

TRex Advance stateful support

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	Audience	1
2	Advance Stateful support	2
2.1	Feature overview	2
2.1.1	Status	3
2.1.1.1	What works in the current version	4
2.1.1.2	Current high priority tasks	4
2.1.1.3	On radar	4
2.1.2	Can we address the above requirements using existing DPDK TCP stacks?	5
2.1.3	The main properties of scalable TCP for traffic generation	5
2.1.4	Tx Scale to millions of flows	7
2.1.5	Rx Scale to millions of flows	8
2.1.6	Simulation of latency/jitter/drop in high scale	9
2.1.7	Emulation of L7 application	9
2.1.8	Stateful(STF) vs Advance Stateful (ASTF)	10
2.2	ASTF package folders	10
2.3	Getting started Tutorials	10
2.3.1	Tutorial: Prepare TRex configuration file	10
2.3.2	Tutorial: Run TRex with simple HTTP profile	10
2.3.3	Tutorial: Profile with two templates	14
2.3.4	Tutorial: Profile with two templates same ports	14
2.3.5	Tutorial: Profile with two range of tuple generator	15
2.3.6	Tutorial: IPv6 traffic	16
2.3.7	Tutorial: Change tcp.mss using tunables mechanism	17
2.3.8	Tutorial: Python automation	19
2.3.9	Tutorial: Simple simulator	22
2.3.10	Tutorial: Advanced simulator	25
2.3.11	Tutorial: Manual L7 emulation program building	25
2.3.12	Tutorial: Profile CLI tunable	26
2.3.13	Tutorial: L7 emulation - fin/ack/fin/ack	27

2.3.14	Tutorial: L7 emulation - syn-syn/ack-ack/rst	27
2.3.15	Tutorial: L7 emulation - delay server response	28
2.3.16	Tutorial: L7 emulation - delay client request	29
2.3.17	Tutorial: L7 emulation - Elefeant flows	29
2.3.18	Tutorial: L7 emulation - http pipeline	30
2.3.19	Tutorial: L7 emulation - UDP example	30
2.4	Performance	30
2.5	Client/Server only mode	31
2.5.1	Limitation	33
2.6	Client clustering configuration	33
2.7	Traffic profile reference	33
2.8	Tunables reference	33
2.9	Counters reference	34
2.9.1	TSO/LRO NIC support	37
2.10	FAQ	37
2.10.1	Why should I use TRex in this mode?	37
2.10.2	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?	38
2.10.3	Is your core TCP implementation based on prior work?	38
2.10.4	What TCP RFCs are supported?	38
2.10.5	Could you have a more recent TCP implementation?	38
2.10.6	Can I reduce the active flows with ASTF mode, there are too many of them?	38
2.10.7	Will NAT64 work in ASTF mode?	38
2.10.8	Is TSO/LRO NIC hardware optimization supported?	39
2.10.9	Can I get the ASTF counters per port/template?	39

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. Consider this document as an extension to the manual, we might integrate both of them 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 support 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.

