic

TRex provides a method for using pre-recorded traffic as a profile template. Typically, there are two ways to create a profile or a test based on a PCAP:

- Local PCAP push
- Server-based push

2.15.1.1 Local PCAP push

In this method, the PCAP file is loaded locally by the Python client, transformed to a list of streams, each one with a single packet carrying a payload and pointing to the next packet.

This method can provide every type of functionality that a regular list of streams might have. However, due to the overhead of processing and sending a list of streams, the file size is limited (default: 1 MB).

Pro:

- Supports most CAP file formats
- Supports Field Engine
- Provides a way of locally manipulating packets as streams

- Supports the same rate as regular streams

Con:

- Limited file size
- High configuration time due to transmitting the CAP file as streams

2.15.1.2 Server-based push

The server-based push method enables TRex to inject larger PCAP files. The mechanism is quite different from the local PCAP push method, with distinct advantages and limitations.

In this method, you provide a server with a PCAP file. The server loads the file and injects the packets, one after another. The file size is unlimted, enabling any number of packets to be injected. Setting up the server with the required configuration involves less overhead than the local PCAP push method.

Pro:

- Unlimited PCAP file size
- No overhead in sending any size of PCAP to the server

Con:

- Does not support Field Engine
- Supports only PCAP and ERF formats
- "Dual" mode is usable only with ERF format
- File path must be accessible from the server
- Rate of transmition (and IPG) is usually limited by I/O performance and buffering (HDD).

2.15.2 Tutorial: Simple PCAP file - Profile

Goal
Load a PCAP file with a **number** of packets, creating a stream with a burst value of 1 for each packet. The inter-stream gap (ISG) for each stream is equal to the inter-packet gap (IPG).

File
pcap.py

```
    def get_streams (self,
                     ipg_usec = 10.0,                           ❶
                     loop_count = 1):                           ❷

        profile = STLProfile.load_pcap(self.pcap_file,          ❸
                                       ipg_usec = ipg_usec,
                                       loop_count = loop_count)
```

❶ The inter-stream gap in microseconds.

❷ Loop count.

❸ Input PCAP file.

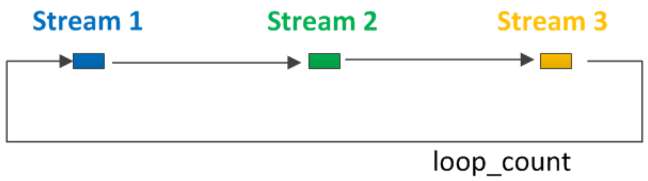


Figure 2.22: Example of multiple streams

The figure shows the streams for a PCAP file with 3 packets, with a loop configured.

- Each stream is configured to Burst mode with 1 packet.
- Each stream triggers the next stream.
- The last stream triggers the first with `action_loop=loop_count` if `loop_count` > 1.

The profile runs on one DP thread because it has a burst with 1 packet. (Cannot split in this case.)

To run this example, enter:

```
[bash]>./stl-sim -f stl/pcap.py --yaml
```

The following output appears:

```
$./stl-sim -f stl/pcap.py --yaml
- name: 1
  next: 2                              (1)
  stream:
    action_count: 0
    enabled: true
    flags: 0
    isg: 10.0
    mode:
      percentage: 100
      total_pkts: 1
      type: single_burst
    packet:
      meta: ''
    rx_stats:
      enabled: false
    self_start: true
    vm:
      instructions: []
      split_by_var: ''
- name: 2
  next: 3
  stream:
    action_count: 0
    enabled: true
    flags: 0
    isg: 10.0
    mode:
      percentage: 100
      total_pkts: 1
      type: single_burst
    packet:
      meta: ''
    rx_stats:
      enabled: false
```

```
    self_start: false
    vm:
      instructions: []
      split_by_var: ''
- name: 3
  next: 4
  stream:
    action_count: 0
    enabled: true
    flags: 0
    isg: 10.0
    mode:
      percentage: 100
      total_pkts: 1
      type: single_burst
    packet:
      meta: ''
    rx_stats:
      enabled: false
    self_start: false
    vm:
      instructions: []
      split_by_var: ''
- name: 4
  next: 5
  stream:
    action_count: 0
    enabled: true
    flags: 0
    isg: 10.0
    mode:
      percentage: 100
      total_pkts: 1
      type: single_burst
    packet:
      meta: ''
    rx_stats:
      enabled: false
    self_start: false
    vm:
      instructions: []
      split_by_var: ''
- name: 5
  next: 1                           ❷
  stream:
    action_count: 1                 ❸
    enabled: true
    flags: 0
    isg: 10.0
    mode:
      percentage: 100
      total_pkts: 1
      type: single_burst
    packet:
      meta: ''
    rx_stats:
      enabled: false
    self_start: false               ❹
    vm:
      instructions: []
      split_by_var: ''
```

1. Each stream triggers the next stream.
2. The last stream triggers the first.
3. The current loop count is given in: `action_count:1`
4. `Self_start` is enabled for the first stream, disabled for all other streams.

2.15.3 Tutorial: Simple PCAP file - API

For this case we can use the local push method:

```
c = STLClient(server = "localhost")

try:

    c.connect()
    c.reset(ports = [0])

    d = c.push_pcap(pcap_file = "my_file.pcap",            # our local PCAP file
                    ports = 0,                              # use port 0
                    ipg_usec = 100,                         # IPG
                    count = 1)                              # inject only once

    c.wait_on_traffic()


    stats = c.get_stats()
    opackets = stats[port]['opackets']
    print("{0} packets were Tx on port {1}\n".format(opackets, port))

  except STLError as e:
      print(e)
      sys.exit(1)

  finally:
      c.disconnect()
```

2.15.4 Tutorial: PCAP file iterating over dest IP

For this case we can use the local push method:

```
c = STLClient(server = "localhost")

try:

    c.connect()
    port = 0
    c.reset(ports = [port])

    vm = STLIPRange(dst = {'start': '10.0.0.1', 'end': '10.0.0.254', 'step' : 1})

    c.push_pcap(pcap_file = "my_file.pcap",            # our local PCAP file
                ports = port,                           # use 'port'
                ipg_usec = 100,                         # IPG
                count = 1,                              # inject only once
                vm = vm                                 # provide VM object
                )

    c.wait_on_traffic()
```

```
    stats = c.get_stats()
    opackets = stats[port]['opackets']
    print("{0} packets were Tx on port {1}\n".format(opackets, port))

  except STLError as e:
      print(e)
      sys.exit(1)

  finally:
      c.disconnect()
```

2.15.5 Tutorial: PCAP file with VLAN

This is an interesting case where we can provide the push API with a function hook. The hook is called for each packet that is loaded from the PCAP file.

```
# generate a packet hook function with a VLAN ID
def packet_hook_generator (vlan_id):

    # this function will be called for each packet and will expect
    # the new packet as a return value
    def packet_hook (packet):
        packet = Ether(packet)

        if vlan_id >= 0 and vlan_id <= 4096:
            packet_l3 = packet.payload
            packet = Ether() / Dot1Q(vlan = vlan_id) / packet_l3

        return str(packet)

    return packet_hook

c = STLClient(server = "localhost")

try:

    c.connect()
    port = 0
    c.reset(ports = [port])

    vm = STLIPRange(dst = {'start': '10.0.0.1', 'end': '10.0.0.254', 'step' : 1})

    d = c.push_pcap(pcap_file = "my_file.pcap",
                    ports = port,
                    ipg_usec = 100,
                    count = 1,
                    packet_hook = packet_hook_generator(vlan_id = 1)
                    )

    c.wait_on_traffic()

    stats = c.get_stats()
    opackets = stats[port]['opackets']
    print("{0} packets were Tx on port {1}\n".format(opackets, port))

  except STLError as e:
      print(e)
      sys.exit(1)
```

```
finally:
    c.disconnect()
```

2.15.6 Tutorial: PCAP file and Field Engine - Profile

The following example loads a PCAP file to many streams, and attaches the Field Engine program to each stream. For example, the Field Engine can change the `IP.src` of all the streams to a random IP address.

File
stl/pcap_with_vm.py

```
    def create_vm (self, ip_src_range, ip_dst_range):
        if not ip_src_range and not ip_dst_range:
            return None

        # until the feature of offsets will be fixed for PCAP use hard coded offsets

        vm = []

        if ip_src_range:
            vm += [STLVmFlowVar(name="src",
                                min_value = ip_src_range['start'],
                                max_value = ip_src_range['end'],
                                size = 4, op = "inc"),
                   #STLVmWrFlowVar(fv_name="src",pkt_offset= "IP.src")
                   STLVmWrFlowVar(fv_name="src",pkt_offset = 26)
                  ]

        if ip_dst_range:
            vm += [STLVmFlowVar(name="dst",
                                min_value = ip_dst_range['start'],
                                max_value = ip_dst_range['end'],
                                size = 4, op = "inc"),

                   #STLVmWrFlowVar(fv_name="dst",pkt_offset= "IP.dst")
                   STLVmWrFlowVar(fv_name="dst",pkt_offset = 30)
                   ]

        vm += [#STLVmFixIpv4(offset = "IP")
               STLVmFixIpv4(offset = 14)
               ]

        return vm


    def get_streams (self,
                     ipg_usec = 10.0,
                     loop_count = 5,
                     ip_src_range = None,
                     ip_dst_range = {'start' : '10.0.0.1',
                                     'end': '10.0.0.254'}):

        vm = self.create_vm(ip_src_range, ip_dst_range)                  ❶
        profile = STLProfile.load_pcap(self.pcap_file,
                                      ipg_usec = ipg_usec,
                                      loop_count = loop_count,
                                      vm = vm)                            ❷

        return profile.get_streams()
```

