TRex

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
2.1	2017-02-21-a		H

Contents

Chapter 1

Introduction

1.1 A word on traffic generators

Traditionally, routers have been tested using commercial traffic generators, while performance typically has been measured using packets per second (PPS) metrics. As router functionality and services have become more complex, stateful traffic generators have become necessary to provide more realistic traffic scenarios.

Advantages of realistic traffic generators:

- Accurate performance metrics.
- Discovering bottlenecks in realistic traffic scenarios.

1.1.1 Current Challenges:

- **Cost**: Commercial stateful traffic generators are very expensive.
- **Scale**: Bandwidth does not scale up well with feature complexity.
- **Standardization**: Lack of standardization of traffic patterns and methodologies.
- **Flexibility**: Commercial tools are not sufficiently agile when flexibility and customization are needed.

1.1.2 Implications

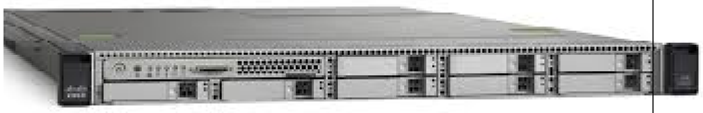
- High capital expenditure (capex) spent by different teams.
- Testing in low scale and extrapolation became a common practice. This is non-ideal and fails to indicate bottlenecks that appear in real-world scenarios.
- Teams use different benchmark methodologies, so results are not standardized.
- Delays in development and testing due to dependence on testing tool features.
- Resource and effort investment in developing different ad hoc tools and test methodologies.

1.2 Overview of TRex

TRex addresses the problems associated with commercial stateful traffic generators, through an innovative and extendable software implementation, and by leveraging standard and open software and x86/UCS hardware.

- Generates and analyzes L4-7 traffic. In one package, provides capabilities of commercial L7 tools.
- Stateful traffic generator based on pre-processing and smart replay of real traffic templates.
- Generates and **amplifies** both client- and server-side traffic.
- Customized functionality can be added.
- Scales to 200Gb/sec for one UCS (using Intel 40Gb/sec NICs).
- Low cost.
- Self-contained package that can be easily installed and deployed.
- Virtual interface support enables TRex to be used in a fully virtual environment without physical NICs. Example use cases:
  - Amazon AWS
  - Cisco LaaS
  - TRex on your laptop

Table 1.1: TRex Hardware

Cisco UCS Platform	**Intel NIC**
	

1.3 Purpose of this guide

This guide explains the use of TRex internals and the use of TRex together with Cisco ASR1000 Series routers. The examples illustrate novel traffic generation techniques made possible by TRex.

Chapter 2

Download and installation

2.1 Hardware recommendations

TRex is a Linux application, interacting with Linux kernel modules. It uses DPDK (there is **no** need to install DPDK as a library). TRex should work on any COTS x86 server (it can be compiled to ARM but not tested in our regression). Our regression setups uses Cisco UCS hardware for high performance low latency use cases. The following platforms have been tested and are recommended for operating TRex.

Another option to run TRex for low performance and low footprint (~1MPPS limited by the kernel) is to use the kernel interfaces in raw socket mode (require super user). In this way TRex can run almost on any Linux platform and any Linux interfaces (e.g. veth/tap/physical wireless interfaces. tun interfaces won't work as there is no L2). Docker example is provided Docker for more info see low footprint [low_end] and Linux interfaces [linux_interfaces]

Note

```
Not all supported DPDK interfaces are supported by TRex.
```

Table 2.1: Preferred UCS hardware

UCS Type	Comments
UCS C220 Mx	**Preferred Low-End**. Supports up to 40Gb/sec with 540-D2. With newer Intel NIC (recommended), supports 80Gb/sec with 1RU. See table below describing components.
UCS C200	Early UCS model.
UCS C210 Mx	Supports up to 40Gb/sec PCIe3.0.
UCS C240 Mx	**Preferred, High-End** Supports up to 200Gb/sec. 6x XL710 NICS (PCIex8) or 2xFM10K (PCIex16). See table below describing components.
UCS C260M2	Supports up to 30Gb/sec (limited by V2 PCIe).

Table 2.2: Low-end UCS C220 Mx - Internal components

Components	Details
CPU	2x E5-2620 @ 2.0 GHz.
CPU Configuration	2-Socket CPU configurations (also works with 1 CPU).
Memory	2x4 banks f.or each CPU. Total of 32GB in 8 banks.

Table 2.2: (continued)

Components	Details
RAID	No RAID.

Table 2.3: High-end C240 Mx - Internal components

Components	Details
CPU	2x E5-2667 @ 3.20 GHz.
PCIe	1x Riser PCI expansion card option A PID UCSC-PCI-1A-240M4 enables 2 PCIex16.
CPU Configuration	2-Socket CPU configurations (also works with 1 CPU).
Memory	2x4 banks for each CPU. Total of 32GB in 8 banks.
RAID	No RAID.
Riser 1/2	Both left and right should support x16 PCIe. Right (Riser1) should be from option A x16 and Left (Riser2) should be x16. Need to order both.

Important
In all bare metal cases, it's important to have 4 DRAM channels. Fewer channels will impose a performance issue. To test it you can run `sudo dmidecode -t memory | grep CHANNEL` and check CHANNEL x

Table 2.4: Supported NICs

Chipset	Bandwidth (Gb/sec)	LSO	LRO	Example
Any Kernel Linux interface	x	x	x	veth,tap,tun,eth0,wireless interface, up to ~1MPPS, one thread. see low footprint [low_end] and Linux interfaces [linux_interfaces]
Intel I350	1	+	-	Intel 4x1GE 350-T4 NIC
Intel 82599	0.1/1/2.5/5/10	+	+	Cisco part ID:N2XX-AIPCI01 Intel x520-D2, Intel X520 Dual Port 10Gb SFP+ Adapter. X550-T2,x540-T2 for tbase Cisco part ID:UCSC-PCIE-ID10GC
Intel 82599 VF	x	+	+	
Intel X710	10	+	-	Cisco part ID:UCSC-PCIE-IQ10GF SFP+, **Preferred** support per stream stats in hardware Silicom PE310G4i71L
Intel XL710	40	+	-	Cisco part ID:UCSC-PCIE-ID40GF, QSFP+ (copper/optical) **Preferred** support per stream stats in hardware
Intel XXV710	1/10/25	+	-	SFP28 Intel XXV710 **Preferred** support per stream stats in hardware

Table 2.4: (continued)

Chipset	Bandwidth (Gb/sec)	LSO	LRO	Example
Intel XL710/X710 VF	x	+	-	
Napatech SmartNICs NT40E3-4	10	-	-	./b configure --with-ntacc to build the library
Napatech SmartNICs NT80E3-2	40	-	-	./b configure --with-ntacc to build the library
Napatech SmartNICs NT100E3-1	100	-	-	./b configure --with-ntacc to build the library. The only Napatech NIC in our regression. more info [napatech_support]
Napatech SmartNICs NT200A01	100	-	-	./b configure --with-ntacc to build the library **Note:** This NIC require a BIOS with PCIe bifurcation support. PCIe bifurcation
Mellanox ConnectX-4/Lx	25/40/50/56/100	+	+	SFP28/QSFP28, ConnectX-4 ConnectX-4-brief (copper/optical) supported from v2.11 more details and issues TRex Support [connectx_support]
Mellanox ConnectX-5	25/40/50/56/100	+	+	Supported, see issues TRex Support [connectx_support]
Cisco 1300 series	40	+	-	QSFP+, VIC 1380, VIC 1385, VIC 1387 see more TRex Support [ciscovic_support]
VMXNET3	VMware paravirtualized	+	-	Connect using VMware vSwitch
E1000	paravirtualized	+	-	VMware/KVM/VirtualBox
Virtio	paravirtualized	+	-	KVM
Amazon ENA	paravirtualized	+	-	Amazon Cloud
MS Failsafe	paravirtualized	+	-	Azure with DPDK mlx5 support
memif	shared memory rings	+	-	see memif support multi core for STL. ASTF only one core

Note

LSO (Large Send Offload) and LRO (Large Receive Offload) are techniques to increase egress and ingress throughput in high-bandwidth network connections by reducing CPU overheard. These are generally accomplished using multipacket buffering. LSO is also known as TCP segmentation offload (TSO) when applied to TCP, or generic segmentation offload (GSO). See Large Receive Offload and link:https://en.wikipedia.org/wiki/Large_send_offload(Large Send Offload) on wikipedia for further information.

Table 2.5: SFP+ support

SFP+	Intel Ethernet Converged X710-DAX	Silicom PE310G4i71L (Open optic)	82599EB 10-Gigabit
Cisco SFP-10G-SR	Not supported	**Supported**	**Supported**

Table 2.5: (continued)

SFP+	**Intel Ethernet Converged X710-DAX**	**Silicom PE310G4i71L (Open optic)**	**82599EB 10-Gigabit**
Cisco SFP-10G-LR	Not supported	**Supported**	**Supported**
Cisco SFP-H10GB-CU1M	**Supported**	**Supported**	**Supported**
Cisco SFP-10G-AOC1M	**Supported**	**Supported**	**Supported**

Note

```
Intel X710 NIC (example: FH X710DA4FHBLK) operates *only* with Intel SFP+. For ↩
    open optic, use the Silicom PE310G4i71L NIC, available here:
http://www.silicom-usa.com/PE310G4i71L_Quad_Port_Fiber_SFP+ ↩
    _10_Gigabit_Ethernet_PCI_Express_Server_Adapter_49
```

Note

```
For Intel XXV710 Cisco NIC, make sure to upgrade cimc to latest at lest v4.0(2f) ↩
    , the firmware of the XXV710 will be upgraded to v6 that can support most of ↩
    sfps.
```

Table 2.6: XL710 NIC base QSFP+ support

QSFP+	**Intel Ethernet Converged XL710-QDAX**	**Silicom PE340G2Qi71 Open optic**
QSFP+ SR4 optics	**Supported**: APPROVED OPTICS. **Not supported**: Cisco QSFP-40G-SR4-S	**Supported**: Cisco QSFP-40G-SR4-S
QSFP+ LR-4 Optics	**Supported**: APPROVED OPTICS. **Not supported**: Cisco QSFP-40G-LR4-S	**Supported**: Cisco QSFP-40G-LR4-S
QSFP Active Optical Cables (AoC)	**Supported**: Cisco QSFP-H40G-AOC	**Supported**: Cisco QSFP-H40G-AOC
QSFP+ Intel Ethernet Modular Supported	N/A	N/A
QSFP+ DA twin-ax cables	N/A	N/A
Active QSFP+ Copper Cables	**Supported**: Cisco QSFP-4SFP10G-CU	**Supported**: Cisco QSFP-4SFP10G-CU

Note

```
For Intel XL710 NICs, Cisco SR4/LR QSFP+ does not operate. Use Silicom with Open ↩
    Optic.
```

Table 2.7: ConnectX-4 NIC base QSFP28 support (100gb)

QSFP28	ConnectX-4
QSFP28 SR4 optics	N/A
QSFP28 LR-4 Optics	N/A
QSFP28 (AoC)	**Supported**: Cisco QSFP-100G-AOCxM
QSFP28 DA twin-ax cables	**Supported**: Cisco QSFP-100G-CUxM

Table 2.8: Cisco VIC NIC base QSFP+ support

QSFP+	Intel Ethernet Converged XL710-QDAX
QSFP+ SR4 optics	N/A
QSFP+ LR-4 Optics	N/A
QSFP Active Optical Cables (AoC)	**Supported**: Cisco QSFP-H40G-AOC
QSFP+ Intel Ethernet Modular Optics	N/A
QSFP+ DA twin-ax cables	N/A
Active QSFP+ Copper Cables	N/A

Table 2.9: FM10K QSFP28 support

QSFP28	Example
todo	todo

Important

- Intel SFP+ 10Gb/sec is the only one supported by default on the standard Linux driver. TRex also supports Cisco 10Gb/sec SFP+.
- For operating high speed throughput (example: several Intel XL710 40Gb/sec), use different NUMA nodes for different NICs.
  To verify NUMA and NIC topology: `lstopo (yum install hwloc)`
  To display CPU info, including NUMA node: `lscpu`
  NUMA usage: example [numa-example]
- For the Intel XL710 NIC, verify that the NVM is v5.04. Info [xl710-firmware].
  - `> sudo ./t-rex-64 -f cap2/dns.yaml -d 0 *-v 6* --nc | grep NVM`
    `PMD:FW 5.0 API 1.5 NVM 05.00.04 eetrack 800013fc`

Table 2.10: Sample order for recommended low-end Cisco UCSC-C220-M3S with 4x10Gb ports

Component	Quantity
UCSC-C220-M3S	1
UCS-CPU-E5-2650	2
UCS-MR-1X041RY-A	8

Table 2.10: (continued)

Component	Quantity
A03-D500GC3	1
N2XX-AIPCI01	2
UCSC-PSU-650W	1
SFS-250V-10A-IS	1
UCSC-CMA1	1
UCSC-HS-C220M3	2
N20-BBLKD	7
UCSC-PSU-BLKP	1
UCSC-RAIL1	1

Note
Purchase the 10Gb/sec SFP+ separately. Cisco would be fine with TRex (but not for plain Linux driver).

2.2 Installing OS

2.2.1 Supported versions

Supported Linux versions:

- CentOS/RHEL 7.6, 64-bit kernel (not 32-bit) (**Recommended**) — This is the only working option for ConnectX-4.

Note
Additional OS versions may be supported by compiling the necessary drivers.

To check whether a kernel is 64-bit, verify that the ouput of the following command is `x86_64`.

```
[bash]>uname -m
x86_64
```

2.2.2 Verify Intel NIC installation

Use `lspci` to verify the NIC installation.

Example: 4x 10Gb/sec TRex configuration (see output below):

- I350 management port
- 4x Intel Ethernet Converged Network Adapter model x520-D2 (82599 chipset)

```
[bash]>lspci | grep Ethernet
01:00.0 Ethernet controller: Intel Corporation I350 Gigabit Network Connection (rev 01)  ↩
                   #❶
01:00.1 Ethernet controller: Intel Corporation I350 Gigabit Network Connection (rev 01)  ↩
                   #❷
03:00.0 Ethernet controller: Intel Corporation 82599EB 10-Gigabit SFI/SFP+ Network  ↩
    Connection (rev 01) #❸
```

```
03:00.1 Ethernet controller: Intel Corporation 82599EB 10-Gigabit SFI/SFP+ Network ↩
    Connection (rev 01)
82:00.0 Ethernet controller: Intel Corporation 82599EB 10-Gigabit SFI/SFP+ Network ↩
    Connection (rev 01)
82:00.1 Ethernet controller: Intel Corporation 82599EB 10-Gigabit SFI/SFP+ Network ↩
    Connection (rev 01)
```

1. Management port
2. CIMC port
3. 10Gb/sec traffic ports (Intel 82599EB)

2.3 Obtaining the TRex package

Use `ssh` to connect to the TRex machine and execute the commands described below.

Note
Only **https** is supported

Latest (stable) release:

```
[bash]>mkdir -p /opt/trex
[bash]>cd /opt/trex
[bash]>wget --no-cache https://trex-tgn.cisco.com/trex/release/latest
[bash]>tar -xzvf latest
```

Latest (bleeding edge) version:

```
[bash]>wget --no-cache https://trex-tgn.cisco.com/trex/release/be_latest
```

To obtain a specific version:

```
[bash]>wget --no-cache https://trex-tgn.cisco.com/trex/release/vX.XX.tar.gz #1
```

1. where X.XX = major.minor version number as per https://trex-tgn.cisco.com/trex/doc/release_notes.html

Chapter 3

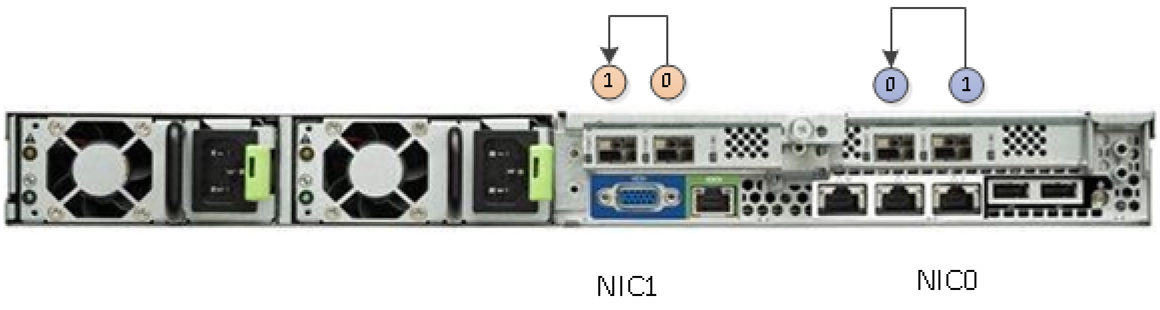
First time Running

3.1 Configuring for loopback

Before connecting TRex to your DUT, it is strongly advised to verify that TRex and the NICs work correctly in loopback.

Note

1. For best performance, loopback the interfaces on the same NUMA (controlled by the same physical processor). If you are unable to check this, proceed without this step.
2. If you are using a 10Gbs NIC based on an Intel 520-D2 NIC, and you loopback ports on the same NIC using SFP+, the device might not sync, causing a failure to link up.
   Many types of SFP+ (Intel/Cisco/SR/LR) have been verified to work.
   If you encounter link issues, try to loopback interfaces from different NICs, or use Cisco twinax copper cable.



Loopback example

3.1.1 Identify the ports

Use the following command to identify ports.

```
[bash]>sudo ./dpdk_setup_ports.py -s

 Network devices using DPDK-compatible driver
 ============================================

 Network devices using kernel driver
 ===================================
 0000:03:00.0 '82599ES 10-Gigabit SFI/SFP+ Network Connection' drv= unused=ixgb #❶
```

```
0000:03:00.1 '82599ES 10-Gigabit SFI/SFP+ Network Connection' drv= unused=ixgb
0000:13:00.0 '82599ES 10-Gigabit SFI/SFP+ Network Connection' drv= unused=ixgb
0000:13:00.1 '82599ES 10-Gigabit SFI/SFP+ Network Connection' drv= unused=ixgb
0000:02:00.0 '82545EM Gigabit Ethernet Controller (Copper)' if=eth2 drv=e1000 unused= ↩
    igb_uio *Active* #❷

Other network devices
=====================
<none>
```

❶ If you have not run any DPDK applications, the command output shows a list of interfaces bound to the kernel or not bound at all.

❷ The interface marked *active* is the one used by your ssh connection. **Never** put this interface into TRex config file.

Choose the ports to use and follow the instructions in the next section to create a configuration file.

3.1.2 Creating minimum configuration file

Default configuration file name: `/etc/trex_cfg.yaml`.

For a full list of YAML configuration file options, see Platform YAML [trex_config_yaml_config_file].

For many purposes, it is convenient to begin with a copy of the basic configuration file template, available in the cfg folder:

```
[bash]>cp  cfg/simple_cfg.yaml /etc/trex_cfg.yaml
```

Next, edit the configuration file, adding the interface and IP address details.

Example:

```
<none>
- port_limit      : 2
  version         : 2
#List of interfaces. Change according to your setup. Use ./dpdk_setup_ports.py -s to see  ↩
    available options.
interfaces    : ["03:00.0", "03:00.1"]   #❶
port_info       :  # Port IPs. Change according to your needs. In case of loopback, you can ↩
     leave as is.
          - ip         : 1.1.1.1
            default_gw : 2.2.2.2
          - ip         : 2.2.2.2
            default_gw : 1.1.1.1
```

❶ Edit this line to match the interfaces you are using. All NICs must have the same type - do not mix different NIC types in one config file. For more info, see trex-201.

3.2 Script for creating config file

A script is available to automate the process of tailoring the basic configuration file to your needs. The script gets you started, and then you can then edit the resulting configuration file directly for advanced options. For details, see Platform YAML [trex_config_yaml_config_file].

There are two ways to run the script:

- Interactive mode: Script prompts you for parameters.
- Command line mode: Provide all parameters using command line options.

3.2.1 Interactive mode

The script provides a list of available interfaces with interface-related information. Follow the instructions to create a basic config file.

```
[bash]>sudo ./dpdk_setup_ports.py -i
```

3.2.2 Command line mode

Run the following command to display a list of all interfaces and interface-related information:

```
[bash]>sudo ./dpdk_setup_ports.py -t
```

- In case of **Loopback** and/or only **L1-L2 Switches** on the way, IPs and destination MACs are not required. The script assumes the following interface connections: 0↔1, 2↔3 etc.

Run the following:

```
[bash]>sudo ./dpdk_setup_ports.py -c <TRex interface 0> <TRex interface 1> ...
```

- In case of a **Router** (or other next hop device, such as **L3 Switch**), specify the TRex IPs and default gateways, or MACs of the router, as described below.

