# An Introduction to Userland Networking

## New Directions in Operating Systems

25.11.2014 Franco Fichtner operatingsystems.io, London

## Overview

## Author info: Franco

*"I like nifty things. I read a lot. Sometimes I write stuff."*

- o Chief Software Architect at Packetwerk
- o DragonFly BSD committer
- o libpeak (bootstrapping code)
- o minor: mandoc, manuals, pfSense, zmq, netmap, FreeBSD ports
- o new BSD firewall project coming soon...

*@fitchitis*

## Definition: status quo

- Everbody is doing it, very few talk about it.
- The minority shares viable open source code.
- The networking people have a long history of undermining established system design. :)
- No standards API, only shared ideas across different hardware and software with varying degrees of toolkit size.

## Definition: motivations

- o Speed. (Should go to 11.)
- o Complex architectures and code size may clog the kernel, cause panics. Stack limitations, etc.
- o Multithreading is a lot easier in userland.
- o Shared libraries reside in userland.
- o Observing context and content on the application layer.
- o Old school proxies are expensive and maybe a wee too opaque.
- o Coding is faster and cheaper in userland. Debugging...
- o Eventually portability across operating systems.
- o Kernel network stack protection.

## Definition: an attempt

Userland networking is the school of networking which moves as much of packet receive and send out of the kernel for performance and complexity reasons. It currently focuses more on transit traffic than endpoint traffic, but that may change. Motivations are plenty.

## Bottlenecks (1)

o Modern networking is all about alleviating bottlenecks
o Weakest link's throughput equals maximum performance
o A list of examples: sockets, context switches, manual memory copy, scheduling, IPC, mutexes, asynchronicity, cache misses, memory allocation, bus speed, memory access rates, CPU frequency, clock cycles, NUMA, data structures and eventually your own production code!

## Bottlenecks (1)

- Modern networking is all about alleviating bottlenecks
- Weakest link's throughput equals maximum performance
- A list of examples: sockets, context switches, manual memory copy, scheduling, IPC, mutexes, asynchronicity, cache misses, memory allocation, bus speed, memory access rates, CPU frequency, clock cycles, NUMA, data structures and eventually your own production code!

**Everything is a bottleneck. Solving one issue brings up the next one...**

## Bottlenecks (2)

gettimeofday(2) system call -- a simple clock service fail

- o each time the clock is fetched, the maximum throughput decreases
- o coalescing calls is better for performance but skews the notion of time in your system
- o solution: the clock needs to be piggybacked via packet grabbing code
- o god mode: the NIC does the timestamping at true nanosecond resolution :)

## Bottlenecks (3)

- any syscall is bad as it may yield the CPU and causes an expensive context switch to happen
- concurrent processes are bad as they steal your precious CPU time (no thanks to you, scheduler)
- any performance drop may cause the receive buffer to fill up, eventually causing packet drops

## Zero copy (1)

o zero copy isn't zero
o 1 copy into the memory inbound "from the wire"
o 1 copy out of the memory(*) outbound "to the wire"
o any more than those two should be avoided

(*) let's assume the packet was created by the host

## Zero copy (1)

o zero copy isn't zero
o 1 copy into the memory inbound "from the wire"
o 1 copy out of the memory(*) outbound "to the wire"
o any more than those two should be avoided

**Assuming each packet is copied once, that's the same rate of your bandwidth.**

(*) let's assume the packet was created by the host

## Zero copy (2)

- o solution: memory mapping between userland and kernel
- o contiguous ringbuffers, rings/slots or bulk IO
- o works great between NIC and userland, not so much in conjunction with the kernel (routing, endpoints)

## Frameworks: DPDK

- o Devs: 6WIND/Intel
- o OS: Linux, FreeBSD
- o License: BSD
- o Website: http://www.dpdk.org/
- o Pros: speed, stability
- o Cons: raw API, Intel hardware only (1G/10G/40G)