1. Creates Field Engine program.
2. Applies the Field Engine to all packets → converts to streams.

Table 2.15: Output

pkt	IPv4	flow
1	10.0.0.1	1
2	10.0.0.1	1
3	10.0.0.1	1
4	10.0.0.1	1
5	10.0.0.1	1
6	10.0.0.1	1
7	10.0.0.2	2
8	10.0.0.2	2
9	10.0.0.2	2
10	10.0.0.2	2
11	10.0.0.2	2
12	10.0.0.2	2

2.15.7 Tutorial: Server-side method with large PCAP file

The example below uses the remote push API method, providing a PCAP file to a remote server. **Note**: The file path to the PCAP file must be visible to the server.

```
c = STLClient(server = "localhost")

try:

    c.connect()
    c.reset(ports = [0])

    # use an absolute path so the server can reach this
    pcap_file = os.path.abspath(pcap_file)

    c.push_remote(pcap_file = pcap_file,
                  ports = 0,
                  ipg_usec = 100,
                  count = 1)

    c.wait_on_traffic()


    stats = c.get_stats()
    opackets = stats[port]['opackets']
    print("{0} packets were Tx on port {1}\n".format(opackets, port))

  except STLError as e:
      print(e)
      sys.exit(1)

  finally:
      c.disconnect()
```

2.15.8 Tutorial: A long list of PCAP files of varied sizes

A scenario with several PCAP files is a good candidate for the remote push API. The total overhead for sending the PCAP files will be high if there is a long list of separate PCAP files. So in this case, it is preferable to inject them with a remote API and to save the transmission of the packets.

```
c = STLClient(server = "localhost")

try:

    c.connect()
    c.reset(ports = [0])

    # iterate over the list and send each file to the server
    for pcap_file in pcap_file_list:
        pcap_file = os.path.abspath(pcap_file)

        c.push_remote(pcap_file = pcap_file,
                      ports = 0,
                      ipg_usec = 100,
                      count = 1)

        c.wait_on_traffic()


        stats = c.get_stats()
        opackets = stats[port]['opackets']
        print("{0} packets were Tx on port {1}\n".format(opackets, port))

  except STLError as e:
      print(e)
      sys.exit(1)

  finally:
      c.disconnect()
```

2.16 Performance Tweaking

This section describes some advanced features that can help to optimize TRex performance. These features are not active "out of the box" because they might have some impact on other functionality, and in general, might sacrafice one or more properties. Users can decide on any trade-offs individually before employing these optimizations.

2.16.1 Caching MBUFs

see here [trex_cache_mbuf]

2.16.2 Core masking per interface

By default, TRex will regard any TX command with a **greedy approach**: All DP cores associated with this port will be assigned in order to produce the maximum throughput.

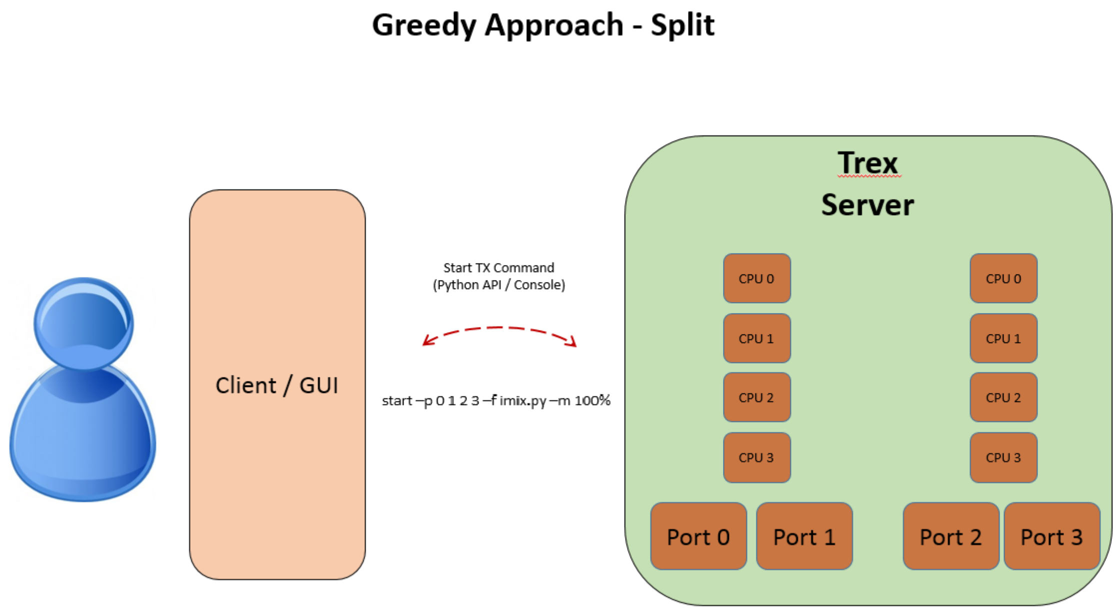


Figure 2.23: Greedy Approach - Splitting

However, in some cases it might be beneficial to provide a port with a subset of the cores to use, such as when injecting traffic on two ports, when the following conditions are met:

- The two ports are adjacent.
- The profile is symmetric.

Due to TRex architecture, adjacent ports (example: port 0 and port 1) share the same cores, and using the greedy approach will cause all the cores to transmit on both port 0 and port 1.

When the profile is **symmetric**, performance can be improved by pinning half of the cores to port 0, and half of the cores to port 1, thus avoiding cache trashing and bouncing. If the profile is **not symmetric**, the static pinning may deny CPU cycles from the more congested port.

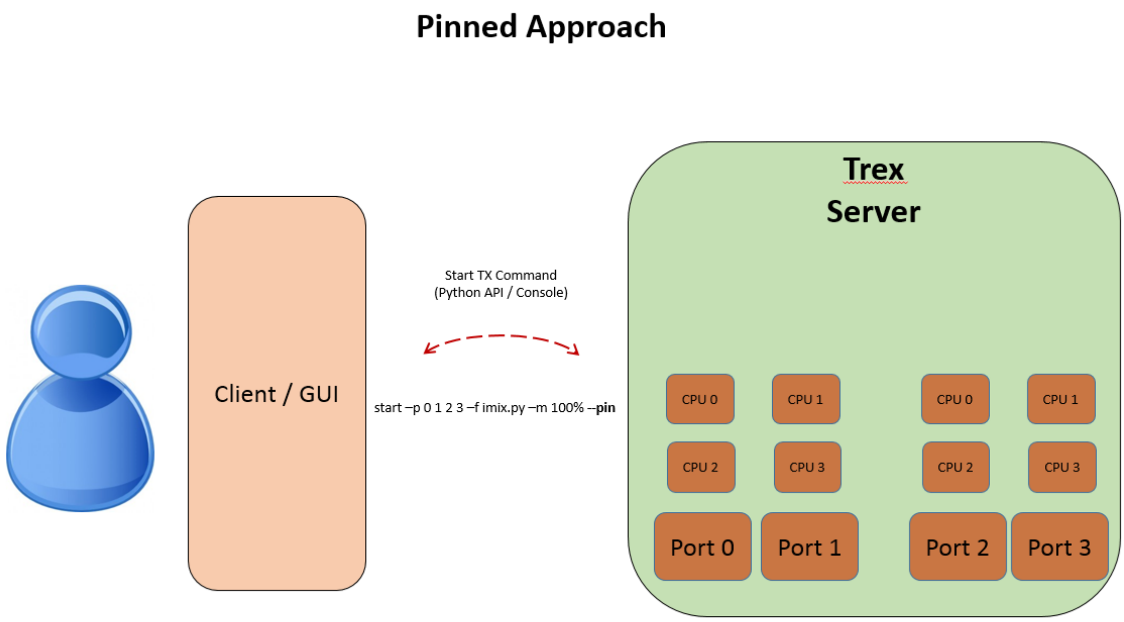


Figure 2.24: Pinning Cores To Ports

TRex provides this in two ways, described below.

2.16.3 Predefind modes

As described above, the default mode is *split* mode, but you can configure a predefined mode called *pin*. This can be done by API or from the console.

```
trex>start -f stl/syn_attack.py -m 40mpps --total -p 0 1 --pin        <-- provide '--pin' ↩
    to the command

Removing all streams from port(s) [0, 1]:                        [SUCCESS]


Attaching 1 streams to port(s) [0]:                              [SUCCESS]


Attaching 1 streams to port(s) [1]:                              [SUCCESS]


Starting traffic on port(s) [0, 1]:                              [SUCCESS]

60.20 [ms]

trex>
```

API example to PIN cores

```
c.start(ports = [port_a, port_b], mult = rate,core_mask=STLClient.CORE_MASK_PIN) ❶
```

❶ core_mask = STLClient.CORE_MASK_PIN

API example to MASK cores

```
c.start(ports = [port_a, port_b], mult = rate, core_mask=[0x1,0x2]) (1)
```

(1) DP Core 0 (mask==1) is assigned to port 1, and DP core 1 (mask==2) assigned to port 2.

