MoonGen Tutorial

Jonas Jelten
Betreuer: Paul Emmerich, Daniel Raumer
Seminar Innovative Internet-Technologien und Mobilkommunikation WS2015
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: jelten@in.tum.de

ABSTRACT
This paper provides a short introduction into the usage of MoonGen, a high performance packet generation framework written in Lua. It is based on DPDK which mediates the hardware access. You will learn how you can interact with the MoonGen API to craft and send custom packets, gather statistics and verify received data. Communication between tasks running in parallel is demonstrated. The usage of hardware features like queues and rate control is illustrated and explained. You will see that MoonGen is simple to use for many load generation use cases.

Keywords
networking, MoonGen, tutorial, howto, Lua, Linux

Version
This is the tutorial version `v1.0`.

1. WHAT'S MOONGEN?
After reading this tutorial, you will be able to use MoonGen to benchmark and test your network setup in any way you like. You'll learn the concepts, the architecture and basics of the MoonGen API.

MoonGen [2] is a software based packet generator framework, designed for easy use and high speeds at 10Gbit and more. Executed on common hardware, it can be used for just load generation in benchmarking applications, or to check the response validity for error detection by executing custom code for each packet without expensive special hardware. This way, firewalls, network address translation and quality of service setups can be tested and verified to operate correctly even under enormous load. Sub-microsecond latency and packet drops can be measured and checked if they match the expected behavior in benchmarks.

MoonGen is based on Data Plane Development Kit`DPDK` [1], which is granting direct hardware access via `DMA`, thus allowing the `LuaJIT` [6] machine to interact with the network interface at maximum speeds. In order to use MoonGen, you should have a basic knowlege of Lua, for example from a quick tutorial at
`https://learnxinyminutes.com/docs/lua/` [4].

2. SETUP
MoonGen is intended to run on any GNU/Linux distribution. This guide was created on Ubuntu 14.04.

To install, clone the `git` repository from the upstream url at `https://github.com/emmericp/moongen` [5], then follow the prerequisite requirements and installation directions in the `README` file.

After you built the project successfully, try if you can execute `./MoonGen` and get the usage information printed. If that works, you may continue with the tutorial.

3. ARCHITECTURE
In principle, MoonGen is a high-level frontend for DPDK. DPDK provides a low-level API for hardware access, packet generation and response processing, mainly designed for data plane development [1]. MoonGen's core is a convenient lua wrapper for that API. To use it, you create custom lua files containing code instead of config files: This allows much more flexibility for any measurement application you intend to conduct.

The entry point for all the custom code is a control script, containing a `master` function. It is called by MoonGen, should set up the interfaces and request the desired settings. Internally, configuration is then passed to dpdk, which performs the actual hardware setup.

This control script can spawn new tasks as separate LuaJIT VMs. That way, packet generation, receive measurements, verifications, etc. can easily be implemented apart and executed in parallel.

After packet fields are composed in some task, they're passed to dpdk which crafts the payload and sends it out of the device. Data received from the hardware is mediated through dpdk to the lua script which can then do any verification and processing.

The running tasks can only communicate through the MoonGen API, for example via pipes and namespaces, as tasks are separate Luajit VMs in a different address space.

A graphical representation of the just described data and control flow can be seen in Figure 1. Only the custom scripts ("Userscript") are visible to the user, which then communicates with the config or data api with the hardware device.

doi: 10.2313/NET-2016-07-1_06

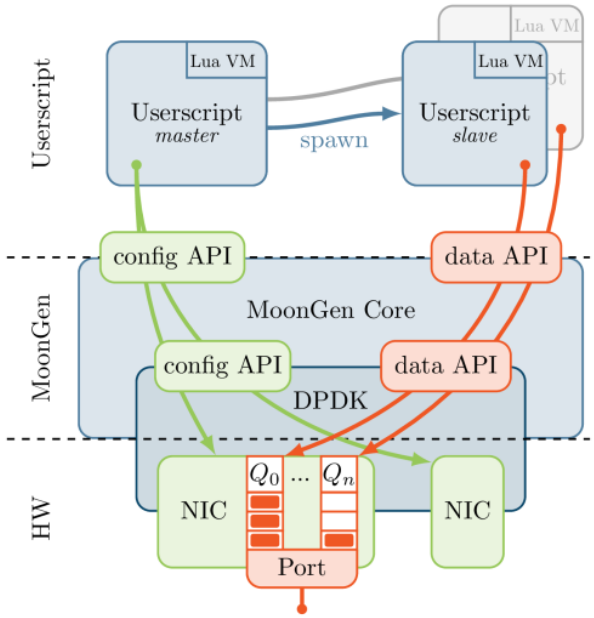

Figure 1: MoonGen architecture [2]

