TRex Advance stateful support

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Audience

This document assumes basic knowledge of TRex, and assumes that TRex is installed and configured. For information, see the manual especially the material up to the Basic Usage section and stateless for better understanding the interactive model. Consider this document as an extension to the manual, it might be integrated in the future.

Chapter 2

Advance Stateful support

2.1 Feature overview

TRex supports Stateless (STL) and Stateful (STF) modes.

This document describes the new Advance Stateful mode (ASTF) that supports TCP layer.

The following UDP/TCP related use-cases will be addressed by ASTF mode.

- Ability to work when the DUT terminates the TCP stack (e.g. compress/uncompress, see figure 1). In this case there is a different TCP session on each side, but L7 data are **almost** the same.

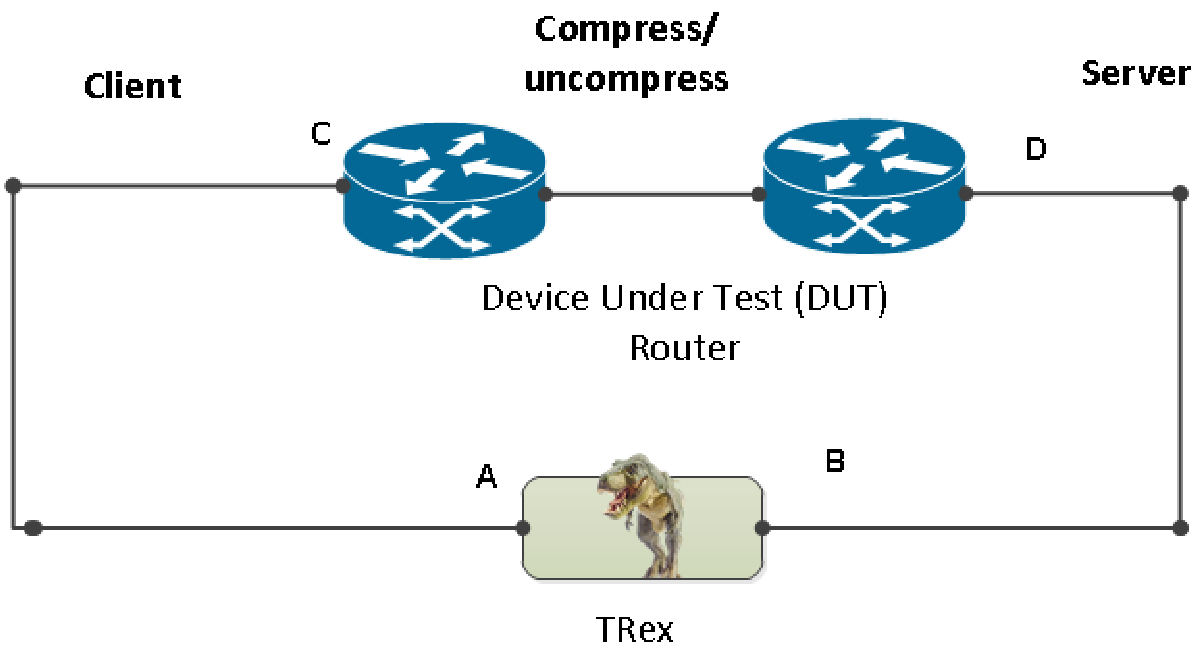


Figure 2.1: DUT is TCP proxy

- Ability to work in either client mode or server mode. This way TRex client side could be installed in one physical location on the network and TRex server in another. figure 2 shows such an example

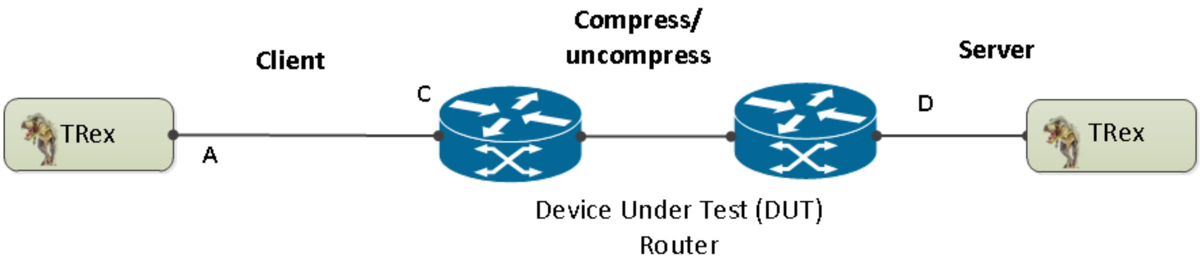


Figure 2.2: C/S mode

- Performance and scale
  - High bandwidth - ~200gb/sec with many realistic flows (not one elephant flow)
  - High connection rate - order of MCPS
  - Scale to millions of active established flows
- Simulate latency/jitter/drop in high rate
- Emulate L7 application, e.g. HTTP/HTTPS/Citrix- there is no need to implement the exact application.
- Simulate L7 application on top of TLS using OpenSSL
- BSD baseTCP implementation
- Ability to change fields in the L7 stream application - for example, change HTTP User-Agent field
- Interactive support - Fast Console, GUI
- TCP/UDP/Application statistics (per client side/per template)
- Verify incoming IP/TCP/UDP checksum
- Python 2.7/3.0 Client API
- Ability to build a realistic traffic profile that includes TCP and UDP protocols (e.g. SFR EMIX)
- IPv6/IPv4
- Fragmentation support
- Accurate latency for TCP flows - SYN/SYN ACK and REQ/RES latency histogram, usec resolution

2.1.1 Status

Warning
ASTF support was released, however it is under constant improvement.

2.1.1.1 What works in the current version

- Profile with multi templates of TCP/UDP
- IPv4 and IPv6
- VLAN configuration
- Enable client only or server only or both
- High scale with flows/BW/PPS
- Ability to change IPv4/IPv6 configuration like default TOS etc
- Flexible tuple generator
- Automation support - fast interactive support, Fast Console
- Ability to change the TCP configuration (default MSS/buffer size/RFC enabled etc)
- Client Cluster (same format as STF for batch, new Python format for interactive)
- Basic L7 emulation capability e.g. Random delay, loops, variables, Spirent and IXIA like TCP traffic patterns, Elephant flows
- Tunable profile support — give a few tunable from console to the python profile (e.g. --total-bw 10gbps)
- More than one core per dual-ports
- Ability to use all ports as clients or server
- TCP statistics per template

2.1.1.2 On radar

- TLS support
- IPv6 traffic is assumed to be generated by TRex itself (only the 32bit LSB is taken as a key)
- Simulation of Jitter/Latency/drop
- Field Engine support - ability to change a field inside the stream
- Accurate latency for TCP session. Measure sample of the flows in EF/low latency queue. Measure the SYN=SYN-ACK and REQ-RES latency histogram
- Fragmentation is not supported
- Advanced L7 emulation capability
  - Add to send command the ability to signal in the middle of queue size (today is always at the end)
  - Change TCP/UDP stream fields (e.g. user Agent)
  - Protocols specific commands (e.g. wait_for_http() will parse the header and wait for the size)
  - Commands for l7 dynamic counters (e.g. wait_for_http() will register dynamic counters)

2.1.2 Can we address the above requirements using existing DPDK TCP stacks?

Can we leverage one of existing DPDK TCP stacks for our need? The short answer is no.
We chose to take a BSD4.4 original code base with FreeBSD bug fixes patches and improve the scalability to address our needs. More on the reasons why in the following sections, but let me just say the above TCP DPDK stacks are optimized for real client/server application/API while in most of our traffic generation use cases, **most** of the traffic is known ahead of time allowing us to do much better.
Let's take a look into what are the main properties of TRex TCP module and understand what were the main challenges we tried to solve.

2.1.3 The main properties of scalable TCP for traffic generation

- Interact with DPDK API for batching of packets
- Multi-instance - lock free. Each thread will get its own TCP context with local counters/configuration, flow-table etc ,RSS
- Async, Event driven - No OS API/threads needed
  - Start write buffer
  - Continue write
  - End Write
  - Read buffer /timeout
  - OnConnect/OnReset/OnClose
- Accurate with respect to TCP RFCs - at least derive from BSD to be compatible - no need to reinvent the wheel
- Enhanced tcp statistics - as a traffic generator we need to gather as many statistics as we can, for example per template tcp statistics.
- Ability to save descriptors for better simulation of latency/jitter/drop

The folowing figure shows the block diagram of new TRex TCP design

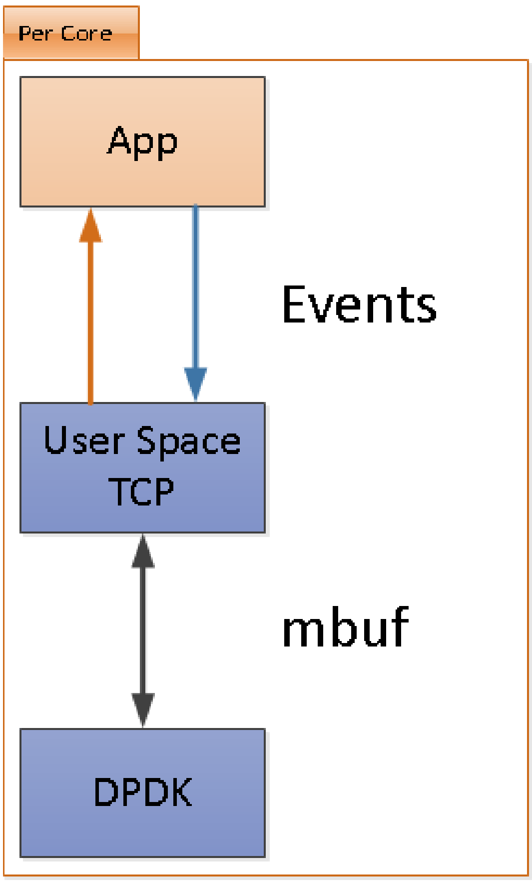


Figure 2.3: Stack

And now lets proceed to our challenges, let me just repeat the objective of TRex, it is not to reach a high rate with one flow, it is to simulate a realistic network with many clients using small flows. Let's try to see if we can solve the scale of million of flows.

2.1.4 Tx Scale to millions of flows

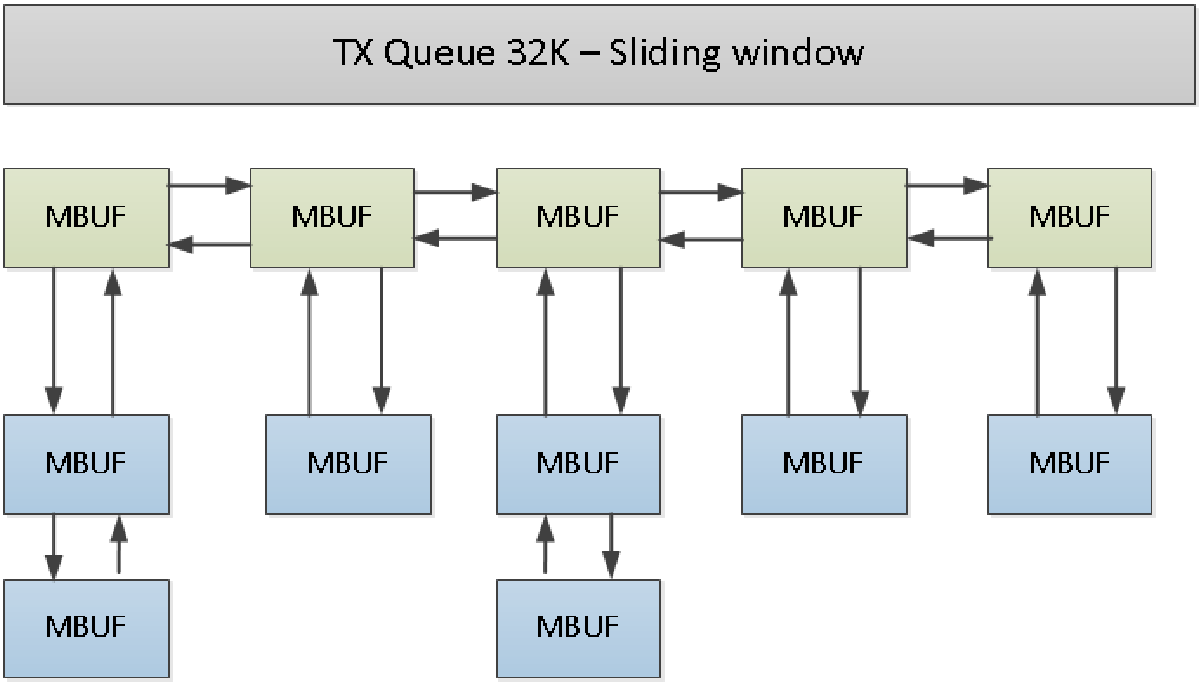


Figure 2.4: TCP Tx side

Most TCP stacks have an API that allow the user to provide his buffer for write (push) and the TCP module will save them until the packets are acknowledged by the remote side. Figure 4 shows how one TX queue of one TCP flow looks like on the Tx side. This could create a scale issue in worst case. Let's assume we need 1M active flows with 64K TX buffer (with reasonable buffer, let's say RTT is small). The worst case buffer in this case could be 1M x 64K * mbuf-factor (let's assume 2) = 128GB. The mbuf resource is expensive and needs to be allocated ahead of time. the solution we chose for this problem (which from a traffic generator's point of view) is to change the API to be a poll API, meaning TCP will request the buffers from the application layer only when packets need to be sent (lazy). Now because most of the traffic is constant in our case, we could save a lot of memory and have an unlimited scale (both of flows and tx window).

Note
This optimization won't work with TLS since constant sessions will have new data

---

2.1.5 Rx Scale to millions of flows

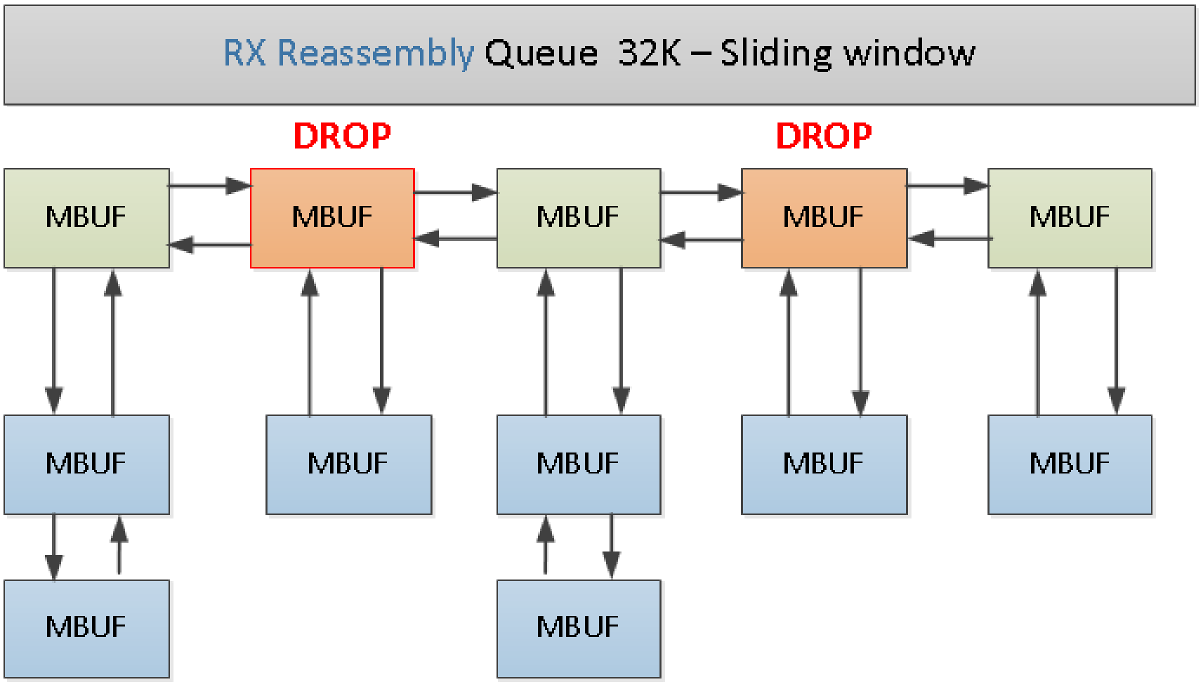


Figure 2.5: Example of multiple streams

The same problem exists in the case of reassembly in the rx side, in worst case there is a need to store a lot of memory in reassembly queue. To fix this we can add a filter API for the application layer. Let's assume that the application layer can request only a partial portion of the data since the rest is less important, for example data in offset of 61K-64K and only in case of restransmission (simulation). In this case we can give the application layer only the filtered data that is really important to it and still allow TCP layer to work in the same way from seq/ack perspective.

Note
This optimization won't work with TLS since constant sessions will have new data

---

2.1.6 Simulation of latency/jitter/drop in high scale

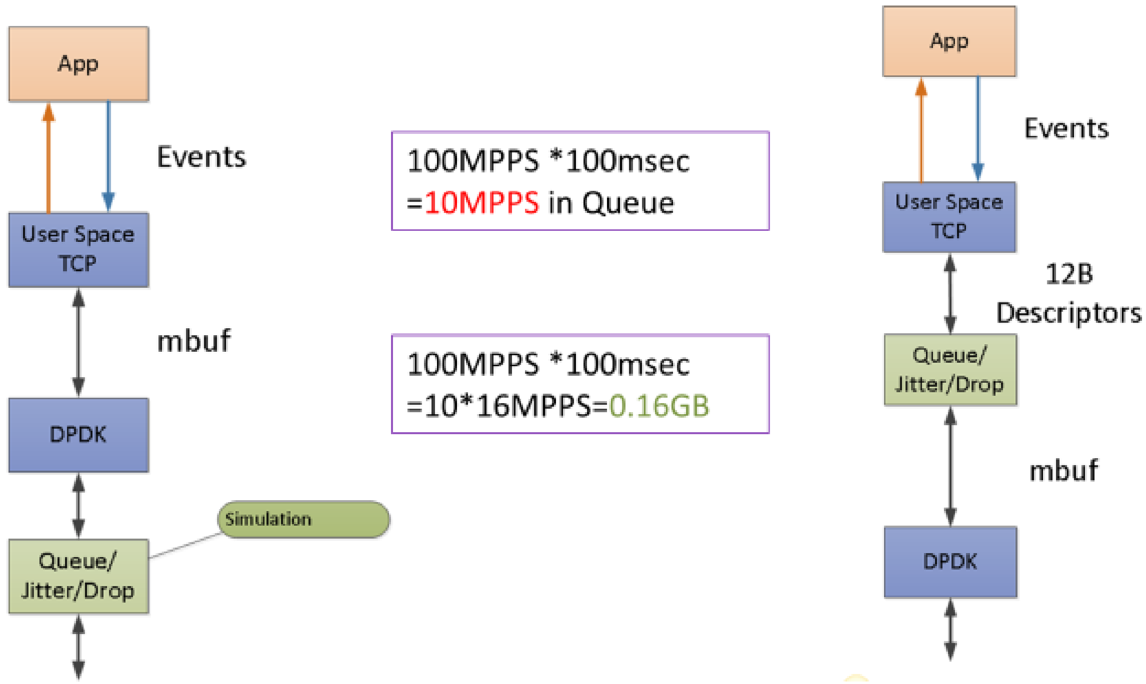


Figure 2.6: TCP Rx side

There is a requirement to simulate latency/jitter/drop in the network layer. Simulating drop in high rate it is not a problem, but simulating latency/jitter in high rate is a challenge because there is a need to queue a high number of packets. See figure 6 on the left. A better solution is to queue a pointer to both the TCP flow and the TCP descriptor (with TSO information) and only when needed (i.e. when it has already left the tx queue) build the packet again (lazy). The memory footprint in this case can be reduced dramatically.

