MoonGen: A Fast and Flexible Packet Generator

Paul Emmerich
emmericp@net.in.tum.de
Technical University of Munich
Chair of Network Architectures and Services
IETF-100, 16.11.2017
Research at net.in.tum

- AS56357: Chair of Network Architectures and Services
  - Prof. Dr.-Ing. Georg Carle
  - 5 Post-docs
  - 15 PhD students/research associates
- Broad range of network research topics
  - Traffic measurement and analysis
  - Software-defined networking
  - Security
  - Privacy
  - Peer-to-peer networks
  - IoT
  - Performance analysis and modeling
Performance analysis and modeling

• Packet processing becomes more complex
  – Software-defined networking, network function virtualization, …
• More and more can be done in software nowadays
  – Frameworks like DPDK
  – Complex virtualized network functions, e.g., in 5G
  – Performance impacts unclear

• Research questions
  – What are important performance metrics?
  – How to measure them in a realistic scenario?
  – How to make measurements reproducible?
  – How can performance be predicted with models?
Our testbed

- 15 servers, 36 x 10 Gbit/s ports, 8 x 40 Gbit/s ports
  - NICs from Intel, Mellanox, and Netronome
  - SDN switches/routers
- Fully automated test workflow from a management server
  - Allocate servers exclusively
  - Define and run experiment test scripts
  - Get results in a Jupyter notebook
- Servers boot pre-built live images via PXE
  - Ensures reproducibility
  - Collection of different kernel versions/distributions
About me

• PhD student at Technical University of Munich
• Started in 2014
• PhD thesis about testing network devices
• Built the MoonGen packet generator for this
  - Used quite often in academia nowadays :)


Packet generators
Commodity hardware

Source: www.intel.com
MoonGen - A fast software packet generator

Combines the advantages of software (cheap, flexible) and hardware (precise, accurate) packet generators.

- *Fast*: DPDK for packet I/O, explicit multi-core support
- *Flexible*: Craft all packets in user-controlled Lua scripts
- *Timestamping*: Hardware features found on NICs
- *Rate control*: Hardware features and novel software approach
- *Free and open source*: Code available on GitHub

https://github.com/emmericp/MoonGen

Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle.

Traffic patterns matter: CBR is hard!

- Forwarding latency of Open vSwitch (kernel), increasing load
- Baseline latency: CBR traffic, varying burst sizes

- Bursts are important for performance
- Typical default burst sizes: 16 to 256
- Packet generators often fail to generate CBR reliably
CBR can lead to weird effects

- Forwarding latency of Open vSwitch (kernel), increasing load
- Dynamic interrupt throttling (ixgbe driver) and poll-mode (NAPI) don’t play well with CBR traffic
Real-world traffic isn’t CBR

- Only change: time between packets
- Real-world traffic is a self-similar pattern
- Can be approximated with a Poisson process on short time scales
Latency measurements

The total forwarding latency $l$ consists of the delay introduced by the connection from the packet generator to the switch $l_{\text{gen}}$, the forwarding latency of the switch $l_{\text{switch}}$, and the number of hops $n$: 

$$l = 2 \cdot l_{\text{gen}} + n \cdot l_{\text{switch}}$$

We measured the forwarding latency through the switch with various loop lengths from $n = 0$ (sending the traffic back directly) to $n = 23$. Figure 6 shows the CDFs of different loop lengths up to $n = 15$ to improve the readability of the graph as the remaining CDFs look similar. We can calculate the following median latencies from these results:

$$l_{\text{gen}} = 480 \text{ ns} \quad \text{and} \quad l_{\text{switch}} = 729 \text{ ns}.$$ 

These values include propagation delay due to varying cable lengths, we used copper cables with various lengths between 0.5 and 3 meter. This introduces an additional error of $12 \text{ ns}$ (assuming a propagation speed of $0.7 c$) in addition to the granularity of $12.8 \text{ ns}$ of the packet generator.

Note that these results are crucial for FLOWer: The latency of the switch is important for further tests using the switch to amplify traffic for a separate DuT. In such a setup, the switch is part of the measurement equipment, and its accuracy therefore limits the total accuracy of the experiment.

These results show that forwarding latency does not depend on the switch ports. This indicates the high accuracy of the packet generator and that latency is independent from the used switch port. We did not test all combinations of ports, one should repeat this test with the appropriate set of ports to verify this before relying on a switch to run latency-critical experiments. There may be differences in the latency between ports on a switch due to the internal architecture of the switch.