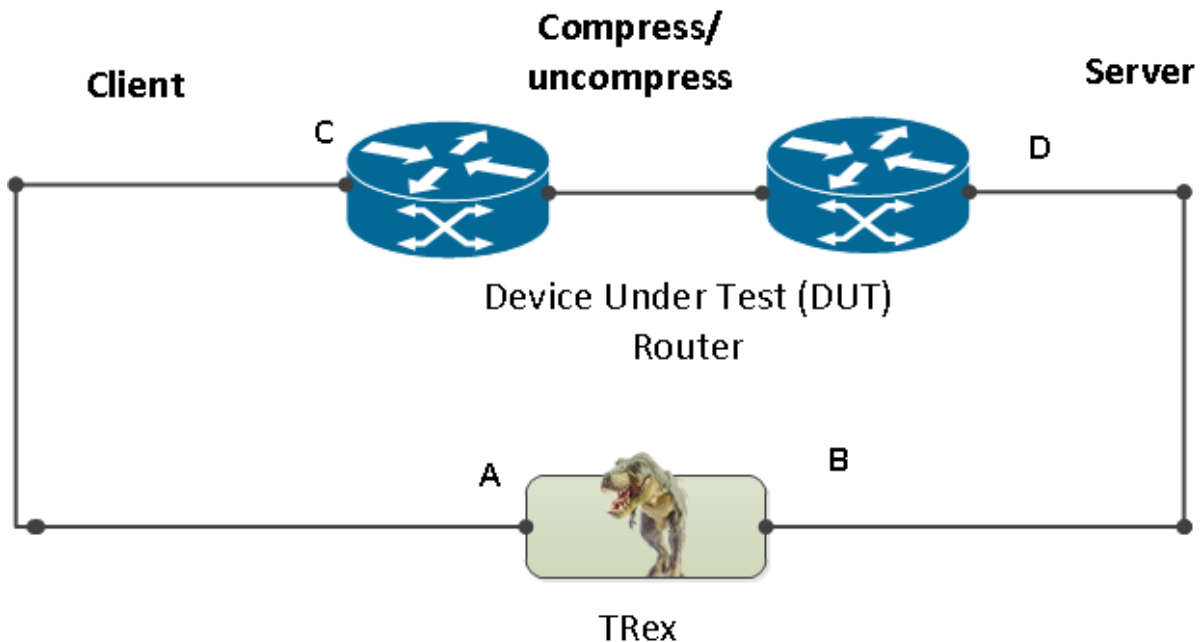


Figure 2.1: DUT is TCP proxy

- 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

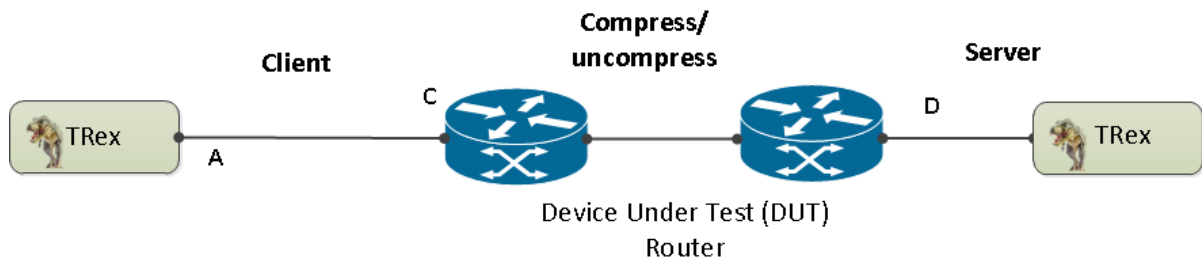


Figure 2.2: C/S mode

- 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/per port)
- 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

Note

Not all the features will be available in the first release.

2.1.1 Status**Warning**

The basic 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
- One DP core per dual-ports
- 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 (using STF batch framework)
- Ability to change the TCP configuration (default MSS/buffer size/RFC enabled etc)
- Client Cluster
- 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)

2.1.1.2 Current high priority tasks

- Interactive support - work with Console to start/stop/get a statistic. This will enable the L2 emulation framework (e.g. Wireless AP)
- More than one core per dual-ports

2.1.1.3 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
- TCP statistic per template
- 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 17 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 following figure shows the block diagram of new TRex TCP design

