Файловый и сетевой стеки в Userland: почему их нужно использовать в 2022 году

Константин Ушаков

Дисковый и сетевой стеки в Userland: почему их нужно использовать в 2022 году

Константин Ушаков

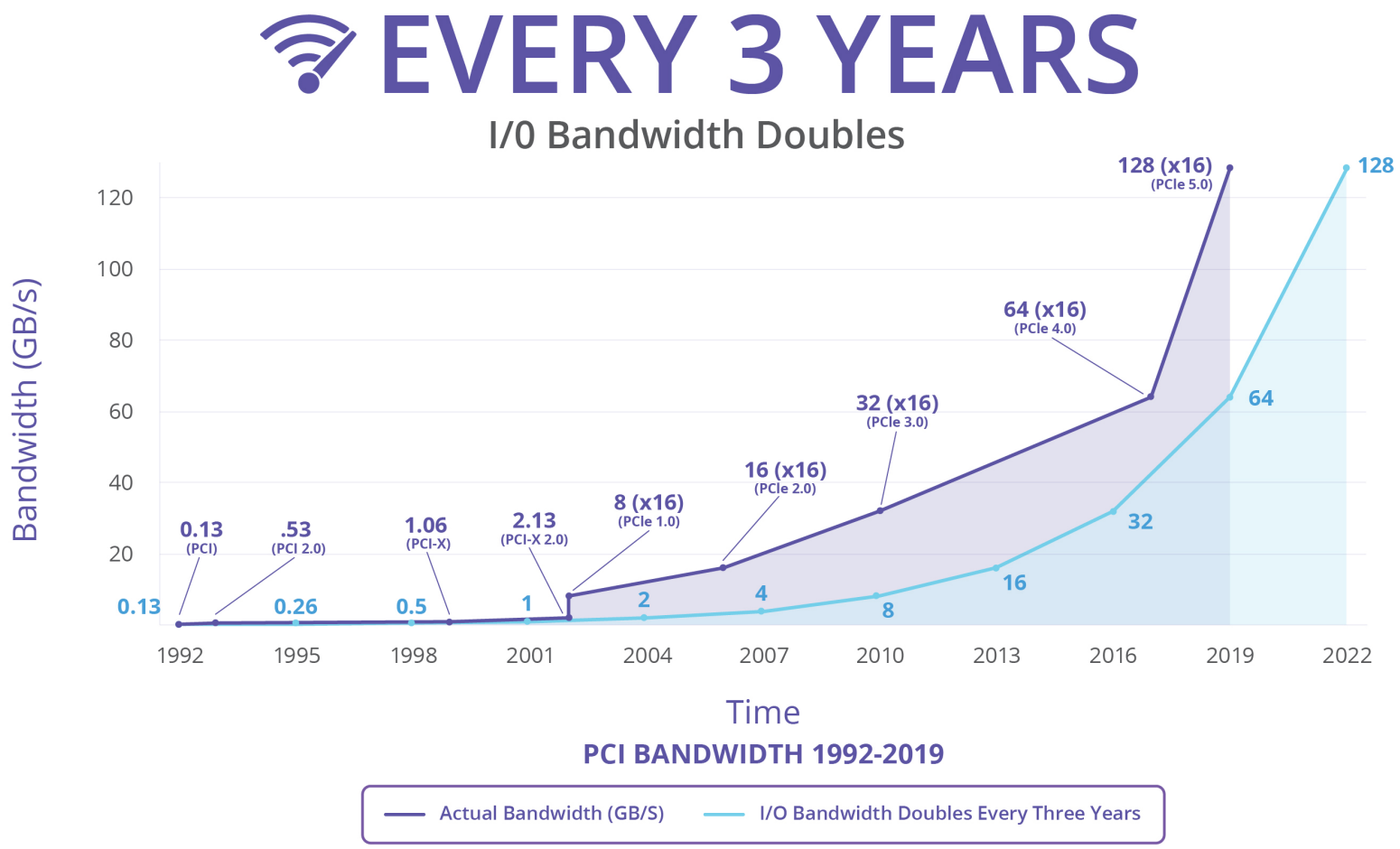
Content

- Why and what has changed?
- Networking
  - Kernel bypass networking
  - Speed-Stack Net
  - Performance
- Block
  - Speed-Stack Block
  - Performance
- Filesystems: stay tuned

Applications

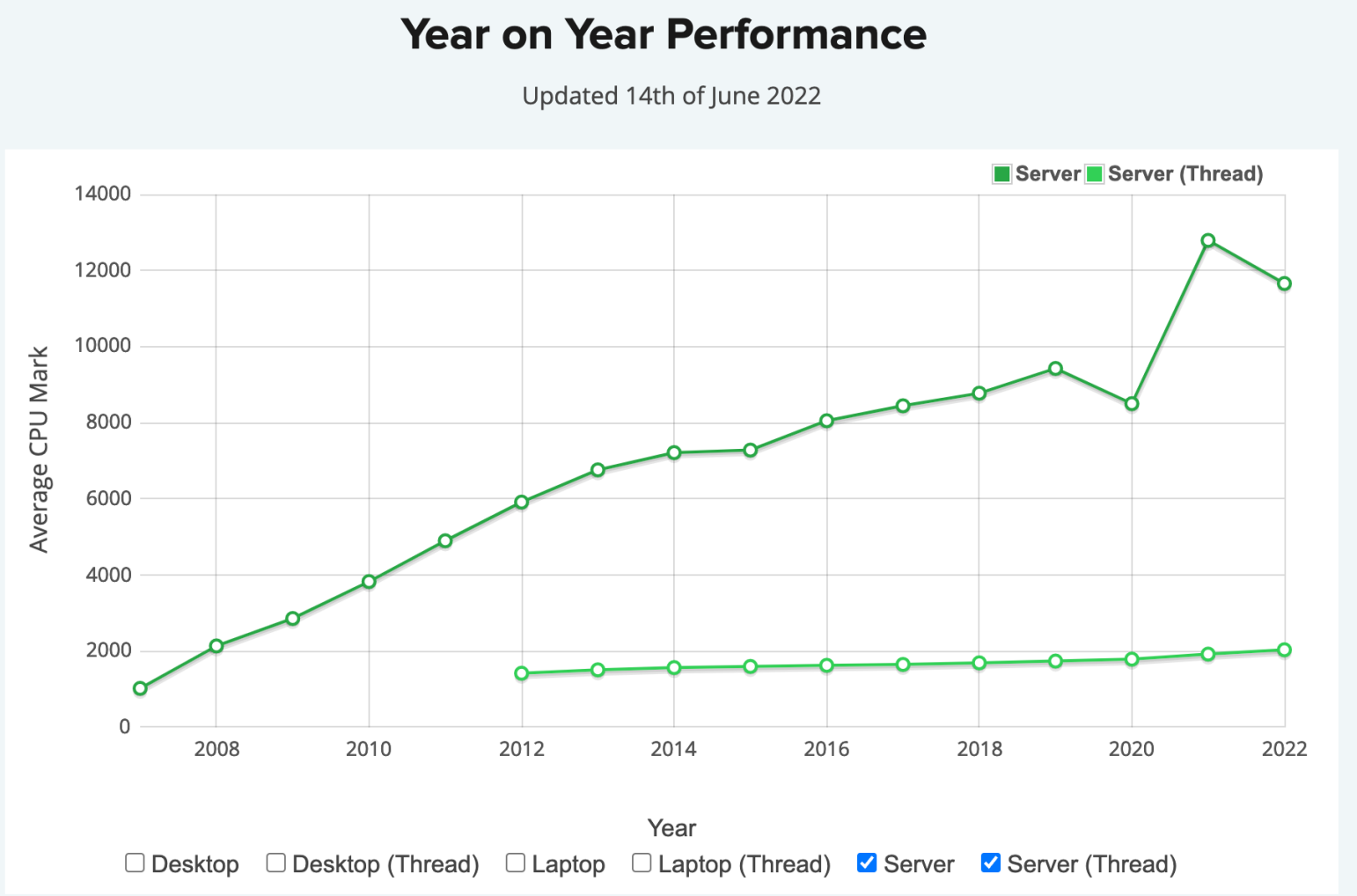
- Compute-intensive
  - GPU offload
  - FPGA offload
  - Cross-machine parallelism
- IO-Intensive
  - HW offload: SmartNICs/DPUs
    - Packets processing (csum, encryption)
    - Tunneling
    - HW interfaces (like NVMeoF)
  - Move processing to userland?

Change#1: PCIe is getting faster



https://www.anandtech.com/show/14447/pcisig-finalizes-pcie-50-specification

Change#2: More of the same cores



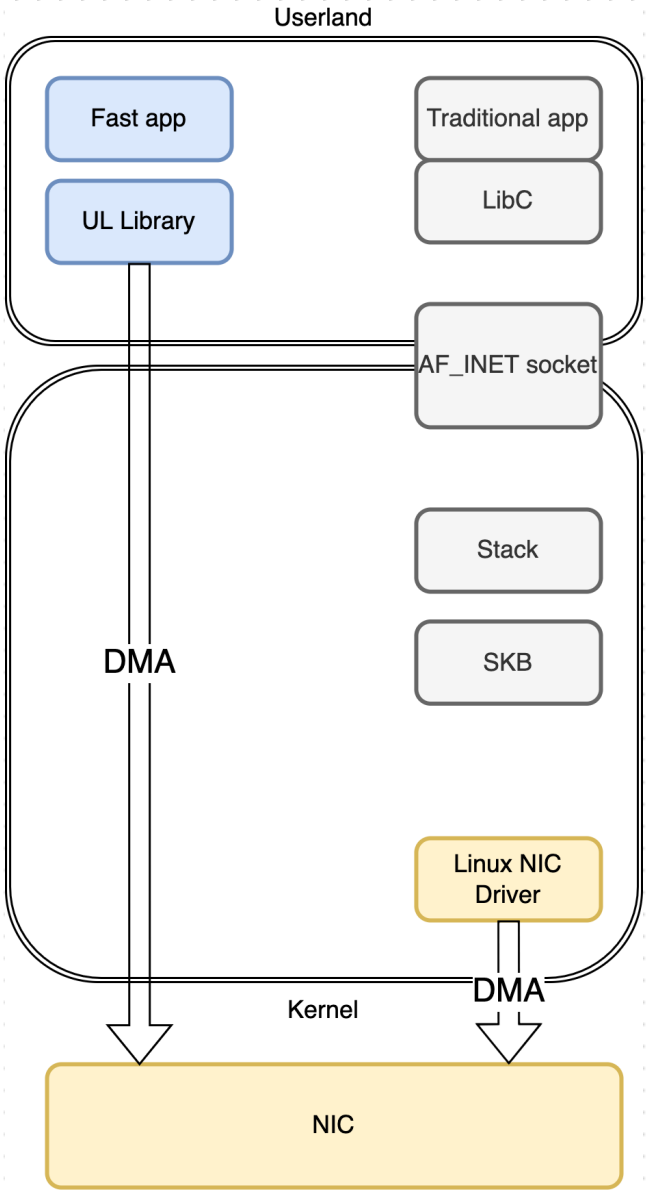
https://www.cpubenchmark.net/year-on-year.html

Content

- Why and what has changed?
- **Networking**
  - Kernel bypass networking
  - Speed-Stack Net
  - Performance
- Block
  - Speed-Stack Block
  - Performance
- Filesystems: stay tuned

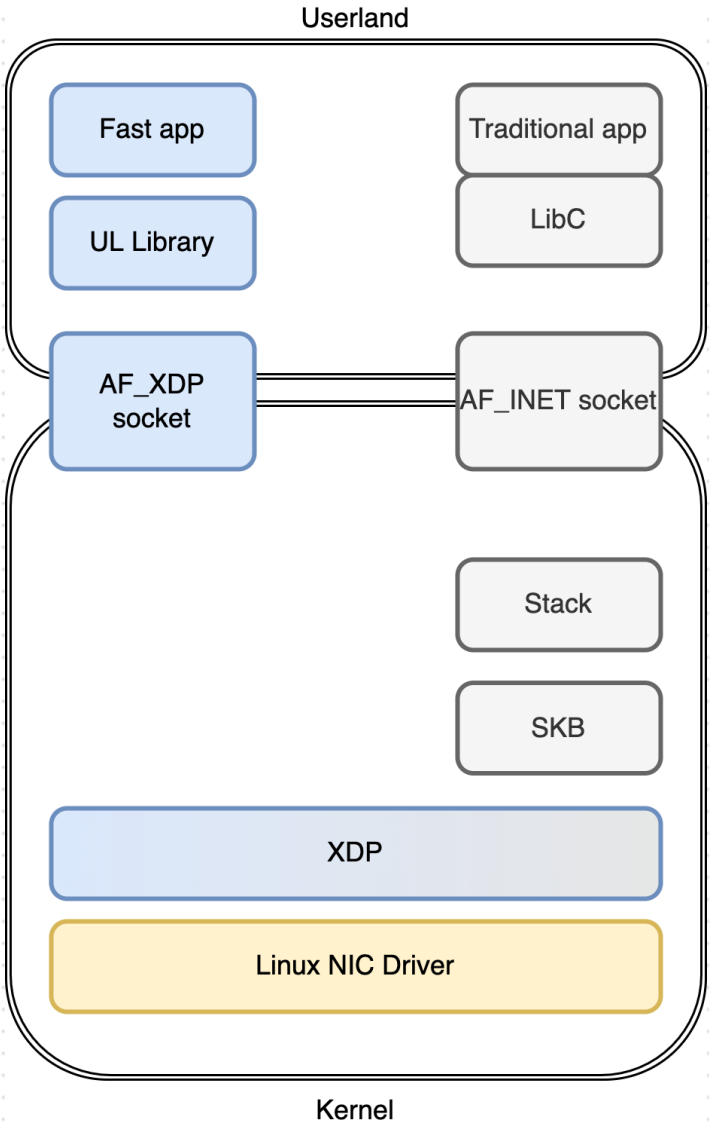
Net: kernel bypass to NIC

- NIC: Network Interface Controller
- Getting application closer to the network
- Working directly with the HW
- Requires individual NIC support in the UL library
- Very efficient and fast



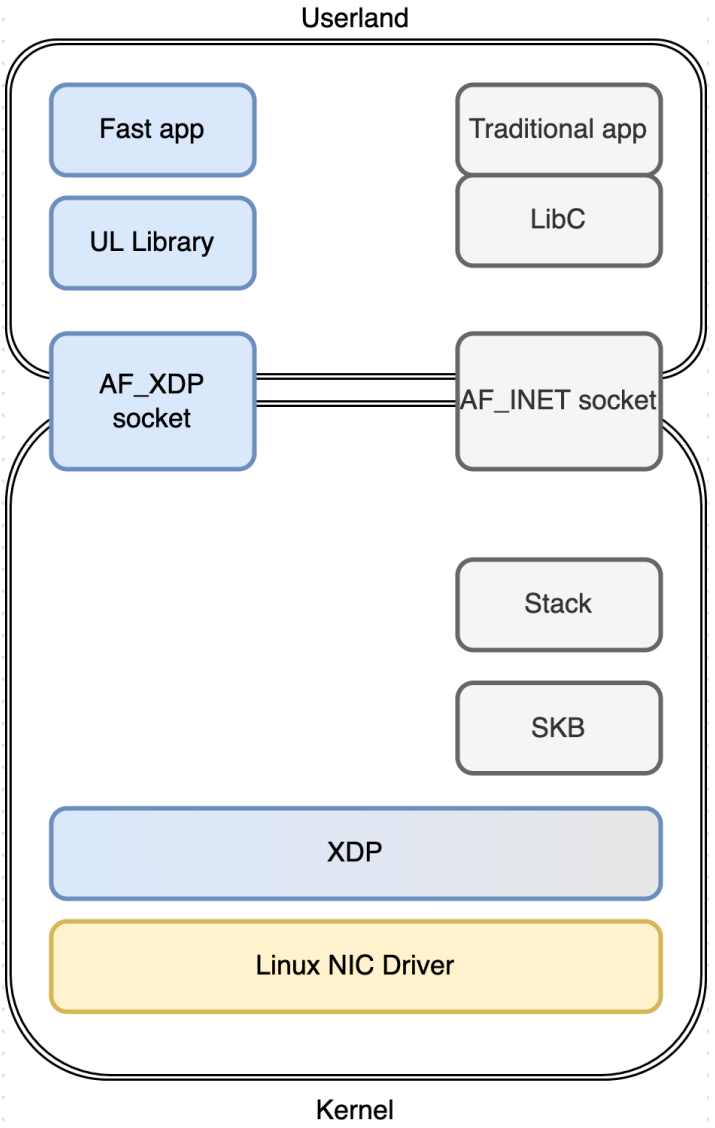
Net: kernel bypass to XDP

- XDP: eXpress Data Path
- Performance:
  - Zero copy between userland and kernel
  - ”Next gen” of AF_PACKET
- XDP programs redirect packets to userland buffers
- User can install eBPF program to do packet processing and forward packets to application



Net: kernel bypass to XDP

- Included in modern distros:
  - Ubuntu LTS 20.04
  - Ubuntu LTS 22.04
  - Debian 10 with Linux kernel 5.10
  - Debian 11
  - Redhat Enterprise Linux 8.3, 8.4
  - Linux kernel in the range 5.3 - 5.18+
- Requires NIC driver support to get reasonable performance:
  - Zero Copy for Transmission
  - Feed packets into XDP (aka XDP_DRV)
- If no driver support: XDP_SKB is used, and it's slow



How do we get there: Ideal world

- Limited engineering efforts
  - No application changes
  - No recompilation
  - No kernel patches
  - No protocol changes
- TCP/UDP stacks should come for free
  - Users should not have to write them!
- It should be possible for Fast and Traditional applications to co-exist
- Applications should just work!

Approaches: frameworks

- Socket API (socket(), send(), poll(), epoll_wait() etc.)
  - OpenOnload
  - **Speed-Stack Net (SSN)**: OpenOnload based
  - Libvma (NVIDIA/Melanox)
  - ExaNIC Sockets (Exablaze)
- Special APIs: require application update!
  - IO_uring
  - DPDK
  - Netmap
  - AMD EF_VI / TCP Direct
  - Infiniband verbs
  - etc.

HW specific vs. HW independent

- HW Specific
  - Libvma
  - ExaNIC Sockets
  - OpenOnload (mostly, has experimental single-threaded AF_XDP support)
  - Infiniband verbs
  - AMD ef_vi/AMD TCP Direct: HFT only
- HW Independent
  - **Speed-Stack Net (SSN)**
  - IO_Uring
  - DPDK (if driver exists or on top of AF_XDP)
    - Need to rewrite your application and implement your own stack
  - Netmap

Kernel Bypass: DPDK

- Still requires special HW
  - And your benefits depend on your HW
- PCI Function is either used for DPDK or for non-DPDK
  - Some vendors have workarounds
- Multi-process support : limited
  - Good handling of NUMA/CPU
- API is specific to DPDK: you have to modify your application
- No protocols support:
  - ANS : limited number of applications start; NGINX required modification to run

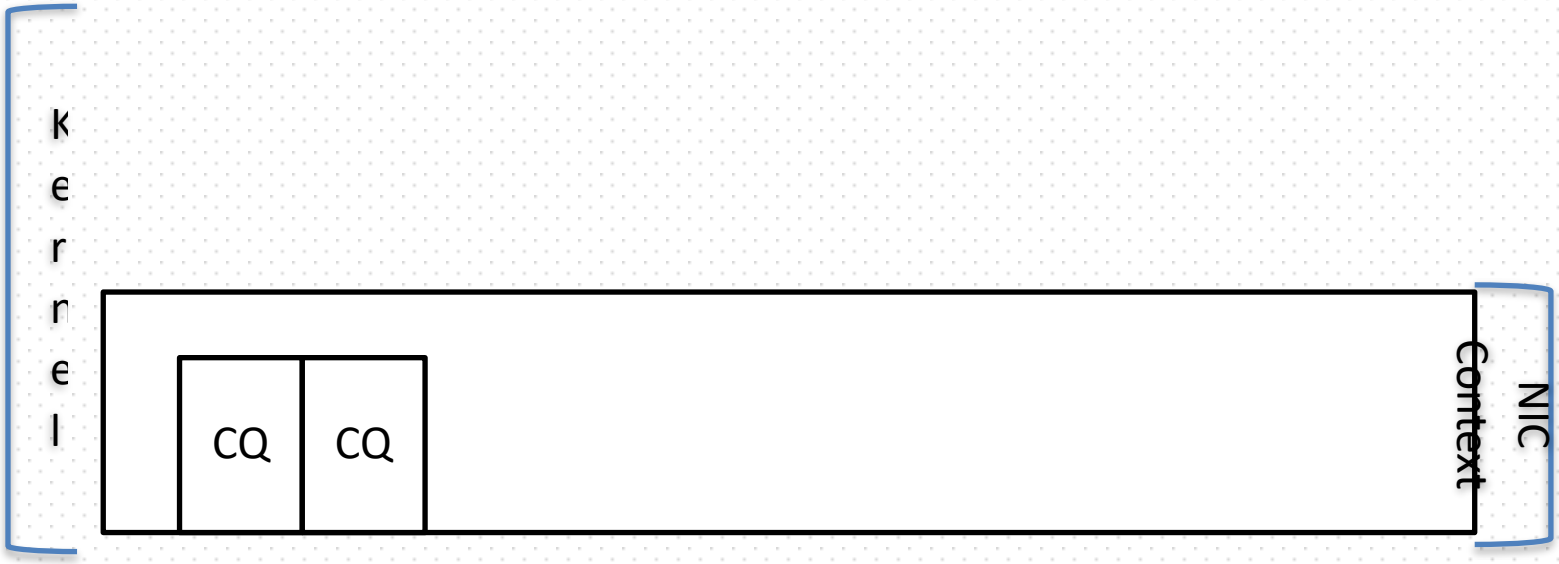
- It is literally a user level networking data plane dev kit. Not a full solution.

