End-Host Networking

These slides are taken from the lessons of Prof. Gianni Antichi @Polimi

for the Network Computing course

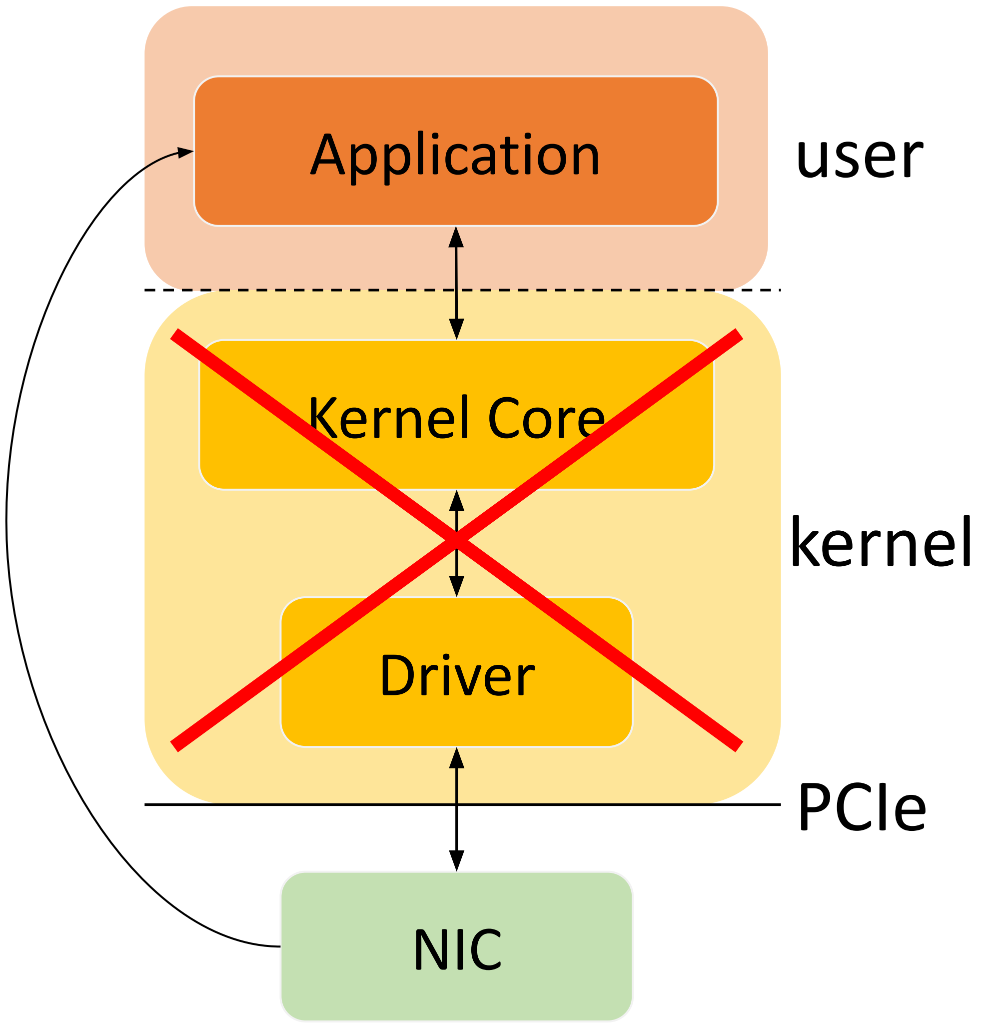
Advanced network processing

DPDK,RDMA and eBPF

Option #1: kernel bypass

Option #1: kernel bypass with DPDK

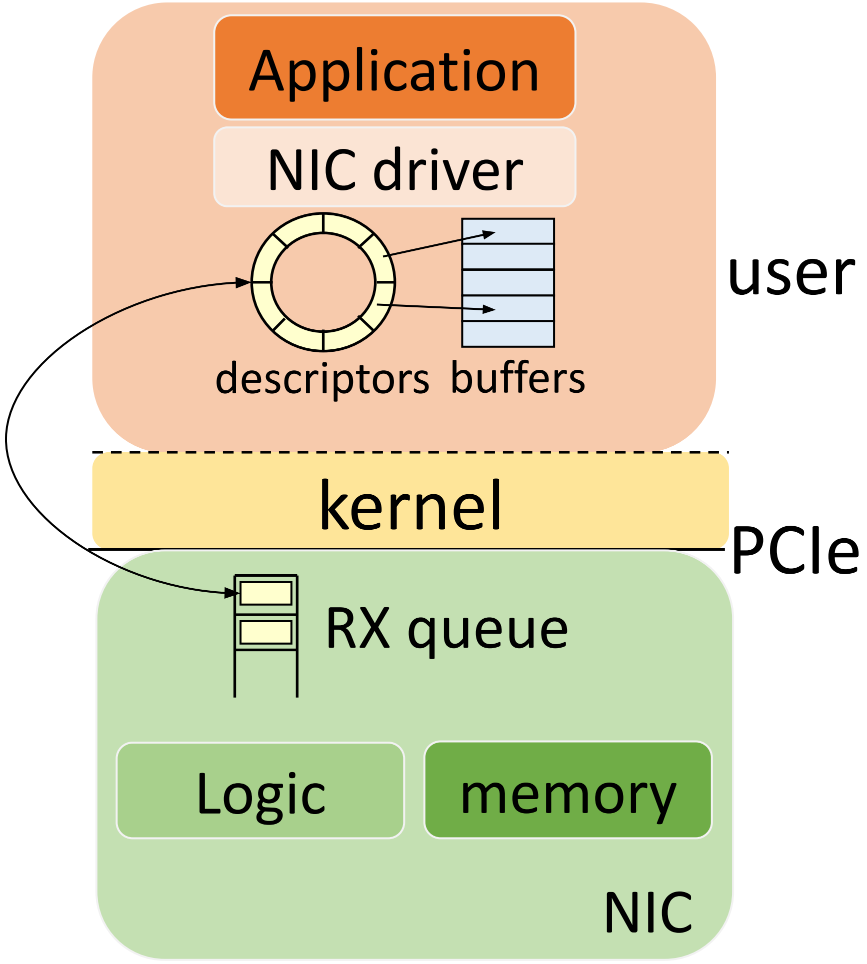
- Bypass the Kernel!
- Example: Data Plane Development Kit (DPDK)
- DPDK is a set of libraries and drivers that allows to bypass the Linux kernel stack and pass the packet directly at user-space



Option #1: kernel bypass with DPDK

- DPDK enables NICs to DMA data directly in user-space memory

- The NIC driver is a user-space application that interface directly with the NIC updating its RX queue with available descriptors

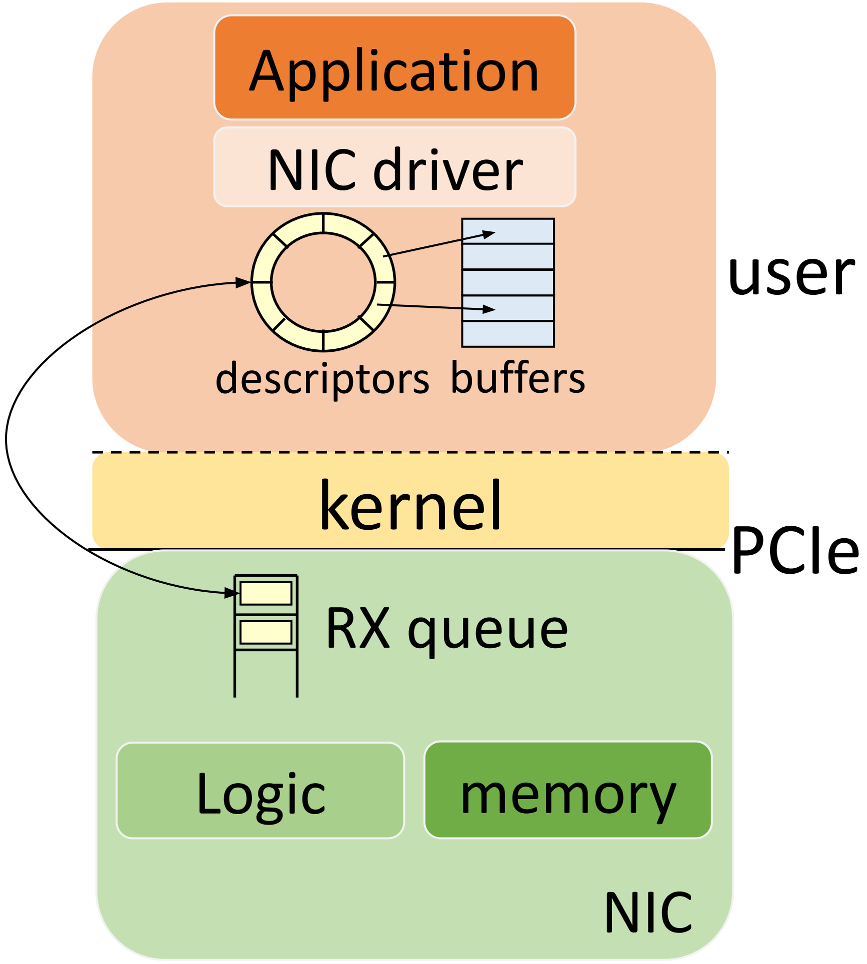


Option #1: kernel bypass with DPDK

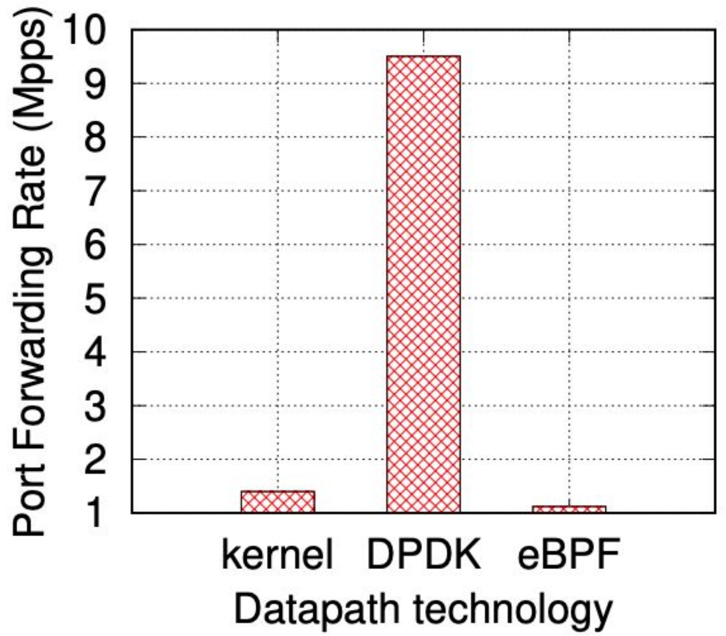
- DPDK drivers work in Poll mode

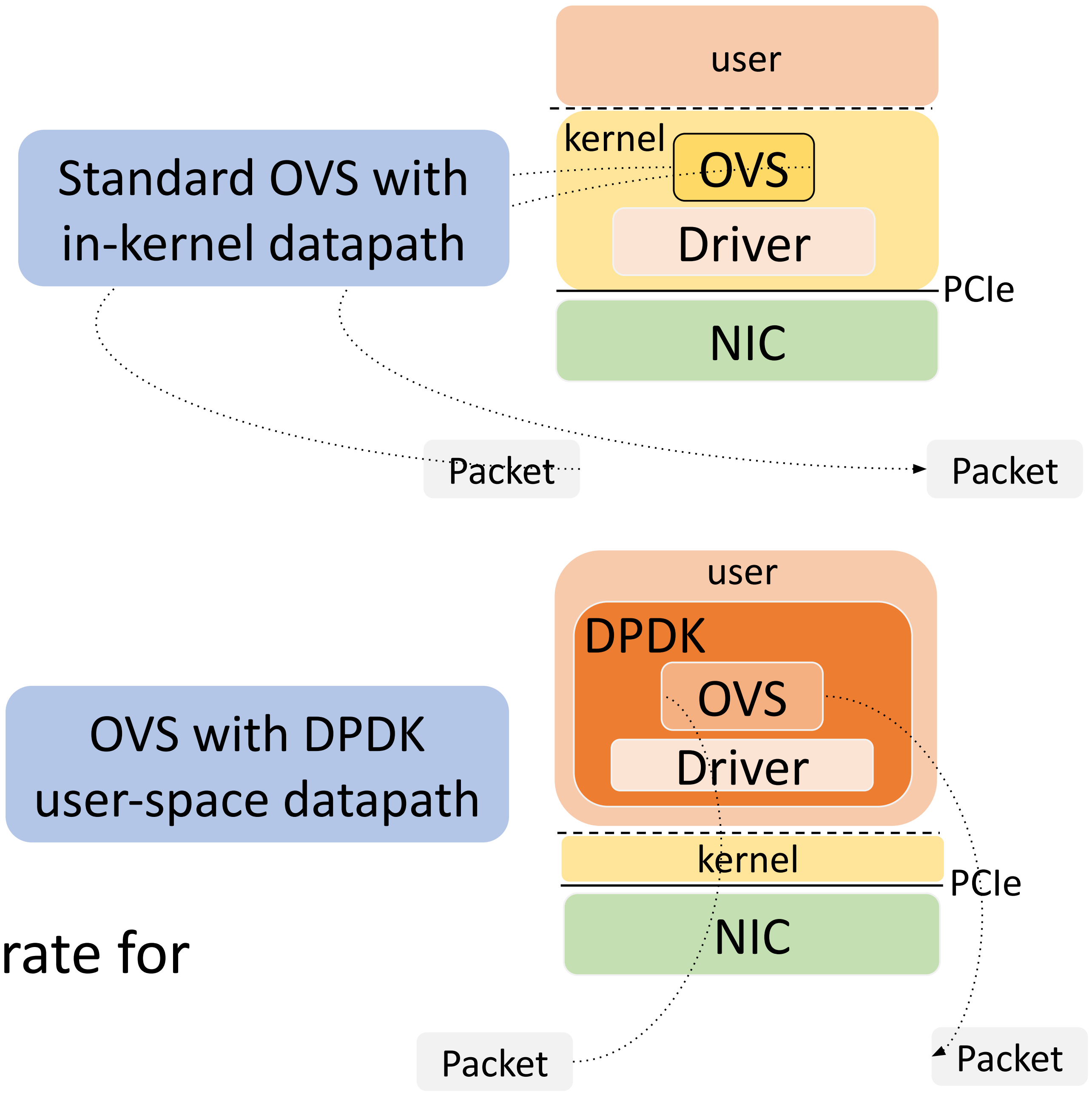
- Instead of the NIC raising an interrupt to the CPU when a packet is received, the CPU runs a **poll** mode driver (PMD) to constantly poll the NIC for new packets.

- As a consequence, a CPU core must be dedicated and assigned to running PMD.



