Test Your Limits With TRex Traffic Generator

Hanoch Haim
Cisco systems

***Abstract*—Performance measurement tools are an integral part of network testing. There is no shortage of open source tools for network performance testing in the Linux world. To enumerate a few popular tools in the Linux world: Netperf, iperf, Linux kernel based pktgen. These tools tend to fall into two categories:**

- **Stateless packet shooting such as the linux kernel pktgen traffic generator**
- **Stateful client-server tools such as netperf and iperf.**

When very high performance network performance testing is required (quantified as many 10s of Gigabits per second/100MPPS and/or hundreds of thousands of flows) or more advanced functionality (e.g. realistic) is required the linux classical tools are insufficient. Most organizations will opt for very expensive commercial tools such as Ixia, Spirent. In this paper we will introduce TRex a high performance realistic traffic generator and illustrate sample stateless and stateful use cases that apply to testing Linux networking. We will also discuss its design and tricks that help us achieve such high performance.

***Index Terms*—tcp, performance, scale,realistic traffic generation**

I. Introduction

TRex [1] is an advanced traffic generator, it has the following interesting features:

- It leverages COTS x86/ARM servers and Physical NICs (Intel, Mellanox etc) for high scale
- Can support linux driver or paravirtual (e.g. virtio) for low scale advance feature
- It can serve both Stateless and Stateful traffic generation. tcp stack for stateful traffic and emulation layer to simulate L7 applications
- It outperforms all of iperf, netperf and pktgen
  - It can generate upto 200gbps/100mpps advance traffic pattern and millions of real world tcp/udp flows.
  - High connection rate - order of Millions of Connection/s (CPS)
- It is extensible
  - Emulate L7 application (on top of TCP/IP), e.g. HTTP/HTTPS/Citrix using a programable language
  - Ability to change fields in the L7 application - for example, change HTTP User-Agent field
- Support routing protocols like BGP/OSPF/RIP using BIRD [4]
- Support high scale client simulation protocols (arp.ipv6-nd,dhcp,802.1x,icmp)

Although TRex is implemented on top of DPDK, a lot of the issues we had to deal with when writing the tool apply equally to scaling Linux networking; we share our experiences in that regard and hope to inspire some of the techniques to be adopted in Linux.

II. Software design high level

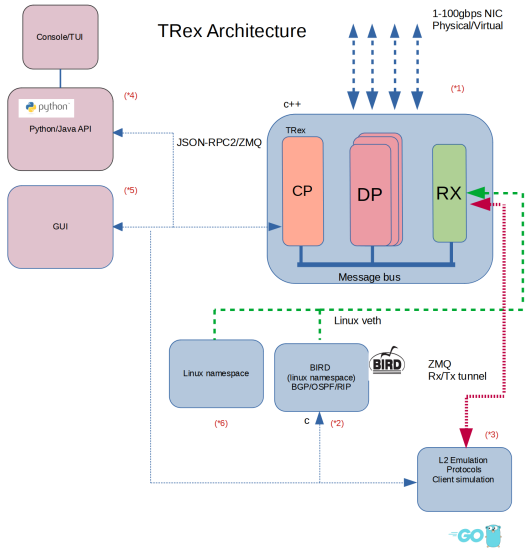

Fig. 1: architecture

Figure 1 presents the main processes. TRex server (*1) is a multi-threaded process, each thread is pinned to a core and works in event driven fashion using a user-space scheduler with a few hierarchy. There is one control-plane (CP) thread that handle the RPC over ZMQ requests and maintenance tasks. The RX thread responsible to handle the low latency traffic for accurate latency measurement. This thread is usually in very low CPU. The DP threads are generating the traffic using and interact with DPDK to transmit/receive traffic via PMD queues. The number of DP threads can be scale up as the number of Tx/Rx queues. There is almost no sharing data structure and no locks to get to best performance. Any information is moved between the threads using a messaging bus which is a shared rings (DP→CP, CP→DP, Rx→DP, RX→CP). There is minimum system call to the kernel (only when required for example with PF
_PACKET driver) (*4) is a Python wrapper to the JSON-RPC2 API over ZMQ so it would be easy automation (e.g. load a profile, get statistics etc) on top of Python API there is a Console that can simplify the way to work with the API. (*5) is the GUI process that written in Java that works directly with the JSON-RPC2 and supports only the Stateless mode. (*2) is a BIRD [4] process that works inside a linux namespace and

connected to TRex via a programmable veths. Inside TRex RX thread there is a Switch that forward packets to/from the veth related to the linux namespace. BIRD is used to simulate routing protocols like BGP/OSPF/RIP. (*6) and (*3) is used for simulating clients slow-path protocols like ARP/IPV6-ND/IGMP/MLD/802.1x/DHCP/DHCPv6 while TRex server is for the fast-path high speed TCP/UDP