SSN Design

- **Architecture**
- API
- Shared state: why?
- Shared state: adventures
- Shared state: internals
- Internals

Architecture [1/4]

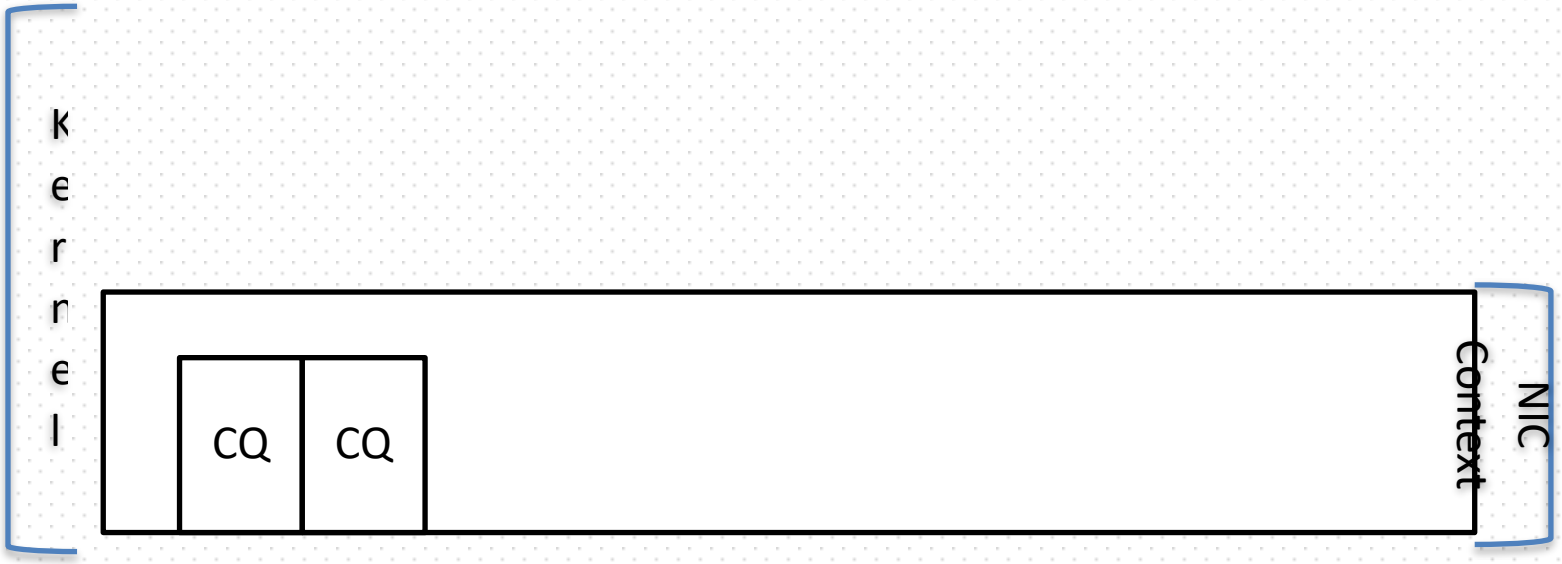
- Combined Queue: minimum set of resources required to send/receive traffic (HW people might call it VQ/ VI)
  - TX
  - RX
- Filtering



Architecture [1/4]

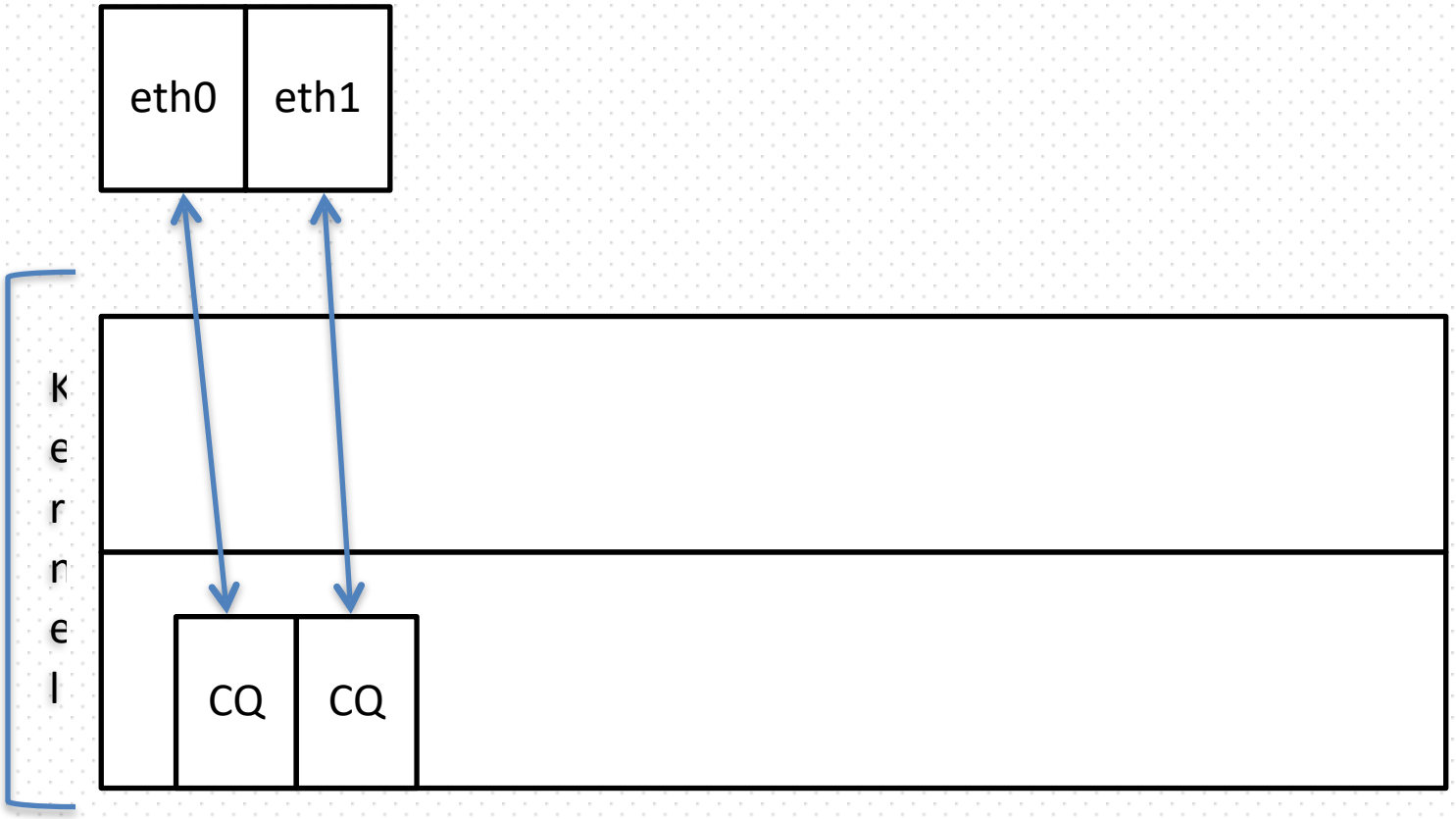
- Combined Queue: minimum set of resources required to send/receive traffic
  - TX
  - RX
- Filtering

- Combined Queue != PCI Function (and VF)
- PCI Function “has” multiple Combined Queues
- VF can be passed through to VM, queue can’t

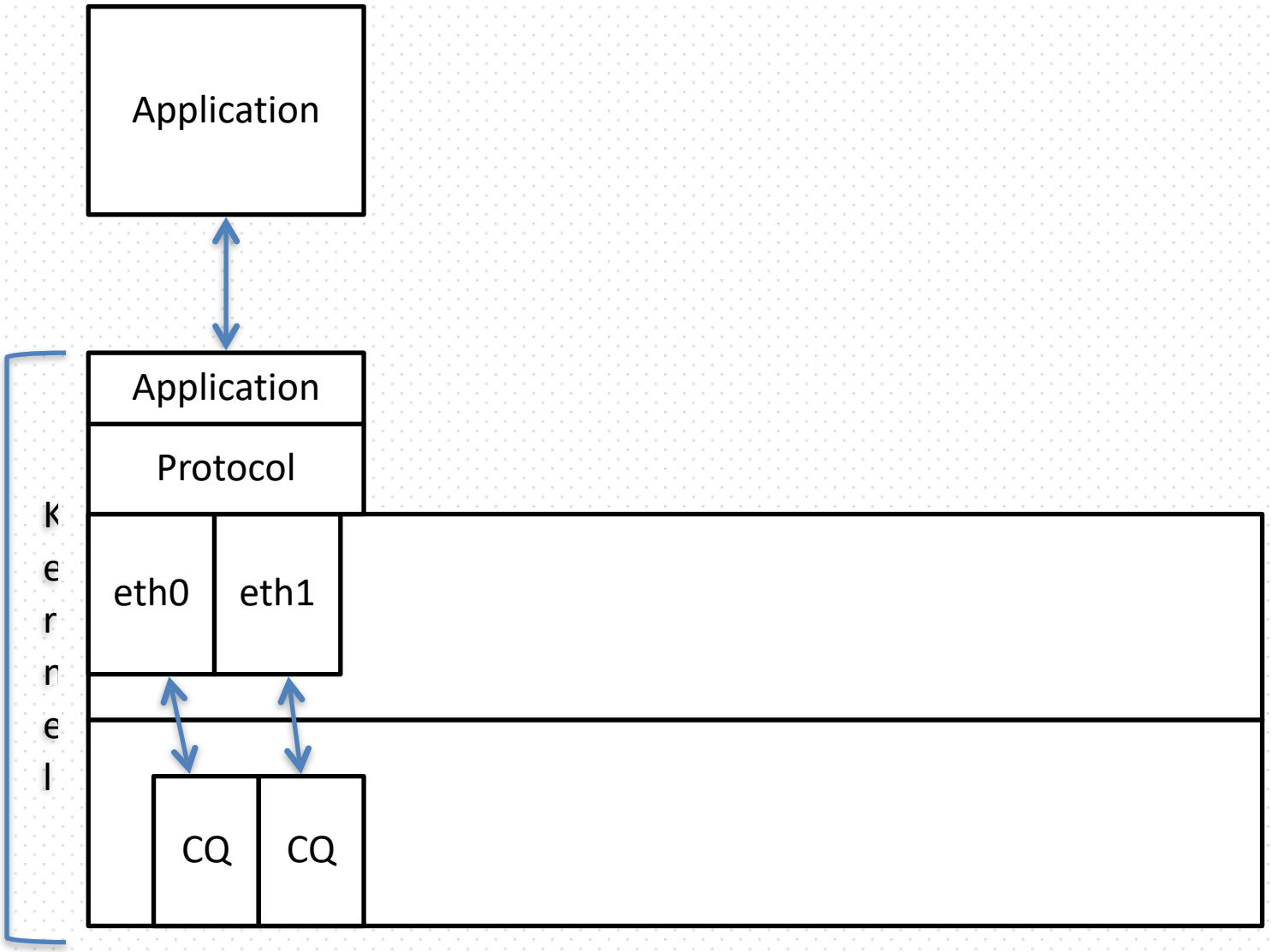


Architecture [2/4]

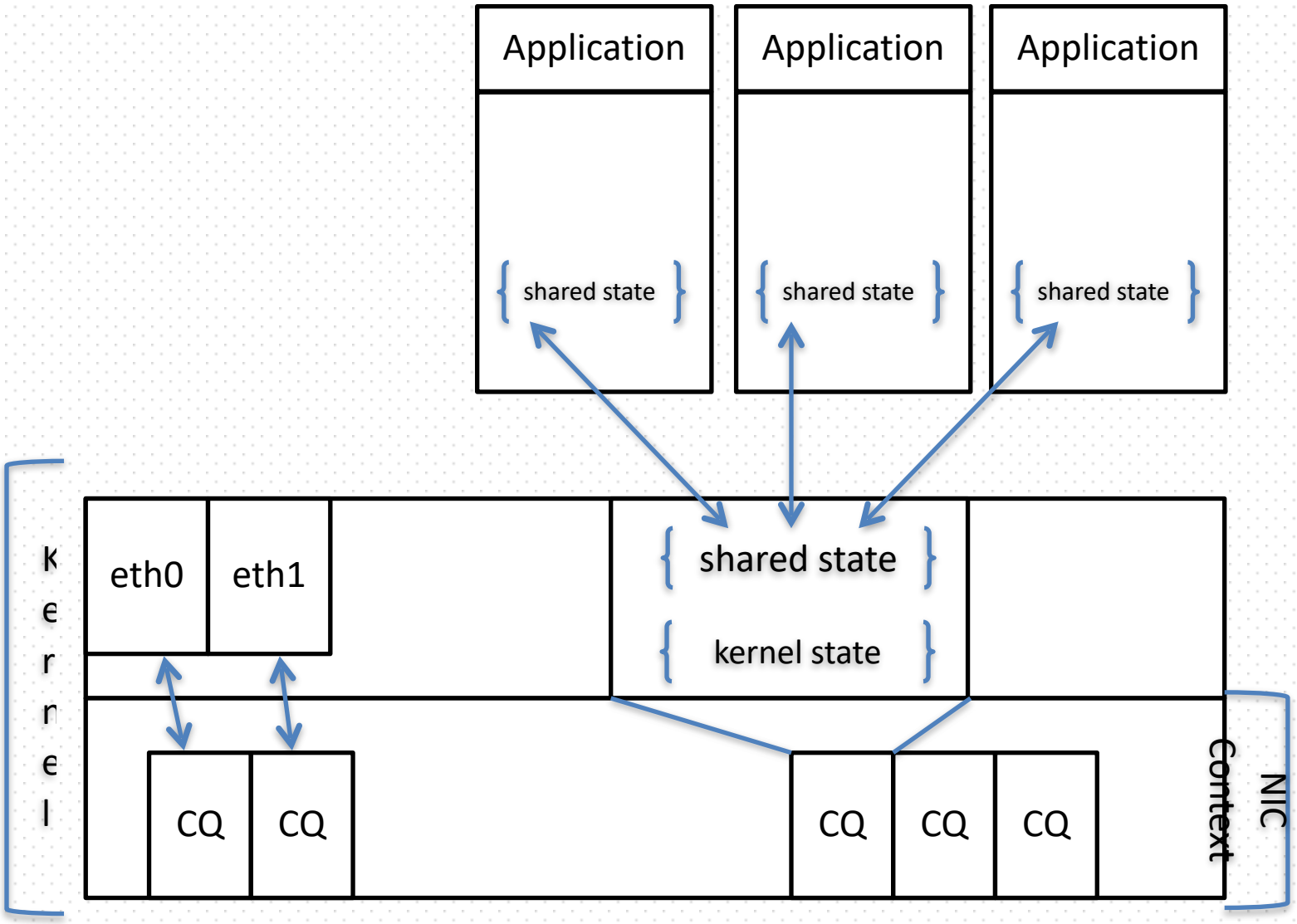
- Traditional interfaces



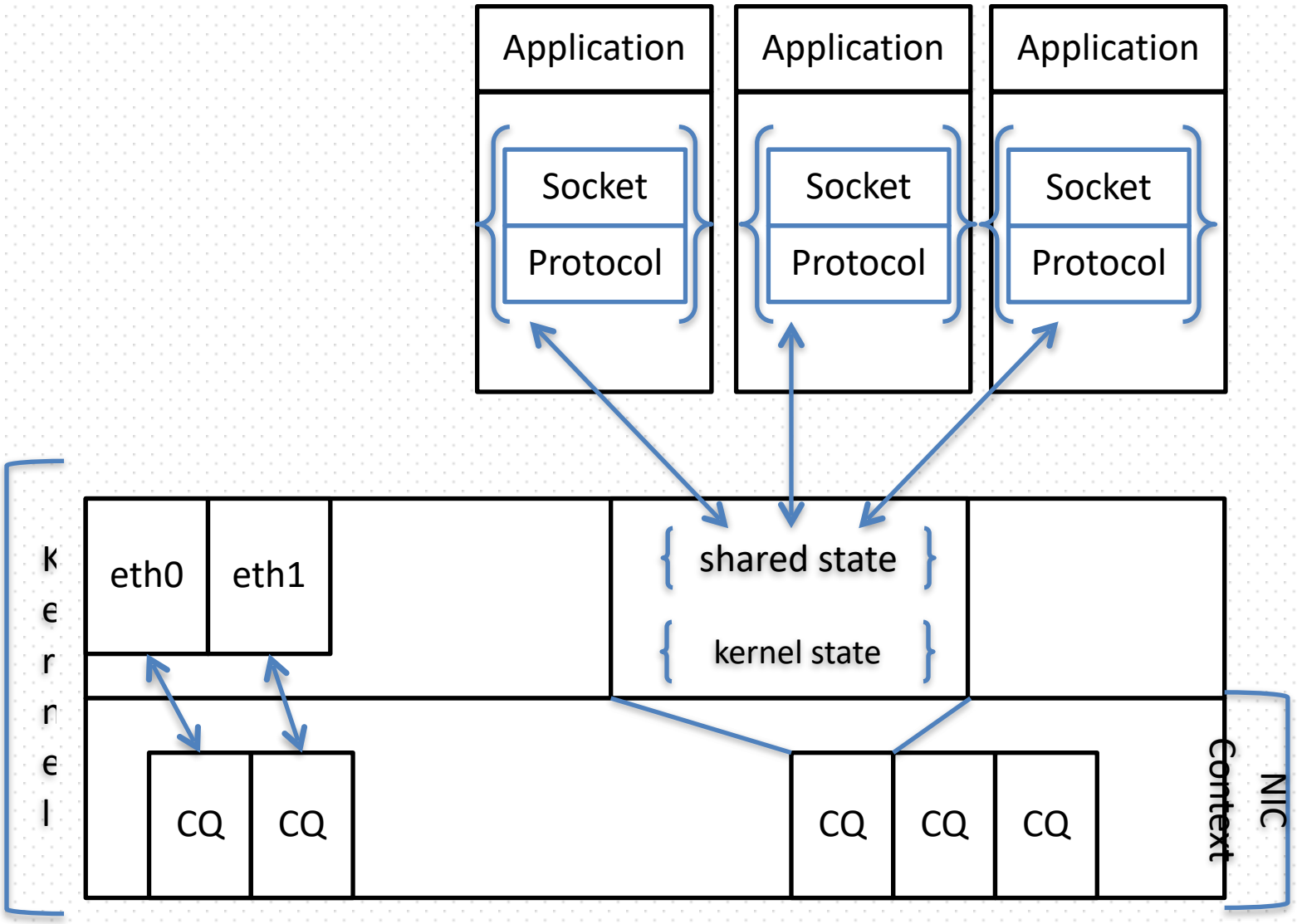
Architecture [2/4]



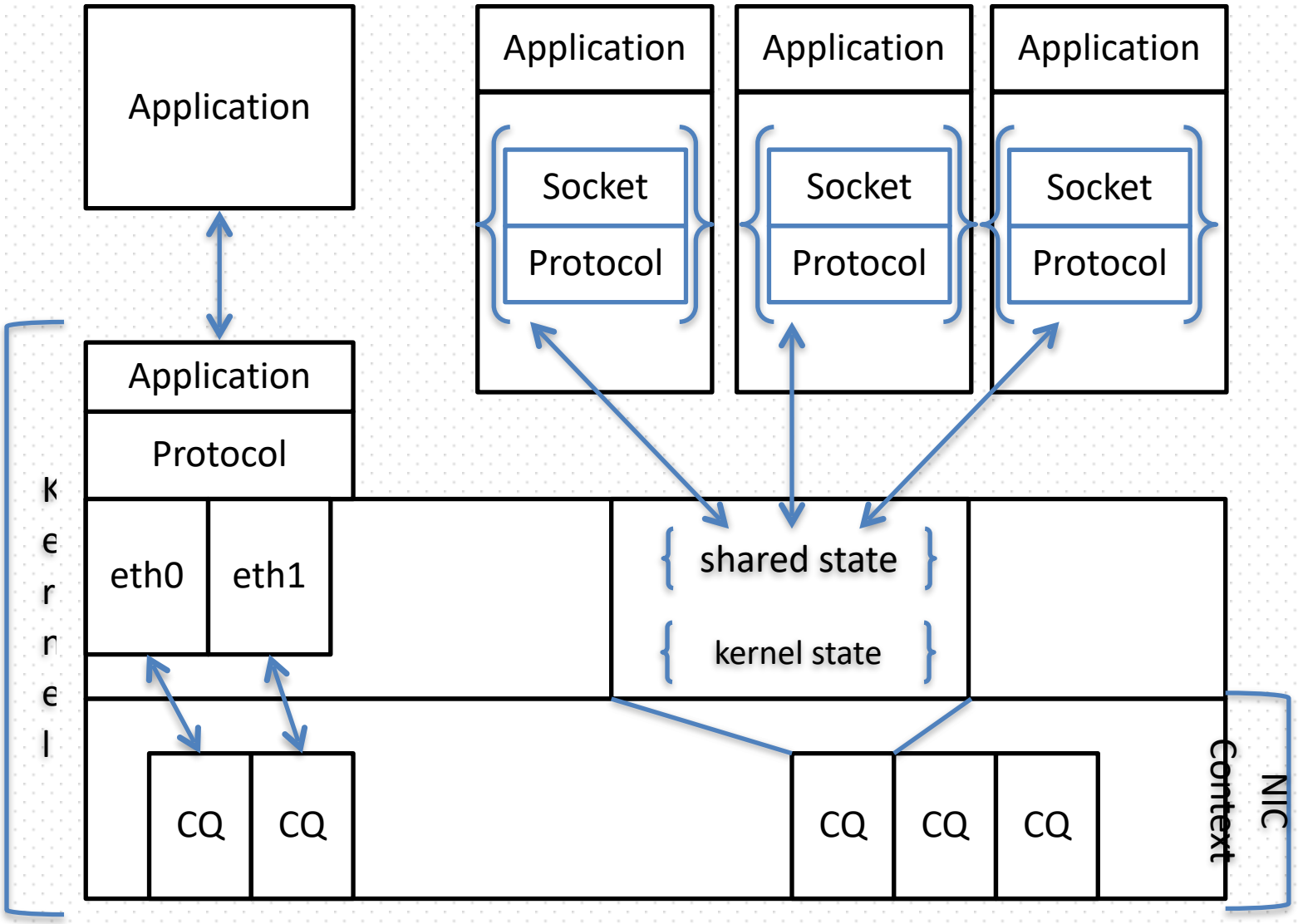
Architecture [3/4]