Table 3.1: Command line options for the configuration file creation script (dpdk_setup_ports.py -c)

Argument	Description	Example
-c	Create a configuration file by specified interfaces (PCI address or Linux names: eth1 etc.)	-c 03:00.1 eth1 eth4 84:00.0
--dump	Dump created configuration to screen.	
-o	Output the configuration to a file.	-o /etc/trex_cfg.yaml
--dest-macs	Destination MACs to be used per each interface. Use this option for MAC-based configuration instead of IP-based. Do not use this option together with --ip and --def_gw	--dest-macs 11:11:11:11:11:11 22:22:22:22:22:22
--ip	List of IPs to use for each interface. If --ip and --dest-macs are not specified, the script assumes loopback connections (0↔1, 2↔3 etc.).	--ip 1.2.3.4 5.6.7.8
--def-gw	List of default gateways to use for each interface. When using the --ip option, also use the --def_gw option.	--def-gw 3.4.5.6 7.8.9.10
--ci	Cores include: White list of cores to use. Include enough cores for each NUMA.	--ci 0 2 4 5 6
--ce	Cores exclude: Black list of cores to exclude. When excluding cores, ensure that enough remain for each NUMA.	--ci 10 11 12
--no-ht	No HyperThreading: Use only one thread of each core specified in the configuration file.	
--prefix	(Advanced option) Prefix to be used in TRex configuration in case of parallel instances.	--prefix first_instance
--zmq-pub-port	(Advanced option) ZMQ Publisher port to be used in TRex configuration in case of parallel instances.	--zmq-pub-port 4000
--zmq-rpc-port	(Advanced option) ZMQ RPC port to be used in the TRex configuration in case of parallel instances.	--zmq-rpc-port
--ignore-numa	(Advanced option) Ignore NUMAs for configuration creation. This option may reduce performance. Use only if necessary - for example, in case of a pair of interfaces at different NUMAs.	

3.3 TRex on ESXi

General recommendation: For best performance, run TRex on "bare metal" hardware, without any type of VM. Bandwidth on a VM may be limited, and IPv6 may not be fully supported.

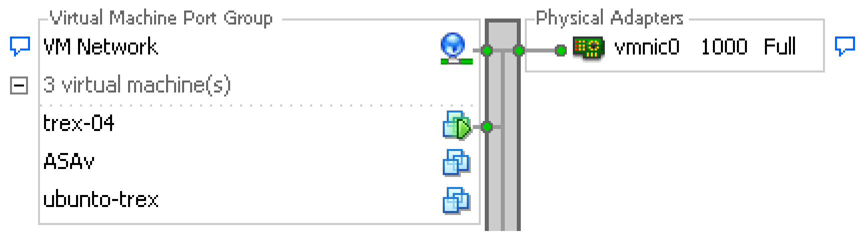
In special cases, it may be reasonable or advantageous to run TRex on VM:

- If you already have VM installed, and do not require high performance.
- Virtual NICs can be used to bridge between TRex and NICs not supported by TRex.

3.3.1 Configuring ESXi for running TRex

1. Click the host machine, then select Configuration → Networking.

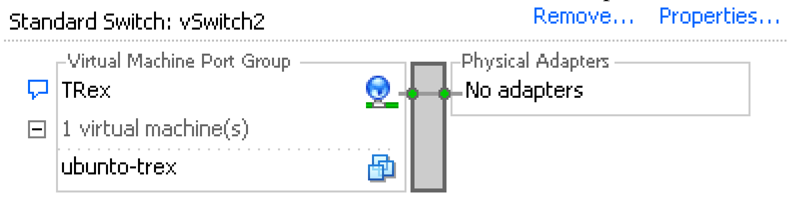
   a. One of the NICs must be connected to the main vSwitch network for an "outside" connection for the TRex client and ssh:



Virtual Machine Port Group
VM Network
3 virtual machine(s)
trex-04
ASAv
ubunto-trex

Physical Adapters
vmnic0 1000 Full

   b. Other NICs that are used for TRex traffic must be in a separate vSwitch:

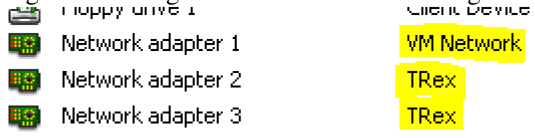


Standard Switch: vSwitch2

Virtual Machine Port Group
TRex
1 virtual machine(s)
ubunto-trex

Physical Adapters
No adapters

2. Right-click the guest machine → Edit settings → Ensure the NICs are set to their networks:



Network adapter 1	VM Network
Network adapter 2	TRex
Network adapter 3	TRex

Note
Before version 2.10, the following command did not function correctly:

```
sudo ./t-rex-64 -f cap2/dns.yaml --lm 1 --lo -l 1000 -d 100
```

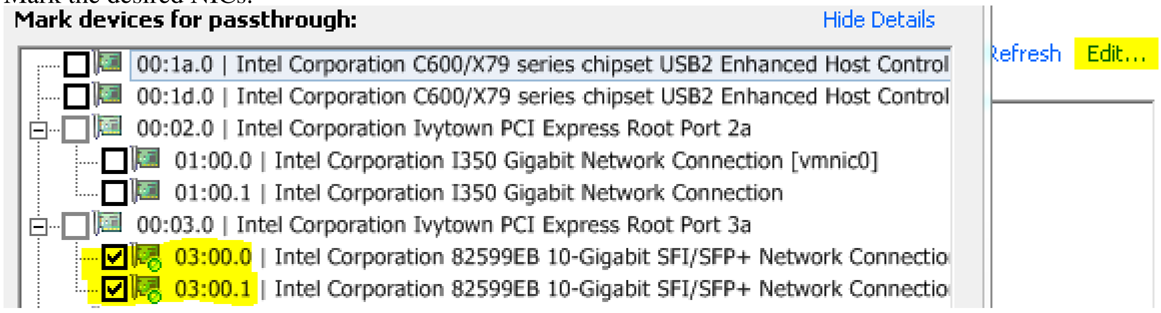
The vSwitch did not route packets correctly. This issue was resolved in version 2.10 when TRex started to support ARP.

3.3.2 Configuring Pass-through

Pass-through enables direct use of host machine NICs from within the VM. Pass-through access is generally limited only by the NIC/hardware itself, but there may be occasional spikes in latency (~10ms). Passthrough settings cannot be saved to OVA.

1. Click the host machine. Enter Configuration → Advanced settings → Edit.

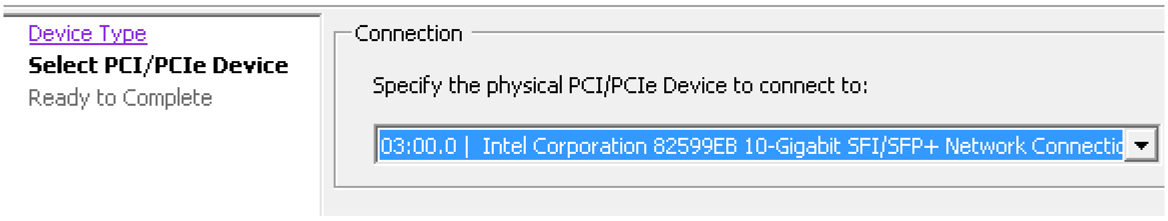
2. Mark the desired NICs.



00:1a.0 | Intel Corporation C600/X79 series chipset USB2 Enhanced Host Control
00:1d.0 | Intel Corporation C600/X79 series chipset USB2 Enhanced Host Control
00:02.0 | Intel Corporation Ivytown PCI Express Root Port 2a
01:00.0 | Intel Corporation I350 Gigabit Network Connection [vmnic0]
01:00.1 | Intel Corporation I350 Gigabit Network Connection
00:03.0 | Intel Corporation Ivytown PCI Express Root Port 3a
[✓] 03:00.0 | Intel Corporation 82599EB 10-Gigabit SFI/SFP+ Network Connectio
[✓] 03:00.1 | Intel Corporation 82599EB 10-Gigabit SFI/SFP+ Network Connectio

3. Reboot the ESXi to apply.

4. Right-click the guest machine. Edit settings → Add → **PCI device** → Select the NICs individually.



Specify the physical PCI/PCIe Device to connect to:

03:00.0 | Intel Corporation 82599EB 10-Gigabit SFI/SFP+ Network Connectic

3.4 Low-end machines

Starting from version 2.37, was added feature to run TRex on low-end machines like weak laptops.

To enable this mode, define low_end argument in platform config file:

```
- version: 2
  interfaces: ['82:00.0', '82:00.1']
  low_end: true                          ❶
  low_end_core: 3                        ❷
  ...
```

❶ Enable low-end mode.

❷ Set process affinity to this core (default is 0)

This mode implies following:

- All TRex threads will be assigned to single core
  (by default to 0, configurable via low_end_core)
- Lower memory allocation/requirement.
  If you have already started without low_end argument, reboot the Linux to free hugepages.
- Sleeps in scheduler instead of busy wait
  (less accurate, but saves power)
- Do not retry sending packets in case of queue full.
- "platform" section in config file is ignored.

We have tested with following VM:

- 1 core

- 475MB of RAM
- Two virtual NICs E1000

For example, stateless IMIX:
>start -f stl/imix.py -m 100kpps

```
Tx bps L2  |       289.67 Mbps |       289.67 Mbps |       579.34 Mbps
Tx bps L1  |       305.68 Mbps |       305.68 Mbps |       611.36 Mbps
Tx pps     |       100.07 Kpps |       100.07 Kpps |       200.14 Kpps
Line Util. |           30.57 % |           30.57 % |
```

"top" output:

```
%CPU COMMAND
 1.0 _t-rex-64
 0.0  `- eal-intr-thread
13.0  `- Trex DP core 1
 3.0  `- TRex RX
 0.0  `- Trex Publisher
 0.0  `- Trex Publisher
 0.0  `- Trex ZMQ sync
 0.0  `- Trex Watchdog
 0.0  `- Trex ZMQ sync
 0.0  `- Trex ZMQ sync
```

3.5 Linux interfaces

Another interesting low-end use case is using native Linux interfaces.
(Starting from version 2.37)

Pros:

- Any NIC type (supported by Linux) can be used, particularly, virtual interfaces without PCI address.
- Does not require hugepages.
- Can capture the traffic in Linux (Wireshark / tcpdump etc.).
- Startup is very fast (NICs are not initialized)
- Does not require special Kernel module (igb_uio.ko etc.)

Cons:

- Single TX/RX core, particularly the rate will be limited to ~1Mpps.
- Demands more CPU utilization.
- Slows down the Linux overall.

Platform config file example:

```
  - port_limit: 2
    version: 2
    interfaces: ['eth2', 'eth3']
...
```

Under the hood it's being replaced with the following (this full syntax can also be used, possibly containing another DPDK arguments):

```
  - port_limit: 2
    version: 2
    interfaces: ['--vdev=net_af_packet0,iface=eth2', '--vdev=net_af_packet1,iface=eth3']
...
```

3.6 memif interfaces

It wraps a share memory infra ring. There is a foursome of arguments to interconnect 2 memifs to communicate: `socket`, `id`, `role` and `secret`.

Socket, id and secret needs to be the same (for pair of connected memifs), role has to differ.

One of pair needs to have role `master` and second `slave`.

Socket is for exchange of protocol control messages. (by default /run/vpp/memif.sock in FD.io VPP) Id just has to match. it's unique for every pair of memifs. Secret is optional string (by default empty) wich has to match.

For example for yaml trex memif interface definition mentioned above:

trex_cfg.yaml

```
  - port_limit: 2
    version: 2
    interfaces: ["--vdev=net_memif0,socket=/run/vpp/memif.sock,role=slave,id=0",
                 "--vdev=net_memif1,socket=/run/vpp/memif.sock,role=slave,id=1"]
```

To create corresponding interfaces in VPP I used following CLI commands:

```
vpp# create interface memif id 0 master
vpp# set interface state memif0/0 up
vpp# create interface memif id 1 master
vpp# set interface state memif0/1 up
```

Then you can check the state by:

```
vpp# show memif
```

3.7 Emulation stacks

One can choose the method of how TRex answers to ping/ARP in interactive modes (STL/ASTF). (Starting from version 2.43)

Valid values:

- "legacy" - the default value, TRex uses it's own code to answer pings/ARPs.
- "linux_based" - ARPs/pings are redirected to Linux interfaces, that deal with them.
  The Linux interfaces reside in separate network namespace in order to have their own ARP tables and not interfere with the rest of system.
  It can be extended in future to:
  - Support other protocols that Linux natively supports
  - Many [thousands] emulated clients, while TRex port is behaving as switch connecting all that clients.

Platform config file example:

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

Table 3.2: Features of stacks

Stack:	**legacy**	**linux_based**
L2:	yes	yes
IPv4:	yes	yes
IPv6:	**no**	yes
VLAN:	yes	yes
STL:	fully	**in service**
ASTF:	fully	fully

Warning
linux_based stack requires service mode to process RX packets in STL mode

Examples of configuration:

Console:

L2, IPv4, VLAN - configured in same way with both stacks.

IPv6:

```
trex>ipv6 --help
usage: port [-h] -p PORT (--off | --auto | -s SRC_IPV6)

Configures IPv6 of a port

optional arguments:
  -h, --help            show this help message and exit
  -p PORT, --port PORT  source port for the action
  --off                 Disable IPv6 on port.
  --auto                Enable IPv6 on port with automatic address.
  -s SRC_IPV6, --src SRC_IPV6
                        Enable IPv6 on port with specific address.
```

API:

see IPv6 configuration

3.8 Dummy ports

Dummy ports are available to solve 2 issues:

- Odd number of interfaces. For example, TRex with only one interface.
- Performance degradation when adjacent interfaces belong to different NUMAs.

Configuration example:

```
    ...
    interfaces: ['07:00.0', 'dummy', 'dummy', '8a:00.0']
    ...
```

Incorrect configuration example:

```
    ...
    interfaces: ['dummy', 'dummy', 'dummy', '8a:00.0']
    ....
```

(Each pair of ports should have at least one non-dummy port)

Note

1. Supported in TRex as of v2.38+.
2. Client (Python API, trex-console) older than v2.38 will not work with dummy ports. Update your client to v2.38 or later.
3. Dummy ports are available for all TRex modes (stateless, statefull, and ASTF).

Consider the following if using dummy ports for a single interface TRex generator:

Configuration	Mode	Support Status	Comments
Single TRex Generator, Single Interface	Stateless	functional	unidirectional
	Stateful	semi-functional	will only send unidirectional side of the flows
	ASTF	depends on remote	requires bidirectional TCP stack, if remote supports TCP stack, will work, otherwise will not work
Two TRex Generators ,Single Interface	Stateless	functional	bidirectional
	Stateful	semi-functional	will "work", however flows will not be in sync
	ASTF	functional	full TCP stack

3.9 Configuring for running with router (or other L3 device) as DUT

You can follow this presentation for an example of how to configure the router as a DUT.

3.10 Running TRex, understanding output

After configuration is complete, use the following command to start basic TRex run for 10 seconds (it will use the default config file name /etc/trex_cfg.yaml):

```
[bash]>sudo ./t-rex-64 -f cap2/dns.yaml -c 4 -m 1 -d 10
```

3.10.1 TRex output

After running TRex successfully, the output will be similar to the following:

```
$ sudo ./t-rex-64 -f cap2/dns.yaml -d 10
Starting  TRex 2.09 please wait  ...
zmq publisher at: tcp://*:4500
 number of ports found : 4
```

```
  port : 0
  ------------
  link          :  link : Link Up - speed 10000 Mbps - full-duplex      ❶
  promiscuous   : 0
  port : 1
  ------------
  link          :  link : Link Up - speed 10000 Mbps - full-duplex
  promiscuous   : 0
  port : 2
  ------------
  link          :  link : Link Up - speed 10000 Mbps - full-duplex
  promiscuous   : 0
  port : 3
  ------------
  link          :  link : Link Up - speed 10000 Mbps - full-duplex
  promiscuous   : 0


 -Per port stats table
      ports |               0 |               1 |               2 |               3
 -------------------------------------------------------------------------------------
   opackets |            1003 |            1003 |            1002 |            1002
     obytes |           66213 |           66229 |           66132 |           66132
   ipackets |            1003 |            1003 |            1002 |            1002
     ibytes |           66225 |           66209 |           66132 |           66132
    ierrors |               0 |               0 |               0 |               0
    oerrors |               0 |               0 |               0 |               0
      Tx Bw |      217.09 Kbps |      217.14 Kbps |      216.83 Kbps |      216.83 Kbps

 -Global stats enabled
 Cpu Utilization : 0.0  % ❷  29.7 Gb/core ❸
 Platform_factor : 1.0
 Total-Tx        :     867.89 Kbps                                          ❹
 Total-Rx        :     867.86 Kbps                                          ❺
 Total-PPS       :       1.64 Kpps
 Total-CPS       :       0.50  cps

 Expected-PPS    :       2.00  pps    ❻
 Expected-CPS    :       1.00  cps    ❼
 Expected-BPS    :       1.36 Kbps    ❽

 Active-flows    :         0 ❾ Clients :      510   Socket-util  : 0.0000 %
 Open-flows      :         1 ❿ Servers :      254   Socket    :        1  Socket/Clients :   ↩
    0.0
 drop-rate       :       0.00  bps    ⓫
 current time    : 5.3 sec
 test duration   : 94.7 sec
```

❶ Link must be up for TRex to work.

❷ Average CPU utilization of transmitters threads. For best results it should be lower than 80%.

❸ Gb/sec generated per core of DP. Higher is better.

❹ Total Tx must be the same as Rx at the end of the run.

❺ Total Rx must be the same as Tx at the end of the run.

❻ Expected number of packets per second (calculated without latency packets).

❼ Expected number of connections per second (calculated without latency packets).

8 Expected number of bits per second (calculated without latency packets).

9 Number of TRex active "flows". Could be different than the number of router flows, due to aging issues. Usually the TRex number of active flows is much lower than that of the router because the router ages flows slower.

10 Total number of TRex flows opened since startup (including active ones, and ones already closed).

11 Drop rate.

3.10.2 latency stats

Though not present in the above output, measuring latency and jitter is possible! See the Measuring Jittery/Latency [jitter_latency] section for more details.

3.10.3 additional information about statistics in output

socket
Same as the active flows.

Socket/Clients
Average of active flows per client, calculated as active_flows/#clients.

Socket-util
Estimate of number of L4 ports (sockets) used per client IP. This is approximately (100*active_flows/#clients)/64K, calculated as (average active flows per client*100/64K). Utilization of more than 50% means that TRex is generating too many flows per single client, and that more clients must be added in the generator configuration.

Platform_factor
In some cases, users duplicate traffic using a splitter/switch. In this scenario, it is useful for all numbers displayed by TRex to be multiplied by this factor, so that TRex counters will match the DUT counters.

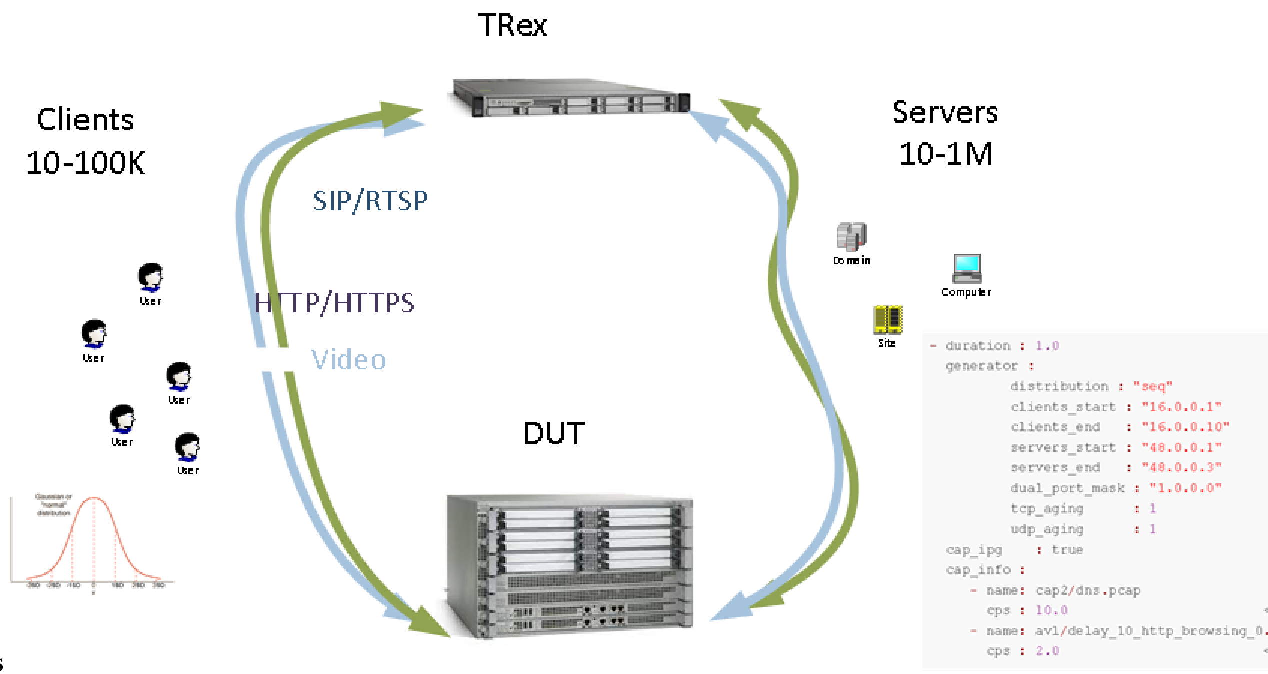
Warning
If you do not see Rx packets, review the MAC address configuration.