DPDK vs Kernel

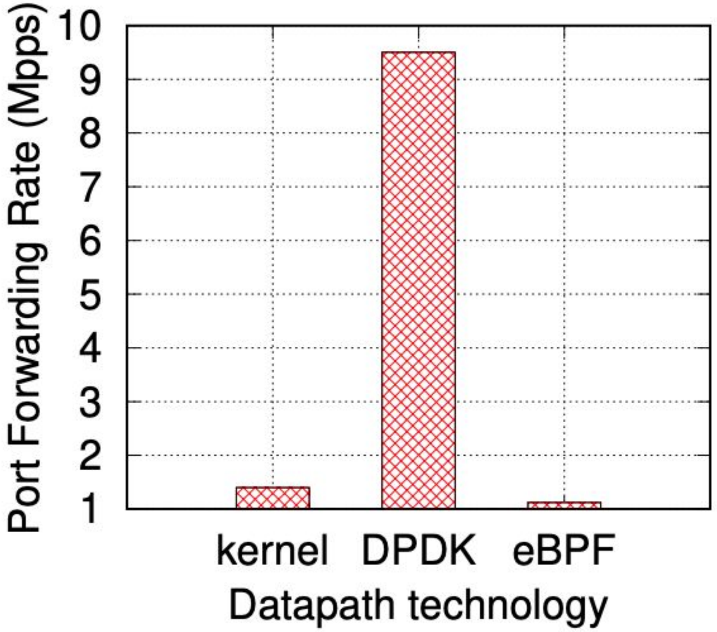


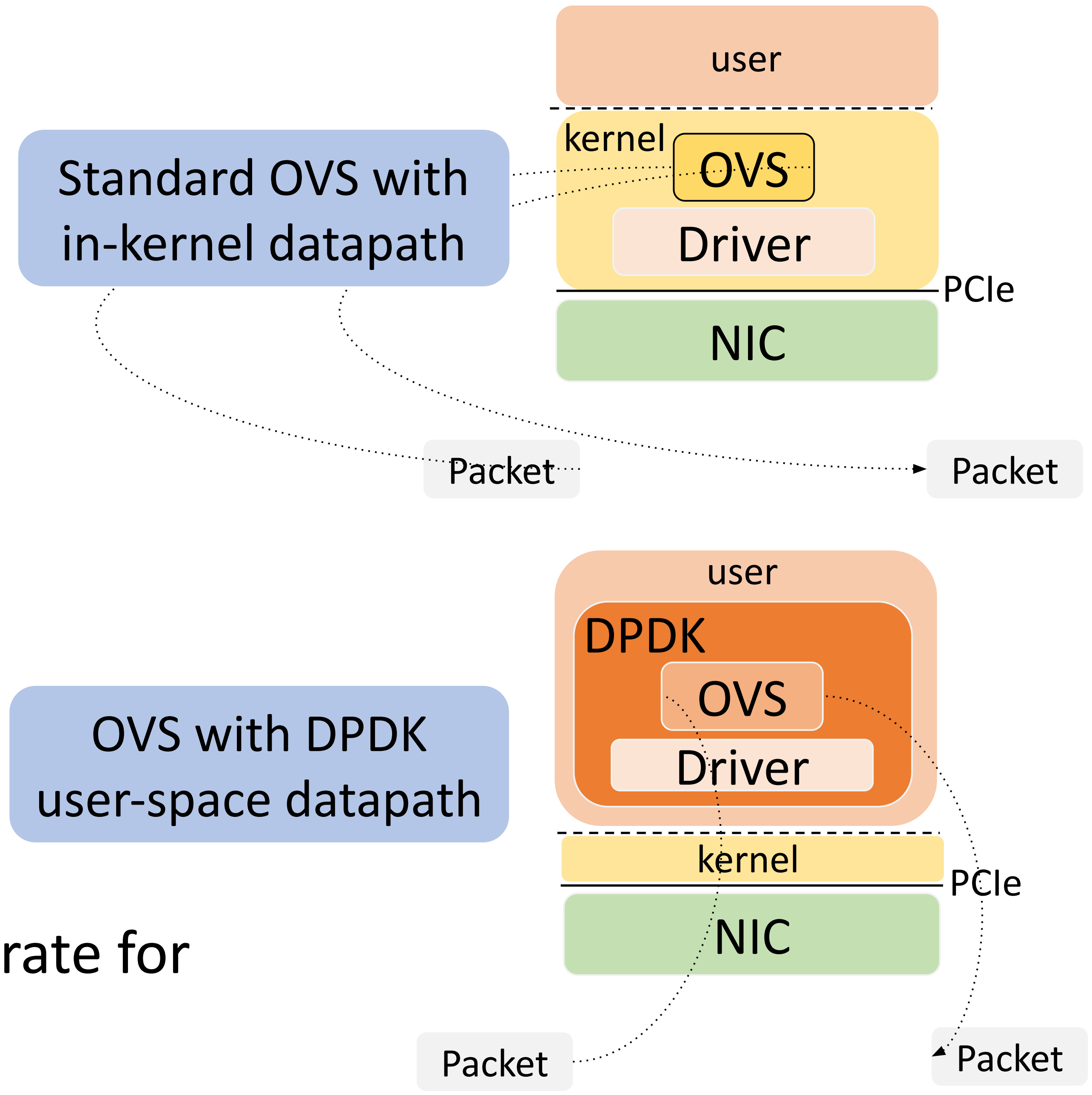


- This is OVS (single core) forwarding rate for 64B packets

DPDK vs Kernel

DPDK is fast ☺️

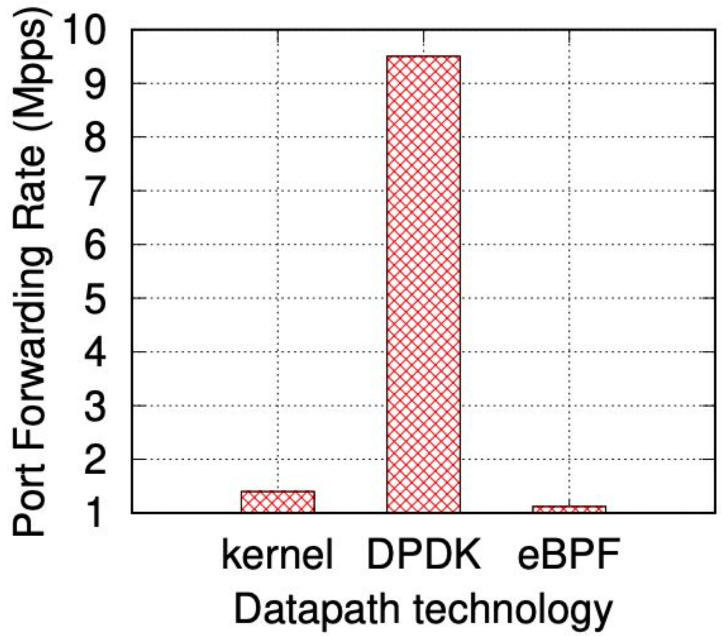




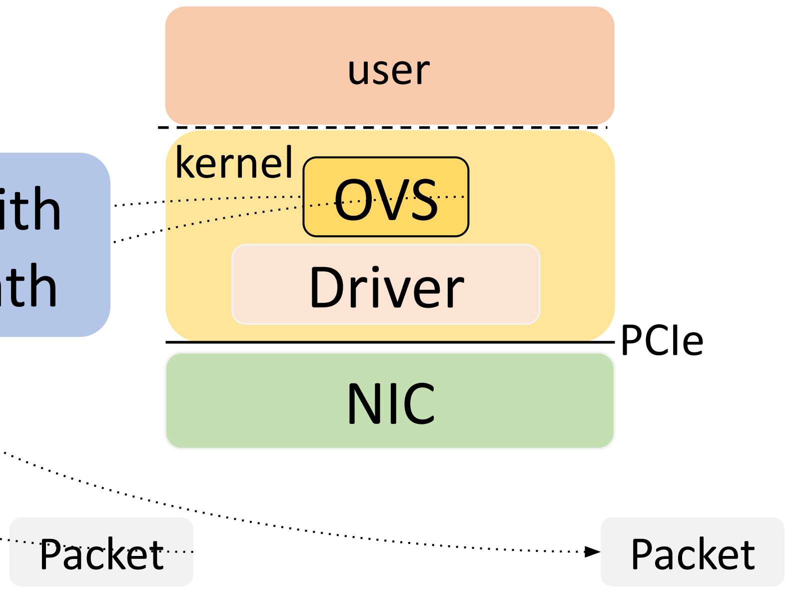
- This is OVS (single core) forwarding rate for 64B packets

DPDK vs Kernel

DPDK is fast ☺️

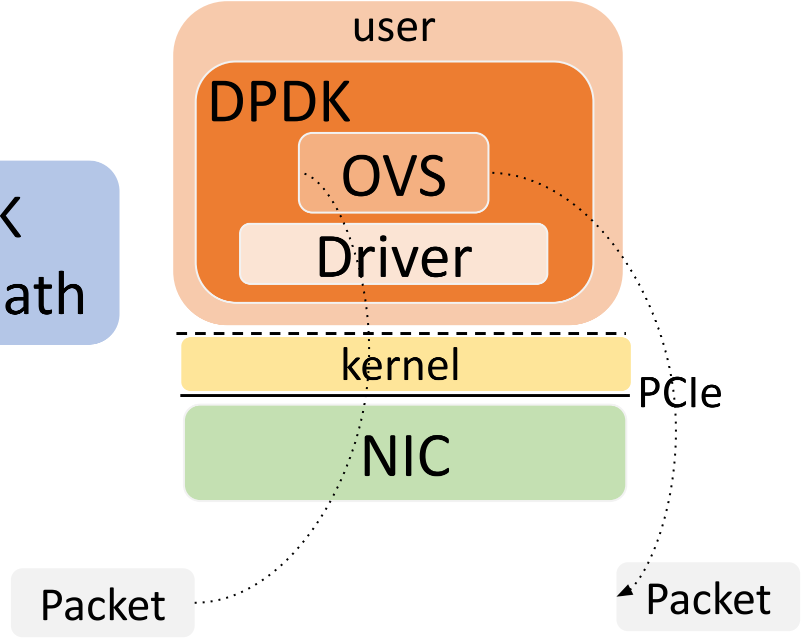


Standard OVS with in-kernel datapath



What's the catch then?

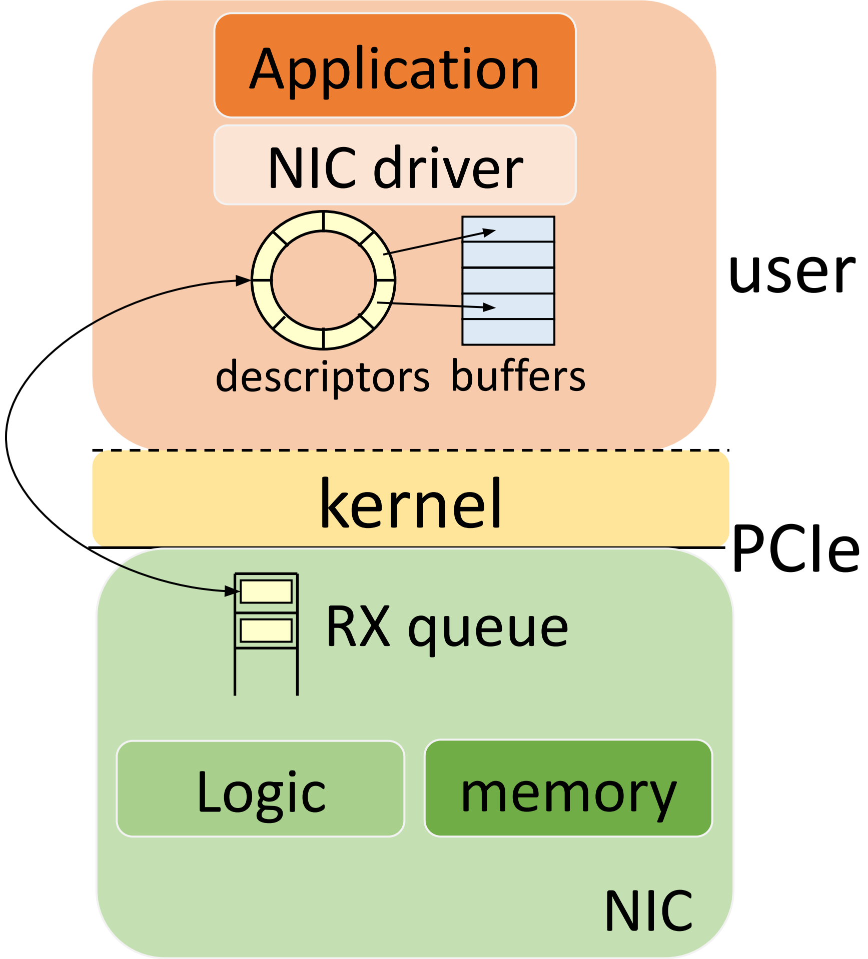
OVS with DPDK user-space datapath



- This is OVS (single core) forwarding rate for 64B packets

Problems with DPDK

- If you bypass the kernel, you give up all the features the kernel provides to you.
  - TCP/IP processing
  - NATting/Firewalling
- *ping* does not work anymore ☹️
- *tcpdump* does not work ☹️
- *ip link* does not work ☹️

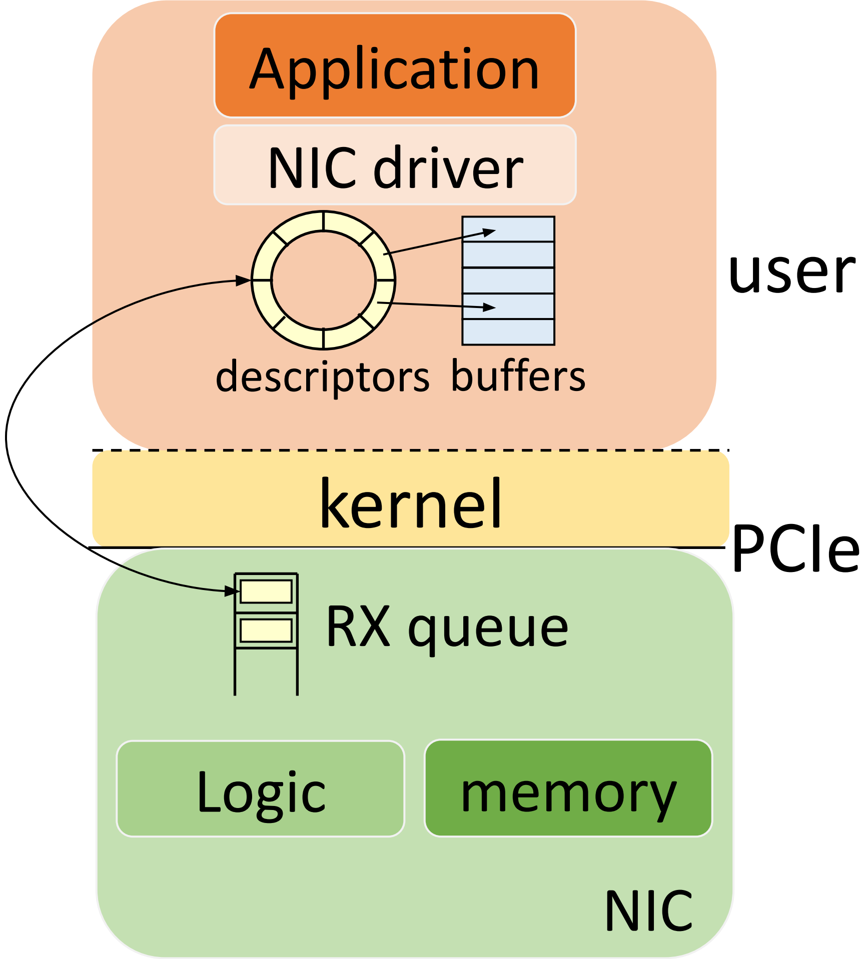


Problems with DPDK

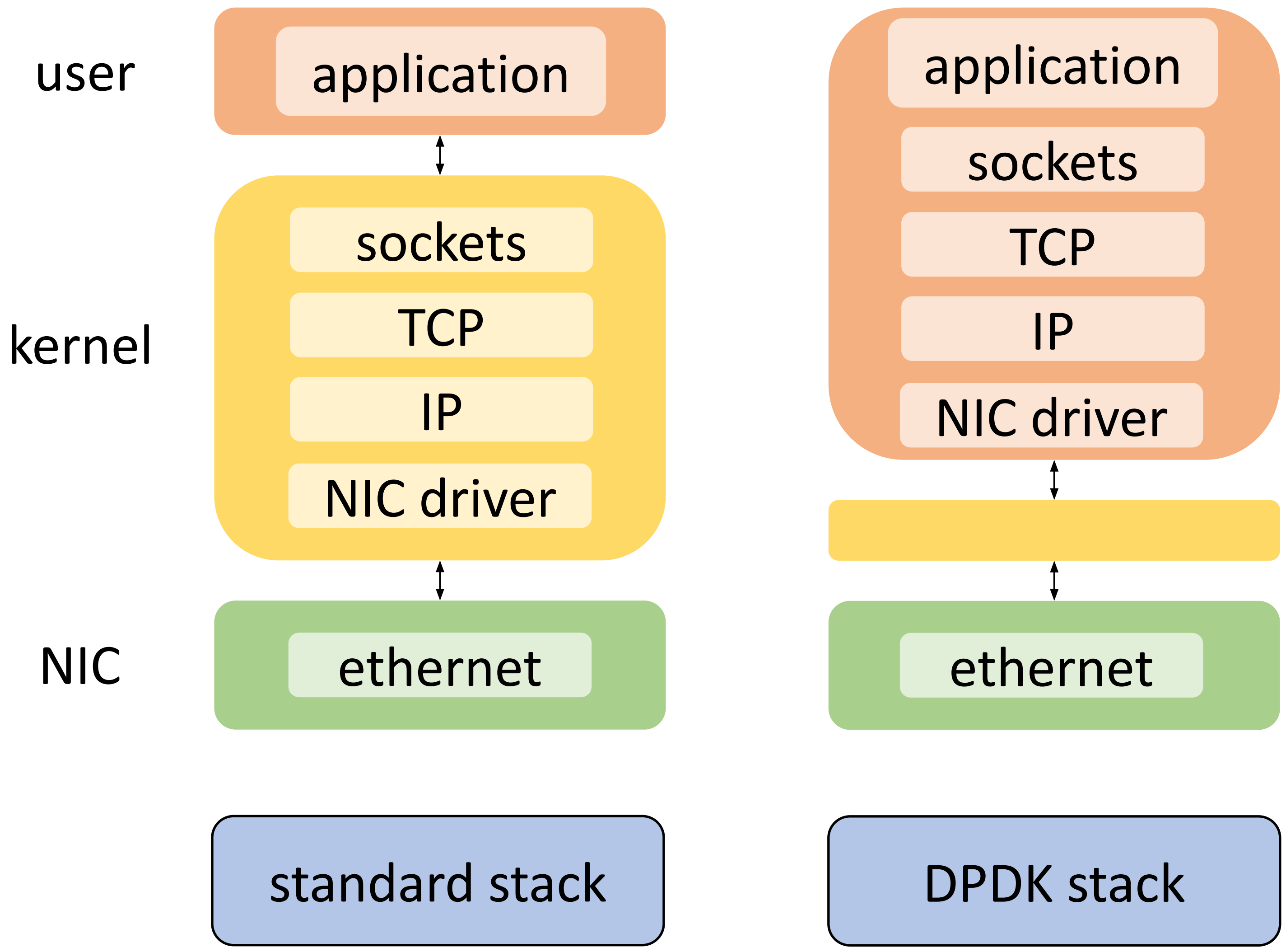
- No sharing of network resources
  - A single application take entire ownership of the DPDK port

- You can have different VMs sharing with Single Root Input/Output Virtualization (SR-IOV)