The difference between the minimum and maximum observed forwarding latency was only 217.6 ns (cf. the steep CDFs in Figure 6, each based on 48,000 timestamped packets over 48 seconds).
Latency measurements
Generating complex packets

- Arbitrarily complex header stacks
- Generates and JIT compiles C structs
- Defaults for all header fields
  - E.g., calculates lengths, ports based on upper protocol
- Getters and setters, automatic endianness handling
- Following example code based on

```lua
local vxlanStack = packetCreate(  
  "eth", "ip4", "udp", "vxlan",  
  {"eth_8021q", "innerEth"},  
  {"ip4", "innerIp4"},  
  {"udp", "innerUdp"}  
)
```
Generating complex packets

• Create a mempool with a packet archetype

```lua
local mempool = memory.createMemPool(function(buf)
    local pkt = vxlanStack(buf)
    pkt:fill{
        -- fields not explicitly set here are initialized to defaults
        ethSrc = queue, -- MAC of the tx device
        ethDst = arpTask.lookup("10.0.0.3"),
        ip4Src = "10.0.0.2",
        ip4Dst = "10.0.0.3",
        vxlanVNI = 10100,
        -- outer UDP ports are set automatically by the VXLAN handler
        innerEthSrc = "12:34:56:78:90:ab",
        innerEthDst = eth.BROADCAST,
        innerEthVlan = 100,
        innerIp4Src = "192.168.0.1",
        innerIp4Dst = "255.255.255.255",
        innerUdpSrc = 1024,
        innerUdpDst = 1024,
        pktLength = 128
    }
    pkt.innerIp4:calculateChecksum()
end)
```
Generating complex packets

• Write a transmit loop

```lua
local bufs = mempool:bufArray()
while mg.running() do
    bufs:alloc()
    for i, buf in ipairs(bufs) do
        local pkt = vxlanStack(buf)
        pkt.innerUdp:setDstPort(1000 + math.random(0, 1000))
        -- randomize other fields here
    end
    bufs:offloadUdpChecksums()
    queue:send(bufs)
end
```
Don’t want to write a script? Use our CLI!

• Define one or multiple flows in a config file, e.g.

```plaintext
Flow{"syn-flood6", Packet.Tcp6{
  ethSrc = txQueue(),
  ethDst = mac"12:34:56:78:90:00",
  ip6Dst = ip"2a00:4700::2:225:90ff:fe74:7716",
  ip6Src = range(ip"fe80::1", ip"fe80::ffff:ffff"),
  tcpSrc = randomRange(0, 2^16 - 1),
  tcpDst = 80,
  tcpSyn = 1,
  tcpSeqNumber = randomRange(0, 2^32 - 1),
  tcpWindow = 10

}}
```
Don’t want to write a script? Use our CLI!

• Send out previously defined flows

  ./moongen-simple start syn-flood6:<dev>,<dev>:rate=40Gbit/s

• Combine arbitrary flows
• Different traffic patterns: CBR, Poisson, …
• Time limits for automated tests
• Per-flow packet counters
• Quick debugging by printing instead of sending
• See ./moongen-simple help for more

• Caution: the CLI is still new and you might encounter bugs
How are others using MoonGen?

- OPNFV project: Test/benchmark framework VSPERF, MoonGen is one of multiple supported packet generators
- PISCES, SIGCOMM’16: Software P4 switch, performance evaluation
- NFVnice, SIGCOMM’17: NFV service chain scheduling, performance evaluation
- Flurries, CoNEXT’16: NFV framework, performance evaluation
- DNS DDoS Resilience Tests, RIPE 74: DNS traffic generation
## How are others using MoonGen?

<table>
<thead>
<tr>
<th>Project and authors</th>
<th>Publication venue</th>
<th>Doing what</th>
</tr>
</thead>
<tbody>
<tr>
<td>PISCES Shahbaz et al.</td>
<td>SIGCOMM’16</td>
<td>Software P4 switch, performance evaluated with MoonGen. Contributed timestamping code for Intel 40 Gbit/s NIC.</td>
</tr>
<tr>
<td>Neutral Net Neutrality Yiakoumis et al.</td>
<td>SIGCOMM’16</td>
<td>Privacy-preserving quality of service, MoonGen used for the evaluation. Custom protocol/payload for test traffic.</td>
</tr>
<tr>
<td>NFVnice Kulkarni et al.</td>
<td>SIGCOMM’17</td>
<td>NFV chaining and scheduling, performance evaluated with MoonGen.</td>
</tr>
<tr>
<td>DNS DDoS Resilience Rincón et al.</td>
<td>RIPE-74</td>
<td>Replicating large DDoS attacks against DNS servers. Contributed DNS protocol code for MoonGen.</td>
</tr>
<tr>
<td>OPNFV VSPERF Linux Foundation</td>
<td>-</td>
<td>MoonGen is one of multiple supported packet generators to test and benchmark the OPNFV project. Complex MoonGen script as test harness.</td>
</tr>
</tbody>
</table>
Check out MoonGen on GitHub

MoonGen comes with a lot of examples
See if one fits your use case

https://github.com/emmericp/MoonGen
Questions?