Chapter 4

Basic usage

4.1 DNS basic example

The following is a simple example helpful for understanding how TRex works. The example uses the TRex simulator. This simulator can be run on any Cisco Linux including on the TRex itself. TRex simulates clients and servers and generates traffic based on the pcap files provided.



```
- duration : 1.0
  generator :
          distribution : "seq"
          clients_start : "16.0.0.1"
          clients_end   : "16.0.0.10"
          servers_start : "48.0.0.1"
          servers_end   : "48.0.0.3"
          dual_port_mask : "1.0.0.0"
          tcp_aging      : 1
          udp_aging      : 1
  cap_ipg    : true
  cap_info :
     - name: cap2/dns.pcap
       cps : 10.0
     - name: avl/delay_10_http_browsing_0.
       cps : 2.0
```

Clients/Servers

The following is an example YAML-format traffic configuration file (cap2/dns_test.yaml), with explanatory notes.

```
[bash]>cat cap2/dns_test.yaml
- duration : 10.0
  generator :
          distribution : "seq"
          clients_start : "16.0.0.1"      ❶
          clients_end   : "16.0.0.255"
          servers_start : "48.0.0.1"      ❷
          servers_end   : "48.0.0.255"
          clients_per_gb : 201
          min_clients    : 101
          dual_port_mask : "1.0.0.0"
          tcp_aging      : 1
          udp_aging      : 1
  cap_info :
     - name: cap2/dns.pcap                ❸
       cps : 1.0                          ❹
       ipg : 10000                        ❺
       rtt : 10000                        ❻
       w   : 1
```

❶ Range of clients (IPv4 format).

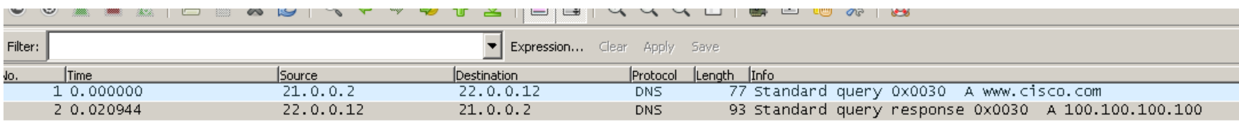
❷ Range of servers (IPv4 format).

❸ pcap file, which includes the DNS cap file that will be used as a template.

❹ Number of connections per second to generate. In the example, 1.0 means 1 connection per secod.

❺ Inter-packet gap (microseconds). 10,000 = 10 msec.

❻ Should be the same as ipg.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	21.0.0.2	22.0.0.12	DNS	77	Standard query 0x0030 A www.cisco.com
2	0.020944	22.0.0.12	21.0.0.2	DNS	93	Standard query response 0x0030 A 100.100.100.100

DNS template file

The DNS template file includes:

1. **One** flow
2. Two packets
3. First packet: from the initiator (client → server)
4. Second packet: response (server → client)

TRex replaces the client_ip, client_port, and server_ip. The server_port will be remain the same.

```
[bash]>./bp-sim-64-debug -f cap2/dns.yaml -o my.erf -v 3
 -- loading cap file cap2/dns.pcap
 id,name                  , tps, cps,f-pkts,f-bytes, duration,   Mb/sec,   MB/sec,   #❶
 00, cap2/dns.pcap        ,1.00,1.00,     2 ,     170 ,    0.02 ,   0.00 ,    0.00 ,
 00, sum                  ,1.00,1.00,     2 ,     170 ,    0.00 ,   0.00 ,    0.00 ,

 Generating erf file ...
pkt_id,time,fid,pkt_info,pkt,len,type,is_init,is_last,type,thread_id,src_ip,dest_ip, ↩
    src_port #❷
 1 ,0.010000,1,0x9055598,1,77,0,1,0,0,0,10000001,30000001,1024
 2 ,0.020000,1,0x9054760,2,93,0,0,1,0,0,10000001,30000001,1024
 3 ,2.010000,2,0x9055598,1,77,0,1,0,0,0,10000002,30000002,1024
 4 ,2.020000,2,0x9054760,2,93,0,0,1,0,0,10000002,30000002,1024
 5 ,3.010000,3,0x9055598,1,77,0,1,0,0,0,10000003,30000003,1024
 6 ,3.020000,3,0x9054760,2,93,0,0,1,0,0,10000003,30000003,1024
 7 ,4.010000,4,0x9055598,1,77,0,1,0,0,0,10000004,30000004,1024
 8 ,4.020000,4,0x9054760,2,93,0,0,1,0,0,10000004,30000004,1024
 9 ,5.010000,5,0x9055598,1,77,0,1,0,0,0,10000005,30000005,1024
 10 ,5.020000,5,0x9054760,2,93,0,0,1,0,0,10000005,30000005,1024
 11 ,6.010000,6,0x9055598,1,77,0,1,0,0,0,10000006,30000006,1024
 12 ,6.020000,6,0x9054760,2,93,0,0,1,0,0,10000006,30000006,1024
 13 ,7.010000,7,0x9055598,1,77,0,1,0,0,0,10000007,30000007,1024
 14 ,7.020000,7,0x9054760,2,93,0,0,1,0,0,10000007,30000007,1024
 15 ,8.010000,8,0x9055598,1,77,0,1,0,0,0,10000008,30000008,1024
 16 ,8.020000,8,0x9054760,2,93,0,0,1,0,0,10000008,30000008,1024
 17 ,9.010000,9,0x9055598,1,77,0,1,0,0,0,10000009,30000009,1024
 18 ,9.020000,9,0x9054760,2,93,0,0,1,0,0,10000009,30000009,1024
 19 ,10.010000,a,0x9055598,1,77,0,1,0,0,0,1000000a,3000000a,1024
 20 ,10.020000,a,0x9054760,2,93,0,0,1,0,0,1000000a,3000000a,1024

file stats
=================
 m_total_bytes                                :       1.66 Kbytes
 m_total_pkt                                  :      20.00  pkt
 m_total_open_flows                           :      10.00  flows
 m_total_pkt                                  : 20
 m_total_open_flows                           : 10
 m_total_close_flows                          : 10
 m_total_bytes                                : 1700
```

1. Global statistics on the templates given. cps=connection per second. tps is template per second. they might be different in case of plugins where one template includes more than one flow. For example RTP flow in SFR profile (avl/delay_10_rtp_160k_full.pcap)

2. Generator output.

```
[bash]>wireshark  my.erf
```

gives

TRex generated output file images/dns_trex_run.png

As the output file shows. . .

- TRex generates a new flow every 1 sec.
- Client IP values are taken from client IP pool .
- Servers IP values are taken from server IP pool .
- IPG (inter packet gap) values are taken from the configuration file (10 msec).

Note
In basic usage, TRex does not wait for an initiator packet to be received. The response packet will be triggered based only on timeout (IPG in this example).
In advanced scenarios (for example, NAT), the first packet of the flow will be processed by TRex and initiate the response packet only when a packet is received.
Consequently, it is necessary to **process** the template pcap file offline and ensure that there is enough round-trip delay (RTT) between client and server packets. Take a look at: astf-sim for fixing pcaps
Another approach is to change the `yaml ipg` field to a high enough value (bigger than 10msec).

Converting the simulator text results in a table similar to the following:

Table 4.1: DNS example formatted results

pkt	time sec	fid	flow-pkt-id	client_ip	client_port	server_ip	direction
1	0.010000	1	1	16.0.0.1	1024	48.0.0.1	→
2	0.020000	1	2	16.0.0.1	1024	48.0.0.1	←
3	2.010000	2	1	16.0.0.2	1024	48.0.0.2	→
4	2.020000	2	2	16.0.0.2	1024	48.0.0.2	←

Table 4.1: (continued)

pkt	time sec	fid	flow-pkt-id	client_ip	client_port	server_ip	direction
5	3.010000	3	1	16.0.0.3	1024	48.0.0.3	→
6	3.020000	3	2	16.0.0.3	1024	48.0.0.3	←
7	4.010000	4	1	16.0.0.4	1024	48.0.0.4	→
8	4.020000	4	2	16.0.0.4	1024	48.0.0.4	←
9	5.010000	5	1	16.0.0.5	1024	48.0.0.5	→
10	5.020000	5	2	16.0.0.5	1024	48.0.0.5	←
11	6.010000	6	1	16.0.0.6	1024	48.0.0.6	→
12	6.020000	6	2	16.0.0.6	1024	48.0.0.6	←
13	7.010000	7	1	16.0.0.7	1024	48.0.0.7	→
14	7.020000	7	2	16.0.0.7	1024	48.0.0.7	←
15	8.010000	8	1	16.0.0.8	1024	48.0.0.8	→
16	8.020000	8	2	16.0.0.8	1024	48.0.0.8	←
17	9.010000	9	1	16.0.0.9	1024	48.0.0.9	→
18	9.020000	9	2	16.0.0.9	1024	48.0.0.9	←
19	10.010000	a	1	16.0.0.10	1024	48.0.0.10	→
20	10.020000	a	2	16.0.0.10	1024	48.0.0.10	←

where: fid:: Flow ID - different IDs for each flow.

low-pkt-id
Packet ID within the flow. Numbering begins with 1.

client_ip
Client IP address.

client_port
Client IP port.

server_ip
Server IP address.

direction
Direction. "→" is client-to-server; "←" is server-to-client.

The following enlarges the CPS and reduces the duration.

```
[bash]>more cap2/dns_test.yaml
- duration : 1.0                                  ❶
  generator :
          distribution : "seq"
          clients_start : "16.0.0.1"
          clients_end   : "16.0.0.255"
          servers_start : "48.0.0.1"
          servers_end   : "48.0.0.255"
          clients_per_gb : 201
          min_clients    : 101
          dual_port_mask : "1.0.0.0"
          tcp_aging      : 1
          udp_aging      : 1
  mac        : [0x00,0x00,0x00,0x01,0x00,0x00]
  cap_info :
     - name: cap2/dns.pcap
       cps : 10.0                                  ❷
```

```
        ipg : 50000                          ❸
        rtt : 50000
        w   : 1
```

❶ Duration is 1 second.

❷ CPS is 10.0.

❸ IPG is 50 msec.

Running this produces the following output:

```
[bash]>./bp-sim-64-debug -f cap2/dns_test.yaml -o my.erf -v 3
```

Table 4.2: Formated results

pkt	time sec	template	fid	flow-pkt-id	client_ip	client_port	server_ip	desc
1	0.010000	0	1	1	16.0.0.1	1024	48.0.0.1	→
2	0.060000	0	1	2	16.0.0.1	1024	48.0.0.1	←
3	0.210000	0	2	1	16.0.0.2	1024	48.0.0.2	→
4	0.260000	0	2	2	16.0.0.2	1024	48.0.0.2	←
5	0.310000	0	3	1	16.0.0.3	1024	48.0.0.3	→
6	0.360000	0	3	2	16.0.0.3	1024	48.0.0.3	←
7	0.410000	0	4	1	16.0.0.4	1024	48.0.0.4	→
8	0.460000	0	4	2	16.0.0.4	1024	48.0.0.4	←
9	0.510000	0	5	1	16.0.0.5	1024	48.0.0.5	→
10	0.560000	0	5	2	16.0.0.5	1024	48.0.0.5	←
11	0.610000	0	6	1	16.0.0.6	1024	48.0.0.6	→
12	0.660000	0	6	2	16.0.0.6	1024	48.0.0.6	←
13	0.710000	0	7	1	16.0.0.7	1024	48.0.0.7	→
14	0.760000	0	7	2	16.0.0.7	1024	48.0.0.7	←
15	0.810000	0	8	1	16.0.0.8	1024	48.0.0.8	→
16	0.860000	0	8	2	16.0.0.8	1024	48.0.0.8	←
17	0.910000	0	9	1	16.0.0.9	1024	48.0.0.9	→
18	0.960000	0	9	2	16.0.0.9	1024	48.0.0.9	←
19	1.010000	0	a	1	16.0.0.10	1024	48.0.0.10	→
20	1.060000	0	a	2	16.0.0.10	1024	48.0.0.10	←

Use the following to display the output as a chart, with: x axis: time (seconds) y axis: flow ID The output indicates that there are 10 flows in 1 second, as expected, and the IPG is 50 msec

Note
Note the gap in the second flow generation. This is an expected schedular artifact and does not have an effect.

4.2 DNS, take flow IPG from pcap file

In the following example the IPG is taken from the IPG itself.

```
- duration : 1.0
  generator :
          distribution : "seq"
          clients_start : "16.0.0.1"
          clients_end   : "16.0.0.255"
          servers_start : "48.0.0.1"
          servers_end   : "48.0.0.255"
          clients_per_gb : 201
          min_clients    : 101
          dual_port_mask : "1.0.0.0"
          tcp_aging      : 0
          udp_aging      : 0
  mac         : [0x00,0x00,0x00,0x01,0x00,0x00]
  cap_ipg     : true            ❶
  #cap_ipg_min     : 30
  #cap_override_ipg     : 200
  cap_info :
     - name: cap2/dns.pcap
       cps : 10.0
       ipg : 10000
       rtt : 10000
       w   : 1
```

❶ IPG is taken from pcap.

Table 4.3: dns ipg from pcap file

pkt	time sec	template	fid	flow-pkt-id	client_ip	client_port	server_ip	desc
1	0.010000	0	1	1	16.0.0.1	1024	48.0.0.1	→
2	0.030944	0	1	2	16.0.0.1	1024	48.0.0.1	←
3	0.210000	0	2	1	16.0.0.2	1024	48.0.0.2	→
4	0.230944	0	2	2	16.0.0.2	1024	48.0.0.2	←
5	0.310000	0	3	1	16.0.0.3	1024	48.0.0.3	→
6	0.330944	0	3	2	16.0.0.3	1024	48.0.0.3	←
7	0.410000	0	4	1	16.0.0.4	1024	48.0.0.4	→
8	0.430944	0	4	2	16.0.0.4	1024	48.0.0.4	←
9	0.510000	0	5	1	16.0.0.5	1024	48.0.0.5	→
10	0.530944	0	5	2	16.0.0.5	1024	48.0.0.5	←
11	0.610000	0	6	1	16.0.0.6	1024	48.0.0.6	→
12	0.630944	0	6	2	16.0.0.6	1024	48.0.0.6	←
13	0.710000	0	7	1	16.0.0.7	1024	48.0.0.7	→
14	0.730944	0	7	2	16.0.0.7	1024	48.0.0.7	←
15	0.810000	0	8	1	16.0.0.8	1024	48.0.0.8	→
16	0.830944	0	8	2	16.0.0.8	1024	48.0.0.8	←
17	0.910000	0	9	1	16.0.0.9	1024	48.0.0.9	→
18	0.930944	0	9	2	16.0.0.9	1024	48.0.0.9	←
19	1.010000	0	a	1	16.0.0.10	1024	48.0.0.10	→
20	1.030944	0	a	2	16.0.0.10	1024	48.0.0.10	←

In this example, the IPG was taken from the pcap file, which is closer to 20 msec and not 50 msec (taken from the configuration file).

```
  #cap_ipg_min     : 30            ❶
```

```
#cap_override_ipg    : 200      ❷
```

❶ Sets the minimum IPG (microseconds) which should be overriden : (if (pkt_ipg<cap_ipg_min) { pkt_ipg = cap_override_ipg })

❷ Value to override (microseconds).

4.3 DNS, Set one server ip

In this example the server IP is taken from the template.

```
- duration : 10.0
  generator :
          distribution : "seq"
          clients_start : "16.0.0.1"
          clients_end   : "16.0.1.255"
          servers_start : "48.0.0.1"
          servers_end   : "48.0.0.255"
          clients_per_gb : 201
          min_clients    : 101
          dual_port_mask : "1.0.0.0"
          tcp_aging      : 1
          udp_aging      : 1
  mac        : [0x00,0x00,0x00,0x01,0x00,0x00]
  cap_ipg    : true
  #cap_ipg_min    : 30
  #cap_override_ipg    : 200
  cap_info :
     - name: cap2/dns.pcap
       cps : 1.0
       ipg : 10000
       rtt : 10000
       server_addr : "48.0.0.7"     ❶
       one_app_server : true        ❷
       w   : 1
```

❶ All flows of this template will use the same server.

❷ Must be set to "true".

Table 4.4: dns ipg from pcap file

pkt	time sec	fid	flow-pkt-id	client_ip	client_port	server_ip	desc
1	0.010000	1	1	16.0.0.1	1024	48.0.0.7	→
2	0.030944	1	2	16.0.0.1	1024	48.0.0.7	←
3	2.010000	2	1	16.0.0.2	1024	48.0.0.7	→
4	2.030944	2	2	16.0.0.2	1024	48.0.0.7	←
5	3.010000	3	1	16.0.0.3	1024	48.0.0.7	→
6	3.030944	3	2	16.0.0.3	1024	48.0.0.7	←
7	4.010000	4	1	16.0.0.4	1024	48.0.0.7	→
8	4.030944	4	2	16.0.0.4	1024	48.0.0.7	←
9	5.010000	5	1	16.0.0.5	1024	48.0.0.7	→
10	5.030944	5	2	16.0.0.5	1024	48.0.0.7	←
11	6.010000	6	1	16.0.0.6	1024	48.0.0.7	→
12	6.030944	6	2	16.0.0.6	1024	48.0.0.7	←

Table 4.4: (continued)

pkt	time sec	fid	flow-pkt-id	client_ip	client_port	server_ip	desc
13	7.010000	7	1	16.0.0.7	1024	48.0.0.7	→
14	7.030944	7	2	16.0.0.7	1024	48.0.0.7	←
15	8.010000	8	1	16.0.0.8	1024	48.0.0.7	→
16	8.030944	8	2	16.0.0.8	1024	48.0.0.7	←
17	9.010000	9	1	16.0.0.9	1024	48.0.0.7	→
18	9.030944	9	2	16.0.0.9	1024	48.0.0.7	←
19	10.010000	a	1	16.0.0.10	1024	48.0.0.7	→
20	10.030944	a	2	16.0.0.10	1024	48.0.0.7	←

4.4 DNS, Reduce the number of clients

```
- duration : 10.0
  generator :
          distribution : "seq"
          clients_start : "16.0.0.1"     ❶
          clients_end   : "16.0.0.1"
          servers_start : "48.0.0.1"
          servers_end   : "48.0.0.3"
          clients_per_gb : 201
          min_clients     : 101
          dual_port_mask : "1.0.0.0"
          tcp_aging      : 1
          udp_aging      : 1
  mac          : [0x00,0x00,0x00,0x01,0x00,0x00]
  cap_ipg     : true
  #cap_ipg_min     : 30
  #cap_override_ipg    : 200
  cap_info :
     - name: cap2/dns.pcap
       cps : 1.0
       ipg : 10000
       rtt : 10000
       w   : 1
```

❶ Only one client.

Table 4.5: dns ipg from pcap file

pkt	time sec	fid	flow-pkt-id	client_ip	client_port	server_ip	desc
1	0.010000	1	1	16.0.0.1	1024	48.0.0.1	→
2	0.030944	1	2	16.0.0.1	1024	48.0.0.1	←
3	2.010000	2	1	16.0.0.1	1025	48.0.0.2	→
4	2.030944	2	2	16.0.0.1	1025	48.0.0.2	←
5	3.010000	3	1	16.0.0.1	1026	48.0.0.3	→
6	3.030944	3	2	16.0.0.1	1026	48.0.0.3	←
7	4.010000	4	1	16.0.0.1	1027	48.0.0.4	→
8	4.030944	4	2	16.0.0.1	1027	48.0.0.4	←
9	5.010000	5	1	16.0.0.1	1028	48.0.0.5	→

Table 4.5: (continued)

pkt	time sec	fid	flow-pkt-id	client_ip	client_port	server_ip	desc
10	5.030944	5	2	16.0.0.1	1028	48.0.0.5	←
11	6.010000	6	1	16.0.0.1	1029	48.0.0.6	→
12	6.030944	6	2	16.0.0.1	1029	48.0.0.6	←
13	7.010000	7	1	16.0.0.1	1030	48.0.0.7	→
14	7.030944	7	2	16.0.0.1	1030	48.0.0.7	←
15	8.010000	8	1	16.0.0.1	1031	48.0.0.8	→
16	8.030944	8	2	16.0.0.1	1031	48.0.0.8	←
17	9.010000	9	1	16.0.0.1	1032	48.0.0.9	→
18	9.030944	9	2	16.0.0.1	1032	48.0.0.9	←
19	10.010000	a	1	16.0.0.1	1033	48.0.0.10	→
20	10.030944	a	2	16.0.0.1	1033	48.0.0.10	←

In this case there is only one client so only ports are used to distinc the flows you need to be sure that you have enogth free sockets when running TRex in high rates

```
Active-flows    :          0  Clients :       1 ❶  Socket-util : 0.0000 %     ❷
Open-flows      :          1  Servers :     254   Socket :          1 Socket/Clients :  0.0
drop-rate       :        0.00  bps
```

❶ Number of clients

❷ sockets utilization (should be lower than 20%, enlarge the number of clients in case of an issue).