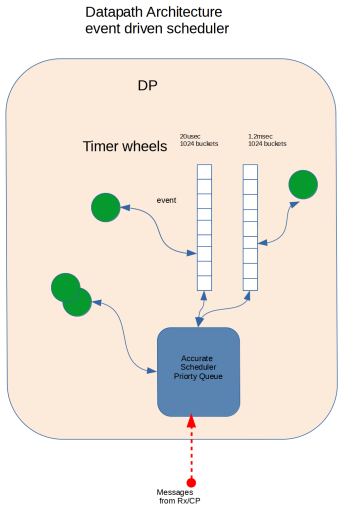

Fig. 2: DP Scheduler

Figure 2 presents the schedulers in each DP thread. The priority queue is the low level scheduler that can schedule events in nsec resolution. On top of that there are two timer wheels for lower resolution events. The first has resolution of 20usec with 1024 buckets for maximum of 2msec time. the second timer wheel has resolution of 1msec with 1024 buckets maximum of 1sec. Each event in the second level is spread each 20usec tick to reduce processing spikes. DP transmit/receive messages from the share rings using events. This design gives linear scale with of performance about 4-20MPPS/core and 200gbps for one COGS server.

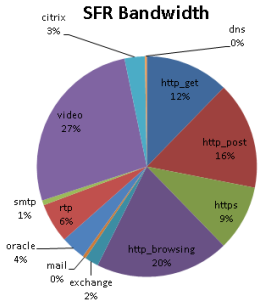

Fig. 3: emix

From functionality point of view TRex has two main operation modes. Stateful and Stateless. Stateful is for testing L7 services that care about clients/flows/L7 application like DPI/NAT/Firewall, Figure 3 is an example of mix of traffic that can be generated using this mode. Stateless is for testing Switch/Filters/ACL/Qos services and has no flow/client state

III. Stateless mode

Stateless traffic model is for testing Switch and device under test (DUT) that are relatively simple and does not have flow/client context.

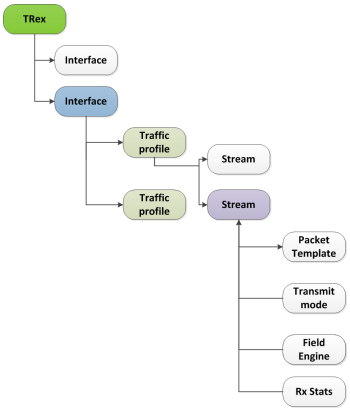

Fig. 4: Stateless objects

Figure 4 shows the model of a profile. Each interface supports one or more traffic profiles in parallel. Each traffic profile supports one or more streams. Each stream includes

- **Packet**: Packet template up to 9 KB
- **Field Engine**: A program that determines which field/packet size
- **Mode**: Specifies how to send packets [Continuous,Burst,Multi-burst]
- **Rx Stats**: Statistics to collect for each stream
- **Rate**: Rate (pps or bps)
- **Action**: Specifies stream to follow when the current stream is complete (valid for Continuous or Burst modes)

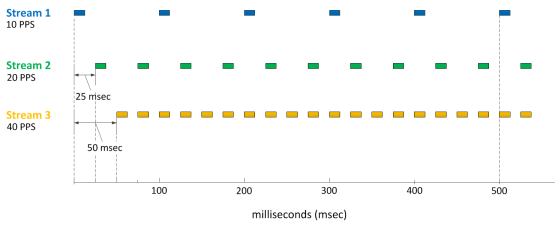

Fig. 5: Stateless profile example

Figure 5 shows an example of a profile, three streams configured for Continuous, Burst, and Multi-burst traffic.