4. USAGE INTRODUCTION

After successful compilation of MoonGen, the invocation is very simple:

```
./MoonGen yourscript.lua [yourargs...]
```

`yourscript.lua` is your lua file is the entry script. It must contain a `function master(...)` which then performs the device setup, described in Section 4.1. Once this is done, worker tasks can be spawned as explained in Section 4.3, which then perform their duty to generate packets (Section 4.2), receive and count them (Section 4.4), communicate with each other (Section 4.6) or do whatever is appropriate for your use case.

4.1 Device setup

Before a NIC (network interface card) can be used, it must be taken away from the Linux driver so that dpdk can use it. The Python script `deps/dpkg/tools/dpdk_nic_bind.py` can detach PCI devices and set their driver to e.g. `igb_uio` or `vfio-pci` to use them with MoonGen.

Modern network interfaces have hardware features that allow a huge speedup by parallelization: They have send and receive queues, around 32 to 128 each (depending on NIC model) which allow to prepare sending out packets in parallel or assign received packets already on hardware by custom filter rules like hashing protocol header fields. This comes in handy for multicore processing, as the queues can be assigned to processors or threads independently [3].

In MoonGen, the queues are used independently and are usually requested in the entry point script. The devices are aquired and set up, queues registered and then, for example, used to send some packets.

```
local device = require "device"

function master(txNum)
    -- use specified NIC number with
    -- no listening and one transmission queues
    txDev = device.config{
        port = txNum,
        rxQueues = 0,
        txQueues = 1,
    }
    device.waitForLinks()
    send(txDev:getTxQueue(0))
end
```

To interact with an allocated queue, it is fetched from the device object by calling `queue = txDev:getTxQueue(nr)` or `getRxQueue`. You'll see this in Section 4.2.

To use the receive filter configuration mentioned earlier, configure the device upon creation. When calling `device.config`, set `rssNQueues = N` to the number of queues where packets shall be placed in. This enables automatic hashing by IPv4/6 and TCP/UDP headers to place same-header packets (from the same "flow") into the same queue selected from `0` to `N-1`.

The optional `rssFunctions` parameter controls which of the hash functions are enabled. If you don't specify it, all supported hashes are enabled. You can create a list of methods you want to use out of `RSS_FUNCTION_IPV4, ..._IPV6, ..._IPVX_TCP, ..._IPVX_UDP`:

```
local device = require "device"
txDev = device.config{
    port = 0,
    rxQueues = 4,
    rssNQueues = 4,
    rssFunctions = {
        device.RSS_FUNCTION_IPV4,
        device.RSS_FUNCTION_IPV4_TCP,
    }
}
```

4.2 Packet generation

To compose the data to send, a memory buffer is required first. As packets are sent out asynchronously, the buffers where you are crafting them must be allocated independently, in a buffer array. The array manages many allocations of same-sized packets, maintained in a memory pool. When the data was actually sent out, the allocated buffer can be freed from the array.

In such a buffer, most data will stay the same though, so the skeleton is defined via a function previously. It's important to set up the default values of the packet in this function for the memory pool and not in the generation loop. Otherwise, performance problems will arrise.

In this example, the most simple-stupid way is used to manually set up an ethernet header in a `char` buffer.

doi: 10.2313/NET-2016-07-1_06

```lua
local dpdk = require "dpdk"
local memory = require "memory"

function send(queue)
    local mem = memory.createMemPool(function(buf)
        local data = ffi.cast("uint8_t*", buf.pkt.data)
        for i = 0, 11 do
            data[i] = i  -- fill in mac addresses
        end

        data[12] = 0x12  -- set type to ethernet
        data[13] = 0x34
    end)
    local bufs = mem:bufArray()
    while dpdk.running()
        bufs:alloc(60)   -- size of each packet
        -- ⇑ sets up each packet with the function above
        -- ← here, single packets could be modified
        queue:send(bufs) -- schedule sending
    end
end
```

To simplify crafting of packets, the raw buffer can be casted into easy-to-use protocol header objects. Those conversions are defined in `lua/include/proto/` for all kinds of protocols, for example `getIP6Packet()` or `getUdp4Packet()`.

```lua
local mem = memory.createMemPool(function(buf)
    buf:getEthernetPacket():fill{
        ethSrc = txDev,  -- use device mac
        ethDst = "00:01:02:03:04:05",
        ethType = 0x1234,
    }
end)
```

If you want to implement a new protocol packet format, please copy and adapt the template file provided in `lua/include/proto/newProtocolTemplate.lua`.