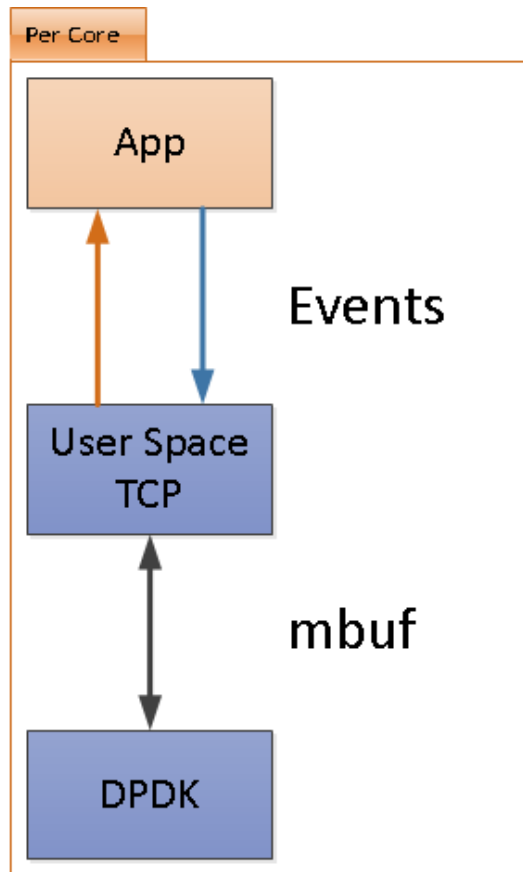


Figure 2.3: Stack

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

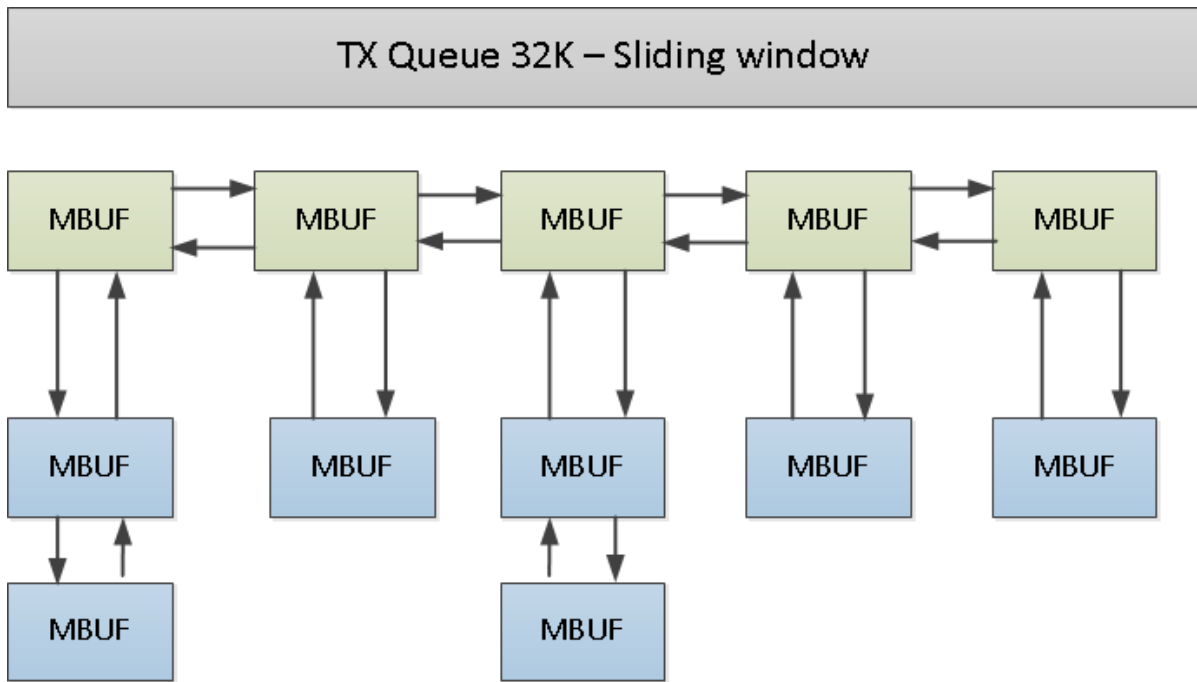


Figure 2.4: TCP Tx side

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 $1M \times 64K \times \text{mbuf-factor}$ (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