4.5 DNS, W=1

`w` is a tunable to the IP clients/servers generator. w=1 is the default behavior. Setting `w=2` configures a burst of two allocations from the same client. See the following example.

```
- duration : 10.0
  generator :
          distribution : "seq"
          clients_start : "16.0.0.1"
          clients_end   : "16.0.0.10"
          servers_start : "48.0.0.1"
          servers_end   : "48.0.0.3"
          clients_per_gb : 201
          min_clients    : 101
          dual_port_mask : "1.0.0.0"
          tcp_aging      : 1
          udp_aging      : 1
  mac        : [0x00,0x00,0x00,0x01,0x00,0x00]
  cap_ipg    : true
  #cap_ipg_min    : 30
  #cap_override_ipg    : 200
  cap_info :
     - name: cap2/dns.pcap
       cps : 1.0
       ipg : 10000
       rtt : 10000
       w   : 2
```

Table 4.6: DNS ipg from pcap file

pkt	time sec	fid	flow-pkt-id	client_ip	client_port	server_ip	desc
1	0.010000	1	1	16.0.0.1	1024	48.0.0.1	→
2	0.030944	1	2	16.0.0.1	1024	48.0.0.1	←
3	2.010000	2	1	16.0.0.1	1025	48.0.0.1	→
4	2.030944	2	2	16.0.0.1	1025	48.0.0.1	←
5	3.010000	3	1	16.0.0.2	1024	48.0.0.2	→
6	3.030944	3	2	16.0.0.2	1024	48.0.0.2	←
7	4.010000	4	1	16.0.0.2	1025	48.0.0.2	→
8	4.030944	4	2	16.0.0.2	1025	48.0.0.2	←
9	5.010000	5	1	16.0.0.3	1024	48.0.0.3	→
10	5.030944	5	2	16.0.0.3	1024	48.0.0.3	←
11	6.010000	6	1	16.0.0.3	1025	48.0.0.3	→
12	6.030944	6	2	16.0.0.3	1025	48.0.0.3	←
13	7.010000	7	1	16.0.0.4	1024	48.0.0.4	→
14	7.030944	7	2	16.0.0.4	1024	48.0.0.4	←
15	8.010000	8	1	16.0.0.4	1025	48.0.0.4	→
16	8.030944	8	2	16.0.0.4	1025	48.0.0.4	←
17	9.010000	9	1	16.0.0.5	1024	48.0.0.5	→
18	9.030944	9	2	16.0.0.5	1024	48.0.0.5	←
19	10.010000	a	1	16.0.0.5	1025	48.0.0.5	→
20	10.030944	a	2	16.0.0.5	1025	48.0.0.5	←

4.6 Mixing HTTP and DNS templates

The following example combines elements of HTTP and DNS templates:

```
- duration : 1.0
  generator :
          distribution : "seq"
          clients_start : "16.0.0.1"
          clients_end   : "16.0.0.10"
          servers_start : "48.0.0.1"
          servers_end   : "48.0.0.3"
          clients_per_gb : 201
          min_clients    : 101
          dual_port_mask : "1.0.0.0"
          tcp_aging      : 1
          udp_aging      : 1
  mac        : [0x00,0x00,0x00,0x01,0x00,0x00]
  cap_ipg    : true
  cap_info :
     - name: cap2/dns.pcap
       cps : 10.0                                    ❶
       ipg : 10000
       rtt : 10000
       w   : 1
     - name: avl/delay_10_http_browsing_0.pcap
       cps : 2.0                                     ❷
       ipg : 10000
       rtt : 10000
       w   : 1
```

1, 2 Same CPS for both templates.

This creates the following output:

Table 4.7: DNS ipg from pcap file

pkt	time sec	template	fid	flow-pkt-id	client_ip	client_port	server_ip	desc
1	0.010000	0	1	1	16.0.0.1	1024	48.0.0.1	→
2	0.030944	0	1	2	16.0.0.1	1024	48.0.0.1	←
3	0.093333	1	2	1	16.0.0.2	1024	48.0.0.2	→
4	0.104362	1	2	2	16.0.0.2	1024	48.0.0.2	←
5	0.115385	1	2	3	16.0.0.2	1024	48.0.0.2	→
6	0.115394	1	2	4	16.0.0.2	1024	48.0.0.2	→
7	0.126471	1	2	5	16.0.0.2	1024	48.0.0.2	←
8	0.126484	1	2	6	16.0.0.2	1024	48.0.0.2	←
9	0.137530	1	2	7	16.0.0.2	1024	48.0.0.2	→
10	0.148609	1	2	8	16.0.0.2	1024	48.0.0.2	←
11	0.148621	1	2	9	16.0.0.2	1024	48.0.0.2	←
12	0.148635	1	2	10	16.0.0.2	1024	48.0.0.2	←
13	0.159663	1	2	11	16.0.0.2	1024	48.0.0.2	→
14	0.170750	1	2	12	16.0.0.2	1024	48.0.0.2	←
15	0.170762	1	2	13	16.0.0.2	1024	48.0.0.2	←
16	0.170774	1	2	14	16.0.0.2	1024	48.0.0.2	←
17	0.176667	0	3	1	16.0.0.3	1024	48.0.0.3	→
18	0.181805	1	2	15	16.0.0.2	1024	48.0.0.2	→
19	0.181815	1	2	16	16.0.0.2	1024	48.0.0.2	→
20	0.192889	1	2	17	16.0.0.2	1024	48.0.0.2	←
21	0.192902	1	2	18	16.0.0.2	1024	48.0.0.2	←

Template_id
0: DNS template 1: HTTP template

The output above illustrates two HTTP flows and ten DNS flows in 1 second, as expected.

4.7 SFR traffic YAML

SFR traffic includes a combination of traffic templates. This traffic mix in the example below was defined by SFR France. This SFR traffic profile is used as our traffic profile for our ASR1k/ISR-G2 benchmark. It is also possible to use EMIX instead of IMIX traffic.

The traffic was recorded from a Spirent C100 with a Pagent that introduce 10msec delay from client and server side.

```
- duration : 0.1
  generator :
          distribution : "seq"
          clients_start : "16.0.0.1"
          clients_end   : "16.0.1.255"
          servers_start : "48.0.0.1"
          servers_end   : "48.0.20.255"
          clients_per_gb : 201
          min_clients    : 101
          dual_port_mask : "1.0.0.0"
```

```
        tcp_aging      : 0
        udp_aging      : 0
  mac        : [0x0,0x0,0x0,0x1,0x0,0x00]
  cap_ipg    : true
  cap_info :
     - name: avl/delay_10_http_get_0.pcap
       cps : 404.52
       ipg : 10000
       rtt : 10000
       w   : 1
     - name: avl/delay_10_http_post_0.pcap
       cps : 404.52
       ipg : 10000
       rtt : 10000
       w   : 1
     - name: avl/delay_10_https_0.pcap
       cps : 130.8745
       ipg : 10000
       rtt : 10000
       w   : 1
     - name: avl/delay_10_http_browsing_0.pcap
       cps : 709.89
       ipg : 10000
       rtt : 10000
       w   : 1
     - name: avl/delay_10_exchange_0.pcap
       cps : 253.81
       ipg : 10000
       rtt : 10000
       w   : 1
     - name: avl/delay_10_mail_pop_0.pcap
       cps : 4.759
       ipg : 10000
       rtt : 10000
       w   : 1
     - name: avl/delay_10_mail_pop_1.pcap
       cps : 4.759
       ipg : 10000
       rtt : 10000
       w   : 1
     - name: avl/delay_10_mail_pop_2.pcap
       cps : 4.759
       ipg : 10000
       rtt : 10000
       w   : 1
     - name: avl/delay_10_oracle_0.pcap
       cps : 79.3178
       ipg : 10000
       rtt : 10000
       w   : 1
     - name: avl/delay_10_rtp_160k_full.pcap
       cps : 2.776
       ipg : 10000
       rtt : 10000
       w   : 1
       one_app_server : false
       plugin_id : 1          ❶
     - name: avl/delay_10_rtp_250k_full.pcap
       cps : 1.982
       ipg : 10000
       rtt : 10000
       w   : 1
```

```
        one_app_server : false
        plugin_id : 1
      - name: avl/delay_10_smtp_0.pcap
        cps : 7.3369
        ipg : 10000
        rtt : 10000
        w   : 1
      - name: avl/delay_10_smtp_1.pcap
        cps : 7.3369
        ipg : 10000
        rtt : 10000
        w   : 1
      - name: avl/delay_10_smtp_2.pcap
        cps : 7.3369
        ipg : 10000
        rtt : 10000
        w   : 1
      - name: avl/delay_10_video_call_0.pcap
        cps : 11.8976
        ipg : 10000
        rtt : 10000
        w   : 1
        one_app_server : false
      - name: avl/delay_10_sip_video_call_full.pcap
        cps : 29.347
        ipg : 10000
        rtt : 10000
        w   : 1
        plugin_id : 2    ❷
        one_app_server : false
      - name: avl/delay_10_citrix_0.pcap
        cps : 43.6248
        ipg : 10000
        rtt : 10000
        w   : 1
      - name: avl/delay_10_dns_0.pcap
        cps : 1975.015
        ipg : 10000
        rtt : 10000
        w   : 1
```

❷ Plugin for SIP protocol, used to replace the IP/port in the control flow base on the data-flow.

❶ Plugin for RTSP protocol used to replace the IP/port in the control flow base on the data-flow.

4.8 Running examples

TRex commands typically include the following main arguments, but only `-f` is required.

```
[bash]>sudo ./t-rex-64 -f <traffic_yaml> -m <multiplier>  -d <duration>  -l <latency test  ↩
    rate>  -c <cores>
```

Full command line reference can be found here [cml-line]

4.8.1 TRex command line examples

Simple HTTP 1Gb/sec for 100 sec

```
[bash]>sudo ./t-rex-64 -f cap2/simple_http.yaml -c 4 -m 100 -d 100
```

Simple HTTP 1Gb/sec with latency for 100 sec

```
[bash]>sudo ./t-rex-64 -f cap2/simple_http.yaml -c 4 -m 100 -d 100 -l 1000
```

SFR 35Gb/sec traffic

```
[bash]>sudo ./t-rex-64 -f avl/sfr_delay_10_1g.yaml -c 4 -m 35 -d 100 -p
```

SFR 20Gb/sec traffic with latency

```
[bash]>sudo ./t-rex-64 -f avl/sfr_delay_10_1g.yaml -c 4 -m 20 -d 100 -l 1000
```

SFR ipv6 20Gb/sec traffic with latency

```
[bash]>sudo ./t-rex-64 -f avl/sfr_delay_10_1g_no_bundeling.yaml -c 4 -m 20 -d 100 -l 1000  ↩
    --ipv6
```

Simple HTTP 1Gb/sec with NAT translation support

```
[bash]>sudo ./t-rex-64 -f cap2/simple_http.yaml -c 4 -m 100 -d 100 -l 1000 --learn-mode 1
```

IMIX 1G/sec ,1600 flows

```
[bash]>sudo ./t-rex-64 -f cap2/imix_fast_1g.yaml -c 4 -m 1 -d 100 -l 1000
```

IMIX 1Gb/sec,100K flows

```
[bash]>sudo ./t-rex-64 -f cap2/imix_fast_1g_100k.yaml -c 4 -m 1 -d 100 -l 1000
```

64bytes ~1Gb/sec,1600 flows

```
[bash]>sudo ./t-rex-64 -f cap2/imix_64.yaml -c 4 -m 1 -d 100 -l 1000
```

4.9 Traffic profiles provided with the TRex package

name	description
cap2/dns.yaml	simple dns pcap file
cap2/http_simple.yaml	simple http cap file
avl/sfr_delay_10_1g_no_bundeling.yaml	sfr traffic profile capture from Avalanche - Spirent without bundeling support with RTT=10msec (a delay machine), this can be used with --ipv6 and --learn-mode
avl/sfr_delay_10_1g.yaml	head-end sfr traffic profile capture from Avalanche - Spirent with bundeling support with RTT=10msec (a delay machine), it is normalized to 1Gb/sec for m=1
avl/sfr_branch_profile_delay_10.yaml	branch sfr profile capture from Avalanche - Spirent with bundeling support with RTT=10msec it, is normalized to 1Gb/sec for m=1
cap2/imix_fast_1g.yaml	imix profile with 1600 flows normalized to 1Gb/sec.
cap2/imix_fast_1g_100k_flows.yaml	imix profile with 100k flows normalized to 1Gb/sec.
cap2/imix_64.yaml	64byte UDP packets profile

4.10 Mimicking stateless traffic under stateful mode

Note
TRex supports also true stateless traffic generation. If you are looking for stateless traffic, please visit the following link: TRex Stateless Support [?]

With this feature you can "repeat" flows and create stateless, **IXIA** like streams. After injecting the number of flows defined by `limit`, TRex repeats the same flows. If all templates have `limit` the CPS will be zero after some time as there are no new flows after the first iteration.

IMIX support:
Example:

```
[bash]>sudo ./t-rex-64 -f cap2/imix_64.yaml  -d 1000 -m 40000  -c 4 -p
```

Warning
The **-p** is used here to send the client side packets from both interfaces. (Normally it is sent from client ports only.) With this option, the port is selected by the client IP. All the packets of a flow are sent from the same interface. This may create an issue with routing, as the client's IP will be sent from the server interface. PBR router configuration solves this issue but cannot be used in all cases. So use this `-p` option carefully.

imix_64.yaml

```
  cap_info :
     - name: cap2/udp_64B.pcap
       cps    : 1000.0
       ipg    : 10000
       rtt    : 10000
       w      : 1
       limit  : 1000                          (1)
```

(1) Repeats the flows in a loop, generating 1000 flows from this type. In this example, udp_64B includes only one packet.

The cap file "cap2/udp_64B.pcap" includes only one packet of 64B. This configuration file creates 1000 flows that will be repeated as follows: f1 , f2 , f3 . . . . f1000 , f1 , f2 . . . where the PPS == CPS for -m=1. In this case it will have PPS=1000 in sec for -m==1. It is possible to mix stateless templates and stateful templates.

Imix YAML cap2/imix_fast_1g.yaml example

```
- duration : 3
  generator :
          distribution : "seq"
          clients_start : "16.0.0.1"
          clients_end   : "16.0.0.255"
          servers_start : "48.0.0.1"
          servers_end   : "48.0.255.255"
          clients_per_gb : 201
          min_clients    : 101
          dual_port_mask : "1.0.0.0"
          tcp_aging      : 0
          udp_aging      : 0
  mac         : [0x0,0x0,0x0,0x1,0x0,0x00]
  cap_info :
     - name: cap2/udp_64B.pcap
```

```
       cps   : 90615
       ipg   : 10000
       rtt   : 10000
       w     : 1
       limit : 199
     - name: cap2/udp_576B.pcap
       cps   : 64725
       ipg   : 10000
       rtt   : 10000
       w     : 1
       limit : 199
     - name: cap2/udp_1500B.pcap
       cps   : 12945
       ipg   : 10000
       rtt   : 10000
       w     : 1
       limit : 199
     - name: cap2/udp_64B.pcap
       cps   : 90615
       ipg   : 10000
       rtt   : 10000
       w     : 1
       limit : 199
     - name: cap2/udp_576B.pcap
       cps   : 64725
       ipg   : 10000
       rtt   : 10000
       w     : 1
       limit : 199
     - name: cap2/udp_1500B.pcap
       cps   : 12945
       ipg   : 10000
       rtt   : 10000
       w     : 1
       limit : 199
```

The templates are duplicated here to better utilize DRAM and to get better performance.

Imix YAML cap2/imix_fast_1g_100k_flows.yaml example

```
- duration : 3
  generator :
          distribution : "seq"
          clients_start : "16.0.0.1"
          clients_end   : "16.0.0.255"
          servers_start : "48.0.0.1"
          servers_end   : "48.0.255.255"
          clients_per_gb : 201
          min_clients    : 101
          dual_port_mask : "1.0.0.0"
          tcp_aging      : 0
          udp_aging      : 0
  mac        : [0x0,0x0,0x0,0x1,0x0,0x00]
  cap_info :
     - name: cap2/udp_64B.pcap
       cps   : 90615
       ipg   : 10000
       rtt   : 10000
       w     : 1
       limit : 16666
     - name: cap2/udp_576B.pcap
       cps   : 64725
       ipg   : 10000
       rtt   : 10000
       w     : 1
       limit : 16666
     - name: cap2/udp_1500B.pcap
       cps   : 12945
       ipg   : 10000
       rtt   : 10000
       w     : 1
       limit : 16667
     - name: cap2/udp_64B.pcap
       cps   : 90615
       ipg   : 10000
       rtt   : 10000
       w     : 1
       limit : 16667
     - name: cap2/udp_576B.pcap
       cps   : 64725
       ipg   : 10000
       rtt   : 10000
       w     : 1
       limit : 16667
     - name: cap2/udp_1500B.pcap
       cps   : 12945
       ipg   : 10000
       rtt   : 10000
       w     : 1
       limit : 16667
```

The following example of a simple simulation includes 3 flows, with CPS=10.

```
$more cap2/imix_example.yaml
#
# Simple IMIX test (7x64B, 5x576B, 1x1500B)
#
- duration : 3
  generator :
          distribution : "seq"
          clients_start : "16.0.0.1"
          clients_end   : "16.0.0.255"
          servers_start : "48.0.0.1"
          servers_end   : "48.0.255.255"
          clients_per_gb : 201
          min_clients    : 101
          dual_port_mask : "1.0.0.0"
          tcp_aging      : 0
          udp_aging      : 0
  mac        : [0x0,0x0,0x0,0x1,0x0,0x00]
  cap_info :
     - name: cap2/udp_64B.pcap
       cps   : 10.0
       ipg   : 10000
       rtt   : 10000
       w     : 1
       limit : 3                           (1)
```

(1) Number of flows: 3

```
[bash]>./bp-sim-64-debug -f cap2/imix_example.yaml  -o my.erf -v 3 > a.txt
```

Table 4.8: IMIX example limit=3

pkt	time sec	template	fid	flow-pkt-id	client_ip	client_port	server_ip	desc
1	0.010000	0	1	1	16.0.0.1	1024	48.0.0.1	→
2	0.210000	0	2	0	16.0.0.2	1024	48.0.0.2	→
3	0.310000	0	3	0	16.0.0.3	1024	48.0.0.3	→
4	0.310000	0	1	0	16.0.0.1	1024	48.0.0.1	→
5	0.510000	0	2	0	16.0.0.2	1024	48.0.0.2	→
6	0.610000	0	3	0	16.0.0.3	1024	48.0.0.3	→
7	0.610000	0	1	0	16.0.0.1	1024	48.0.0.1	→
8	0.810000	0	2	0	16.0.0.2	1024	48.0.0.2	→
9	0.910000	0	1	0	16.0.0.1	1024	48.0.0.1	→
10	0.910000	0	3	0	16.0.0.3	1024	48.0.0.3	→
11	1.110000	0	2	0	16.0.0.2	1024	48.0.0.2	→
12	1.210000	0	3	0	16.0.0.3	1024	48.0.0.3	→
13	1.210000	0	1	0	16.0.0.1	1024	48.0.0.1	→
14	1.410000	0	2	0	16.0.0.2	1024	48.0.0.2	→
15	1.510000	0	1	0	16.0.0.1	1024	48.0.0.1	→
16	1.510000	0	3	0	16.0.0.3	1024	48.0.0.3	→
17	1.710000	0	2	0	16.0.0.2	1024	48.0.0.2	→
18	1.810000	0	3	0	16.0.0.3	1024	48.0.0.3	→
19	1.810000	0	1	0	16.0.0.1	1024	48.0.0.1	→
20	2.010000	0	2	0	16.0.0.2	1024	48.0.0.2	→
21	2.110000	0	1	0	16.0.0.1	1024	48.0.0.1	→
22	2.110000	0	3	0	16.0.0.3	1024	48.0.0.3	→
23	2.310000	0	2	0	16.0.0.2	1024	48.0.0.2	→
24	2.410000	0	3	0	16.0.0.3	1024	48.0.0.3	→
25	2.410000	0	1	0	16.0.0.1	1024	48.0.0.1	→
26	2.610000	0	2	0	16.0.0.2	1024	48.0.0.2	→
27	2.710000	0	1	0	16.0.0.1	1024	48.0.0.1	→
28	2.710000	0	3	0	16.0.0.3	1024	48.0.0.3	→
29	2.910000	0	2	0	16.0.0.2	1024	48.0.0.2	→
30	3.010000	0	3	0	16.0.0.3	1024	48.0.0.3	→
31	3.010000	0	1	0	16.0.0.1	1024	48.0.0.1	→

- Average CPS: 10 packets per second (30 packets in 3 sec).
- Total of 3 flows, as specified in the configuration file.
- The flows come in bursts, as specified in the configuration file.

4.11 Clients/Servers IP allocation scheme

Currently, there is one global IP pool for clients and servers. It serves all templates. All templates will allocate IP from this global pool. Each TRex client/server "dual-port" (pair of ports, such as port 0 for client, port 1 for server) has its own generator offset, taken from the config file. The offset is called `dual_port_mask`.

Example:

```
generator :
  distribution : "seq"
  clients_start : "16.0.0.1"
  clients_end   : "16.0.0.255"
  servers_start : "48.0.0.1"
  servers_end   : "48.0.0.255"
  dual_port_mask : "1.0.0.0"                        (1)
  tcp_aging      : 0
  udp_aging      : 0
```

(1) Offset to add per port pair. The reason for the “dual_port_mask” is to make static route configuration per port possible. With this offset, different ports have different prefixes.