To change some data for single packets, perform your operation after allocating the buffer array and before enqueuing the send-out. You can change any data of the packet and again use the convenience casts for implemented protocols. To configure the hardware transmission rate of a queue, use `queue:setRate(Mbit/s)`.

If you need to to pause the LuaJIT VM for some time period, call the `dpdk.sleepMillis(time)` function.

The following send function will only transmit data for 10 seconds, then it terminates. It creates UDP on IPv4 packets with a randomized source address.

```lua
local timer = require "timer"

function send(queue)
    queue:setRate(100)      -- hardware rate in Mbit/s
    dpdk.sleepMillis(1000)  -- wait one second
    local mem = memory.createMemPool(function(buf)
        buf:getUdp4Packet():fill{
            pktLength = 124,
            ethSrc = queue,  -- device mac
            ethDst = "10:11:12:13:14:15",
            -- ipSrc will be randomized
            ip4Dst = "10.13.37.1",
            udpSrc = 4321,
            udpDst = 1234,
            -- payload = \x00 (mempool initialization)
        }
    end)

    local bufs = mem:bufArray()
    local runtime = timer:new(10)  -- 10 seconds

    while runtime:running() and dpdk.running() do
        bufs:alloc(250)
        for _, buf in ipairs(bufs) do
            local pkt = buf:getUdpPacket()
            -- select a randomized source IP address
            pkt.ip4.src:set(
                parseIPAddress("10.0.42.1")
                + math.random(235))
        end
        bufs:offloadUdpChecksums()  -- harware checksums
        queue:send(bufs)
    end
end
```

4.3 Running parallel tasks

While running, MoonGen often needs parallel tasks: To send and receive packets, to create and write out statistics and counters or to do response verification. To achieve this, "slave" tasks are spawned.

The function used for this is `dpdk.launchLua("funcname", arg0, ...)`, which spawns a single slave task as a new LuaJIT VM. The task can then execute any code within the called function. Arguments are passed, this allows you to access e.g. the devices or queues within the task. The slave tasks can also be created dynamically on demand, although this should be used rarely to avoid spawning new VMs too quickly and often.

```lua
local dpdk = require "dpdk"
dpdk.launchLua("somefunctionname", arg0, arg1, ...)
dpdk.launchLua("otherfunction", txDev, ipaddr)
dpdk.waitForSlaves()  -- wait for child termination
```

In such a task, contents of every single received packet can be processed. The data arrives in batches, so the analysis has to loop over all packets in that batch.

```lua
local dpdk = require "dpdk"
local memory = require "memory"

function master(txPort, rxPort)
    local txDev = device.config{port = txPort}
    local rxDev = device.config{port = rxPort}
    device.waitForLinks()
    dpdk.launchLua("send", txDev:getTxQueue(0))
    dpdk.launchLua("recv", rxDev:getRxQueue(0))
    dpdk.waitForSlaves()
end

function recv(queue)
    local bufs = memory.bufArray()
    while dpdk.running() do
        local rx = queue:recv(bufs)
        for i = 1, rx do
            local pkt = bufs[i]:getUdp4Packet()
            print("Packet: " .. pkt.ip4:getString())
        end
        bufs:freeAll()
    end
end
```


doi: 10.2313/NET-2016-07-1_06

4.4 Statistics

The `stats` module allows counting packets for statistics. After statistics were gathered, they can be written to `stdout` or to some file. The supported formats are `"plain"`, `"csv"` and `"ini"`.

To create a new packet counter that is attached to a device, call `local rxCtr = stats:newDevRxCounter(device, "plain")` or `newDevTxCounter(..)`. Information is gathered automatically by performing queries to the harware counters of the NIC.

The `newPktRxCounter("your counter name", "plain")` or `newPktTxCounter(..)` can be updated by passing packet buffers to them via its `countPacket(singleBuffer)` method.

The `newManualTxCounter("your counter name", "plain")` is suitable for counting packets and other data manually. `updateWithSize(packet_count, each_size)` must be called to increase the datameter internally.

For all those counters, their `:update()` method should be called regularly, as it will show current statistics at runtime.

The `histogram` module allows gathering statistics about a frequency distribution.