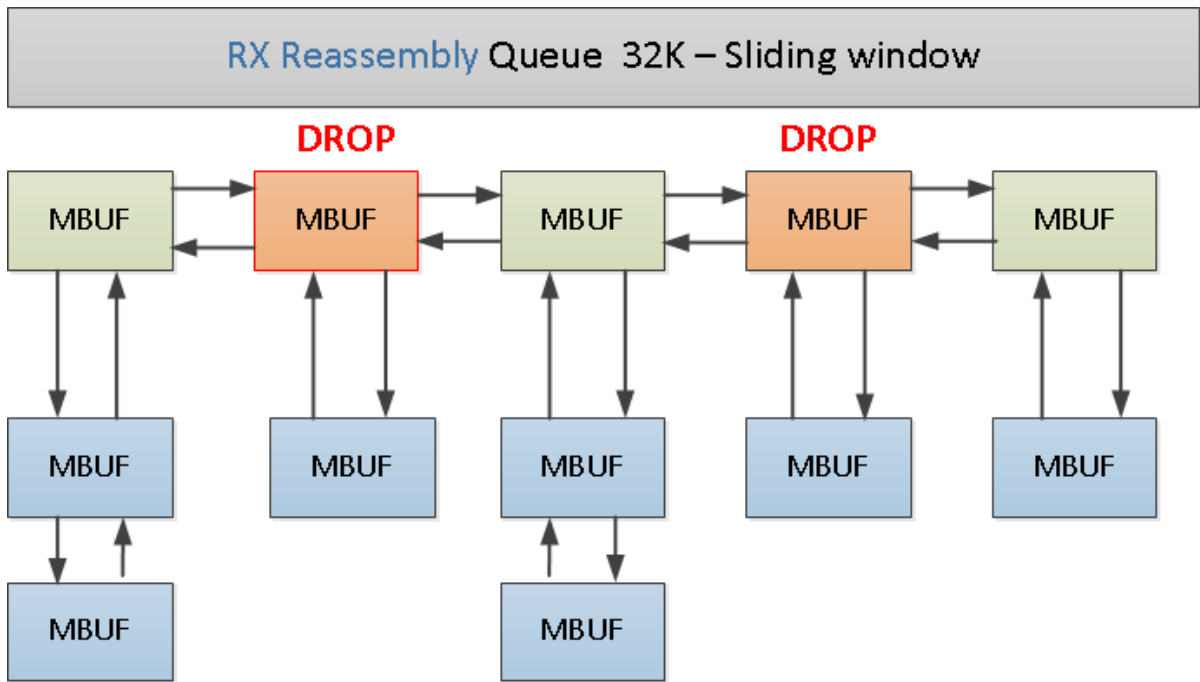


Figure 2.5: Example of multiple streams

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 retransmission (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

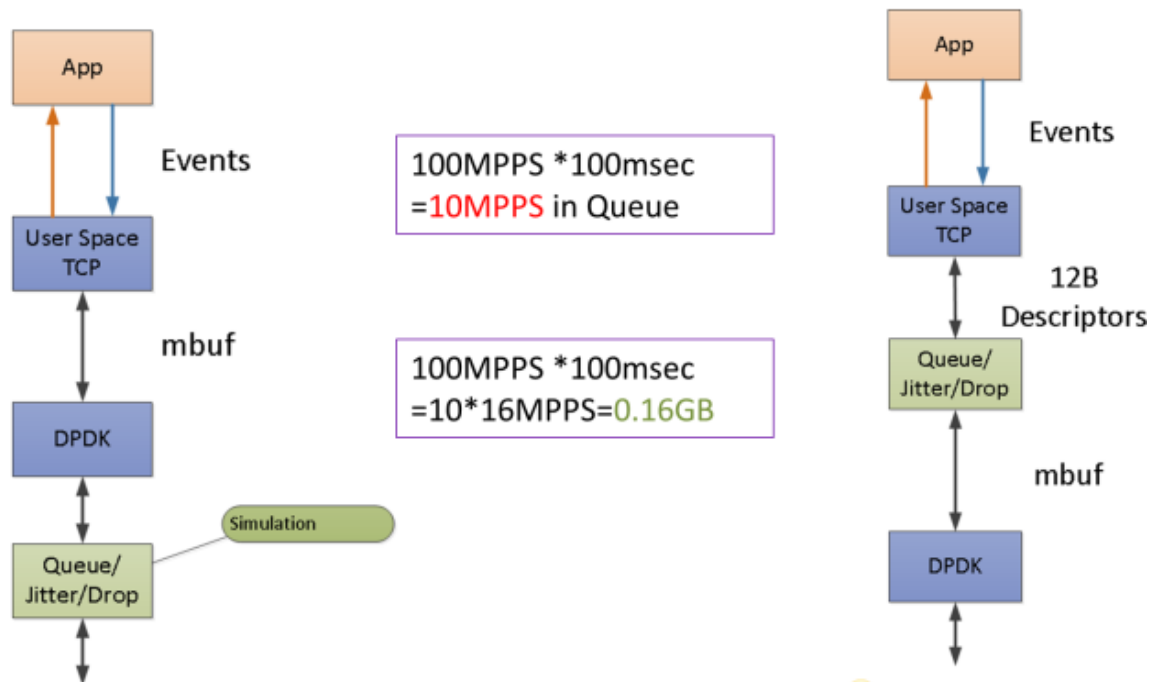


Figure 2.6: TCP Rx side

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/Nagle/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 request, 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)