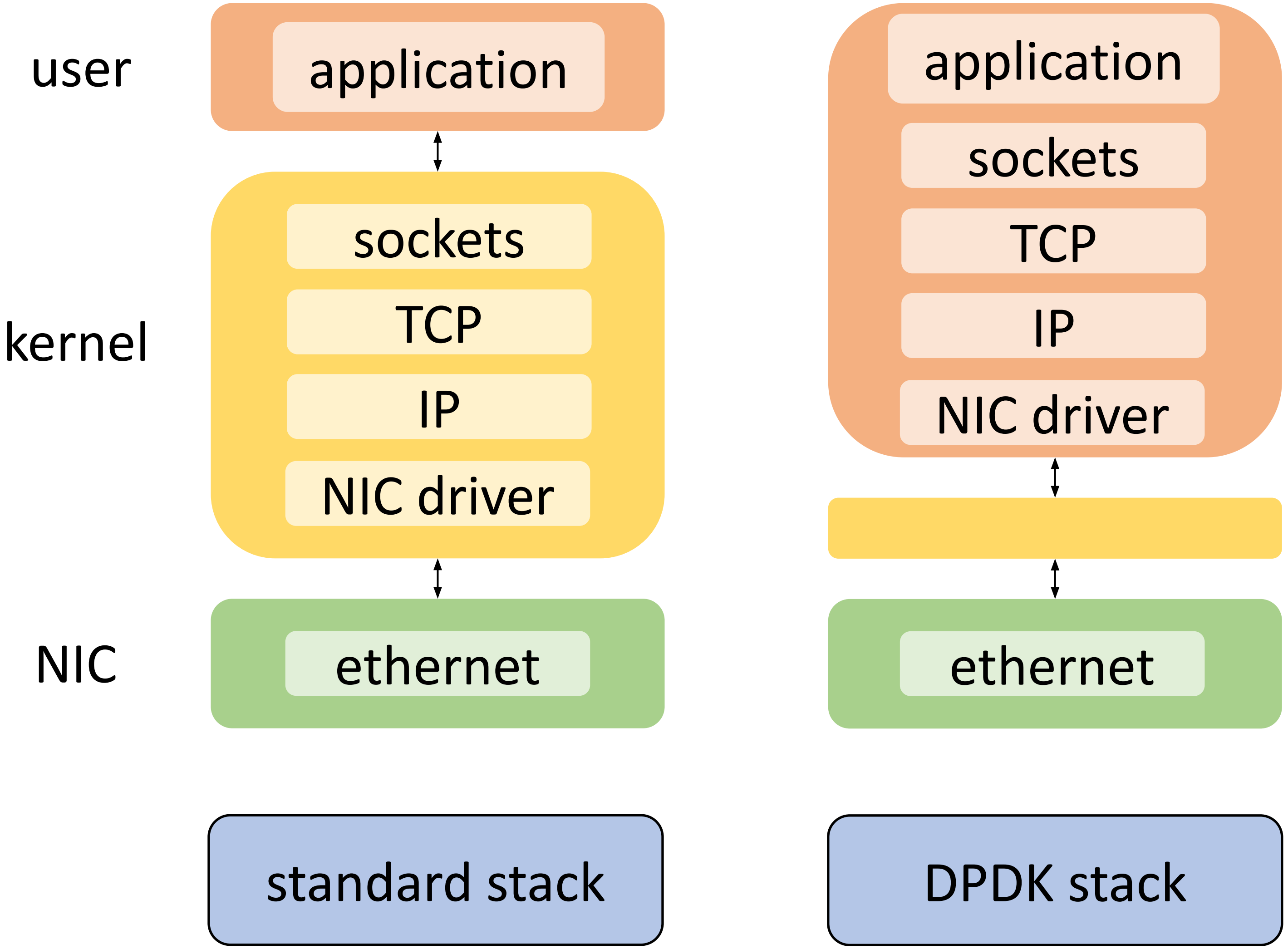
- SR-IOV is a specification that allows the isolation of PCIe addresses



Option #2: kernel bypass with a twist

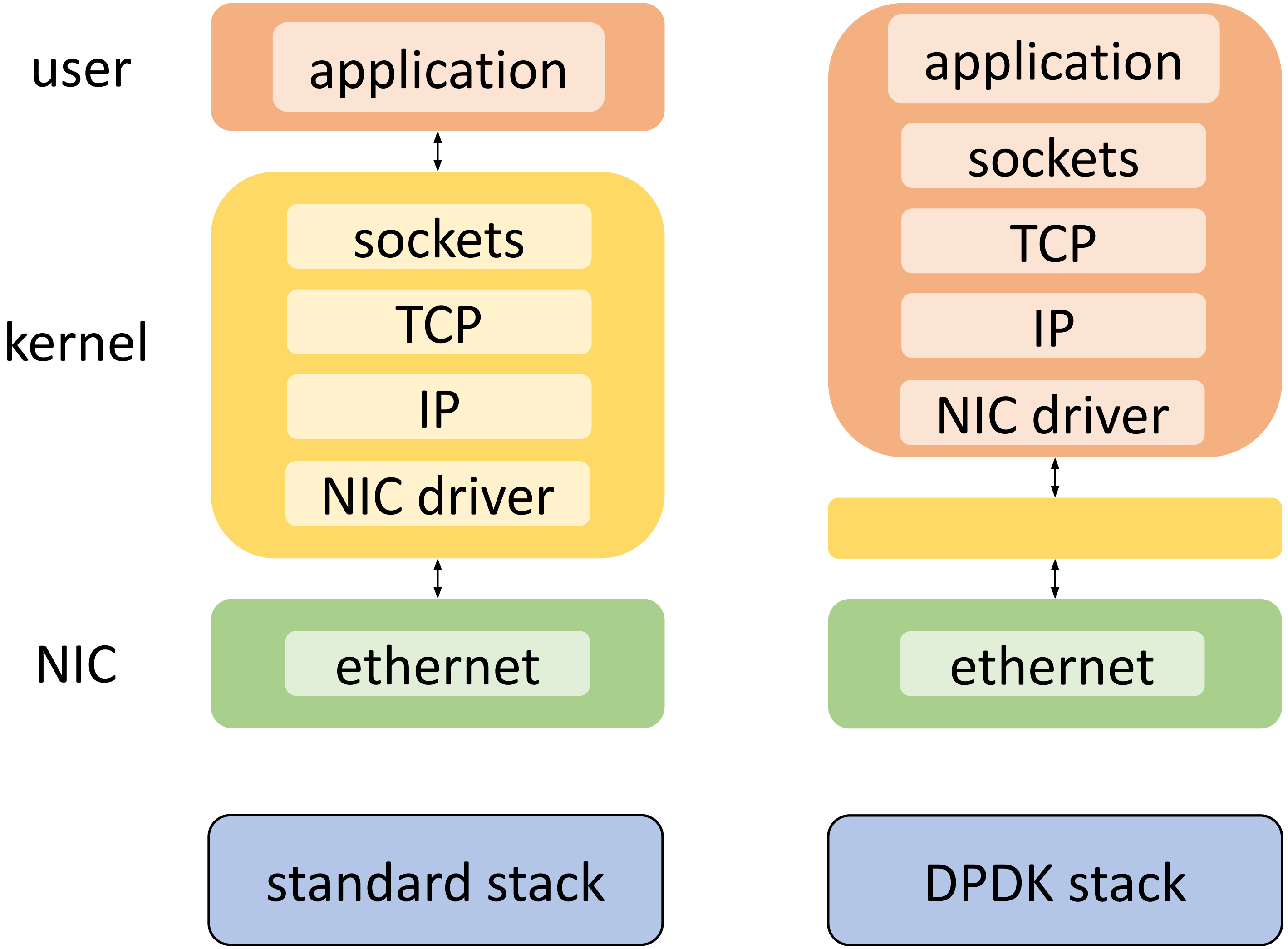


Option #2: kernel bypass with a twist



- **The problem:** we want to sustain high throughput, but the end of Moore's law and Dennard scaling imposes limits on computational capabilities of CPU cores
- More packets, more CPU cores spent in processing them, less resources for applications ☹ (CPU = $$$)

Option #2: kernel bypass with a twist

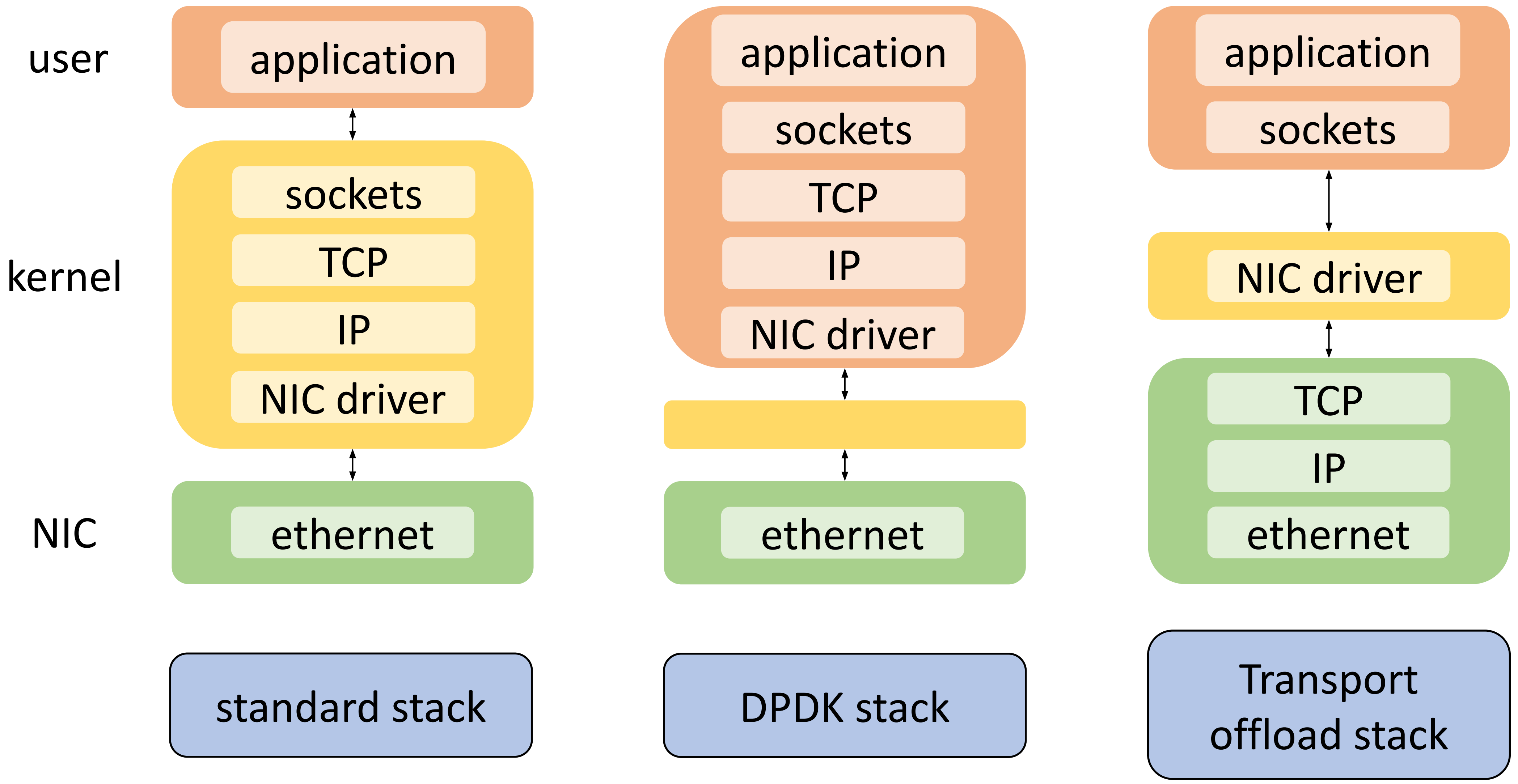


- The problem: we want to sustain high throughput, but the end of Moore's law and Dennard scaling imposes limits on computational capabilities of CPU cores
- More packets, more CPU cores spent in processing them, less resources for applications ☹ (CPU = $$$)

How to fix this?

Option #2: kernel bypass with a twist

Bring computation down to the NIC!

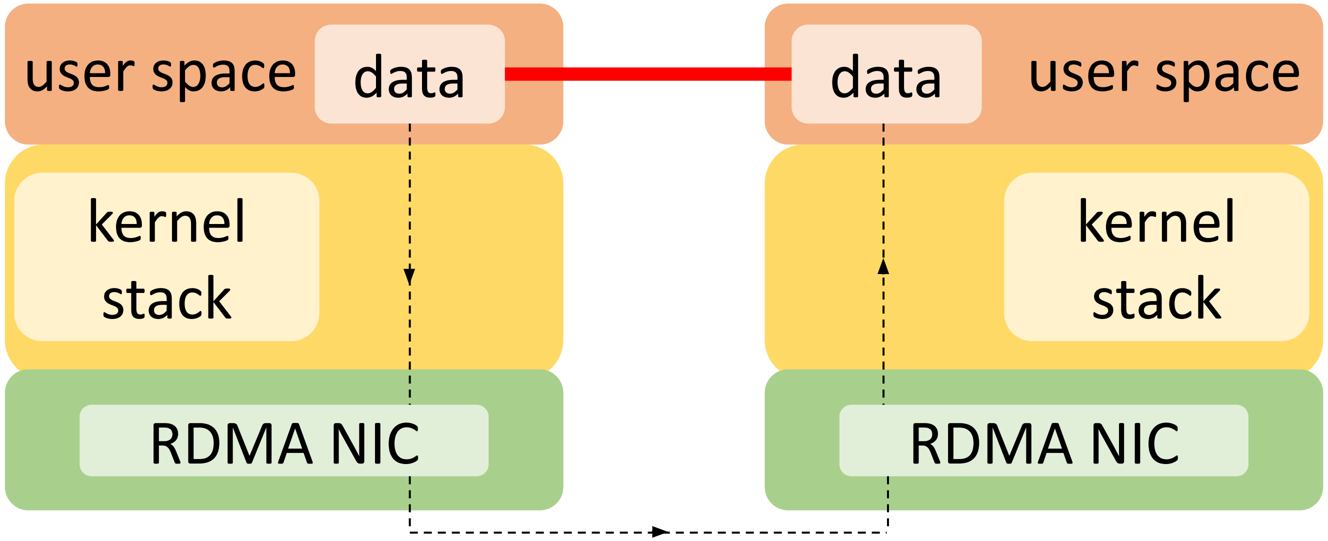


Option #2: RDMA

Option #2: kernel bypass with RDMA!

- **RDMA** is a mechanism that allow to access memory on a remote system bypassing the kernel stack

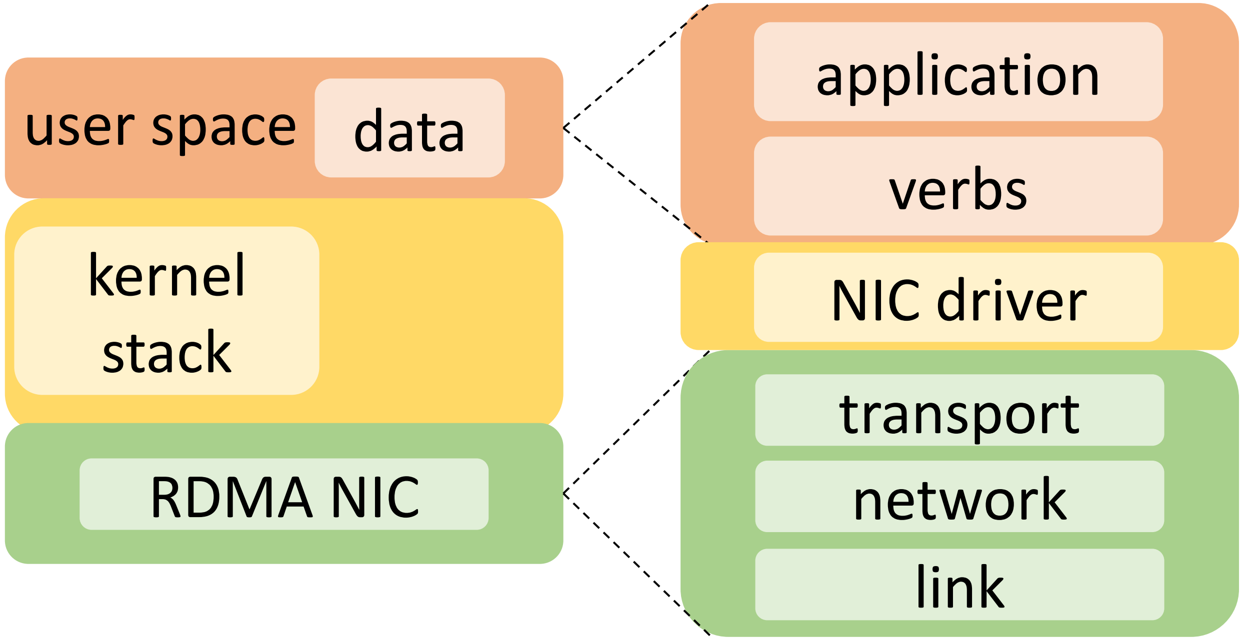
- RDMA can run over ethernet (RoCE v2) and it is encapsulated over IP/UDP



Option #2: kernel bypass with RDMA!