The next example incorporates the device and package counters, it just listens for packets on the given hardware port and counts the occurrences of UDP ports. The packet sizes are logged in a histogram.

```
local dpdk = require "dpdk"
local device = require "device"
local histogram = require "histogram"
local memory = require "memory"
local stats = require "stats"

function master(rxPort, saveInterval)
    local saveInterval = saveInterval or 60
    local rxDev = device.config{
        port = rxPort,
        dropEnable = false,
    }
    device.waitForLinks()

    local queue = rxDev:getRxQueue(0)
    local bufs = memory.bufArray()

    -- create the device receive counter
    local rxCtr = stats:newDevRxCounter(queue.dev)
    -- and the packet receive counter to detect
    -- packets that were dropped on the NICNIC
    local pktCtr = stats:newPktRxCounter("pkts", "plain")

    local hist = histogram:create()
    local timer = timer:new(saveInterval)
    while dpdk.running() do
        -- wait max 100ms for new data
        local rx = queue:tryRecv(bufs, 100)
        for i = 1, rx do
            local buf = bufs[i]
            local size = buf:getSize()
            hist:update(size)
            pktCtr:countPacket(buf)
        end
        bufs:free(rx)
        rxCtr:update()
        pktCtr:update()
        if timer:expired() then
            timer:reset()
            hist:print()
            hist:save("packet_sizes.csv")
        end
    end

    -- and print statistics, those should be the same.
    rxCtr:finalize()
    pktCtr:finalize()
end
```

To use the manual counter, the return value of `queue:send()` can be used. Note here that the send call is asynchronous, but the return value can still be recorded for statistics. As in the previous example, the `finalize()` call actually prints the final result, and `updateWithSize` prints the runtime status every second:

```
function send(queue)
    local mem = ...
    local packetSize = 250

    -- create manual counter
    local txCtr = stats:newManualTxCounter(port, "plain")
    local bufs = mem:bufArray()

    while dpdk.running() do
        bufs:alloc(packetSize)
        bufs:offloadUdpChecksums()
        local sentCount = queue:send(bufs)

        -- register new data: sentCount * packetSize
        txCtr:updateWithSize(sentCount, packetSize)
    end
    txCtr:finalize()
end
```

It's also easily possible to collect statistics about packet contents, for example their UDP destination port. The task is blocked until some data is received. Then, all packet buffers received are casted to UDP in IPv4 packets, which then trigger a counter creation, if it doesn't exist already. This demonstrates that counters can be created and updated dynamically as well.

```
function recv(queue)
    local bufs = memory.bufArray()
    local counters = {}

    while dpdk.running() do
        -- block until some data was received
        local rx = queue:recv(bufs)
        for i = 1, rx do
            local buf = bufs[i]

            -- cast the buffer to a known protocol
            local port = buf:getUdpPacket().udp:getDstPort()
            local ctr = counters[port]

            -- create counters dynamically
            if not ctr then
                ctr = stats:newPktRxCounter(port, "plain")
                counters[port] = ctr
            end
```


doi: 10.2313/NET-2016-07-1_06

```
            -- record the packet
            ctr:countPacket(buf)
        end
        bufs:freeAll()
    end

    -- for each observed destination port, print stats:
    for _, ctr in pairs(counters) do
        ctr:finalize()
    end
end
```

4.5 Timestamping

To measure sub-microsecond delays in fiber and copper cables, MoonGen can utilize hardware timestamping features from modern NICs. The packets sent are defined in the `lua/include/proto/ptp.lua`, the precision time protocol. You can create a timestamper to use either as a layer 2 (via `timestamping:newTimestamper(txq, rxq)` or transfer it as PTP via UDP in IPv4 with `timestamping:newUdpTimestamper`.

This example measures the latency between both queues via hardware timestamping every 0.01 seconds. The queues have to be connected at the peer side, so the sent packet can take a round trip.

```
local ts = require "timestamping"

function timerTask(txq, rxq, size)
    -- create the timestamper for measuring
    -- between those queues
    local timestamper = ts:newTimestamper(txq, rxq)
    local hist = histogram:new()
    local rateLimiter = timer:new(0.01)
    while dpdk.running() do
        rateLimiter:reset()
        hist:update(timestamper:measureLatency(size))
        rateLimiter:busyWait()
    end
    hist:print()
    hist:save("histogram.csv")
end
```