Architecture [4/4]

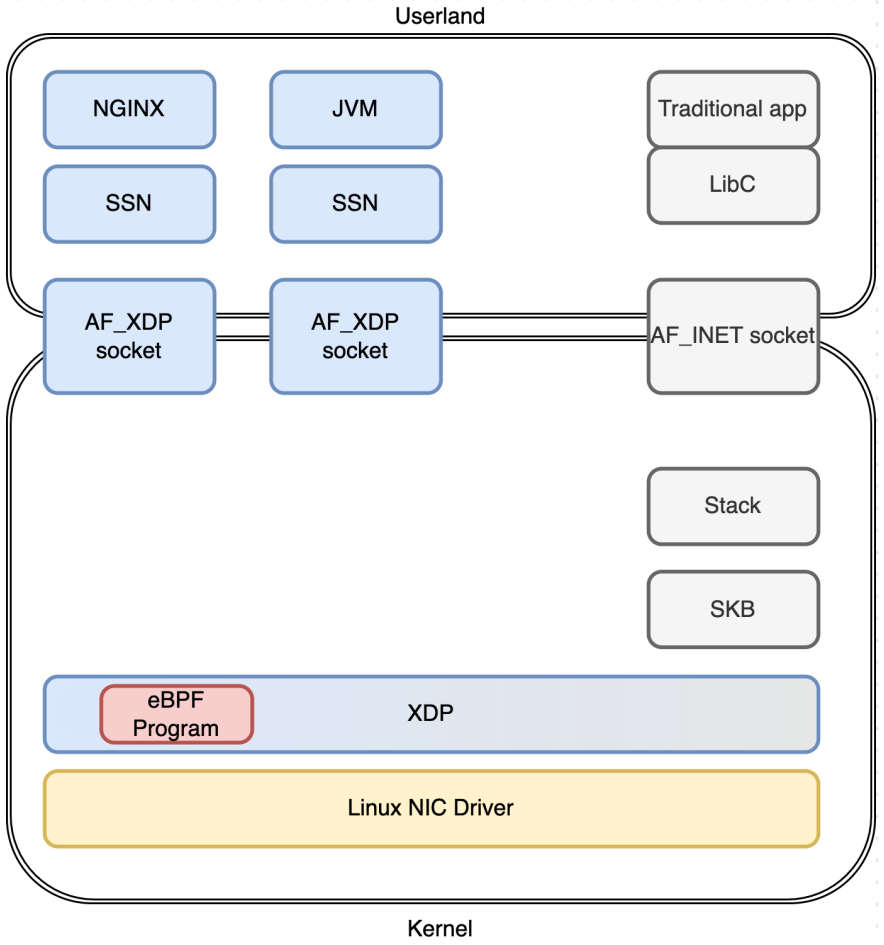


Architecture [4/4]



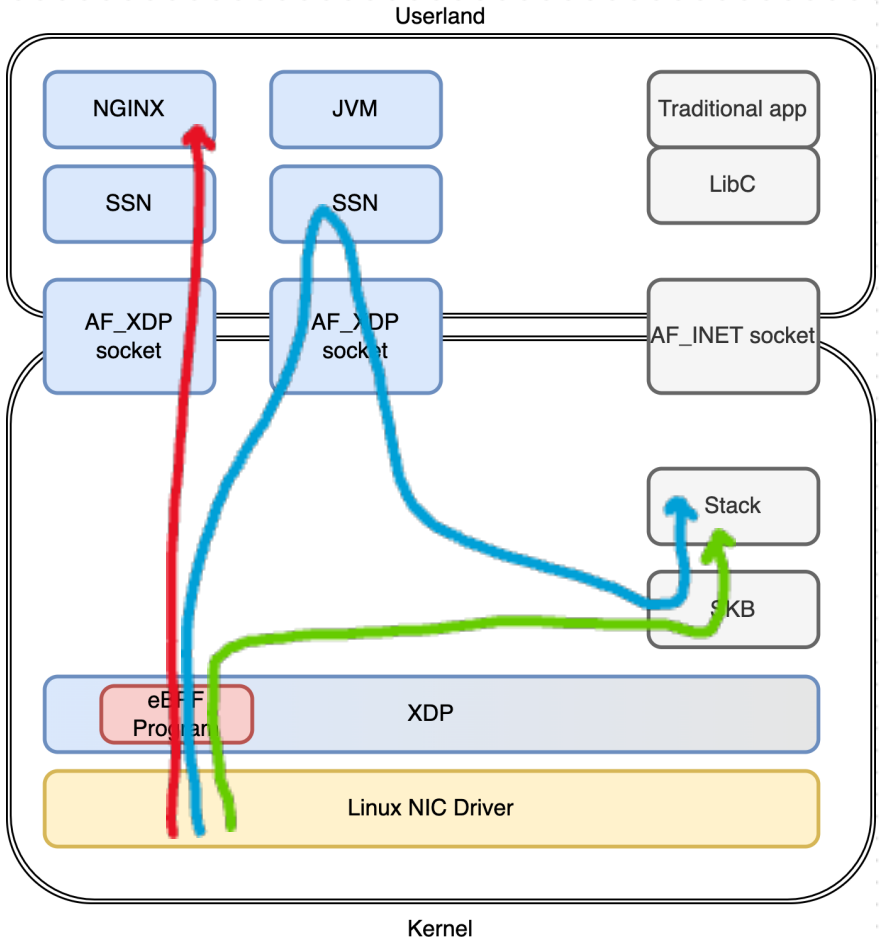
Full stack: AF_XDP

- Multiple applications
- Both traditional and accelerated application can co-exist
- eBPF program for filtering
- Limited context switches on fast path
  - Direct HW has zero, but requires support for each NIC



Full stack: AF_XDP

- Red: fast path!
- Blue: no so fast path
  - If eBPF config is correct (port wise) – packets should not use it.
  - We collect packets that got to us by mistake
  - We push them back where they belong in a “good” moment



SSN Design

- Architecture
- **API**
- Shared state: why?
- Shared state: adventures
- Shared state: internals
- Internals

API: socket API

- socket(), listen(), connect(), accept(), recv(), send(), read(), write(), select(), poll(), epoll_wait(), fcntl(), dup(), accept4(), ioctl(), setsockopt() etc. – full Socket API
  - you don't know how many functions people use and in which ways…
- Access point of the API is **socket file descriptor**
- **LD_PRELOAD** loads the shared library
- In real life you call:
  >$ onload my_application
  and it make sure your application gets correct environment
- SSN provides Linux compatible Socket API

API: recv()/poll()

1) if receive queue not empty -> return data;
2) <here we have don't have data>
3) spin?
   - spin (for some time) in userland waiting for an event
4) go to the kernel ← slow
5) wait for interrupt and "wake up the socket"
6) wake up in userland -> return data

→: copy to user buffer; Zero Copy API gets rid of it, but! it requires one to update the application

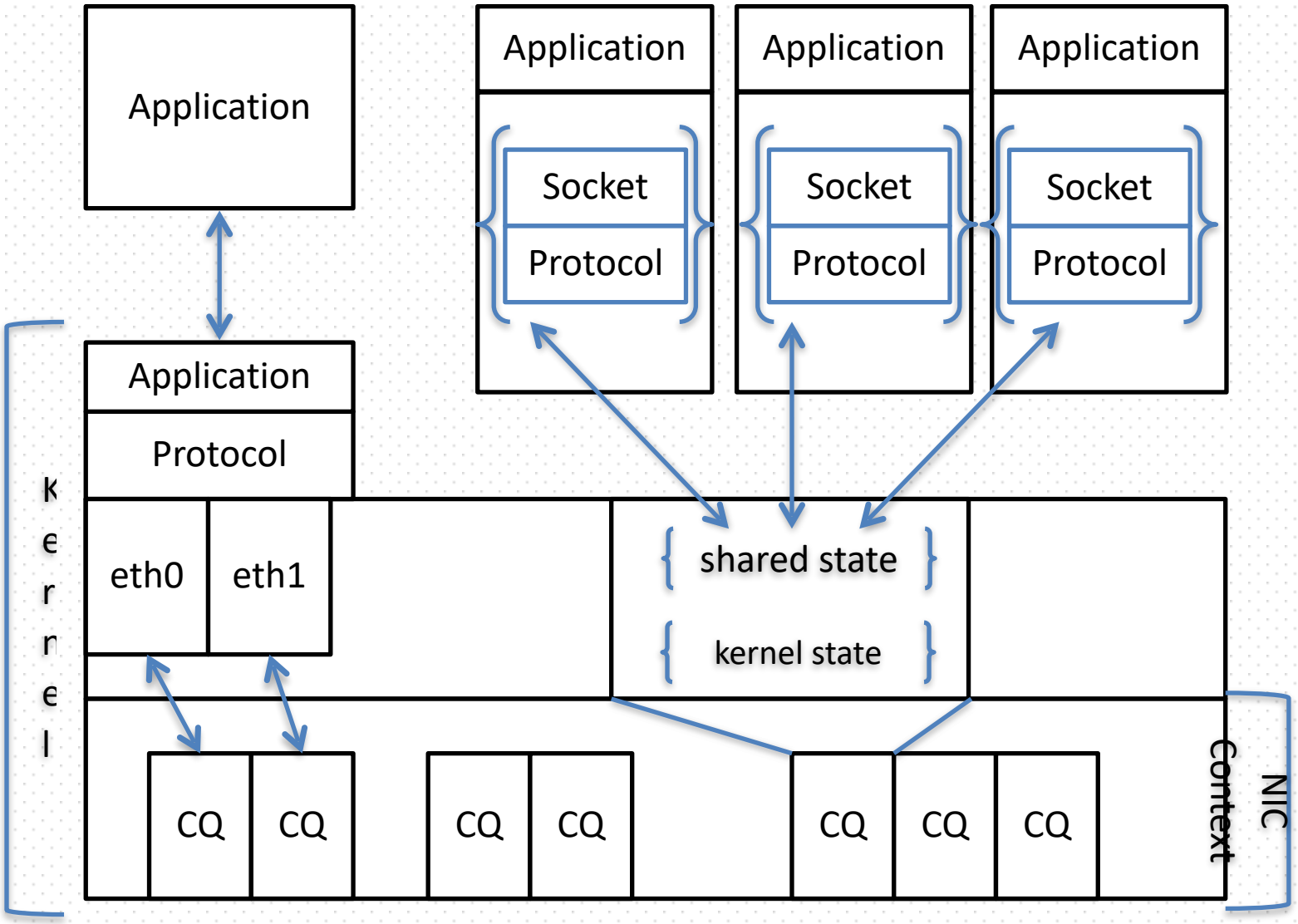
API: TCP send()

- copy user data -> packet buffer
- packet buffer is added into socket sendq
  - sendq is in shared state
- send window & congestion window OK
  - can send => put into the CQ (NIC)
    - This step has another memory copy if NIC does not support AF_XDP ZC
  - poke the NIC (via sendmsg()) to transmit a batch of packets
    - Batch is configurable
    - **Switches to kernel in case of AF_XDP**
- otherwise send provoked by event handler (in userland OR kernel)

SSN Design

- Architecture
- API
- **Shared state: why?**
- Shared state: adventures
- Shared state: internals
- Internals

Shared state: why

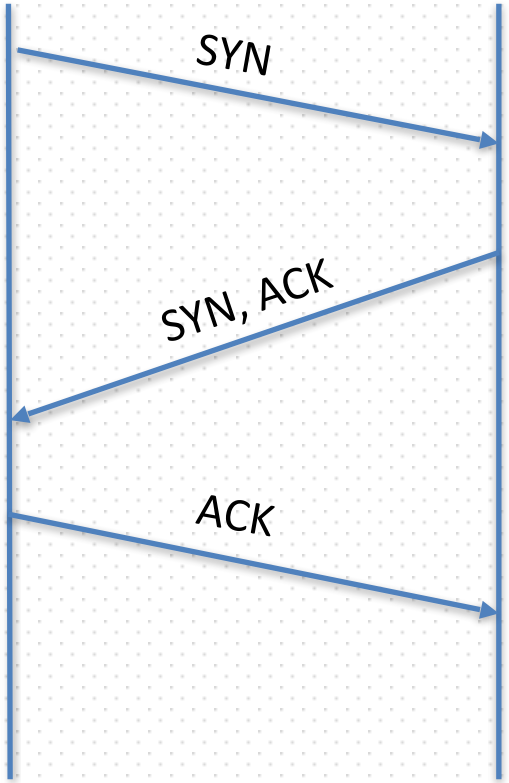


Why do we need Shared State [1/2]

- fork() : duplicates everything, need to be in sync
- exec() : just wipes everything out
- Process can send fd/socket via UNIX domain socket
- Process exits (perhaps dies?): data should be delivered + socket should be shut down
  - Graceful shutdown

Why do we need Shared State [2/2]

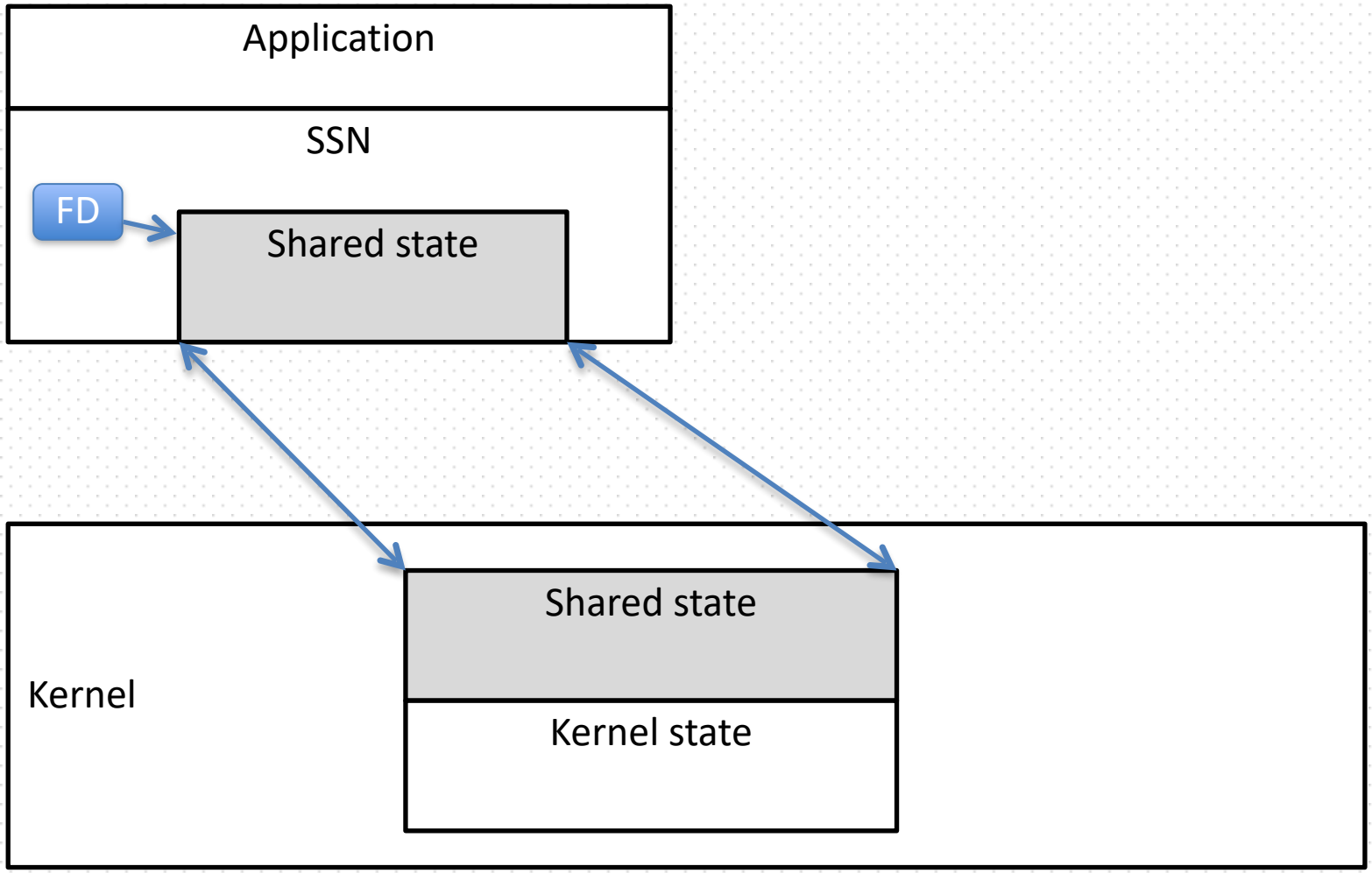
- TCP Machine should have a context
  - Say your application is hanging on poll() or accept()
  - Example: who sends {SYN,ACK} if you haven't yet called accept()
- Who handles timer events?
- Who handles NIC events?
- Can I handle it in a separate thread in userland: possible.