The CPU Util table, available in the TUI window, shows that each core was reserved for an interface:

```
Global Stats:

Total Tx L2  : 20.49 Gb/sec
Total Tx L1  : 26.89 Gb/sec
Total Rx     : 20.49 Gb/sec
Total Pps    : 40.01 Mpkt/sec       <-- Performance meets the requested rate
Drop Rate    : 0.00 b/sec
Queue Full   : 0 pkts


Cpu Util(%)

  Thread   | Avg | Latest | -1  | -2  | -3  | -4  | -5  | -6  | -7  | -8

 0   (0)   |  92 |     92 |  92 |  91 |  91 |  92 |  91 |  92 |  93 |  94
 1 (IDLE)  |   0 |      0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0
 2   (1)   |  96 |     95 |  95 |  96 |  96 |  96 |  96 |  95 |  94 |  95
 3 (IDLE)  |   0 |      0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0
 4   (0)   |  92 |     93 |  93 |  91 |  91 |  93 |  93 |  93 |  93 |  93
 5 (IDLE)  |   0 |      0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0
 6   (1)   |  88 |     88 |  88 |  88 |  88 |  88 |  88 |  88 |  87 |  87
 7 (IDLE)  |   0 |      0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0
```

If we had used the **default mode**, the table would have looked like the following, with significantly worse performance:

```
Global Stats:

Total Tx L2  : 12.34 Gb/sec
Total Tx L1  : 16.19 Gb/sec
Total Rx     : 12.34 Gb/sec
Total Pps    : 24.09 Mpkt/sec       <-- Performance is much lower than requested
Drop Rate    : 0.00 b/sec
Queue Full   : 0 pkts

Cpu Util(%)

  Thread   | Avg | Latest | -1  | -2  | -3  | -4  | -5  | -6  | -7  | -8

 0  (0,1)  | 100 |    100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100
 1 (IDLE)  |   0 |      0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0
 2  (0,1)  | 100 |    100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100
 3 (IDLE)  |   0 |      0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0
 4  (0,1)  | 100 |    100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100
 5 (IDLE)  |   0 |      0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0
 6  (0,1)  | 100 |    100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100
 7 (IDLE)  |   0 |      0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0
```

This feature is also available from the Python API by providing: **CORE_MASK_SPLIT** or **CORE_MASK_PIN** to the start API.

2.16.4 Manual mask

For debugging or advanced core scheduling you might choose to provide a manual masking to specify to the server which cores to use.

Example:

- 2 interfaces: interface 0 and interface 1
- A profile that utilizes 95% of the traffic on one side, and provides 5% of the traffic in the other direction.
- 8 cores assigned to the two interfaces.

To assign 3 cores to interface 0 and 1 core to interface 1, execute the following in the console (or if using the API, provide a list of masks to the start command):

```
trex>start -f stl/syn_attack.py -m 10mpps --total -p 0 1 --core_mask 0xE 0x1

Removing all streams from port(s) [0, 1]:                   [SUCCESS]


Attaching 1 streams to port(s) [0]:                         [SUCCESS]


Attaching 1 streams to port(s) [1]:                         [SUCCESS]


Starting traffic on port(s) [0, 1]:                         [SUCCESS]

37.19 [ms]

trex>
```

```
c.start(ports = [port_a, port_b], mult = rate,core_mask=[0x0xe,0x1]) (1)
```

(1) Mask of cores per port.

The following output is appears in the TUI CPU Util window:

```
Total Tx L2  : 5.12 Gb/sec
Total Tx L1  : 6.72 Gb/sec
Total Rx     : 5.12 Gb/sec
Total Pps    : 10.00 Mpkt/sec
Drop Rate    : 0.00 b/sec
Queue Full   : 0 pkts

Cpu Util(%)

  Thread   | Avg | Latest | -1  | -2  | -3  | -4  | -5  | -6  | -7  | -8

 0   (1)   |  45 |     45 |  45 |  45 |  45 |  45 |  46 |  45 |  46 |  45
 1 (IDLE)  |   0 |      0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0
 2   (0)   |  15 |     15 |  14 |  15 |  15 |  14 |  14 |  14 |  14 |  14
 3 (IDLE)  |   0 |      0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0
 4   (0)   |  14 |     14 |  14 |  14 |  14 |  14 |  14 |  14 |  15 |  14
 5 (IDLE)  |   0 |      0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0
 6   (0)   |  15 |     15 |  15 |  15 |  15 |  15 |  15 |  15 |  15 |  15
 7 (IDLE)  |   0 |      0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0
```

2.17 Reference

Additional profiles and examples are available in the `stl/hlt` folder.

For information about the Python client API, see the Python Client API documentation.

2.18 Console commands

2.18.1 Overview

The console uses the TRex client API to control TRex.

Important information about console usage:

- The console does not save its own state. It caches the server state. It is assumed that there is only one console with R/W permission at any given time, so once connected as R/W console (per user/interface), it can read the server state and then cache all operations.
- Many read-only clients can exist for the same user interface.
- The console syncs with the server to get the state during connection stage, and caches the server information locally.
- In case of crash or exit of the console, it will sync again at startup.
- The order of command line parameters is not important.
- The console can display TRex stats in real time. You can open two consoles simultaneously - one for commands (R/W) and one for displaying statistics (read only).

2.18.2 Ports State

State	Meaning
IDLE	`No streams`
STREAMS	`Has streams. Not transmitting (did not start transmission, or transmission was stopped).`
WORK	`Has streams. Transmitting.`
PAUSE	`Has streams. Transmission paused.`

```
IDLE -> (add streams) -> STREAMS (start) -> WORK (stop) ->  STREAMS (start)
                                           |  WORK (pause) -> PAUSE (resume)----
                                           |                                    |
                                           |                                    |
                                           -------------------------------------
```

2.18.3 Common Arguments

The following command line arguments are common to many commands.

2.18.3.1 Help

You can specify -h or --help after each command to display full description of its purpose and arguments.

Example

```
trex>streams -h
```

2.18.3.2 Port mask

Port mask enables selecting range, or set of ports.

Example

```
trex><command>   [-a] [--port 1 2 3]  [--port 0xff]  [--port clients/servers]

  port mask :
    [-a]           : all ports
    [--port 1 2 3]  : port 1,2 3
    [--port 0xff]   : port by mask 0x1 for port 0 0x3 for port 0 and 1
```

2.18.3.3 Duration

Duration is expressed in seconds, minutes, or hours.

Example

```
trex><command> [-d 100] [-d 10m] [-d 1h]

  duration:
   -d 100 : Seconds
   -d 10m : Minutes
   -d 1h  : Hours
```

2.18.3.4 Multiplier

The traffic profile defines the default bandwidth for each stream. Using the multiplier command line argument, you can set a different bandwidth. You can specify either packets or bytes per second, percentage of total port rate, or simply a factor by which to multiply the original rate.

Example

```
trex><command> [-m 100] [-m 10gb] [-m 10kpps] [-m 40%]

  multiplier :

  -m 100    : Multiply original rate by given factor.
  -m 10gbps : From graph calculate the maximum rate as this bandwidth for all streams (for ←
      each port)
  -m 10kpps : From graph calculate the maximum rate as this pps for all streams (for each ←
      port)
  -m 40%    : From graph calculate the maximum rate as this precent from total port rate ( ←
      for each port)
```

2.18.4 Commands

2.18.4.1 connect

Attempts to connect to the server you were connected to. Can be used after a server restart. Cannot be used to connect to a different server.

Also:

- Syncs the port info and stream info state.
- Reads all counter statistics for reference.

Example

```
$connect
```

2.18.4.2 reset

Resets the server and client to a known state. Not used in normal scenarios.

- Forces acquire on all ports.
- Stops all traffic on all ports.
- Removes all streams from all ports.

  Example

```
trex>reset
```

2.18.4.3 portattr

Configures port attributes.

Example

```
trex>portattr --help
usage: port_attr [-h] [--port PORTS [PORTS ...] | -a] [--prom {on,off}]
                 [--link {up,down}] [--led {on,off}] [--fc {none,tx,rx,full}]
                 [--supp]

Sets port attributes

optional arguments:
  -h, --help            show this help message and exit
  --port PORTS [PORTS ...], -p PORTS [PORTS ...]
                        A list of ports on which to apply the command
  -a                    Set this flag to apply the command on all available
                        ports
  --prom {on,off}       Set port promiscuous on/off
  --link {up,down}      Set link status up/down
  --led {on,off}        Set LED status on/off
  --fc {none,tx,rx,full}
                        Set Flow Control type
  --supp                Show which attributes are supported by current NICs
```

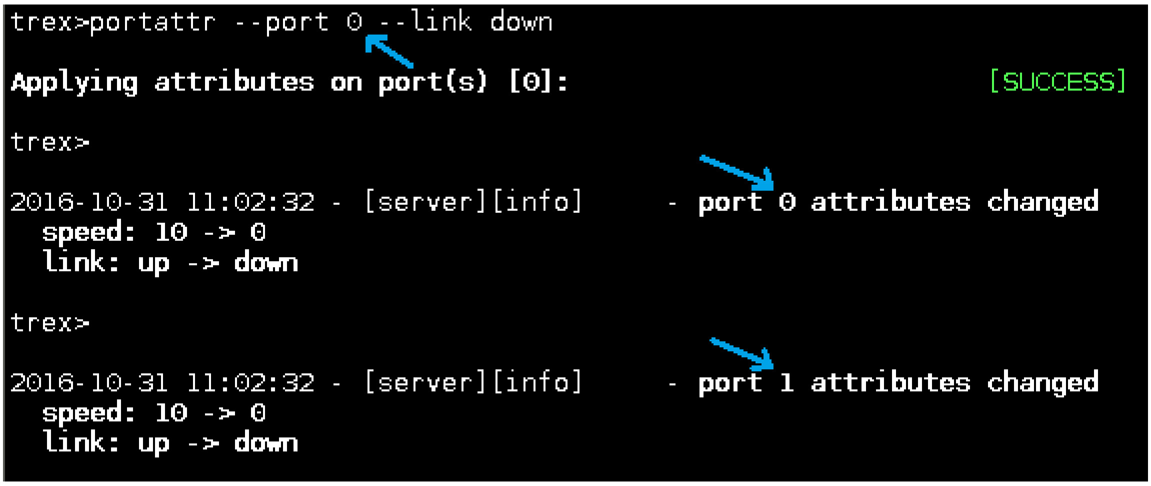


Figure 2.25: Setting link down on port 0 affects port 1 at loopback

2.18.4.4 clear

Clears all port stats counters.

Example

```
trex>clear -a
```

2.18.4.5 stats

Can be used to show global/port/stream statistics. Can also be used to retrieve extended stats from port (xstats).