2.1.7 Emulation of L7 application

To emulate L7 application on top of the TCP layer we can define a set of simple operations. The user would be able to build an application emulation layer from Python API or by a utility that we will provide that will analyze a pcap file and convert it to TCP operations. Another thing that we can learn from pcap is the TCP parameters like MSS/Window size/Nagel/TCP options etc.. Let's give a simple example of a L7 emulation of HTTP Client and HTTP Server:

HTTP Client

```
send(request,len=100)
wait_for_response(len<=1000)
delay(random(100-1000)*usec)
send(request2,len=200)
wait_for_response(len<=2000)
close()
```

HTTP Server

```
wait_for_request(len<=100)
send_response(data,len=1000)
wait_for_request(len<=200)
send_response(data,len=2000)
close()
```

This way both Client and Server don't need to know the exact application protocol, they just need to have the same story/program. In real HTTP server, the server parses the HTTP requeset, learns the Content-Length field, waits for the rest of the data and finally retrieves the information from disk. With our L7 emulation there is no need. Even in cases where the data length is changed (for example NAT/LB that changes the data length) we can give some flexibility within the program on the value range of the length In case of UDP it is a message base protocols like send_msg/wait_for_msg etc.

2.1.8 Stateful(STF) vs Advance Stateful (ASTF)

- Same Flexible tuple generator
- Same Clustering mode
- Same VLAN support
- NAT - no need for complex learn mode. ASTF supports NAT64 out of the box.
- Flow order. ASTF has inherent ordering verification using the TCP layer. It also checks IP/TCP/UDP checksum out of the box.
- Latency measurement is supported in both.
- In ASTF mode, you can't control the IPG, less predictable (concurrent flows is less deterministic)
- ASTF can be interactive (start, stop, stats)

2.1.9 GPRS Tunnelling Protocol (GTP)

- Supports Encapsulation(adding GTP header)/Decapsulation(removing GTP header) of GTP traffic
- Port which is GTP enabled will perform Encapsulation/Decapsulation functionality (client side)
- TRex server should be started with option --gtpu <port number> (should be zero)
- Limitation: Can use only 2 ports and only port 0 (client side) can be configured as GTPU tunnel

2.2 ASTF package folders

Location	Description
/astf	ASTF native (py) profiles
/automation/trex_control_plane/interactive/trex/examples/astf	automation examples
/automation/trex_control_plane/interactive/trex/astf	ASTF lib compiler (convert py to JSON), Interactive lib
/automation/trex_control_plane/stf	STF automation (used by ASTF mode)

2.3 Getting started Tutorials

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

2.3.1 Tutorial: Prepare TRex configuration file

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

Follow this chapter first time configuration

2.3.2 Tutorial: Run TRex with simple HTTP profile

Goal
Send a simple HTTP flows

Traffic profile
The following profile defines one template of HTTP

File
astf/http_simple.py

```
from trex.astf.api import *


class Prof1():
    def get_profile(self):
        # ip generator
        ip_gen_c = ASTFIPGenDist(ip_range=["16.0.0.0", "16.0.0.255"],
                                 distribution="seq")
        ip_gen_s = ASTFIPGenDist(ip_range=["48.0.0.0", "48.0.255.255"],
                                  distribution="seq")
        ip_gen = ASTFIPGen(glob=ASTFIPGenGlobal(ip_offset="1.0.0.0"),        ❶
                           dist_client=ip_gen_c,
                           dist_server=ip_gen_s)

        return ASTFProfile(default_ip_gen=ip_gen,
                            cap_list=[ASTFCapInfo(
                                      file="../avl/delay_10_http_browsing_0.pcap"
                                      cps=1)
                                     ])                                        ❷


def register():
    return Prof1()
```

❶ Define the tuple generator range for client side and server side

❷ The template list with relative CPS (connection per second)

Running TRex with this profile interactive (v2.47 and up)
.Start ASTF in interactive mode

Start ASTF in interactive mode

```
sudo ./t-rex-64 -i --astf
```

Console

```
./trex-console -s [server-ip]
```

From Console

```
trex>start -f astf/http_simple.py -m 1000 -d 1000 -l 1000
trex>tui
trex>[press] t/l for astf statistics and latency
trex>stop
```

Cosole ASTF stats

```
                     |     client  |         server |
--------------------------------------------------------------------------------
  m_active_flows   |       39965  |          39966  |  active flows
     m_est_flows   |       39950  |          39952  |  active est flows
    m_tx_bw_l7_r   | 31.14 Mbps  |     4.09 Gbps  |  tx bw
    m_rx_bw_l7_r   |  4.09 Gbps  |    31.14 Mbps  |  rx bw
      m_tx_pps_r   |140.36 Kpps  |   124.82 Kpps  |  tx pps
      m_rx_pps_r   |156.05 Kpps  |   155.87 Kpps  |  rx pps
      m_avg_size   |    1.74 KB  |       1.84 KB  |  average pkt size
               -   |        ---  |           ---  |
             TCP   |        ---  |           ---  |
               -   |        ---  |           ---  |
tcps_connattempt   |       73936  |              0  |  connections initiated
    tcps_accepts   |           0  |          73924  |  connections accepted
   tcps_connects   |       73921  |          73910  |  connections established
     tcps_closed   |       33971  |          33958  |  conn. closed (includes drops)
  tcps_segstimed   |      213451  |         558085  |  segs where we tried to get rtt
 tcps_rttupdated   |      213416  |         549736  |  times we succeeded
     tcps_delack   |      344742  |              0  |  delayed acks sent
   tcps_sndtotal   |      623780  |         558085  |  total packets sent
    tcps_sndpack   |       73921  |         418569  |  data packets sent
    tcps_sndbyte   |    18406329  |     2270136936  |  data bytes sent
    tcps_sndctrl   |       73936  |              0  |  control (SYN,FIN,RST) packets sent
    tcps_sndacks   |      475923  |         139516  |  ack-only packets sent
    tcps_rcvpack   |      550465  |         139502  |  packets received in sequence
    tcps_rcvbyte   |  2269941776  |       18403590  |  bytes received in sequence
 tcps_rcvackpack   |      139495  |         549736  |  rcvd ack packets
 tcps_rcvackbyte   |    18468679  |     2222057965  |  tx bytes acked by rcvd acks
    tcps_preddat   |      410970  |              0  |  times hdr predict ok for data pkts
  tcps_rcvoopack   |           0  |              0  |  *out-of-order packets received    #1
               -   |        ---  |           ---  |
      Flow Table   |        ---  |           ---  |
               -   |        ---  |           ---  |
  redirect_rx_ok   |           0  |              1  |  redirect to rx OK
```

1. Counters with asterisk prefix (*) means that there is some kind of error, see counters description for more information

Running TRex with this profile in batch mode (planned to be deprecated)

```
[bash]>sudo ./t-rex-64 -f astf/http_simple.py -m 1000 -d 1000 -c 1 --astf -l 1000 -k 10
```

- `--astf` is mandatory to enable ASTF mode
- (Optional) Use `-c` to 1, in this version it is limited to 1 core for each dual interfaces
- (Optional) Use `--cfg` to specify a different configuration file. The default is /etc/trex_cfg.yaml.

pressing '**t**' while traffic is running you can see the TCP JSON counters as table

```
                     |     client  |         server |
--------------------------------------------------------------------------------
  m_active_flows   |       39965  |          39966  |  active flows
     m_est_flows   |       39950  |          39952  |  active est flows
    m_tx_bw_l7_r   | 31.14 Mbps  |     4.09 Gbps  |  tx bw
    m_rx_bw_l7_r   |  4.09 Gbps  |    31.14 Mbps  |  rx bw
      m_tx_pps_r   |140.36 Kpps  |   124.82 Kpps  |  tx pps
      m_rx_pps_r   |156.05 Kpps  |   155.87 Kpps  |  rx pps
      m_avg_size   |    1.74 KB  |       1.84 KB  |  average pkt size
```

```
                -  |         ---  |         ---  |
              TCP  |         ---  |         ---  |
                -  |         ---  |         ---  |
 tcps_connattempt  |       73936  |           0  |  connections initiated
     tcps_accepts  |           0  |       73924  |  connections accepted
    tcps_connects  |       73921  |       73910  |  connections established
      tcps_closed  |       33971  |       33958  |  conn. closed (includes drops)
   tcps_segstimed  |      213451  |      558085  |  segs where we tried to get rtt
  tcps_rttupdated  |      213416  |      549736  |  times we succeeded
      tcps_delack  |      344742  |           0  |  delayed acks sent
    tcps_sndtotal  |      623780  |      558085  |  total packets sent
     tcps_sndpack  |       73921  |      418569  |  data packets sent
     tcps_sndbyte  |    18406329  |  2270136936  |  data bytes sent
     tcps_sndctrl  |       73936  |           0  |  control (SYN,FIN,RST) packets sent
    tcps_sndacks   |      475923  |      139516  |  ack-only packets sent
     tcps_rcvpack  |      550465  |      139502  |  packets received in sequence
     tcps_rcvbyte  |  2269941776  |    18403590  |  bytes received in sequence
  tcps_rcvackpack  |      139495  |      549736  |  rcvd ack packets
  tcps_rcvackbyte  |    18468679  |  2222057965  |  tx bytes acked by rcvd acks
     tcps_preddat  |      410970  |           0  |  times hdr predict ok for data pkts
   tcps_rcvoopack  |           0  |           0  |  *out-of-order packets received   #1
                -  |         ---  |         ---  |
       Flow Table  |         ---  |         ---  |
                -  |         ---  |         ---  |
   redirect_rx_ok  |           0  |           1  |  redirect to rx OK
```

1. Counters with asterisk prefix (*) means that there is some kind of error, see counters description for more information

Discussion

When a template with pcap file is specified, like in this example the python code analyzes the L7 data of the pcap file and TCP configuration and build a JSON that represent

- The client side application
- The server side application (opposite from client)
- TCP configuration for each side

Client side pseudo code

```
template = choose_template()                                         (1)

src_ip,dest_ip,src_port = generate from pool of client
dst_port                = template.get_dest_port()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)                (2)

s.connect(dest_ip,dst_port)                                          (3)

# program                                                            (4)
s.write(template.request)      #write the following taken from the pcap file

                               # GET /3384 HTTP/1.1
                               # Host: 22.0.0.3
                               # Connection: Keep-Alive
                               # User-Agent: Mozilla/4.0
                               # Accept: */*
                               # Accept-Language: en-us
                               # Accept-Encoding: gzip, deflate, compress


s.read(template.request_size)  # wait for 32K bytes and compare some of it
```

```
                                #HTTP/1.1 200 OK
                                #Server: Microsoft-IIS/6.0
                                #Content-Type: text/html
                                #Content-Length: 32000
                                # body ..


s.close();
```

1. Tuple-generator is used to generate tuple for client and server side and choose a template
2. Flow is created
3. Connect to the server
4. Run the program base on JSON (in this example created from the pcap file)

Server side pseudo code

```
# if this is SYN for flow that already exist, let TCP handle it

if ( flow_table.lookup(pkt) == False ) :
    # first SYN in the right direction with no flow
    compare (pkt.src_ip/dst_ip to the generator ranges) # check that it is in the range or ↩
        valid server IP (src_ip,dest_ip)
    template= lookup_template(pkt.dest_port) #get template for the dest_port

    # create a socket for TCP server
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)                  (1)

    # bind to the port
    s.bind(pkt.dst_ip, pkt.dst_port)

    s.listen(1)

    #program of the template                                               (2)
    s.read(template.request_size)       # just wait for x bytes, don't check them

                                        # GET /3384 HTTP/1.1
                                        # Host: 22.0.0.3
                                        # Connection: Keep-Alive
                                        # User-Agent: Mozilla/4.0 ..
                                        # Accept: */*
                                        # Accept-Language: en-us
                                        # Accept-Encoding: gzip, deflate, compress


    s.write(template.response)         # just wait for x bytes,
                                       # don't check them (TCP check the seq and checksum)

                                       #HTTP/1.1 200 OK
                                       #Server: Microsoft-IIS/6.0
                                       #Content-Type: text/html
                                       #Content-Length: 32000
                                       # body ..

    s.close()
```

1. As you can see from the pseudo code there is no need to open all the servers ahead of time, we open and allocate socket only when packet match the criteria of server side

2 The program is the opposite of the client side.

The above is just a pseudo code that was created to explain how logically TRex works. It was simpler to show a pseudo code that runs in one thread in blocking fashion, but in practice it is run in an event driven and many flows can multiplexed in high performance and scale. The L7 program can be written using Python API (it is compiled to micro-code event driven by TRex server).

2.3.3 Tutorial: Profile with two templates

Goal
Simple browsing, HTTP and HTTPS flow. In this example, each template has different destination port (80/443)

Traffic profile
The profile include HTTP and HTTPS profile. Each second there would be 2 HTTPS flows and 1 HTTP flow.

File
astf/http_https.py

```
class Prof1():
    def get_profile(self):
        # ip generator
        ip_gen_c = ASTFIPGenDist(ip_range=["16.0.0.0", "16.0.0.255"],
                                 distribution="seq")
        ip_gen_s = ASTFIPGenDist(ip_range=["48.0.0.0", "48.0.255.255"],
                                 distribution="seq")
        ip_gen = ASTFIPGen(glob=ASTFIPGenGlobal(ip_offset="1.0.0.0"),
                           dist_client=ip_gen_c,
                           dist_server=ip_gen_s)

        return ASTFProfile(default_ip_gen=ip_gen,
                            cap_list=[
                             ASTFCapInfo(file="../avl/delay_10_http_browsing_0.pcap",
                                         cps=1),      1
                             ASTFCapInfo(file="avl/delay_10_https_0.pcap",
                                         cps=2)       2
                                      ])


def register():
    return Prof1()
```

1 HTTP template

2 HTTPS template

Discussion
The server side chooses the template base on the **destination** port. Because each template has a unique destination port (80/443) there is nothing to do. In the next example we will show what to do in case both templates has the same destination port. From the client side, the scheduler will schedule in each second 2 HTTPS flows and 1 HTTP flow base on the CPS

Note
In the real file cps=1 in both profiles.

2.3.4 Tutorial: Profile with two templates same ports

Goal
Create profile with two HTTP templates. In this example, both templates have the same destination port (80)

Traffic profile
The profile includes same HTTP profile only for demonstration.

File

```
class Prof1():
    def get_profile(self):
        # ip generator
        ip_gen_c = ASTFIPGenDist(ip_range=["16.0.0.0", "16.0.0.255"],
                                 distribution="seq")
        ip_gen_s = ASTFIPGenDist(ip_range=["48.0.0.0", "48.0.255.255"],
                                  distribution="seq")
        ip_gen = ASTFIPGen(glob=ASTFIPGenGlobal(ip_offset="1.0.0.0"),
                           dist_client=ip_gen_c,
                           dist_server=ip_gen_s)

        return ASTFProfile(default_ip_gen=ip_gen,
                            cap_list=[ASTFCapInfo(file="../avl/delay_10_http_browsing_0. ↩
                                pcap",
                                      cps=1),             ❶
                                      ASTFCapInfo(file="../avl/delay_10_http_browsing_0. ↩
                                          pcap",
                                      cps=2,port=8080)  ❷
                                     ])


def register():
    return Prof1()
```

❶ HTTP template

❷ HTTP template override the pcap file destination port

Discussion
In the real world, the same server can handle many types of transactions on the same port based on the request. In this TRex version we have this limitation as it is only an emulation. Next, we would add a better engine that could associate the template based on server IP-port socket or by L7 data

2.3.5 Tutorial: Profile with two range of tuple generator

Goal
Create a profile with two sets of client/server tuple pools.

Traffic profile
The profile includes the same HTTP template for demonstration.

File
astf/http_simple_different_ip_range.py

```
class Prof1():
    def __init__(self):
        pass  # tunables

    def create_profile(self):
        ip_gen_c1 = ASTFIPGenDist(ip_range=["16.0.0.1", "16.0.0.255"],
                                  distribution="seq")        ❶
        ip_gen_s1 = ASTFIPGenDist(ip_range=["48.0.0.1", "48.0.255.255"],
                                  distribution="seq")
        ip_gen1 = ASTFIPGen(glob=ASTFIPGenGlobal(ip_offset="1.0.0.0"),
                            dist_client=ip_gen_c1,
                            dist_server=ip_gen_s1)

        ip_gen_c2 = ASTFIPGenDist(ip_range=["10.0.0.1", "10.0.0.255"],
                                  distribution="seq")        ❷
        ip_gen_s2 = ASTFIPGenDist(ip_range=["20.0.0.1", "20.255.255"],
                                  distribution="seq")
        ip_gen2 = ASTFIPGen(glob=ASTFIPGenGlobal(ip_offset="1.0.0.0"),
                            dist_client=ip_gen_c2,
                            dist_server=ip_gen_s2)

        profile = ASTFProfile(cap_list=[
            ASTFCapInfo(file="../cap2/http_get.pcap",
                        ip_gen=ip_gen1),                     ❸
            ASTFCapInfo(file="../cap2/http_get.pcap",
                        ip_gen=ip_gen2, port=8080)           ❹
            ])

        return profile
```

1. Define generator range 1
2. Define generator range 2
3. Assign generator range 1 to the first template
4. Assign generator range 2 to the second template

Discussion
The tuple generator ranges should not overlap.

2.3.6 Tutorial: IPv6 traffic

ASTF can run a profile that includes a mix of IPv4 and IPv6 template using ipv6. enables tunable. The tunable could be global (for all profile) or per template. However, for backward compatibility, a CLI flag (in start) can convert all the profile to ipv6 automatically

Interactive

```
trex>start -f astf/http_simple.py -m 1000 -d 1000 --astf -l 1000 --ipv6
```

Batch

```
[bash]>sudo ./t-rex-64 -f astf/http_simple.py -m 1000 -d 1000 -c 1 --astf -l 1000 --ipv6
```

```
::x.x.x.x where LSB is IPv4 addrees
```

The profile includes same HTTP template for demonstration.

File
astf/param_ipv6.py

Another way to enable IPv6 globally (or per template) is by tunables in the profile file

IPv6 tunable (global)

```
class Prof1():
    def get_profile(self):

        # ip generator
        ip_gen_c = ASTFIPGenDist(ip_range=["16.0.0.0", "16.0.0.255"], distribution="seq")
        ip_gen_s = ASTFIPGenDist(ip_range=["48.0.0.0", "48.0.255.255"], distribution="seq")
        ip_gen = ASTFIPGen(glob=ASTFIPGenGlobal(ip_offset="1.0.0.0"),
                           dist_client=ip_gen_c,
                           dist_server=ip_gen_s)

        c_glob_info = ASTFGlobalInfo()
        # Enable IPV6 for client side and set the default SRC/DST IPv6 MSB
        # LSB will be taken from ip generator
        c_glob_info.ipv6.src_msb ="ff02::"          ❶
        c_glob_info.ipv6.dst_msb ="ff03::"          ❷
        c_glob_info.ipv6.enable  =1                 ❸


        return ASTFProfile(default_ip_gen=ip_gen,
                           #  Defaults affects all files
                           default_c_glob_info=c_glob_info,
                           cap_list=[
                                     ASTFCapInfo(file="../avl/delay_10_http_browsing_0.pcap ↩
                                         ",
                                     cps=1)
                                     ]
                           )
```

❶ Set default for source IPv6 addr (32bit LSB will be set by IPv4 tuple generator)

❷ Set default for destination IPv6 addr (32bit LSB will be set by IPv4 tuple generator)

❸ Enable ipv6 for all templates

In this case there is **no** need for `--ipv6` in CLI

```
trex>start -f astf/param_ipv6.py -m 1000 -d 1000 -l 1000
```

2.3.7 Tutorial: Change tcp.mss using tunables mechanism

Profile tunable is a mechanism to tune the behavior of ASTF traffic profile. TCP layer has a set of tunables. IPv6 and IPv4 have another set of tunables.

There are two types of tunables:

- Global tunable: per client/server will affect all the templates in specific side.
- Per-template tunable: will affect only the associated template (per client/server). Will have higher priority relative to global tunable.