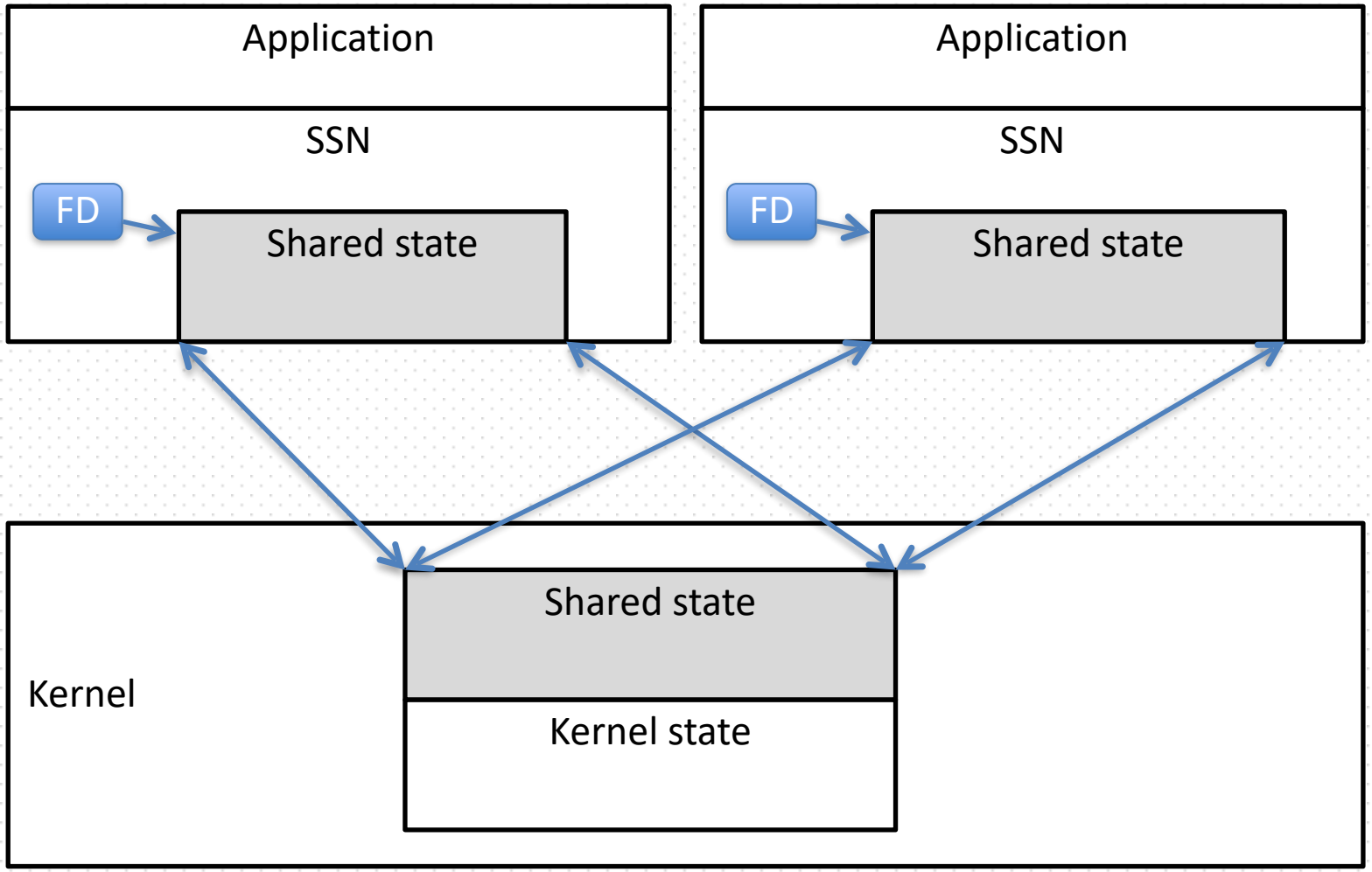
SSN Design

- Architecture
- API
- Shared state: why?
- **Shared state: adventures**
- Shared state: internals
- Internals

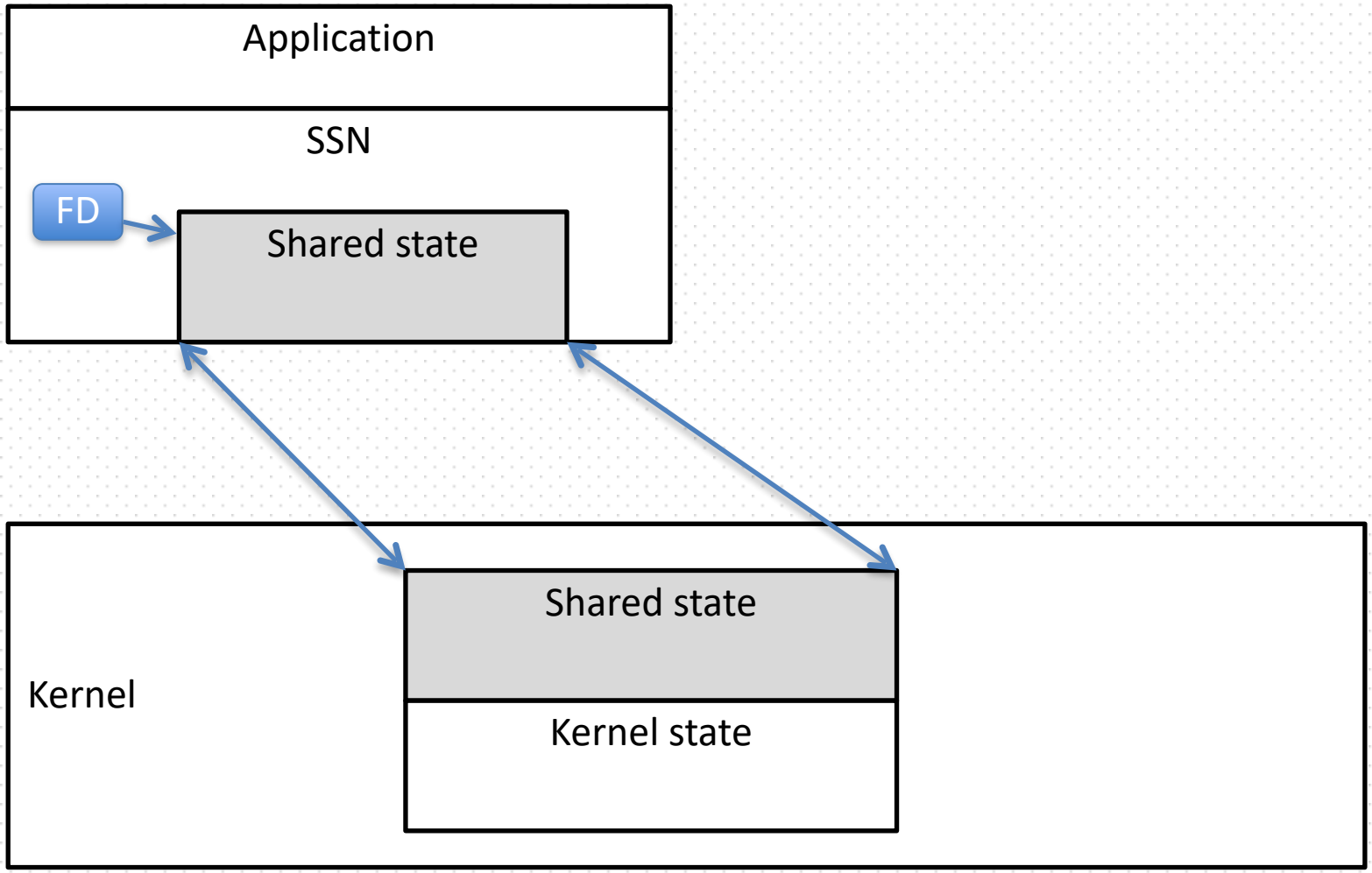
Shared state: fork() [1/2]



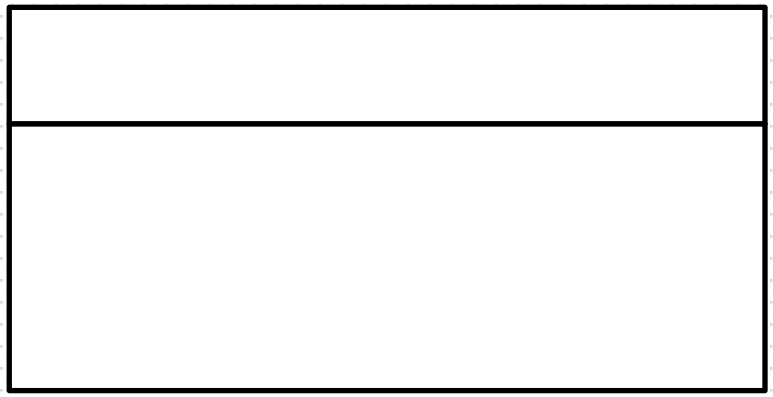
Shared state: fork() [2/2]

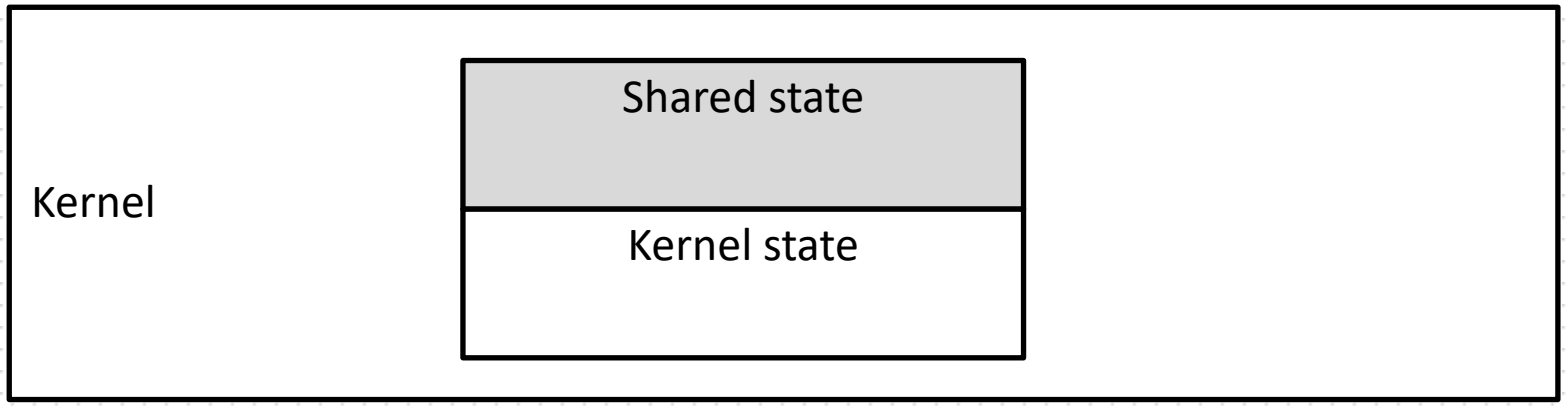


Shared state: exec() [1/4]

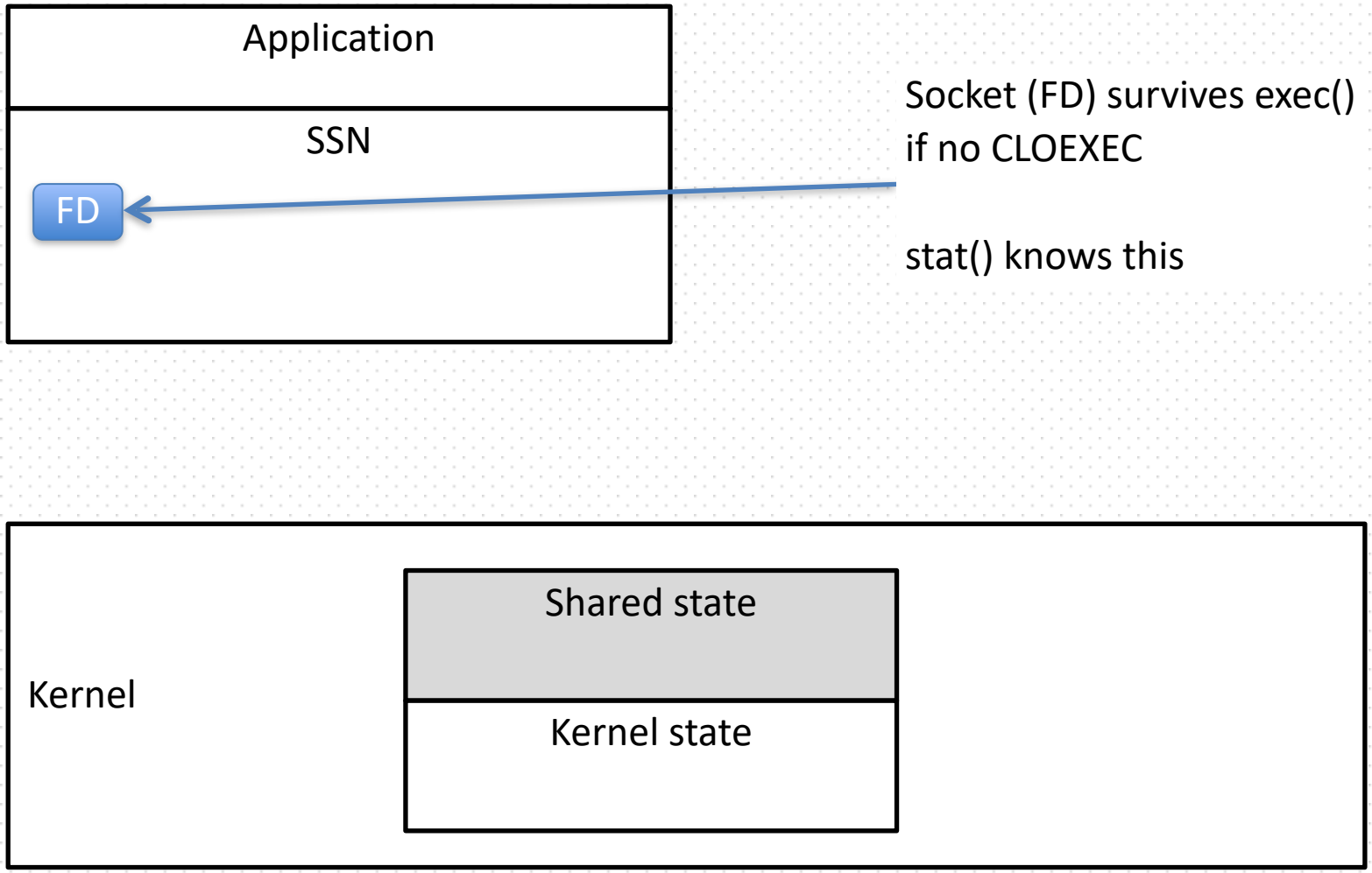


Shared state: exec() – wipes it all [2/4]

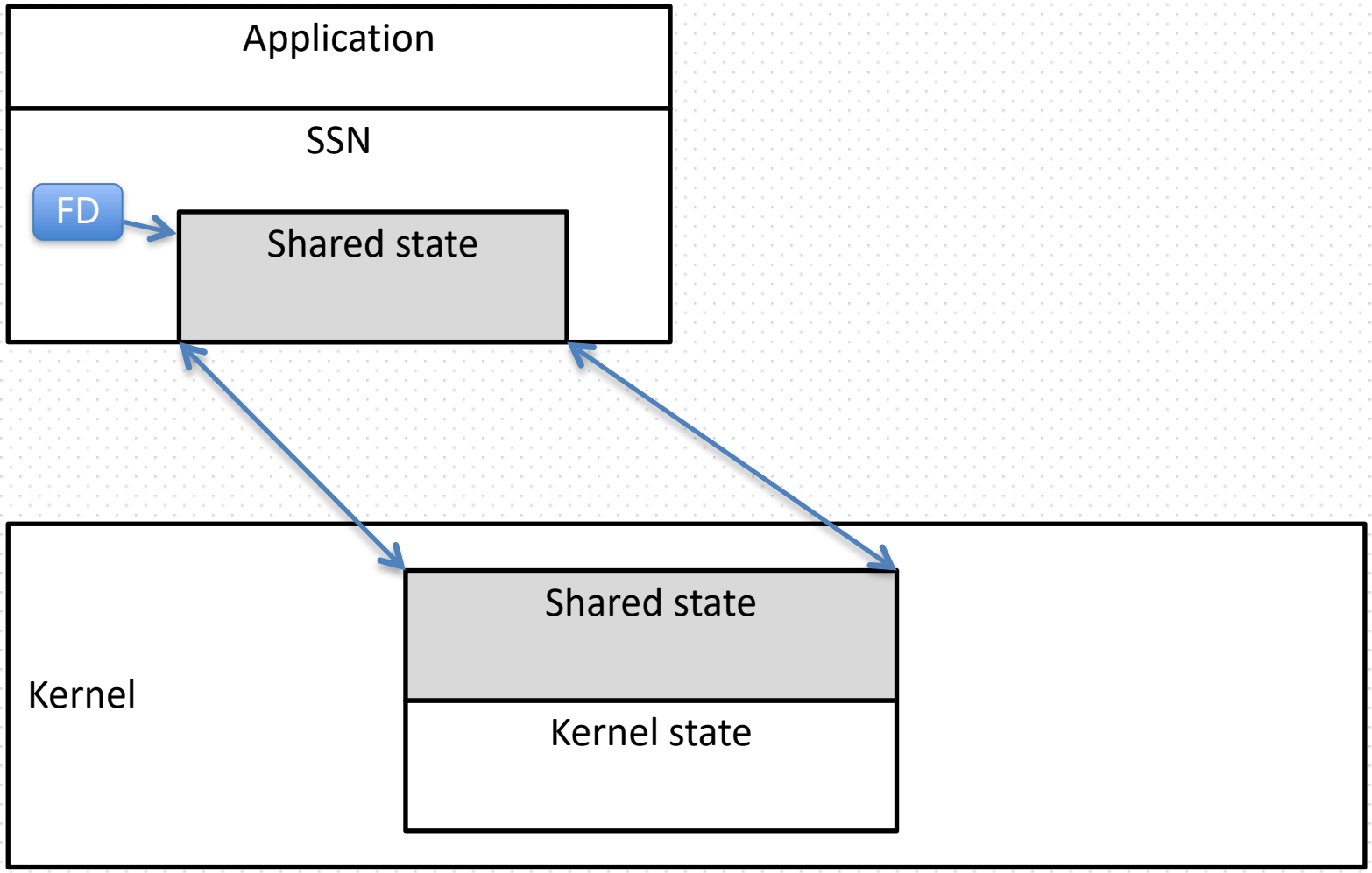




Shared state: exec() [3/4]



Shared state: exec() [4/4]



SSN Design

- Architecture
- API
- Shared state: why?
- Shared state: adventures
- **Shared state: internals**
- Internals

Shared state: internals

- What is in it:
  - sockets,
  - packet buffers,
  - CQ/VI state,
  - timers (retransmit, keepalive etc.),
  - free resources,
  - configuration,
  - demux table (selects socket using SW filters).

SSN design

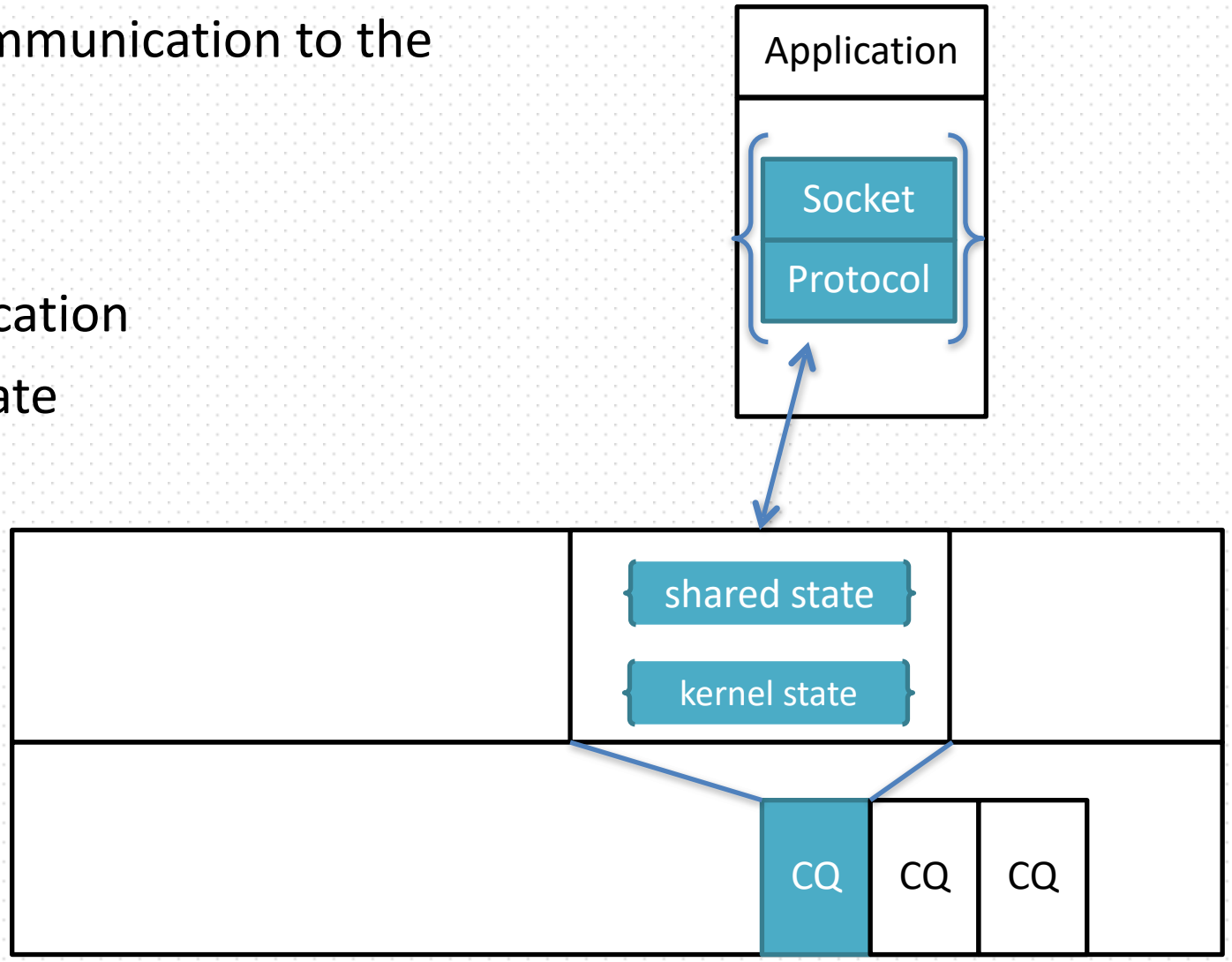
- Architecture
- API
- Shared state: why?
- Shared state: adventures
- Shared state: internals
- **Internals**

Internals

- **Stack <> application**
- Onload FD
- OS Interaction:
  - Accelerated vs non-accelerate interfaces
  - Control plane
  - Process termination and graceful shutdown