4.6 Task communication

The simplest inter-task communication API provided by MoonGen are pipes. For example, you can send rate adjustment messages, pass statistics or transfer any communication that is not performance-critical through a pipe shared by two tasks. To use, create the pipe in a common context, for example the master function. This pipe can communicate accross LuaJIT VMs and can send arbitrary data, which is serialized and unserialized using the “serpent” library.

```
local pipe = require "pipe"

-- create a new pipe in the parent task
local p = pipe:newSlowPipe()
p:send(0, 13, 37, 42)           -- send array
p:send("the cake is a lie")     -- send string
-- or send a table
p:send({235, lol = "rofl", subtable = {1}})

-- number of waiting messages
local enqueued = p:count()

-- receiving
local a, b, c, d = p:recv() -- equals tryRecv(10)
local txt = p:tryRecv(100)  -- wait time microseconds
                            -- and return answer

-- pass this pipe when creating another task
-- it can then access it like above.
dpdk.launchLua("somefunction", p)
dpdk.waitForSlaves()
```

The alternative to pipes are namespaces, which are global variables between LuaJIT VMs, implemented as lua table. They are also slow like the pipes from above, as data is transferred between VMs.

`local space = namespaces::get("name")` will create or fetch an already existing global namespace, which can then be accessed like this:

```
local dpdk = require "dpdk"
local namespaces = require "namespaces"

function master()
    local space = namespaces:get("mine")
    space.string = "data!"
    space.answer = 42
    space.table = { black = "mesa", { 1 } }
    dpdk.launchLua("slave"):wait()
end

function slave()
    -- can access the same namespace!
    local slavespace = namespaces:get("mine")
    print("data? " .. slavespace.string)
end
```

4.7 Traffic patterns

The hardware rate control feature of some NICs can only be set to a constant rate. To send non-constant traffic patterns of valid packets, MoonGen fills the gaps between the packets with invalid data. That way, the sending card is kept busy and times sends correctly, but devices along the tested way will hopefully drop the broken packages and process only the correct ones. The send delay is configured in bytes, which equals the amount of garbage, as seen in Figure 2. Various traffic patterns like exponential distributed bursts are easily possible with that approach.

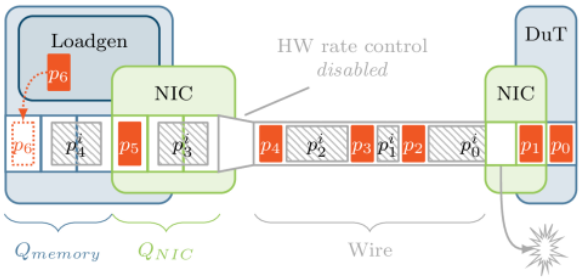

Figure 2: MoonGen rate control [2]

doi: 10.2313/NET-2016-07-1_06

A MoonGen packet buffer can be set a waiting gap duration by `buf:setDelay(bytes)`. On 10GbE one byte would have a delay of 0.8 nanoseconds. A randomly exponentially-distributed delay can be generated with the poisson process by `poissonDelay(average_wait)`, where the parameter specifies the average wait time between two packets. The exponentially-distributed wait time can directly be fed into `setDelay(poissonDelay(..))`, this will then be the amount of garbage sent out between real packets.

```
function send(txDev)
    local mem = ...
    local bufs = mem:bufArray()
    while dpdk.running() do
        bufs:alloc(size)
        for _, buf in ipairs(bufs) do
            local avg = rateToByteDelay(rate, size)
            local delay = poissonDelay(avg)
            buf:setDelay(delay)
        end
        queue:sendWithDelay(bufs)
    end
end
```

5. CONCLUSION

This introduction should have prepared you to achieve any measurement task in MoonGen. Complete and directly executable examples are shipped with MoonGen in its `examples/` subfolder. This tutorial should have prepared you to implement your particular test setup and understand complex examples like `router.lua`. With namespace, you can try implementing an `ARP` lookup task that figures out peer addresses for your packet crafting.

If you encounter any issue while using MoonGen and think it's not your or your setup's fault, please create an issue at https://github.com/emmericp/MoonGen.git/issues so the developers can assist you or fix the bug you may have discovered. We hope you enjoy using this tool and encourage you to improve it further and adapt it to your needs, as it's free software for a reason.

References

[1] Data Plane Development Kit. http://dpdk.org/. Accessed: 2015-12-05.

[2] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference 2015 (IMC'15)*, Tokyo, Japan, October 2015.

[3] Linux network stack scaling. https://www.kernel.org/doc/Documentation/networking/scaling.txt. Accessed: 2015-12-19.

[4] Lua tutorial. https://learnxinyminutes.com/docs/lua/. Accessed: 2015-12-19.

[5] MoonGen repository. https://github.com/emmericp/moongen. Accessed: 2015-12-10.

[6] Mike Pall. Luajit. http://luajit.org/. Accessed: 2015-12-05.

doi: 10.2313/NET-2016-07-1_06