2.2 ASTF package folders

Location	Description
/astf	astf native (py) profiles
/automation/trex_control_plane/astf/examples	automation examples
/automation/trex_control_plane/astf/trex_astf_lib	astf lib compiler (convert py to JSON)
/automation/trex_control_plane/stf	stf automation (used by astf mode)
/automation/trex_control_plane/astf/examples	stf automation example

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_lib.api import *

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

    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)
                            ])

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

```
[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

- ❶ 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() ❶

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) ❷

s.connect(dest_ip,dst_port) ❸

# program ❹
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();

```

- ❶ Tuple-generator is used to generate tuple for client and server side and choose a template
- ❷ Flow is created
- ❸ Connect to the server
- ❹ 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,dst_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)

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

    s.listen(1)

    #program of the template
    s.read(template.request_size)

    # 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)

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

s.close()

```

- ❶ 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
- ❷ 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 __init__(self):
        pass

    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_https_0.pcap",
                                              cps=2)           ❷
                             ])

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

- ❶ HTTP template
- ❷ 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

2.3.4 Tutorial: Profile with two templates same ports

Goal

create profile with two HTTP templates. In this example, both templates has the same destination port (80)

Traffic profile

The profile includes same HTTP profile only for demonstration.

File

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

    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 a real world the same server can handle many types of transaction on the same port base on the request. In this TRex version we have this limitation as it is only an emulation. next we would add better engine that could associate the template base 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 set of client/server tuple pools

Traffic profile

The profile includes 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

```

- ❶ define generator range 1
- ❷ define generator range 2
- ❸ assigne generator range 1 to the first template
- ❹ assigne generator range 2 to the second template

Discussion

The tuple generator ranges should not overlap

2.3.6 Tutorial: IPv6 traffic

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

just add '--ipv6' to the CLI. The IPv6 packets will be look like this

```
:::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 and in future per template is by tunables in the profile file

IPv6 tunable

```

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

    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

```
[bash]>sudo ./t-rex-64 -f astf/param_ipv6.py -m 1000 -d 1000 -c 1 --astf -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 __init__(self):
        pass

    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 __init__(self):
        pass

    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), ❶
                   ]
                   )
```

- ❶ 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

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`.

We are using for the first version the same STF python API framework, we are working to move to interactive mode like the STL mode

