SIGCOMM 2022
Amsterdam
ACM Special Interest Group on Data Communication (SIGCOMM), 36th edition.
August 22 – 26, 2022, Amsterdam, the Netherlands.

Programming socket-independent network functions with nethuns

Nicola Bonelli, Fabio Del Vigna, Alessandra Fais,
Giuseppe Lettieri, Gregorio Procissi

UNIVERSITÀ DI PISA

Agenda

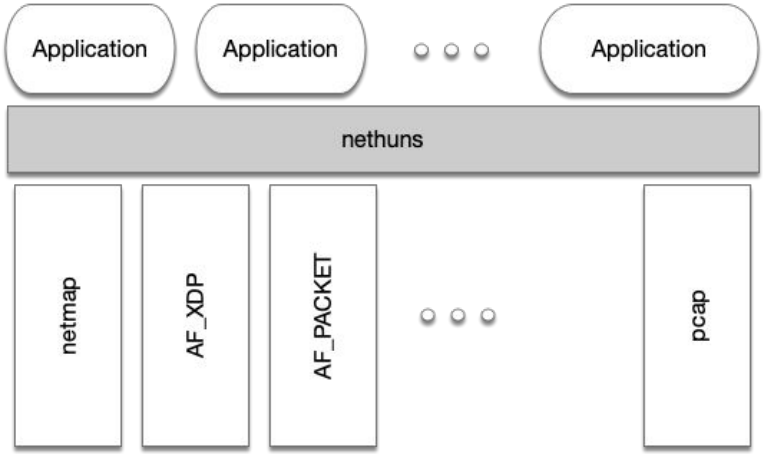
- Motivations
- The `nethuns` architecture
- An essential API
- Engine specific discussion
  - `AF_XDP` and `netmap` cases
- Performance
  - Speed tests
- Practical use cases
  - High-speed traffic replay
  - Open vSwitch

Motivations

- Software data planes are very popular and flexible solutions in real deployments
  - need accelerated I/O to scale up with speed
    - `netmap`, `AF_XDP`, …, each of them with its own "grammar" (API)
  - lack of portability
- `nethuns`: a **unified abstraction layer** against multiple network APIs
  - `nethuns` is open source and available at `https://github.com/larthia/nethuns`

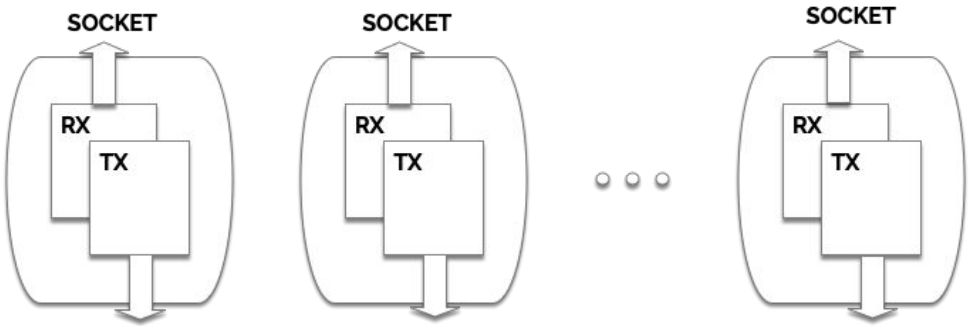
nethuns at a glance

- `nethuns` is a user-space library
  - offers a socket-like API implemented by multiple network I/O engines
- Apps built on top of `nethuns` must select an I/O engine at compile time
  - this allows compile-time specialization/optimization
  - porting the app over another engine/system only requires recompiling the app itself



nethuns general API

- **Socket** abstraction
  - points of access to hardware interface queues
  - not a “real” socket though
- **Ring** abstraction
  - circular queue of packet descriptors
  - include packet metadata and an *in-use* flag



- The actual memory for rings and packet buffers is allocated and exposed/disposed *through* the library
- Each socket can have *at most one ring* for direction (TX/RX)

nethuns at work: opening a socket

- The first step is opening a socket

```
sock = nethuns_open(opt)
```

- The list of options is quite long…
  - General options
    - `rx`, `tx`, both, ring sizes, zero copy, blocking/non-blocking, etc…
  - Engine specific options
    - name of the `eBPF` program to inject in the kernel for preprocessing

nethuns at work: binding the socket

- The socket must be bound to a network device
  - A NIC or a specific hw queue of a NIC

```
nethuns_bind(sock, dev, queue)
```