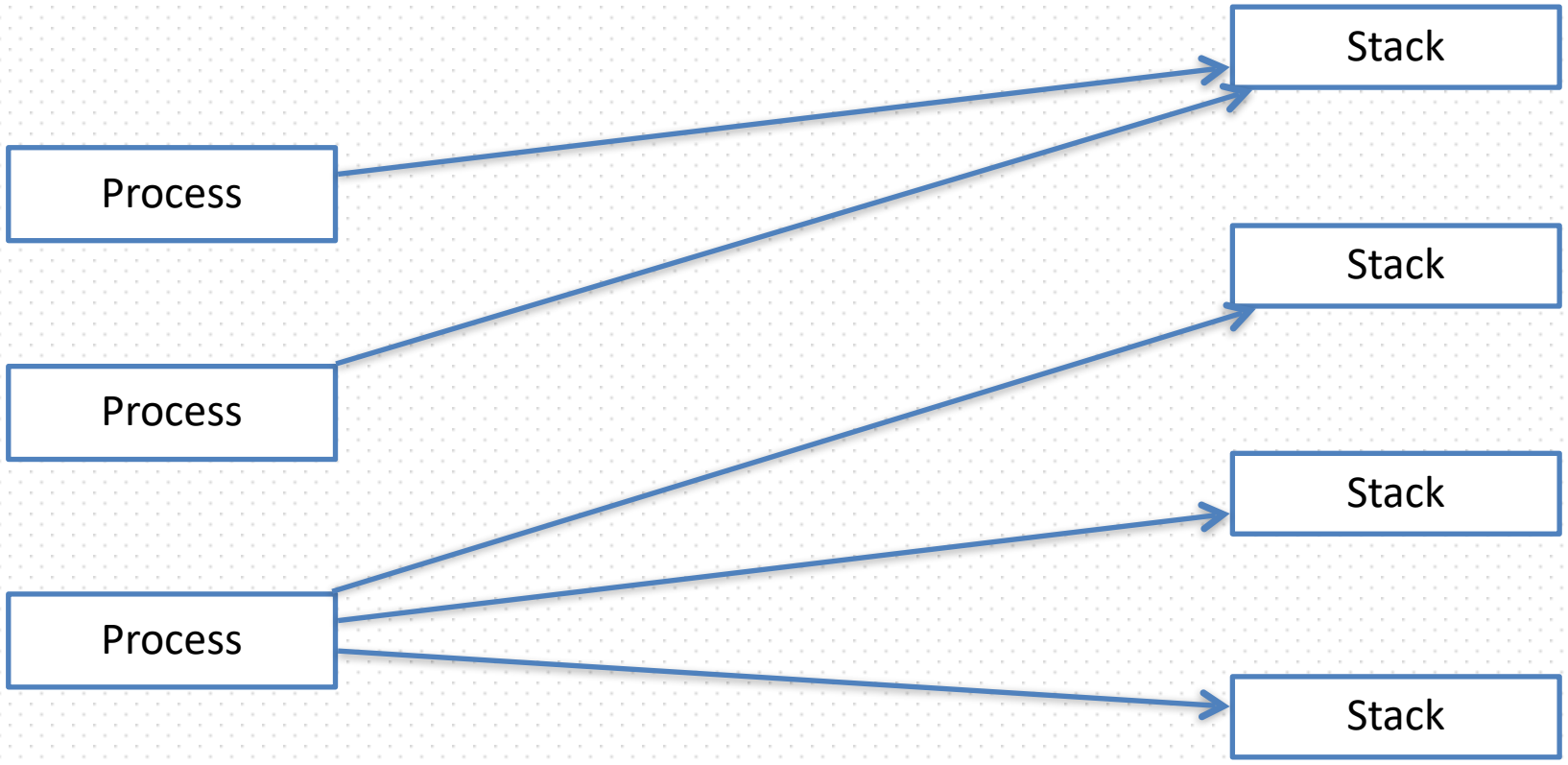
Stack: basics

- Stack: entity that allows socket communication to the NIC/CQ
  - application only entrance point is
    socket = socket()
- Lifetime: independent of the application
- Tightly connected to the shared state

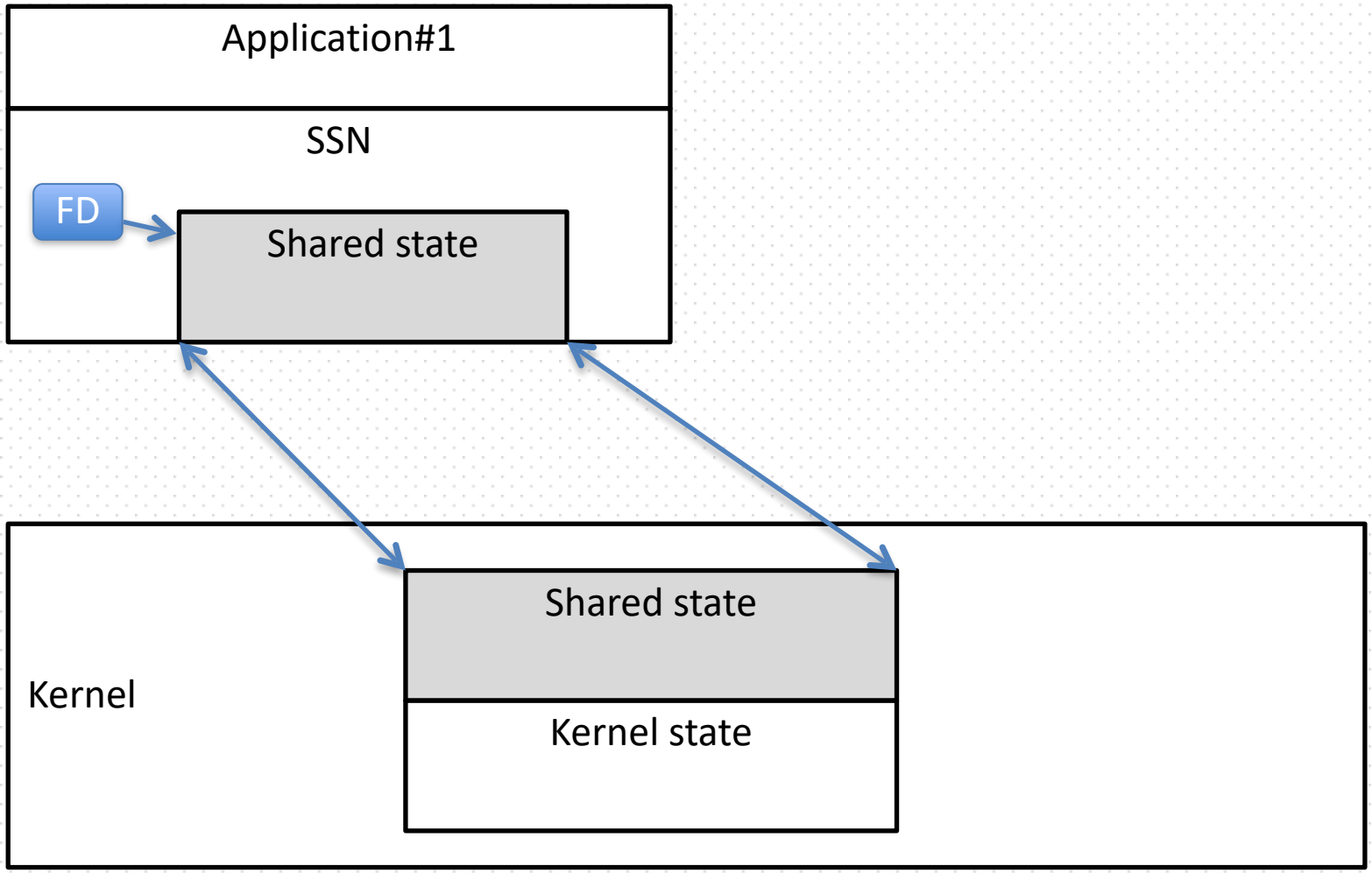


Stack: <-> processes

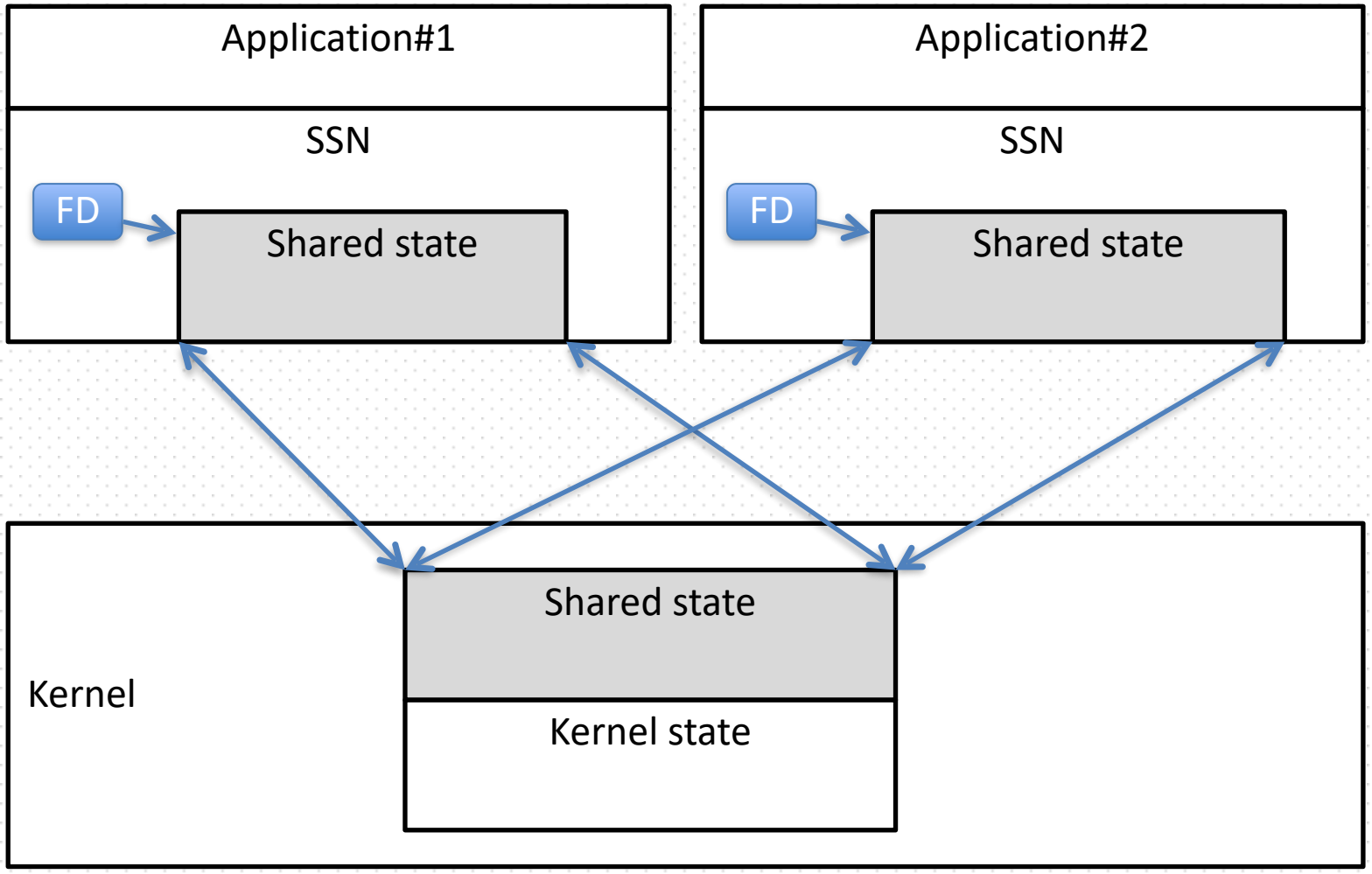
- Arbitrary mapping
- Can change over time
- Highly configurable



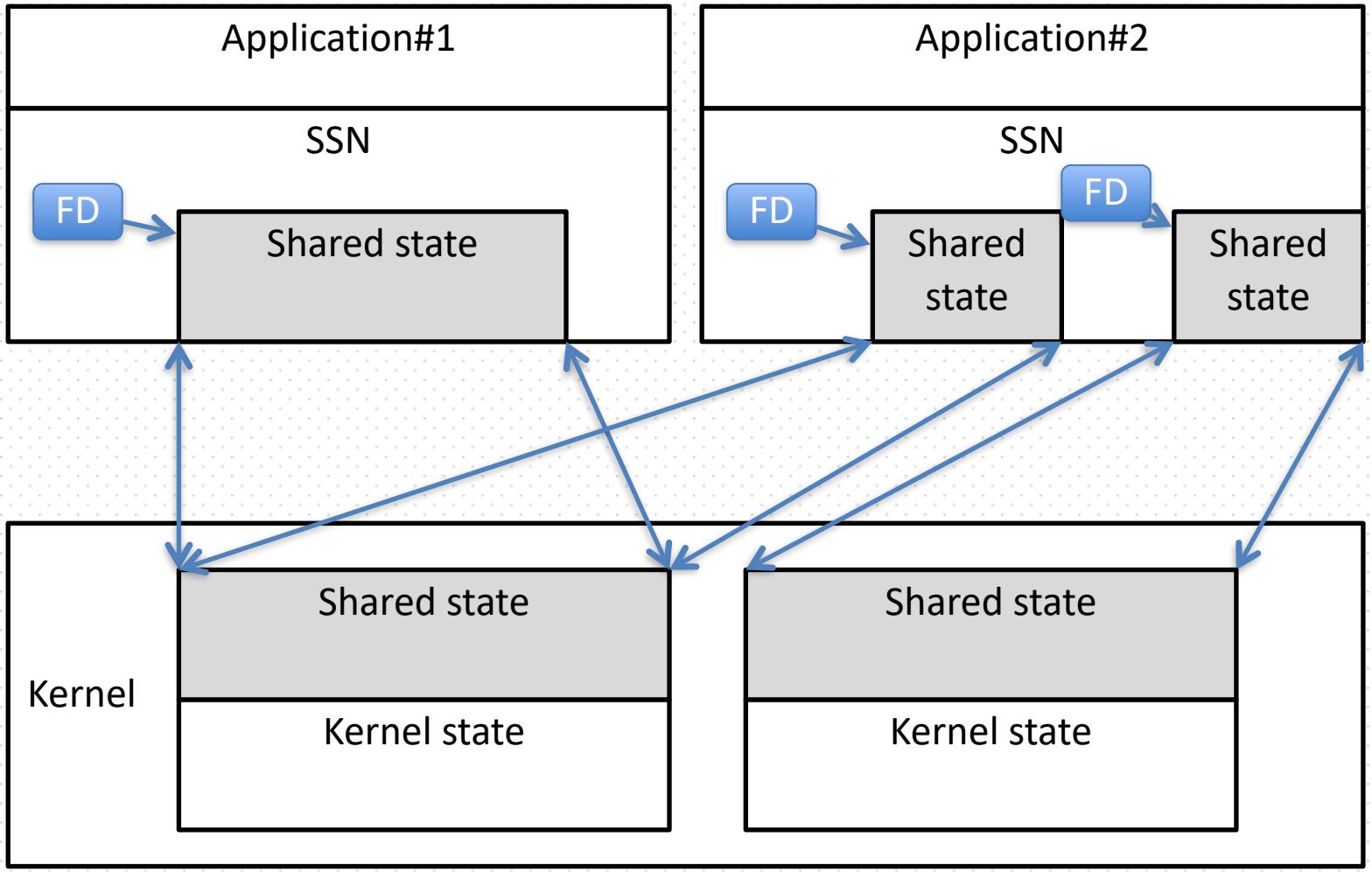
Default stack: socket()



Default stack: fork()



Default stack: socket() in Application#2: default behaviour



Internals

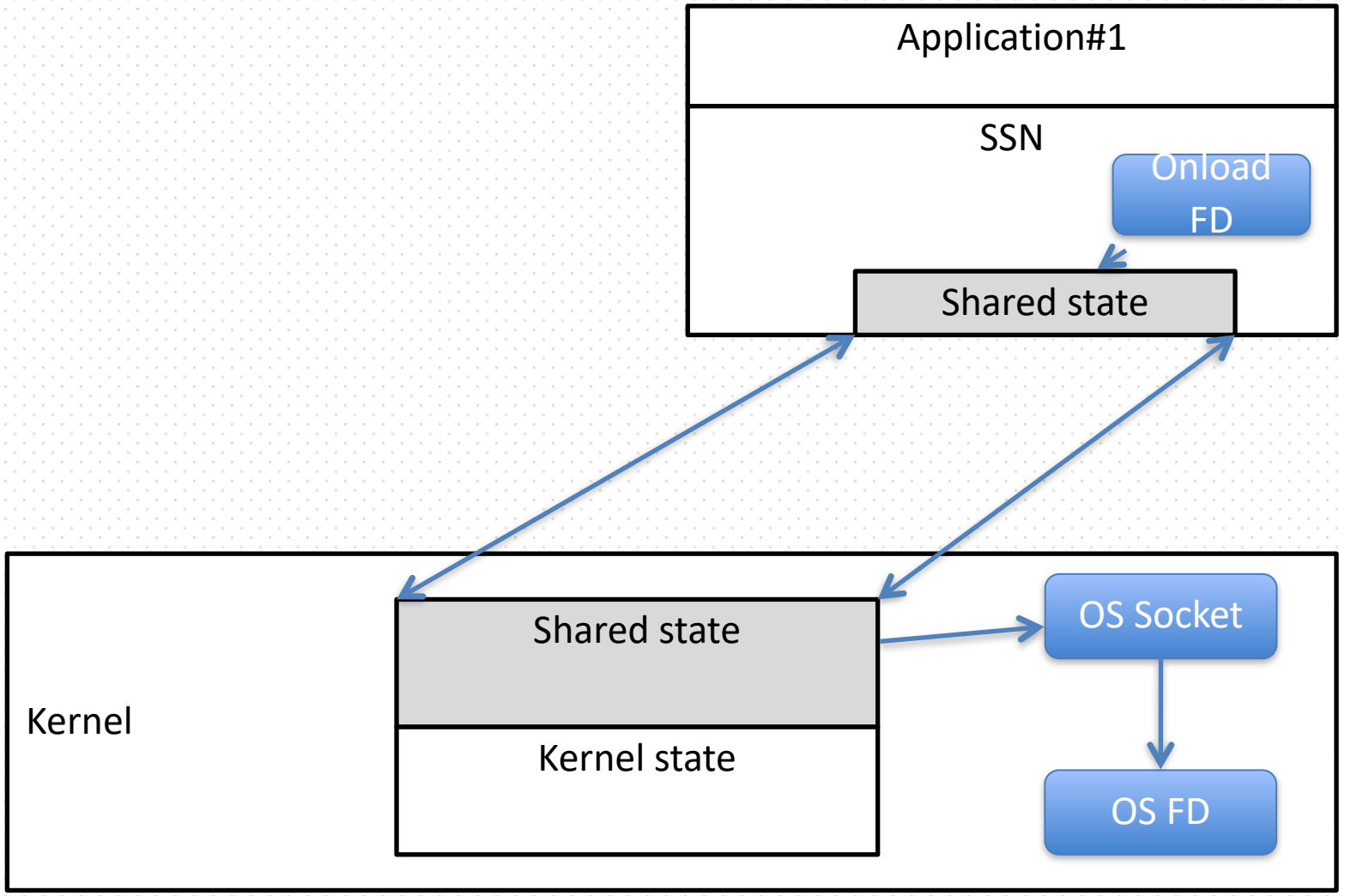
- Stack <> application
- **Onload FD**
- OS Interaction:
  - Accelerated vs non-accelerate interfaces
  - Control plane
  - Process termination and graceful shutdown

Onload FD: onloadfs

- Socket is an FD
- Onload socket is also an FD
- /proc/pid/fd/239 : special onloadfs
  - similar to socketfs
- It's an FD, so even **without SSN:**
  - read()
  - write()
  - poll(), epoll_wait(), select()

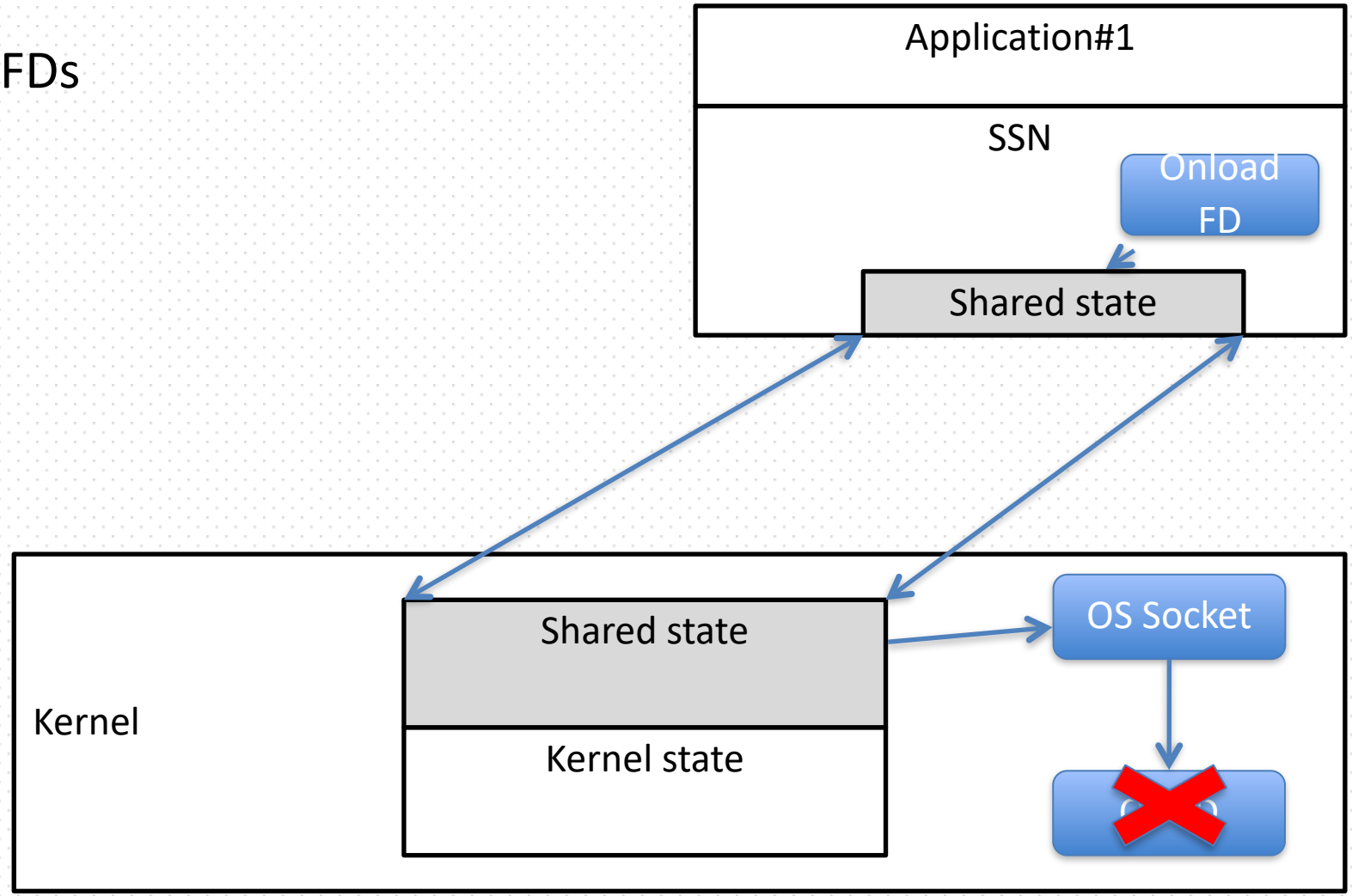
Onload FD: OS socket

- OS Socket: reserve ports



Onload FD: OS socket

- OS Socket: reserve ports
- No OS FD: don’t waste 2 FDs per socket



Internals

- Stack <> application
- Onload FD
- **OS Interaction:**
  - Accelerated vs non-accelerate interfaces
  - Control plane
  - Process termination and graceful shutdown

OS Interaction: accelerated interfaces

- You need to register interface to get acceleration:
  - 1: socket(, SOCK_DGRAM,) -> s1
  - 2: sendto(s1,) → accelerated interface
  - 3: sendto(s1,) → non-accelerated interface

  - SSN detects that you're working with non-accelerated interface and passes packet in (3) to the kernel.