Example

```
trex>stats --port 0 -p
trex>stats -s
```

Xstats error example

```
trex>stats -x --port 0 2
Xstats:

            Name:                 |      Port 0:      |      Port 2:
\-----------------------------------------------------------------
rx_good_packets                   |        154612905 |        153744994
tx_good_packets                   |        154612819 |        153745136
rx_good_bytes                     |       9895225920 |       9839679168
tx_good_bytes                     |       9276768500 |       9224707392
rx_unicast_packets                |        154611873 |        153743952
rx_unknown_protocol_packets       |        154611896 |        153743991
tx_unicast_packets                |        154612229 |        153744562
mac_remote_errors                 |                1 |                0 #1
rx_size_64_packets                |        154612170 |        153744295
tx_size_64_packets                |        154612595 |        153744902
```

1. Error that can be seen only with this command.

2.18.4.6 streams

Shows info about loaded streams on TRex port(s). By default, streams are displayed in a brief summary table:

```
trex>streams

Port 0:

    ID      |      packet type      |  length  |        mode        |        rate        | next ↩
       stream

    1       | Ethernet:IP:UDP:Raw |       64 |    continuous      |              1 pps |       -1
    2       | Ethernet:IP:UDP:Raw |       64 |    continuous      |         1.00 Kpps |       -1

Port 1:

    ID      |      packet type      |  length  |        mode        |        rate        | next ↩
       stream

    1       | Ethernet:IP:UDP:Raw |       64 |    continuous      |              1 pps |       -1
    2       | Ethernet:IP:UDP:Raw |       64 |    continuous      |         1.00 Kpps |       -1
```

Streams can be filtered by port ID(s) and/or stream ID(s).

```
trex>streams --port 0 1 --id 0

 ..
 ..
```

The `--code` argument is used as follows:

- `--code` can be used without argument to display generated Python code that produces the streams.

```
trex>streams -i 2 --code
Port: 0

Stream ID: 2
    packet = (Ether(type=2048) /
              IP(proto=17, chksum=14507, len=576, ihl=5) /
              UDP(len=556, chksum=51769) /
              Raw(load='x' * 548))
    vm = STLVM()
    vm.var(name='src', size=4, op='inc', init_value='16.0.0.1', min_value='16.0.0.1',  ↩
        max_value='16.0.0.254', step=1)
    vm.var(name='dst', size=4, op='inc', init_value='48.0.0.1', min_value='48.0.0.1',  ↩
        max_value='48.0.0.254', step=1)
    vm.write(fv_name='src', pkt_offset='IP.src', add_val=0, byte_order='big')
    vm.write(fv_name='dst', pkt_offset='IP.dst', add_val=0, byte_order='big')
    vm.fix_chksum(offset='IP')
    stream = STLStream(packet = STLPktBuilder(pkt = packet, vm = vm),
                       isg = 0.1,
                       mac_src_override_by_pkt = False,
                       mac_dst_override_mode = 0,
                       mode = STLTXCont(pps = 20))
```

- `--code` can be used with a filename argument to save the streams as a Python profile. In the following example, the specified filename is: **/tmp/my_imix.py**

```
trex>streams -p 0 --code /tmp/my_imix.py

Saving file as: /tmp/my_imix.py                              [SUCCESS]

...

bash> cat /tmp/my_imix.py

# !!! Auto-generated code !!!
from trex_stl_lib.api import *

class STLS1(object):
    def get_streams(self, direction = 0, **kwargs):
        streams = []
        packet = (Ether(type=2048) /
                  IP(proto=17, chksum=15037, len=46, ihl=5) /
                  UDP(len=26, chksum=33554) /
                  Raw(load='x' * 18))
        vm = STLVM()
        vm.var(name='src', size=4, op='inc', init_value='16.0.0.1', min_value='16.0.0.1',  ↩
            max_value='16.0.0.254', step=1)
        vm.var(name='dst', size=4, op='inc', init_value='48.0.0.1', min_value='48.0.0.1',  ↩
            max_value='48.0.0.254', step=1)
        vm.write(fv_name='src', pkt_offset='IP.src', add_val=0, byte_order='big')
        vm.write(fv_name='dst', pkt_offset='IP.dst', add_val=0, byte_order='big')
        vm.fix_chksum(offset='IP')
        stream = STLStream(packet = STLPktBuilder(pkt = packet, vm = vm),
                           mac_src_override_by_pkt = False,
                           mac_dst_override_mode = 0,
                           mode = STLTXCont(pps = 28))
        streams.append(stream)
        packet = (Ether(type=2048) /
                  IP(proto=17, chksum=14507, len=576, ihl=5) /
                  UDP(len=556, chksum=51769) /
                  Raw(load='x' * 548))
        vm = STLVM()
```

```
        vm.var(name='src', size=4, op='inc', init_value='16.0.0.1', min_value='16.0.0.1', ↩
            max_value='16.0.0.254', step=1)
        vm.var(name='dst', size=4, op='inc', init_value='48.0.0.1', min_value='48.0.0.1', ↩
            max_value='48.0.0.254', step=1)
        vm.write(fv_name='src', pkt_offset='IP.src', add_val=0, byte_order='big')
        vm.write(fv_name='dst', pkt_offset='IP.dst', add_val=0, byte_order='big')
        vm.fix_chksum(offset='IP')
        stream = STLStream(packet = STLPktBuilder(pkt = packet, vm = vm),
                           isg = 0.1,
                           mac_src_override_by_pkt = False,
                           mac_dst_override_mode = 0,
                           mode = STLTXCont(pps = 20))
        streams.append(stream)
        packet = (Ether(type=2048) /
                  IP(proto=17, chksum=13583, len=1500, ihl=5) /
                  UDP(len=1480, chksum=22936) /
                  Raw(load='x' * 1472))
        vm = STLVM()
        vm.var(name='src', size=4, op='inc', init_value='16.0.0.1', min_value='16.0.0.1', ↩
            max_value='16.0.0.254', step=1)
        vm.var(name='dst', size=4, op='inc', init_value='48.0.0.1', min_value='48.0.0.1', ↩
            max_value='48.0.0.254', step=1)
        vm.write(fv_name='src', pkt_offset='IP.src', add_val=0, byte_order='big')
        vm.write(fv_name='dst', pkt_offset='IP.dst', add_val=0, byte_order='big')
        vm.fix_chksum(offset='IP')
        stream = STLStream(packet = STLPktBuilder(pkt = packet, vm = vm),
                           isg = 0.2,
                           mac_src_override_by_pkt = False,
                           mac_dst_override_mode = 0,
                           mode = STLTXCont(pps = 4))
        streams.append(stream)

        return streams

def register():
    return STLS1()
```

2.18.4.7 start

Start transmitting traffic on set of ports.

- Removes all streams.
- Loads new streams.
- Starts traffic (can set multiplier, duration, and other parameters).
- Acts only on ports in "stopped: mode. If `--force` is specified, port(s) are first stopped.
- Note: If any ports are not in "stopped" mode, and `--force` is not used, the command fails.

Example
Start a profile on all ports, with a maximum bandwidth of 10 GB.

```
trex>start -a -f stl/imix.py  -m 10gb
```

Example
Start a profile on ports 1 and 2, and multiply the bandwidth specified in the traffic profile by 100.

```
trex>start -port 1 2 -f stl/imix.py  -m 100
```

2.18.4.8 stop

- Operates on a set of ports.
- Changes the mode of the port(s) to "stopped".
- Does not remove streams.

Example
Use this command to stop the specified ports.

```
trex>stop --port 0
```

2.18.4.9 pause

- Operates on a set of ports.
- Changes a working set of ports to "pause" (no traffic transmission) state.

Example

```
trex>pause --port 0
```

2.18.4.10 resume

- Operates on a set of ports.
- Changes a working set of port(s) to "resume" state (transmitting traffic again).
- All ports should be in "paused" status. If any of the ports is not paused, the command fails.

Example

```
trex>resume --port 0
```

2.18.4.11 update

Update the bandwidth multiplier for a set of ports.

- All ports must be in "work" state. If any ports are not in "work" state, the command fails.

Example
Multiply traffic on all ports by a factor of 5.

```
trex>update -a -m 5
```

2.18.4.12 per stream operations

In version v2.34 was added option to pause/resume/update specific stream(s).
In future this functionality might be altered/removed.
Internally uses API commands:

- pause_streams
- resume_streams
- update_streams

In the console, stream ids can be specified via --id argument.
(Stream IDs are shown in output of "streams" command)
For example, one could use following:

> start -f stl/imix.py -p 0
> pause -p 0 --id 1 2
> resume -p 0 --id 1
> update -p 0 --id 1 -m 1kpps

> **Warning**
>
> - Pausing/resuming specific stream(s) does **not** change state of port.
> - Changing rates of specific streams causes out of sync between CP and DP regarding streams rate.
>   Prior to updating rate of whole port, need to revert changes made to rates of specific streams.

2.18.4.13 TUI

The textual user interface (TUI) displays constantly updated TRex statistics in a text window.

Example

```
trex>tui
```

Enters a Stats mode and displays three types of TRex statistics:

- Global/port stats/version/connected, etc.
- Per port
- Per port stream

Keyboard commands in the TUI window:

Command	Description
q	Quit the TUI window (return to console)
c	Clear all counters
d, s, l	Change display between dashboard (d), streams (s) and l (latency) info

2.19 Benchmarks of 40G NICs

TRex stateless benchmarks

2.20 Bird integration

2.20.1 Overview

Bird Internet Routing Daemon is a project aimed to develop a fully functional linux dynamic IP routing daemon. It was integrated into TRex to run alongside in order to exploit it's features together with Python automation API.