For example, with four ports, TRex will produce the following ip ranges:

```
  port pair-0 (0,1) --> C (16.0.0.1-16.0.0.128  ) <-> S( 48.0.0.1 - 48.0.0.128)
  port pair-1 (2,3) --> C (17.0.0.129-17.0.0.255  ) <-> S( 49.0.0.129 - 49.0.0.255) + mask ↩
      ("1.0.0.0")
```

- Number of clients : 255
- Number of servers : 255
- The offset defined by “dual_port_mask” (1.0.0.0) is added for each port pair, but the total number of clients/servers will remain constant (255), and will not depend on the amount of ports.
- TCP/UDP aging is the time it takes to return the socket to the pool. It is required when the number of clients is very small and the template defines a very long duration.

If “dual_port_mask” was set to 0.0.0.0, both port pairs would have used the same ip range. For example, with four ports, we would have get the following ip range is :

```
  port pair-0 (0,1) --> C (16.0.0.1-16.0.0.128  ) <-> S( 48.0.0.1 - 48.0.0.128)
  port pair-1 (2,3) --> C (16.0.0.129-16.0.0.255  ) <-> S( 48.0.0.129 - 48.0.0.255)
```

Router configuration for this mode:
PBR is not necessary. The following configuration is sufficient.

```
interface TenGigabitEthernet1/0/0      (1)
 mac-address 0000.0001.0000
 mtu 4000
 ip address 11.11.11.11 255.255.255.0
!
`
interface TenGigabitEthernet1/1/0       (2)
 mac-address 0000.0001.0000
 mtu 4000
 ip address 22.11.11.11 255.255.255.0
!
interface TenGigabitEthernet1/2/0       (3)
 mac-address 0000.0001.0000
 mtu 4000
 ip address 33.11.11.11 255.255.255.0
!
interface TenGigabitEthernet1/3/0        (4)
 mac-address 0000.0001.0000
 mtu 4000
 ip address 44.11.11.11 255.255.255.0
 load-interval 30


ip route 16.0.0.0 255.0.0.0 22.11.11.12
ip route 48.0.0.0 255.0.0.0 11.11.11.12
ip route 17.0.0.0 255.0.0.0 44.11.11.12
ip route 49.0.0.0 255.0.0.0 33.11.11.12
```

1. Connected to TRex port 0 (client side)
2. Connected to TRex port 1 (server side)
3. Connected to TRex port 2 (client side)
4. Connected to TRex port 3(server side)

One server:
To support a template with one server, you can add "server_addr" keyword. Each port pair will be get different server IP (According to the "dual_port_mask" offset).

```
- name: cap2/dns.pcap
  cps : 1.0
  ipg : 10000
  rtt : 10000
  w   : 1
  server_addr : "48.0.0.1"      ❶
  one_app_server : true         ❷
```

1. Server IP.
2. Enable one server mode.

In TRex server, you will see the following statistics.

```
         Active-flows    :    19509  Clients :       504   Socket-util : 0.0670 %
         Open-flows      :   247395  Servers :     65408   Socket :     21277 Socket/Clients  ↩
             :  42.2
```

Note

- No backward compatibility with the old generator YAML format.
- When using -p option, TRex will not comply with the static route rules. Server-side traffic may be sent from the client side (port 0) and vice-versa. If you use the -p option, you must configure policy based routing to pass all traffic from router port 1 to router port 2, and vice versa.
- VLAN [?] feature does not comply with static route rules. If you use it, you also need policy based routing rules to pass packets from VLAN0 to VLAN1 and vice versa.
- Limitation: When using template with plugins (bundles), the number of servers must be higher than the number of clients.

4.11.1 More Details about IP allocations

Each time a new flow is created, TRex allocates new Client IP/port and Server IP. This 3-tuple should be distinct among active flows.

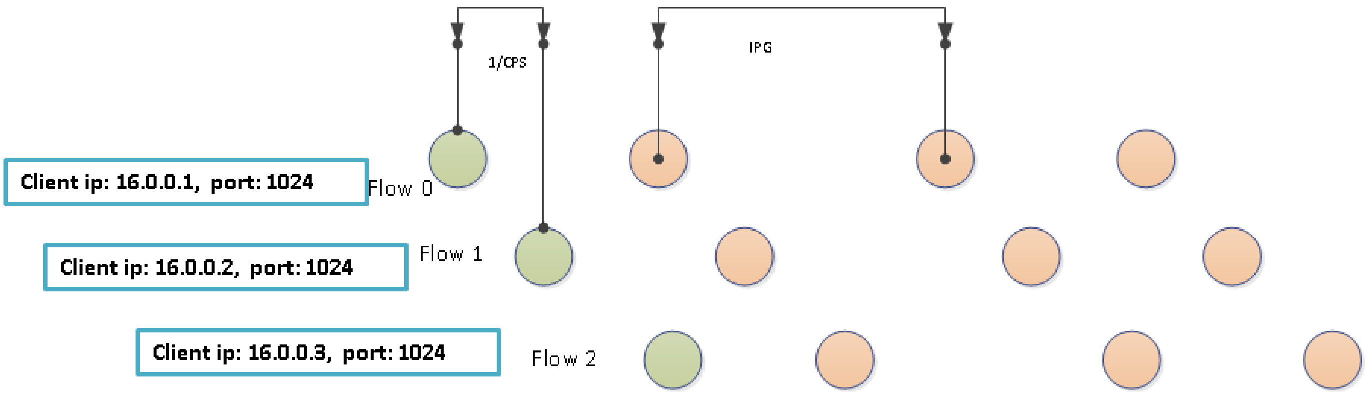
Currently, only sequential distribution is supported in IP allocation. This means the IP address is increased by one for each flow.

For example, if we have a pool of two IP addresses: 16.0.0.1 and 16.0.0.2, the allocation of client src/port pairs will be

```
16.0.0.0.1 [1024]
16.0.0.0.2 [1024]
16.0.0.0.1 [1025]
16.0.0.0.2 [1025]
16.0.0.0.1 [1026]
16.0.0.0.2 [1026]
...
```

4.11.2 How to determine the packet per second(PPS) and Bit per second (BPS)

- Let's look at an example of one flow with 4 packets.
- Green circles represent the first packet of each flow.
- The client ip pool starts from 16.0.0.1, and the distribution is seq.


1/CPS
IPG
Client ip: 16.0.0.1, port: 1024
Flow 0
Client ip: 16.0.0.2, port: 1024
Flow 1
Client ip: 16.0.0.3, port: 1024
Flow 2

$TotalPPS = \sum_{k=0}^{n}(CPS_k \times flow\_pkts_k)$

$Concurrentflow = \sum_{k=0}^{n} CPS_k \times flow\_duration_k$

The above formulas can be used to calculate the PPS. The TRex throughput depends on the PPS calculated above and the value of m (a multiplier given as command line argument -m).

The m value is a multiplier of total pcap files CPS. CPS of pcap file is configured on yaml file.

Let's take a simple example as below.

```
cap_info :
     - name: avl/first.pcap  < -- has 2 packets
       cps : 102.0
       ipg : 10000
       rtt : 10000
       w   : 1
     - name: avl/second.pcap < -- has 20 packets
       cps : 50.0
       ipg : 10000
       rtt : 10000
       w   : 1
```

The throughput is: *m*(CPS_1*flow_pkts+CPS_2*flow_pkts)*

So if the m is set as 1, the total PPS is : 102*2+50*20 = 1204 PPS.

The BPS depends on the packet size. You can refer to your packet size and get the BPS = PPS*Packet_size.

4.11.3 Per template allocation + future plans

- **1) per-template generator**

Multiple generators can be defined and assigned to different pcap file templates.

The YAML configuration is something like this:

```
generator :
         distribution : "seq"
         clients_start : "16.0.0.1"
         clients_end   : "16.0.1.255"
         servers_start : "48.0.0.1"
         servers_end   : "48.0.20.255"
         clients_per_gb : 201
         min_clients    : 101
         dual_port_mask : "1.0.0.0"
         tcp_aging      : 0
         udp_aging      : 0
         generator_clients :
           - name : "c1"
             distribution : "random"
             ip_start : "38.0.0.1"
             ip_end : "38.0.1.255"
             clients_per_gb : 201
             min_clients    : 101
             dual_port_mask : "1.0.0.0"
             tcp_aging      : 0
             udp_aging      : 0
         generator_servers :
            - name : "s1"
              distribution : "seq"
              ip_start : "58.0.0.1"
              ip_end : "58.0.1.255"
              dual_port_mask : "1.0.0.0"
cap_info :
    - name: avl/delay_10_http_get_0.pcap
      cps : 404.52
      ipg : 10000
      rtt : 10000
      w   : 1
    - name: avl/delay_10_http_post_0.pcap
      client_pool : "c1"
      server_pool : "s1"
      cps : 404.52
      ipg : 10000
      rtt : 10000
      w   : 1
```

- **2) More distributions will be supported in the future (normal distribution for example)**

Currently, only sequence and random are supported.

- **3) Histogram of tuple pool will be supported**

This feature will give the user more flexibility in defining the IP generator.

```
generator :
         client_pools:
            - name          : "a"
             distribution : "seq"
             clients_start : "16.0.0.1"
             clients_end   : "16.0.1.255"
             tcp_aging      : 0
             udp_aging      : 0

            - name          : "b"
```

```
              distribution : "random"
              clients_start : 26.0.0.1"
              clients_end   : 26.0.1.255"
              tcp_aging      : 0
              udp_aging      : 0

             - name          : "c"
               pools_list :
                  - name:"a"
                    probability: 0.8
                  - name:"b"
                    probability: 0.2
```

4.12 Measuring Jitter/Latency

To measure jitter/latency using independent flows (SCTP or ICMP), use `-l <Hz>` where Hz defines the number of packets to send from each port per second. This option measures latency and jitter. We can define the type of traffic used for the latency measurement using the `--l-pkt-mode <0-3>` option.

Option ID	Type
0	**default**, SCTP packets
1	ICMP echo request packets from both sides
2	Send ICMP requests from one side, and matching ICMP responses from other side. This is particulary usefull if your DUT drops traffic from outside, and you need to open pin hole to get the outside traffic in (for example when testing a firewall)
3	Send ICMP request packets with a constant 0 sequence number from both sides.

4.12.1 Interpretting Jitter/Latency Output

The command output of the `t-rex` utility with latency stats enabled looks similar to the following:

```
-Latency stats enabled
Cpu Utilization : 0.2 %  ❶                ❷          ❸          ❹           ❺
if|   tx_ok , rx_ok  , rx check ,error,       latency (usec) ,    Jitter        max window
  |         ,    ❻  ,        ❼ , ❽ ,    average   ,   max  ,    (usec)
-------------------------------------------------------------------------------------------------------

0 |    1002,    1002,          0,    0,         51  ,      69,        3       |   0  69   67  ↩
      ...
1 |    1002,    1002,          0,    0,         53  ,     196,        2       |   0  196   53  ↩
     ...
2 |    1002,    1002,          0,    0,         54  ,      71,        5       |   0  71   69  ↩
      ...
3 |    1002,    1002,          0,    0,         53  ,     193,        2       |   0  193   52  ↩
     ...
```

❶ Rx check and latency thread CPU utilization.

❻ `tx_ok` on port 0 should equal `rx_ok` on port 1, and vice versa, for all paired ports.

❼ Rx check rate as per `--rx-check`. For more information on Rx check, see Flow order/latency verification [rx_check] section.

❽ Number of packet errors detected. (various reasons, see `stateful_rx_core.cpp`)

❷ Average latency (in microseconds) across **all** samples since startup.

3. Maximum latency (in microseconds) across **all** samples since startup.

4. Jitter calculated as per RC3550 Appendix-A.8.

5. Maximum latency within a sliding window of 500 milliseconds. There are few values shown per port according to terminal width. The odlest value is on the left and most recent value (latest 500msec sample) on the right. This can help in identifying spikes of high latency clearing after some time. Maximum latency (<6>) is the total maximum over the entire test duration. To best understand this, run TRex with the latency option (-l) and watch the results.

Chapter 5

Advanced features

5.1 VLAN (dot1q) support

To add a VLAN tag to all traffic generated by TRex, add a “vlan” keyword in each port section in the platform config file, as described in the Platform YAML [trex_config_yaml_config_file] section.

You can specify a different VLAN tag for each port, or use VLAN only on some ports.

One useful application of this can be in a lab setup where you have one TRex and many DUTs, and you want to test a different DUT on each run, without changing cable connections. You can put each DUT on a VLAN of its own, and use different TRex platform configuration files with different VLANs on each run.

5.2 Utilizing maximum port bandwidth in case of asymmetric traffic profile

Note
If you want simple VLAN support, this is probably **not** the feature to use. This feature is used for load balancing. To configure VLAN support, see the “vlan” field in the Platform YAML [trex_config_yaml_config_file] section.

The VLAN Trunk TRex feature attempts to solve the router port bandwidth limitation when the traffic profile is asymmetric (example: Asymmetric SFR profile).

This feature converts asymmetric traffic to symmetric, from the port perspective, using router sub-interfaces. This requires TRex to send the traffic on two VLANs, as described below.

YAML format - This goes in the traffic YAML file.

```
  vlan       : { enable : 1  ,  vlan0 : 100 , vlan1 : 200 }
```

Example

```
- duration : 0.1
  vlan       : { enable : 1  ,  vlan0 : 100 , vlan1 : 200 }    ❶
```

❶ Enable load balance feature: vlan0==100 , vlan1==200
For a full file example, see the TRex source in: scripts/cap2/ipv4_load_balance.yaml

Problem definition:
Scenario: TRex with two ports and an SFR traffic profile.

Without VLAN/sub interfaces, all client emulated traffic is sent on port 0, and all server emulated traffic (example: HTTP response) on port 1.

```
TRex port 0 ( client) <-> [  DUT ] <-> TRex port 1 ( server)
```

Without VLAN support, the traffic is asymmetric. 10% of the traffic is sent from port 0 (client side), 90% from port 1 (server). Port 1 is the bottlneck (10Gb/s limit).

With VLAN/sub interfaces

```
TRex port 0 ( client VLAN0) <->  | DUT  | <-> TRex port 1 ( server-VLAN0)
TRex port 0 ( server VLAN1) <->  | DUT  | <-> TRex port 1 ( client-VLAN1)
```

In this case, traffic on vlan0 is sent as before, while for traffic on vlan1, the order is reversed (client traffic sent on port1 and server traffic on port0). TRex divides the flows evenly between the vlans. This results in an equal amount of traffic on each port.

Router configuation:

```
        !
        interface TenGigabitEthernet1/0/0      (1)
         mac-address 0000.0001.0000
         mtu 4000
         no ip address
         load-interval 30
        !
        i
        interface TenGigabitEthernet1/0/0.100
         encapsulation dot1Q 100               (2)
         ip address 11.77.11.1 255.255.255.0
         ip nbar protocol-discovery
         ip policy route-map vlan_100_p1_to_p2 (3)
        !
        interface TenGigabitEthernet1/0/0.200
         encapsulation dot1Q 200               (4)
         ip address 11.88.11.1 255.255.255.0
         ip nbar protocol-discovery
         ip policy route-map vlan_200_p1_to_p2 (5)
        !
        interface TenGigabitEthernet1/1/0
         mac-address 0000.0001.0000
         mtu 4000
         no ip address
         load-interval 30
        !
        interface TenGigabitEthernet1/1/0.100
         encapsulation dot1Q 100
         ip address 22.77.11.1 255.255.255.0
         ip nbar protocol-discovery
         ip policy route-map vlan_100_p2_to_p1
        !
        interface TenGigabitEthernet1/1/0.200
         encapsulation dot1Q 200
         ip address 22.88.11.1 255.255.255.0
         ip nbar protocol-discovery
         ip policy route-map vlan_200_p2_to_p1
        !
        arp 11.77.11.12 0000.0001.0000 ARPA        (6)
        arp 22.77.11.12 0000.0001.0000 ARPA

        route-map vlan_100_p1_to_p2 permit 10      (7)
         set ip next-hop 22.77.11.12
        !
        route-map vlan_100_p2_to_p1 permit 10
         set ip next-hop 11.77.11.12
        !

        route-map vlan_200_p1_to_p2 permit 10
         set ip next-hop 22.88.11.12
        !
        route-map vlan_200_p2_to_p1 permit 10
         set ip next-hop 11.88.11.12
        !
```

(1) Main interface must not have IP address.

(2) Enable VLAN1

(3) PBR configuration

(4) Enable VLAN2

(5) PBR configuration

(6) TRex destination port MAC address

(7) PBR configuration rules

5.3 Alternating MAC address by IP

With this feature, TRex replaces the MAC address(es) with the IP address.

`Note: This feature was originally requested by the Cisco ISG group.`

Example

```
- duration: 0.1
  ...
  mac_override_by_ip: true
```

There are several modes:

- **0**, (legacy value: **false**) - Take the MACs from port config/ARP query. This is default mode.
- **1**, (legacy value: **true**) - Replace only source MACs only in packets sent by client
- **2** - Replace all MACs in all directions.

5.3.1 Version v2.61 and older

- Only boolean values were accepted (mode 0 and 1)
- The MAC will consist of 4 IP bytes followed by 2 last bytes of MAC of TRex interface.
  For example, if interface MAC is AA:BB:CC:DD:EE:FF and IP is 1.2.3.4:
  Result MAC = 01:02:03:04:EE:FF

5.3.2 Version v2.62 and newer

- Added mode 2 to replace all MACs in all directions.
- Boolean values are accepted for backward compatibility.
- The MAC will consist of 2 first bytes of MAC of TRex interface followed by 4 IP bytes.
  For example, if interface MAC is AA:BB:CC:DD:EE:FF and IP is 1.2.3.4:
  Result MAC = AA:BB:01:02:03:04

5.4 IPv6 support

Support for IPv6 includes:

1. Support for pcap files containing IPv6 packets.
2. Ability to generate IPv6 traffic from pcap files containing IPv4 packets.
   The `--ipv6` command line option enables this feature. The keywords `src_ipv6` and `dst_ipv6` specify the most significant 96 bits of the IPv6 address. Example:

```
- duration : 10
  src_ipv6 : [0xFE80,0x0232,0x1002,0x0051,0x0000,0x0000]
  dst_ipv6 : [0x2001,0x0DB8,0x0003,0x0004,0x0000,0x0000]
  generator :
        distribution : "seq"
        clients_start : "16.0.0.1"
        clients_end   : "16.0.1.255"
        servers_start : "48.0.0.1"
        servers_end   : "48.0.0.255"
        clients_per_gb : 201
        min_clients    : 101
        dual_port_mask : "1.0.0.0"
        tcp_aging      : 1
        udp_aging      : 1
```

Note
src_ipv6 and dst_ipv6 should be in the root scope, not at IP pools.

The IPv6 address is formed by placing what would typically be the IPv4 address into the least significant 32 bits and copying the value provided in the src_ipv6/dst_ipv6 keywords into the most signficant 96 bits. If src_ipv6 and dst_ipv6 are not specified, the default is to form IPv4-compatible addresses (most signifcant 96 bits are zero).

There is IPv6 support for all plugins.

Example:

```
[bash]>sudo ./t-rex-64 -f cap2l/sfr_delay_10_1g.yaml -c 4 -p -l 100 -d 100000 -m 30  --ipv6
```

Limitations:

- TRex cannot generate both IPv4 and IPv6 traffic.
- The `--ipv6` switch must be specified even when using a pcap file containing only IPv6 packets.