By default, the TRex server has a default value for all the tunables and only when you set a specific tunable the server will override the value. Example of a tunable is tcp.mss. You can change the tcp.mss:

- Per all client side templates
- Per all server side templates
- For a specific template per client side
- For a specific template per server side

Example global client/server tcp.mss tunable

```
class Prof1():
    def get_profile(self):

        # ip generator
        ip_gen_c = ASTFIPGenDist(ip_range=["16.0.0.0", "16.0.0.255"], distribution="seq")
        ip_gen_s = ASTFIPGenDist(ip_range=["48.0.0.0", "48.0.255.255"], distribution="seq")
        ip_gen = ASTFIPGen(glob=ASTFIPGenGlobal(ip_offset="1.0.0.0"),
                           dist_client=ip_gen_c,
                           dist_server=ip_gen_s)

        c_glob_info = ASTFGlobalInfo()
        c_glob_info.tcp.mss = 1400                    ❶
        c_glob_info.tcp.initwnd = 1

        s_glob_info = ASTFGlobalInfo()
        s_glob_info.tcp.mss = 1400                    ❷
        s_glob_info.tcp.initwnd = 1


        return ASTFProfile(default_ip_gen=ip_gen,
                           #  Defaults affects all files
                           default_c_glob_info=c_glob_info,
                           default_s_glob_info=s_glob_info,

                           cap_list=[
                                     ASTFCapInfo(file="../avl/delay_10_http_browsing_0.pcap ↩
                                         ", cps=1)
                                     ]
                           )
```

❶ Set client side global tcp.mss/tcp.initwnd to 1400,1

❷ Set server side global tcp.mss/tcp.initwnd to 1400,1

Example per template tcp.mss tunable

```
class Prof1():
    def get_profile(self):

        # ip generator
        ip_gen_c = ASTFIPGenDist(ip_range=["16.0.0.0", "16.0.0.255"], distribution="seq")
        ip_gen_s = ASTFIPGenDist(ip_range=["48.0.0.0", "48.0.255.255"], distribution="seq")
        ip_gen = ASTFIPGen(glob=ASTFIPGenGlobal(ip_offset="1.0.0.0"),
                           dist_client=ip_gen_c,
                           dist_server=ip_gen_s)

        c_info = ASTFGlobalInfo()
        c_info.tcp.mss = 1200
        c_info.tcp.initwnd = 1

        s_info = ASTFGlobalInfo()
```

```
        s_info.tcp.mss = 1400
        s_info.tcp.initwnd = 10


        return ASTFProfile(default_ip_gen=ip_gen,
                           #  Defaults affects all files

                           cap_list=[
                                     ASTFCapInfo(file="../avl/delay_10_http_browsing_0.pcap ←
                                         ", cps=1)
                                     ASTFCapInfo(file="../avl/delay_10_http_browsing_0.pcap ←
                                         ", cps=1,
                                     port=8080,
                                     c_glob_info=c_info, s_glob_info=s_info), (1)

                                     ]
                           )
```

(1) Only the second template will get the c_info/s_info

Examples Files

- astf/param_ipv6.py
- astf/param_mss_err.py
- astf/param_mss_initwnd.py
- astf/param_tcp_delay_ack.py
- astf/param_tcp_keepalive.py
- astf/param_tcp_no_timestamp.py
- astf/param_tcp_rxbufsize.py
- astf/param_tcp_rxbufsize_8k.py
- astf/tcp_param_change.py

For reference of all tunables see: here [tunables]

2.3.8 Tutorial: Python automation (v2.47 and up)

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

File
astf_example.py

For this mode to work, TRex server should have started with interactive mode.

Start ASTF in interactive mode

```
sudo ./t-rex-64 -i --astf
```

ASTF automation

```
    c = ASTFClient(server = server)

    c.connect()                                                                    (1)

    c.reset()

    if not profile_path:
```

```
        profile_path = os.path.join(astf_path.ASTF_PROFILES_PATH, 'http_simple.py')

    c.load_profile(profile_path)

    c.clear_stats()

    c.start(mult = mult, duration = duration, nc = True)                    ❷

    c.wait_on_traffic()                                                     ❸

    stats = c.get_stats()                                                   ❹

    # use this for debug info on all the stats
    #pprint(stats)

    if c.get_warnings():
        print('\n\n*** test had warnings ****\n\n')
        for w in c.get_warnings():
            print(w)
```

❶ Connect

❷ Start the traffic

❸ Wait for the test to finish

❹ Get all stats

2.3.9 Tutorial: Python automation batch mode (planned to be deprecated)

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

Directories
Python API examples: `automation/trex_control_plane/stf/examples`.

Python API library: `automation/trex_control_plane/stf/trex_stl_lib`.

This mode works with STF python API framework and it is deprecated.

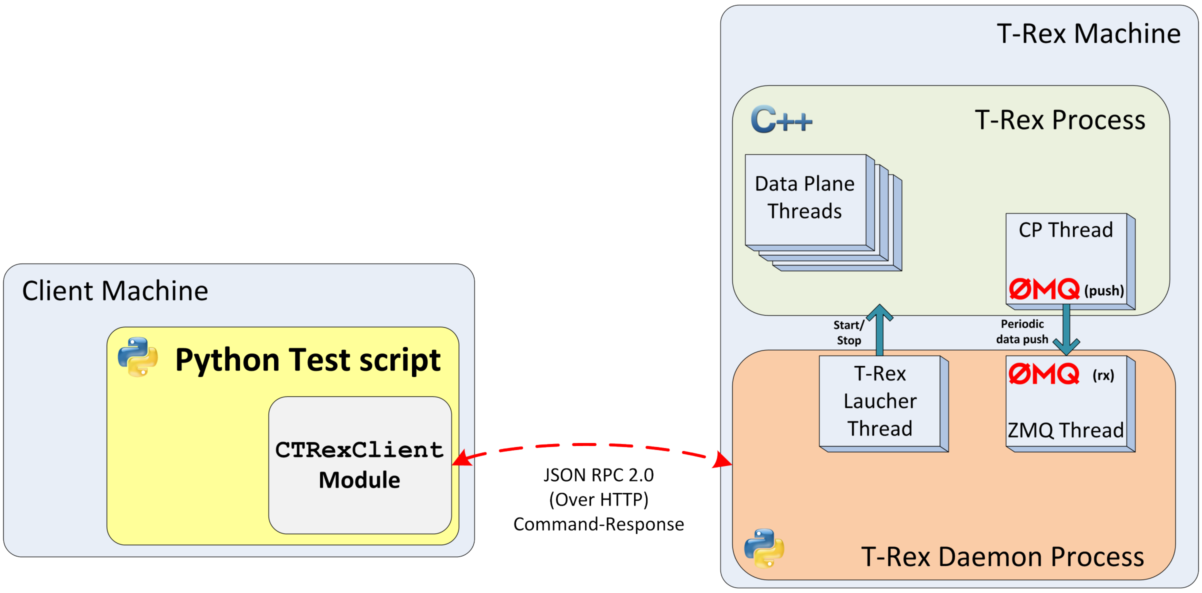


Figure 2.7: RPC Server Components

File

stl_tcp.py

ASTF automation

```
import argparse
import stf_path
from trex_stf_lib.trex_client import CTRexClient          ❶
from pprint import pprint


def validate_tcp (tcp_s):
    if 'err' in tcp_s :
        pprint(tcp_s);
        return(False);
    return True;


def run_stateful_tcp_test(server):

    trex_client = CTRexClient(server)

    trex_client.start_trex(
            c = 1, #                          ❷
            m = 1000,
            f = 'astf/http_simple.py',                ❸
            k=10,
            d = 20,
            l = 1000,
            astf =True, #enable TCP                    ❹
            nc=True
            )

    result = trex_client.sample_until_finish()

    c = result.get_latest_dump()
    pprint(c["tcp-v1"]["data"]);                       ❺
```

```
    tcp_c= c["tcp-v1"]["data"]["client"];
    if not validate_tcp(tcp_c):
       return False
    tcp_s= c["tcp-v1"]["data"]["server"];
    if not validate_tcp(tcp_s):
        return False


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="tcp example")

    parser.add_argument('-s', '--server',
                        dest='server',
                        help='Remote trex address',
                        default='127.0.0.1',
                        type = str)
    args = parser.parse_args()

    if run_stateful_tcp_test(args.server):
        print("PASS");
```

1. Imports the old trex_stf_lib
2. One DP core, could be more
3. load a astf profile
4. enable astf mode
5. check astf client server counters.

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

ASTF JSON format

```
{'client': {'all': {
                    'm_active_flows': 6662,
                    'm_avg_size': 1834.3,
                    },
            'err' : { 'some error counter name' : 'description of the counter'} (1)
            },
 'server': { 'all': {},
             'err' : {}
           }
 }
```

1. *err* object won't exist in case of no error

ASTF example of ASTF counters with no errors

```
{'client': {'all': {'__last': 0,
                    'm_active_flows': 6662,
                    'm_avg_size': 1834.3,
                    'm_est_flows': 6657,
                    'm_rx_bw_l7_r': 369098181.6,
                    'm_rx_pps_r': 12671.8,
                    'm_tx_bw_l7_r': 2804666.1,
                    'm_tx_pps_r': 12672.2,
                    'redirect_rx_ok': 120548,
                    'tcps_closed': 326458,
```

```
                     'tcps_connattempt': 333120,
                     'tcps_connects': 333115,
                     'tcps_delack': 1661439,
                     'tcps_preddat': 1661411,
                     'tcps_rcvackbyte': 83275862,
                     'tcps_rcvackpack': 664830,
                     'tcps_rcvbyte': 10890112648,
                     'tcps_rcvpack': 2326241,
                     'tcps_rttupdated': 997945,
                     'tcps_segstimed': 997962,
                     'tcps_sndacks': 2324887,
                     'tcps_sndbyte': 82945635,
                     'tcps_sndctrl': 333120,
                     'tcps_sndpack': 333115,
                     'tcps_sndtotal': 2991122}},
 'server': {'all': {'__last': 0,
                     'm_active_flows': 6663,
                     'm_avg_size': 1834.3,
                     'm_est_flows': 6657,
                     'm_rx_bw_l7_r': 2804662.9,
                     'm_rx_pps_r': 14080.0,
                     'm_tx_bw_l7_r': 369100825.2,
                     'm_tx_pps_r': 11264.0,
                     'redirect_rx_ok': 120549,
                     'tcps_accepts': 333118,
                     'tcps_closed': 326455,
                     'tcps_connects': 333112,
                     'tcps_rcvackbyte': 10882823775,
                     'tcps_rcvackpack': 2657980,
                     'tcps_rcvbyte': 82944888,
                     'tcps_rcvpack': 664836,
                     'tcps_rttupdated': 2657980,
                     'tcps_segstimed': 2659379,
                     'tcps_sndacks': 664842,
                     'tcps_sndbyte': 10890202264,
                     'tcps_sndpack': 1994537,
                     'tcps_sndtotal': 2659379}}}
```


In case there are no errors the *err* object won't be there. In case of an error counters the *err* section will include the counter and the description. The *all* section includes the good and error counters value.

2.3.10 Tutorial: Simple simulator

Goal
Use the TRex ASTF simple simulator.

The TRex package includes a simulator tool, `astf-sim`. The simulator operates as a Python script that calls an executable. The platform requirements for the simulator tool are the same as for TRex. There is no need for super user in case of simulation.

The TRex simulator can:

Demonstrate the most basic use case using TRex simulator. In this simple simulator there is **one** client flow and **one** server flow and there is **only** one template (the first one). The objective of this simulator is to verify the TCP layer and application layer. In this simulator, it is possible to simulate many abnormal cases for example:

- Drop of specific packets.
- Change of packet information (e.g. wrong sequence numbers)
- Man in the middle RST and redirect

- Keepalive timers.
- Set the round trip time
- Convert the profile to JSON format

We didn't expose all the capabilities of the simulator tool but you could debug the emulation layer using this tool and explore the pcap output files.

Example traffic profile:

File
stl/http_simple.py

The following runs the traffic profile through the TRex simulator, and storing the output in a pcap file.

```
[bash]>./astf-sim -f astf/http_simple.py -o b
```

Those are the pcap file that generated:

- b_c.pcap client side pcap
- b_s.pcap server side pcap

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

Adding `--json` displays the details of the JSON profile

```
[bash]>./astf-sim -f astf/http_simple.py --json
{
    "templates": [                                          ❶
        {
            "client_template": {
                "tcp_info": {
                    "index": 0
                },
                "port": 80,                  # dst port
                "cps": 1,                    # rate in CPS
                "program_index": 0,          # index into program_list
                "cluster": {},
                "ip_gen": {
                    "global": {
                        "ip_offset": "1.0.0.0"
                    },
                    "dist_client": {
                        "index": 0         # index into ip_gen_dist_list
                    },
                    "dist_server": {
                        "index": 1         # index into ip_gen_dist_list
                    }
                }
            },
            "server_template": {
                "program_index": 1,
                "tcp_info": {
                    "index": 0
                },
                "assoc": [
                    {
                        "port": 80         # Which dst port will be associated with this ←
                            template
                    }
```

```
                ]
            }
        }
    ],
    "tcp_info_list": [                              ❷
        {
            "options": 0,
            "port": 80,
            "window": 32768
        }
    ],
    "program_list": [                               ❸
        {
            "commands": [
                {
                    "name": "tx",
                    "buf_index": 0          # index into "buf_list"
                },
                {
                    "name": "rx",
                    "min_bytes": 32089
                }
            ]
        },
        {
            "commands": [
                {
                    "name": "rx",
                    "min_bytes": 244
                },
                {
                    "name": "tx",
                    "buf_index": 1          # index into "buf_list"
                }
            ]
        }
    ],
    "ip_gen_dist_list": [                           ❹
        {
            "ip_start": "16.0.0.1",
            "ip_end": "16.0.0.255",
            "distribution": "seq"
        },
        {
            "ip_start": "48.0.0.1",
            "ip_end": "48.0.255.255",
            "distribution": "seq"
        }
    ],
    "buf_list": [                                   ❺
        "R0VUIC8zMzg0IEhUVFAvMS4xDQpIb3",
        "SFRUUC8xLjEgMjAwIE9LDQpTZXJ2ZX"
    ]
}
```

❶ A list of templates with the properties of each template

❷ A list of indirect distinct tcp/ip options

❸ A list of indirect distinct emulation programs

❹ A list of indirect distinct tuple generator

5. A list of indirect distinct L7 buffers, used by emulation program (indirect) (e.g. "buf_index": 1)

Note
We might change the JSON format in the future as this is a first version

---

2.3.11 Tutorial: Advanced simulator

Goal
Use the TRex ASTF advanced simulator.

It is like the simple simulator but simulates multiple templates and flows exacly like TRex server would do with one DP core.

```
[bash]>./astf-sim -f astf/http_simple.py --full -o b.pcap
```

- Use '--full' to initiate the full simulation mode.
- b.pcap output pcap file will be generated, it is the client side multiplex pcap.
- There is no server side pcap file in this simulation because we are not simulating latency/jitter/drop in this case so the server should be the same as client side.

Another example that will run sfr profile in release mode and will show the counters

```
[bash]>./astf-sim -f astf/sfr.py --full -o o.pcap -d 1 -r -v
```

2.3.12 Tutorial: Manual L7 emulation program building

Goal
Build the L7 program using low level commands.

Manual L7 emulation program

```
# we can send either Python bytes type as below:
http_req = b'GET /3384 HTTP/1.1\r\nHost: 22.0.0.3\r\nConnection: Keep-Alive\r\nUser-Agent: ↩
    Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322; .NET CLR ↩
    2.0.50727)\r\nAccept: */*\r\nAccept-Language: en-us\r\nAccept-Encoding: gzip, deflate, ↩
    compress\r\n\r\n'
# or we can send Python string containing ascii chars, as below:
http_response = 'HTTP/1.1 200 OK\r\nServer: Microsoft-IIS/6.0\r\nContent-Type: text/html\r\ ↩
    nContent-Length: 32000\r\n\r\n<html><pre>**********</pre></html>'

class Prof1():
    def __init__(self):
        pass  # tunables

    def create_profile(self):
        # client commands
        prog_c = ASTFProgram()            1
        prog_c.send(http_req)             2
        prog_c.recv(len(http_response))  3

        prog_s = ASTFProgram()
        prog_s.recv(len(http_req))
        prog_s.send(http_response)
```

```
        # ip generator
        ip_gen_c = ASTFIPGenDist(ip_range=["16.0.0.0", "16.0.0.255"], distribution="seq")
        ip_gen_s = ASTFIPGenDist(ip_range=["48.0.0.0", "48.0.255.255"], distribution="seq")
        ip_gen = ASTFIPGen(glob=ASTFIPGenGlobal(ip_offset="1.0.0.0"),
                           dist_client=ip_gen_c,
                           dist_server=ip_gen_s)

        tcp_params = ASTFTCPInfo(window=32768)

        # template
        temp_c = ASTFTCPClientTemplate(program=prog_c, tcp_info=tcp_params, ip_gen=ip_gen)
        temp_s = ASTFTCPServerTemplate(program=prog_s, tcp_info=tcp_params)  # using ←
            default association
        template = ASTFTemplate(client_template=temp_c, server_template=temp_s)

        # profile
        profile = ASTFProfile(default_ip_gen=ip_gen, templates=template)
        return profile

    def get_profile(self):
        return self.create_profile()
```

1. Build the emulation program
2. First send http request
3. Wait for http response

We will expose in the future a capability that could take a pcap file and convert it to a Python code so you could tune it yourself.

2.3.13 Tutorial: Profile CLI tunable

Goal
Tune a profile by the CLI arguments. For example change the response size by given args.

Every traffic profile must define the following function:

```
def create_profile(self,**kwargs)
```

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

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

Usage notes for defining parameters

- All parameters require default values.
- A profile must be loadable with no parameters specified.
- Every tunable must be expressed as key-value pair with a default value.
- `-t key=val,key=val` is the way to provide the key-value to the profile.

Example http_res_size

```
def create_profile (self, **kwargs):
    # the size of the response size
    http_res_size = kwargs.get('size',1)

    # use http_res_size
    http_response = http_response_template.format('*'*http_res_size)
```

Change tunable from CLI using -t

```
[bash]>sudo ./t-rex-64 -f astf/http_manual_cli_tunable.py -m 1000 -d 1000 -c 1 --astf -l ↩
    1000 -t size=1
[bash]>sudo ./t-rex-64 -f astf/http_manual_cli_tunable.py -m 1000 -d 1000 -c 1 --astf -l ↩
    1000 -t size=10000
[bash]>sudo ./t-rex-64 -f astf/http_manual_cli_tunable.py -m 1000 -d 1000 -c 1 --astf -l ↩
    1000 -t size=1000000
```

Simulator

```
[bash]>./astf-sim  -f astf/http_manual_cli_tunable.py --json
[bash]>./astf-sim  -f astf/http_manual_cli_tunable.py -t size=1000 --json
[bash]>./astf-sim  -f astf/http_manual_cli_tunable.py -t size=1000 -o a.cap --full
```

2.3.14 Tutorial: L7 emulation - fin/ack/fin/ack

By default when the L7 emulation program is ended the socket is closed implicitly.

This example forces the **server** side to wait for close from peer (client) and only then will send FIN.

fin-ack example

```
        # client commands
        prog_c = ASTFProgram()
        prog_c.send(http_req)
        prog_c.recv(len(http_response))
        # implicit  close

        prog_s = ASTFProgram()
        prog_s.recv(len(http_req))
        prog_s.send(http_response)
        prog_s.wait_for_peer_close(); # wait for client to close the socket the issue a ↩
            close
```

The packets trace should look like this:

```
client server
-------------
  FIN
       ACK
       FIN
 ACK
-------------
```

See astf-program for more info

2.3.15 Tutorial: L7 emulation - syn-syn/ack-ack/rst

By default, when the L7 emulation program is started the sending buffer waits inside the socket.

This is seen as SYN/SYN-ACK/GET-ACK in the trace (piggyback ack in the GET requests).

To force the client side to send ACK and only **then** send the data use the *connect()* command.

```
client server
-------------
  SYN
        SYN-ACK
  ACK
  GET
-------------
```

Connect() example with RST

```
        prog_c = ASTFProgram()
        prog_c.connect(); ## connect
        prog_c.reset();   ## send RST from client side

        prog_s = ASTFProgram()
        prog_s.wait_for_peer_close(); # wait for client to close the socket
```

This example will wait for connect and then will send RST packet to shutdown peer and current socket.

```
client server
-------------
  SYN
        SYN-ACK
  ACK
  RST
-------------
```

See astf-program for more info.

Server Connect() server send the traffic first

```
        prog_c = ASTFProgram()
        prog_c.recv(len(http_resp))

        prog_s = ASTFProgram()
        prog_s.connect()
        prog_s.send(http_resp)
        prog_s.wait_for_peer_close(); # wait for client to close the socket
```

In this example the server send the request first and there should be a connect from the server side else the program won't work.