OS Interaction: Control Plane

- ARP
- route (no multi-table setup)
- Interface addresses
- ip rule (limited source-based routing)
- iptables:
  - limited support,
- Bonding and teaming (active-backup, load-balance)
- Control plane structures are RO for userland

OS Interaction: signals and process termination

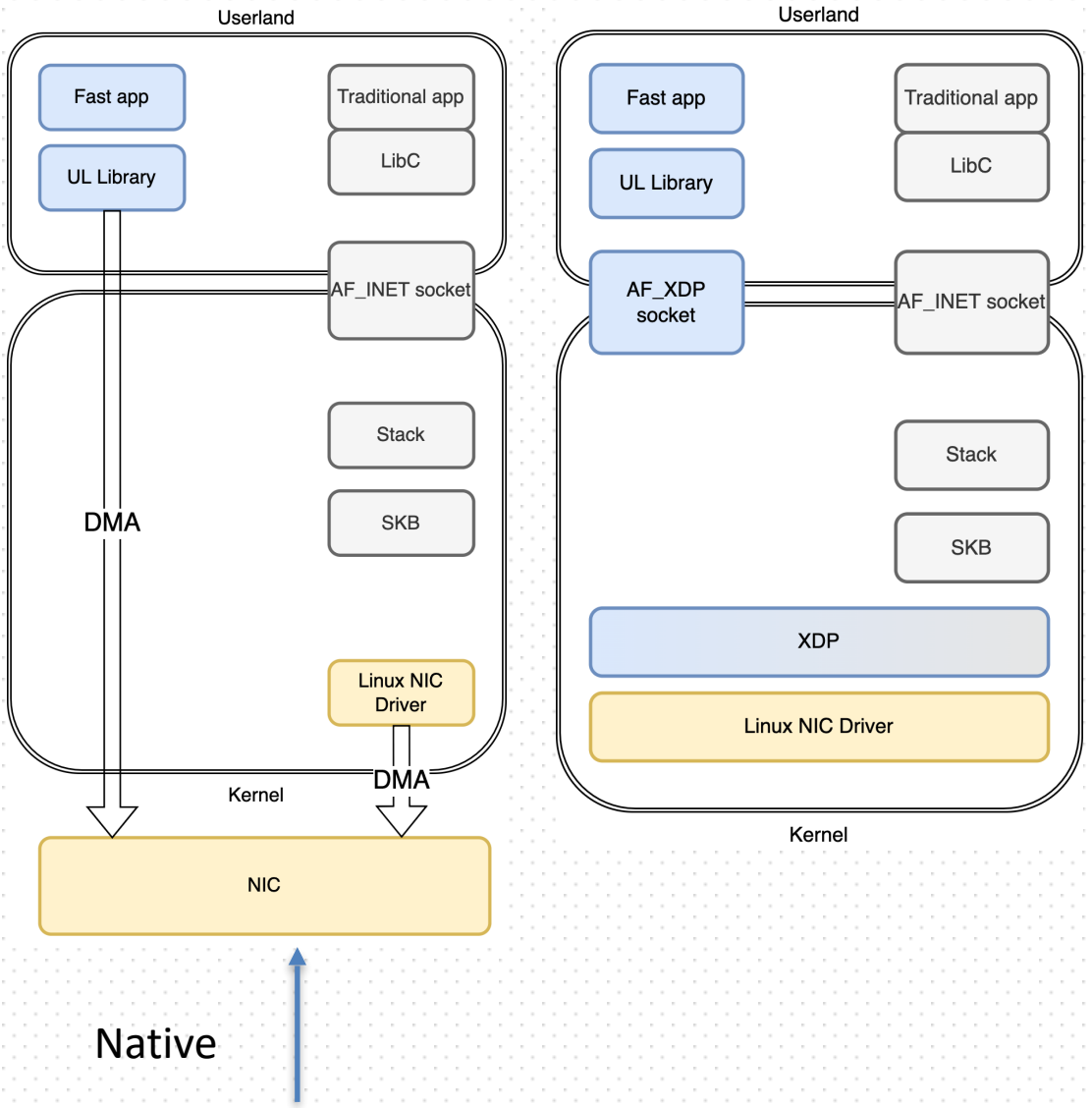
- Process live and processes die, SSN handles this
- Cause of death:
  - exit()
  - Signals (not only termination)
- Problems:
  - Resources cleanup?
  - Deadlocks
  - Graceful TCP shutdown
  - pselect()/ppoll()/epoll_pwait()

Performance time

- Why and what has changed?
- Networking
  - Kernel bypass networking
  - Speed-Stack Net
  - **Performance**
- Block
  - Speed-Stack Block
  - Performance
- Filesystems: stay tuned

Acceleration: some examples

- Native over AMD/Solarflare NIC
  - NGINX reverse proxy
  - Memcached
- AF_XDP
  - NGINX server on SFC
  - NGINX server on MLX
  - NGINX server on Intel

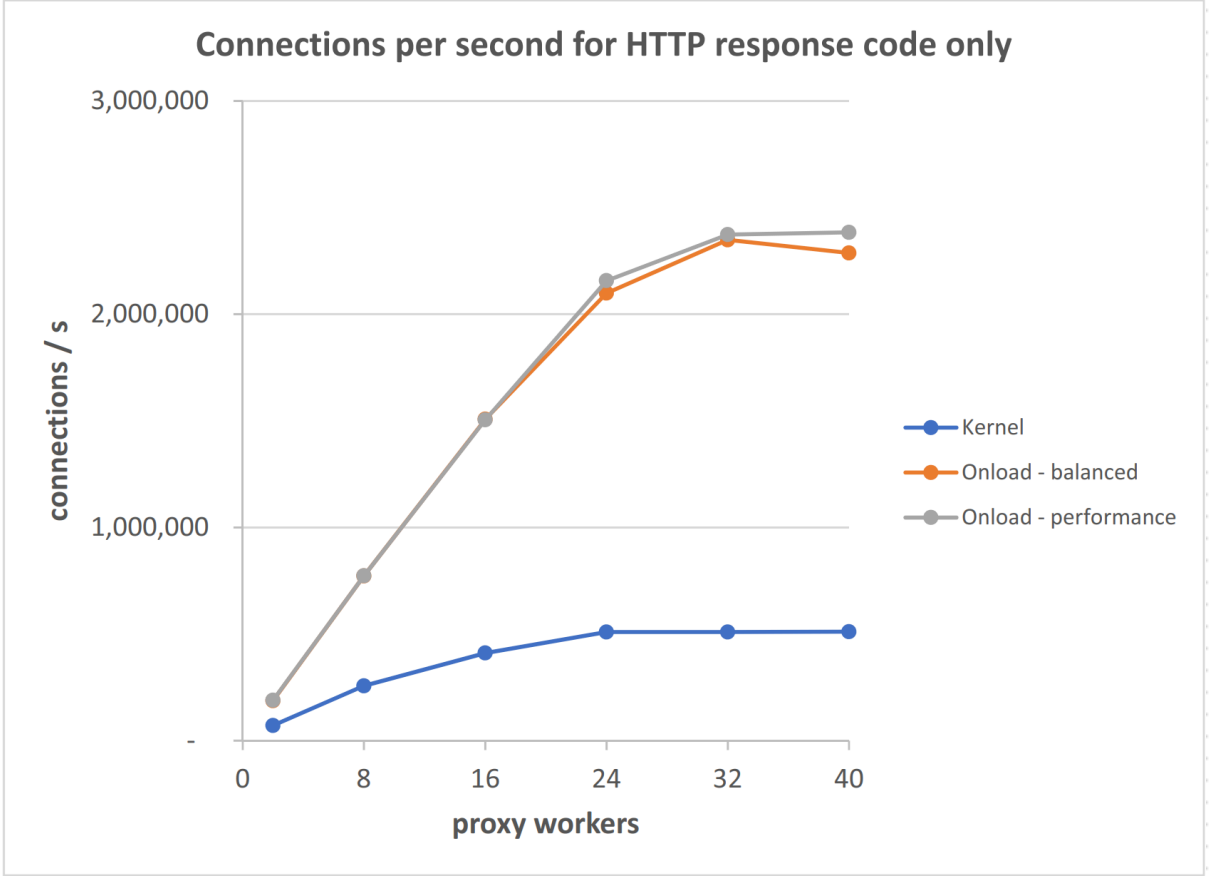


Nginx proxy: testbench for SFC

- SFC NIC X2541 (100G)
- NGINX proxy
- 2 × Intel® Xeon® Gold 6148 CPU @ 2.40GHz
- Load is created with wrk2 (many instances of it!)

Nginx proxy: connection rate

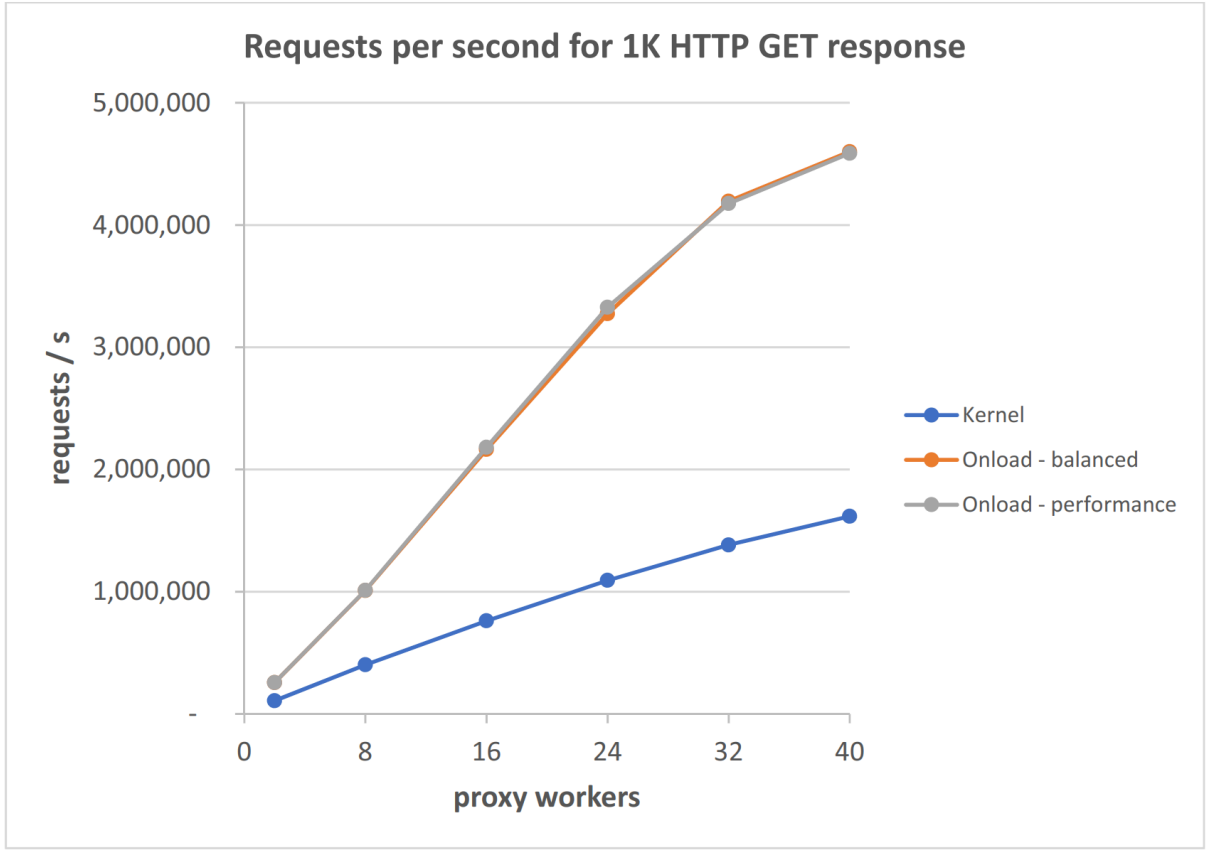
- Why:
  - Socket caching: reuse sockets that were just closed
  - Reduced cost of networking calls
  - epoll_wait() scaling improved: O(1)
    - And it does not go into the kernel
- Caching:
  - listen()
  - accept() → s1
  - accept() → s2
  - close(s1): we don't really close it!
  - → SYN received:
    - take s1
    - no need to go to the kernel!



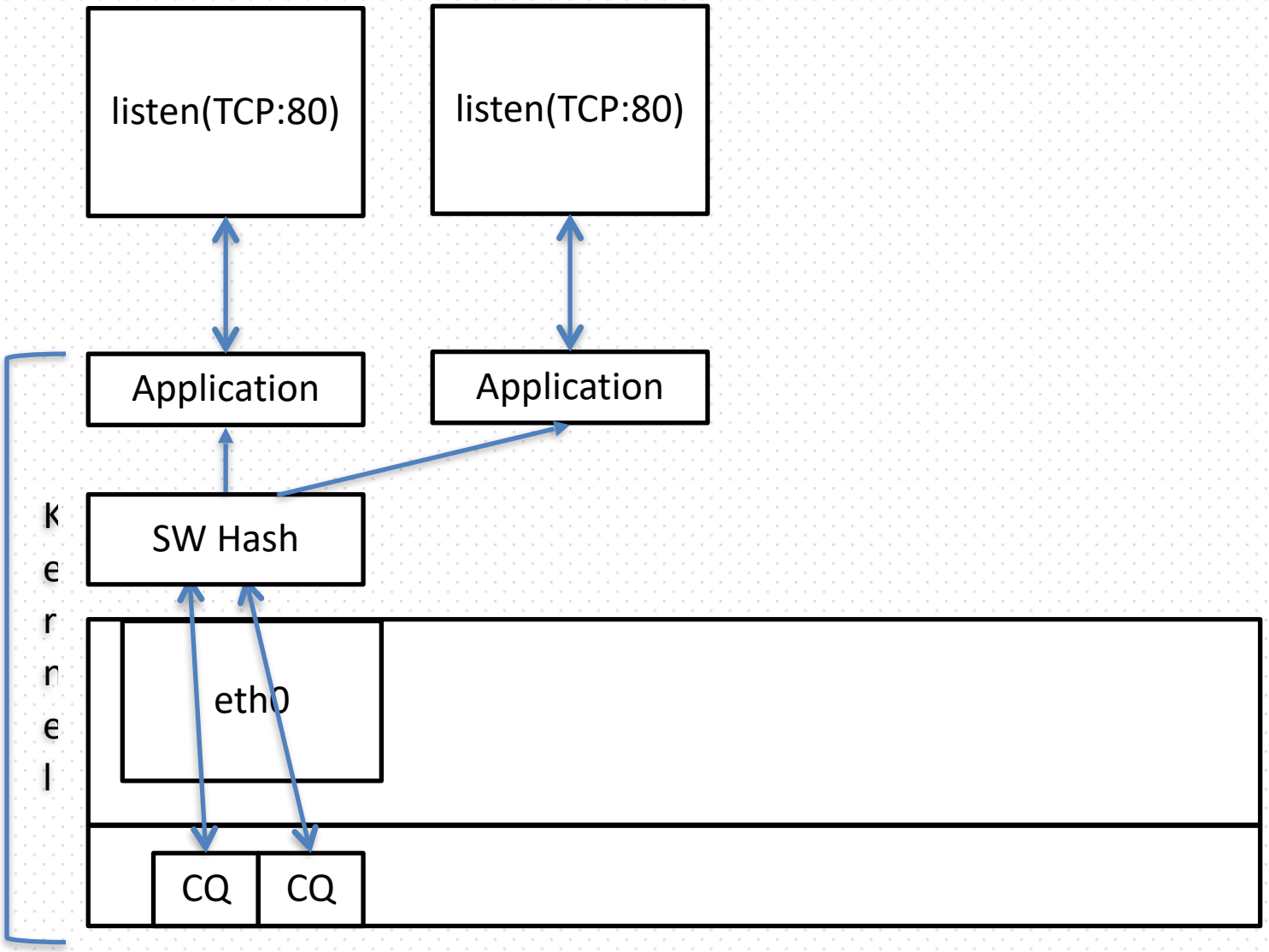
https://www.xilinx.com/publications/onload/sf-onload-nginx-proxy-cookbook.pdf

Nginx proxy: 1K HTTP GET

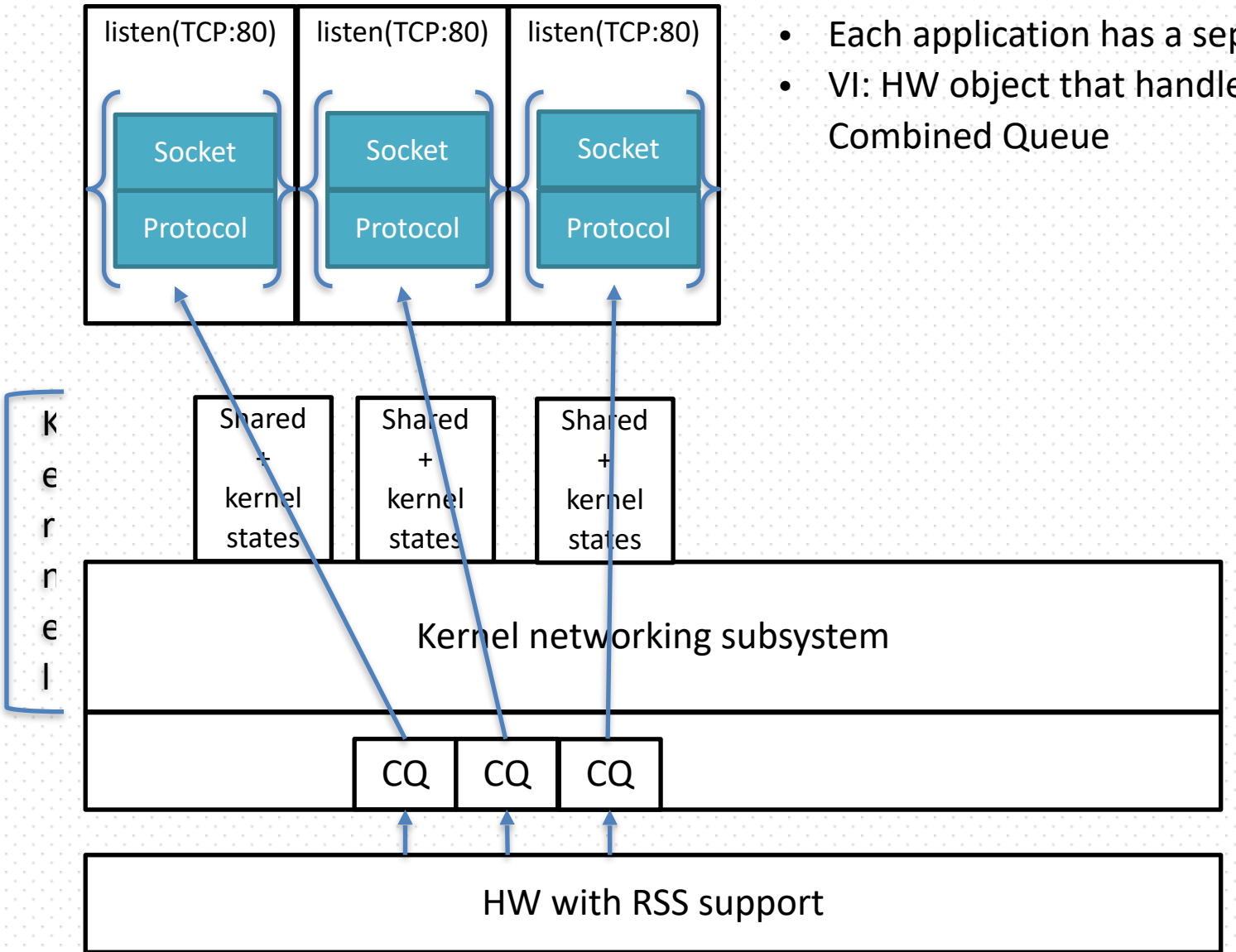
- Why:
  - Fast networking
  - Good RSS
  - Careful work with memory (no cache bouncing, coherency)
- Smart interlocking between processes working on different CPUs
  - If they use different stacks!
- All processing within the stack sticks to one CPU