## Frameworks: PF_RING ZC

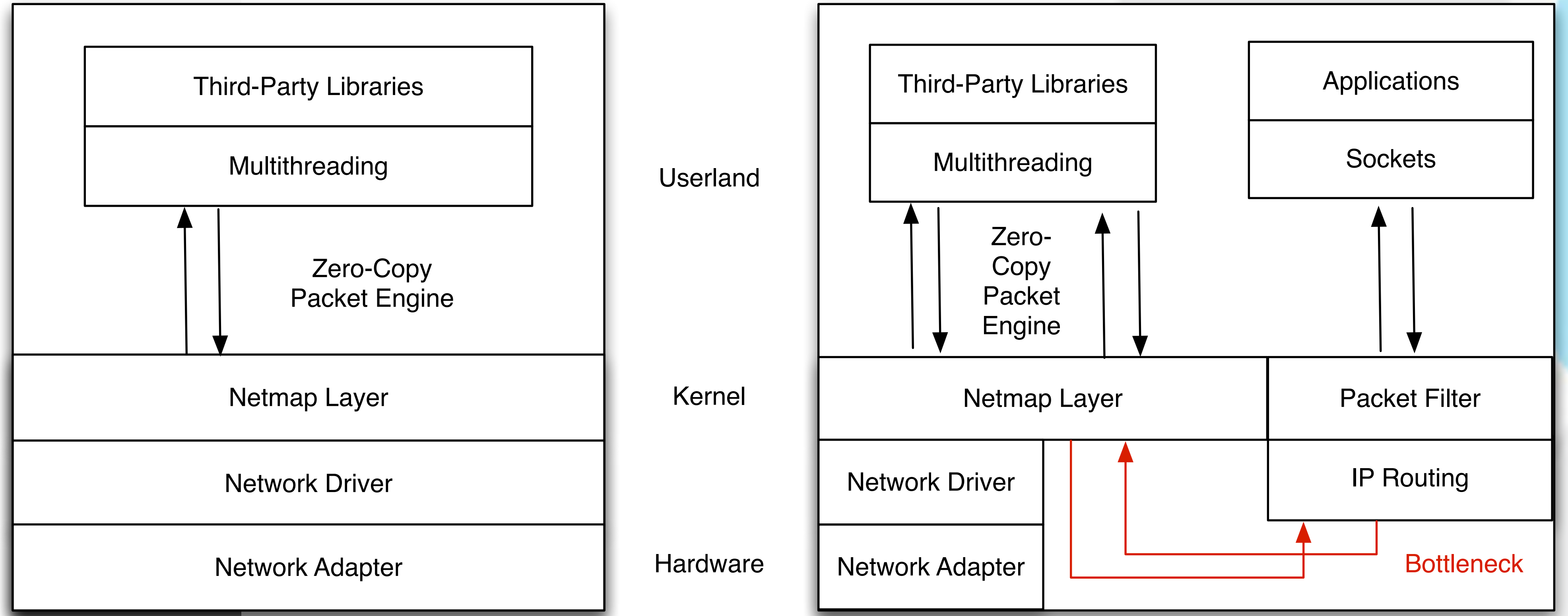
o Devs: ntop
o OS: Linux
o License: GPL
o Website: http://www.ntop.org/products/pf_ring/
o Pros: balancer API, theoretically cross-vendor
o Cons: bound to 1G/10G Intel hardware ;)

## Frameworks: Netmap (1)

o Devs: Luigi Rizzo / Pisa University
o OS: FreeBSD (since 2011), Linux (since 2013)
o License: BSD
o Website: http://info.iet.unipi.it/~luigi/netmap/
o Pros: broad driver support (*), emulation mode
o Cons: uses syscalls for synchronisation, API changes

(*) Realtek, Intel, Chelsio

## Frameworks: Netmap (2)



normal mode
(dev to dev)

transparent mode
(dev to host)

## Architecture: traditional use cases

o routing (no peeking)
o connect endpoints
o capture (and peek)

Packet receive (RX) is always great, packet send (TX) differs.