2.20.2 FAQ

see: Bird's FAQ

2.20.3 High level Features

- BIRD 2.0 Protocols support Bird
  - Run on top of IPv4 and IPv6 (using kernel veth)
  - BGP (eBGP/iBGP), RPKI (RFC 6480)/RFC 6483 records type are pv4,ipv6,vpn4,vpn6,multicast,flow4,flow6
    * RFC 4271 - Border Gateway Protocol 4 (BGP)
    * RFC 1997 - BGP Communities Attribute
    * RFC 2385 - Protection of BGP Sessions via TCP MD5 Signature
    * RFC 2545 - Use of BGP Multiprotocol Extensions for IPv6
    * RFC 2918 - Route Refresh Capability
    * RFC 3107 - Carrying Label Information in BGP
    * RFC 4360 - BGP Extended Communities Attribute
    * RFC 4364 - BGP/MPLS IPv4 Virtual Private Networks
    * RFC 4456 - BGP Route Reflection
    * RFC 4486 - Subcodes for BGP Cease Notification Message
    * RFC 4659 - BGP/MPLS IPv6 Virtual Private Networks
    * RFC 4724 - Graceful Restart Mechanism for BGP
    * RFC 4760 - Multiprotocol extensions for BGP
    * RFC 4798 - Connecting IPv6 Islands over IPv4 MPLS
    * RFC 5065 - AS confederations for BGP
    * RFC 5082 - Generalized TTL Security Mechanism
    * RFC 5492 - Capabilities Advertisement with BGP
    * RFC 5549 - Advertising IPv4 NLRI with an IPv6 Next Hop
    * RFC 5575 - Dissemination of Flow Specification Rules
    * RFC 5668 - 4-Octet AS Specific BGP Extended Community
    * RFC 6286 - AS-Wide Unique BGP Identifier
    * RFC 6608 - Subcodes for BGP Finite State Machine Error
    * RFC 6793 - BGP Support for 4-Octet AS Numbers
    * RFC 7311 - Accumulated IGP Metric Attribute for BGP
    * RFC 7313 - Enhanced Route Refresh Capability for BGP
    * RFC 7606 - Revised Error Handling for BGP UPDATE Messages
    * RFC 7911 - Advertisement of Multiple Paths in BGP

    * RFC 7947 - Internet Exchange BGP Route Server
    * RFC 8092 - BGP Large Communities Attribute
    * RFC 8203 - BGP Administrative Shutdown Communication
    * RFC 8212 - Default EBGP Route Propagation Behavior without Policies
  - OSPF (v2/v3) RFC 2328/ RFC 5340
  - RIP - RIPv1 (RFC 1058),RIPv2 (RFC 2453), RIPng (RFC 2080), and RIP cryptographic authentication (RFC 4822).

- Scale of Millions of routes (depends on the protocol scale e.g. BGP) in a few seconds
- Integration with Multi-RX software model (`-software` and `-c` higher than 1) to supprt dynamic filters for BIRD protocols while keeping high rates of traffic
- Can support up to 10K veth (virtual interfaces) each with different QinQ/VLAN configuration
- Simple automation Python API for pushing configuration and read statistics

2.20.4 Integration Topology

Bird process is running in it's own namespace on trex machine. Using veth's to communicate the default namespace and sending data to DUT. Client will use "PyBird Client" in order to control the BIRD throw ZMQ JSON-RPC2 API (each component described below). BIRD is using the TRex linux namespace feature

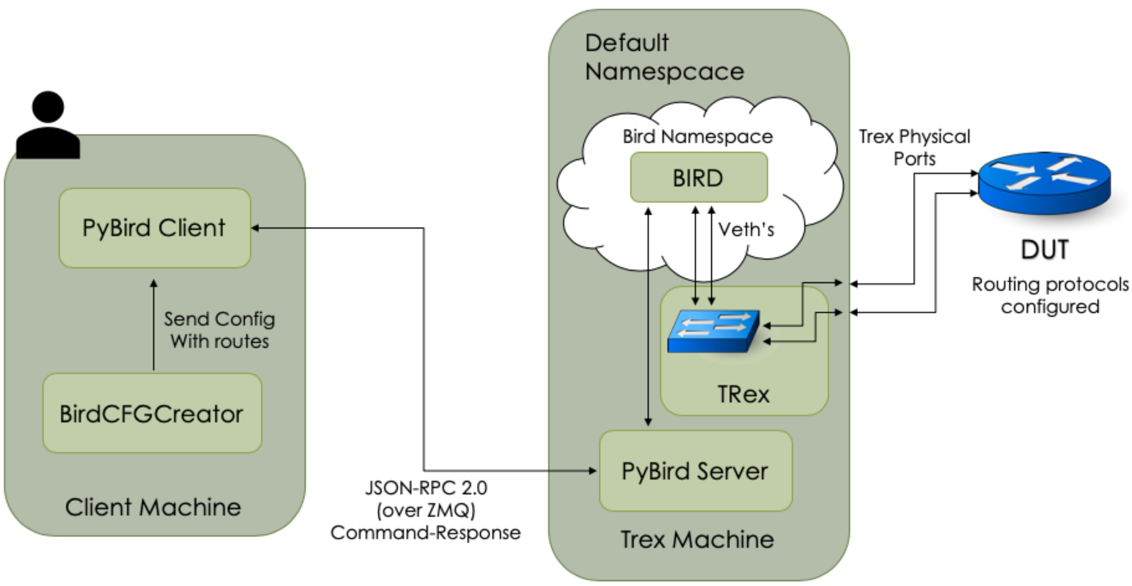


Figure 2.26: TRex-Bird General Scheme

This image shows 2 veth (one per physical interface) that switched throw TRex

2.20.5 First Time Running

Le't start with a simple example, pushing a few BGP routes to ASR1K

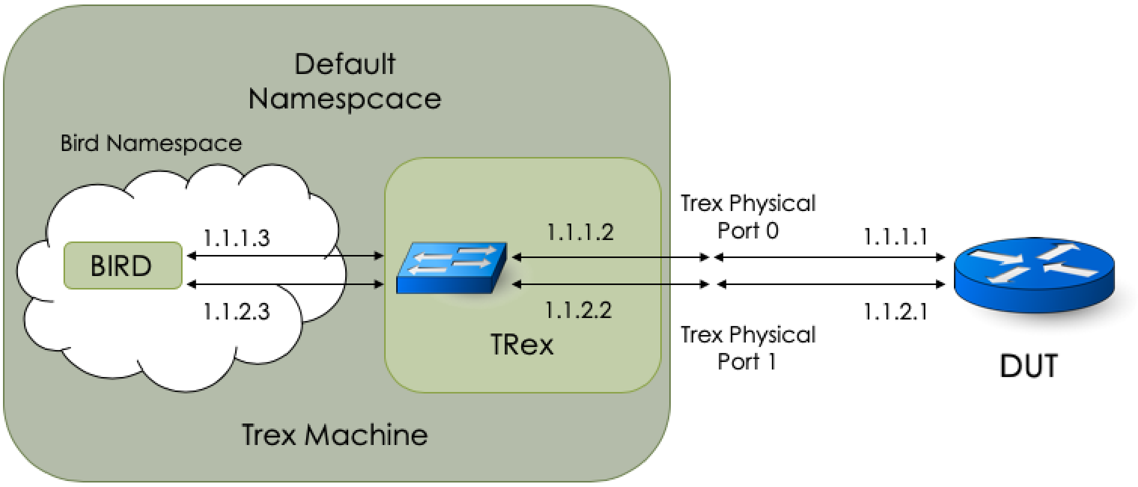


Figure 2.27: TRex Bird General Scheme With ips

- Configure trex yaml file with "stack: linux_based" (see Linux Stack):

In this configuration the BIRD veth are on the same subnet of TRex ports. There is a need to change the port to promiscuous mode so Physical ports will get the veth packets and forward them to the BIRD daemon The BIRD pps is rather low to achive many routes in a short time

linux_based stack configuration

```
  - version: 2
    interfaces: ['82:00.0', '82:00.1']
    stack: linux_based
...
```

- Configure DUT routing protocols:

ASR1001-01 configuration to enable BGP

```
asr1001-01#configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
asr1001-01(config)#router bgp 65000
asr1001-01(config-router)#neighbor 1.1.1.3 remote-as 65000
asr1001-01(config-router)#neighbor 1.1.2.3 remote-as 65000
asr1001-01(config-router)#end
```

- Run TRex in "ONE_QUEUE \ MULTI_QUEUE" mode and set bird flag on :

```
[bash]>sudo ./t-rex-64 -i --bird-server --software -c 1

OR

[bash]>sudo ./t-rex-64 -i --bird-server --software -c 2
```

The `--bird-server` will load the bird daemon and the ZMQ server

- In another shell run python script located at: `scripts/automation/trex_control_plane/interactive/trex/examples/stl/add_bird_node.py` (see Simple bird node adding)

2.20.6 Tutorial: Bird node adding using TRex console

All the toturial based on the topology here [Bird_topo]

Goal
Demonstrate the simple way of adding a bird node.