- **RDMA** is a mechanism that allow to access memory on a remote system bypassing the kernel stack

- RDMA can run over ethernet (RoCE v2) and it is encapsulated over IP/UDP



RDMA properties

Remote data is transferred between nodes in a network

RDMA properties

Remote data is transferred between nodes in a network

Direct no CPU or OS kernel is involved in the data transfer

RDMA properties

Remote data is transferred between nodes in a network

Direct no CPU or OS kernel is involved in the data transfer

Memory data transferred between two applications and their virtual address spaces

RDMA properties

Remote	data is transferred between nodes in a network
Direct	no CPU or OS kernel is involved in the data transfer
Memory	data transferred between two applications and their virtual address spaces
Access	support to send, receive, read, write and do atomic operations

RDMA properties

<u>Main characteristics:</u>

- Zero-copy data
- Bypasses the OS kernel

Remote data is transferred between nodes in a network

Direct no CPU or OS kernel is involved in the data transfer

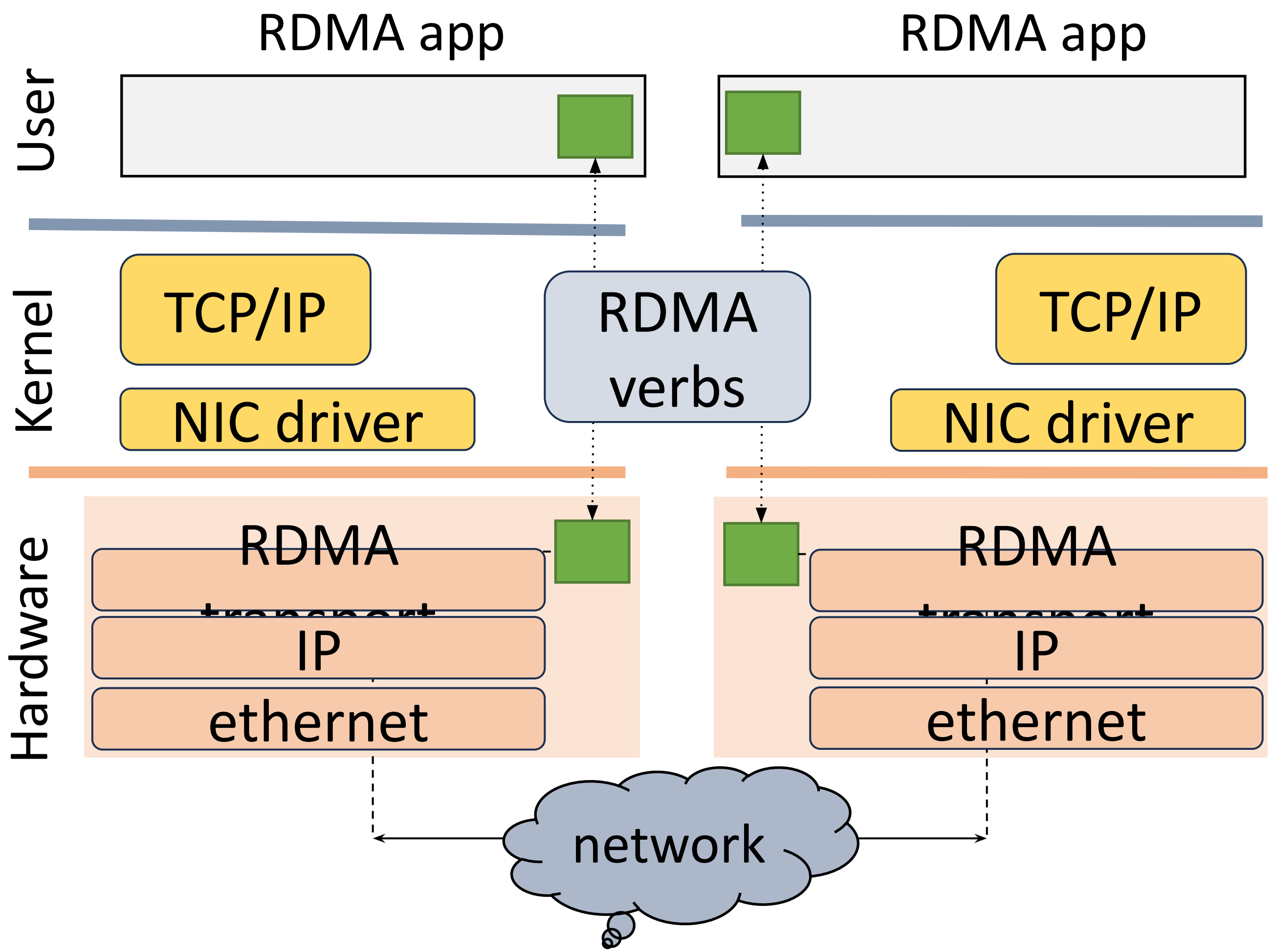
Memory data transferred between two applications and their virtual address spaces

Access support to send, receive, read, write and do atomic operations

Benefits of using RDMA

- High throughput
- Low end-to-end latencies
- Low CPU utilization
  - some RDMA operations do not involve the remote CPU at all
- Low memory bus contention
  - no data is copied between the user and kernel space

RDMA overview



Applications bypass the kernel and interact directly with the RDMA NIC using the verbs API provided by the NIC driver

RDMA verbs

RDMA networks support two types of memory access models

One-sided

Examples: RDMA read, RDMA write, RDMA atomic

Those are memory verbs:

- They specify the remote memory address on which operations should be carried out
- No involvement on the receive side

RDMA verbs

RDMA networks support two types of memory access models

<u>One-sided</u>

Examples: RDMA read, RDMA write, RDMA atomic

Those are <u>memory verbs:</u>

- They specify the remote memory address on which operations should be carried out
- No involvement on the receive side

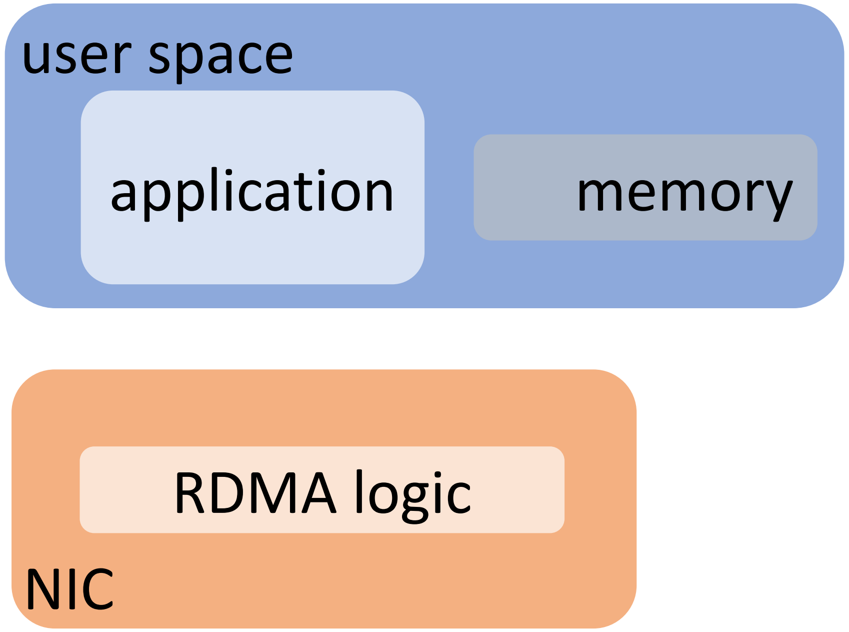
<u>Two-sided</u>

Examples: RDMA send, RDMA receive

Those are <u>messaging verbs:</u>

- This is similar to TCP sockets. Receiver must listen before sender issues messages
- Sender and receiver do not know each other virtual memory location

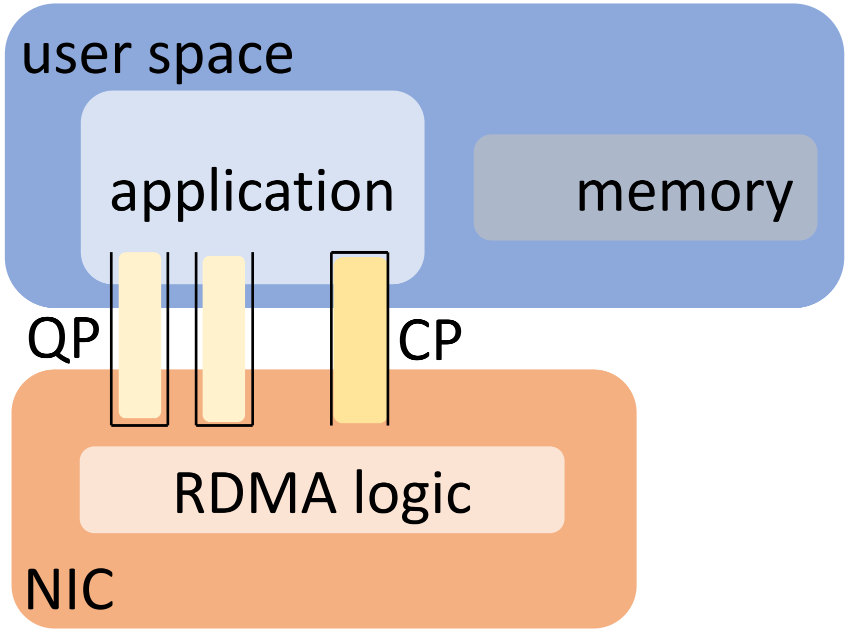
Setting up RDMA data channels



- Applications register memory regions with the NIC
  - Those are where data to be sent or received will be stored

- During the registration process:
  - **Pin memory** so it cannot be swapped by the OS
  - Store the address translation information in the NIC (from virtual to physical)
  - Set permissions for the memory region

Work Queues



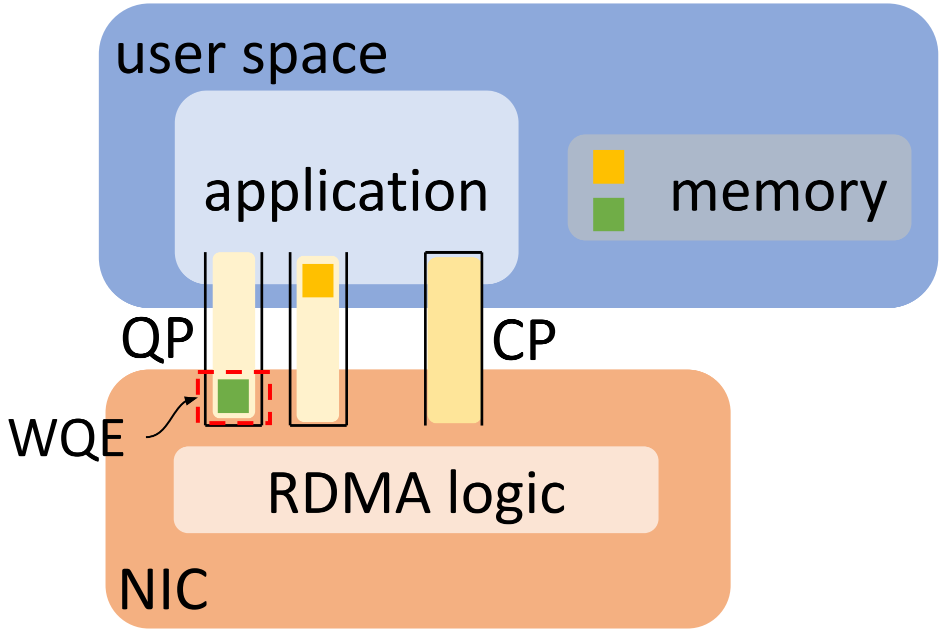
- RDMA communication is based on a set of three queues
  - Send
  - Receive
  - Completion

work queues, always created as a Queue Pair (QP)

- The send and receive queues are there to schedule the work to be done

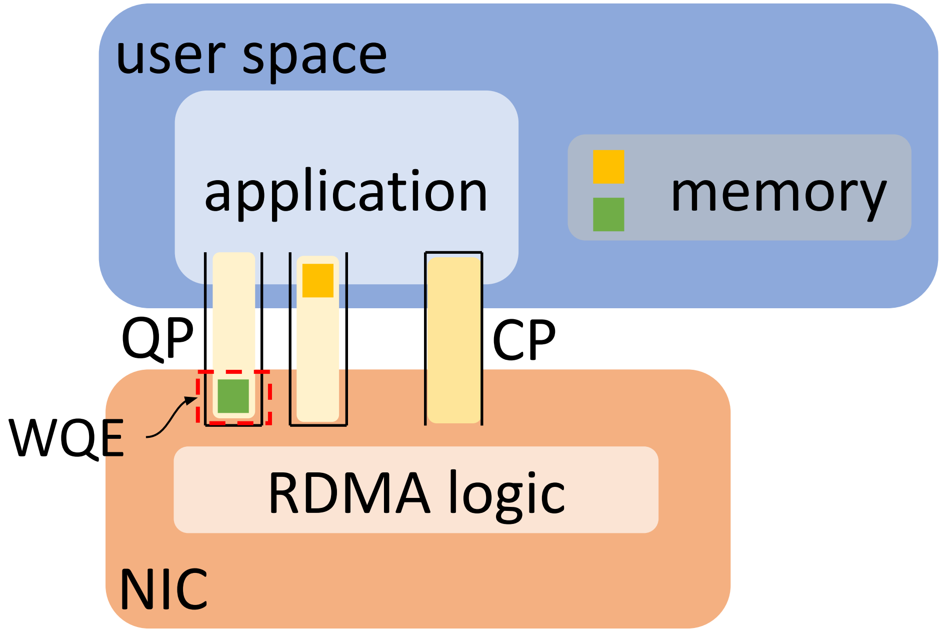
- A completion queue is used to **notify** when the work has been completed

Queue elements



- Applications issue (post) a job using a Work Request Element (WQE)
- A **WQE** is a small struct with a pointer to a buffer
  - In a *send queue*: it is the pointer to a message to be sent
  - In a *receive queue*: it shows where an incoming message shall be places
- Once a work request has been completed, the NIC creates a Completion Queue Element (**CQE**) and enqueues it in the *completion queue*

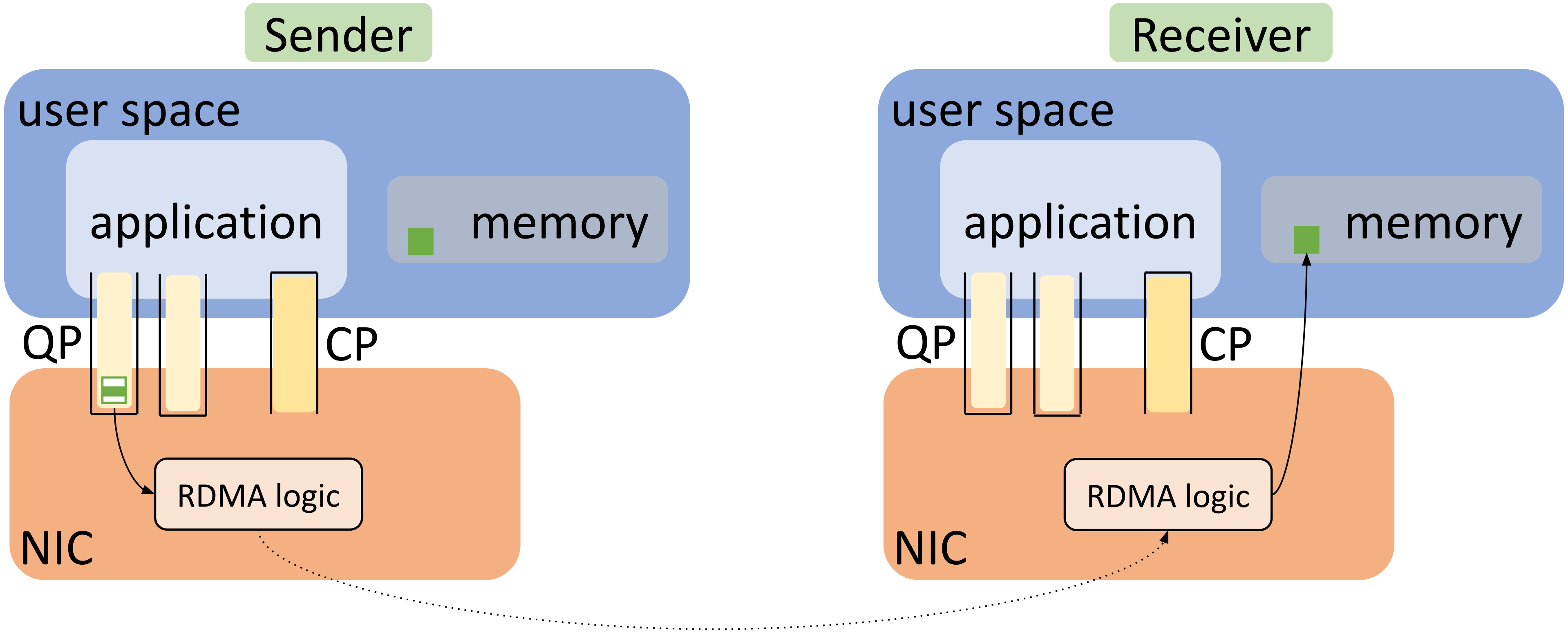
Queue elements



Between posting and completion the state of the memory involved shall not be touched! ☺️