List 1 shows a profile with one UDP stream. The mode is continues

A. Stateless Features

- Large scale - Supports about 10-22 million packets per second (mpps) per core, scalable with the number of cores
- Support for 1, 10, 25, 40, and 100 Gb/sec interfaces with Linux driver vs PF
  _PACKET

```python
from trex_stl_lib.api import *

class STLS1(object):

  def create_stream (self):

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

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

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

Listing 1: Profile with one continues UDP stream

- Support for multiple traffic profiles per interface
- Programable Field Engine to change any field inside the packet (e.g. `src_ip=10.0.0.1..10.0.0.255`)
- Ability to change the packet size
- API,Console, GUI
- Statistics, per interface,per stream
- Latency and jitter per stream
- Multi-user support

B. Multi stream example

Figure 6 shows two streams the blue one is a burst stream that activate a multi-burst stream. List 2 shows the python profile to create this profile

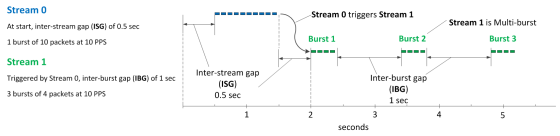

Fig. 6: Multi stream profile

C. Field Engine

The field engine (FE) is a programable engine that can change any field in the packet and is loaded from python to a fast bytecode. The implementation challenge was to provide an engine that change packet fields on number of core in parallel but as a black-box behave like it runs on a single thread (hardware like) Let gives an example of syn-attack and explain it

List 3 shows a FE program that generate a syn attack using one stream. every stream has a context of a FE. in this context in this example there are two variables **ip _src** for the range of the source ipv4 ips and the **source _port** for the range of the source protocols. Those variables are written to the right offset in the packet and fix the checksum (using hardware assist if exists)

```python
def create_stream (self):

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


return STLProfile(

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

  STLStream( self_start = False,
             name    ='S1',
             packet  = STLPktBuilder(pkt=base_pkt1/pad),
             mode    = STLTXSingleBurst(pps=10,
            total_pkts = 20),
             next    = 'S2' ),

  STLStream(  self_start = False,
              name    ='S2',
              packet = STLPktBuilder(pkt= base_pkt2/pad),
              mode = STLTXSingleBurst(pps=10,
              total_pkts=30)
              )
]).get_streams()
```

Listing 2: Multi profile example

D. Automation using Python API

List 4 shows a simple script to automate TRex. it self explained

E. Stateles Performance

Table 1. Traffic Profile = Field Engine Cached 255

Packet size	Line Utilization (%)	Total L1 (Gb/s)	Total L2 (Gb/s)	CPU Util (%)	Total MPPS	BW per core (Gb/s) ❶	MPPS per core ❷	Multiplier
Imix	100.04	80.03	76.03	2.7	25.03	89.74	28.07	100%
1514	100.12	80.1	79.05	1.33	6.53	430.18	35.07	100%
590	99.36	79.49	76.89	3.2	16.29	177.43	36.36	99.5%
128	99.56	79.65	68.89	15.4	67.27	36.94	31.2	99.5%
64	52.8	42.3	32.23	14.1	62.95	21.43	31.89	31.5mpps

Table 2. Traffic Profile = Field Engine with 1 variable

Packet size	Line Utilization (%)	Total L1 (Gb/s)	Total L2 (Gb/s)	CPU Util (%)	Total MPPS	BW per core (Gb/s) ❶	MPPS per core ❷	Multiplier
Imix	100.04	80.03	76.03	12.6	25.03	45.37	14.19	100%
1514	100.12	80.1	79.05	2.6	6.53	220.05	17.94	100%
590	99.36	79.49	76.89	5.6	16.29	101.39	20.78	99.5%
128	99.56	79.65	68.89	33.1	67.27	17.19	14.52	99.5%
64	52.8	42.3	32.23	31.3	63.06	9.65	14.37	31.5mpps

❶ Extrapolated L1 bandwidth per 1 core @ 100% CPU utilization.
❷ Extrapolated amount of MPPS per 1 core @ 100% CPU utilization.

Fig. 7: Stateless performance with Intel XL710

Figure 7 [5] shows the measured performance on a Cisco UCS server with dual socket and two Intel XL710 NICS

```
class STLS1(object):
""" attack 48.0.0.1 at port 80
"""

def __init__ (self):
    self.max_pkt_size_l3  =9*1024;

def create_stream (self):

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

    # create an empty program (VM)
    vm = STLVM()

    # define two vars
    vm.var(name = "ip_src",
           min_value = "16.0.0.0",
           max_value = "18.0.0.254",
           size = 4,
           op = "random")

    vm.var(name = "src_port",
          min_value = 1025,
          max_value = 65000,
          size = 2,
          op = "random")