2.3.16 Tutorial: L7 emulation - server send the first data

When the server is the first to send the data (e.g. citrix,telnet) there is a need to wait for the server to accept the connection.

accept() server example

```
        prog_c = ASTFProgram()
        prog_c.recv(len(http_response))
        prog_c.send(http_req)

        prog_s = ASTFProgram()
        prog_s.accept()                # server waits for the connection to be established
        prog_s.send(http_response)
        prog_s.recv(len(http_req))
```

See astf/http_reverse2.py

2.3.17 Tutorial: L7 emulation - delay server response

Constant delay

```
        prog_c = ASTFProgram()
        prog_c.send(http_req)
        prog_c.recv(len(http_response))

        prog_s = ASTFProgram()
        prog_s.recv(len(http_req))
        prog_s.delay(500000); # delay 500msec (500,000usec)
        prog_s.send(http_response)
```

This example will delay the server response by 500 msec.

Random delay

```
prog_c = ASTFProgram()
prog_c.send(http_req)
prog_c.recv(len(http_response))

prog_s = ASTFProgram()
prog_s.recv(len(http_req))
prog_s.delay_rand(100000,500000); # delay random number betwean 100msec-500msec
prog_s.send(http_response)
```

This example will delay the server by a random delay between 100-500 msec

See astf-program for more info.

2.3.18 Tutorial: L7 emulation - delay client request

This example will delay the client side.

In this example the client sends partial request (10 bytes), waits 100msec and then sends the rest of the request (there would be two segments for one request).

Client delay

```
prog_c = ASTFProgram()
prog_c.send(http_req[:10])
prog_c.delay(100000); # delay 100msec
prog_c.send(http_req[10:])
prog_c.recv(len(http_response))

prog_s = ASTFProgram()
prog_s.recv(len(http_req))
prog_s.send(http_response)
```

Client delay after connect

```
# client commands
prog_c = ASTFProgram()
prog_c.delay(100000); # delay 100msec
prog_c.send(http_req)
prog_c.recv(len(http_response))

prog_s = ASTFProgram()
prog_s.recv(len(http_req))
prog_s.send(http_response)
```

In this example the client connects first, waits for 100msec and only then sends full request (there would be **one** segment for one request).

See astf-program for more info.

A side effect of this delay is more active-flows.

2.3.19 Tutorial: L7 emulation - Elephant flows

Let say we would like to send only 50 flows with very big size (4GB). Loading a 4GB buffer would be a challenge as TRex's memory is limited. What we can do is loop inside the server side to send 1MB buffer 4096 times and then finish with termination.

Client Delay

```
        prog_c = ASTFProgram()
        prog_c.send(http_req)
        prog_c.recv(0xffffffff)

        prog_s = ASTFProgram()
        prog_s.recv(len(http_req))
        prog_s.set_var("var2",4096);    #1
        prog_s.set_label("a:");         #2
        prog_s.send(http_response_1mbyte)
        prog_s.jmp_nz("var2","a:")      #3 dec var "var2". in case it is *not* zero jump a:
```

1. Set varibale
2. Set label
3. Jump to label 4096 times

See astf-program for more info.

Usually in case of very long flows there is need to cap the number of active flows, this can be done by `limit` directive.

Limit the number to flows

```
  cap_list=[ASTFCapInfo(file="../avl/delay_10_http_browsing_0.pcap",
                             cps=1,limit=50)])  #1
```

1. Use `limit` field to control the total flows generated.

2.3.20 Tutorial: L7 emulation - Elephant flows with non-blocking send

By default `send()` command waits for the ACK on the last byte. To make it non-blocking, especially in case big BDP (large window is required) it is possible to work in non-blocking mode, this way to achieve full pipeline.

Have a look at `astf/htttp_eflow2.py` example.

Non-blocking send

```
    def create_profile(self,size,loop,mss,win,pipe):

        http_response = 'HTTP/1.1'

        bsize = len(http_response)

        r=self.calc_loops (bsize,loop)

        # client commands
        prog_c = ASTFProgram()
        prog_c.send(http_req)

        if r[1]==0:
          prog_c.recv(r[0])
        else:
            prog_c.set_var("var1",r[1]);
            prog_c.set_label("a:");
            prog_c.recv(r[0],True)
            prog_c.jmp_nz("var1","a:")
            if r[2]:
               prog_c.recv(bsize*r[2])
```

```
        prog_s = ASTFProgram()
        prog_s.recv(len(http_req))
        if pipe:
          prog_s.set_send_blocking (False)    #❶
        prog_s.set_var("var2",loop-1);
        prog_s.set_label("a:");
        prog_s.send(http_response)
        prog_s.jmp_nz("var2","a:")
        prog_s.set_send_blocking (True)       #❷
        prog_s.send(http_response)
```

❶ Set all send mode to be non-blocking from now on

❷ Back to blocking mode. To make the last send blocking

See astf-program for more info.

2.3.21 Tutorial: L7 emulation - Elephant flows with non-blocking send with tick var

Same as the previous example, only instead of using `loop count` we are using time as a measurement. In the prevous example we calculated the received bytes in advance and use the rcv command with the right bytes values. However, sending & receiving data according to time is tricky and errors/dtops might occur (see stats from runnnig example below).

Have a look at `astf/htttp_eflow4.py` example.

Tick var

```
        ...
        # client commands
        prog_c = ASTFProgram()
        prog_c.send(http_req)
        prog_c.set_tick_var("var1")  ❶
        prog_c.set_label("a:")
        prog_c.recv(len(http_response), clear = True)
        prog_c.jmp_dp("var1", "a:", recv_time)  ❷
        prog_c.reset()

        # server commands
        prog_s = ASTFProgram()
        prog_s.recv(len(http_req), clear = True)
        prog_s.set_tick_var("var2")  ❸
        prog_s.set_label("b:")
        prog_s.send(http_response)
        prog_s.jmp_dp("var2", "b:", send_time)  ❹
        prog_s.reset()
        ...
```

❶ Start the clock at client side.

❷ In case time passed since "var1" is less than "recv_time", jump to a:

❸ Start the clock at server side.

❹ In case time passed since "var2" is less than "recv_time", jump to b:

Note
reset() command is required in both sides in order to ignore some ASTF errors.

2.3.21.1 Inspecting stats with running example

Let's test the script using tunables:

```
[bash]> sudo ./t-rex-64 --astf -f astf/http_eflow4.py -t send_time=2,recv_time=5
```

TUI stats:

```
                    TCP |                   |                   |
                      - |                   |                   |
        tcps_connattempt |                 1 |                 0 | connections initiated
            tcps_accepts |                 0 |                 1 | connections accepted
           tcps_connects |                 1 |                 1 | connections established
             tcps_closed |                 1 |                 1 | conn. closed (includes ↩
                 drops)
          tcps_segstimed |                 2 |               998 | segs where we tried to ↩
              get rtt
          tcps_rttupdated |                 2 |               998 | times we succeeded
           tcps_sndtotal |               999 |               999 | total packets sent
            tcps_sndpack |                 1 |               997 | data packets sent
            tcps_sndbyte |               249 |           1138574 | data bytes sent by ↩
                application
          tcps_sndbyte_ok |               249 |           1138574 | data bytes sent by tcp
            tcps_sndctrl |                 1 |                 1 | control (SYN|FIN|RST) ↩
                packets sent
            tcps_sndacks |               997 |                 1 | ack-only packets sent
            tcps_rcvpack |               997 |                 1 | packets received in ↩
                sequence
            tcps_rcvbyte |           1138574 |               249 | bytes received in ↩
                sequence
          tcps_rcvackpack |                 1 |               998 | rcvd ack packets
          tcps_rcvackbyte |               249 |           1138574 | tx bytes acked by rcvd ↩
             acks
       tcps_rcvackbyte_of |                 0 |                 1 | tx bytes acked by rcvd ↩
          acks - overflow acked
            tcps_preddat |               996 |                 0 | times hdr predict ok for ↩
                data pkts
              tcps_drops |                 1 |                 1 | connections dropped
            tcps_predack |                 0 |               977 | times hdr predict ok for ↩
                acks
                      - |                   |                   |
                    UDP |                   |                   |
                      - |                   |                   |
                      - |                   |                   |
              Flow Table |                   |                   |
                      - |                   |                   |
```

We got `tcps_drops` error on client side, and `connections dropped` on server side. These can be related as an artifact of synchronize commands.

Let's see the opposite case using tunables:

```
[bash]> sudo ./t-rex-64 --astf -f astf/http_eflow4.py -t send_time=5,recv_time=2
```

TUI stats:

```
                    TCP |                   |                   |
                      - |                   |                   |
        tcps_connattempt |                 1 |                 0 | connections initiated
            tcps_accepts |                 0 |                 1 | connections accepted
           tcps_connects |                 1 |                 1 | connections established
             tcps_closed |                 1 |                 1 | conn. closed (includes ↩
                 drops)
```

```
        tcps_segstimed |              2 |             1001 | segs where we tried to ↩
            get rtt
       tcps_rttupdated |              2 |             1000 | times we succeeded
         tcps_sndtotal |           1002 |             1001 | total packets sent
          tcps_sndpack |              1 |             1000 | data packets sent
          tcps_sndbyte |            249 |          1142000 | data bytes sent by  ↩
              application
       tcps_sndbyte_ok |            249 |          1142000 | data bytes sent by tcp
          tcps_sndctrl |              2 |                0 | control (SYN|FIN|RST)  ↩
              packets sent
          tcps_sndacks |            999 |                1 | ack-only packets sent
          tcps_rcvpack |            999 |                1 | packets received in  ↩
              sequence
          tcps_rcvbyte |        1140858 |              249 | bytes received in  ↩
              sequence
       tcps_rcvackpack |              1 |             1000 | rcvd ack packets
       tcps_rcvackbyte |            249 |          1140858 | tx bytes acked by rcvd  ↩
           acks
    tcps_rcvackbyte_of |              0 |                1 | tx bytes acked by rcvd  ↩
          acks - overflow acked
          tcps_preddat |            998 |                0 | times hdr predict ok for  ↩
              data pkts
            tcps_drops |              1 |                1 | connections dropped
          tcps_predack |              0 |              979 | times hdr predict ok for  ↩
              acks
                     - |                |                  |
                   UDP |                |                  |
                     - |                |                  |
                     - |                |                  |
            Flow Table |                |                  |
                     - |                |                  |
               err_cwf |              1 |                0 | client pkt without flow
            err_no_syn |              0 |                1 | server first flow packet  ↩
                with no SYN
```

In addition to the drops we got `err_cwf` and `err_no_syn` errors, These also can be related as an artifact of synchronize commands.

Note
Depending on time for send/recv commands will almost always cause errors/drops. In most cases the wanted behavior is by loop count with bytes, not time.

2.3.22 Tutorial: L7 emulation - http pipeline

In this example, there would be 5 parallel requests and wait for 5 responses. The first response could come while we are sending the first request as the Rx side and Tx side work in parallel.

Pipeline

```
        pipeline=5;
        # client commands
        prog_c = ASTFProgram()
        prog_c.send(pipeline*http_req)
        prog_c.recv(pipeline*len(http_response))

        prog_s = ASTFProgram()
        prog_s.recv(pipeline*len(http_req))
        prog_s.send(pipeline*http_response)
```

See astf-program for more info.

2.3.23 Tutorial: L7 emulation - UDP example

This example will show an UDP example.

Client delay

```
        # client commands
        prog_c = ASTFProgram(stream=False)
        prog_c.send_msg(http_req)                  #1
        prog_c.recv_msg(1)                         #2

        prog_s = ASTFProgram(stream=False)
        prog_s.recv_msg(1)
        prog_s.send_msg(http_response)
```

1. Send UDP message
2. Wait for number of packets

In case of pcap file, it will be converted to send_msg/recv_msg/delay taken from the pcap file.

2.3.24 Tutorial: Template group.

A template group is, as the name suggests, a group of templates that share statistics. Sharing statistics for templates has a number of use cases. For example, one would like to track the statistics of two different programs under the same profile. Another use case would be grouping different programs and comparing statistics between groups. Let us demonstrance a simple case, in which we run the same program with different cps in two different template groups.

File
astf/template_groups.py

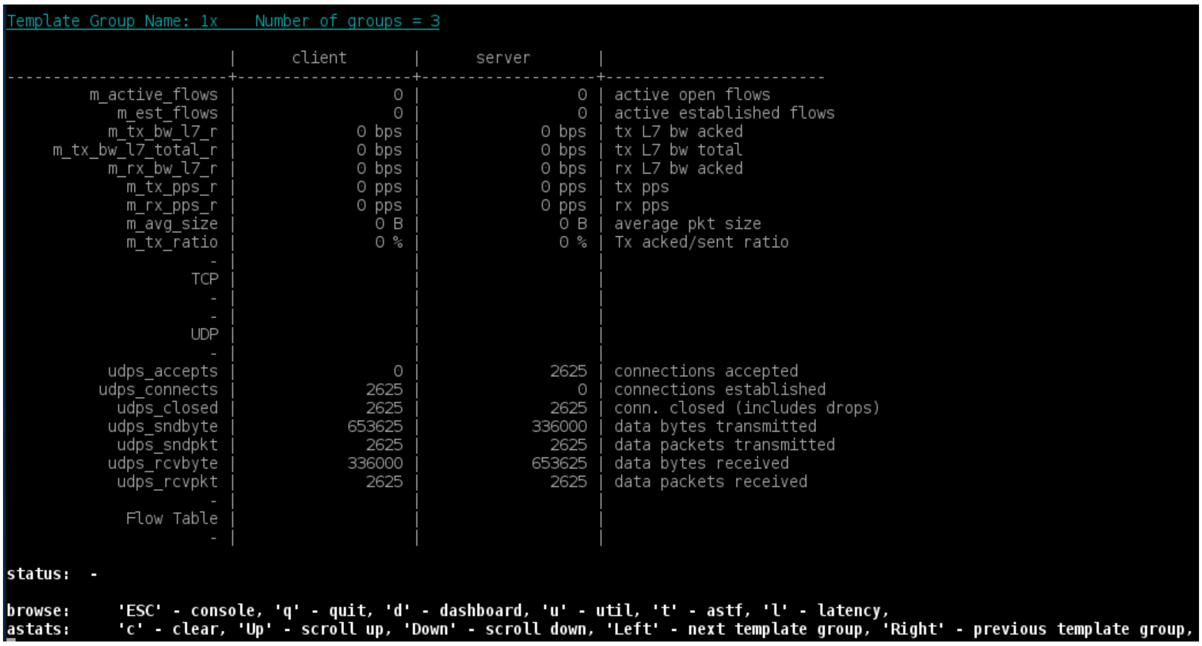
Two simple template groups

```
    def create_profile(self):
        # client commands
        prog_c = ASTFProgram(stream=False)
        prog_c.send_msg(http_req)
        prog_c.recv_msg(1)

        prog_s = ASTFProgram(stream=False)
        prog_s.recv_msg(1)
        prog_s.send_msg(http_response)

        # ip generator
        ip_gen_c = ASTFIPGenDist(ip_range=["16.0.0.0", "16.0.0.255"], distribution="seq")
        ip_gen_s = ASTFIPGenDist(ip_range=["48.0.0.0", "48.0.255.255"], distribution="seq")
        ip_gen = ASTFIPGen(glob=ASTFIPGenGlobal(ip_offset="1.0.0.0"),
                           dist_client=ip_gen_c,
                           dist_server=ip_gen_s)


        # template
        temp_c1 = ASTFTCPClientTemplate(port=80, program=prog_c,ip_gen=ip_gen, cps = 1)
        temp_c2 = ASTFTCPClientTemplate(port=81, program=prog_c, ip_gen=ip_gen, cps = 2)
        temp_s1 = ASTFTCPServerTemplate(program=prog_s, assoc=ASTFAssociationRule(80))
        temp_s2 = ASTFTCPServerTemplate(program=prog_s, assoc=ASTFAssociationRule(81))
        t1 = ASTFTemplate(client_template=temp_c1, server_template=temp_s1, tg_name = '1x') ←
            1
        t2 = ASTFTemplate(client_template=temp_c2, server_template=temp_s2, tg_name = '2x')
```

```
    # profile
    profile = ASTFProfile(default_ip_gen=ip_gen, templates=[t1, t2])
    return profile
```

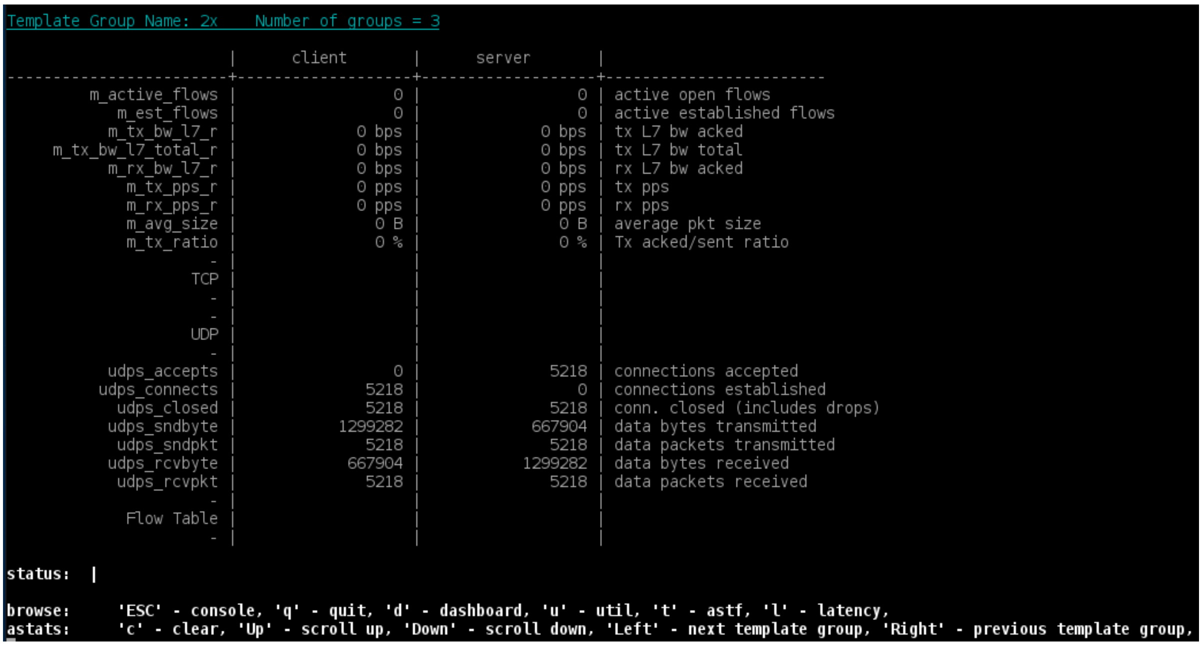
1. The group name is *1x*. In this simple profile both template groups contain a single template.