Architecture: RSS + SO_REUSEPORT



Architecture: RSS + SO_REUSEPORT



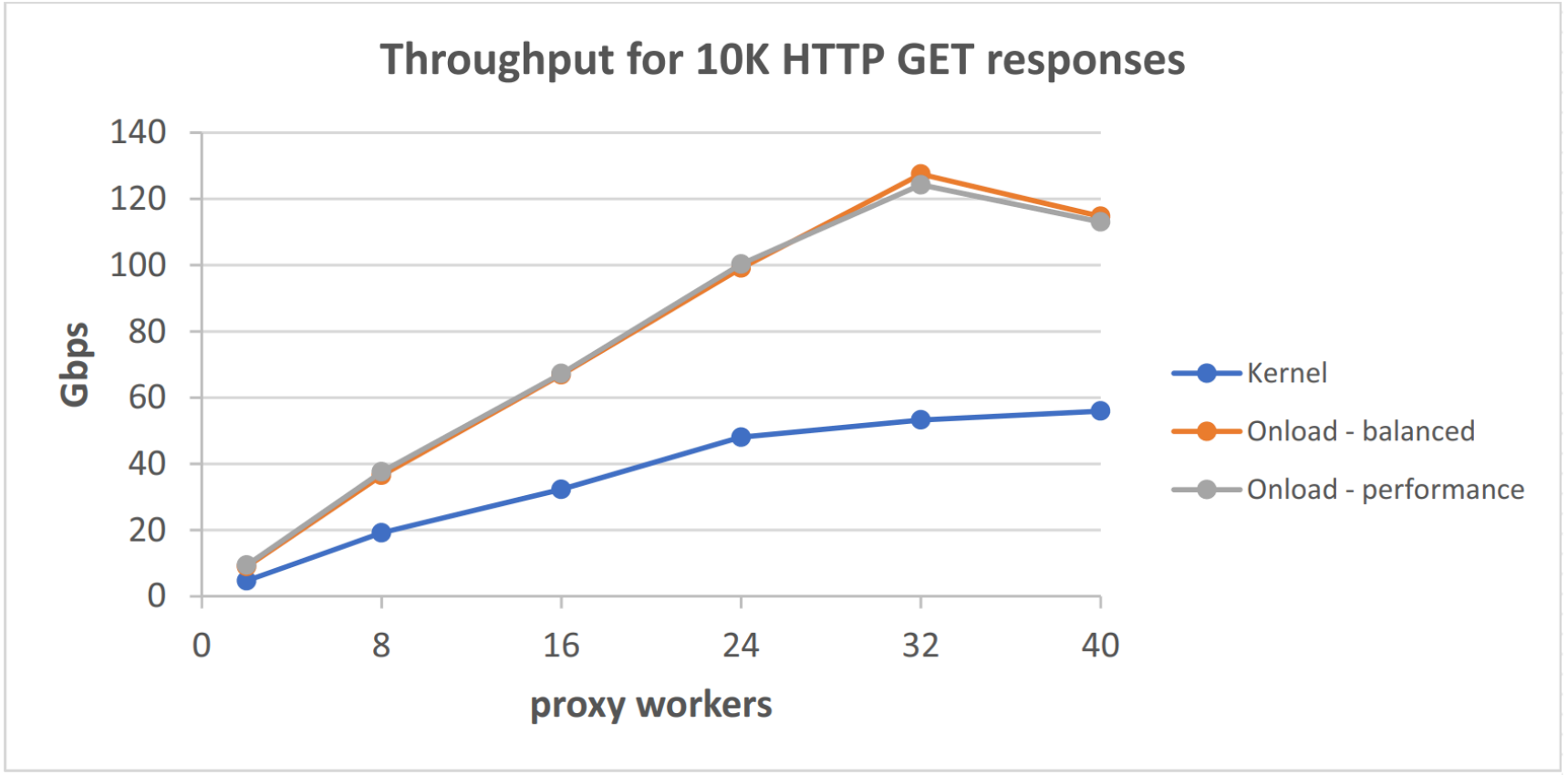
- Each application has a separate stack!
- VI: HW object that handles the Combined Queue

Architecture: RSS + SO_REUSEPORT

- RSS (even on NUMA): separate stack per worker means that almost nothing is shared: **no lock contention and cache bouncing**
- HW RSS distributes packets across userland processes and HW delivers packets **directly** to the process context
- Respect process CPU:
  - all memory is allocated in a smart way
  - interrupts are routed properly
  - etc.

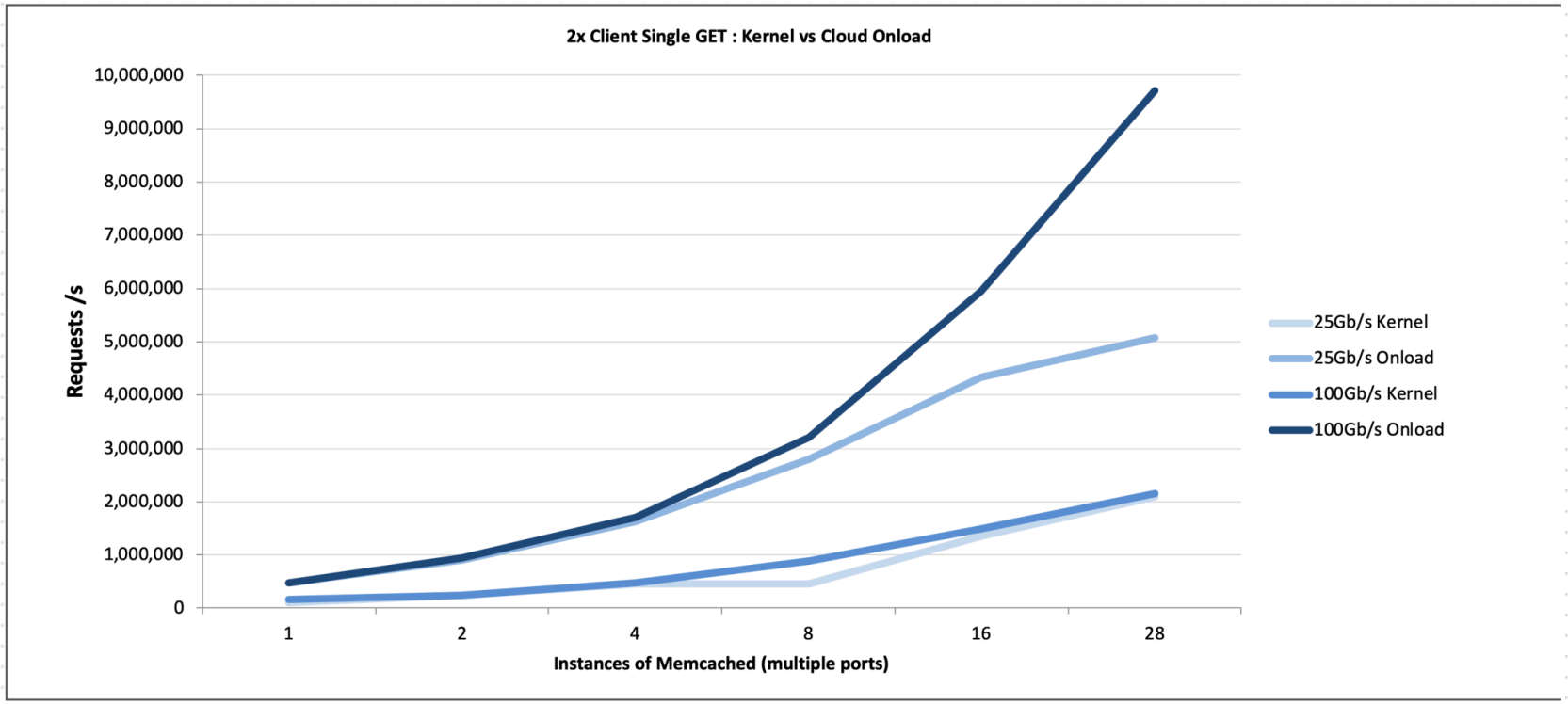
Nginx: response bandwidth (100G)

- Bandwidth is fine as well
- sendfile(): not yet supported, but it's faster even w/o it.



memcached: GET/SET

- Why:
  - Everything mentioned earlier
- 128byte GET
- 128byte SET: similar gains



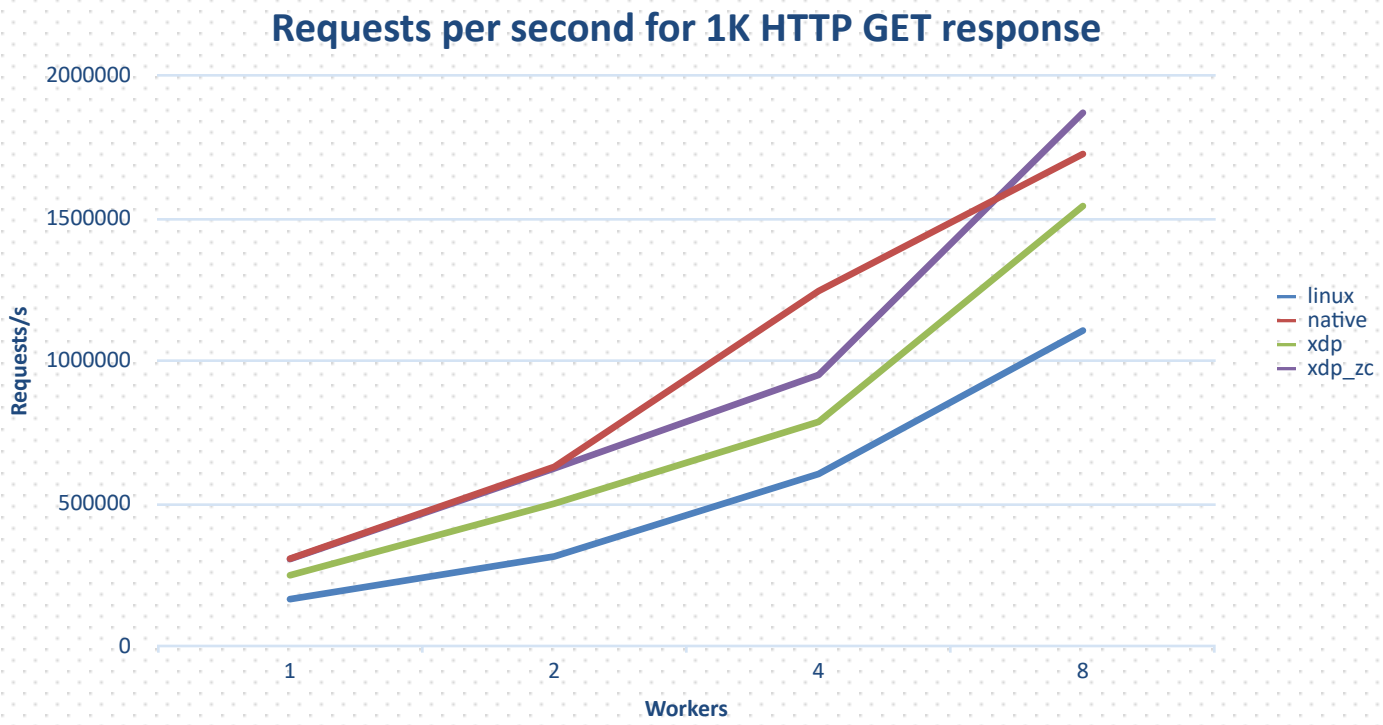
https://www.xilinx.com/publications/results/onload-memcached-performance-results.pdf

AF_XDP Performance

- Depends on:
  - The quality of the driver
  - Level of Zero Copy support
  - Decent RSS support (in HW and in the driver)
- HW:
  - AMD/SFC: how do we compare with native interface
  - NVIDIA/Mellanox/MLX
  - Intel
- Testbench:
  - Host with the NIC running NGINX: Intel(R) Core(TM) i9-10900X CPU @ 3.70GHz, 10 cores ← fast cores
  - Wrk: AMD Ryzen 7 5800X 8-Core Processor, 16 cores
- Tests:
  - NGINX Server
  - NGINX Proxy: July 2022 ☺

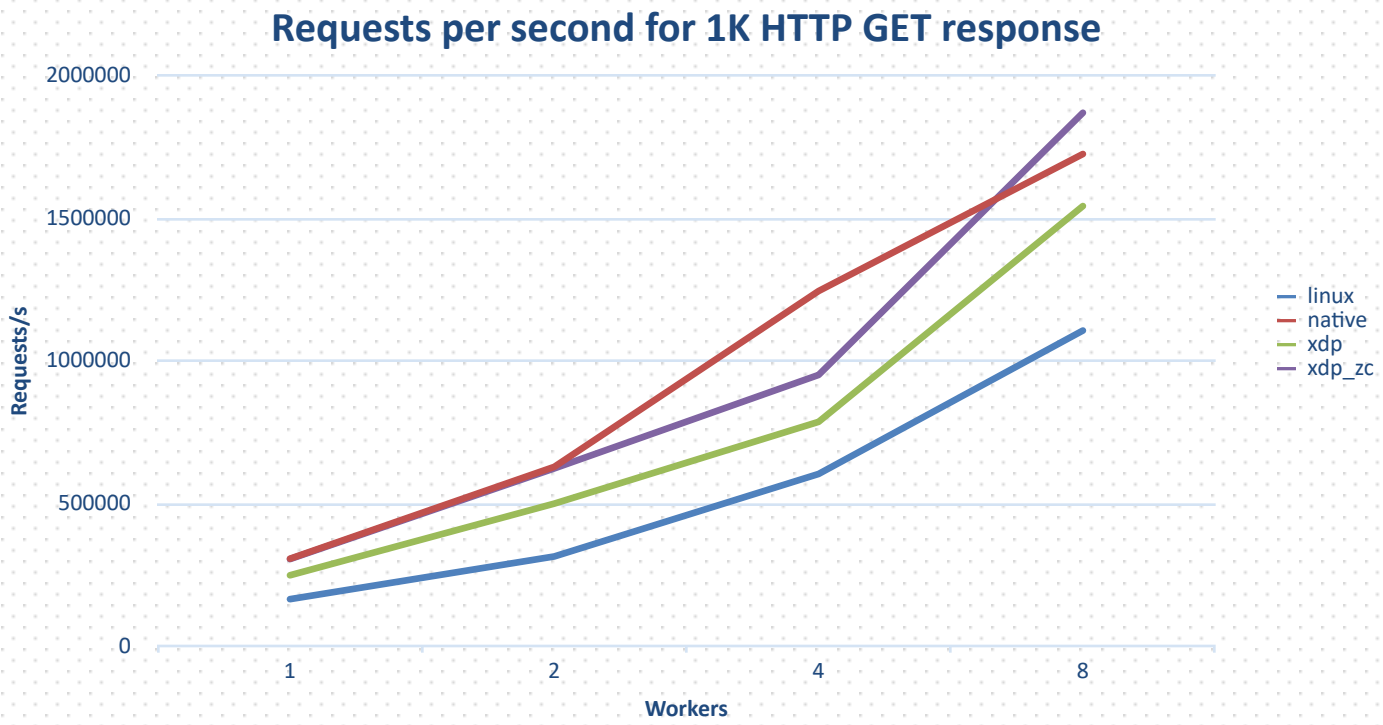
NGINX Server: SFC 25G NIC: 1K HTTP GET

	1	2	4	8
linux	167578.33	317125.77	606054.37	1108346.94
native	309284.29	630970.34	1246613.87	1726635.18
xdp	251184.04	502178.79	788586.34	1544329.47
xdp_zc	307911.72	625460.47	952858.99	1871078.5



NGINX Server: SFC 25G NIC: 1K HTTP GET

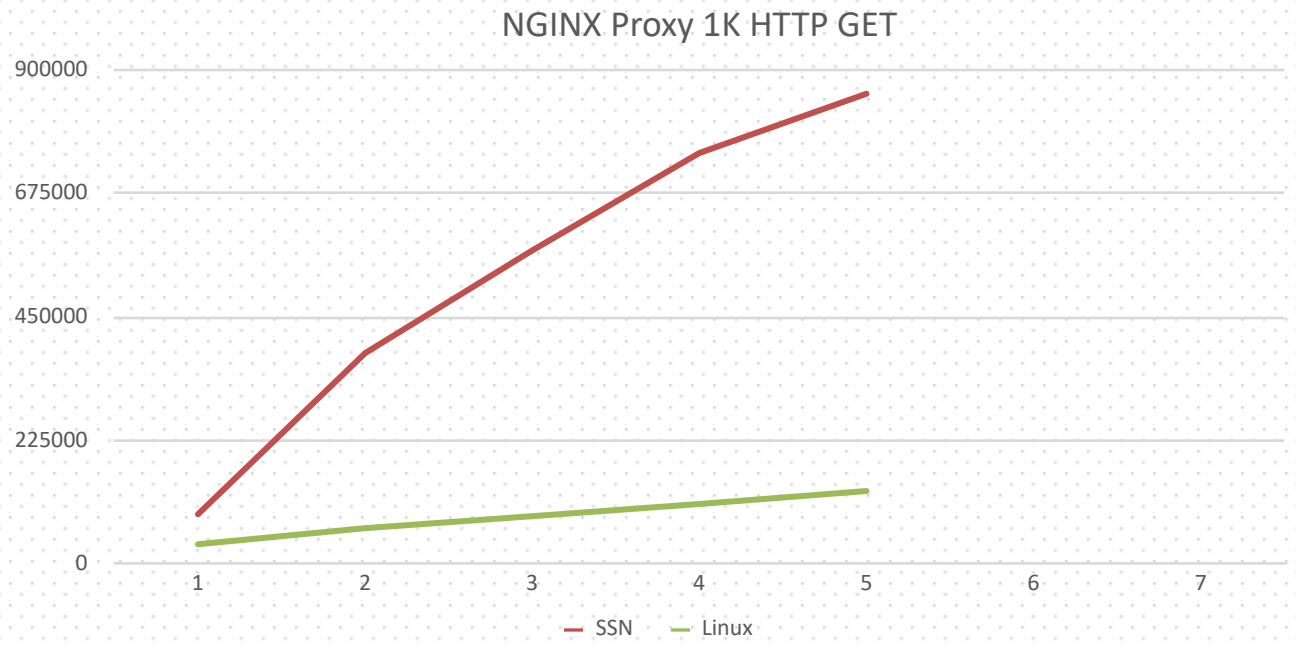
- Absolute numbers: high, cores are faster than on the 40core system
- Zero Copy works
- And ZC XDP is roughly as as good as native HW interface



NGINX Server vs. Proxy: quick check

- Gain in proxy mode is higher, than in server
- Happy with server → happy with Proxy
- Example: native HW interface (no AF_XDP), SSN vs. Linux

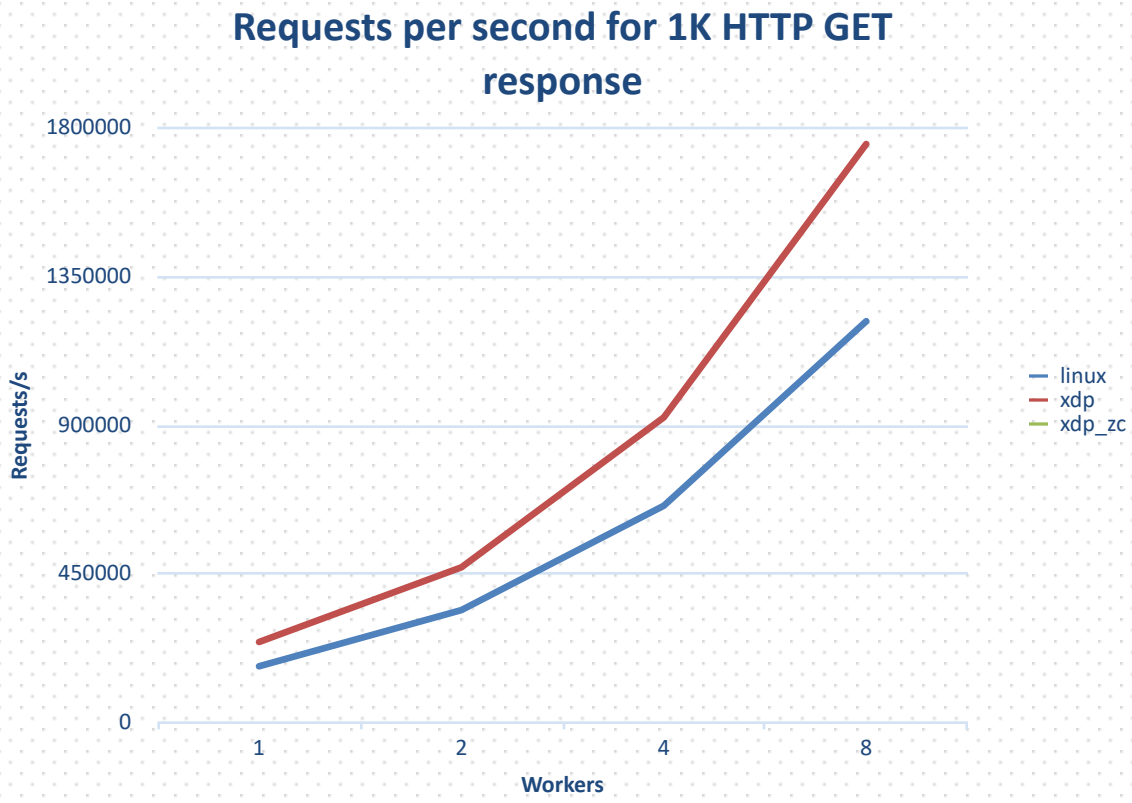
Workers	SSN	Linux
1	91219.53	36738.04
2	384276.65	65952.9
3	571066.69	87719.54
4	748314.86	109909.16
5	856125.79	133497.12