    # write src IP and fix checksum
    vm.write(fv_name = "ip_src",
             pkt_offset = "IP.src")

    vm.fix_chksum()

    # write TCP source port
    vm.write(fv_name = "src_port",
             pkt_offset = "TCP.sport")

    # create the packet
    pkt = STLPktBuilder(pkt = base_pkt, vm = vm)

    return STLStream(packet = pkt,
                     random_seed = 0x1234,
                     mode = STLTXCont())
```

Listing 3: FE syn attack 48.0.0.1 server

IV. Stateful mode

Stateful model objective is to simulate realistic L7 applications on top of TCP/UDP stack (based on BSD source code) at high scale. The scale could reach millions of flows and 100k clients/servers up to 200gbps for one server. It is important to test Stateful features using realistic traffic scenarios because this is the only way to estimate more accurate performance metrics and identify bottlenecks in the design. Figure 8 present the traffic generation model.

Each profile includes:

- **Client pool**: Range of clients with a distribution model e.g. random,seq. a profile can include a few pool
- **Server pool**: Range of servers, profile can include a few pools
- **Template**: A model that describe an application on top of TCP/UDP. A pcap for L7 data can be taken
  - **CPS**: How many connection per second for this template
  - **flowLimit**: to limit the flows

In this model each core has its own context of TCP/UDP stack with no sharing and no locks. It works on top of the

```
c = STLClient(username="itay",
              server = "10.0.0.10",
              verbose_level = "error")
try:
    # connect to server
    c.connect()

    # prepare our ports
    c.reset(ports = [0, 1])

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

    # clear the stats before injecting
    c.clear_stats()

    c.start(ports = [0, 1],
            mult = "5mpps",
            duration = 10)

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

    # check for any warnings
    if c.get_warnings():
      # handle warnings here
      pass

finally:
    c.disconnect()
```

Listing 4: Stateless automation example

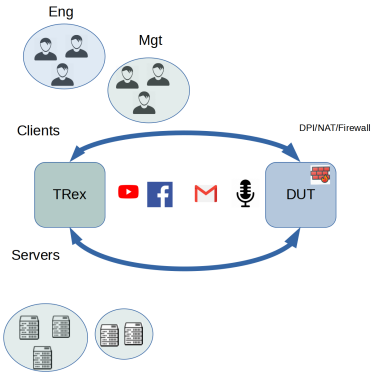

Fig. 8: Stateful model

scheduler hierarchy in Figure 2 The changes to the BSD stack were

- Each stack has a context per thread
- Tx work in pool mode (build the packets only when required) and save reference to the template data. This saves three order of magnitude of memory resource

Figure 9 shows the stack of the programable application emulation layer. this module responsible to simulate application on top of the TCP/UDP stack. Example for commands are:

- Start write buffer
- Continue write
- End Write
- Wait for buffer/timeout
- OnConnect/OnReset/OnClose

A. *Stateful Profile*

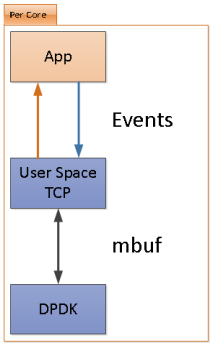

Fig. 9: Stateful stack

```
from trex.astf.api import *

class Prof1():
    def get_profile(self):
        # ip generator
        ip_gen_c = ASTFIPGenDist(
          ip_range=["16.0.0.1", "16.0.0.254"],
          distribution="seq")
        ip_gen_s = ASTFIPGenDist(
          ip_range=["48.0.0.1", "48.0.255.254"],
          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)
        ])
```

Listing 5: Stateful profile

List 5 shows a simple profile with one pool of client (16.0.0.1-16.0.0.254) and one pool of servers (48.0.0.1-48.0.255.254) The pcap file is parsed and the L7 data is converted to instructions on top of the TCP stack

B. Emulation layer instructions

List 6 shows a simple example of a low level instructions of the emulation layer. This example the client send request and wait for response while the server wait for request and send a response.

```
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.send(http_response)
```

Listing 6: Emulation layer instructions

C. Features

- high scale (flows, bandwidth, connection per second)
- measure latency/jitter/drop in high rate
- Emulate L7 application, e.g. HTTP/HTTPS/Citrix- there is no need to implement the exact application.