We can track the statistics using the Trex console. Within the console we can use the TUI (see previous examples) or the template group section api for live tracking of the counters. To move between template groups in TUI use the right and left arrow.



```
Template Group Name: 1x    Number of groups = 3

                       |      client       |      server       |
-----------------------+-------------------+-------------------+-----------------------
        m_active_flows |                 0 |                 0 | active open flows
           m_est_flows |                 0 |                 0 | active established flows
          m_tx_bw_l7_r |             0 bps |             0 bps | tx L7 bw acked
    m_tx_bw_l7_total_r |             0 bps |             0 bps | tx L7 bw total
          m_rx_bw_l7_r |             0 bps |             0 bps | rx L7 bw acked
            m_tx_pps_r |             0 pps |             0 pps | tx pps
            m_rx_pps_r |             0 pps |             0 pps | rx pps
            m_avg_size |               0 B |               0 B | average pkt size
            m_tx_ratio |               0 % |               0 % | Tx acked/sent ratio
                     - |                   |                   |
                   TCP |                   |                   |
                     - |                   |                   |
                     - |                   |                   |
                   UDP |                   |                   |
                     - |                   |                   |
          udps_accepts |                 0 |              2625 | connections accepted
         udps_connects |              2625 |                 0 | connections established
           udps_closed |              2625 |              2625 | conn. closed (includes drops)
          udps_sndbyte |            653625 |            336000 | data bytes transmitted
           udps_sndpkt |              2625 |              2625 | data packets transmitted
          udps_rcvbyte |            336000 |            653625 | data bytes received
           udps_rcvpkt |              2625 |              2625 | data packets received
                     - |                   |                   |
            Flow Table |                   |                   |
                     - |                   |                   |

status:  -

browse:     'ESC' - console, 'q' - quit, 'd' - dashboard, 'u' - util, 't' - astf, 'l' - latency,
astats:     'c' - clear, 'Up' - scroll up, 'Down' - scroll down, 'Left' - next template group, 'Right' - previous template group,
```

Figure 2.8: Statistics for template group '1x'



```
Template Group Name: 2x    Number of groups = 3

                         |      client       |      server       |
-------------------------+-------------------+-------------------+-----------------------
         m_active_flows |                 0 |                 0 | active open flows
            m_est_flows |                 0 |                 0 | active established flows
           m_tx_bw_l7_r |             0 bps |             0 bps | tx L7 bw acked
     m_tx_bw_l7_total_r |             0 bps |             0 bps | tx L7 bw total
           m_rx_bw_l7_r |             0 bps |             0 bps | rx L7 bw acked
             m_tx_pps_r |             0 pps |             0 pps | tx pps
             m_rx_pps_r |             0 pps |             0 pps | rx pps
             m_avg_size |               0 B |               0 B | average pkt size
             m_tx_ratio |               0 % |               0 % | Tx acked/sent ratio
                      - |                   |                   |
                    TCP |                   |                   |
                      - |                   |                   |
                      - |                   |                   |
                    UDP |                   |                   |
                      - |                   |                   |
           udps_accepts |                 0 |              5218 | connections accepted
          udps_connects |              5218 |                 0 | connections established
            udps_closed |              5218 |              5218 | conn. closed (includes drops)
           udps_sndbyte |           1299282 |            667904 | data bytes transmitted
            udps_sndpkt |              5218 |              5218 | data packets transmitted
           udps_rcvbyte |            667904 |           1299282 | data bytes received
            udps_rcvpkt |              5218 |              5218 | data packets received
                      - |                   |                   |
             Flow Table |                   |                   |
                      - |                   |                   |

status:  |

browse:     'ESC' - console, 'q' - quit, 'd' - dashboard, 'u' - util, 't' - astf, 'l' - latency,
astats:     'c' - clear, 'Up' - scroll up, 'Down' - scroll down, 'Left' - next template group, 'Right' - previous template group,
```

Figure 2.9: Statistics for template group '2x'

We can see that the ratio of *udps_connects* between the two groups converges to the ratio of cps, namely 2.

Next we explore how to use the *template_group* section directly from the console. For complete information, from the TRex console write:

```
[trex]>template_group --help
```

You will see that the section has two commands, *names* and *stats*.
The *names* command can receive two parameters, *start* (default value is 0) and *amount* (default value is 50). It will show in the screen the list of names [start : start+amount]. You can use it this way:

```
[trex]>template_group names --start 3 --amount 20
```

Lastly, the *stats* command receives as parameter the name of a template group and shows the statistics of this group. For example, one can write:

```
[trex]>template_group stats --name 1x
```

2.3.24.1 Client API

If you are performing automation, the client API might come in handy (see more in TRex ASTF API). Using the API we can get the names of all the active template groups. We can also get statistics for each template group or groups. The following example clarifies the use:

```
        self.c.load_profile(profile)
        self.c.clear_stats()
        self.c.start(duration = 60, mult = 100)
        self.c.wait_on_traffic()
        names = self.c.get_tg_names()                  (1)
        stats = self.c.get_traffic_tg_stats(names[:2]) (2)
```

1. Returns a list of the template group names on the profile.
2. Receives a list of template group names, and returns a dictionary of statistics per template group name.

2.3.25 Tutorial: Wrapping it up example.

ASTF mode can do much more than what we saw in the previous examples. We can generate payloads offline and send them to the server. These payloads can also be updated resembling the STL field engine.

Note
The actual file contains more templates than this example.

File
astf/wrapping_it_up_example.py

ASTF creating the payloads offline

```
    def __init__(self):
        self.cq_depth = 256
        self.base_pkt_length = 42
        self.payload_length = 14 + 8 + 16
        self.packet_length = self.base_pkt_length + self.payload_length
        self.cmpl_ofst = 14 + 8
        self.cq_ofst = 14
        self.cmpl_base = (1 << 14) | (1 << 15)


    def create_first_payload(self, color):
        cqe = "%04X%04X%08X%02X%02X%04X%04X%02X%02X" % (
                     0,      # placeholder for completed index
                     0,      # q_number_rss_type_flags
                     0,      # RSS hash
                     self.packet_length,  # bytes_written_flags
                     0,
                     0,      # vlan
                     0,      # cksum
                     ((1 << 0) | (1 << 1) | (1 << 3) | (1 << 5)), # flags
                     7 | color
                 )
        return ('z' * 14 + 'x' * 8 + base64.b16decode(cqe))

    def update_payload(self, payload, cmpl_ofst, cmpl_idx, cq_ofst, cq_addr):       #1
        payload = payload[0:cmpl_ofst] + struct.pack('<H', cmpl_idx) + payload[cmpl_ofst ↩
            +2:]
        payload = payload[0:cq_ofst] + struct.pack('!Q', cq_addr) + payload[cq_ofst+8:]
        return payload
```

1. Updates the payload based on the previous payload and on two variables (cmpl_idx, cq_addr),

ASTF template

```
    def create_template(self, sip, dip, cq_addr1, cq_addr2, color1, color2, pps):

        prog_c = ASTFProgram(stream=False)                                  #1

        # Send the first 256 packets
        cmpl_idx = self.cmpl_base
        my_cq_addr = cq_addr1
        payload = self.create_first_payload(color1)

        for _ in range(self.cq_depth):
```

```
            payload = self.update_payload(payload, self.cmpl_ofst, cmpl_idx,
                                          self.cq_ofst, my_cq_addr)         #2
            prog_c.send_msg(payload)                                        #3
            prog_c.delay(1000000/pps)                                       #4
            cmpl_idx += 1
            my_cq_addr += 16

        # Send the second 256 packets
        cmpl_idx = self.cmpl_base
        my_cq_addr = cq_addr2
        payload = self.create_first_payload(color2)

        for _ in range(self.cq_depth):
            payload = self.update_payload(payload, self.cmpl_ofst, cmpl_idx,
                                          self.cq_ofst, my_cq_addr)
            prog_c.send_msg(payload)
            prog_c.delay(1000000/pps)
            cmpl_idx += 1
            my_cq_addr += 16

        ip_gen_c = ASTFIPGenDist(ip_range=[sip, sip], distribution="seq")       #5
        ip_gen_s = ASTFIPGenDist(ip_range=[dip, dip], distribution="seq")
        ip_gen = ASTFIPGen(glob=ASTFIPGenGlobal(ip_offset="1.0.0.0"),
                           dist_client=ip_gen_c,
                           dist_server=ip_gen_s)

        prog_s = ASTFProgram(stream=False)
        prog_s.recv_msg(2*self.cq_depth)                                    #6

        temp_c = ASTFTCPClientTemplate(program=prog_c, ip_gen=ip_gen, limit=1)  #7
        temp_s = ASTFTCPServerTemplate(program=prog_s)  # using default association
        return ASTFTemplate(client_template=temp_c, server_template=temp_s)
```

1. stream = False means UDP
2. Update the payload as the Field Engine in STL would.
3. Send the message to the server.
4. pps = packets per second - therefore delay is 1 sec / pps
5. IP range can be configured. In this example the IP is fixed.
6. Server expects to receive twice 256 packets.
7. limit = 1 means that the template will generate only one flow.

In the end we create a profile with two templates (could be much more).

ASTF profile

```
    def create_profile(self, pps):

        # ip generator
        source_ips = ["10.0.0.1", "33.33.33.37"]
        dest_ips = ["10.0.0.3", "199.111.33.44"]
        cq_addrs1 = [0x84241d000, 0x1111111111111111]
        cq_addrs2 = [0x84241d000, 0x1818181818181818]
        colors1 = [0x80, 0]
        colors2 = [0x00, 0x80]
        templates = []
```

```
        for i in range(2):
            templates.append(self.create_template(sip=source_ips[i], dip=dest_ips[i],
                             cq_addr1=cq_addrs1[i], cq_addr2=cq_addrs2[i],
                             color1=colors1[i], color2=colors2[i], pps=pps))

        # profile
        ip_gen_c = ASTFIPGenDist(ip_range=[source_ips[0], source_ips[0]], distribution="seq ↩
            ")
        ip_gen_s = ASTFIPGenDist(ip_range=[dest_ips[0], dest_ips[0]], distribution="seq")
        ip_gen = ASTFIPGen(glob=ASTFIPGenGlobal(ip_offset="1.0.0.0"),
                           dist_client=ip_gen_c,
                           dist_server=ip_gen_s)

        return ASTFProfile(default_ip_gen=ip_gen, templates=templates)

    def get_profile(self, **kwargs):
        pps = kwargs.get('pps', 1)                                   #❶
        return self.create_profile(pps)
```

❶ pps is a tunable like the ones shown in the previous tutorials. You can add it while using the CLI with -t pps=20. In case it wasn't specified pps=1.

2.3.26 Tutorial: Server port sharing between templates

In the test environment with firewalls, the number of ports can be limited. In this situation, a server port needs to be shared by multiple templates from multiple profiles.

In general, different templates should use different port numbers for their different actions.

Typical server port usage

```
ASTFAssociationRule()             # for template 1, default port is 80
ASTFAssociationRule(port=81)      # for template 2
```

This example will show how to share a server port between templates.

File
astf/shared_port.py

By different IP range

On the server port sharing, different templates can use the same port. At first, they are differentiated by IPs. `ip_start` and `ip_end` parameters can be used for this.

Server port sharing by different IP range

```
ASTFAssociationRule(ip_start="48.0.0.0", ip_end="48.0.0.255")    # for template 1
ASTFAssociationRule(ip_start="48.0.1.0", ip_end="48.0.255.255") # for template 2
```

By different L7 contents

If multiple templates need to use the same port and IP for their different actions, the `l7_map` parameter can be used. It can specify the L7 contents to distinguish the template.

Server port sharing by different L7 contents

```
prog_c1.send(b"GET /3384 HTTP/1.1....")      # for template 1 -- ❶
prog_c2.send(b"POST /3384 HTTP/1.1....")     # for template 2 -- ❷

ASTFAssociationRule(port=80, ip_start="48.0.0.0", ip_end="48.0.0.255",
                    l7_map={"offset":[0,1,2,3],         # -- ❸
                            "mask":[255,255,255,255]    # -- ❹
                            }                           # -- ❺
                    )
```

1, 2 The client program should send diffent L7 contents in the specified "offset". The value is "GET " for template 1 and "POST" for template 2.

3 mandatory, increment offsets is available up to 8.

4 optional, default is 255 for each offset value. The value at each offset can be masked. value =<value at offset> & mask

5 For short, l7_map=[0,1,2,3].

In addition, all the TCP tuneable should be the same for server port sharing. The TCP protocol action needs to be the same until the first TCP payload delivered and a template is identified. For UDP, it is recommended that all the UDP payloads have the same contents at the specified "offset". Packet loss should be considered for UDP.

Server Mode

If you want to use it for the server mode, "value" should be used. The sizes of "offset" and "value" should be the same.

Server port sharing by specific offset and value

```
ASTFAssociationRule(l7_map={"offset":[0,1,2,3], "value":[71,69,84,32]}) # port=80, "GET "
ASTFAssociationRule(l7_map={"offset":[0,1,2,3], "value":[80,79,83,84]}) # port=80, "POST"
```

Mixed usage by IP range and L7 contents

You can use the IP range and L7 contents for the same port. At first, the L7 contents matching will be performed. If there is no matching, then the IP address will be matched.

```
ASTFAssociationRule(port=80, ip_start="48.0.0.0", ip_end="48.0.0.255")  # for template 1
ASTFAssociationRule(port=80, ip_start="48.0.0.0", ip_end="48.0.0.255",
                    l7_map=[0,1,2,3])                                   # for template 2
```

2.3.27 Tutorial: Run TRex in GTP mode with simple HTTP profile

Goal

Send HTTP flows with GTP tunnel added and remove GTP tunnel when traffic comes back to trex client

Traffic profile

The following profile defines one template of HTTP

File

astf/http_simple.py

```
from trex.astf.api import *


class Prof1():
    def get_profile(self):
        # ip generator
        ip_gen_c = ASTFIPGenDist(ip_range=["16.0.0.0", "16.0.0.255"],
                                 distribution="seq")
        ip_gen_s = ASTFIPGenDist(ip_range=["48.0.0.0", "48.0.255.255"],
                                  distribution="seq")
        ip_gen = ASTFIPGen(glob=ASTFIPGenGlobal(ip_offset="1.0.0.0"),        1
                           dist_client=ip_gen_c,
                           dist_server=ip_gen_s)

        return ASTFProfile(default_ip_gen=ip_gen,
                            cap_list=[ASTFCapInfo(
                                      file="../avl/delay_10_http_browsing_0.pcap"
                                      cps=1)
```

```
                                        ])                                         ❷


def register():
    return Prof1()
```

❶ Define the tuple generator range for client side and server side

❷ The template list with relative CPS (connection per second)

Start Trex Server in GTPU mode

```
sudo ./t-rex-64 -i --astf --gtpu 0
```

START TREX CLIENT WITH FOLLOWING LOGIC

- Connect to Trex server
- Load profile
- Update GTP header information for a client using API <update_tunnel_client_record>
  - Source ip of the tunnel
  - Destination ip of the tunnel
  - IP version of the tunnel
  - TEID (Tunnel endpoint identifier)
- Example is at: scripts/automation/trex_control_plane/interactive/trex/examples/astf/tunnel_test.py

2.4 Performance

see ASTF Performance

2.5 Port service mode

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

The **modes**:

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

Note
Filter mode is new from version v2.66

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

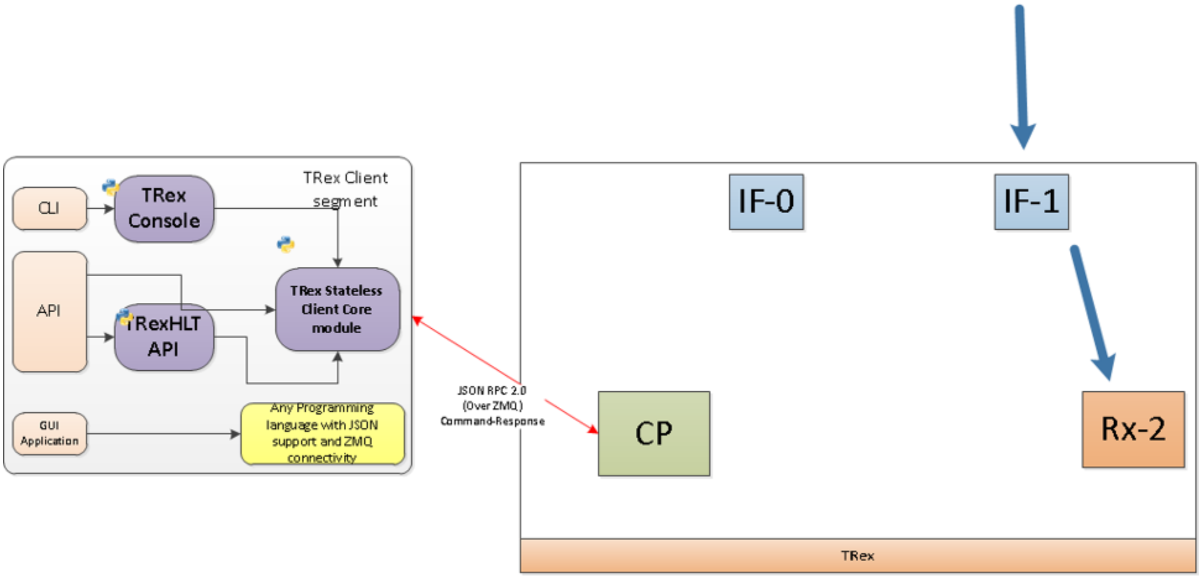


Figure 2.10: Port Under Normal Mode

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

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

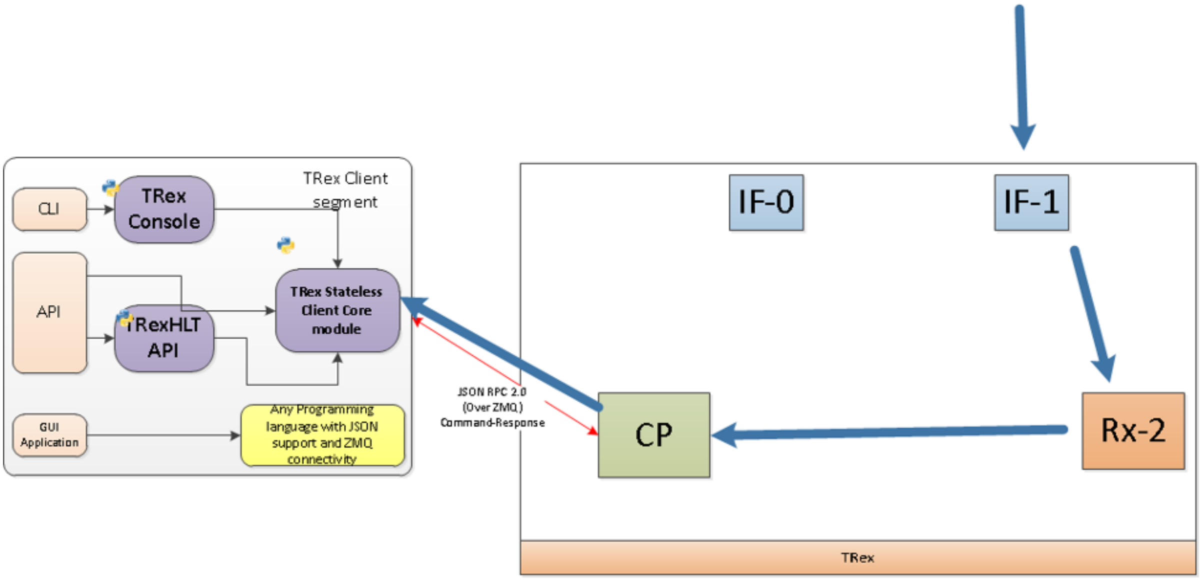


Figure 2.11: Port Under Service Mode

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

Example of switching between Service and Normal modes

```
trex>service --help
usage: service [-h] [-p PORTS [PORTS ...] | -a] [--bgp] [--dhcp] [--mdns]
               [--emu] [--tran] [--no-tcp-udp] [--all] [--off]

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

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

  --bgp                 filter mode with bgp packets forward to rx
  --dhcp                filter mode with dhcpv4/dhcpv6 packets forward to rx
  --mdns                filter mode with mDNS packets forward to rx
  --emu                 filter mode for all emu services rx
  --tran                filter mode with tcp/udp packets forward to rx
                        (generated by emu)
  --no-tcp-udp          filter mode with no_tcp_udp packets forward to rx
  --all                 Allow every filter possible
  --off                 Deactivates services on port(s)

trex>service

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

trex(service)>service --off

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

Example Of switching between Service and Normal modes: API

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

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

2.5.1 ARP / ICMP response

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

Table 2.1: Different types of service mode

Service mode type	No TCP UDP	BGP	TCP or UDP
On	Redirect to rx	Redirect to rx	Only when idle
Filtered	Redirect to rx	Redirect to rx only when BGP flag is on	Only when idle
Off (Default mode)	Redirect to rx	-	-

2.5.2 Filters for EMU

Emu can simulate different services, for example DHCP, DHCPv6, DNS, MDNS etc. Many of these protocols run of top of TCP and UDP. TRex needs a way to know if it should filter these services or not. This way it can know if a packet belongs to ASTF or Emu. For packets to be filtered to Emu we need to apply the filters:

- `dchp`: Forward DHCP packets to DHCP plugin in Emu
- `mdns`: Forward MDNS packets to MDNS plugin in Emu
- `tran`: Forward packets to Transport plugin in Emu. Transport plugin simulates UDP/TCP. Emu transport generated packets use high source ports: [0xFF00-0xFFFF]. In order to separate the Emu's TCP/UDP from ASTF's TCP/UDP this filter forwards all packets whose source **or** destination port is in the aforementioned ranges to Emu.
- `emu`: All of the previous flags together.

Caution
If you have applied the `trans` or `emu` filter, packets whose source or destination port is in [0xFF00-0xFFFF] will be forwarded to Emu, hence you will not get them in ASTF.

2.6 Client/Server only mode

With ASTF mode, it is possible to work in either client mode or server mode. This way TRex client side could be installed in one physical location on the network and TRex server in another.

Warning
We are in the process to move to interactive model, so the following ways to configure the C/S modes (as batch) is changing. This is a temporary solution and it going to be more flexible in the future. The roadmap is to give a RCP command to configure each port to client or server mode.

The current way to control the C/S mode is using the following CLI switch. There is only a way to disable the Client side ports for transmission, but there is no way to change the mode of each port.

Table 2.2: batch CLI options

CLI	Description
--astf-server-only	Only server side ports (1,3..) are enabled with ASTF service. Traffic won't be transmitted on clients ports.
--astf-client-mask [mask]	Enable only specific client side ports with ASTF service. 0x1 means to enable only port 0. 0x2 means to enable only port 2. 0x5 to enable port 0 and port 4.

Some examples:

Table 2.3: Normal mode, 4 ports --astf

Port id	Mode
0	C-Enabled
1	S-Enabled

Table 2.3: (continued)

Port id	Mode
2	C-Enabled
3	S-Enabled

Table 2.4: Normal mode, 4 ports --astf-server-only

Port id	Mode
0	**C-Disabled**
1	S-Enabled
2	**C-Disabled**
3	S-Enabled

Table 2.5: Normal mode, 4 ports --astf-client-mask 1

Port id	Mode
0	C-Enabled
1	S-Enabled
2	**C-Disabled**
3	S-Enabled

Table 2.6: Normal mode, 4 ports --astf-client-mask 2

Port id	Mode
0	**C-Disabled**
1	S-Enabled
2	C-Enabled
3	S-Enabled

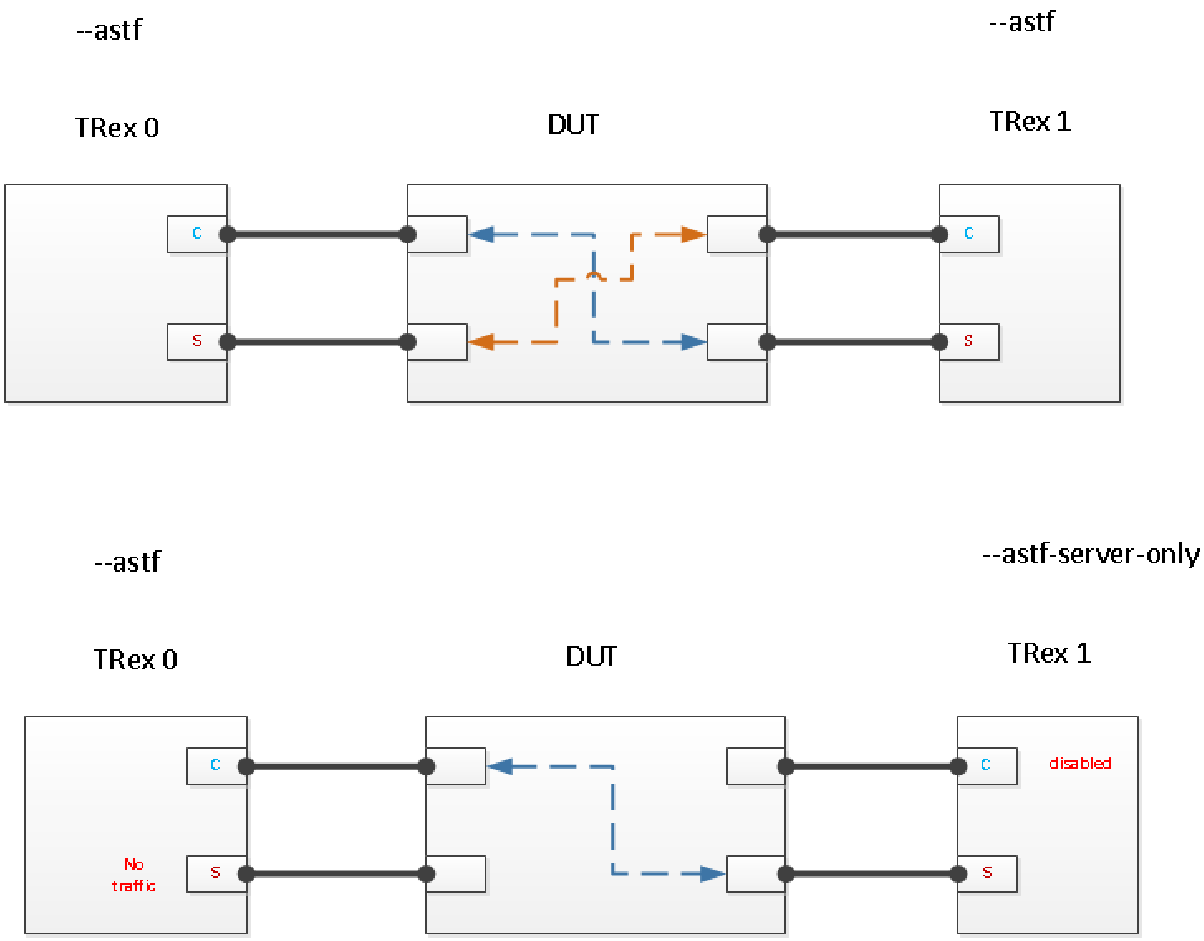


Figure 2.12: C/S modes examples

2.6.1 Limitation

- Latency (`-l` switch) should be disabled when working in this mode because both Client and Server will send ICMP packets to all the ports
- To configure TRex with all ports acting as client side or all ports acting as server side use "dummy" port configuration (see the main manual)

2.7 Client clustering configuration

2.7.1 Batch mode

TRex ASTF 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.

For more information about this feature have a look here Client clustering configuration

The format of the client cluster file is YAML, the same as STF mode, and it will be changed in interactive model.

To run simulation with this mode add `--cc` to the CLI:

Simulation 255 destination DUT

```
[bash]>./astf-sim -f astf/http_simple.py --full -o b.pcap --cc astf/cc_http_simple2.yaml
```

Simulation,Two virtual interfaces per one physical interface

```
[bash]>./astf-sim -f astf/http_simple.py --full -o b.pcap --cc astf/cc_http_simple.yaml
```

To run TRex add `--client_cfg` CLI :

Running TRex with client cluster file

```
[bash]>sudo ./t-rex-64 -f astf/http_simple.py -m 1000 -d 1000 -c 1 --astf -l 1000 -k 10 -- ↩
    client_cfg astf/cc_http_simple.yaml
```

Note

The `responder` information is ignored in ASTF mode as the server side learn the VLAN/MAC from the DUT. Another fact is the TRex ports is behaving like trunk ports with all VLAN allowed. This means that when working with a Switch, the Switch could flood the wrong packet (SYN with the wrong VLAN) to the TRex server side port and TRex will answer to this packet by mistake (as all VLAN are allowed, e.g. client side packet with client side VLAN will be responded in the server side). To overcome this either use ` allowed vlan` command in the Switch or use a dummy ARP resolution to make the Switch learn the VLAN on each port

---

2.7.2 Interactive mode

For interactive mode, Python profiles of topology can be used.
This topology consists of set of two objects:

- Virtual interface defining src MAC and optional VLAN. Not to be confused with linux vifs, topo vifs cannot answer to ARP. You can create a Linux network namespace or using our new TRex-emu nodes.
- Route table for traffic (GW). It can be applied for both TRex port or Virtual interface mentioned above.

Using this method along with linux network namespace (see STL spec for more info) it is possible to create a client side network with many clients on the same network each client with different:

1. Source MAC
2. Default gateway (and reoslved Destination MAC)
3. dot1q (qinq is not supported in ASTF)

The Linux network namespace will handle the ARP/IPv6 ND protocols requirement while traffic profile will take the src_mac/dst_mac(re
from the topo table — but this is only for Client side.

The server side learn the information from the traffic. The server side responds to src/dst IP traffic with the opposite ips (e.g. SYN from 10.0.0.1→40.0.0.1 will be answered by TRex 40.0.0.1→10.0.01) and it is not related to the client pool — as there could be NAT that change inside/outside. The server only cares about the TCP/UDP destination port to associate the port with the right L7 template. The MAC address used by the server side port are the L2 ports information (src == TRex MAC addr and dst == default GW). TRex in the server side won't learn and reverse the MAC as it does to IP in the current version.

So the servers pool should be next hop to the configured TRex IP/DG (e.g. trex ip= 1.1.1.1, dg=1.1.1.2 while pool of servers ips is 48.0.0.1 - 48.0.0.255). The Dot1q will be learned from the traffic too.

so back to the above example, this packet :

```
(MACA->MACB)  (10.0.0.1->40.0.0.1) TCP SYN
```

will be answered by TRex

```
(TRex SRC_MAC->TRex DST_MAC)  (40.0.0.1->10.0.0.1) TCP SYN
```

and **not**

```
(MACB -> MACA)  (40.0.0.1->10.0.0.1) TCP SYN
```

So to summerize: Topo is used for the client side. The server side learns dot1q/ip/ipv6 but not the MAC layer (uses TRex L2)

Topology profile example

```
from trex.astf.topo import *


def get_topo():                                      ❶
    topo = ASTFTopology()

# VIFs
    topo.add_vif(
        port_id = '0.2',                             ❷
        src_mac = '12:12:12:12:12:12',               ❸
        src_ipv4 = '5.5.5.5',                        ❹
        vlan = 30,                                   ❺
    )

# GWs

    topo.add_gw(
        port_id = '0.2',                             ❻
        src_start = '16.0.0.0',                      ❼
        src_end = '16.0.0.2',                        ❽
        dst = '45:45:45:45:45:45',                   ❾
    )

    topo.add_gw(
        port_id = '0',                               ❿
        src_start = '16.0.0.3',
        src_end = '16.0.0.4',
        dst = '2.2.2.2',                             ⓫
    )

    return topo
```

❶ Reserved name, must be used.

❷ Virtual interface ID, in this case it’s sub-interface of TRex port 0, sub-if ID 2

❸ Source MAC of VIF (required)

❹ Source IP of VIF, (optional, used for resolving dst MAC of GWs, if need resolve). In future versions, TRex will answer on this IP for ARP/Ping etc.

❺ VLAN of VIF (optional)

❻ This GW is used with VIF 0.2

❼, ❽ Range of traffic to apply current routing

❾ Ecplicit destination MAC for given range

❿ This GW is used with TRex port 0

⓫ Destination IP of this range, needs resolving

Note
In order to use custom src MAC for VIF and receive packets, need to apply promiscuous mode on TRex interface.

Using topology from console example

```
trex>portattr --prom on

Applying attributes on port(s) [0, 1]:                          [SUCCESS]

trex>topo load -f my_topo.py

trex>topo show
Virtual interfaces

Port |        MAC        | VLAN |      IPv4        | IPv6
-----+-------------------+------+-----------------+------
0.2  | 12:12:12:12:12:12 |  20  |     5.5.5.5     |   -

Gateways for traffic

Port  |   Range start    |    Range end     |        Dest       |     Resolved
------+------------------+------------------+-------------------+------------------
 0.2  |    16.0.0.0      |    16.0.0.1      | 45:45:45:45:45:45 | 45:45:45:45:45:45
 0.2  |    16.0.0.2      |    16.0.0.3      |      2.2.2.2      |         -
  0   |    16.0.0.4      |   16.0.0.255     |      2.2.2.2      |         -


trex>topo resolve
1 dest(s) resolved for 2 GW(s), uploading to server
240.41 [ms]

trex>topo show
Virtual interfaces

Port |        MAC        | VLAN |      IPv4        | IPv6
-----+-------------------+------+-----------------+------
0.2  | 12:12:12:12:12:12 |  20  |     5.5.5.5     |   -

Gateways for traffic

Port  |   Range start    |    Range end     |        Dest       |     Resolved
------+------------------+------------------+-------------------+------------------
 0.2  |    16.0.0.0      |    16.0.0.1      | 45:45:45:45:45:45 | 45:45:45:45:45:45
 0.2  |    16.0.0.2      |    16.0.0.3      |      2.2.2.2      | 00:0c:29:32:d3:4d
  0   |    16.0.0.4      |   16.0.0.255     |      2.2.2.2      | 00:0c:29:32:d3:4d


trex>start -f astf/udp1.py

Loading traffic at acquired ports.                              [SUCCESS]


Starting traffic.                                               [SUCCESS]

76.22 [ms]

trex>
```

Produced traffic (captured at port 1):

No.	Time	src MAC	dst MAC	Source	Destination
1	0.000000	12:12:12:12:12:12	45:45:45:45:45:45	16.0.0.0	48.0.0.0
2	0.000003	12:12:12:12:12:12	45:45:45:45:45:45	16.0.0.0	48.0.0.0
3	1.000022	12:12:12:12:12:12	45:45:45:45:45:45	16.0.0.1	48.0.0.1
4	1.000025	12:12:12:12:12:12	45:45:45:45:45:45	16.0.0.1	48.0.0.1
5	2.000005	12:12:12:12:12:12	00:0c:29:32:d3:4d	16.0.0.2	48.0.0.2
6	2.000007	12:12:12:12:12:12	00:0c:29:32:d3:4d	16.0.0.2	48.0.0.2
7	3.000041	12:12:12:12:12:12	00:0c:29:32:d3:4d	16.0.0.3	48.0.0.3
8	3.000043	12:12:12:12:12:12	00:0c:29:32:d3:4d	16.0.0.3	48.0.0.3
9	3.999982	00:0c:29:32:d3:43	00:0c:29:32:d3:4d	16.0.0.4	48.0.0.4
10	4.000000	00:0c:29:32:d3:43	00:0c:29:32:d3:4d	16.0.0.4	48.0.0.4
11	5.000022	00:0c:29:32:d3:43	00:0c:29:32:d3:4d	16.0.0.5	48.0.0.5
12	5.000024	00:0c:29:32:d3:43	00:0c:29:32:d3:4d	16.0.0.5	48.0.0.5

Figure 2.13: Produced traffic using topology

Note
This feature does not work in the astf-sim tool beacuse there is a need to resolve destination MAC

2.7.3 TRex emulation client clustering

2.7.3.1 Background

TRex emulation server is able to build clients network fast and in high scale. We will create a simple namespace on port 0 with 3 emu clients. We will generate traffic from clients through DUT back to second TRex port and all the way back.

Note
Namespace port must be even (client side)

Our setup will look like that:

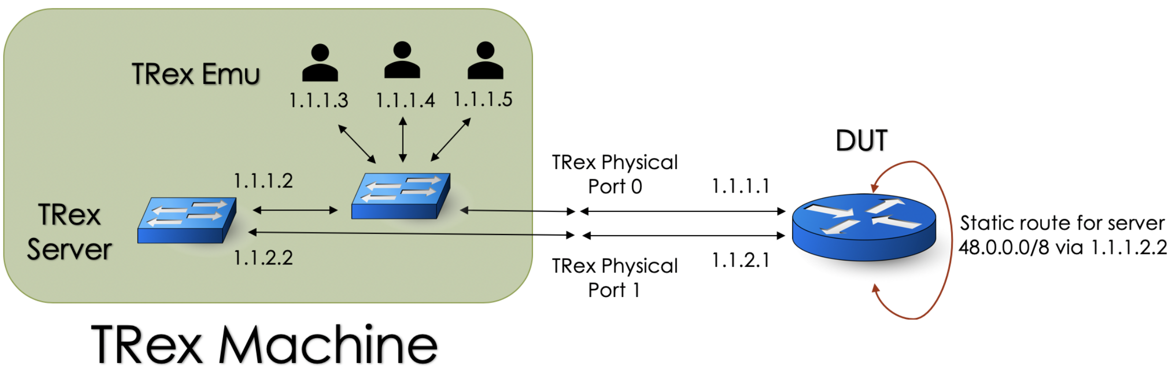


Figure 2.14: TRex Emu setup

2.7.3.2 Preparation

1) Starting TRex server with --astf mode and --emu.

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

2) In another shell, connect with trex console with --emu.

```
[bash]> ./trex-console --emu
```

3) Enable promiscuous mode on clients port/s.

```
trex> portattr -p 0 --prom on
```

4) Load emu profile with at least arp and icmp plugins, we will use 3 clients as follows:

simple_icmp.py emu profile

```
from trex.emu.api import *
import argparse


class Prof1():
    def __init__(self):
        self.def_ns_plugs  = {'icmp': {},
                              'arp' : {'enable': True}}

        self.def_c_plugs  = {'arp':  {'enable': True},
                             'icmp': {},
                             }

    def create_profile(self, ns_size, clients_size):
        ns_list = []

        # create different namespace each time
        vport, tci, tpid = 0, [0, 0], [0x00, 0x00]
        for i in range(vport, ns_size + vport):
            ns_key = EMUNamespaceKey(vport  = i,
                                     tci     = tci,
                                     tpid    = tpid)
            ns = EMUNamespaceObj(ns_key = ns_key, def_c_plugs = self.def_c_plugs)

            mac = Mac('00:00:00:70:00:03')
            ipv4 = Ipv4('1.1.1.3')
            dg = Ipv4('1.1.1.1')

            # create a different client each time
            for j in range(clients_size):
                client = EMUClientObj(mac     = mac[j].V(),
                                      ipv4    = ipv4[j].V(),
                                      ipv4_dg = dg.V())
                ns.add_clients(client)
            ns_list.append(ns)

        return EMUProfile(ns = ns_list, def_ns_plugs = self.def_ns_plugs)

    def get_profile(self, tuneables):
        # Argparse for tunables
        parser = argparse.ArgumentParser(description='Argparser for simple emu profile.')
        parser.add_argument('--ns', type = int, default = 1,
                    help='Number of namespaces to create')
        parser.add_argument('--clients', type = int, default = 15,
```

```
                        help='Number of clients to create in each namespace')

        args = parser.parse_args(tuneables)

        assert args.ns > 0, 'namespaces must be positive!'
        assert args.clients > 0, 'clients must be positive!'

        return self.create_profile(args.ns, args.clients)


def register():
    return Prof1()
```

Loading the emu profile with tuneables:

```
trex>emu_load_profile -f emu/simple_icmp.py -t --ns 1 --cl 3
trex>emu_show_all
Namespace #1 Information

Port | Vlan tags | Tpids |  Plugins   | #Clients
-----+-----------+-------+-----------+---------
 0   |     -     |   -   | arp, icmp |    3

Clients Information

       MAC         |  IPv4    | DG-IPv4 | MTU  |  Plugins
-------------------+---------+---------+------+----------
00:00:00:70:00:03 | 1.1.1.3 | 1.1.1.1 | 1500 | arp, icmp
00:00:00:70:00:04 | 1.1.1.4 | 1.1.1.1 | 1500 | arp, icmp
00:00:00:70:00:05 | 1.1.1.5 | 1.1.1.1 | 1500 | arp, icmp
```

5) Load `emu_astf` plugin and sync topo. It's necessary because TRex must change the client's mac addresses. It requires all clients to have a resolved default gateway mac. Notice how each client has it's own vif and gateway.

> **Note**
> `sync_topo` command will load client's information (**not the traffic**) from trex-emu into ASTF topo.