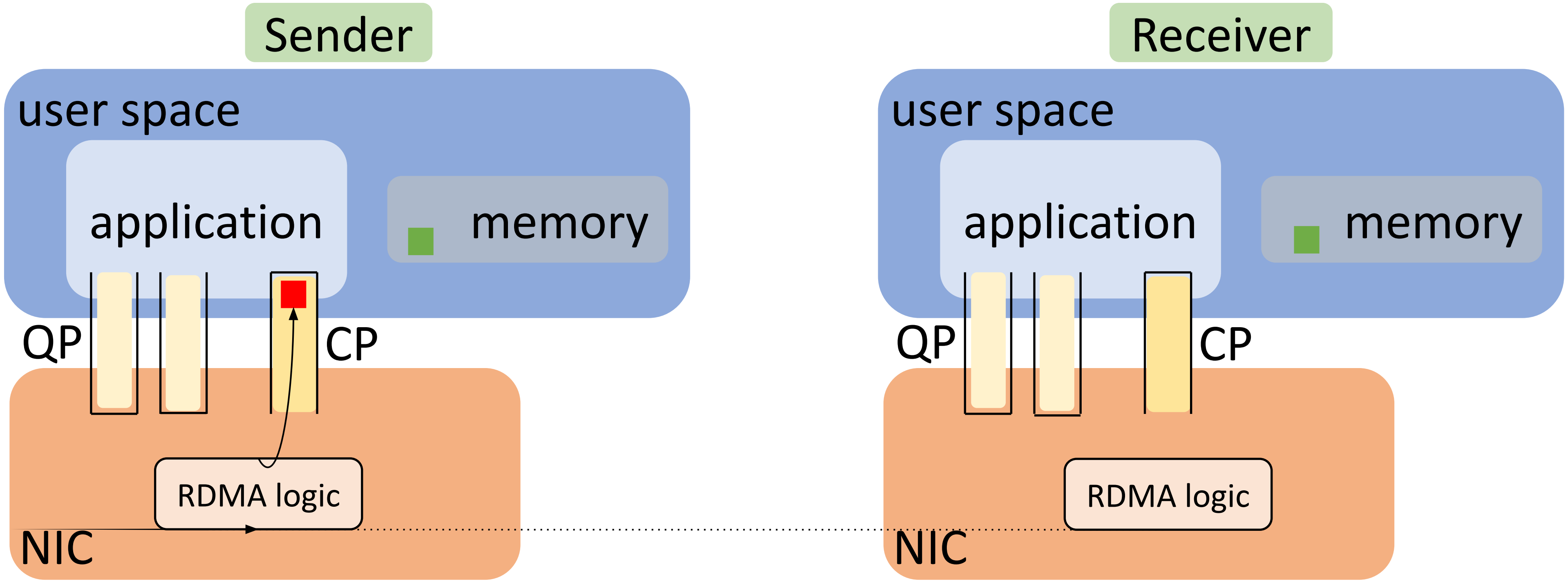
- Applications issue (post) a job using a Work Request Element (WQE)
- A **WQE** is a small struct with a pointer to a buffer
  - In a *send queue*: it is the pointer to a message to be sent
  - In a *receive queue*: it shows where an incoming message shall be places
- Once a work request has been completed, the NIC creates a Completion Queue Element (**CQE**) and enqueues it in the *completion queue*

RDMA Write



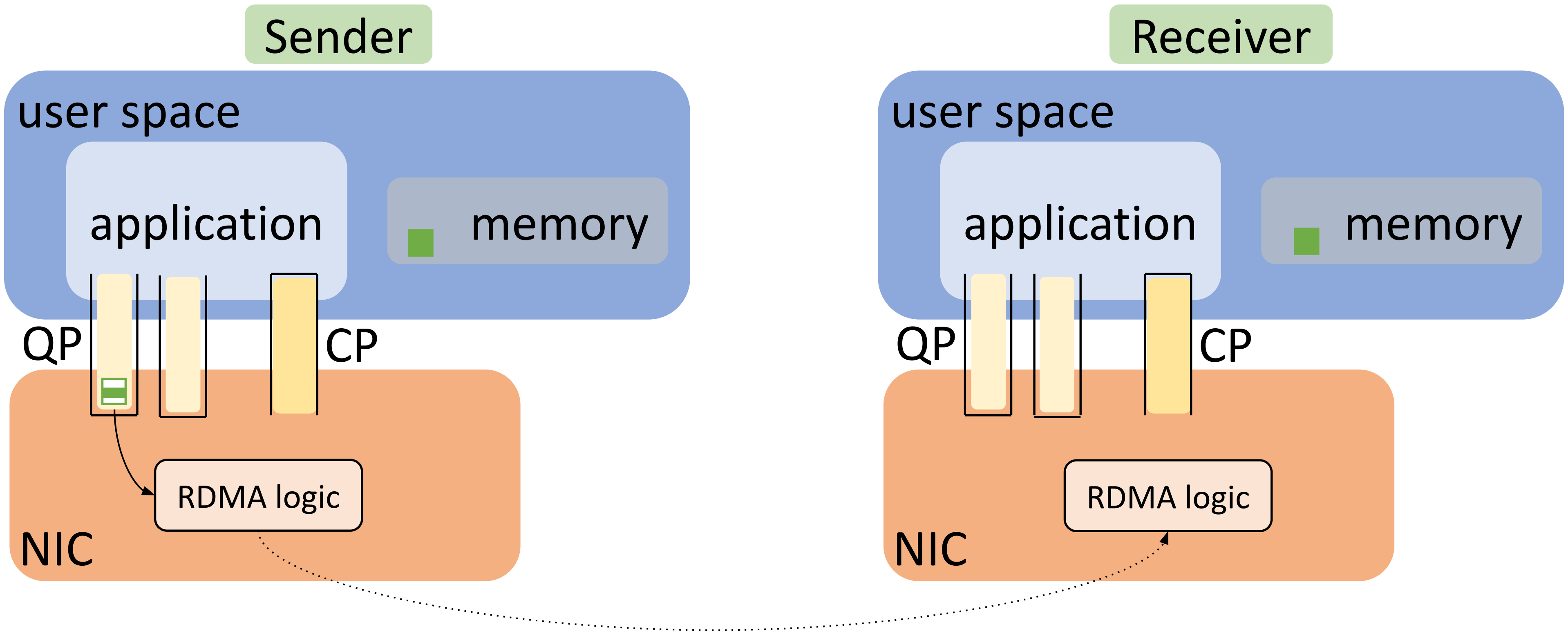
- Only the sender side is active, and the receiver is passive (no operations, no CPU cycles, no indication that a read or write has happened)
- The sender needs to specify the remote side's virtual memory address

RDMA Write



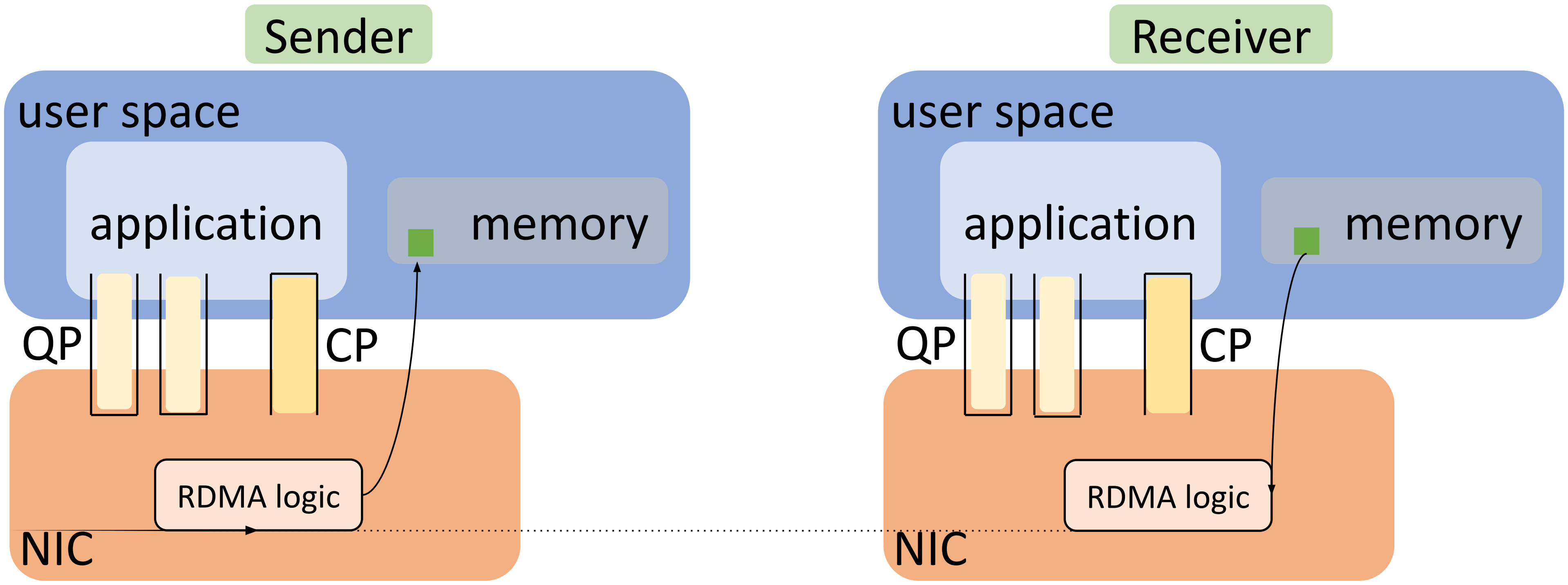
- Upon reception of an ACK, the NIC sender enqueue a completion message in the CP queue

RDMA Read



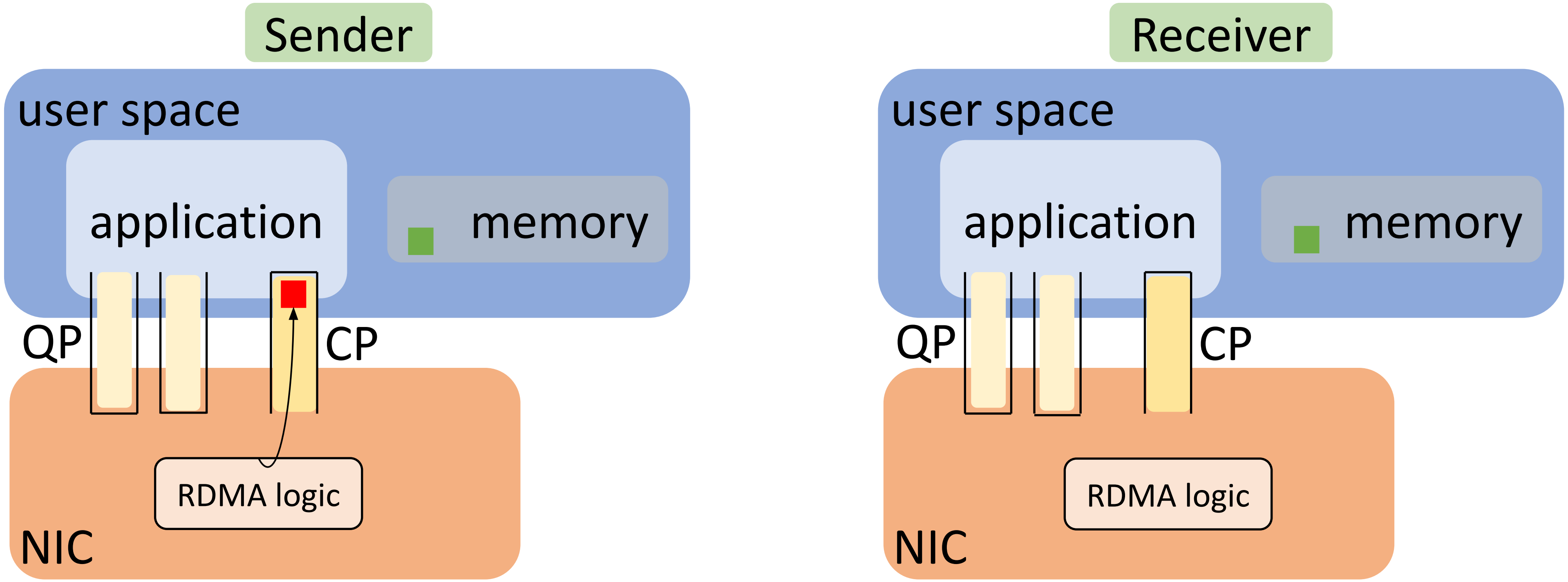
- Only the sender side is active, and the receiver is passive (no operations, no CPU cycles, no indication that a read or write has happened)
- The sender needs to specify the remote side's virtual memory address

RDMA Read



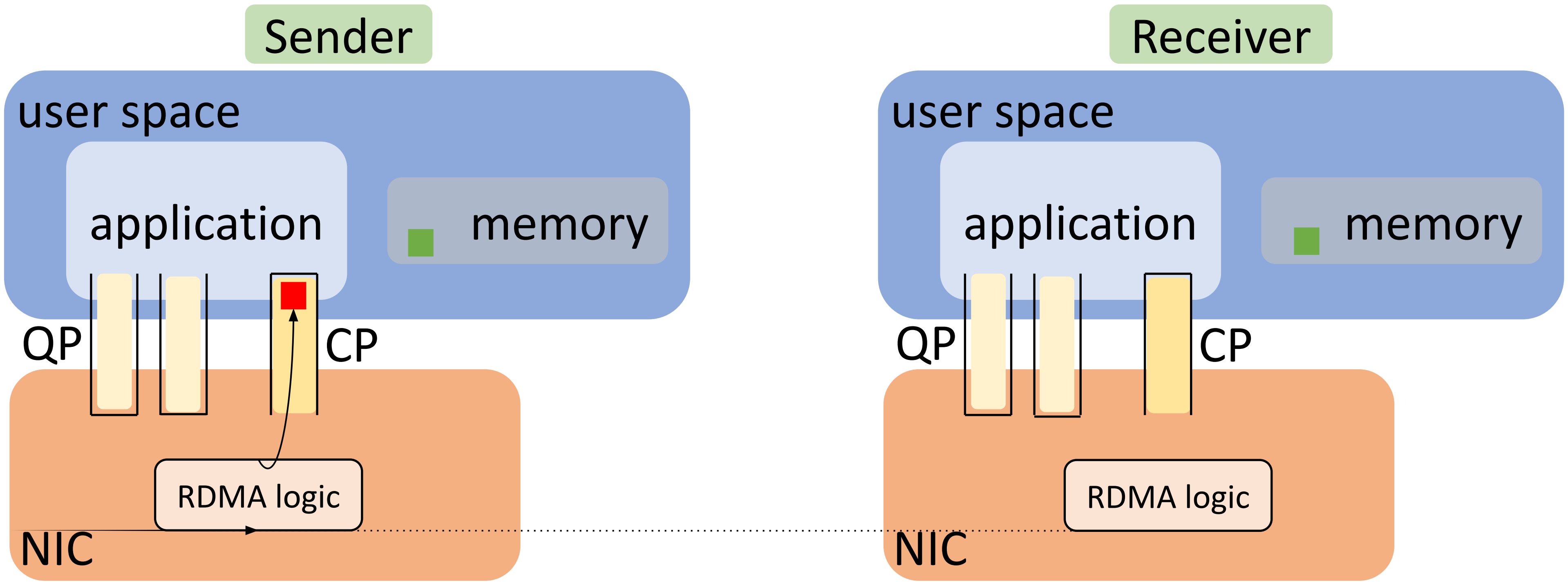
- Data are read from the remote memory and written in the initiator memory

RDMA Read



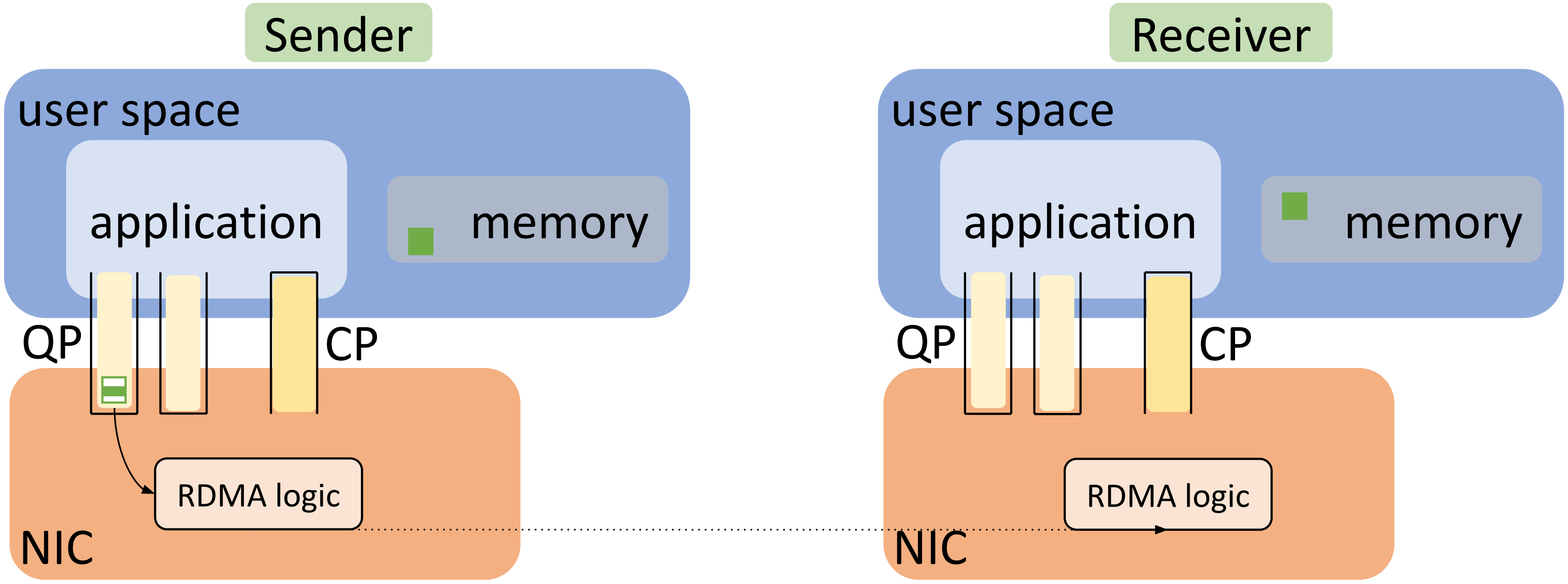
- Upon reception of an ACK, the NIC sender enqueue a completion message in the CP queue