```
template = choose_template()

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)

s.connect(dest_ip,dst_port)

# program
s.write(template.request)
  # 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)
   #HTTP/1.1 200 OK
   #Server: Microsoft-IIS/6.0
   #Content-Type: text/html
   #Content-Length: 32000
   # body ..


s.close();
```

Listing 7: Client pseudo code

- TCP implementation based on BSD
- automation Python API
- TCP/UDP/Application statistics (per client side/per template)

D. Automation example

Listing 9 shows an example of python script to automate load a stateful profile and read port and TCP statistics

E. Memory saving

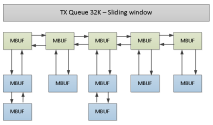

Fig. 10: One Flow Tx Ring

Most TCP stacks have an API that allow the user to provide a buffer (write operation). The TCP module saves the buffer until the data is acknowledged by the remote side. With big windows (required with high RTT) and many flows this could create a memory scale issue. Figure 10 shows one TCP flow TX queue. For 1M active flows with 64K TX buffer the worst case memory requirement is 1M x 64K * mbuf-factor (let's assume 2) = 128GB. The mbuf resource is expensive and needs to be allocated ahead of time. The chosen solution for this problem is to change the API to be a poll API, meaning TCP Tx queue will just save a reference to the constant traffic and offset into this ring the mbufs are allocated only when packets need to be sent (lazy) but virtually have a tx queue only for management of queue (two pointers). Now because most of the traffic is almost constant in traffic generation case (per template) and known ahead of time it was possible

```
# 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)
    #get template for the dest_port
    template= lookup_template(pkt.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()
```

Listing 8: Server pseudo code

to implement and save most of the memory. The same idea happen in the Rx side with reassembly [1]

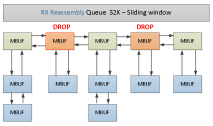

Fig. 11: One Flow Rx Ring

F. Benchmark TRex vs Linux kernel

To evaluate the performance and memory scale of TRex and compare it against standard linux tools the following was done. Linux **curl** as a client and **nginx** as a server were compared to TRex for stressing a device under test.

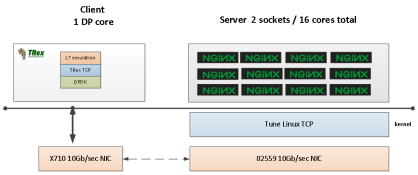

Fig. 12: TRex vs NGINX

The benchmark setup was designed to take a good event-driven Linux server application and to test a TRex client against it. TRex is the client requesting the pages. Figure 12 shows the topology in this case. TRex generates requests using **one** DP core/thread and exercises the **whole** 16 cores of the NGINX/Linux server. The server is installed with the NGINX process on all 16 cores. After some trial and error, it was determined that is is more difficult to separate Linux kernel/IRQ contexts events from user space process CPU, so it was chosen to give the NGINX all server resources, and monitor to determine the limiting factor. The objective is to simulate HTTP sessions as it was defined in our benchmark (e.g. new session for each REQ/RES, initwnd=2 etc.) and not to make the fastest most efficient TCP configuration. This might be the main **difference** between NGINX benchmark configuration and this document configuration. In both cases (client/server), the same type of x86 server was used (2 sockets,CPU E5-2667, Intel X710)

[1]This will not work for TLS streams

```
c = ASTFClient(server = server)

c.connect()

try:
    c.reset()

    c.load_profile(profile_path)

    c.clear_stats()

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

    c.wait_on_traffic()

    stats = c.get_stats()
         2
    pprint.pprint(stats)

    if c.get_warnings():
        for w in c.get_warnings():
            print(w)


except TRexError as e:
    print(e)

except AssertionError as e:
    print(e)