Router configuration:

```
interface TenGigabitEthernet1/0/0
 mac-address 0000.0001.0000
 mtu 4000
 ip address 11.11.11.11 255.255.255.0
 ip policy route-map p1_to_p2
 load-interval 30
 ipv6 enable   ==> IPv6
 ipv6 address 2001:DB8:1111:2222::1/64                    ❶
 ipv6 policy route-map ipv6_p1_to_p2                      ❷
!


ipv6 unicast-routing                                      ❸

ipv6 neighbor 3001::2 TenGigabitEthernet0/1/0 0000.0002.0002   ❹
ipv6 neighbor 2001::2 TenGigabitEthernet0/0/0 0000.0003.0002

route-map ipv6_p1_to_p2 permit 10                              ❺
 set ipv6 next-hop 2001::2
!
route-map ipv6_p2_to_p1 permit 10
 set ipv6 next-hop 3001::2
!


asr1k(config)#ipv6 route 4000::/64 2001::2
asr1k(config)#ipv6 route 5000::/64 3001::2
```

❶ Enable IPv6

❷ Add pbr

❸ Enable IPv6 routing

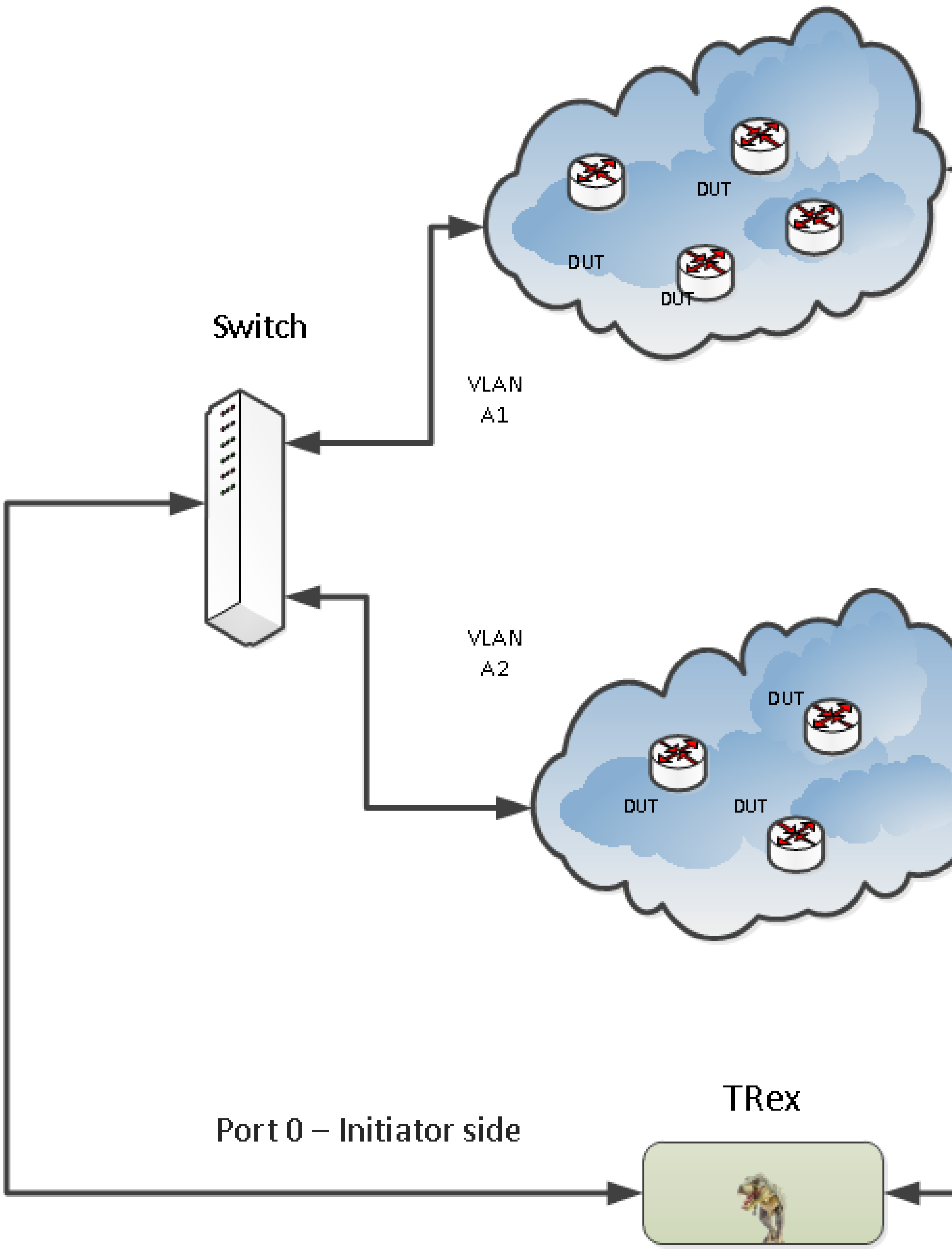
❹ MAC address setting. Should be TRex MAC.

❺ PBR configuraion

5.5 Client clustering configuration

TRex supports testing complex topologies with more than one DUT, using a feature called "client clustering". This feature allows specifying the distribution of clients that TRex emulates.

Consider the following topology:



Topology example

There are two clusters of DUTs. Using the configuration file, you can partition TRex emulated clients into groups, and define how they will be spread between the DUT clusters.

Group configuration includes:

- IP start range.

- IP end range.
- Initiator side configuration: Parameters affecting packets sent from client side.
- Responder side configuration: Parameters affecting packets sent from server side.

Note

It is important to understand that this is **complimentary** to the client generator configured per profile. It only defines how the clients will be spread between clusters.

In the following example, a profile defines a client generator.

```
[bash]>cat cap2/dns.yaml
- duration : 10.0
  generator :
          distribution : "seq"
          clients_start : "16.0.0.1"
          clients_end   : "16.0.0.255"
          servers_start : "48.0.0.1"
          servers_end   : "48.0.0.255"
          dual_port_mask : "1.0.0.0"
  cap_info :
     - name: cap2/dns.pcap
       cps : 1.0
       ipg : 10000
       rtt : 10000
       w   : 1
```

Goal:

- Create two clusters with 4 and 3 devices, respectively.
- Send **80%** of the traffic to the upper cluster and **20%** to the lower cluster. Specify the DUT to which the packet will be sent by MAC address or IP. (The following example uses the MAC address. The instructions after the example indicate how to change to IP-based.)

Create the following cluster configuration file:

```
#
# Client configuration example file
# The file must contain the following fields
#
# 'vlan'    - if the entire configuration uses VLAN,
#             each client group must include vlan
#             configuration
#
# 'groups'  - each client group must contain range of IPs
#             and initiator and responder section
#             'count' represents the number of different DUTs
#             in the group.
#

# 'true' means each group must contain VLAN configuration. 'false' means no VLAN config ↩
    allowed.
vlan: true

groups:

-    ip_start  : 16.0.0.1
     ip_end    : 16.0.0.204
     initiator :
                 vlan    : 100
                 dst_mac : "00:00:00:01:00:00"
     responder :
                 vlan    : 200
                 dst_mac : "00:00:00:02:00:00"

     count     : 4

-    ip_start  : 16.0.0.205
     ip_end    : 16.0.0.255
     initiator :
                 vlan    : 101
                 dst_mac : "00:00:01:00:00:00"

     responder:
                 vlan    : 201
                 dst_mac : "00:00:02:00:00:00"

     count     : 3
```

The above configuration divides the generator range of 255 clients to two clusters. The range of IPs in all groups in the client configuration file must cover the entire range of client IPs from the traffic profile file.

MAC addresses will be allocated incrementally, with a wrap around after “count” addresses.

Example:

Initiator side (packets with source in 16.x.x.x net):

- 16.0.0.1 → 48.x.x.x - dst_mac: 00:00:00:01:00:00 vlan: 100
- 16.0.0.2 → 48.x.x.x - dst_mac: 00:00:00:01:00:01 vlan: 100
- 16.0.0.3 → 48.x.x.x - dst_mac: 00:00:00:01:00:02 vlan: 100
- 16.0.0.4 → 48.x.x.x - dst_mac: 00:00:00:01:00:03 vlan: 100
- 16.0.0.5 → 48.x.x.x - dst_mac: 00:00:00:01:00:00 vlan: 100
- 16.0.0.6 → 48.x.x.x - dst_mac: 00:00:00:01:00:01 vlan: 100

Responder side (packets with source in 48.x.x.x net):

- 48.x.x.x → 16.0.0.1 - dst_mac(from responder) : "00:00:00:02:00:00" , vlan:200
- 48.x.x.x → 16.0.0.2 - dst_mac(from responder) : "00:00:00:02:00:01" , vlan:200

and so on.

The MAC addresses of DUTs must be changed to be sequential. Another option is to replace: `dst_mac :<ip-address>` with:
`next_hop :<ip-address>`

For example, the first group in the configuration file would be:

```
-    ip_start  : 16.0.0.1
     ip_end    : 16.0.0.204
     initiator :
                 vlan     : 100
                 next_hop : 1.1.1.1
                 src_ip   : 1.1.1.100
     responder :
                 vlan     : 200
                 next_hop : 2.2.2.1
                 src_ip   : 2.2.2.100

     count     : 4
```

In this case, TRex attempts to resolve the following addresses using ARP:

1.1.1.1, 1.1.1.2, 1.1.1.3, 1.1.1.4 (and the range 2.2.2.1-2.2.2.4)

If not all IPs are resolved, TRex exits with an error message.

`src_ip` is used to send gratuitous ARP, and for filling relevant fields in ARP request. If no `src_ip` is given, TRex looks for the source IP in the relevant port section in the platform configuration file (/etc/trex_cfg.yaml). If none is found, TRex exits with an error message.

If a client config file is given, TRex ignores the `dest_mac` and `default_gw` parameters from the platform configuration file.

Now, streams will look like:

Initiator side (packets with source in 16.x.x.x net):

- 16.0.0.1 → 48.x.x.x - dst_mac: MAC of 1.1.1.1 vlan: 100
- 16.0.0.2 → 48.x.x.x - dst_mac: MAC of 1.1.1.2 vlan: 100
- 16.0.0.3 → 48.x.x.x - dst_mac: MAC of 1.1.1.3 vlan: 100
- 16.0.0.4 → 48.x.x.x - dst_mac: MAC of 1.1.1.4 vlan: 100
- 16.0.0.5 → 48.x.x.x - dst_mac: MAC of 1.1.1.1 vlan: 100
- 16.0.0.6 → 48.x.x.x - dst_mac: MAC of 1.1.1.2 vlan: 100

Responder side (packets with source in 48.x.x.x net):

- 48.x.x.x → 16.0.0.1 - dst_mac: MAC of 2.2.2.1 , vlan:200
- 48.x.x.x → 16.0.0.2 - dst_mac: MAC of 2.2.2.2 , vlan:200

Note
It is important to understand that the IP to MAC coupling (with either MAC-based or IP-based configuration) is done at the beginning and never changes. For example, in a MAC-based configuration:

- Packets with source IP 16.0.0.2 will always have VLAN 100 and dst MAC 00:00:00:01:00:01.
- Packets with destination IP 16.0.0.2 will always have VLAN 200 and dst MAC 00:00:00:02:00:01.

Consequently, you can predict exactly which packet (and how many packets) will go to each DUT.

Usage:

```
[bash]>sudo ./t-rex-64 -f cap2/dns.yaml --client_cfg my_cfg.yaml
```

5.5.1 Latency with Cluster mode

Latency streams client IP is taken from the first IP in the default client pool. Each dual port will have one Client IP. In case of cluster configuration this is a limitation as you can have a topology with many paths.

Traffic profile

```
- duration : 10.0
  generator :
          distribution : "seq"
          clients_start : "16.0.0.1"
          clients_end   : "16.0.0.255"
          servers_start : "48.0.0.1"
          servers_end   : "48.0.0.255"
          dual_port_mask : "1.0.0.0"
  cap_info :
     - name: cap2/dns.pcap
       cps : 1.0
       ipg : 10000
       rtt : 10000
       w   : 1
```

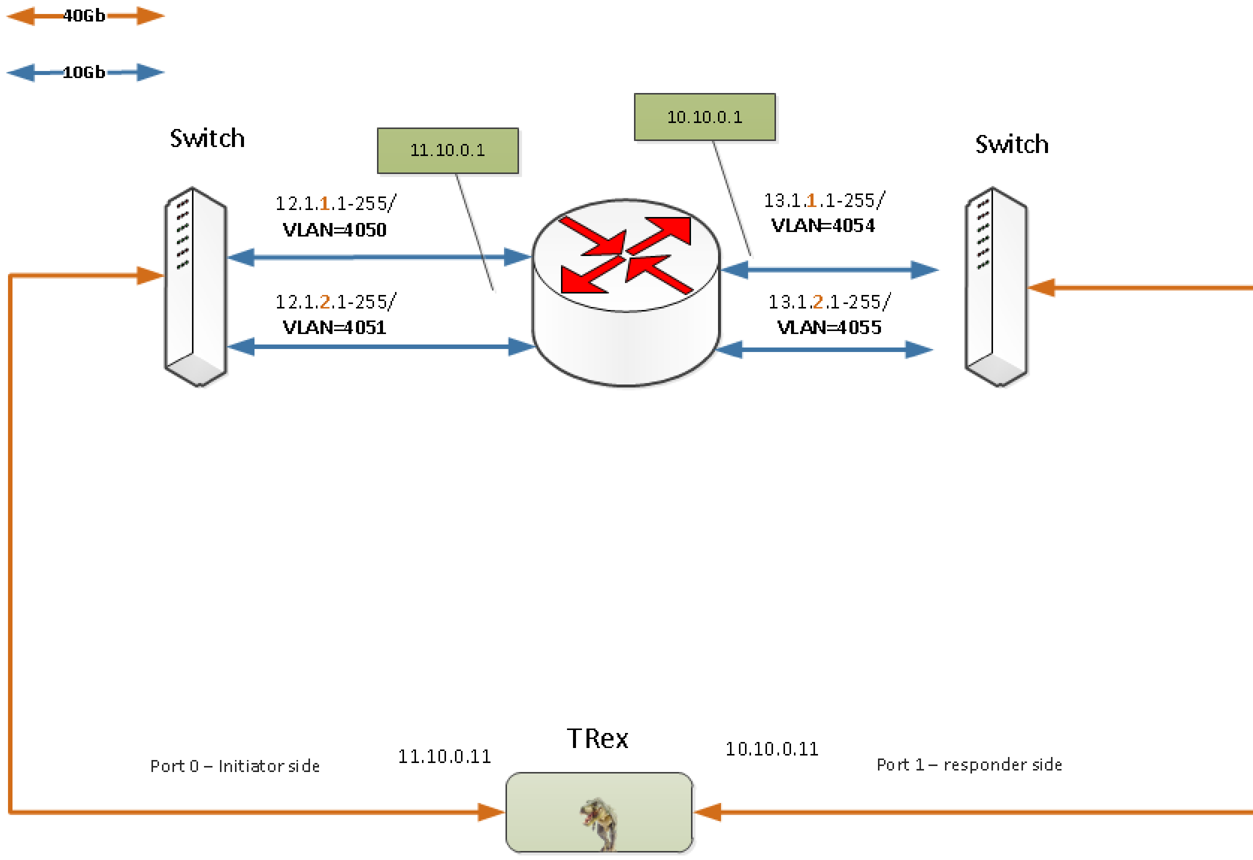
Cluster profile

```
vlan: true

groups:

-    ip_start  : 16.0.0.1
     ip_end    : 16.0.0.204
     initiator :
                 vlan    : 100
                 dst_mac : "00:00:00:01:00:00"
     responder :
                 vlan    : 200
                 dst_mac : "00:00:00:02:00:00"

     count     : 4
```

For example, in this case 16.0.0.1→48.0.0.1 ICMP will be the flow for latency. The Cluster configuration of this flow will be taken from cluster file (VLAN=100, dst_mac : "00:00:00:01:00:00")

5.5.2 Clustering example

In this example we have one DUT with four 10gb interfaces and one TRex with two 40Gb/sec interfaces and we want to convert the traffic from 2 TRex interfaces to 4 DUT Interfaces. The folowing figure shows the topology



Cluster example

For this topology the following traffic and cluster configuration file were created

Traffic profile

```
- duration : 10.0
  generator :
          distribution : "seq"
          clients_start : "12.1.1.1"        ❶
          clients_end   : "12.1.1.1"
          servers_start : "13.1.1.1"
          servers_end   : "13.1.1.1"
          dual_port_mask : "0.1.0.0"
          generator_clients :
            - name : "c1"                     ❷
              distribution : "seq"
              ip_start : "12.1.1.2"
              ip_end :    "12.1.1.255"
            - name : "c2"                     ❸
              distribution : "seq"
              ip_start : "12.1.2.1"
              ip_end :    "12.1.2.255"
          generator_servers :
             - name : "s1"
               distribution : "seq"
               ip_start : "13.1.1.2"
               ip_end :    "13.1.1.255"
             - name : "s2"
               distribution : "seq"
               ip_start : "13.1.2.1"
               ip_end :    "13.1.2.255"
  cap_ipg : true
  cap_info :
    - name: file10k.pcap
```

```
      client_pool : "c2"
      server_pool : "s2"
      cps : 1
      ipg : 20000
      rtt : 20000
      w   : 1
    - name: file10k.pcap
      client_pool : "c1"
      server_pool : "s1"
      cps : 1
      ipg : 20000
      rtt:  20000
      w   : 1
```

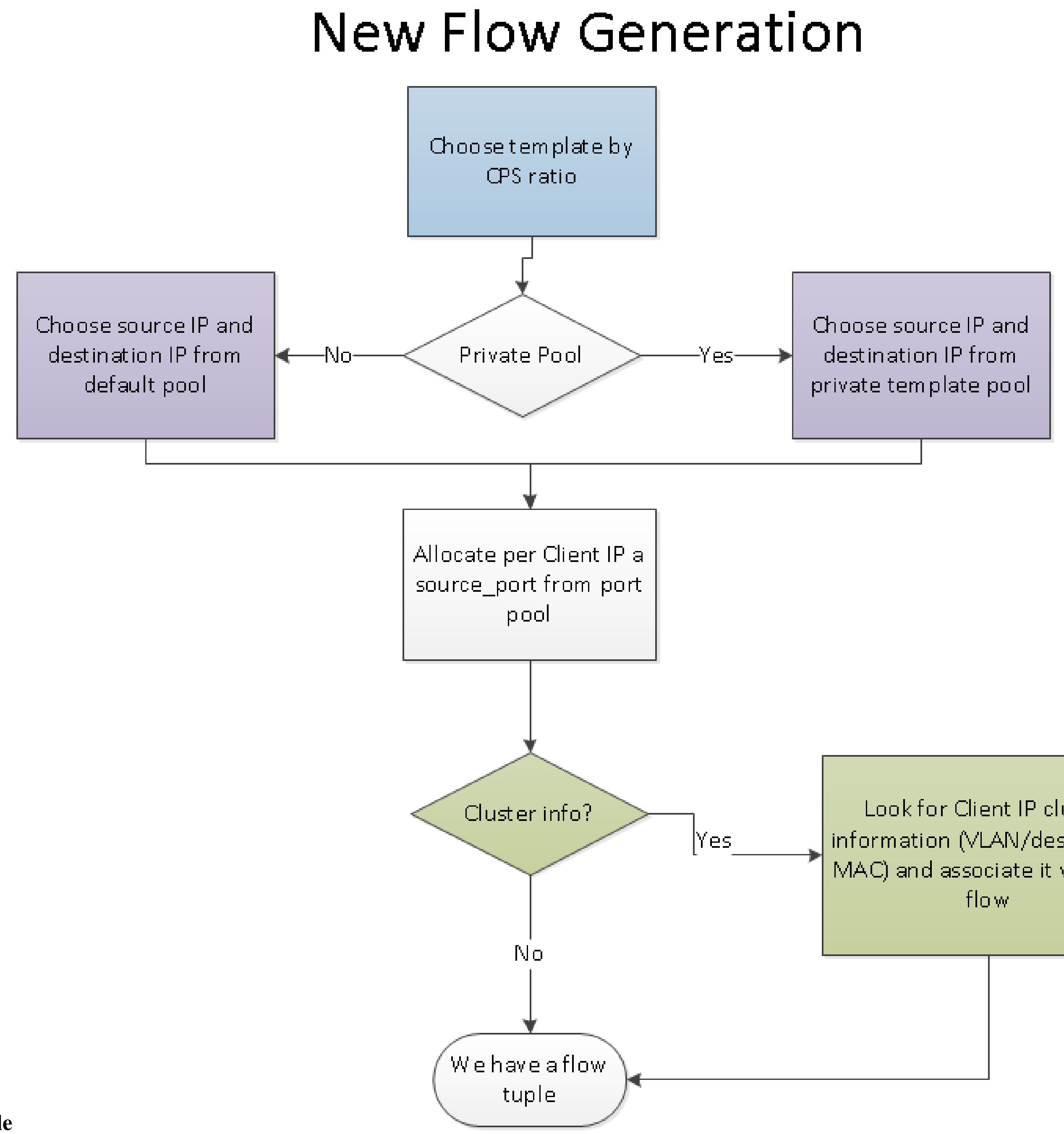
1. Latency flow will be IP 12.1.1.1<→13.1.1.1/vlan=4050/next-hop=11.10.0.1
2. First clients pool
3. Second clients pool

Cluster profile

```
lan: true

groups:

-    ip_start  : 12.1.1.1         (1)
     ip_end    : 12.1.1.255
     initiator :
                 vlan     : 4050
                 next_hop : 11.10.0.1
                 src_ip   : 11.10.0.11
     responder :
                 vlan     : 4054
                 next_hop : 10.10.0.1
                 src_ip   : 10.10.0.11
     count     : 1

-    ip_start  : 12.1.2.1
     ip_end    : 12.1.2.255
     initiator :
                 vlan     : 4051
                 next_hop : 11.10.0.2
                 src_ip   : 11.10.0.11
     responder :
                 vlan     : 4055
                 next_hop : 10.10.0.2
                 src_ip   : 10.10.0.11
     count     : 1
```

1. **All** the IPs from client pools and default pool should be maped in this file, it is possible to have wider range in cluster file

The following diagram shows how new flow src_ip/dest_ip/next-hop/vlan is choosen with cluster file


New Flow Generation
Choose template by CPS ratio
Private Pool
No
Yes
Choose source IP and destination IP from default pool
Choose source IP and destination IP from private template pool
Allocate per Client IP a source_port from port pool
Cluster info?
Yes
No
Look for Client IP cl information (VLAN/des MAC) and associate it flow
We have a flow tuple

Cluster example

Note
Latency stream will check only 12.1.1.1/4050 path (one DUT intertface) there is no way to check latency on all the ports in current version

Note
DUT should have a static route to move packets from client to server and vice versa, as traffic is not in the same subnet of the ports

An example of one flow generation

1. next hop resolotion. TRex resolve all the next hop option e.g. 11.10.0.1/4050 11.11.0.1/4051