- Finally, to close the socket

```
nethuns_close(opt)
```

Receiving packets

```
pktid = nethuns_recv(sock, &pkthdr, &pkt)
```

- Returns
  - the identifier to the next packet in the stream
  - two pointers to
    - a packet header containing metadata
    - a **buffer** containing the full packet
- Once data has been consumed, the buffer must be returned to the library through

```
nethuns_release(sock, pktid)
```

Nethuns also provides an optimized cache-friendly SPSC queue for dispatching packets to multiple workers!

Transmitting packets

- Packet is queued for transmission by

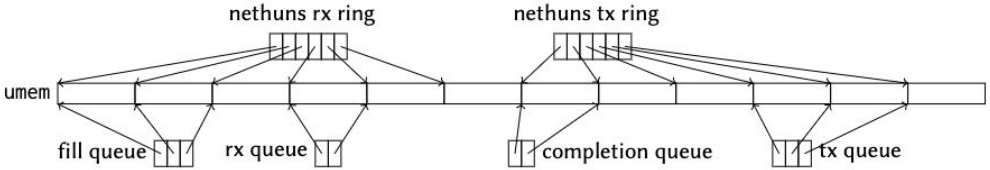
```
nethuns_send(sock, pkt, size)
```

- To wake up transmission, need to call

```
nethuns_flush(sock)
```

nethuns and AF_XDP (1)

- When a nethuns socket is opened, the library creates
  - `rx` and/or `tx` rings
  - the `umem`, with buffers permanently mapped to the corresponding nethuns slot



- `nethuns_recv()`
  - scan for released slots and push them in the fill queue
  - scan for freshly arrived frames in the rx queue and mark the corresponding ring slot accordingly
- Similar behavior is performed on transmission by the `nethuns_flush()`

nethuns and AF_XDP (2)

- In the RX path, `AF_XDP` uses the `eBPF/XDP` infrastructure to divert packets from their "canonical" path (through the kernel)
  - need to download and attach to the `XDP` hook an `eBPF` program to redirect each packet
  - `nethuns` default `XDP` program diverts all packets to `AF_XDP` (at user space) and manages the kernel maps for opening/closing new sockets
- User defined eBPF program can be loaded by specifying its name in the options upon socket opening
- The **`nethuns_close()`** function unloads the XDP program attached to the interface

nethuns and netmap

- `netmap` abstracts hw rings through "`netmap` rings"
  - the number of slots in a `netmap` ring is bound to the HW queue size
  - each ring contains pointers to `netmap` buffers (which contain the full packet) shared with the hardware
- The number of `netmap` buffers is configurable (so-called "extra buffers")
  - use this feature to allocate a `netmap` buffer for each `nethuns` ring
  - allow to decouple nethuns rings from netmap rings

Performance Evaluation

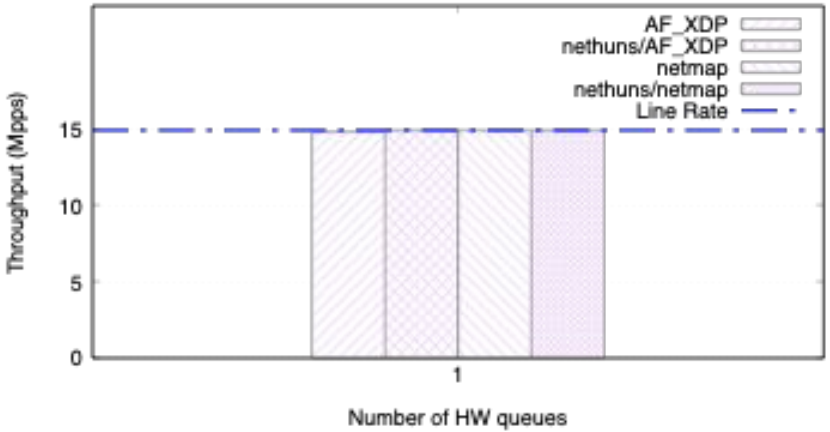
- `nethuns` is an overlay to standard and accelerated network I/O sockets
  - need to access its performance overhead (if any)
  - very simple testbed: sender/receiver 8 core XEON servers running native and *nethunized* apps… let's see what happens
    - nethuns-send and nethuns-meter for TX/RX
    - pkt-gen (`netmap`) and xdpsock (`AF_XDP`)