RDMA Read



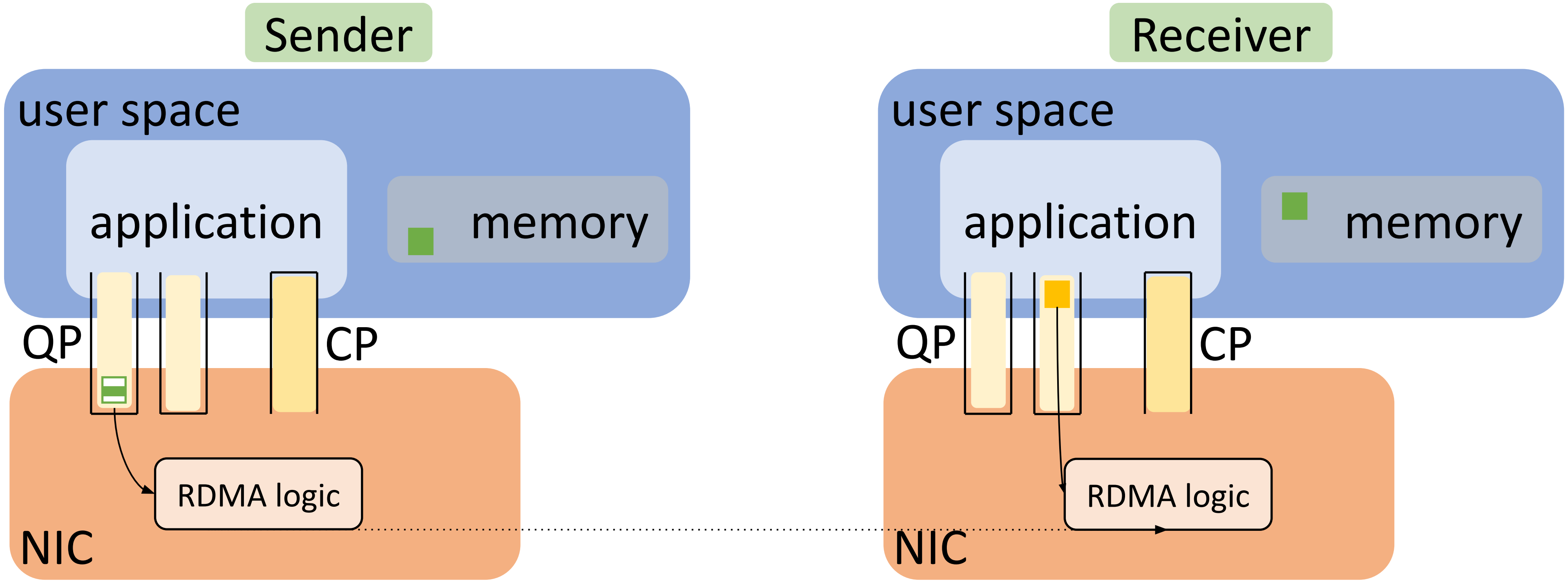
- Upon reception of an ACK, the NIC sender enqueue a completion message in the CP queue

RDMA Send/Receive



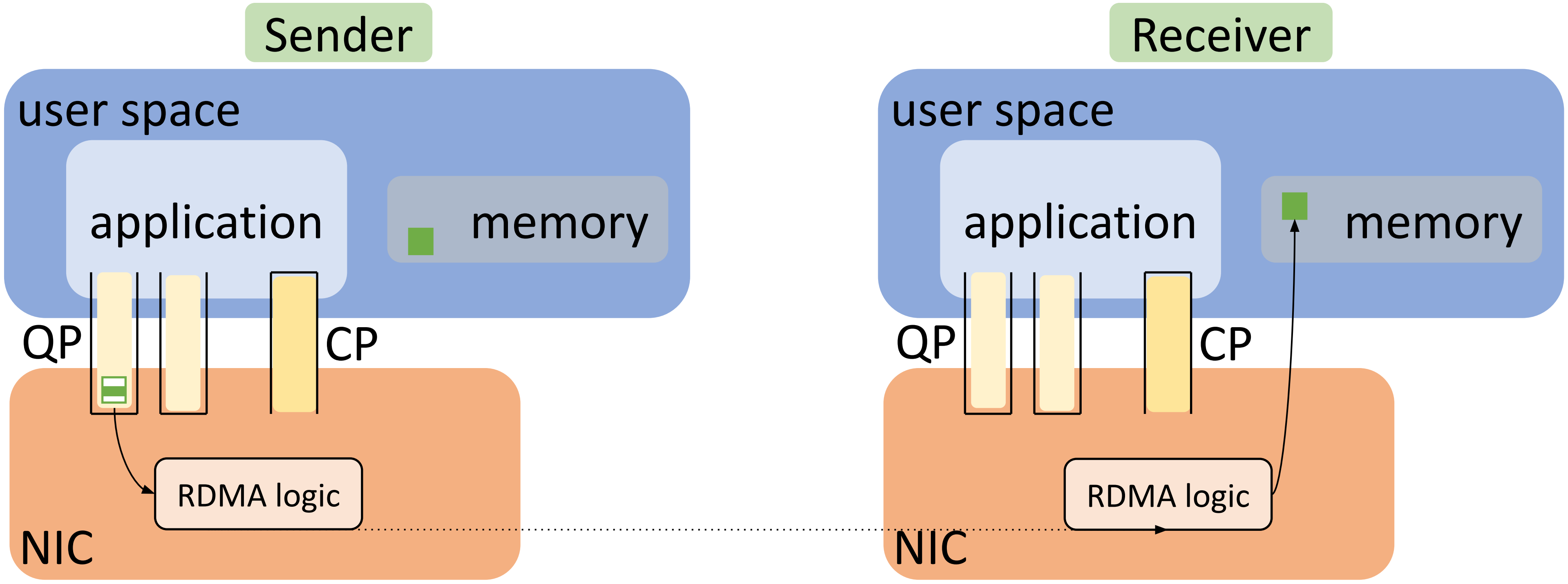
- Sender enqueues a new WQE in the send queue. In this request there is a pointer to a buffer in the memory

RDMA Send/Receive



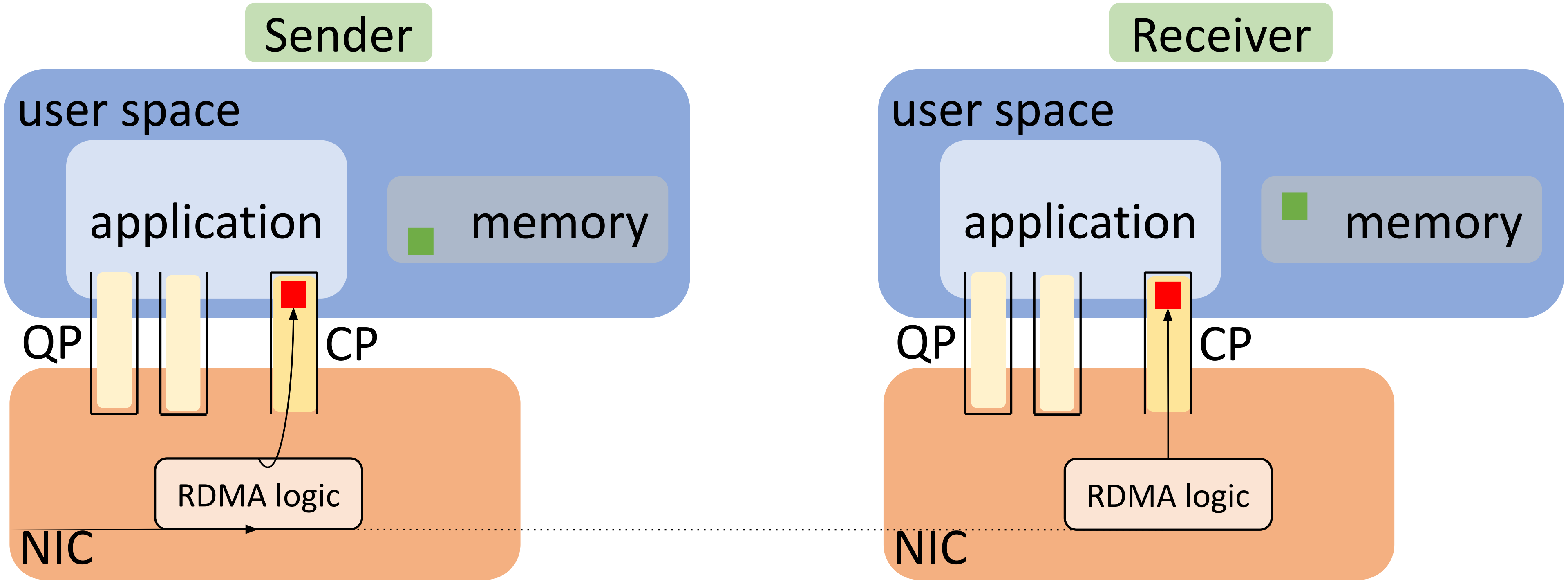
- Sender enqueues a new WQE in the send queue. In this request there is a pointer to a buffer in the memory
- Receiver's queue has a pointer to an empty buffer for receiving the message

RDMA Send/Receive



- The receiver NIC uses that pointer in the memory to know where to store the data

RDMA Send/Receive



- The NIC receiver then notifies the local application about the presence of new data in the CP queue and ack back the completion to the sender

Few final notes on RDMA

- With RDMA, NICs take control of the transport protocol
  - When the receiver receives an **out-of-order packet**, it simply discards it and sends a negative acknowledgement (NACK) to the sender
  - When the sender sees a NACK, it retransmits all packets that were sent after the last acknowledged packet (i.e., it performs a **go-back-N** retransmission)

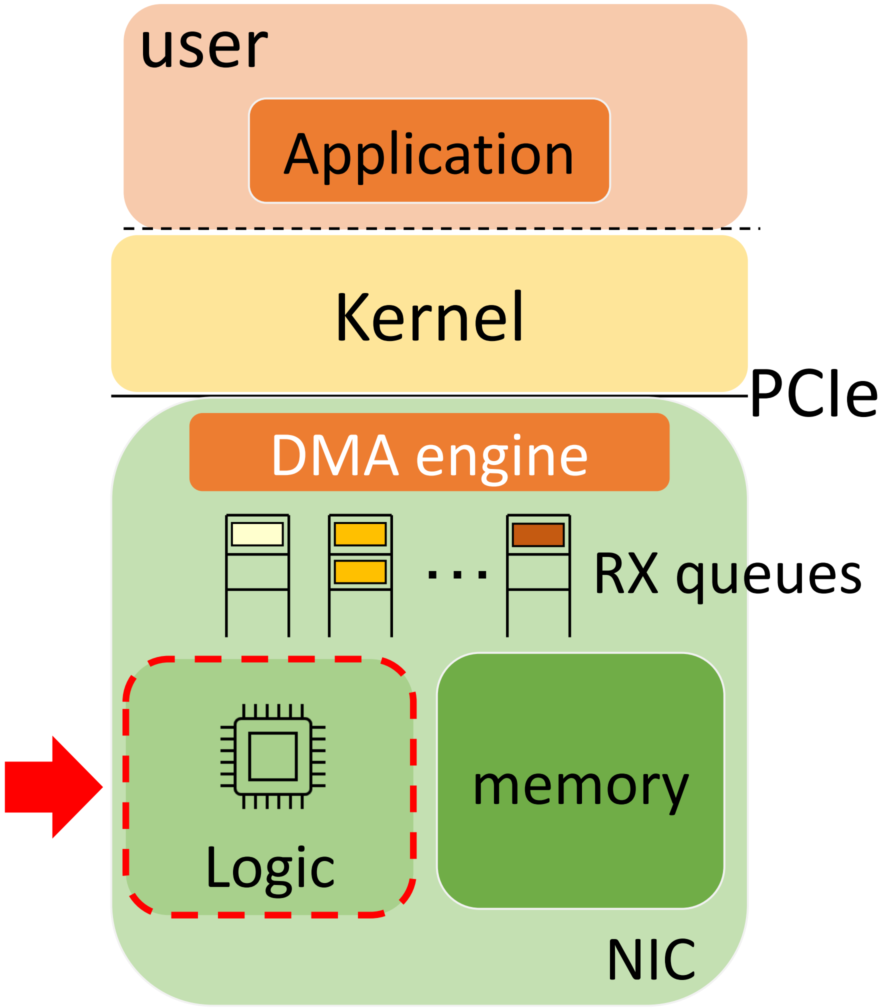
- There is an extensive area of research in designing congestion control algorithms for RDMA

- There is also the opportunity to have unreliable RDMA but in this case, normal NICs do not support all the verbs mentioned before

Option #3: SmartNIC

Option #3: offloading (more) compute on the NIC

- With RDMA we have seen a way to delegate transport-level processing to the NIC

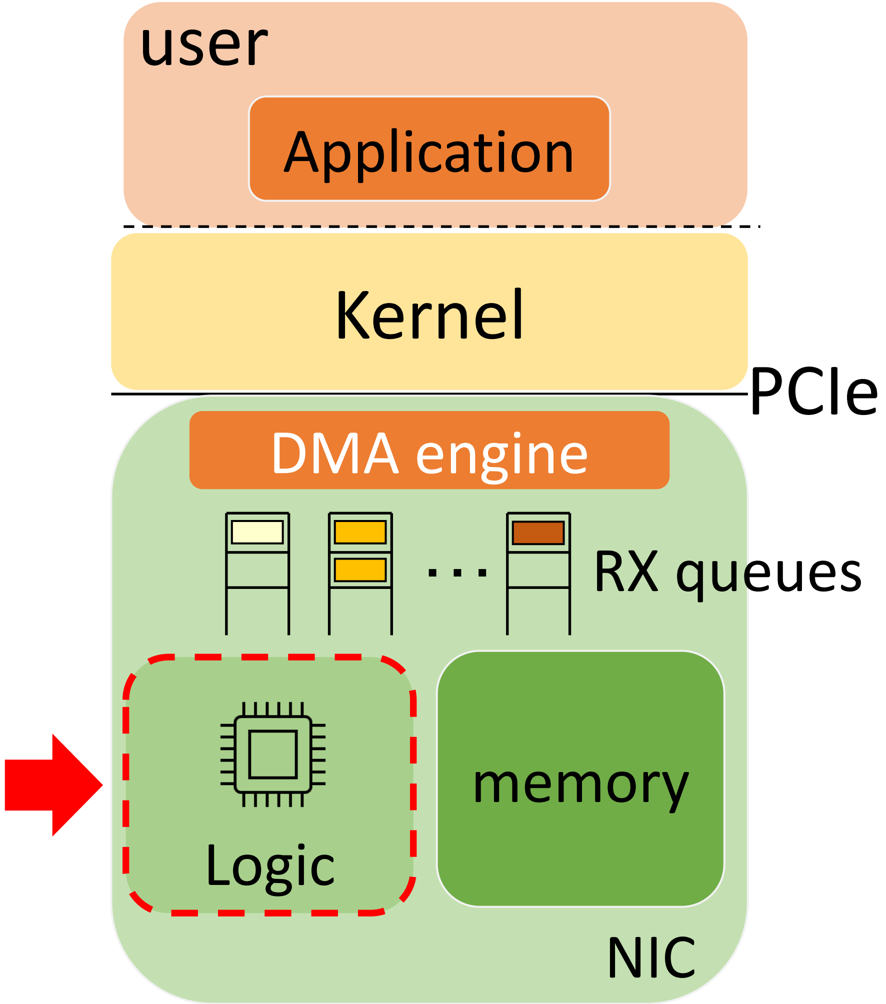


Option #3: offloading (more) compute on the NIC

- With RDMA we have seen a way to delegate transport-level processing to the NIC

What if we could do more?

What if this more can be achieved programmatically?