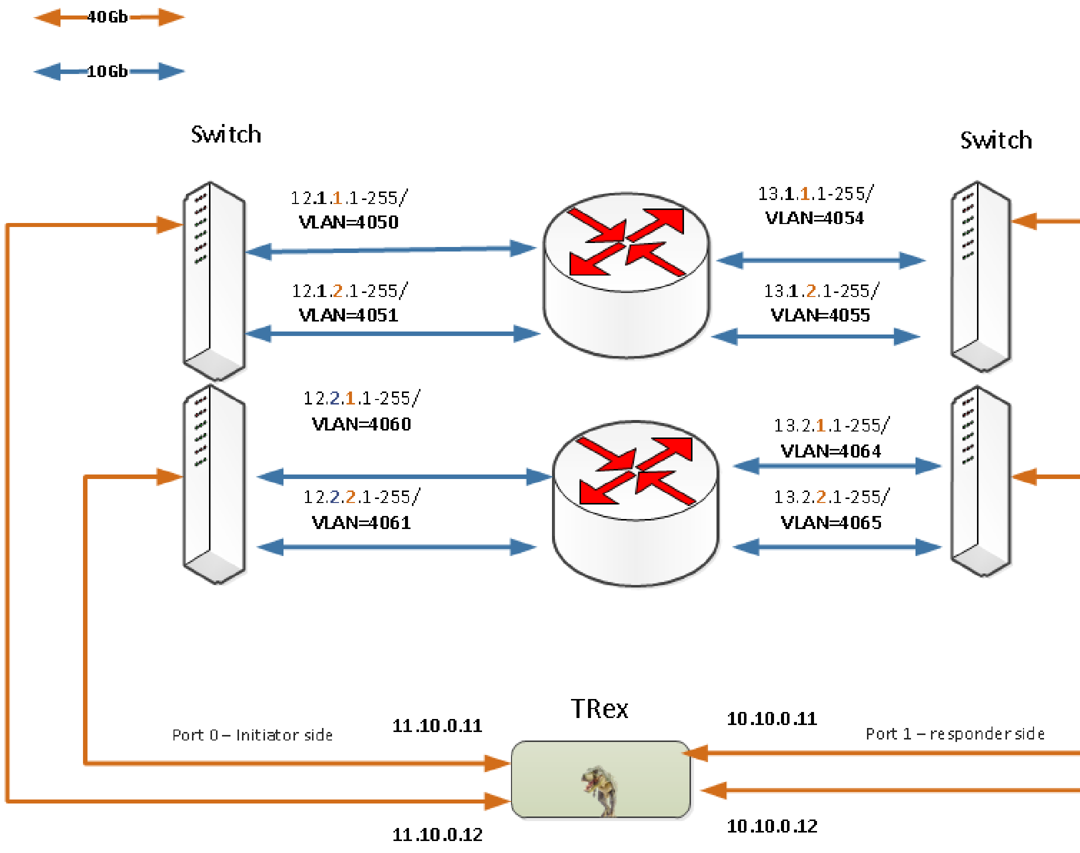
2. Choose template by CPS, 50% probability for each. take template #1
3. SRC_IP=12.1.1.2, DEST_IP=13.1.1.2
4. Allocate src_port for 12.1.1.2 ==>src_port=1025 for the first flow of client=12.1.1.2
5. Associate the next-hop from cluster pool. In this case 12.1.1.2 has the following information 5.1 client side : VLAN=4050 and MAC of 11.10.0.1 (Initiator) 5.2 server side : VLAN=4054 and MAC of 10.10.0.1 (Responder)

to run this using FirePower and NAT learning and checksum offload

```
[bash]>sudo ./t-rex-64 -f profile.yaml --client_cfg cluster.yaml -m 10 -d 1000 -l 1000 --l- ↩
    pkt-mode 2 --learn-mode 1 --checksum-offload
```

5.5.3 Clustering example - four ports

In this example there TRex has two dual ports. We can use traffic mask to differentiate the ip range.



Cluster example with four ports

The profile file is the similar to the previous one with the difference that we need to add more mapping in the cluster file. Latency can be tested on another path in the configuration.

Traffic profile

```
- duration : 10.0
  generator :
          distribution : "seq"
          clients_start : "12.1.1.1"
          clients_end   : "12.1.1.1"
          servers_start : "13.1.1.1"
          servers_end   : "13.1.1.1"
          dual_port_mask : "0.1.0.0"       ❶
          generator_clients :
            - name : "c1"
              distribution : "seq"
              ip_start : "12.1.1.2"
              ip_end :   "12.1.1.255"
            - name : "c2"
              distribution : "seq"
              ip_start : "12.1.2.1"
              ip_end :   "12.1.2.255"
          generator_servers :
             - name : "s1"
               distribution : "seq"
               ip_start : "13.1.1.2"
               ip_end :   "13.1.1.255"
             - name : "s2"
               distribution : "seq"
               ip_start : "13.1.2.1"
               ip_end :   "13.1.2.255"
  cap_ipg : true
  cap_info :
    - name: file10k.pcap
      client_pool : "c2"
      server_pool : "s2"
      cps : 1
      ipg : 20000
      rtt : 20000
      w   : 1
    - name: file10k.pcap
      client_pool : "c1"
      server_pool : "s1"
      cps : 1
      ipg : 20000
      rtt:  20000
      w   : 1
```

1 The mask is 0.1.0.0

Cluster profile

```
lan: true

groups:

-    ip_start  : 12.1.1.1
     ip_end    : 12.1.1.255
     initiator :
                 vlan     : 4050
                 next_hop : 11.10.0.1
                 src_ip   : 11.10.0.11
     responder :
                 vlan     : 4054
                 next_hop : 10.10.0.1
                 src_ip   : 10.10.0.11
     count     : 1

-    ip_start  : 12.1.2.1
     ip_end    : 12.1.2.255
     initiator :
                 vlan     : 4051
                 next_hop : 11.10.0.2
                 src_ip   : 11.10.0.11
     responder :
```

```
                vlan     : 4055
                next_hop : 10.10.0.2
                src_ip   : 10.10.0.11
     count     : 1

-    ip_start  : 12.2.1.1            (1)
     ip_end    : 12.2.1.255
     initiator :
                vlan     : 4060
                next_hop : 11.10.0.3
                src_ip   : 11.10.0.11
     responder :
                vlan     : 4064
                next_hop : 10.10.0.3
                src_ip   : 10.10.0.11
     count     : 1


-    ip_start  : 12.2.2.1
     ip_end    : 12.2.2.255
     initiator :
                vlan     : 4061
                next_hop : 11.10.0.4
                src_ip   : 11.10.0.11
     responder :
                vlan     : 4065
                next_hop : 10.10.0.4
                src_ip   : 10.10.0.11
     count     : 1
```

(1) We added more clusters beacuse more IPs will be generated (+mask)

5.6 NAT support

TRex can learn dynamic NAT/PAT translation. To enable this feature, use the
`--learn-mode <mode>`
switch at the command line. To learn the NAT translation, TRex must embed information describing which flow a packet belongs to, in the first packet of each flow. TRex can do this using one of several methods, depending on the chosen <mode>.

Mode 1:
`--learn-mode 1`
TCP flow: Flow information is embedded in the ACK of the first TCP SYN.
UDP flow: Flow information is embedded in the IP identification field of the first packet in the flow.
This mode was developed for testing NAT with firewalls (which usually do not work with mode 2). In this mode, TRex also learns and compensates for TCP sequence number randomization that might be done by the DUT. TRex can learn and compensate for seq num randomization in both directions of the connection.

Mode 2:
`--learn-mode 2`
Flow information is added in a special IPv4 option header (8 bytes long 0x10 id). This option header is added only to the first packet in the flow. This mode does not work with DUTs that drop packets with IP options (for example, Cisco ASA firewall).

Mode 3:
`--learn-mode 3`
Similar to mode 1, but TRex does not learn the seq num randomization in the server→client direction. This mode can provide better connections-per-second performance than mode 1. But for all existing firewalls, the mode 1 cps rate is adequate.

5.6.1 Examples

simple HTTP traffic

```
[bash]>sudo ./t-rex-64 -f cap2/http_simple.yaml -c 4  -l 1000 -d 100000 -m 30  --learn-mode ↩
     1
```

SFR traffic without bundling/ALG support

```
[bash]>sudo ./t-rex-64 -f avl/sfr_delay_10_1g_no_bundling.yaml -c 4  -l 1000 -d 100000 -m  ↩
    10  --learn-mode 2
```

NAT terminal counters:

```
-Global stats enabled
 Cpu Utilization : 0.6  %  33.4 Gb/core
 Platform_factor : 1.0
 Total-Tx        :       3.77 Gbps   NAT time out    :      917 ❶ (0 in wait for syn+ack) ❷
 Total-Rx        :       3.77 Gbps   NAT aged flow id:        0 ❸
 Total-PPS       :     505.72 Kpps   Total NAT active:      163 ❹ (12 waiting for syn) ❺
 Total-CPS       :      13.43 Kcps   Total NAT opened:    82677 ❻
```

❶ Number of connections for which TRex had to send the next packet in the flow, but did not learn the NAT translation yet. Should be 0. Usually, a value different than 0 is seen if the DUT drops the flow (probably because it cannot handle the number of connections).

❸ Number of flows for which the flow had already aged out by the time TRex received the translation info. A value other than 0 is rare. Can occur only when there is very high latency in the DUT input/output queue.

❹ Number of flows for which TRex sent the first packet before learning the NAT translation. The value depends on the connection per second rate and round trip time.

❻ Total number of translations over the lifetime of the TRex instance. May be different from the total number of flows if template is uni-directional (and consequently does not need translation).

❷ Out of the timed-out flows, the number that were timed out while waiting to learn the TCP seq num randomization of the server→client from the SYN+ACK packet. Seen only in --learn-mode 1.

❺ Out of the active NAT sessions, the number that are waiting to learn the client→server translation from the SYN packet. (Others are waiting for SYN+ACK from server.) Seen only in --learn-mode 1.

Configuration for Cisco ASR1000 Series:

This feature was tested with the following configuration and the
sfr_delay_10_1g_no_bundling.yaml
traffic profile. The client address range is 16.0.0.1 to 16.0.0.255

```
interface TenGigabitEthernet1/0/0                  ❶
 mac-address 0000.0001.0000
 mtu 4000
 ip address 11.11.11.11 255.255.255.0
 ip policy route-map p1_to_p2
 ip nat inside                                     ❷
 load-interval 30
!

interface TenGigabitEthernet1/1/0
 mac-address 0000.0001.0000
 mtu 4000
 ip address 11.11.11.11 255.255.255.0
 ip policy route-map p1_to_p2
 ip nat outside                                  ❸
 load-interval 30

ip  nat pool my 200.0.0.0 200.0.0.255 netmask 255.255.255.0  ❹

ip nat inside source list 7 pool my overload
access-list 7 permit 16.0.0.0 0.0.0.255                      ❺

ip nat inside source list 8 pool my overload                 ❻
access-list 8 permit 17.0.0.0 0.0.0.255
```

❶ Must be connected to TRex Client port (router inside port)

❷ NAT inside

❸ NAT outside

❹ Pool of outside address with overload

❺ Match TRex YAML client range

❻ In case of dual port TRex

Limitations:

1. The IPv6-IPv6 NAT feature does not exist on routers, so this feature can work only with IPv4.
2. Does not support NAT64.
3. Bundling/plugin is not fully supported. Consequently, sfr_delay_10.yaml does not work. Use sfr_delay_10_no_bundling.yan instead.

Note

- `--learn-verify` is a TRex debug mechanism for testing the TRex learn mechanism.
- Need to run it when DUT is configured without NAT. It will verify that the inside_ip==outside_ip and inside_port==outside_port.

5.7 Flow order/latency verification

In normal mode (without the feature enabled), received traffic is not checked by software. Hardware (Intel NIC) testing for dropped packets occurs at the end of the test. The only exception is the Latency/Jitter packets. This is one reason that with TRex, you **cannot** check features that terminate traffic (for example TCP Proxy).

To enable this feature, add `--rx-check <sample>` to the command line options, where <sample> is the sample rate. The number of flows that will be sent to the software for verification is (1/(sample_rate). For 40Gb/sec traffic you can use a sample rate of 1/128. Watch for Rx CPU% utilization.

Note
This feature changes the TTL of the sampled flows to 255 and expects to receive packets with TTL 254 or 255 (one routing hop). If you have more than one hop in your setup, use `--hops` to change it to a higher value. More than one hop is possible if there are number of routers betwean TRex client side and TRex server side.

This feature ensures that:

- Packets get out of DUT in order (from each flow perspective).
- There are no packet drops (no need to wait for the end of the test). Without this flag, you must wait for the end of the test in order to identify packet drops, because there is always a difference between TX and Rx, due to RTT.

Full example

```
[bash]>sudo ./t-rex-64 -f avl/sfr_delay_10_1g.yaml -c 4 -p -l 100 -d 100000 -m 30  --rx- ↩
    check 128
```

```
Cpu Utilization : 0.1 %  ↩
                                                                               (1)
 if|   tx_ok , rx_ok  , rx   ,error,    average   ,   max         , Jitter ,  max window
   |         ,        , check,      , latency(usec),latency (usec) ,(usec)  ,  ↩
                         (2)
 --------------------------------------------------------------------------------
 0 |     1002,    1002,      2501,   0,         61  ,      70,       3      |  60
 1 |     1002,    1002,      2012,   0,         56  ,      63,       2      |  50
 2 |     1002,    1002,      2322,   0,         66  ,      74,       5      |  68
 3 |     1002,    1002,      1727,   0,         58  ,      68,       2      |  52

 Rx Check stats enabled  ↩
                                                                                (3)
  ↩
    ------------------------------------------------------------------------------------------ ↩

 rx check:  avg/max/jitter latency,       94  ,      744,       49      |  252  287  309  ↩
                (4)

 active flows: (5)     10, fif: (6)     308,  drop:        0, errors:        0            (7)
  ↩
    ------------------------------------------------------------------------------------------ ↩
```

1. CPU% of the Rx thread. If it is too high, **increase** the sample rate.
2. The latency-specific packet measurement section. See Measuring Jittery/Latency [jitter_latency] section for detailed output interpretation.
3. Rx Check section. For more detailed info, press *r* during the test or at the end of the test.
4. Average latency, max latency, jitter on the template flows in microseconds. This is usually **higher** than the latency check packet because the feature works more on this packet.
7. Drop counters and errors counter should be zero. If not, press *r* to see the full report or view the report at the end of the test.
6. fif - First in flow. Number of new flows handled by the Rx thread.
5. active flows - number of active flows handled by rx thread

Press R to Display Full Report

```
 m_total_rx                             : 2
 m_lookup                               : 2
 m_found                                : 1
 m_fif                                  : 1
 m_add                                  : 1
 m_remove                               : 1
```

```
m_active                                : 0
                                                           ❶
0  0  0  0  1041  0  0  0  0  0  0  0  0  min_delta  : 10 usec
cnt         : 2
high_cnt    : 2
max_d_time : 1041 usec
sliding_average    : 1 usec                                ❷
precent     : 100.0 %
histogram
-----------
h[1000]  :  2
tempate_id_ 0 , errors:       0,  jitter: 61               ❸
tempate_id_ 1 , errors:       0,  jitter: 0
tempate_id_ 2 , errors:       0,  jitter: 0
tempate_id_ 3 , errors:       0,  jitter: 0
tempate_id_ 4 , errors:       0,  jitter: 0
tempate_id_ 5 , errors:       0,  jitter: 0
tempate_id_ 6 , errors:       0,  jitter: 0
tempate_id_ 7 , errors:       0,  jitter: 0
tempate_id_ 8 , errors:       0,  jitter: 0
tempate_id_ 9 , errors:       0,  jitter: 0
tempate_id_10 , errors:       0,  jitter: 0
tempate_id_11 , errors:       0,  jitter: 0
tempate_id_12 , errors:       0,  jitter: 0
tempate_id_13 , errors:       0,  jitter: 0
tempate_id_14 , errors:       0,  jitter: 0
tempate_id_15 , errors:       0,  jitter: 0
ager :
m_st_alloc                                : 1
m_st_free                                 : 0
m_st_start                                : 2
m_st_stop                                 : 1
m_st_handle                               : 0
```

❶ Errors, if any, shown here

❷ Low pass filter on the active average of latency events

❸ Error per template info

Notes and Limitations:

- To receive the packets TRex does the following:
  - Changes the TTL to 0xff and expects 0xFF (loopback) or oxFE (route). (Use `--hop` to configure this value.)
  - Adds 24 bytes of metadata as ipv4/ipv6 option header.

Chapter 6

Reference

6.1 Traffic YAML (-f argument of stateful)

6.1.1 Global Traffic YAML section

```
- duration : 10.0                                    ❶
  generator :                                        ❷
          distribution : "seq"
          clients_start : "16.0.0.1"                 ❸
          clients_end   : "16.0.0.255"
          servers_start : "48.0.0.1"
          servers_end   : "48.0.0.255"
          clients_per_gb : 201                       ❹
          min_clients    : 101                       ❺
          dual_port_mask : "1.0.0.0"
          tcp_aging      : 1                         ❻
          udp_aging      : 1                         ❼
  cap_ipg    : true                                      ❽
  cap_ipg_min    : 30                                    ❾
  cap_override_ipg    : 200                              ❿
  vlan       : { enable : 1  ,  vlan0 : 100 , vlan1 : 200 } ⓫
  mac_override_by_ip : true  ⓬
```

❶ Test duration (seconds). Can be overridden using the `-d` option.

❷ See full explanation on generator section here.

❽ true (default) indicates that the IPG is taken from the cap file (also taking into account cap_ipg_min and cap_override_ipg if they exist). false indicates that IPG is taken from per template section.

❾ The following two options can set the min ipg in microseconds: (if (pkt_ipg<cap_ipg_min) { pkt_ipg=cap_override_ipg})

❿ Value to override (microseconds), as described in note above.

⓫ Enable load balance feature. See trex load balance section [trex_load_bal] for info.

⓬ Enable MAC address replacement based on IP. Full explanation.

❸ See tuple generator

❹ Deprecated. not used

❺ Deprecated. not used

6 time in sec to linger the deallocation of TCP flows (in particular return the src_port to the pool). Good for cases when there is a very high socket utilization (>50%) and there is a need to verify that socket source port are not wrapped and reuse. Default value is zero. Better to keep it like that from performance point of view. High value could create performance penalty

7 same as #11 for UDP flows

6.1.2 Timer Wheel section configuration

(from v2.13) see Timer Wheel section [timer_w]

6.1.3 Per template section

```
     - name: cap2/dns.pcap            1
       cps : 10.0                     2
       ipg : 10000                    3
       rtt : 10000                    4
       w   : 1                        5
       server_addr : "48.0.0.7"       6
       one_app_server : true          7
       multi_flow_enabled: true       8
       flows_dirs: [0, 1]             9
       keep_src_port: true            10
```

1 The name of the template pcap file. Can be relative path from the t-rex-64 image directory, or an absolute path. The pcap file should include only one flow. (Exception: in case of plug-ins).

2 Connection per second. This is the value that will be used if specifying -m 1 from command line (giving -m x will multiply this

3 If the global section of the YAML file includes `cap_ipg :false`, this line sets the inter-packet gap in microseconds.

4 Should be set to the same value as ipg (microseconds).

5 Default value: w=1. This indicates to the IP generator how to generate the flows. If w=2, two flows from the same template will be generated in a burst (more for HTTP that has burst of flows).

6 If `one_app_server` is set to true, then all flows of this template will use the same server.

7 If the same server address is required, set this value to true. (Default is false)

8 New in version v2.62. When set, allows several flows to be present in template. Requires flows_dirs to be configured to specify direction for **each** flow.
Note: IP values from the pool as well as source port will be the same for all flows in the pcap.

9 New in version v2.62. Sets direction of flow(s). If set to 0, has no effect. If set to 1, server will initiate the flow (with its IP) towards client with original dest port.
Note: does **not** work with --learn modes.

10 New in version v2.66. Keep original TCP/UDP source port from pcap.

6.2 Platform YAML (--cfg argument)

The configuration file, in YAML format, configures TRex behavior, including:

- IP address or MAC address for each port (source and destination).

- Masked interfaces, to ensure that TRex does not try to use the management ports as traffic ports.
- Changing the zmq/telnet TCP port.

You specify which config file to use by adding --cfg <file name> to the command line arguments.
If no --cfg given, the default `/etc/trex_cfg.yaml` is used.
Configuration file examples can be found in the `$TREX_ROOT/scripts/cfg` folder.