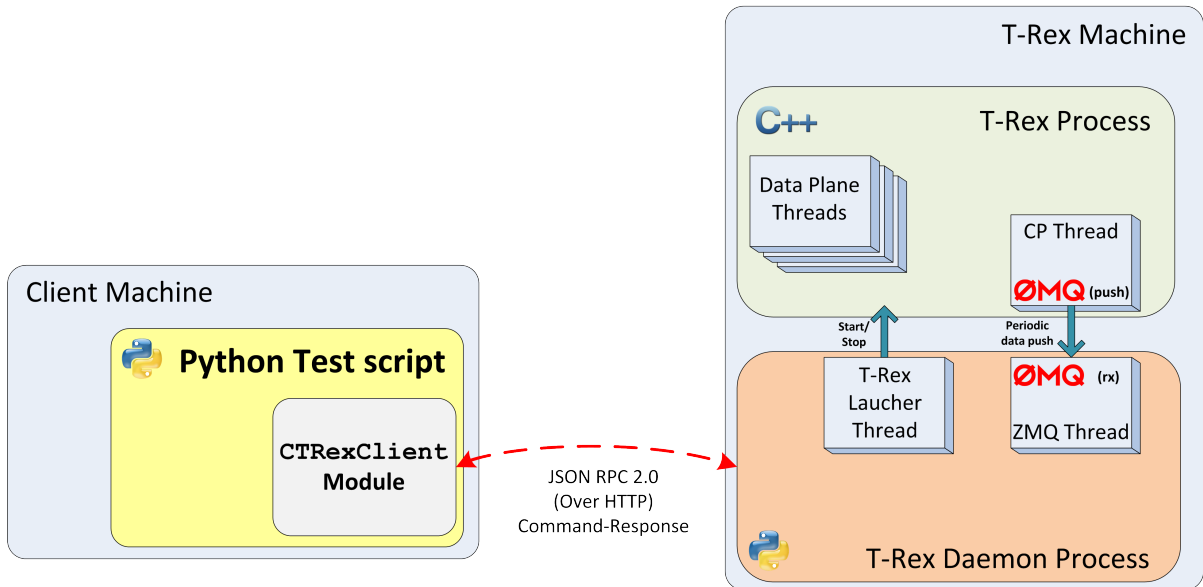


Figure 2.7: RPC Server Components

File

stl_tcp.py

ASTF automation

```

import argparse
import stf_path
from trex_stf_lib.trex_client import CTRexClient
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, # limitation for now
        m = 1000,
        f = 'astf/http_simple.py',
        k=10,
        d = 20,
        l = 1000,
        astf =True, #enable TCP
        nc=True
    )

    result = trex_client.sample_to_run_finish()

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

1

2

3

4

5


```

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");

```

- ❶ Imports the old `trex_stf_lib`
- ❷ force one DP core, limitation for now
- ❸ load a astf profile
- ❹ enable astf mode
- ❺ 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' : 'descriptiuon of the counter' } ❶
},
'server': { 'all': {},
            'err': {}
}
}

```

- ❶ *err* object won't be 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 is 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

2.3.9 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:

Demonstrates 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 capability 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": [
    {
      "client_template": {
        "tcp_info": {
          "index": 0
        },
        "port": 80,
        "cps": 1,
        "program_index": 0,
        "cluster": {},
        "ip_gen": {
          "global": {
            "ip_offset": "1.0.0.0"
          },
          "dist_client": {
            "index": 0
          },
          "dist_server": {
            "index": 1
          }
        }
      },
      "server_template": {
        "program_index": 1,
        "tcp_info": {
          "index": 0
        },
        "assoc": [
          {
            "port": 80
          }
        ]
      }
    }
  ]
}
```

```

    }
  ]
}
],
"tcp_info_list": [
  {
    "options": 0,
    "port": 80,
    "window": 32768
  }
],
"program_list": [
  {
    "commands": [
      {
        "name": "tx",
        "buf_index": 0
      },
      {
        "name": "rx",
        "min_bytes": 32089
      }
    ]
  },
  {
    "commands": [
      {
        "name": "rx",
        "min_bytes": 244
      },
      {
        "name": "tx",
        "buf_index": 1
      }
    ]
  }
],
"ip_gen_dist_list": [
  {
    "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": [
  "ROVUIC8zMzg0IEhUVFAvMS4xDQpIb3",
  "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

- ④ 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

2.3.10 Tutorial: Advanced simulator

Goal

Use the TRex ASTF advanced simulator.

It is like the simple simulator but simulate multiple templates and flows exactly like TRex server would do with one DP core

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

- Use '--full' initiated 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
```

2.3.11 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()           ❶
        prog_c.send(http_req)           ❷
        prog_c.recv(len(http_response))  ❸

        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()

```

- ❶ build the emulation program
- ❷ first send http request
- ❸ 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.

2.3.12 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
```

2.3.13 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

2.3.14 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 will be 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

2.3.15 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 between 100msec-500msec
prog_s.send(http_response)

```

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

see [astf-program](#) for more info

2.3.16 Tutorial: L7 emulation - delay client request

This example will delay the client side.