Option #3: offloading (more) compute on the NIC

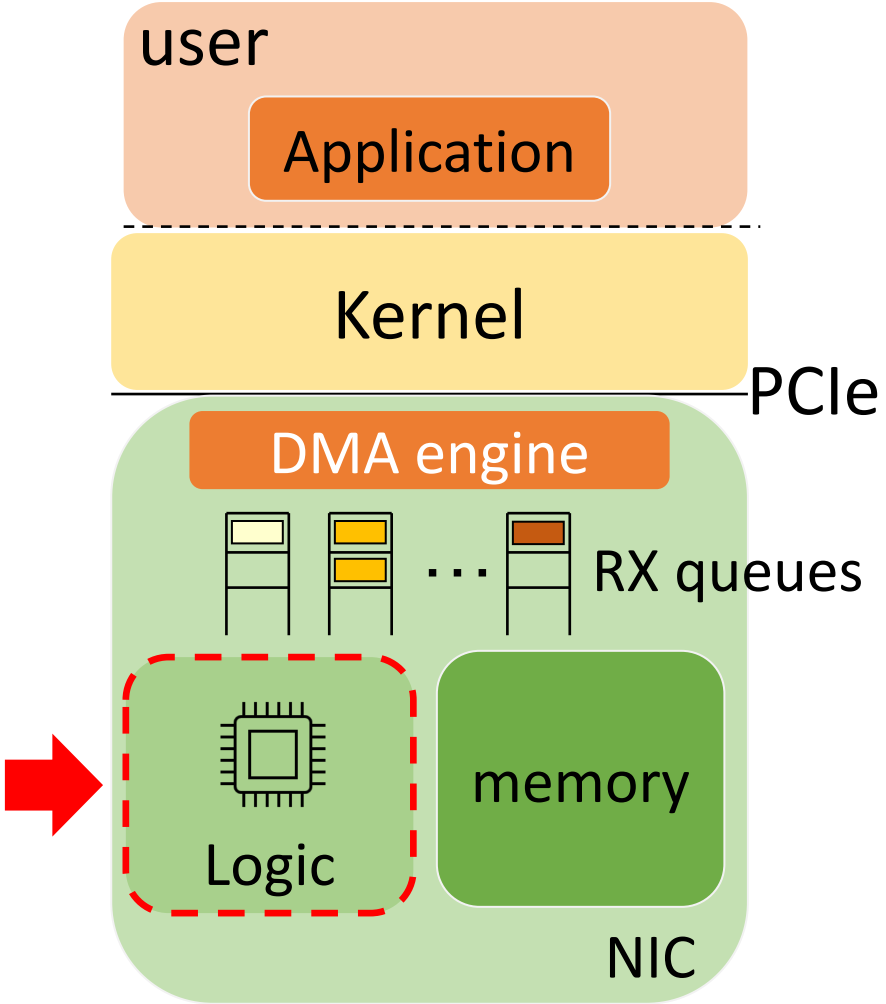
- With RDMA we have seen a way to delegate transport-level processing to the NIC

What if we could do more?

What if this more can be achieved programmatically?

- This is possible now with what are called Programmable NICs or SmartNICs

- Many manufacturers are building those devices



Programmable NICs

SMART ADAPTER CARDS
PRODUCT BRIEF

Mellanox TECHNOLOGIES

BlueField®

NVIDIA® Mellanox® BlueField® SmartNIC for Ethernet

High Performance Ethernet Network Adapter Cards

Combining Arm® processing power with advanced network and storage offloads to accelerate a multitude of security, networking and storage applications with world-leading performance, flexibility and efficiency.

BlueField SmartNIC features the BlueField Data Processing Unit (DPU) — an innovative and high-

HIGHLIGHTS

– Intelligent programmable network adapter

Intel® IPU Platform F2000X-PL 	• 2 x 100 GbE connectivity • Intel® Agilex-F FPGA • Intel® Xeon D-1736 Processor • 32GB DRAM	• Packet processing • OVS • NVMe-oF • Security/Isolation • Crypto • RDMA/RoCEV2

Pensando DSC-200 PCIe Card
(shown with heatsink removed)

AMD

Alveo™ U50 Data Center Accelerator Card

AMD-Xilinx's Alveo U50 data center accelerator card for custom solutions provide the framework for developers to bring differentiated applications to market

AMD-Xilinx's Alveo U50 data center accelerator cards provide acceleration for workloads in financial computing, machine learning, computational storage, data search, and analytics. Built on AMD-Xilinx UltraScale+ architecture and packaged in a 75 watt, small form factor and fitted with 100 Gbps networking I/O, PCIe® Gen4, and HBM, Alveo U50 is designed for deployment in any server.

Enabling Alveo accelerator cards is an ecosystem of AMD-Xilinx and partner applications for common data center workloads. For custom solutions, AMD-Xilinx's application developer tool suite (Vitis™ environment) and machine learning suite provide the frameworks for developers to bring differentiated applications to market.

Features

- Performance and efficiency
  - 8 GB HBM memory and PCIe for faster application performance
  - 100G networking with support for 4 x 10 GbE, 4 x 15 GbE, or 1 x 40 GbE/1 x 100 GbE for low latency network capability
- Adaptable for acceleration of any workload
  - Accelerates computing, network, and storage workloads
  - Maximized application performance as workloads and algorithms evolve through the reconfigurable fabric, unlike fixed-architecture alternatives
- Accessible on cloud or on-premises mobility
  - Built for scale-out architectures for deployment solutions on the cloud or on-premises interchangeably

HIGHLIGHTS

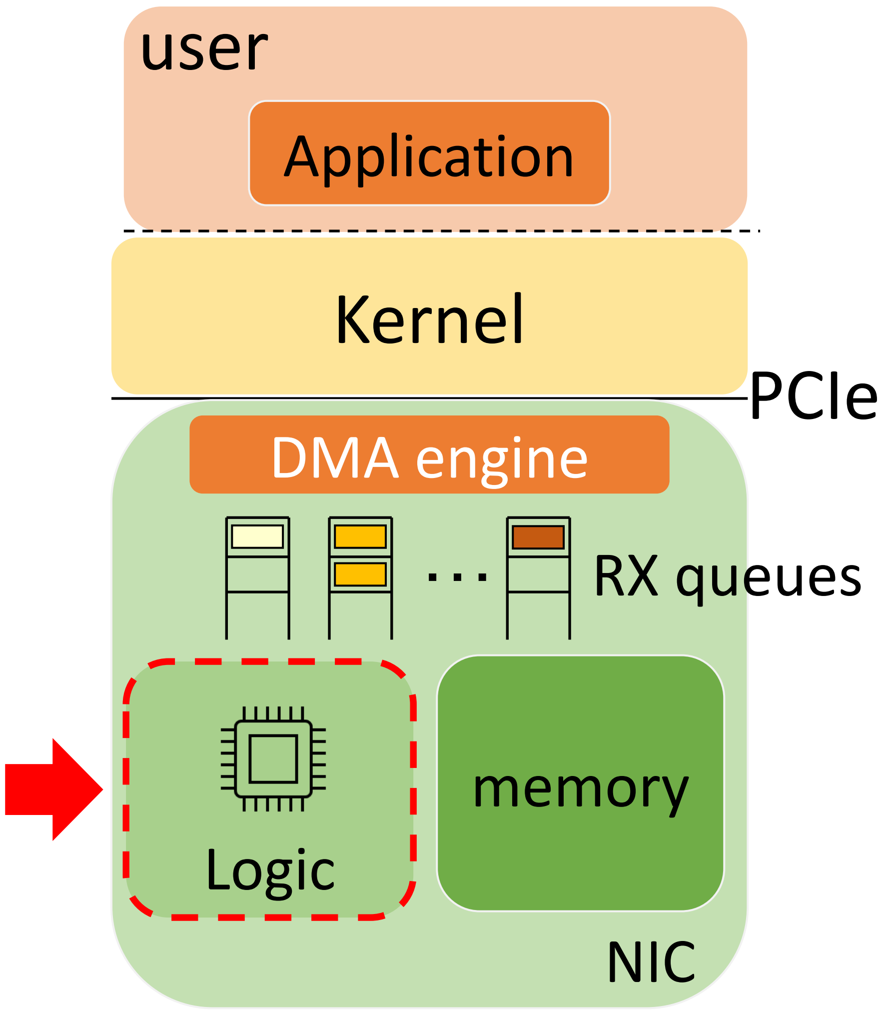
FEATURES

- Integrated security, networking, observability and storage services in a single card
- Incorporates both software-defined data plane and control plane, eliminating host agents
- Customizable control and data plane: customers can develop their own software, and integrate with existing management applications
- Pre-built services packages for a variety of functions, tested for scale
- Supports cloud-scale networks with 100k+ flow table entries and 1M+ routes

Programmable NICs

- There are many types of Programmable NICs with different
  - processing units (crypto, compression, CPU)
  - architectures (FPGA, SoC, ASIC)
  - programming models (Verilog/VHDL, C, P4, DOCA)
  - trade-offs (performance vs costs vs programmability)

- They can also be coupled with fast I/O frameworks such as DPDK

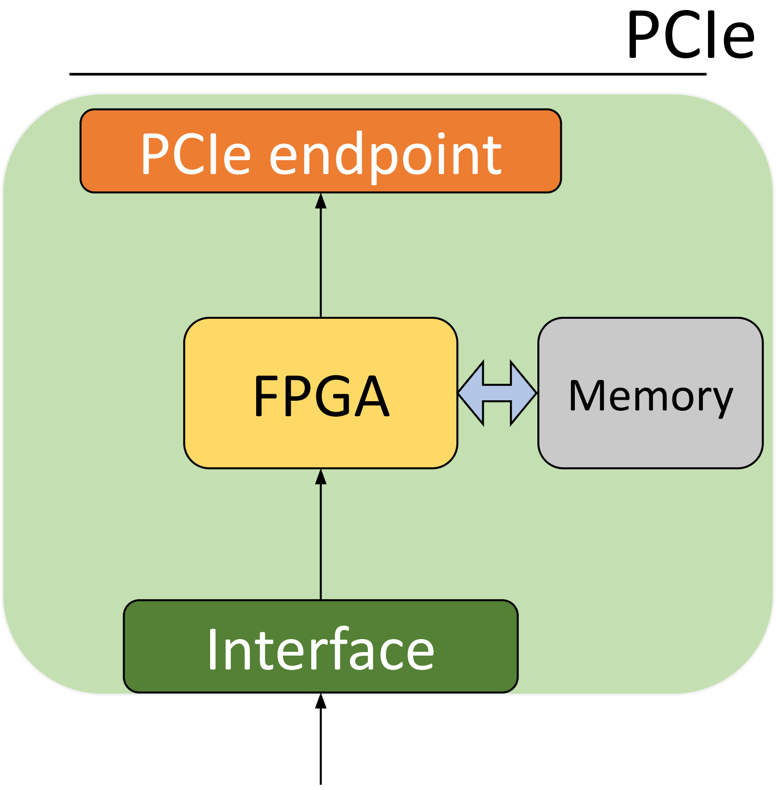


FPGA-based Programmable NICs

- Field Programmable Gate Arrays (FPGAs) are semiconductor devices can be reprogrammed to desired application or functionality requirements after manufacturing.

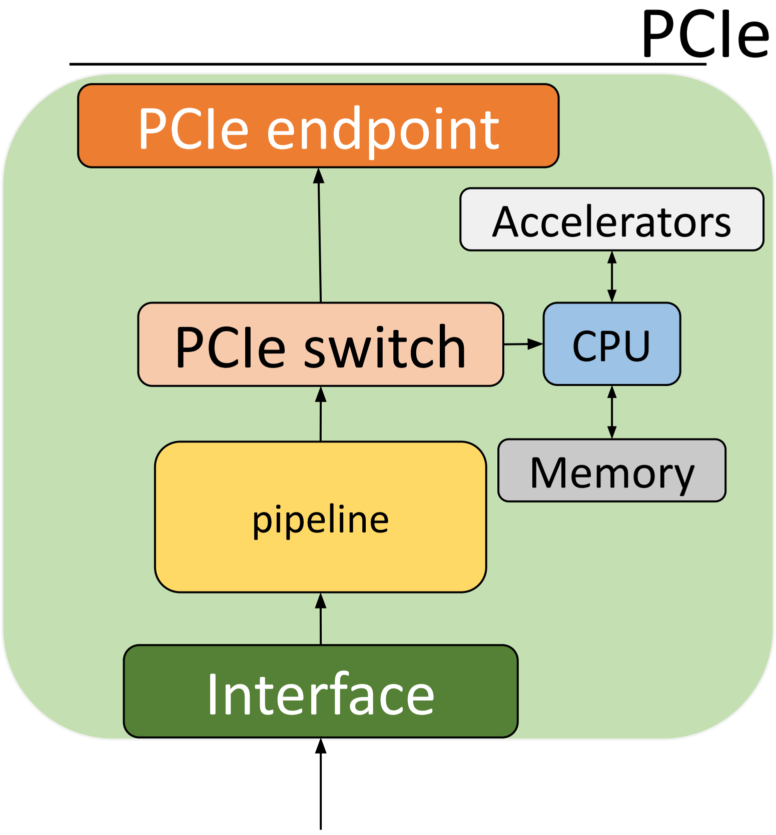
- FPGAs provide a flexible alternative with near-ASIC performance.

- They can be expensive and power-hungry, and programming them requires expert knowledge of the hardware.



SoC based Programmable NICs

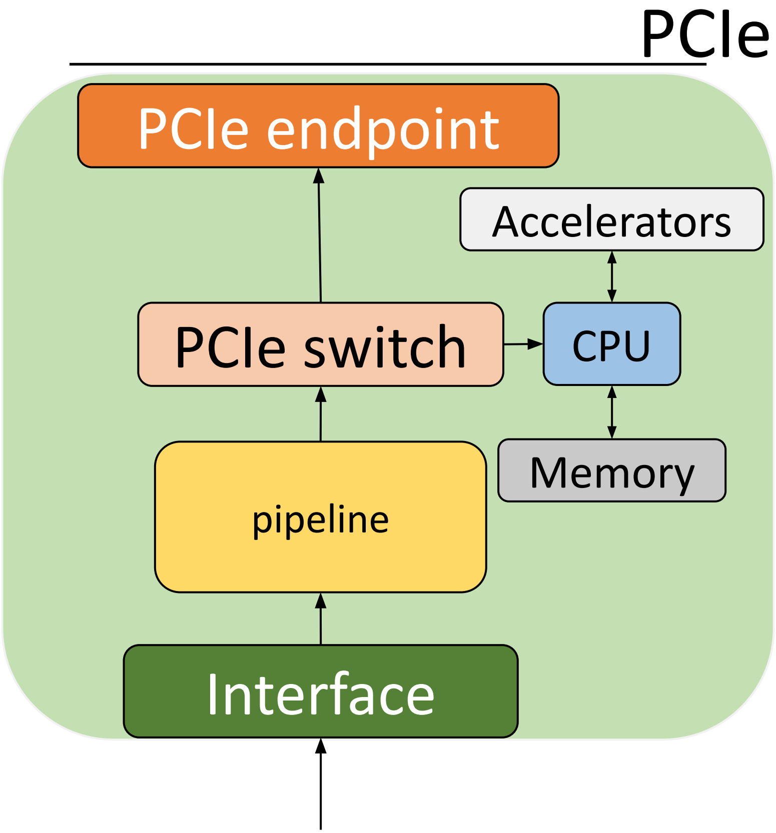
- System-on-a-chip (SoC) SmartNICs combines traditional ASICs with a modest number of general-purpose cores for much easier programming and fixed-function co-processors for custom workload acceleration.



SoC based Programmable NICs

- System-on-a-chip (SoC) SmartNICs combines traditional ASICs with a modest number of general-purpose cores for much easier programming and fixed-function co-processors for custom workload acceleration.

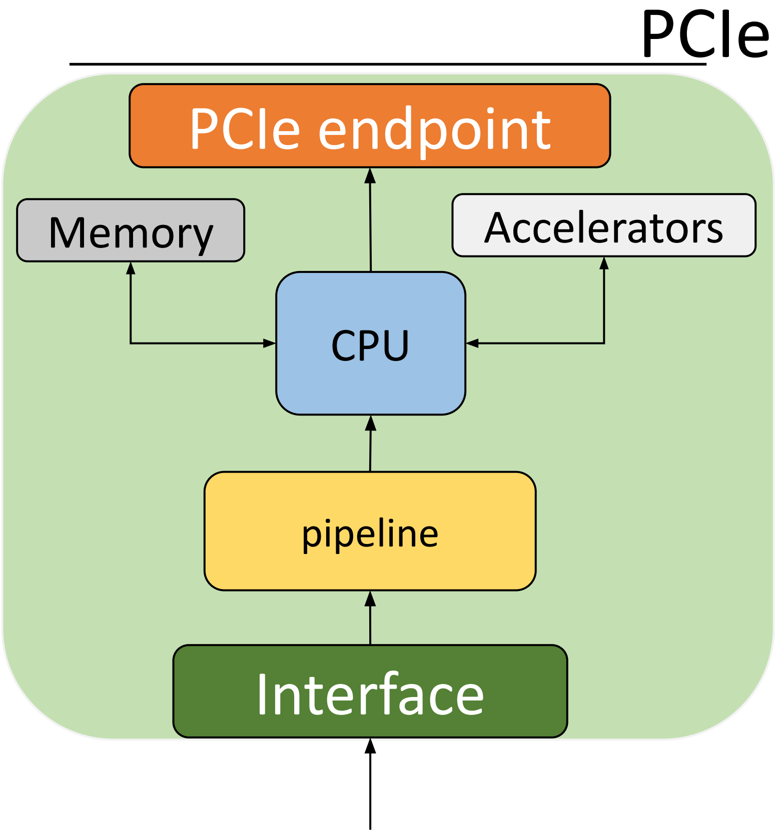
- Off-path design pattern uses an on-NIC switch to route traffic between the network and NIC and host cores



SoC based Programmable NICs

- System-on-a-chip (SoC) SmartNICs combines traditional ASICs with a modest number of general-purpose cores for much easier programming and fixed-function co-processors for custom workload acceleration.