Speed test: packet transmission

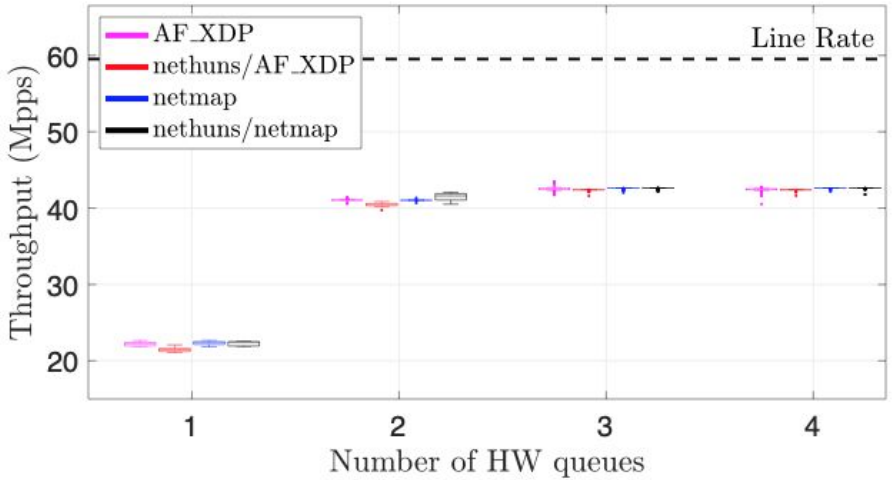
pkt-gen (netmap)
xdpsock (AF_XDP)
nethuns-send

pkt-gen (netmap)

10 Gbps



40 Gbps



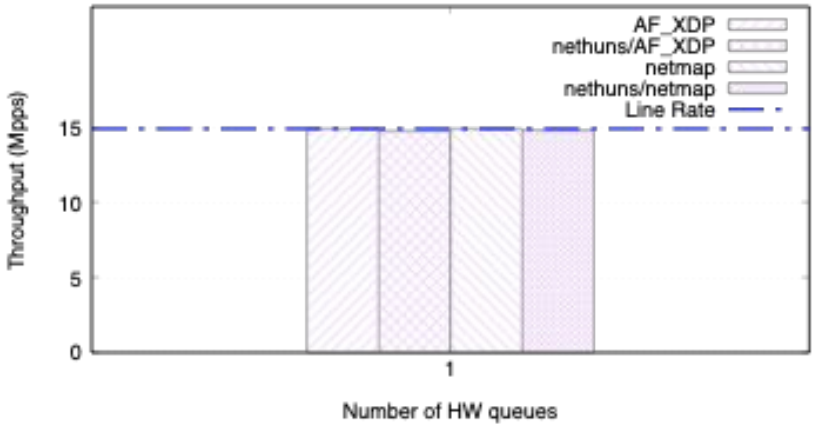
Speed test: packet capture

pkt-gen (netmap)