1) Open TRex console:

```
[bash]> ./trex-console
```

2) Enable service mode for wanted ports (-a for all) also promiscuous and multicast:

```
trex>service -a
trex(service)>portattr -a --prom on --mult on
```

3) Load bird plugin:

```
trex(service)>plugins load bird
```

4) Add bird node using wanted params (need at least port and mac):

```
trex(service)>plugins bird add_node -p 0 --mac 00:00:00:00:00:06 --ipv4 1.1.1.3 --ipv4- ↩
    subnet 24

trex(service)>plugins bird show_nodes -p 0

Bird Nodes Information

    Node MAC       |    ipv4 address     |     ipv4 subnet     |    ipv6 enabled     |    ipv6  ↩
        address     |    ipv6 subnet     |        vlans        |        tpids
------------------+-------------------+-------------------+-------------------+-------------------+--

00:00:00:00:00:06 |      1.1.1.3      |        24         |       False       |         -  ↩
          |         -          |         -          |         -
```

5) Let's take a look at a simple bgp bird profile:

Bgp.conf

```
> more bird/cfg/bgp.conf
router id 100.100.100.100;

protocol device {
    scan time 1;
}

protocol bgp my_bgp1 {
    local 1.1.1.3 as 65000;  # put your local ip and as number here
    neighbor 1.1.1.1 as 65000;  # put your dut ip and as number here
    ipv4 {
        import all;
        export all;
    };
```

```
}

# using a second interface
protocol bgp my_bgp2 {
    local 1.1.2.3 as 65000;  # same for the second interface
    neighbor 1.1.2.1 as 65000;
    ipv4 {
        import all;
        export all;
    };
}
```

Ipv4_routes

```
> more bird/cfg/ipv4_routes.conf

protocol static {
    ipv4 {
        import all;
        export all;
    };

    route 42.42.42.0/32 via 1.1.1.3;
    route 42.42.42.1/32 via 1.1.1.3;
    route 42.42.42.2/32 via 1.1.1.3;
    route 42.42.42.3/32 via 1.1.1.3;
    route 42.42.42.4/32 via 1.1.1.3;
    route 42.42.42.5/32 via 1.1.1.3;
    route 42.42.42.6/32 via 1.1.1.3;
    route 42.42.42.7/32 via 1.1.1.3;
    route 42.42.42.8/32 via 1.1.1.3;
    route 42.42.42.9/32 via 1.1.1.3;
    route 42.42.42.10/32 via 1.1.1.3;
}
```

6) Set your wanted bird config file and your routing file (examples located at "bird/cfg/*.conf"):

```
trex(service)>plugins bird set_config -f bird/cfg/bgp.conf -r bird/cfg/ipv4_routes.conf

Bird configuration result: Configured successfully
```

You can also generate your own routes using three flags (all required):

```
trex(service)>plugins bird set_config -f bird/cfg/bgp.conf --first-ip 42.42.42.0 --total- ↩
    routes 100 --next-hop 1.1.1.3

Bird configuration result: Configured successfully
```

That will add 100 routes with next hop of 1.1.1.3 (if the next hop isn't familiar the router probably won't accept it) starting from 42.42.42.0 like: 42.42.42.0 via 1.1.1.3, 42.42.42.1 via 1.1.1.3, … , 42.42.42.100 via 1.1.1.3

7) Show the bird config:

```
trex(service)>plugins bird show_config
router id 100.100.100.100;

protocol device {
    scan time 1;
}

protocol bgp my_bgp1 {
    local 1.1.1.3 as 65000;  # put your local ip and as number here
```

```
    neighbor 1.1.1.1 as 65000;  # put your dut ip and as number here
    ipv4 {
        import all;
        export all;
    };
}

# using a second interface
protocol bgp my_bgp2 {
    local 1.1.2.3 as 65000;  # same for the second interface
    neighbor 1.1.2.1 as 65000;
    ipv4 {
        import all;
        export all;
    };
}

protocol static {
    ipv4 {
        import all;
        export all;
    };

    route 42.42.42.0/32 via 1.1.1.3;
    route 42.42.42.1/32 via 1.1.1.3;
    route 42.42.42.2/32 via 1.1.1.3;
    route 42.42.42.3/32 via 1.1.1.3;
    route 42.42.42.4/32 via 1.1.1.3;
    route 42.42.42.5/32 via 1.1.1.3;
    route 42.42.42.6/32 via 1.1.1.3;
    route 42.42.42.7/32 via 1.1.1.3;
    route 42.42.42.8/32 via 1.1.1.3;
    route 42.42.42.9/32 via 1.1.1.3;
    route 42.42.42.10/32 via 1.1.1.3;
}
```

8) Show the protocol table as it in bird:

```
trex(service)>plugins bird show_protocols
Name       Proto      Table      State  Since         Info
device1    Device     ---        up     15:09:24.435
my_bgp1    BGP        ---        start  15:09:24.435  Connect       Socket: Network is ↩
    unreachable
my_bgp2    BGP        ---        start  15:09:24.435  Active        Socket: Connection ↩
    closed
static1    Static     master4    up     15:24:44.992
```

9) (At the router: ASR1001) Show the recived routes:

```
asr1001-01#show ip route
Codes: L - local, C - connected, S - static, R - RIP, M - mobile, B - BGP
       D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
       N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
       E1 - OSPF external type 1, E2 - OSPF external type 2
       i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS level-2
       ia - IS-IS inter area, * - candidate default, U - per-user static route
       o - ODR, P - periodic downloaded static route, H - NHRP, l - LISP
       a - application route
       + - replicated route, % - next hop override

Gateway of last resort is not set
```

```
      1.0.0.0/8 is variably subnetted, 4 subnets, 2 masks
C        1.1.1.0/24 is directly connected, TenGigabitEthernet0/0/0
L        1.1.1.1/32 is directly connected, TenGigabitEthernet0/0/0
C        1.1.2.0/24 is directly connected, TenGigabitEthernet0/0/1
L        1.1.2.1/32 is directly connected, TenGigabitEthernet0/0/1
      42.0.0.0/32 is subnetted, 11 subnets
B        42.42.42.0 [200/0] via 1.1.1.3, 00:00:50
B        42.42.42.1 [200/0] via 1.1.1.3, 00:00:50
B        42.42.42.2 [200/0] via 1.1.1.3, 00:00:50
B        42.42.42.3 [200/0] via 1.1.1.3, 00:00:50
B        42.42.42.4 [200/0] via 1.1.1.3, 00:00:50
B        42.42.42.5 [200/0] via 1.1.1.3, 00:00:50
B        42.42.42.6 [200/0] via 1.1.1.3, 00:00:50
B        42.42.42.7 [200/0] via 1.1.1.3, 00:00:50
B        42.42.42.8 [200/0] via 1.1.1.3, 00:00:50
B        42.42.42.9 [200/0] via 1.1.1.3, 00:00:50
B        42.42.42.10 [200/0] via 1.1.1.3, 00:00:50
```

10) Moving to "Service filtered mode" and sending traffic:

Example

```
trex(service)>service --bgp --no-tcp-udp
```

In this mode you can generate tcp/udp traffic while bgp service is on.

11) Removing all nodes:

Example

```
trex(service)>ns remove-all
```

12) Adding bird node with ipv6: .Example

```
trex(service)>plugins bird add_node -p 0 --mac 00:00:00:01:00:07 --ipv6-enable --ipv6 2001: ↩
    db8:0:2222::3 --ipv6-subnet 64
```

13) Setting ipv6 bird config:

```
plugins bird set_config -f bird/cfg/bgp_ipv6.conf
```

See the ipv6 bgp file:

```
> cat scripts/bird/cfg/bgp_ipv6.conf
router id 100.100.100.100;

protocol device {
    scan time 1;
}

protocol bgp my_bgp {
    local 2001:db8:0:2222::3 as 65000;  # put your local ip and as number here
    neighbor 2001:db8:0:2222::1 as 65000;  # put your dut ip and as number here
    ipv6 {
        import all;
        export all;
    };
}
```

Example

```
c = STLClient(verbose_level = 'debug')
my_ports = [0, 1]
c.connect()
c.acquire(ports = my_ports)
c.set_service_mode(ports = my_ports, enabled = True)                    ❶

bird_cfg = BirdCFGCreator()                                              ❷
bgp_data1 = """
    local 1.1.1.3 as 65000;
    neighbor 1.1.1.1 as 65000;
    ipv4 {
            import all;
            export all;
    };
"""
bgp_data2 = """
    local 1.1.2.3 as 65000;
    neighbor 1.1.2.1 as 65000;
    ipv4 {
            import all;
            export all;
    };
"""

bird_cfg.add_protocol(protocol = "bgp", name = "my_bgp1", data = bgp_data1)
bird_cfg.add_protocol(protocol = "bgp", name = "my_bgp2", data = bgp_data2)
bird_cfg.add_route(dst_cidr = "42.42.42.42/32", next_hop = "1.1.1.3")
bird_cfg.add_route(dst_cidr = "42.42.42.43/32", next_hop = "1.1.2.3")
cfg = bird_cfg.build_config() # build combined configuration

pybird_c = PyBirdClient(ip = 'localhost', port = 4509)
pybird_c.connect()
pybird_c.acquire()
pybird_c.set_config(new_cfg = cfg)                                       ❸

c.set_bird_node(node_port      = 0,                                      ❹
                mac            = "00:00:00:01:00:07",
                ipv4           = "1.1.1.3",
                ipv4_subnet    = 24,
                ipv6_enabled   = True,
                vlans          = [22],                                   ❺
                tpids          = [0x8100])

c.set_bird_node(node_port      = 1,
                mac            = "00:00:00:01:00:08",
                ipv4           = "1.1.2.3",
                ipv4_subnet    = 24,
                ipv6_enabled   = True)

pybird_c.check_protocols_up(['my_bgp1, my_bgp2'])                        ❻
pybird_c.release()                                                       ❼
pybird_c.disconnect()
c.disconnect()
```

❶ this is needed for software mode with one QUEUE, going to be replaced with multi-Rx

❷ using BirdCFGCreator in order to make bird configuration

❸ sending our new config

❹ create 2 veth's in bird namespace

5. dot1q is a configuration of veth not namespace
6. ensure bird works on our protocols
7. exiting gracefully PyBirdClient

The python script will create the 2 veth's and send bird a simple config file with bgp protocol and 2 static routes. You can check if it worked on the dut side - look for the two new bgp routes:

ASR1001-01

```
asr1001-01#show ip bgp
BGP table version is 67, local router ID is 1.1.2.1
Status codes: s suppressed, d damped, h history, * valid, > best, i - internal,
              r RIB-failure, S Stale, m multipath, b backup-path, f RT-Filter,
              x best-external, a additional-path, c RIB-compressed,
Origin codes: i - IGP, e - EGP, ? - incomplete
RPKI validation codes: V valid, I invalid, N Not found

     Network          Next Hop            Metric LocPrf Weight Path
 * i 42.42.42.42/32   1.1.1.3                       100      0 i
 * i 42.42.42.43/32   1.1.2.3                       100      0 i
```

2.20.7 Tutorial: Simple bird node adding

Goal
Demonstrate adding a bird node as describrd in "add_bird_node.py".