6.2.1 Basic Configuration

```
     - port_limit    : 2    #mandatory ❶
       version       : 2    #mandatory ❷
       interfaces    : ["03:00.0", "03:00.1"]   #mandatory ❸
       #ext_dpdk_opt: ['--vdev=net_vdev_netvsc0,iface=eth1', '--vdev=net_vdev_netvsc1,iface ↩
           =eth2'] # ask DPDK for failsafe #❹
       #interfaces_vdevs : ['net_failsafe_vsc0','net_failsafe_vsc1'] # use failsafe  #❺
       #enable_zmq_pub  : true #optional ❻
       #zmq_pub_port    : 4500 #optional ❼
       #zmq_rpc_port    : 4501 #optional ❽
       #prefix          : setup1 #optional ❾
       #limit_memory    : 1024 #optional ❿
       #rx_desc         : 1024 # optional ⓫
       #tx_desc         : 1024 # optional ⓬
       c                 : 4 #optional ⓭
       port_bandwidth_gb : 10 #optional ⓮
       port_info         :  # set eh mac addr  mandatory
            - default_gw : 1.1.1.1   # port 0 ⓯
              dest_mac   : '00:00:00:01:00:00' # Either default_gw or dest_mac is mandatory ↩
                 ⓰
              src_mac    : '00:00:00:02:00:00' # optional ⓱
              ip         : 2.2.2.2 # optional ⓲
              vlan       : 15 # optional ⓳
            - dest_mac   : '00:00:00:03:00:00'  # port 1
              src_mac    : '00:00:00:04:00:00'
            - dest_mac   : '00:00:00:05:00:00'  # port 2
              src_mac    : '00:00:00:06:00:00'
            - dest_mac   :   [0x0,0x0,0x0,0x7,0x0,0x01]  # port 3 ⓴
              src_mac    :   [0x0,0x0,0x0,0x8,0x0,0x02] # ㉑
```

❶ Number of ports. Should be equal to the number of interfaces listed in 3. - mandatory

❷ Must be set to 2. - mandatory

❸ List of interfaces to use. Run `sudo ./dpdk_setup_ports.py --show` to see the list you can choose from. - mandatory. there are cases that one PCI can have more than one port (MLX4 driver for example), for this you can use the format dd:dd.d/d for example 03:00.0/1, it means the second port of this device. The order of the list is important the first will the virtual port 0.

❻ Enable the ZMQ publisher for stats data, default is true.

❼ ZMQ port number. Default value is good. If running two TRex instances on the same machine, each should be given distinct number. Otherwise, can remove this line.

❾ If running two TRex instances on the same machine, each should be given distinct name. Otherwise, can remove this line. (Passed to DPDK as --file-prefix arg)

❿ Limit the amount of packet memory used. (Passed to dpdk as -m arg)

⓭ Number of threads (cores) TRex will use per interface pair (Can be overridden by -c command line option)

14 The bandwidth of each interface in Gbs. In this example we have 10Gbs interfaces. For VM, put 1. Used to tune the amount of memory allocated by TRex.

15, 16 TRex need to know the destination MAC address to use on each port. You can specify this in one of two ways:
Specify dest_mac directly.
Specify default_gw (since version 2.10). In this case (only if no dest_mac given), TRex will issue ARP request to this IP, and will use the result as dest MAC. If no dest_mac given, and no ARP response received, TRex will exit.

17 Source MAC to use when sending packets from this interface. If not given (since version 2.10), MAC address of the port will be used.

18 If given (since version 2.10), TRex will issue gratitues ARP for the ip + src MAC pair on appropriate port. In stateful mode, gratitues ARP for each ip will be sent every 120 seconds (Can be changed using --arp-refresh-period argument).

19 If given (since version 2.18), all traffic on the port will be sent with this VLAN tag.

20, 21 Old MAC address format. New format is supported since version v2.09.

8 Stateless ZMQ RPC port number. Default value is good. If running two TRex instances on the same machine, each should be given distinct number. Otherwise, can remove this line.

11, 12 Override the default number of Rx/TX dpdk driver descriptors. For better performance on virtual interfaces better to enlarge these numbers to Rx=4096

4 Advance DPDK options, used by Azure/failsafe driver

5 In case of failsafe choose the right drivers

Note
If you use version earlier than 2.10, or choose to omit the “ip” and have mac based configuration, be aware that TRex will not send any gratitues ARP and will not answer ARP requests. In this case, you must configure static ARP entries pointing to TRex port on your DUT. For an example config, you can look here [trex_config].

Note
Setups with virtual interfaces (SR-IOV/VMXNET3/virtio) has by default, only 512 ex descriptors per queue, better to enlarge it for multi-rx-queue/high performance setups

To find out which interfaces (NIC ports) can be used, perform the following:

```
[bash]>>sudo ./dpdk_setup_ports.py --show

 Network devices using DPDK-compatible driver
 ============================================

 Network devices using kernel driver
 ===================================
 0000:02:00.0 '82545EM Gigabit Ethernet Controller' if=eth2 drv=e1000 unused=igb_uio * ↩
     Active* #1
 0000:03:00.0 '82599ES 10-Gigabit SFI/SFP+ Network Connection' drv= unused=ixgb #2
 0000:03:00.1 '82599ES 10-Gigabit SFI/SFP+ Network Connection' drv= unused=ixgb
 0000:13:00.0 '82599ES 10-Gigabit SFI/SFP+ Network Connection' drv= unused=ixgb
 0000:13:00.1 '82599ES 10-Gigabit SFI/SFP+ Network Connection' drv= unused=ixgb

 Other network devices
 =====================
 <none>
```

1 We see that 02:00.0 is active (our management port).

2 All other NIC ports (03:00.0, 03:00.1, 13:00.0, 13:00.1) can be used.

minimum configuration file is:

```
<none>
- port_limit    : 4
  version       : 2
  interfaces    : ["03:00.0","03:00.1","13:00.1","13:00.0"]
```

6.2.2 Memory section configuration

The memory section is optional. It is used when there is a need to tune the amount of memory used by TRex packet manager. Default values (from the TRex source code), are usually good for most users. Unless you have some unusual needs, you can eliminate this section.

```
        - port_limit      : 2
          version       : 2
          interfaces    : ["03:00.0","03:00.1"]
          memory    :                                               1
             mbuf_64     : 16380                                    2
             mbuf_128    : 8190
             mbuf_256    : 8190
             mbuf_512    : 8190
             mbuf_1024   : 8190
             mbuf_2048   : 4096
             traffic_mbuf_64     : 16380                            3
             traffic_mbuf_128    : 8190
             traffic_mbuf_256    : 8190
             traffic_mbuf_512    : 8190
             traffic_mbuf_1024   : 8190
             traffic_mbuf_2048   : 4096
             dp_flows    : 1048576                                  4
             global_flows : 10240                                   5
```

1 Memory section header

2 Numbers of memory buffers allocated for packets in transit, per port pair. Numbers are specified per packet size.

3 Numbers of memory buffers allocated for holding the part of the packet which is remained unchanged per template. You should increase numbers here, only if you have very large amount of templates.

4 Number of TRex flow objects allocated (To get best performance they are allocated upfront, and not dynamically). If you expect more concurrent flows than the default (1048576), enlarge this.

5 Number objects TRex allocates for holding NAT “in transit” connections. In stateful mode, TRex learn NAT translation by looking at the address changes done by the DUT to the first packet of each flow. So, these are the number of flows for which TRex sent the first flow packet, but did not learn the translation yet. Again, default here (10240) should be good. Increase only if you use NAT and see issues.

6.2.3 Platform section configuration

The platform section is optional. It is used to tune the performance and allocate the cores to the right NUMA a configuration file now has the folowing struct to support multi instance

```
- version       : 2
  interfaces    : ["03:00.0","03:00.1"]
  port_limit    : 2
....
```

```
platform :                                           (1)
      master_thread_id  : 0                          (2)
      latency_thread_id : 5                          (3)
      dual_if   :                                    (4)
           - socket   : 0                            (5)
             threads  : [1,2,3,4]                    (6)
```

(1) Platform section header.

(2) Hardware thread_id for control thread.

(3) Hardware thread_id for RX thread.

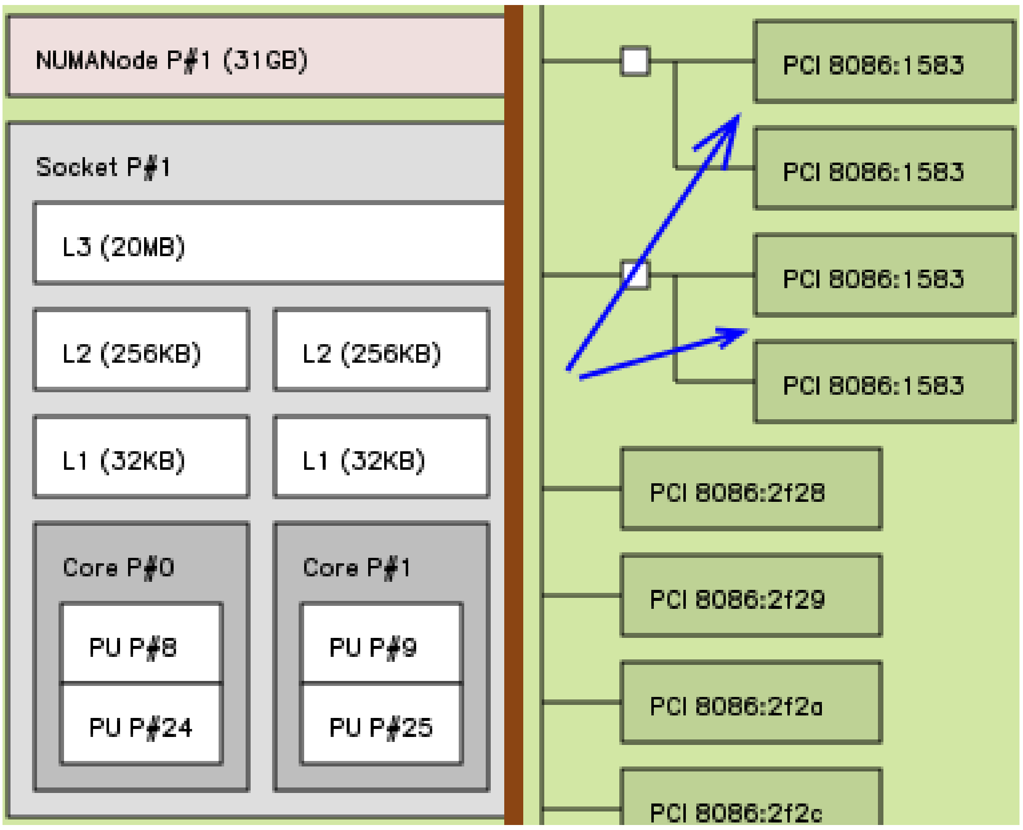
(4) “dual_if” section defines info for interface pairs (according to the order in “interfaces” list). each section, starting with “- socket” defines info for different interface pair.

(5) The NUMA node from which memory will be allocated for use by the interface pair.

(6) Hardware threads to be used for sending packets for the interface pair. Threads are pinned to cores, so specifying threads actually determines the hardware cores.

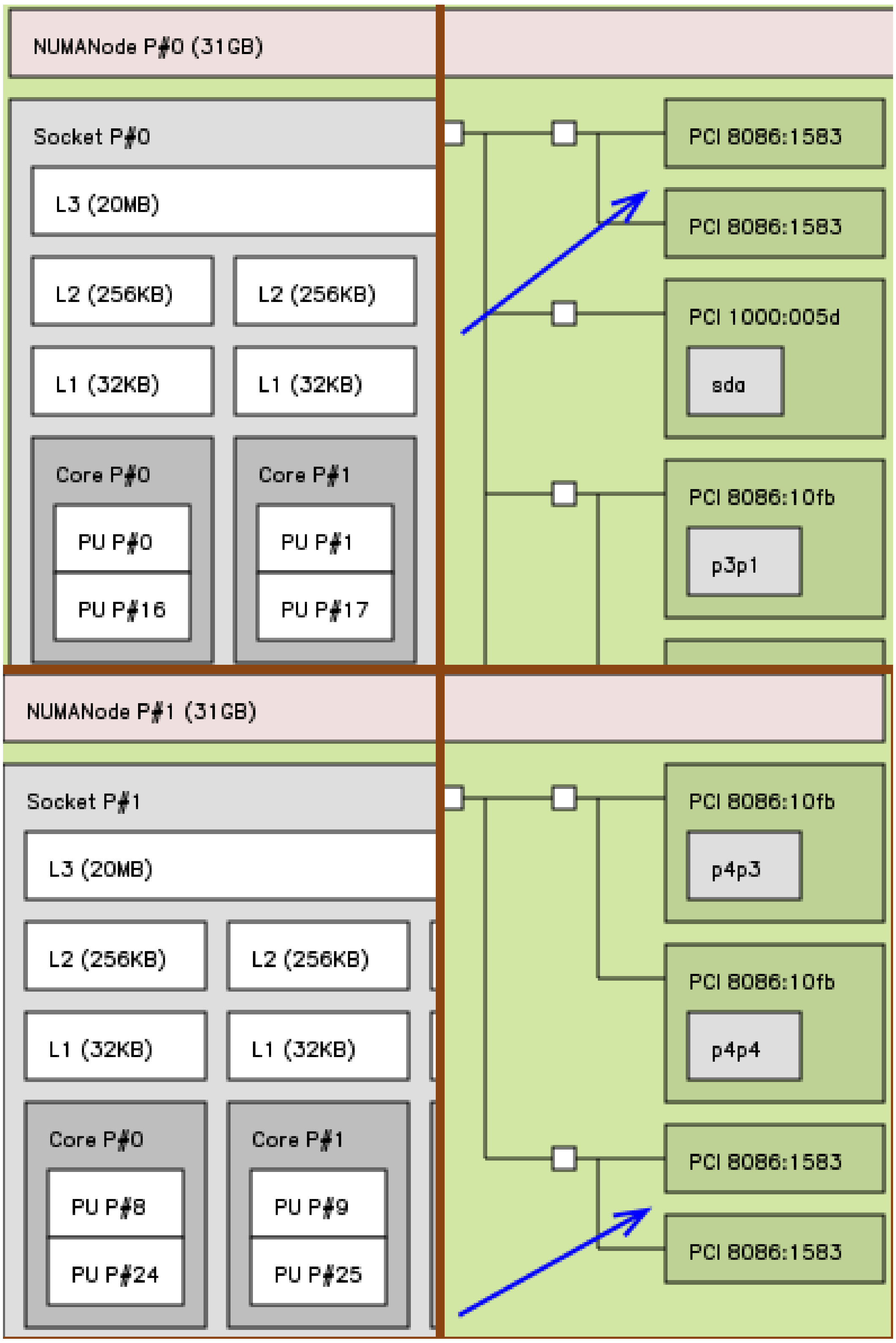
Real example:

We connected 2 Intel XL710 NICs close to each other on the motherboard. They shared the same NUMA:



CPU utilization was very high ~100%, with c=2 and c=4 the results were same.

Then, we moved the cards to different NUMAs:



+ We added configuration to the /etc/trex_cfg.yaml:

```
platform :
      master_thread_id  : 0
      latency_thread_id : 8
      dual_if   :
           - socket   : 0
             threads  : [1, 2, 3, 4, 5, 6, 7]
           - socket   : 1
             threads  : [9, 10, 11, 12, 13, 14, 15]
```

This gave best results: with **~98 Gb/s** TX BW and c=7, CPU utilization became **~21%**! (40% with c=4)

6.2.4 Timer Wheel section configuration

The flow scheduler uses a timer wheel to schedule flows. To tune it for a large number of flows it is possible to change the default values. This is an advanced configuration; don't use it if you don't know what you are doing. It can be configured in the global `trex_cfg.yaml` file and per trex traffic profile.

```
  tw :
     buckets : 1024                   (1)
     levels  : 3                      (2)
     bucket_time_usec : 20.0          (3)
```

(1) the number of buckets in each level, higher number will improve performance, but will reduce the maximum levels.

(2) how many levels.

(3) bucket time in usec. higher number will create more bursts

6.3 Command line options

--active-flows
An experimental switch to scale up or down the number of active flows. It is not accurate due to the quantization of flow scheduler and in some cases does not work. Example: --active-flows 500000 wil set the ballpark of the active flows to be ~0.5M.

--allow-coredump
Allow creation of core dump.

--arp-refresh-period <num>
Period in seconds between sending of gratuitous ARP for our addresses. Value of 0 means ``never send``.

--astf
Since version 2.29.
Flag that specifies advanced stateful mode. In this case, -f argument should be Python ASTF profile. Currently works as batch, WIP on adding interactive support.

-c <num>
Number of hardware threads to use per interface pair. Use at least 4 for TRex 40Gbs.
TRex uses 2 threads for inner needs. Rest of the threads can be used. Maximum number here, can be number of free threads divided by number of interface pairs.
For virtual NICs on VM, we always use one thread per interface pair.

--cfg <file name>
TRex configuration file to use. See relevant manual section for all config file options.

--checksum-offload-disable
Enable IP, TCP and UDP tx checksum offloading, using DPDK. This requires all used interfaces to support this.

--client_cfg <file>
YAML file describing clients configuration. Look here for details.

-d <num>
Duration of the test in seconds.

-e
: Same as `-p`, but change the src/dst IP according to the port. Using this, you will get all the packets of the same flow from the same port, and with the same src/dst IP.
It will not work good with NBAR as it expects all clients ip to be sent from same direction.

-f <yaml file>
: Specify traffic YAML configuration file to use. Mandatory option for stateful mode.

--hops <num>
: Provide number of hops in the setup (default is one hop). Relevant only if the Rx check is enabled. Look here for details.

-i
: Flag that specifies interactive mode. Currently used for stateless (WIP adding advanced stateful to it)

--iom <mode>
: I/O mode. Possible values: 0 (silent), 1 (normal), 2 (short).

--ipv6
: Convert templates to IPv6 mode.

-k <num>
: Run “warm up” traffic for num seconds before starting the test. This is needed if TRex is connected to switch running spanning tree. You want the switch to see traffic from all relevant source MAC addresses before starting to send real data. Works only with the latency test (-l option). Traffic sent is the same used for the latency test.
Current limitation (holds for TRex version 1.82): does not work properly on VM.

-l <rate>
: In parallel to the test, run latency check, sending packets at rate/sec from each interface.

--learn-mode <mode>
: Learn the dynamic NAT translation. Look here for details.

--learn-verify
: Used for testing the NAT learning mechanism. Do the learning as if DUT is doing NAT, but verify that packets are not actually changed.

--limit-ports <port num>
: Limit the number of ports used. Overrides the “port_limit” from config file.

--lm <hex bit mask>
: Mask specifying which ports will send traffic. For example, 0x1 - Only port 0 will send. 0x4 - only port 2 will send. This can be used to verify port connectivity. You can send packets from one port, and look at counters on the DUT.

--lo
: Latency only - Send only latency packets. Do not send packets from the templates/pcap files.

-m <num>
: Rate multiplier. TRex will multiply the CPS rate of each template by num.

--nc
: If set, will terminate exacly at the end of the specified duration. This provides faster, more accurate TRex termination. By default (without this option), TRex waits for all flows to terminate gracefully. In case of a very long flow, termination might prolong.

--no-flow-control-change
: Since version 2.21.
Prevents TRex from changing flow control. By default (without this option), TRex disables flow control at startup for all cards, except for the Intel XL710 40G card.

--no-hw-flow-stat
: Relevant only for Intel x710 stateless mode. Do not use HW counters for flow stats.
Enabling this will support lower traffic rate, but will also report RX byte count statistics.

--no-key
: Daemon mode, don’t get input from keyboard.

--no-watchdog
: Disable watchdog.

-p
: Send all packets of the same flow from the same direction. For each flow, TRex will randomly choose between client port and server port, and send all the packets from this port. src/dst IPs keep their values as if packets are sent from two ports. Meaning, we get on the same port packets from client to server, and from server to client.
If you are using this with a router, you can not relay on routing rules to pass traffic to TRex, you must configure policy based routes to pass all traffic from one DUT port to the other.

-pm <num>
: Platform factor. If the setup includes splitter, you can multiply all statistic number displayed by TRex by this factor, so that they will match the DUT counters.

-pubd
: Disable ZMQ monitor’s publishers.

--queue-drop
: Since version 2.37.
Do not retry to send packets on failure (queue full etc.).

--rx-check <sample rate>
: Enable Rx check module. Using this, each thread randomly samples 1/sample_rate of the flows and checks packet order, latency, and additional statistics for the sampled flows. Note: This feature works on the RX thread.

--sleeps
: Since version 2.37.
Use sleeps instead of busy wait in scheduler (less accurate, more power saving)

--software
: Since version 2.21.
Do not configure any hardware rules. In this mode, all RX packets will be processed by software. No HW assist for dropping (while counting) packets will be used. This mode is good for enabling features like per stream statistics, and latency, support packet types, not supported by HW flow director rules (For example QinQ).
You can also use this mode for running TRex on interfaces which manifest themselves as ones supported by TRex, but in reality support less hardware capabilities. For example, NICs supported by DPDK e1000_igb driver, but with different HW capabilities than i350.
Drawback of this is that because software has to handle all received packets, total rate of RX streams is significantly lower.
Up until **v2.49** (not include) this mode is also limited to using only one TX core (and one RX core as usual). Now it uses multi-queue for RX and TX mainly for scale of virtual interfaces.

-v <verbosity level>
: Show debug info. Value of 1 shows debug info on startup. Value of 3, shows debug info during run at some cases. Might slow down operation.

--vlan
: Relevant only for stateless mode with Intel 82599 10G NIC. When configuring flow stat and latency per stream rules, assume all streams uses VLAN.

-w <num seconds>
: Wait additional time between NICs initialization and sending traffic. Can be useful if DUT needs extra setup time. Default is 1 second.

Chapter 7

Appendix