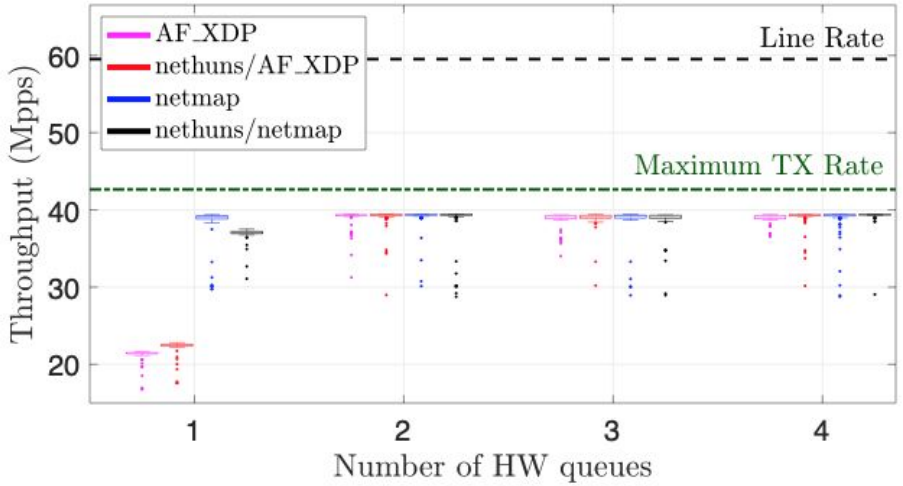
10/40 Gbps

pkt-gen (netmap)
xdpsock (AF_XDP)
nethuns-meter

10 Gbps



40 Gbps



Use case: traffic generator

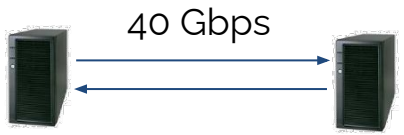
- **`nmreplay`** (from the `netmap` ecosystem) replays packets from a `pcap` trace at configurable speed
  - reaches around 21 Gbps speed over a 40Gbit link when playing a specific `pcap` dump
- ported the code on top of `nethuns`
  - replace the `netmap` API calls to those of `nethuns`
  - quick and painless… in around an hour of work we got a traffic generator for `AF_XDP` too!
- **same performance** with `nethuns` over both `netmap` and `AF_XDP`

Use case: Open vSwitch (1)

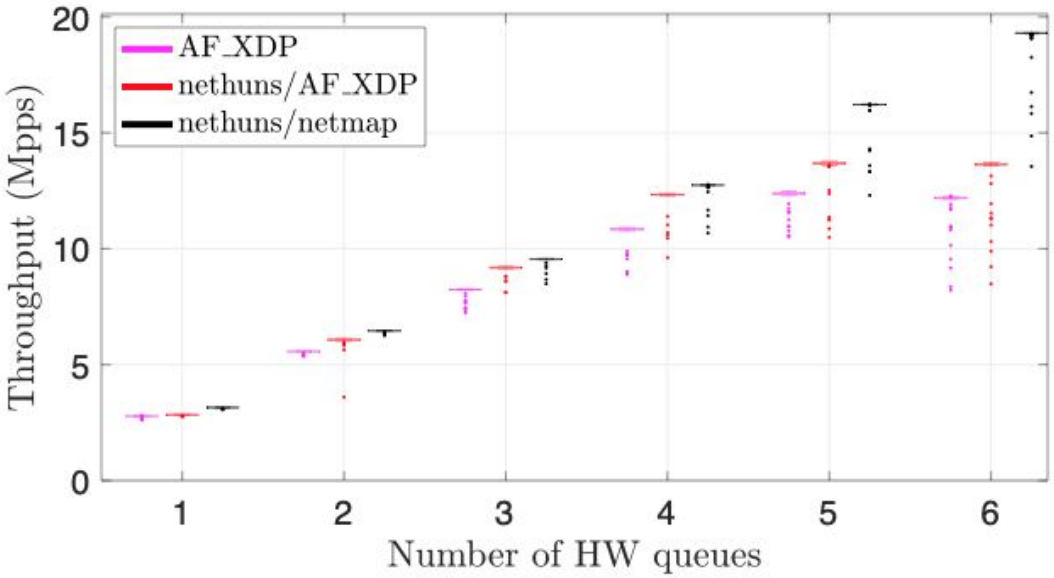
- user space datapath of `OVS` uses a generic netdev interface
  - existing netdevs: `DPDK`, `AF_XDP`
- Implemented a new `nethuns` netdev
  - the `AF_XDP` is much simpler (complexity buried within `nethuns`)
  - around 700 lines of code change
  - got `OVS` for `netmap` “for free”
- And performance?

Use case: Open vSwitch (2)

sender (@21 Mpps)
receiver



native (AF_XDP) **OVS**
nethuned (AF_XDP and netmap) **OVS**



Conclusion

- **`nethuns`** is user space library that provides a **socket independent** abstraction to network programming
- currently supported:
  - `AF_PACKET`, `libpcap`, `netmap`, `AF_XDP`
- **porting** an application over a different I/O framework requires **a simple recompilation**
- **no performance degradation** w.r.t. the use of native sockets
- `nethuns` is completely **open source**! Help yourself at:

  `https://github.com/larthia/nethuns`