Userland scaling requires a load balancer for > 1G.

## Architecture: modern use cases

o peek, then: route, load balance, traffic shape/police, screen some more, accounting, security stuff, ...

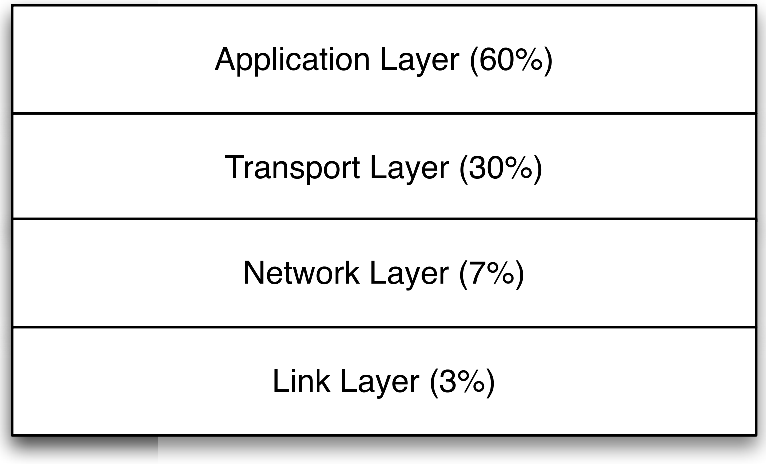
Endpoints are less important. Forwarding is the main operation.

Content/context driven use cases.

Buzz words: IPS, NGFW, UTM, Traffic Manager, etc.

# Architecture: "light-weight" stacks

The network stack focus changed drastically:



Everything is interesting here

Flow tracking, TCP, TLS, SSH

Addresses mostly

Interface/Addresses only

*Focus on connections, context, content, even users, as opposed to routing decisions or endpoint access.*

## Architecture: rules engines

o with vast knowledge comes great chaos
o applications generate immense amounts of data
o describe -> filter -> enforce policies

## Architecture: rules engines

o with vast knowledge comes great chaos
o applications generate immense amounts of data
o describe -> filter -> enforce policies

(1) hardcoding only gets you so far
(2) XML may seem flexible, but it's not
(3) writing good grammar is hard
(4) writing unambiguous grammar is harder
(5) yacc(1)/bison(1) is always right :)

## On the horizon: adaptive network stacks (1)

The network layer is the base for the Internet. Can't touch this.

The application layer is vast and well-defined (mostly).

That leaves the transport layer for us to play around with!

- o transport protocols are immutable because they are embedded into the kernel...
- o transport layer creates the backbone for today's inspection methods: content and connections...

## On the horizon: adaptive network stacks (2)

- ...userland network stacks are more flexible
- adaptive transport layer: more protocols, dialects, non-standard encryption and obfuscation, renegotiation
- end-to-end negotiation thereof

An easy way to disrupt the status quo in tracking can be achieved by reordering the TCP header. Two hosts may talk a standards-compliant TCP, but the connection can not easily be tracked, because it seems "weird" from the outside persepective.

## Reading Material

[1] libpeak: https://github.com/packetwerk/libpeak
[2] S. Gallenmüller, Comparision of Memory Mapping Techniques for High-Speed Packet Processing, http://t.co/fcL9VMpzjt
[3] L. Rizzo, Netmap: a novel framework for fast packet I/O, http://info.iet.unipi.it/~luigi/papers/20120503-netmap-atc12.pdf
[4] P. Kelsey: The FreeBSD TCP/IP Stack as a Userland Library, https://www.bsdcan.org/2014/schedule/attachments/
260_libuinet_bsdcan2014.pdf
[5] S. Alcock: lightweight application detection, https://github.com/wanduow/libprotoident

## That's it. Thank you! :)

Questions, comments, general thoughts?