As for now, in order to add a bird node you need to connect and acquire the wanted port/s and set service mode on. Afterwards you can use "set_bird_node" method as follows:

```
c = STLClient(verbose_level = 'debug')                          (1)
my_ports = [0, 1]
c.connect()
c.acquire(ports = my_ports)
c.set_port_attr(promiscuous=True) # to get the packet back to namespace
c.set_service_mode(ports = my_ports, enabled = True)

# create 2 veth's in bird namespace
c.set_bird_node(node_port      = 0,                             (2)
                mac            = "00:00:00:01:00:07",           (3)
                ipv4           = "1.1.1.3",                     (4)
                ipv6_enabled   = True,
                ipv4_subnet    = 24)


c.wait_for_protocols(['my_bgp1'])                               (5)
c.disconnect()
```

1. preparation for set_bird_node as described above.
2. the port which bird node will be set on.
3. the mac addreass for the new bird node, need to be unique.
4. the ipv4 for the new node, MUST be on the same subnet with dut interface.
5. blocking until bird confirms connection established on "my_bgp1" protocol.

Warning
In case of adding a bird node with the same parameters - nothing will happen. But in case of adding one with different parameters an exception will be raised - mac addresses are unique.

2.20.8 Tutorial: Advanced bird node adding

Goal
Demonstrate more advanced way of adding a bird node.

"set_bird_node" method flow is:

1. Check if a bird node with that mac already exists.
2. Set "promiscuous" mode on.
3. Remove all the namespaces and nodes from the node_port we want bird to work on.
4. Create the bird node.
5. Set ipv4 with a subnet.
6. Enable ipv6.
7. Send all commands asynchronously.
8. Wait for the commands to finished.

You may alter the steps as you wish i.e do not set ipv6 at all. Although you must set "promiscuous" mode on. Notice that for each command we added "is_bird" parameter so the call will look like:

```
...
cmds.add_node(mac, is_bird=True)
...
```

2.20.9 BirdCFGCreator

Goal
Create the final `bird.conf` file with the routing protocols and the routes themselves (as static routes).

In the constructor you may provide your own cfg file as a string, in case you don't provide one it will use our default cfg contains only "device protocol" scanning new devices every 1 second and a dummy "router id" ip (this is only for bird running for the first time with no aviable interface).

see: Stateless Python API - Bird CFG Creator

Note
When supplying bird config file or any data, it must be valid according to Bird's Documentation

2.20.9.1 Adding Routing Protocols

According to Bird's Protocols Documentation you may add protocols like:

```
bird_cfg = BirdCFGCreator()   # using default cfg
bgp_data1 = """
    local 1.1.1.3 as 65000;
    neighbor 1.1.1.1 as 65000;
    ipv4 {
            import all;
            export all;
    };
"""
bird_cfg.add_protocol(protocol = "bgp", name = "my_bgp1", data = bgp_data1)
```

2.20.9.2 Adding Routes

Routes are splitting to 2 types: simple and extended. Simple routes are routes with "next_hop" parameter that can be an ip or interface name. Extended are the routes with more complicated "next_hop", examples bellow.

Here is a simple route addition:

```
bird_cfg = BirdCFGCreator()   # using default cfg
bird_cfg.add_route(dst_cidr = "42.42.42.42/32", next_hop = "1.1.1.3")
```

And here is an extended route addition:

```
bird_cfg = BirdCFGCreator()   # using default cfg
next_hop = """
198.51.100.100 {
                ospf_metric2 = 100;      # Set extended attribute
                ospf_tag = 2;            # Set extended attribute
                bfd;                     # BFD-controlled route
        }
"""
bird_cfg.add_route(dst_cidr = "42.42.42.42/32", next_hop = next_hop)
```

2.20.9.3 Creating The Final Configuration

When you want to create the config file sending it to bird, you can merge everything:

```
bird_config = bird_cfg.build_config()
```

Now you have your final string with all your wanted protocols and routes ready to send to bird using PyBird Client.

2.20.10 PyBird Client

Goal
: Communicating with PyBird Server located on TRex machine using JSON-RPC over ZMQ.

see: Stateless Python API - PyBird Client

2.20.10.1 General Flow

1. connect - connect to server.
2. acquire - get handler from server.
3. some commands like: get_config, set_config e.t.c
4. relase - let go the handler from server.
5. disconnect - end the session.

2.20.10.2 Commands

Commands in PyBird are divided to: query and commands. In case of using a query like: `get_config` connect will be enough. In case of using a command like: `set_config` you have to connect and acquire. The TRex client object hides this object but it is possible to use it directly

2.20.10.3 Flow Example

```
b.connect()                     (1)
b.acquire()                     (2)

b.get_config()                  (3)
b.get_protocols_info()

b.set_empty_config()            (4)

b.release()                     (5)
b.disconnect()                  (6)
```

(1) connect to PyBird server, required for every method.

(2) acquire handler from server, required for command methods.

(3) using some query methods.

(4) setting emtpy config file, command method.

(5) release the acquired handler.

(6) disconnect from server.

Note
PyBird Client will release & disconnect anyway when the object will be destroyed. But it's not recommended to rely on that.

2.20.10.4 PyBird performance numbers

Table 2.16: BGP

Routes	time(sec)	ASR1K CPU%
10	1	1 %
1,000,000	20	100% 70K route/sec

2.20.10.5 Bird Console

If you want to communicate directly with bird you may use "bird console". simply run:

```
cd scripts/bird
./birdcl -s /tmp/trex-bird/bird.ctl   #1
```

1 default bird socket file path

You can learn more of bird consol commands at Bird Remote Constrol Doc.

Warning
Do not apply commands that alter bird configuration while TRex is running with bird server, only use it for query commands.

2.21 Appendix

2.21.1 Scapy packet examples

```
# UDP header
Ether()/IP(src="16.0.0.1",dst="48.0.0.1")/UDP(dport=12,sport=1025)

# UDP over one vlan
Ether()/Dot1Q(vlan=12)/IP(src="16.0.0.1",dst="48.0.0.1")/UDP(dport=12,sport=1025)

# UDP QinQ
Ether()/Dot1Q(vlan=12)/Dot1Q(vlan=12)/IP(src="16.0.0.1",dst="48.0.0.1")/UDP(dport=12,sport ←
    =1025)

#TCP over IP over VLAN
Ether()/Dot1Q(vlan=12)/IP(src="16.0.0.1",dst="48.0.0.1")/TCP(dport=12,sport=1025)

# IPv6 over vlan
Ether()/Dot1Q(vlan=12)/IPv6(src="::5")/TCP(dport=12,sport=1025)

#Ipv6 over UDP over IP
Ether()/IP()/UDP()/IPv6(src="::5")/TCP(dport=12,sport=1025)

#DNS packet
Ether()/IP()/UDP()/DNS()

#HTTP packet
Ether()/IP()/TCP()/"GET / HTTP/1.1\r\nHost: www.google.com\r\n\r\n"
```

2.21.2 HLT supported Arguments

2.21.2.1 connect

Argument	Default	Comment
device	localhost	*ip or hostname of TRex*
trex_rpc_port	None	*TRex extention: RPC port of TRex server (for several TRexes under same OS)*
trex_pub_port	None	*TRex extention: Publisher port of TRex server (for several TRexes under same OS)*

Argument	Default	Comment
trex_timeout_sec	None	*TRex extention: Timeout of rpc/pub connections*
port_list	None	*list of ports*
username	TRexUser	
reset	True	
break_locks	False	

2.21.2.2 cleanup_session

Argument	Default	Comment
maintain_lock	False	*release ports at the end or not*
port_list	None	
port_handle	None	

2.21.2.3 traffic_config

Argument	Default	Comment
mode	None	*(create | modify | remove | reset)*
split_by_cores	split	*(split | duplicate | single) TRex extention: split = split traffic by cores, duplicate = duplicate traffic for all cores, single = run only with sinle core (not implemented yet)*
load_profile	None	*TRex extention: path to filename with stream profile (stream builder parameters will be ignored, limitation: modify)*
consistent_random	False	*TRex extention: False (default) = random sequence will be different every run, True = random sequence will be same every run*
ignore_macs	False	*TRex extention: True = use MACs from server configuration, no MAC VM (workaround on lack of ARP)*
disable_flow_stats	False	*TRex extention: True = don't use flow stats for this stream, (workaround for limitation on type of packet for flow_stats)*
flow_stats_id	None	*TRex extention: uint, for use of STLHltStream, specifies id for flow stats (see stateless manual for flow_stats details)*
port_handle	None	
port_handle2	None	
bidirectional	False	
stream builder parameters		
transmit_mode	continuous	*(continuous | multi_burst | single_burst)*
rate_pps	None	
rate_bps	None	
rate_percent	10	
stream_id	None	
name	None	
direction	0	*TRex extention: 1 = exchange sources and destinations, 0 = do nothing*
pkts_per_burst	1	
burst_loop_count	1	
inter_burst_gap	12	
length_mode	fixed	*(auto | fixed | increment | decrement | random | imix)*
l3_imix1_size	64	
l3_imix1_ratio	7	
l3_imix2_size	570	
l3_imix2_ratio	4	
l3_imix3_size	1518	
l3_imix3_ratio	1	
l3_imix4_size	9230	
l3_imix4_ratio	0	
L2		