```
trex>plugins load emu_astf
trex>plugins emu_astf sync_topo

No need to resolve anything, uploading to server
emu topo synced

trex> topo show
None
[0, 1]
Virtual interfaces

Port |         MAC         | VLAN |       IPv4        | IPv6
-----+--------------------+------+------------------+------
0.1  | 00:00:00:70:00:03 |  -   |      1.1.1.3      |  ::
0.2  | 00:00:00:70:00:04 |  -   |      1.1.1.4      |  ::
0.3  | 00:00:00:70:00:05 |  -   |      1.1.1.5      |  ::

Gateways for traffic

Port   |   Range start    |    Range end     |        Dest        |      Resolved
------+-----------------+-----------------+-------------------+------------------
 0.1   |     1.1.1.3      |     1.1.1.3      | 00:00:00:01:00:02 | 00:00:00:01:00:02
```

```
0.2  |     1.1.1.4     |     1.1.1.4     | 00:00:00:01:00:02 | 00:00:00:01:00:02
0.3  |     1.1.1.5     |     1.1.1.5     | 00:00:00:01:00:02 | 00:00:00:01:00:02
```

6) Make sure your DUT is configured with no_pbr and static routes in order to ensure routing from one port to another.

2.7.4 Sending traffic

1) Start traffic where clients pool ranged: [1.1.1.3 - 1.1.1.5]. We will use http_simple_emu profile.

http_simple_emu profile

```
from trex.astf.api import *
import argparse


class Prof1():
    def __init__(self):
        pass

    def get_profile(self, tunables, **kwargs):
        parser = argparse.ArgumentParser(description='Argparser for {}'.format(os.path. ↩
            basename(__file__)),
                                         formatter_class=argparse. ↩
                                             ArgumentDefaultsHelpFormatter)

        args = parser.parse_args(tunables)
        # ip generator
        ip_gen_c = ASTFIPGenDist(ip_range=["1.1.1.3", "1.1.1.5"], distribution="seq")
        ip_gen_s = ASTFIPGenDist(ip_range=["48.0.0.0", "48.0.0.0"], distribution="seq")
        ip_gen = ASTFIPGen(glob=ASTFIPGenGlobal(ip_offset="1.0.0.0"),
                           dist_client=ip_gen_c,
                           dist_server=ip_gen_s)

        return ASTFProfile(default_ip_gen=ip_gen,
                            cap_list=[ASTFCapInfo(file="../avl/delay_10_http_browsing_0. ↩
                                pcap",
                            cps=2.776)])


def register():
    return Prof1()
```

Starting http_simple_emu profile

```
trex> start -f astf/http_simple_emu.py
```

Let's observe the capture on port 0:

	Time	MAC Src	MAC Dst	Source	Destination	Protocol	Length	Info
1	0.000000	00:00:00:70:00:03	00:00:00:01:00:02	1.1.1.3	48.0.0.0	TCP	74	41668 → 80 [SYN] Seq=0 Win=32768 [TCP CHECKS
2	0.001013	00:00:00:01:00:02	00:00:00:70:00:03	48.0.0.0	1.1.1.3	TCP	74	80 → 41668 [SYN, ACK] Seq=0 Ack=1 Win=32768
3	0.001015	00:00:00:70:00:03	00:00:00:01:00:02	1.1.1.3	48.0.0.0	HTTP	315	GET /3384 HTTP/1.1
4	0.003200	00:00:00:01:00:02	00:00:00:70:00:03	48.0.0.0	1.1.1.3	HTTP	15994	HTTP/1.1 200 OK (text/html)
5	0.003202	00:00:00:70:00:03	00:00:00:01:00:02	1.1.1.3	48.0.0.0	TCP	66	41668 → 80 [ACK] Seq=250 Ack=15929 Win=32768
6	0.005195	00:00:00:01:00:02	00:00:00:70:00:03	48.0.0.0	1.1.1.3	HTTP	15994	Continuation
7	0.005197	00:00:00:01:00:02	00:00:00:70:00:03	48.0.0.0	1.1.1.3	TCP	304	80 → 41668 [PSH, ACK] Seq=31857 Ack=250 Win=
8	0.005198	00:00:00:70:00:03	00:00:00:01:00:02	1.1.1.3	48.0.0.0	TCP	66	41668 → 80 [FIN, ACK] Seq=250 Ack=32095 Win=
9	0.007013	00:00:00:01:00:02	00:00:00:70:00:03	48.0.0.0	1.1.1.3	TCP	66	80 → 41668 [FIN, ACK] Seq=32095 Ack=251 Win=
10	0.007014	00:00:00:70:00:03	00:00:00:01:00:02	1.1.1.3	48.0.0.0	TCP	66	41668 → 80 [ACK] Seq=251 Ack=32096 Win=32768
11	0.360218	00:00:00:70:00:04	00:00:00:01:00:02	1.1.1.4	48.0.0.0	TCP	74	59073 → 80 [SYN] Seq=0 Win=32768 [TCP CHECKS
12	0.362013	00:00:00:01:00:02	00:00:00:70:00:04	48.0.0.0	1.1.1.4	TCP	74	80 → 59073 [SYN, ACK] Seq=0 Ack=1 Win=32768
13	0.362015	00:00:00:70:00:04	00:00:00:01:00:02	1.1.1.4	48.0.0.0	HTTP	315	GET /3384 HTTP/1.1
14	0.364455	00:00:00:01:00:02	00:00:00:70:00:04	48.0.0.0	1.1.1.4	HTTP	15994	HTTP/1.1 200 OK (text/html)
15	0.364457	00:00:00:70:00:04	00:00:00:01:00:02	1.1.1.4	48.0.0.0	TCP	66	59073 → 80 [ACK] Seq=250 Ack=15929 Win=32768
16	0.366455	00:00:00:01:00:02	00:00:00:70:00:04	48.0.0.0	1.1.1.4	HTTP	15994	Continuation
17	0.366456	00:00:00:01:00:02	00:00:00:70:00:04	48.0.0.0	1.1.1.4	TCP	304	80 → 59073 [PSH, ACK] Seq=31857 Ack=250 Win=
18	0.366457	00:00:00:70:00:04	00:00:00:01:00:02	1.1.1.4	48.0.0.0	TCP	66	59073 → 80 [FIN, ACK] Seq=250 Ack=32095 Win=
19	0.368013	00:00:00:01:00:02	00:00:00:70:00:04	48.0.0.0	1.1.1.4	TCP	66	80 → 59073 [FIN, ACK] Seq=32095 Ack=251 Win=
20	0.368014	00:00:00:70:00:04	00:00:00:01:00:02	1.1.1.4	48.0.0.0	TCP	66	59073 → 80 [ACK] Seq=251 Ack=32096 Win=32768
21	0.720456	00:00:00:70:00:05	00:00:00:01:00:02	1.1.1.5	48.0.0.0	TCP	74	10942 → 80 [SYN] Seq=0 Win=32768 [TCP CHECKS
22	0.722013	00:00:00:01:00:02	00:00:00:70:00:05	48.0.0.0	1.1.1.5	TCP	74	80 → 10942 [SYN, ACK] Seq=0 Ack=1 Win=32768
23	0.722014	00:00:00:70:00:05	00:00:00:01:00:02	1.1.1.5	48.0.0.0	HTTP	315	GET /3384 HTTP/1.1
24	0.724235	00:00:00:01:00:02	00:00:00:70:00:05	48.0.0.0	1.1.1.5	HTTP	15994	HTTP/1.1 200 OK (text/html)
25	0.724236	00:00:00:70:00:05	00:00:00:01:00:02	1.1.1.5	48.0.0.0	TCP	66	10942 → 80 [ACK] Seq=250 Ack=15929 Win=32768
26	0.726234	00:00:00:01:00:02	00:00:00:70:00:05	48.0.0.0	1.1.1.5	HTTP	15994	Continuation
27	0.726235	00:00:00:01:00:02	00:00:00:70:00:05	48.0.0.0	1.1.1.5	TCP	304	80 → 10942 [PSH, ACK] Seq=31857 Ack=250 Win=
28	0.726236	00:00:00:70:00:05	00:00:00:01:00:02	1.1.1.5	48.0.0.0	TCP	66	10942 → 80 [FIN, ACK] Seq=250 Ack=32095 Win=
29	0.728013	00:00:00:01:00:02	00:00:00:70:00:05	48.0.0.0	1.1.1.5	TCP	66	80 → 10942 [FIN, ACK] Seq=32095 Ack=251 Win=
30	0.728014	00:00:00:70:00:05	00:00:00:01:00:02	1.1.1.5	48.0.0.0	TCP	66	10942 → 80 [ACK] Seq=251 Ack=32096 Win=32768

Figure 2.15: TRex emu topo capture

Notice how TRex uses client's mac addresses (00:00:00:70:00:03-5) according to our topo.

Note
Currently DHCP plugin isn't supported because ip range must be continuous.

2.7.5 IPv6 Example

Our final setup will look like that:

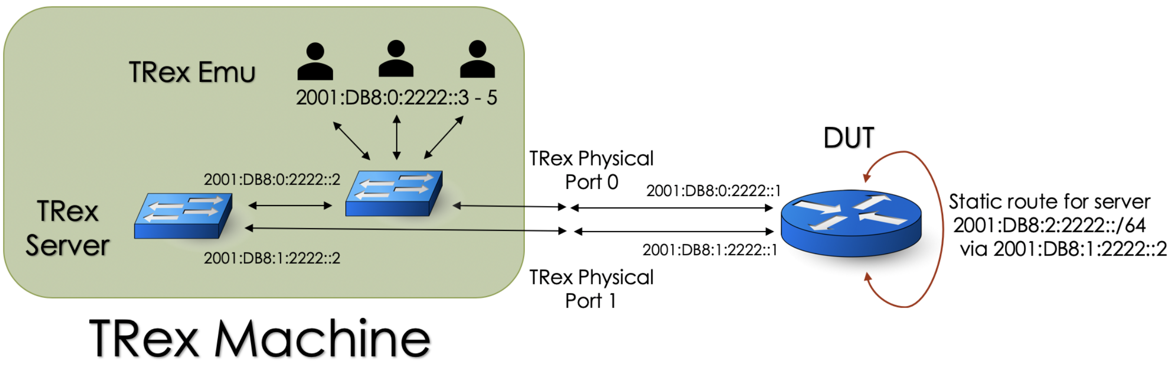


Figure 2.16: TRex emu topo capture

We can easily generate Ipv6 traffic as we did earlier with some minor changes:

1) Run TRex server on linux based stack mode and enable promiscuous mode at trex-console.

```
trex> portattr -a --prom on
```

2) Configure IPv6 on both ports in the same subnet of the DUT (2001:DB8:0:2222::/64 and 2001:DB8:1:2222::/64):

```
trex>ipv6 -p 0 -s 2001:DB8:0:2222::2

Configuring IPv6 on port 0                                     [SUCCESS]

trex>ipv6 -p 1 -s 2001:DB8:1:2222::2

Configuring IPv6 on port 1                                     [SUCCESS]
```

3) Add a static route at DUT for our server:

```
asr1001-01#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
asr1001-01(config)#ipv6 route 2001:DB8:2:2222::/64 2001:db8:1:2222::2
```

4) Load `simple_icmp.py` profile with IPv6 configured on same subnet (2001:db8::) and ipv6 plugin by default:

```
    def __init__(self):
        self.def_ns_plugs  = {'icmp': {},
                             'arp' : {'enable': True}}

        self.def_c_plugs  = {'arp':  {'enable': True},
                            'icmp': {}, 'ipv6': {},  # Must enable IPv6 plugin on all ↩
                                clients
                            }

    def create_profile(self, ns_size, clients_size, mac, ipv4, dg, ipv6):
        ns_list = []

        # create different namespace each time
        vport, tci, tpid = 0, [0, 0], [0x00, 0x00]
        for i in range(vport, ns_size + vport):
            ns_key = EMUNamespaceKey(vport  = i,
                                    tci     = tci,
                                    tpid    = tpid)
            ns = EMUNamespaceObj(ns_key = ns_key, def_c_plugs = self.def_c_plugs)

            mac  = Mac(mac)
            ipv4 = Ipv4(ipv4)
            dg   = Ipv4(dg)
            ipv6 = Ipv6("2001:DB8:0:2222::1.1.1.3")  # our first client

            # create a different client each time
            for j in range(clients_size):
                client = EMUClientObj(mac     = mac[j].V(),
                                      ipv4    = ipv4[j].V(),
                                      ipv4_dg = dg.V(),
                                      ipv6    = ipv6[j].V(),
                                      )
                ns.add_clients(client)
            ns_list.append(ns)

        return EMUProfile(ns = ns_list, def_ns_plugs = self.def_ns_plugs)
```

Loading sim

```
trex>emu_load_profile -f emu/simple_icmp.py -t --ns 1 --cl 3
trex>emu_show_all --6
Namespace #1 Information

Port | Vlan tags | Tpids | Plugins | #Clients
-----+-----------+-------+---------+---------
```

```
 0   |     -     |   -   |  ipv6   |    2

Clients Information

       MAC         |     IPv6      |      IPv6-Local       |           IPv6-Slaac            | ↩
        Plugins
------------------+-------------+---------------------+---------------------------------+-------- ↩

00:00:00:70:00:01 | 2001:db8::3 | fe80::200:ff:fe70:1 | 2001:db8:0:2222:200:ff:fe70:1 |   ↩
    ipv6
00:00:00:70:00:02 | 2001:db8::4 | fe80::200:ff:fe70:2 | 2001:db8:0:2222:200:ff:fe70:2 |   ↩
    ipv6
00:00:00:70:00:03 | 2001:db8::5 | fe80::200:ff:fe70:3 | 2001:db8:0:2222:200:ff:fe70:3 |   ↩
    ipv6
```

5) Load `emu_astf` plugin and sync topo as we did in IPv4:

```
trex>plugins load emu_astf

Loading plugin: emu_astf                                                     [SUCCESS]

trex>plugins emu_astf sync_topo

No need to resolve anything, uploading to server
emu topo synced
```

6) You can generate IPv6 traffic using one of the methods:

- Start command with "--ipv6", i.e: "start -f astf/http_simple_emu.py --ipv6" this will generate ips: "::x.x.x.x" where LSB is the IPv4 addrees.
- Use tunables in your profile for altering src / dst MSB as described bellow.

For more information about IPv6 in ASTF check Tutorial: IPv6 traffic

We will use the second method for changing the MSB correctly:

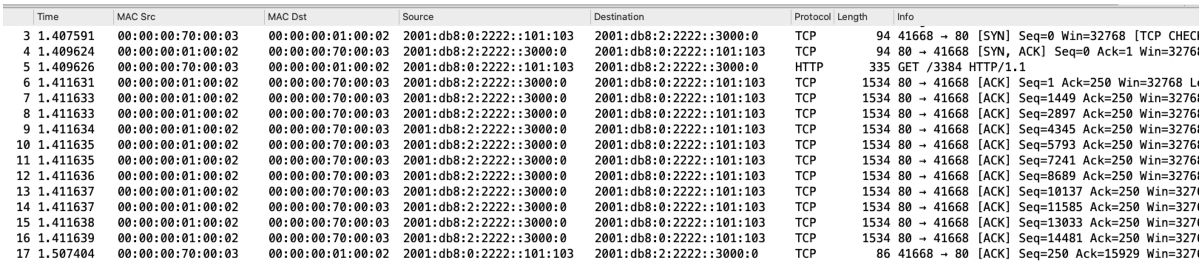
http_simple_emu_ipv6

```
from trex.astf.api import *
import argparse


class Prof1():
    def __init__(self):
        pass

    def get_profile(self, tunables, **kwargs):
        parser = argparse.ArgumentParser(description='Argparser for {}'.format(os.path. ↩
            basename(__file__)),
                                         formatter_class=argparse. ↩
                                             ArgumentDefaultsHelpFormatter)

        args = parser.parse_args(tunables)
        # ip generator
        ip_gen_c = ASTFIPGenDist(ip_range=["1.1.1.3", "1.1.1.5"], distribution="seq")
        ip_gen_s = ASTFIPGenDist(ip_range=["48.0.0.0", "48.0.0.0"], distribution="seq")
        ip_gen = ASTFIPGen(glob=ASTFIPGenGlobal(ip_offset="1.0.0.0"),
                           dist_client=ip_gen_c,
                           dist_server=ip_gen_s)

        c_glob_info = ASTFGlobalInfo()
```

```
        # Enable IPV6 for client side and set the default SRC/DST IPv6 MSB
        # LSB will be taken from ip generator
        c_glob_info.ipv6.src_msb = "2001:DB8:0:2222::"
        c_glob_info.ipv6.dst_msb = "2001:DB8:2:2222::"
        c_glob_info.ipv6.enable  = 1

        return ASTFProfile(default_ip_gen=ip_gen,
                           default_c_glob_info=c_glob_info,
                           cap_list=[ASTFCapInfo(file="../avl/delay_10_http_browsing_0. ↩
                               pcap",
                           cps=2.776)])


def register():
    return Prof1()
```

```
trex>start -f astf/http_simple_emu_ipv6.py
```

Let's observe the capture on port 0:



	Time	MAC Src	MAC Dst	Source	Destination	Protocol	Length	Info
3	1.407591	00:00:00:70:00:03	00:00:00:01:00:02	2001:db8:0:2222::101:103	2001:db8:2:2222::3000:0	TCP	94	41668 → 80 [SYN] Seq=0 Win=32768 [TCP CHEC
4	1.409624	00:00:00:01:00:02	00:00:00:70:00:03	2001:db8:2:2222::3000:0	2001:db8:0:2222::101:103	TCP	94	80 → 41668 [SYN, ACK] Seq=0 Ack=1 Win=3276
5	1.409626	00:00:00:70:00:03	00:00:00:01:00:02	2001:db8:0:2222::101:103	2001:db8:2:2222::3000:0	HTTP	335	GET /3384 HTTP/1.1
6	1.411631	00:00:00:01:00:02	00:00:00:70:00:03	2001:db8:2:2222::3000:0	2001:db8:0:2222::101:103	TCP	1534	80 → 41668 [ACK] Seq=1 Ack=250 Win=32768 L
7	1.411633	00:00:00:01:00:02	00:00:00:70:00:03	2001:db8:2:2222::3000:0	2001:db8:0:2222::101:103	TCP	1534	80 → 41668 [ACK] Seq=1449 Ack=250 Win=3276
8	1.411633	00:00:00:01:00:02	00:00:00:70:00:03	2001:db8:2:2222::3000:0	2001:db8:0:2222::101:103	TCP	1534	80 → 41668 [ACK] Seq=2897 Ack=250 Win=3276
9	1.411634	00:00:00:01:00:02	00:00:00:70:00:03	2001:db8:2:2222::3000:0	2001:db8:0:2222::101:103	TCP	1534	80 → 41668 [ACK] Seq=4345 Ack=250 Win=3276
10	1.411635	00:00:00:01:00:02	00:00:00:70:00:03	2001:db8:2:2222::3000:0	2001:db8:0:2222::101:103	TCP	1534	80 → 41668 [ACK] Seq=5793 Ack=250 Win=3276
11	1.411635	00:00:00:01:00:02	00:00:00:70:00:03	2001:db8:2:2222::3000:0	2001:db8:0:2222::101:103	TCP	1534	80 → 41668 [ACK] Seq=7241 Ack=250 Win=3276
12	1.411636	00:00:00:01:00:02	00:00:00:70:00:03	2001:db8:2:2222::3000:0	2001:db8:0:2222::101:103	TCP	1534	80 → 41668 [ACK] Seq=8689 Ack=250 Win=3276
13	1.411637	00:00:00:01:00:02	00:00:00:70:00:03	2001:db8:2:2222::3000:0	2001:db8:0:2222::101:103	TCP	1534	80 → 41668 [ACK] Seq=10137 Ack=250 Win=327
14	1.411637	00:00:00:01:00:02	00:00:00:70:00:03	2001:db8:2:2222::3000:0	2001:db8:0:2222::101:103	TCP	1534	80 → 41668 [ACK] Seq=11585 Ack=250 Win=327
15	1.411638	00:00:00:01:00:02	00:00:00:70:00:03	2001:db8:2:2222::3000:0	2001:db8:0:2222::101:103	TCP	1534	80 → 41668 [ACK] Seq=13033 Ack=250 Win=327
16	1.411639	00:00:00:01:00:02	00:00:00:70:00:03	2001:db8:2:2222::3000:0	2001:db8:0:2222::101:103	TCP	1534	80 → 41668 [ACK] Seq=14481 Ack=250 Win=327
17	1.507404	00:00:00:70:00:03	00:00:00:01:00:02	2001:db8:0:2222::101:103	2001:db8:2:2222::3000:0	TCP	86	41668 → 80 [ACK] Seq=250 Ack=15929 Win=327

Figure 2.17: TRex emu topo capture

Notice the mac addresses changed as well to our client's mac.

2.7.6 Scaling clients

You can scale your clients in order to make a stress test, just notice:

- DUT interfaces might require changes in their subnet mask in order to avoid ip conflicts.
- TRex cfg might change as well for the same reason.

2.8 Multi core support

There are two distribution models:

- Hardware: Using RSS hardware assist (only for NICs that support that). Using this feature it is possible to support up to 200Gb/sec of TCP/UDP traffic with one server.
- Software: (from v2.87) Using the driver default distribution (e.g. CRC) and fixing it using software. This has a compact performance impact. Example of drivers AWS Azure and KVM virtio.

The hardware RSS mode can work only for NICs that support RSS (almost all the physical NICs) and only for TCP/UDP traffic without tunnels (e.g. GRE). Adding --software will force any driver (including those that support hardware RSS) to work in software mode, hence use this only when you know it is needed.

Table 2.7: NICs that support RSS

Chipset	support	vlan skip	qinq skip
Intel 82599	+	-	-
Intel 710	+	+	-
Mellanox ConnectX-5/4	+	+	-
Napatech	-	-	-

- IPv6 and IPv4 are supported. For IPv6 to work there is a need that the server side IPv6 template e.g. info.ipv6.dst_msb ="ff03::" will have zero in bits 32-40 (LSB is zero).

ipv6.dst_msb should have one zero