finally:
    c.disconnect()
```

Listing 9: Stateless automation example

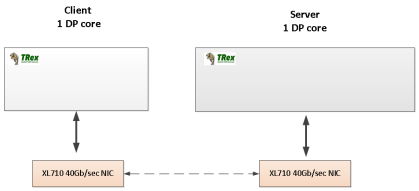

Fig. 13: TRex vs TRex

To compare apples to apples, the nginx server was replaced with a TRex server with **one** DP core, using an XL710 NIC (40Gb). see Figure 13

G. Benchmark traffic profile

Typically, web servers are tested with a constant number of active flows that are opened ahead of time. In the nginx benchmark blog, only 50 TCP constant connections are used with many requests/responses for each TCP connection see here [6]. In our traffic generation use case, each HTTP request/response (for each new TCP connection) requires opening a **new** TCP connection. A simple HTTP flow with a request of 100B and a response of 32KB (about 32 packets/flow with initwnd=2) was used.

H. Benchmark Limitations

The comparison is not perfect, as TRex merely emulates HTTP traffic. It is not a real-world web server or HTTP client. For example, currently the TRex client does not parse the HTTP response for the Length field. TRex simply waits for a specific data size (32KB) over TCP. However the TCP layer is a full featured TCP (e.g. delay-ack/Retransmission/Reassembly/timers) . The bench objective is to compare traffic generation capabilities for stressing network gears.

I. Benchmark results

Comparing 1 DP core running TRex to NGINX running on 16 cores with a kernel that can interrupt any NGINX process with IRQ. Figure 3 shows the performance of one DP TRex. It can scale to about 25Gb/sec of download of HTTP (total of 3MPPS for one core).

	TRex one DP core							
m	CPU (1DP) %%	cps	rps	rx (mb/sec)	pps(tx+rx)	active flows	drop	Memory TCP/MB
1000	0.6	1000	1000	265	34444	600	0	0.3
5000	2.7	5000	5000	1320	172222	3000	0	1.4
10000	5.7	10000	10000	2210	344444	6000	0	2.9
20000	13.5	20000	20000	5410	688889	12000	0	5.7
50000	40.8	50000	50000	13480	1722222	29870	0	14.2
87500	79.0	87500	87500	23670	3013889	55329	0	26.4
90000	86.1	90000	90000	24271	3100000	56217	0.10%	26.8

Fig. 14: TRex 1 DP core

					Linux 16 cores			
m	cps	rps	rx (mb/sec)	active flows	16xCPU	drop	Kernel memory SLAB (MB)	Total Memory used (free-h) MB
1000	1000	1000	265	600		no		21000
5000	5000	5000	1320	3000		no		22000
10000	10000	10000	2210	6000		no		25000
20000	20000	20000	5410	12000	20%/25% 8x cores IRQ break	yes	800	31000
50000	50000	50000	13480	29870				
87500	87500	87500	23670	55329				
90000	90000	90000	24271	56217				

Fig. 15: NGINX 16 cores

nginx cannot handle more than 20K new flows/sec, due to kernel TCP software interrupt and thread processing. The limitation is the kernel and not NGINX user space process. With more NICs and optimized distribution, the number of flows could be increased X2, but not more than that. The total number of packets was approximately 600KPPS (Tx+Rx). The number of active flows were 12K.

TRex with one core could scale to about 25Gb/sec, 3MPPS of the same HTTP profile. The main issue with nginx and Linux setup is the tunning. It is very hard to let the hardware utilizing the full server resource (half of the server was idle in this case and still experience a lot of drop). TRex is not perfect too, we couldn't reach 100
% CPU utilization without a drop (CPU was 84
%). To achieve 100gbps with this profile on the server side requires 4 cores for TRex, vs. 20x16 cores for NGINX servers. TRex is faster by a factor of **80**. In this implementation, each flow requires approximately 1K bytes of memory (Regardless of Tx/Rx rings because of TRex architecture). In the kernel, with a real-world server, TRex optimization can't be applied and each TCP connection must save memory in Tx/Rx rings. For about 5Gb/sec traffic with this profile, approximately 10GB of memory was required (both NGINX and Kernel). For 100Gb/sec traffic, approximately 200GB is required (If we will do the extrapolation) With a TRex optimized implementation, approximately 100MB is required. TRex thus provides an improvement by a factor or **2000** in the use of memory resources.

V. Emulation layers

To simulate clients there is a need for many a slow path

References

[1] Cisco systems: "TRex realistic traffic generator", https://trex-tgn.cisco.com/trex/doc/

[2] Cisco systems, "TRex Stateless",, https://trex-tgn.cisco.com/trex/doc/trex_stateless.html

[3] Cisco systems, "TRex Stateful ASTF",, https://trex-tgn.cisco.com/trex/doc/trex_astf.html

[4] BIRD, Faculty of Math and Physics, Charles University Prague https://bird.network.cz/

[5] Cisco systems, "TRex STL benchmark",, https://trex-tgn.cisco.com/trex/doc/trex_stateless_bench.html

[6] NGINX performance https://www.nginx.com/blog/testing-the-performance-of-nginx