In this example the client sends partial request (10 bytes), wait 100msec and then send 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, wait for 100msec the only then send 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

2.3.17 Tutorial: L7 emulation - Elefant flows

Let say we would like to send only 50 flows with very big size (4GB) Loading a 4GB of buffer would be a challenge as TRex memory is limited. What we can do is loop inside the client 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); # ❶
prog_s.set_label("a:"); # ❷
prog_s.send(http_response_1mbyte)
prog_s.jump_nz("var2", "a:") # ❸ dec var "var2". in case it is *not* zero jump a:
```

- ❶ set varibale
- ❷ set label
- ❸ jump to label 4096

see [astf-program](#) for more info

Limit the number to flows

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

- ❶ use `limit` field to control the total flows generated

2.3.18 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

2.3.19 Tutorial: L7 emulation - UDP example

This example will show UDP example

Client delay

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

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

- ❶ send UDP message
- ❷ 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

2.4 Performance

see [ASTF Performance](#)

2.5 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 2.1: 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 2.2: Normal mode, 4 ports --astf

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

Table 2.3: Normal mode, 4 ports --astf-server-only

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

Table 2.4: Normal mode, 4 ports --astf-client-mask 1

Port id	Mode
0	C-Enabled
1	S-Enabled

Table 2.4: (continued)

Port id	Mode
2	C-Disabled
3	S-Enabled

Table 2.5: Normal mode, 4 ports --astf-client-mask 2

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

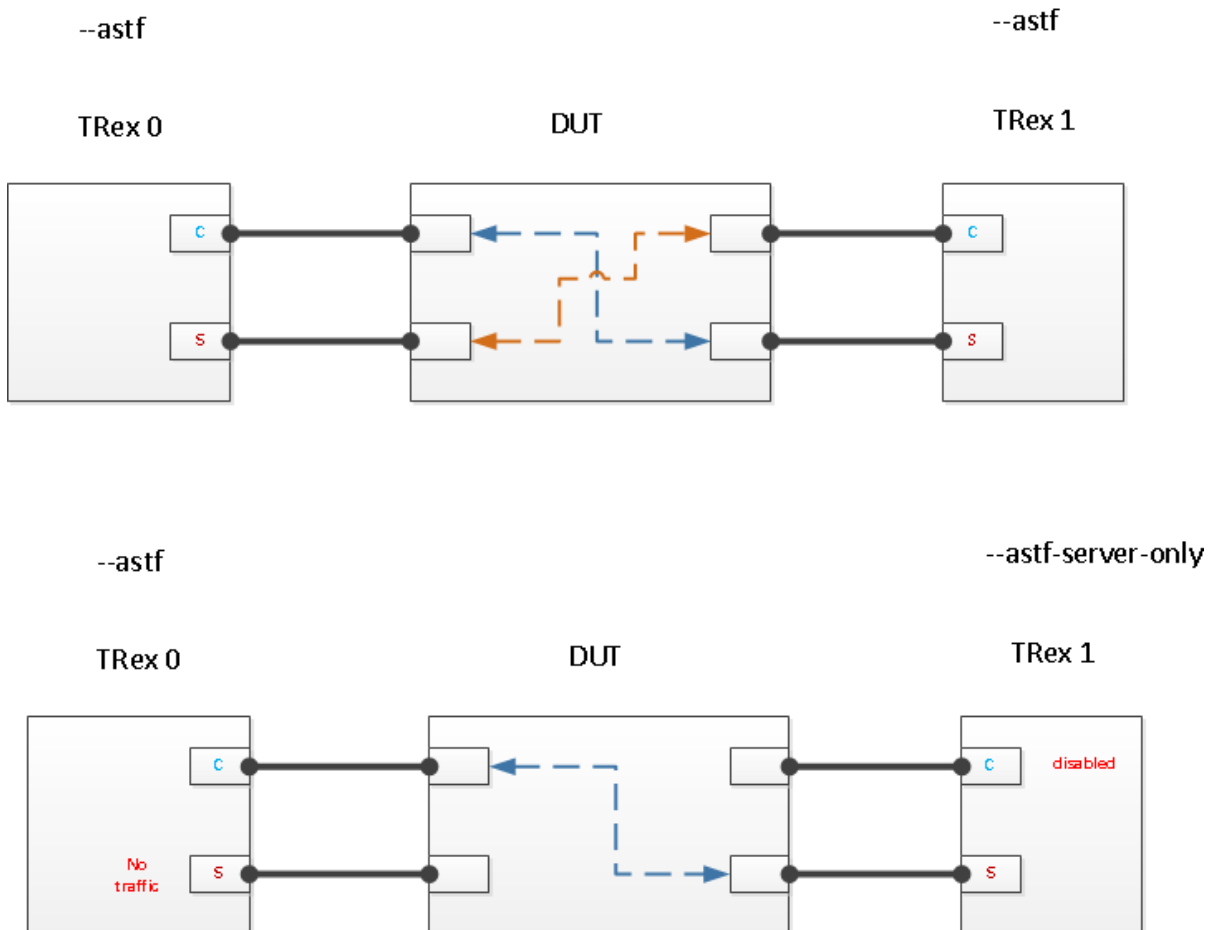


Figure 2.8: C/S modes examples

2.5.1 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
- It is not possible to configure TRex with all ports acting as client side or all ports acting as server side.

2.6 Client clustering configuration

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 Traffic profile reference

[python index](#)

2.8 Tunables reference

tunable	min-max	default	per-template	global	Description
ipv6.src_msb	string	0	-	+	default IPv6.src MSB address.
ipv6.dst_msb	string	0	-	+	default IPv6.dst MSB address.
ipv6.enable	bool	0	-	+	enable IPv6 for all templates.
tcp.mss	10-9K	1460	+	+	default MSS in bytes.

tunable	min-max	default	per-template	global	Description
tcp.initwnd	1-20	10	+	+	init window value in MSS units.
tcp.rxbufsize	1K-1M	32K	+	+	socket rx buffer size in bytes.
tcp.txbufsize	1K-1M	32K	+	+	socket tx buffer size in bytes.
tcp.rexmtthresh	1-10	3	-	+	number of duplicate ack to trigger retransmission.
tcp.do_rfc1323	bool	1	-	+	enable timestamp rfc 1323.
tcp.keepinit	1-254	5	-	+	value in second for TCP keepalive.
tcp.keepidle	1-254	5	-	+	value in second for TCP keepidle
tcp.keepintvl	1-254	5	-	+	value in second for TCP keepalive interval