```
                  this should be zero
                         ||
ipv6: 3ffe:501:8::260:97ff:fe40:efab  #<< not supported
ipv6: 3ffe:501:8::260:9700:fe40:efab  #<< supported
```

Note
When using RSS, the number of sockets (available source ports per clients IP) will be 100% divided by the numbers of cores (c). For example for c=3 each client IP would have 64K/3 source ports = 21K ports. In this case it is advised to add more clients to the pool (in case you are close to the limit). In other words the reported socket-util is the console should be multiplied by c.

2.8.1 Software mode with multicore

Multi queue NICs distribute incoming traffic to a specific queue based on a hash function. Each receive queue is assigned to a separate interrupt; by routing each of those interrupts to different CPUs or CPU cores. The hardware-based distribution of the interrupts, described above, is referred to as receive-side scaling RSS.

However, for NICs that don't support RSS as we aforementioned, the distribution is incorrect. Client side packets from a specific flow can arrive at some core which doesn't have any flow information. These packets need to be redirected to the core they belong to, meaning the core that holds the flow information. When using software mode with multiple core, these packets will be automatically redirected, and appropriate counters will be added to the flow table.

In short, the redirection to the correct core is done using the source port of L4. When a flow is generated, the source port of that flow is aligned to the core identifier that generated the flow. When a packet is received, based on the port, we can unravel the destination core and redirect the packet to it.

Naturally, all this logic has a performance impact. Fortunately, based on our tests, the impact is minor. This offers the possibility to scale in software mode as well. When reaching very high scale, a redirect drop counter will show in the flow table counters. This implies that the cores can't keep up with redirecting at the actual rate.

2.9 Dynamic multiple profiles support

TRex can support adding/removing multiple profiles dynamically in ASTF interactive mode while traffic is active.+

In the console, profile ids can be specified via --pid argument.
The expression in TRex console allows --pid <profile id>.
For example, start template_groups.py profile with profile id *template* and http_simple.py profile with profile_id *http*.
In this way, *template* and *http* profile can be running simultaneously.
On the other hand, if profile id is not specified, it means the default profile_id "_" for backward compatibility.

Most features are managed in profiles except for globally managed global tunables (TCP / IP protocol parameters), topology (multiple target MACs for multiple DUTs), and latency checking. Multiple tunnels, topology, and latency are ignored.

Example

```
# start(add) template_groups.py profile (profile_id : default profile "_")
trex> start -f astf/udp_mix.py
or
trex> start -f astf/udp_mix.py --pid _

# start(add) template_groups.py profile (profile_id : template)
trex> start -f astf/template_groups.py --pid template

# start(add) http_simple.py profile (profile_id : http)
trex> start -f astf/http_simple.py --pid http

# show all profiles
trex> profiles
Profile States

         ID          |        State
---------------------+---------------------
        http         |       STATE_TX
      template       |       STATE_TX
         _           |       STATE_TX

# update traffic rate(profile_id : default profile "_")
trex> update -m 2

# update traffic rate for template profile
trex> update --pid template -m 2

# update traffic rate for all profiles "*"
trex> update --pid * -m 2

# get template_group names (profile_id : template)
trex> template_group names --pid template

# get stats for template_group names(profile_id : template)
trex> template_group stats --name 1x --pid template

# get astf stats(profile_id : http)
trex> stats -a --pid http

# get astf total stats for all profiles
trex> stats -a


# stop default profile "_"
trex> stop

# stop and remove for template profile, http profile is still running
trex> stop --pid template --remove

# show all profiles
trex> profiles
Profile States

         ID          |        State
---------------------+---------------------
        http         |       STATE_TX
         _           |  STATE_ASTF_LOADED

# stop all profiles (asterisk "*")
```

```
trex> stop --pid *

# show all profiles
trex> profiles
Profile States

         ID           |         State
----------------------+---------------------
        http          |   STATE_ASTF_LOADED
         _            |   STATE_ASTF_LOADED

# reset(stop and clear) all profiles, default profile will be IDLE state for backward ↩
    compatibility
trex> reset

# show all profiles
trex> profiles
Profile States

         ID           |         State
----------------------+---------------------
         _            |      STATE_IDLE
```

2.10 Traffic profile reference

python index

2.11 Tunables reference

tunable	min-max	default	per-template	global (profile)	Description
ip.tos	uint8	0	+	+	ipv4/ipv6 TOS/Class. limitation LSB can't be set as it is used by hardware filter on some NICs
ip.ttl	uint8	0	+	+	ipv4/ipv6 TTL/TimeToLive. limitation can't be higher than 0x7f as it might used by some hardware filter on some NICs
ip.dont_use_inbound_mac	bool	0	-	+	server side will use configured src/dest MACs for the port or resolved gateway MAC instead of reflecting incoming MACs
ipv6.src_msb	string	0	+	+	default IPv6.src MSB address. see Priority in ipv6 enable
ipv6.dst_msb	string	0	+	+	default IPv6.dst MSB address. see Priority in ipv6 enable
ipv6.enable	bool	0	+	+	enable IPv6 for all templates. Priority is given for global (template come next) — not as other tunable. It means that for mix of ipv4/ipv6 only per template should be used.
tcp.mss	10-9K	1460	+	+	default MSS in bytes.
tcp.initwnd	1-20	10	+	+	init window value in MSS units.
tcp.rxbufsize	1K-1G	32K	+	+	socket rx buffer size in bytes.

tunable	min-max	default	per-template	global (pro-file)	Description
tcp.txbufsize	1K-1G	32K	+	+	socket tx buffer size in bytes.
tcp.rexmtthresh	1-10	3	-	+	number of duplicate ack to trigger retransimtion.
tcp.do_rfc1323	bool	1	-	+	enable timestamp rfc 1323.
tcp.keepinit	2-65533	5	-	+	value in second for TCP keepalive.
tcp.keepidle	2-65533	5	-	+	value in second for TCP keepidle
tcp.keepintvl	2-65533	7	-	+	value in second for TCP keepalive interval
tcp.blackhole	0,1,2	0	-	+	0 - return RST packet in case of error. 1- return of RST only in SYN. 2- don't return any RST packet, make a blackhole.
tcp.delay_ack_msec	20-50000	100	-	+	delay ack timeout in msec. Reducing this value will reduce the performance but will reduce the active flows
tcp.no_delay	0-3	0	+	+	In case of 1 disable Nagle and force PUSH. from v2.79 it was changed and there are two bits 1- disable nagle, 0x2 force PUSH flag (**NOT** standard it just to simulate Spirent) and will respond with ACK immediately (standard).
tcp.no_delay_counter	0-65533	0	+	+	number of recv bytes to wait until ack is sent. notice ack can be triggered by tcp timer, in order to ensure fixed number of sent packets until ack you should increase the **tcp.initwnd** tunable. otherewise no_delay_counter will race with the tcp timer.
scheduler.rampup_sec	3-60000	disabled	-	+	scheduler rampup in seconds. After this time the throughput would be the maximum. the throughput increases linearly every 1 sec.

- There would be a performance impact when per-template tunables are used. Try to use global tunables.
- The tunables mechanism does not work with the basic simulator mode but only with the advanced simulator mode.
- Please use the same tcp.blackhole value for all your profiles. Otherwise, the result will be unpredictable.

2.12 Counters reference

- Client side aggregates the ASTF counters from all client ports in the system (e.g. 0/2/4)
- Server side aggregates the ASTF counters from all server ports in the system (e.g. 1/3/5)

Table 2.8: General counters

Counter	Error	Description
active_flows		active flows (established + non-established) UDP/TCP
est_flows		active established flows UDP/TCP
tx_bw_l7		tx acked L7 bandwidth in bps TCP
tx_bw_l7_total		tx total L7 bandwidth sent in bps TCP
rx_bw_l7		rx acked L7 bandwidth in bps TCP

Table 2.8: (continued)

Counter	Error	Description
tx_pps_r		tx bandwidth in pps (TSO packets) **1** TCP
rx_pps_r		rx bandwidth in pps (LRO packets) **2** TCP
avg_size		average pkt size (base on TSO/LRO) see **1/2** TCP
tx_ratio		ratio betwean tx_bw_l7_r and tx_bw_l7_total_r 100% means no retransmition TCP

Table 2.9: TCP counters

Counter	Error	Description
tcps_connattempt		connections initiated
tcps_accepts		connections accepted
tcps_connects		connections established
tcps_closed		conn. closed (includes drops) - this counter could be higher than tcps_connects for client side as flow could be be dropped before establishment
tcps_segstimed		segs where we tried to get rtt
tcps_rttupdated		times we succeeded
tcps_delack		delayed acks sent
tcps_sndtotal		total packets sent (TSO)
tcps_sndpack		data packets sent (TSO)
tcps_sndbyte		data bytes sent by application
tcps_sndbyte_ok		data bytes sent by tcp layer could be more than tcps_sndbyte (asked by application)
tcps_sndctrl		control (SYN,FIN,RST) packets sent
tcps_sndacks		ack-only packets sent
tcps_rcvtotal		total packets received (LRO)
tcps_rcvpack		packets received in sequence (LRO)
tcps_rcvbyte		bytes received in sequence
tcps_rcvackpack		rcvd ack packets (LRO) **2**
tcps_rcvackbyte		tx bytes acked by rcvd acks (should be the same as tcps_sndbyte)
tcps_rcvackbyte_of		tx bytes acked by rcvd acks -overflow ack
tcps_preddat		times hdr predict ok for data pkts
tcps_drops	*	connections dropped
tcps_conndrops	*	embryonic connections dropped
tcps_timeoutdrop	*	conn. dropped in rxmt timeout
tcps_rexmttimeo	*	retransmit timeouts
tcps_persisttimeo	*	persist timeouts
tcps_keeptimeo	*	keepalive timeouts
tcps_keepprobe	*	keepalive probes sent
tcps_keepdrops	*	connections dropped in keepalive
tcps_sndrexmitpack	*	data packets retransmitted
tcps_sndrexmitbyte	*	data bytes retransmitted
tcps_sndprobe		window probes sent
tcps_sndurg		packets sent with URG only
tcps_sndwinup		window update-only packets sent
tcps_rcvbadoff	*	packets received with bad offset
tcps_rcvshort	*	packets received too short
tcps_rcvduppack	*	duplicate-only packets received
tcps_rcvdupbyte	*	duplicate-only bytes received
tcps_rcvpartduppack	*	packets with some duplicate data
tcps_rcvpartdupbyte	*	dup. bytes in part-dup. packets
tcps_rcvoopackdrop	*	OOO packet drop due to queue len
tcps_rcvoobytesdrop	*	OOO bytes drop due to queue len
tcps_rcvoopack	*	out-of-order packets received

Table 2.9: (continued)

Counter	Error	Description
tcps_rcvoobyte	*	out-of-order bytes received
tcps_rcvpackafterwin	*	packets with data after window
tcps_rcvbyteafterwin	*	,"bytes rcvd after window
tcps_rcvafterclose	*	packets rcvd after close
tcps_rcvwinprobe		rcvd window probe packets
tcps_rcvdupack	*	rcvd duplicate acks
tcps_rcvacktoomuch	*	rcvd acks for unsent data
tcps_rcvwinupd		rcvd window update packets
tcps_pawsdrop	*	segments dropped due to PAWS
tcps_predack	*	times hdr predict ok for acks
tcps_persistdrop	*	timeout in persist state
tcps_badsyn	*	bogus SYN, e.g. premature ACK
tcps_reasalloc	*	allocate tcp reasembly ctx
tcps_reasfree	*	free tcp reasembly ctx
tcps_nombuf	*	no mbuf for tcp - drop the packets

Table 2.10: UDP counters

Counter	Error	Description
udps_accepts	*	connections accepted
udps_connects	*	connections established
udps_closed	*	conn. closed (including drops)
udps_sndbyte	*	data bytes transmitted
udps_sndpkt	*	data packets transmitted
udps_rcvbyte	*	data bytes received
udps_rcvpkt	*	data packets received
udps_keepdrops	*	keepalive drop
udps_nombuf	*	no mbuf
udps_pkt_toobig	*	packets transmitted too big

Table 2.11: Flow table counters

Counter	Error	Description
err_cwf	*	client pkt that does not match a flow could no happen in loopback. Could happen if DUT generated a packet after TRex close the flow
err_no_syn	*	server first flow packet with no SYN
err_len_err	*	pkt with L3 length error
err_no_tcp	*	no tcp packet- dropped
err_no_template	*	server can't match L7 template no destination port or IP range
err_no_memory	*	no heap memory for allocating flows
err_dct	*	duplicate flows due to aging issues and long delay in the network
err_l3_cs	*	ipv4 checksum error
err_l4_cs	*	tcp/udp checksum error (in case NIC support it)
err_redirect_rx	*	redirect to rx error
redirect_rx_ok		redirect to rx OK
err_rx_throttled		rx thread was throttled due too many packets in NIC rx queue
err_c_nf_throttled		Number of client side flows that were not opened due to flow-table overflow. (to enlarge the number see the trex_cfg file for dp_max_flows)

Table 2.11: (continued)

Counter	Error	Description
err_s_nf_throttled		Number of server side flows that were not opened due to flow-table overflow. (to enlarge the number see the trex_cfg file for dp_max_flows)
err_s_nf_throttled		Number of too many flows events from maintenance thread. It is **not** the number of flows that weren't opened
err_c_tuple_err		How many flows were **not** opened in the client side because there were not enough clients in the pool. When this counter is reached, the TRex performance is affected due to the lookup for free ports. To solve this issue try adding more clients to the pool.
rss_redirect_rx		Number of RSS redirected packets that were received from other cores.
rss_redirect_tx		Number of RSS packets that were redirected to other cores.
rss_redirect_drop	*	Number of RSS packets that were meant to be redirected to other cores but were dropped since other cores buffers were full.

1

See TSO, we count the number of TSO packets with NICs that support that, this number could be significantly smaller than the real number of packets

2

see LRO, we count the number of LRO packets with NICs that support that, this number could be significantly smaller than the real number of packets

Important information

- It is hard to compare the number of TCP tx (client) TSO packets to rx (server) LRO packets as it might be different. The better approach would be to compare the number of bytes
  - client.tcps_rcvackbyte == server.tcps_rcvbyte and vice versa (upload and download)
  - tcps_sndbyte == tcps_rcvackbyte only if the flow were terminated correctly (in other words what was put in the Tx queue was transmitted and acked)
- Total Tx L7 bytes are tcps_sndbyte_ok+tcps_sndrexmitbyte+tcps_sndprobe
- The Console/JSON does not show/sent zero counters

Pseudo code tcp counters

```
if ( (c->tcps_drops ==0) &&
     (s->tcps_drops ==0) ){
   /* flow wasn't initiated due to drop of SYN too many times */

/* client side */
assert(c->tcps_sndbyte==UPLOAD_BYTES);
assert(c->tcps_rcvbyte==DOWNLOAD_BYTES);
assert(c->tcps_rcvackbyte==UPLOAD_BYTES);

/* server side */
assert(s->tcps_rcvackbyte==DOWNLOAD_BYTES);
assert(s->tcps_sndbyte==DOWNLOAD_BYTES);
assert(s->tcps_rcvbyte==UPLOAD_BYTES);
}
```

Some rules for counters:

```
/* for client side */
  1. tcps_connects<=tcps_closed      /* Drop before socket is connected will be counted in ←
      different counter and will be counted in tcps_closed but not in tcps_connects */
  2. tcps_connattempt==tcps_closed

/* for server side */
1. tcps_accepts=tcps_connects=tcps_closed
```

2.12.1 TSO/LRO NIC support

See manual.

2.13 FAQ

2.13.1 Why should I use TRex in this mode?

ASTF mode can help solving the following requirements:

- Test realistic scenario on top of TCP when DUT is acting like TCP proxy
- Test realistic scenario in high scale (flows/bandwidth)
- Flexibility to change the TCP/IP flow option
- Flexibility to emulate L7 application using Python API (e.g. Create many types of HTTP with different user-Agent field)
- Measure latency in high resolution (usec)

2.13.2 Why do I need to reload TRex server again with different flags to change to ASTF mode, In other words, why STL and ASTF can't work together?

In theory, we could have supported that, but it required much more effort because the NIC memory configuration is fundamentally different. For example, in ASTF mode, we need to configure all the Rx queues to receive the packets and to configure the RSS to split the packets to different interface queues. While in Stateful we filter most of the packets and count them in hardware.

2.13.3 Is your core TCP implementation based on prior work?

Yes, BSD4.4-Lite version of TCP with a bug fixes from freeBSD and our changes for scale of high concurrent flow and performance. The reasons why not to develop the tcp **core** logic from scratch can be found here Why do we use the Linux kernel's TCP stack?

2.13.4 What TCP RFCs are supported?

- RFC 793
- RFC 1122
- RFC 1323
- RFC 6928

Not implemented:

- RFC 2018

2.13.5 Could you have a more recent TCP implementation?

Yes, BSD4.4-Lite is from 1995 and does not have RFC 2018. We started as a POC and we plan to merge the latest freeBSD TCP core with our work.

2.13.6 Can I reduce the active flows with ASTF mode, there are too many of them?

The short answer is no. The active (concurrent) flows derived from RTT and responses of the tested network/DUT. You can **increase** the number of active flows by adding delay command.

2.13.7 Will NAT64 work in ASTF mode?

Yes. See IPv6 in the manual. Server side will handle IPv4 sockets

client side NAT64 server side

```
IPv6           ->         IPv4
IPv6           <-         IPv4
```

example

```
     client side IPv6
xx::16.0.0.1->yy::48.0.0.1
                         DUT convert it to IPv4

                        16.0.0.1->48.0.0.1
DUT convert it to IPv6

16.0.0.1<-48.0.0.1
```

xx::16.0.0.1←yy::48.0.0.1

client works in IPV6 server works on IPv4

2.13.8 Is TSO/LRO NIC hardware optimization supported?

Yes. LRO improves the performance. GRO is not there yet.

2.13.9 Can I get the ASTF counters per port/template?

Curently the TCP/App layer counters are per client/server side. We plan to add it in the interactive mode with RPC API.

2.14 Appendix

2.14.1 Blocking vs non-blocking

Let’s simulate a very long HTTP download session with `astf-sim` to understand the difference betwean blocking and non-blocking

1. rtt= 10msec
2. shaper rate is 10mbps (simulate egress shaper of the DUT)

3. write in chucks of 40KB
4. max-window = 24K

BDP (10msec*10mbps=12.5KB) but in case of blocking we will wait the RTT time in idle, reducing the maximum throughput.

Send is block

```
[bash]>./astf-sim -f astf/http_eflow2.py -o ab_np.pcap -r -v -t win=24,size=40,loop=100, ↩
    pipe=0 --cmd "shaper-rate=10000,rtt=10000"
```

In this case the rate is ~7mbps lower than 10mbps due to idle time.

Simulate is non-blocking

```
[bash]>./astf-sim -f astf/http_eflow2.py -o ab_np.pcap -r -v -t win=24,size=40,loop=100, ↩
    pipe=1 --cmd "shaper-rate=10000,rtt=10000"
```

In this case the rate is 10mbps (maximum expected) full-pipeline