Argument	Default	Comment
frame_size	64	
frame_size_min	64	
frame_size_max	64	
frame_size_step	1	
l2_encap	ethernet_ii	*(ethernet_ii | ethernet_ii_vlan)*
mac_src	00:00:01:00:00:01	
mac_dst	00:00:00:00:00:00	
mac_src2	00:00:01:00:00:01	
mac_dst2	00:00:00:00:00:00	
mac_src_mode	fixed	*(fixed | increment | decrement | random)*
mac_src_step	1	*we are changing only 32 lowest bits*
mac_src_count	1	
mac_dst_mode	fixed	*(fixed | increment | decrement | random)*
mac_dst_step	1	*we are changing only 32 lowest bits*
mac_dst_count	1	
mac_src2_mode	fixed	*(fixed | increment | decrement | random)*
mac_src2_step	1	
mac_src2_count	1	
mac_dst2_mode	fixed	*(fixed | increment | decrement | random)*
mac_dst2_step	1	
mac_dst2_count	1	
vlan options below can have multiple values for nested Dot1Q headers		
vlan_user_priority	1	
vlan_priority_mode	fixed	*(fixed | increment | decrement | random)*
vlan_priority_count	1	
vlan_priority_step	1	
vlan_id	0	
vlan_id_mode	fixed	*(fixed | increment | decrement | random)*
vlan_id_count	1	
vlan_id_step	1	
vlan_cfi	1	
vlan_protocol_tag_id	None	
L3, general		
l3_protocol	None	*(ipv4 | ipv6)*
l3_length_min	110	
l3_length_max	238	
l3_length_step	1	
L3, IPv4		
ip_precedence	0	
ip_tos_field	0	
ip_mbz	0	
ip_delay	0	
ip_throughput	0	
ip_reliability	0	
ip_cost	0	
ip_reserved	0	
ip_dscp	0	
ip_cu	0	
l3_length	None	
ip_id	0	
ip_fragment_offset	0	
ip_ttl	64	
ip_checksum	None	
ip_src_addr	0.0.0.0	
ip_dst_addr	192.0.0.1	

Argument	Default	Comment
ip_src_mode	fixed	*(fixed | increment | decrement | random)*
ip_src_step	1	*ip or number*
ip_src_count	1	
ip_dst_mode	fixed	*(fixed | increment | decrement | random)*
ip_dst_step	1	*ip or number*
ip_dst_count	1	
L3, IPv6		
ipv6_traffic_class	0	
ipv6_flow_label	0	
ipv6_length	None	
ipv6_next_header	None	
ipv6_hop_limit	64	
ipv6_src_addr	fe80:0:0:0:0:0:0:12	
ipv6_dst_addr	fe80:0:0:0:0:0:0:22	
ipv6_src_mode	fixed	*(fixed | increment | decrement | random)*
ipv6_src_step	1	*we are changing only 32 lowest bits; can be ipv6 or number*
ipv6_src_count	1	
ipv6_dst_mode	fixed	*(fixed | increment | decrement | random)*
ipv6_dst_step	1	*we are changing only 32 lowest bits; can be ipv6 or number*
ipv6_dst_count	1	
L4, TCP		
l4_protocol	None	*(tcp | udp)*
tcp_src_port	1024	
tcp_dst_port	80	
tcp_seq_num	1	
tcp_ack_num	1	
tcp_data_offset	5	
tcp_fin_flag	0	
tcp_syn_flag	0	
tcp_rst_flag	0	
tcp_psh_flag	0	
tcp_ack_flag	0	
tcp_urg_flag	0	
tcp_window	4069	
tcp_checksum	None	
tcp_urgent_ptr	0	
tcp_src_port_mode	increment	*(increment | decrement | random)*
tcp_src_port_step	1	
tcp_src_port_count	1	
tcp_dst_port_mode	increment	*(increment | decrement | random)*
tcp_dst_port_step	1	
tcp_dst_port_count	1	
L4, UDP		
udp_src_port	1024	
udp_dst_port	80	
udp_length	None	
udp_dst_port_mode	increment	*(increment | decrement | random)*
udp_src_port_step	1	
udp_src_port_count	1	
udp_src_port_mode	increment	*(increment | decrement | random)*
udp_dst_port_step	1	
udp_dst_port_count	1	

2.21.2.4 traffic_control

Argument	Default	Comment
action	None	*(clear_stats | run | stop | sync_run | poll | reset)*
port_handle	None	

2.21.2.5 traffic_stats

Argument	Default	Comment
mode	aggregate	*(all | aggregate | streams)*
port_handle	None	

2.21.3 FD.IO open source project using TRex

here

2.21.4 Using Stateless client via JSON-RPC

For functions that do not require complex objects and can use JSON-serializable input/output, you can use Stateless API via JSON-RPC proxy server.
Thus, you can use Stateless TRex **from any language** supporting JSON-RPC.

2.21.4.1 How to run TRex side:

- Run the Stateless TRex server in one of 2 ways:
  - Run TRex directly in shell:

    ```
    [bash]>sudo ./t-rex-64 -i
    ```

  - Run TRex via JSON-RPC command to trex_daemon_server:

    ```
    start_trex(trex_cmd_options, user, block_to_success = True, timeout = 40, stateless = ↩
        True)
    ```

- Run the RPC "proxy" to stateless in one of 2 ways:
  - Run directly:

    ```
    cd automation/trex_control_plane/interactive/trex/examples/stl
    python rpc_proxy_server.py
    ```

  - Send JSON-RPC command to master_daemon:

    ```
    if not master_daemon.is_stl_rpc_proxy_running():
        master_daemon.start_stl_rpc_proxy()
    ```

Done :)

Now you can send requests to the rpc_proxy_server and get results as an array of 2 values:

- If fail, result will be: [False, <traceback log with error>]
- If success, result will be: [True, <return value of called function>]

In the same directory of rpc_proxy_server.py, there is Python example of usage: **using_rpc_proxy.py**

2.21.4.2 Native Stateless API functions:

- acquire
- connect
- disconnect
- get_stats
- get_warnings
- push_remote
- reset
- wait_on_traffic

These functions can be called directly as server.push_remote('udp_traffic.pcap'). If you need any other function of a stateless client, you can either add it to rpc_proxy_server.py, or use this method:
server.**native_method**(<string of function name>, <args of the function>)

2.21.4.3 HLTAPI Methods can be called here as well:

- connect
- cleanup_session
- interface_config
- traffic_config
- traffic_control
- traffic_stats

Note
In case of name collision with native functions (such as connect), for HLTAPI, function will change to have "hlt_" prefix.

2.21.4.4 Example of running from Java:

```
package com.cisco.trex_example;

import java.net.URL;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Map;
import java.util.HashMap;

import com.googlecode.jsonrpc4j.JsonRpcHttpClient;

public class TrexMain {

    @SuppressWarnings("rawtypes")
    public static Object verify(ArrayList response) {
        if ((boolean) response.get(0)) {
            return response.get(1);
        }
```

```
        System.out.println("Error: " + response.get(1));
        System.exit(1);
        return null;
    }

    @SuppressWarnings("rawtypes")
    public static void main(String[] args) throws Throwable {
        try {
            String trex_host = "csi-trex-11";
            int rpc_proxy_port = 8095;
            Map<String, Object> kwargs = new HashMap<>();
            ArrayList<Integer> ports = new ArrayList<Integer>();
            HashMap res_dict = new HashMap<>();
            ArrayList res_list = new ArrayList();
            JsonRpcHttpClient rpcConnection = new JsonRpcHttpClient(new URL("http://" +  ↩
                trex_host + ":" + rpc_proxy_port));

            System.out.println("Initializing Native Client");
            kwargs.put("server", trex_host);
            kwargs.put("force", true);
            verify(rpcConnection.invoke("native_proxy_init", kwargs, ArrayList.class));
            kwargs.clear();

            System.out.println("Connecting to TRex server");
            verify(rpcConnection.invoke("connect", kwargs, ArrayList.class));

            System.out.println("Resetting all ports");
            verify(rpcConnection.invoke("reset", kwargs, ArrayList.class));

            System.out.println("Getting ports info");
            kwargs.put("func_name", "get_port_info"); // some "custom" function
            res_list = (ArrayList) verify(rpcConnection.invoke("native_method", kwargs,  ↩
                ArrayList.class));
            System.out.println("Ports info is: " + Arrays.toString(res_list.toArray()));
            kwargs.clear();
            for (int i = 0; i < res_list.size(); i++) {
                    Map port = (Map) res_list.get(i);
                    ports.add((int)port.get("index"));
                    }

            System.out.println("Sending pcap to ports: " + Arrays.toString(ports.toArray()) ↩
                );
            kwargs.put("pcap_filename", "stl/sample.pcap");
            verify(rpcConnection.invoke("push_remote", kwargs, ArrayList.class));
            kwargs.clear();
            verify(rpcConnection.invoke("wait_on_traffic", kwargs, ArrayList.class));

            System.out.println("Getting stats");
            res_dict = (HashMap) verify(rpcConnection.invoke("get_stats", kwargs, ArrayList ↩
                .class));
            System.out.println("Stats: " + res_dict.toString());

            System.out.println("Deleting Native Client instance");
            verify(rpcConnection.invoke("native_proxy_del", kwargs, ArrayList.class));

        } catch (Throwable e) {
            e.printStackTrace();
        }
    }
}
```

2.21.4.5 Bird Compilation

Just add "--with-bird" flag to configure command like so:

```
[bash]> ./b configure --with-bird
[bash]> ./b build
```

From Bird's Documentation:

For compiling BIRD you need these programs and libraries:

- GNU C Compiler (or LLVM Clang)
- GNU Make
- GNU Bison
- GNU M4
- Flex
- ncurses library
- GNU Readline library
- libssh library (optional, for RPKI-Router protocol)

Note

In case of compilation problems, try build with "-v" flag in order to see bird compilation output.