tcp.delay_ack_msec	20-500	100	-	+	delay ack timeout in msec. Reducing this value will reduce the performance but will reduce the active flows
tcp.no_delay	bool	0	+	+	In case of 1 disable Nagle. Will push any packet with PUSH flag (NOT standard it just to simulate Spirent) and will respond with ACK immediately (standard).
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.
scheduler.accurate	bool	disabled	-	+	Make the scheduler more accurate in expense of performance. more important in low performance rates (<20gb)

- 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 only with the advanced simulator mode.

2.9 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 client ports in the system (e.g. 1/3/5)

Table 2.6: 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
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 between tx_bw_l7_r and tx_bw_l7_total_r 100% means no retransmission TCP

Table 2.7: 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 dropped before establishment
tcps_segstimed		segs where we tried to get rtt
tcps_rtupdated		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_sndrexitpack	*	data packets retransmitted
tcps_sndrexitbyte	*	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
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

Table 2.7: (continued)

Counter	Error	Description
tcps_badsyn	*	bogus SYN, e.g. premature ACK
tcps_reasalloc	*	allocate tcp reassembly ctx
tcps_reasfree	*	free tcp reassembly ctx
tcps_nombuf	*	no mbuf for tcp - drop the packets

Table 2.8: 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 2.9: 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 flow can't happen
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 elarge the number see the trex_cfg file for dp_max_flows)
err_s_nf_throttled		Number of server side flows that were not opened due to flow-table overflow. (to elarge 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

1

see [TSO](#), we count the number of TSO packets with NIC 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 NIC that support that, this number could be significantly smaller than the real number of packets

Important information

- It 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_sndremitbyte+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

```

2.9.1 TSO/LRO NIC support

see manual for

2.10 FAQ

2.10.1 Why should I use TRex in this mode?

ASTF mode can help to solve the following requirement:

- 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)

2.10.2 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.

2.10.3 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?](#)

2.10.4 What TCP RFCs are supported?

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

Not implemented:

- RFC 2018

2.10.5 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.

2.10.6 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.

2.10.7 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

2.10.8 Is TSO/LRO NIC hardware optimization supported?

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

2.10.9 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.