NGINX Server: MLX5 25G NIC: 1K HTTP GET

- MLX ZC: have "issues" in the driver, should be worked around or fixed

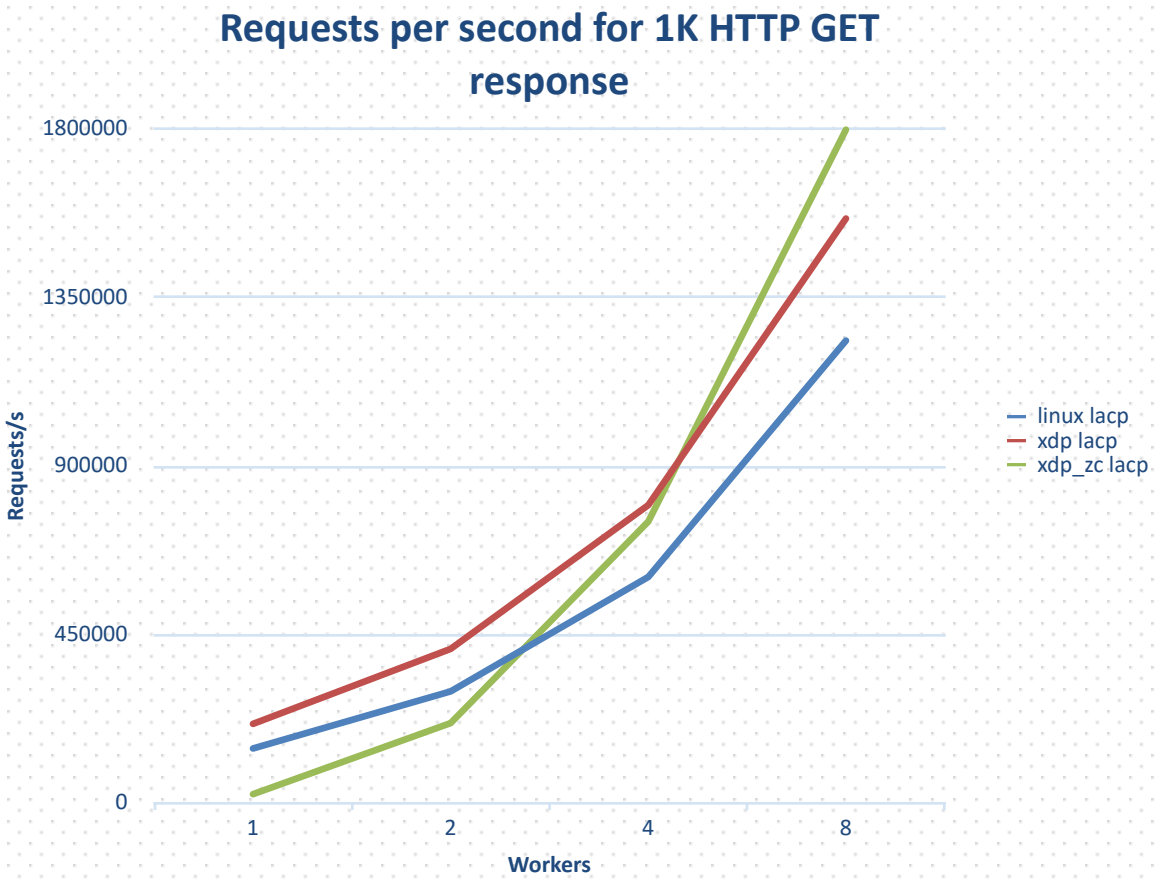
	1	2	4	8
linux	172225.23	342268.78	657950.43	1216125.6
xdp	245660.62	471660.72	925296.79	1752169.76
xdp_zc				



NGINX Server: Intel 10G NIC: 1K HTTP GET

- We do bonding (2x10G)
- Why ZC has problems: we suspect Intel driver has a missing memory barrier

	1	2	4	8
linux lacp	146219.52	298627.41	604154.76	1235179.87
xdp lacp	211911.26	412078.64	796024.93	1561102.68
xdp_zc lacp	24205.81	213999.5	752093.2	1798326.9



Block

- Why and what has changed?
- Networking
  - Kernel bypass networking
  - Speed-Stack Net
  - Performance
- **Block**
  - Speed-Stack Block
  - Performance
- Filesystems: stay tuned

Approaches

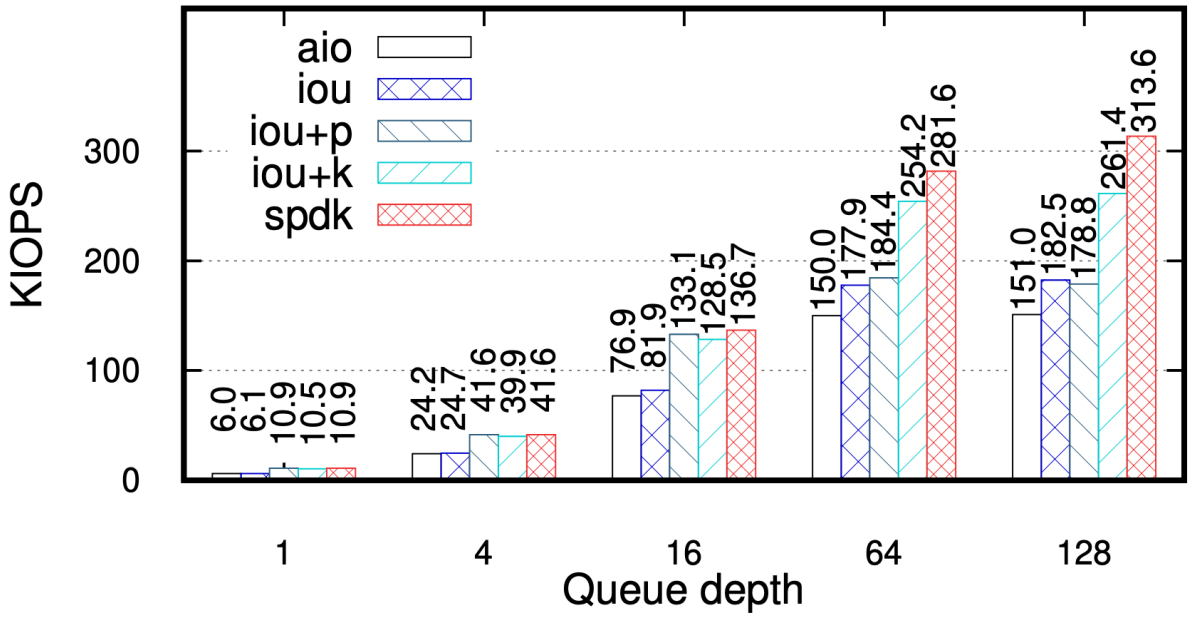
- SPDK:
  - Requires full rewrite
  - Supports a number of block devices (modules): NVMe, NVMeoF, CEPH, RAIDs and others
  - Faster than anything you can come up with
  - Based on DPDK (with all the goodness).
  - Flexible threads configuration (how many IO submitters you have etc.)
- IO_uring:
  - Modern kernel interface
  - Requires application rewrite, but less than SPDK
  - Pretty fast and good for co-routine architecture
  - Has a number of issues in real life scenarios, comparison

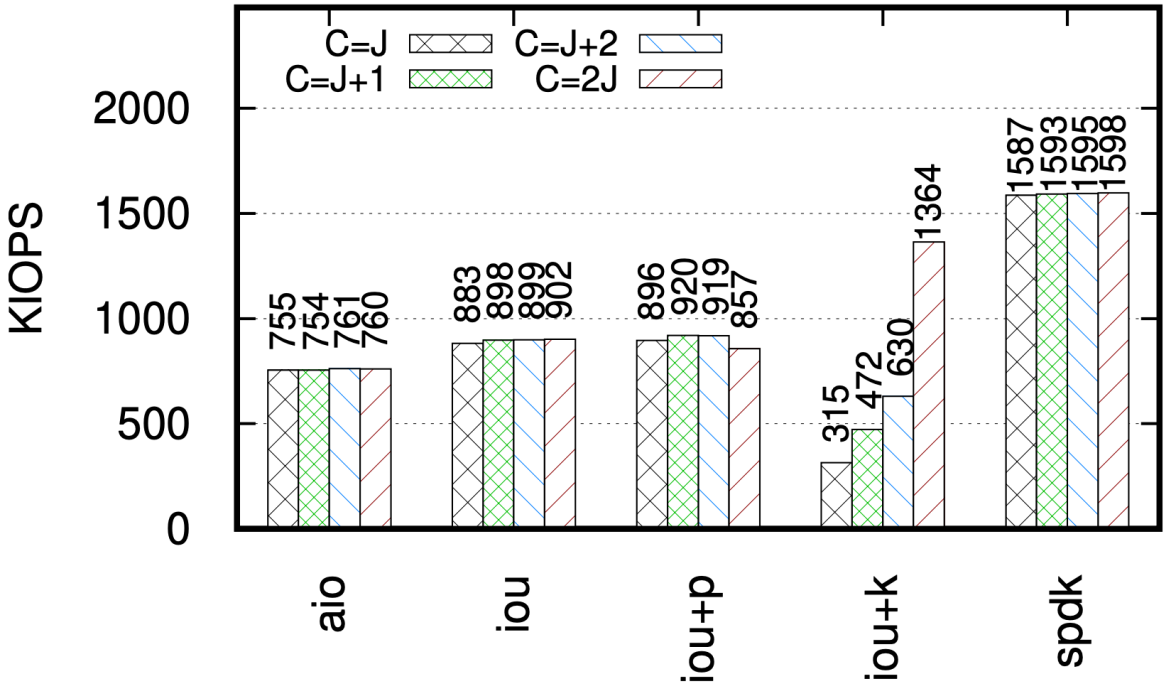
- Note: became a hot subject mostly with NVMe drives, that's why you don't have as many options as for Net

SPDK vs. IO_uring

- IO_uring: performance can be good if you have an IO submitter thread in the kernel
- IO_uring: Works fast only if you have Cores = 2xJobs

- SPDK: allows hand control over all aspects, including making sure data gets to disk etc
- SPDK: requires more work/thoughts… but does it?





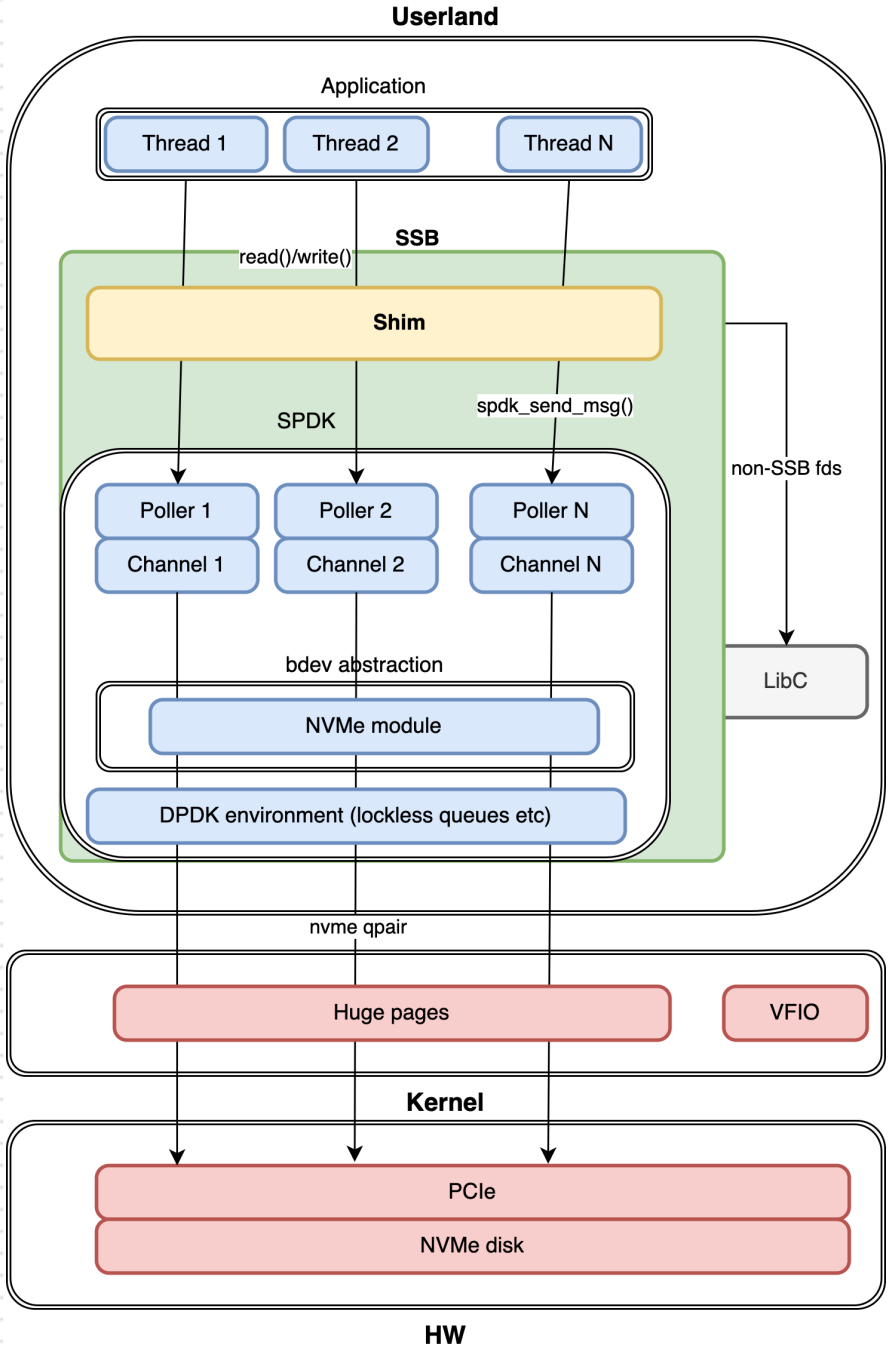
Can we take SSN approach and apply to block IO?

- Applications that use traditional interfaces:
  - read/write/pread/pwrite
  - LibAIO
- Disk/file IO interface is simpler than Socket API
- Make them fast w/o need for rewrite
- Avoid syscalls
- Ideal world: if we say read data and send into the network it should be fast and should not require a copy

- SPDK vs IO_uring: for now SPDK, but it is actually flexible and can be changed later on

Architecture: SPDK-based

- Shim similar to SSN, intercepting relevant POSIX API functions
- Userland-only, kernel only provides memory mapping
- Tricky bits: similar to SSN
  - Fork()/Exec(): main problem since we don't have kernel module



Question: SPDK for block, but not DPDK for net?

- It is possible to reimplement SSN over DPDK
- It has certain challenges
- Main reason: DPDK is all or nothing
  - non-accelerated applications can’t run on the same interface
- Likely doable and will likely be very fast, but harder than it might look

Content

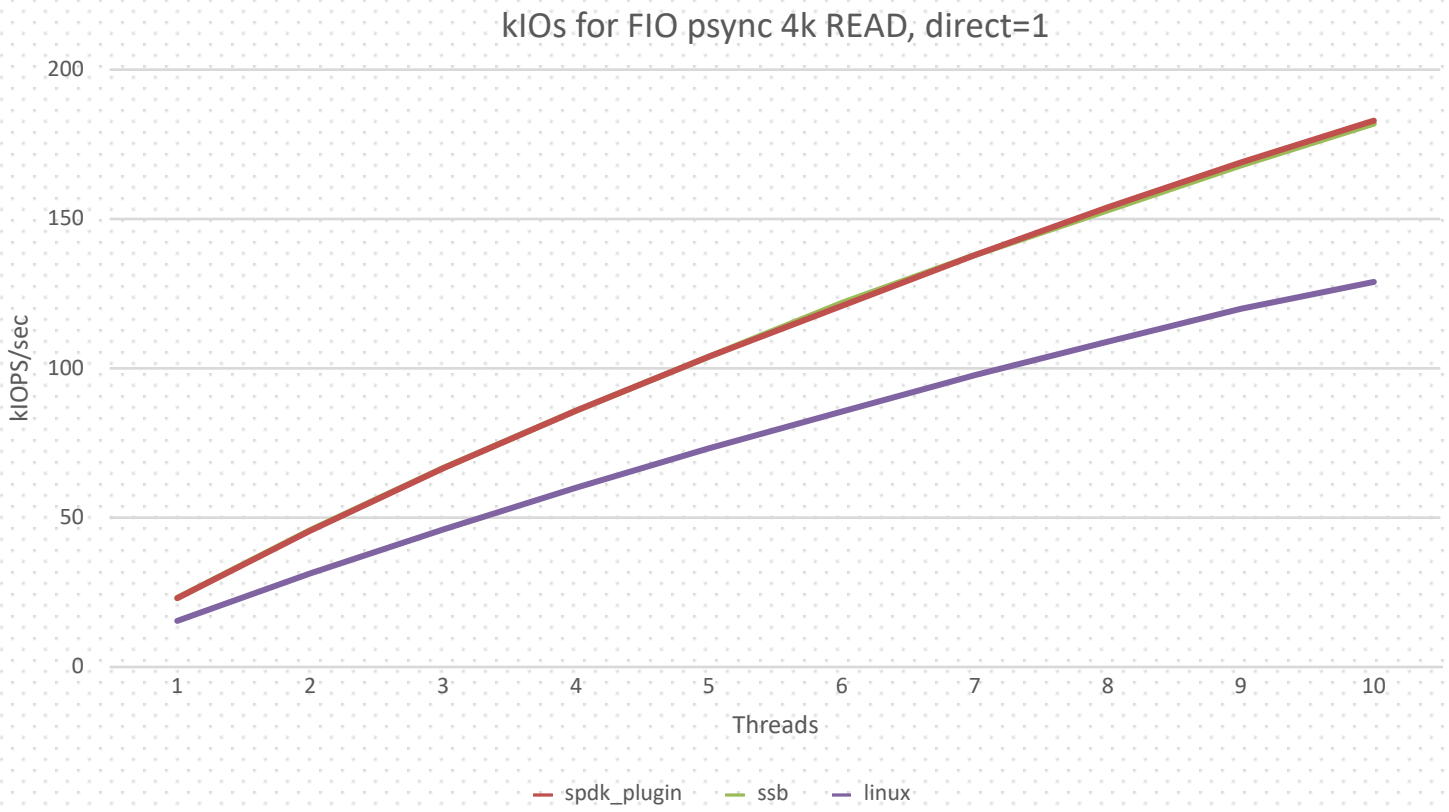
- Why and what has changed?
- Networking
  - Kernel bypass networking
  - Speed-Stack Net
  - Performance
- Block
  - Speed-Stack Block
  - **Performance**
- Filesystems: stay tuned

FIO: testbench

- NVMe driver: Samsung SSD 970 EVO Plus 500GB
  - Gen3
- CPU: Intel(R) Core(TM) i9-10900X CPU @ 3.70GHz, 10 cores
- FIO:
  - psync ioengine: what we want to accelerate
  - spdk_plugin: native SPDK engine that is as fast as FIO can go
- We disable caching (direct=1) to compare apples to apples and not caching systems
  - *SPDK – based cache should be more efficient since CPU locality should be better*

FIO psync with direct: kIOPS for 4k READ

Threads	SPDK	SSB	Linux
1	23.1	23.2	15.5
2	45.7	45.9	31.4
3	66.6	66.7	46.1
4	85.9	85.9	60.1
5	104	104	73.3
6	121	122	85.6
7	138	138	97.8
8	154	153	109
9	169	168	120
10	183	182	129



FIO psync with direct: kIOPS for 4k READ

- As fast as SPDK w/o need to modify the application
- Hitting PCIe/HW limitations

Questions: what's next

- Libaio: July 2022
- Caches: August 2022

- SSN+SSB: seamless integration of two
  - Read and send into the network w/o ever touching or (hopefully) a copy
  - Later 2022

Thank you.

Konstantin.Ushakov@oktetlabs.ru