- Off-path design pattern uses an on-NIC switch to route traffic between the network and NIC and host cores
- On-path approach passes all packets through (a subset of) cores on the NIC on the way to or from the network.

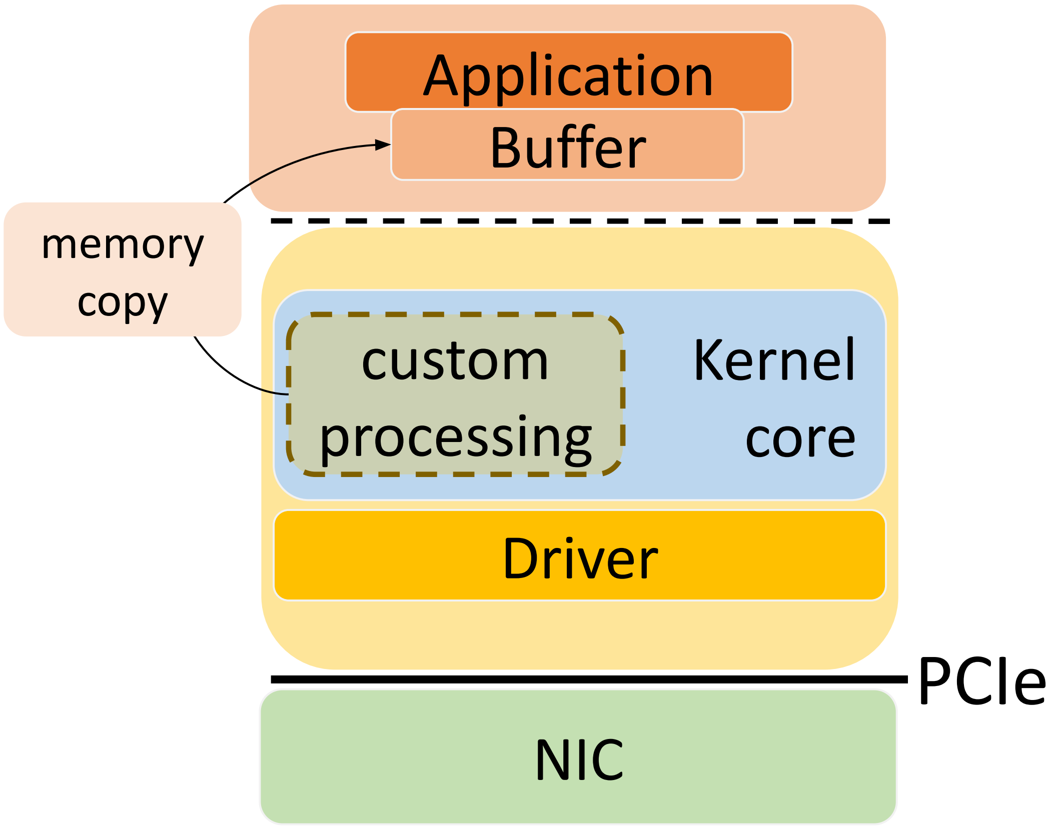


Option #4: eBPF

Option #4: Change kernel behavior

- You can improve system performance and reduce CPU utilization by rethinking the way kernel process packets

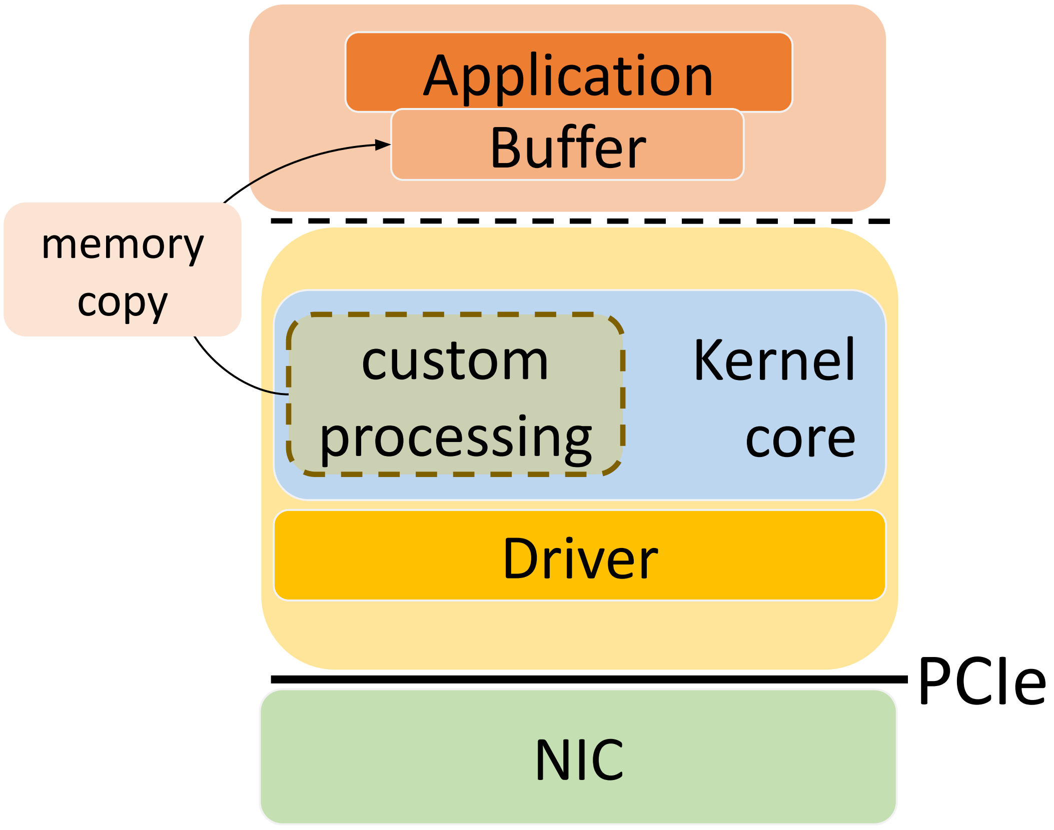
- You can add new functionalities that help in
  - reducing kernel to user-space data movement
  - creating fast-path in the kernel



Ad-Hoc Kernel module

- The Linux Kernel is open source ☺️

- You can change its code if you want

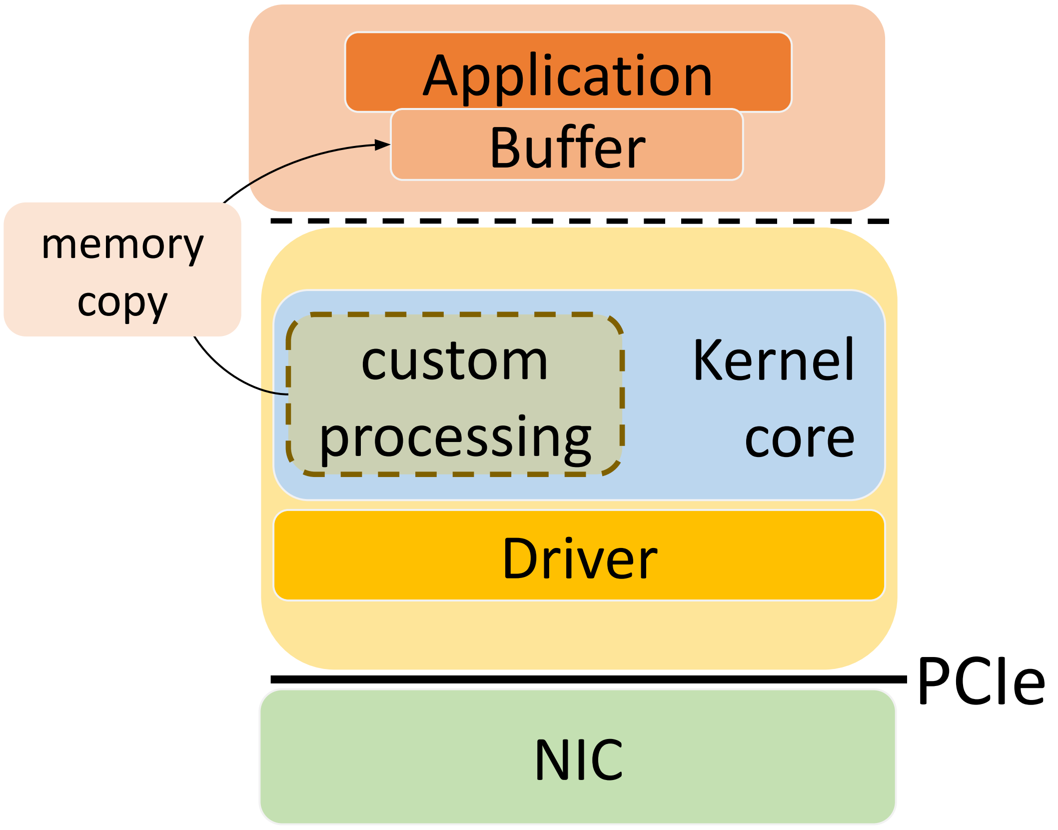


Ad-Hoc Kernel module

- The Linux Kernel is open source 😊

- You can change its code if you want

Problems?

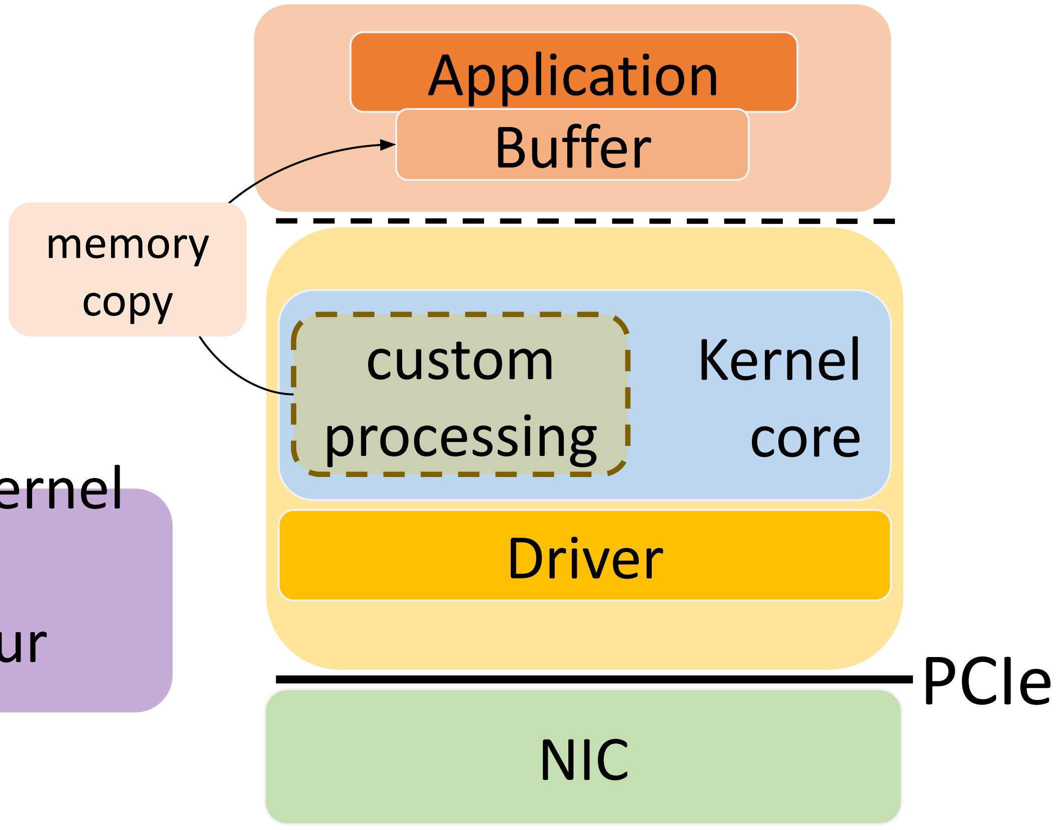


Ad-Hoc Kernel module

- The Linux Kernel is open source ☺️
- You can change its code if you want

Problems?

either you keep a custom Kernel or you need the Linux community to accept your changes ☹️



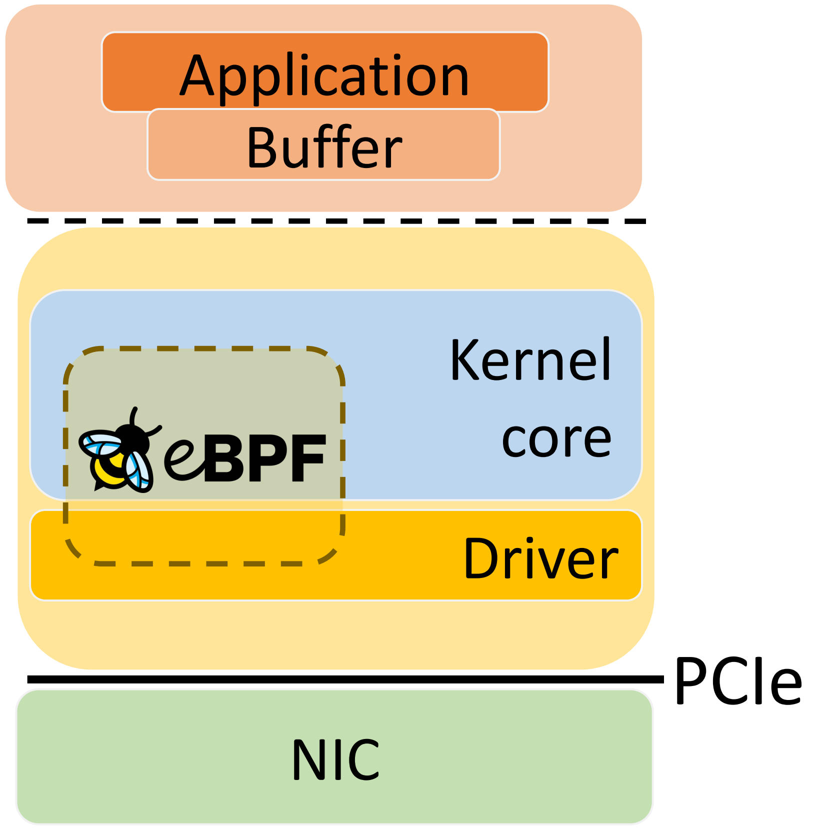
eBPF

- eBPF is a programming language and runtime to extend operating systems
- Was born as an extension of the Berkeley Packet filter:
  - a generic in-kernel, event-based virtual CPU
  - Ad-hoc execution environment for packet filtering
  - Specific memory for packets (separated from the main RAM)
  - Specific vCPU interpreter
- Now is used to provide kernel programmability for:
  - Kernel tracing
  - System observability: profiling, debugging, etc
  - Packet-processing
  - CPU scheduling

eBPF key features

- **<u>Feature 1:</u>** Runtime bytecode injection

- eBPF programs can be dynamically created and injected in the kernel at run-time
  - Vanilla Linux kernel
  - No need for additional kernel modules
  - No need to recompile the kernel

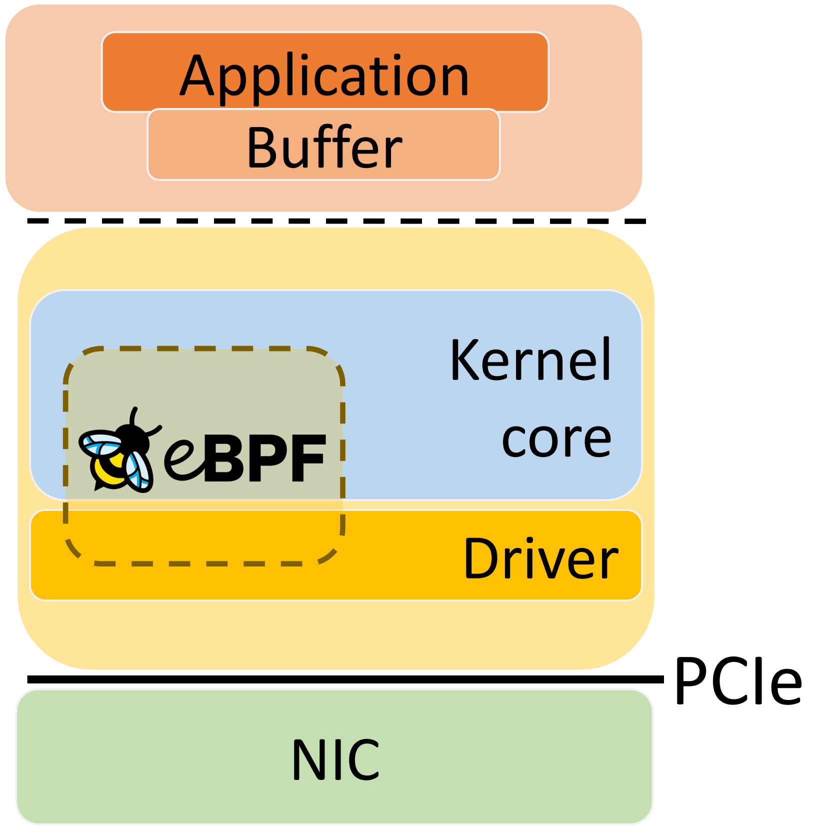


eBPF key features

- **<u>Feature 2:</u>** Safetiness

- Linux kernel must be protected from erroneous or malicious injected programs
  - Achieved with a **sandbox** that prevents critical conditions at **run-time**

- A verifier checks the code before it gets sandboxed to ensure
  - No invalid memory accesses
  - Bounded program size
  - Bounded max number of instructions

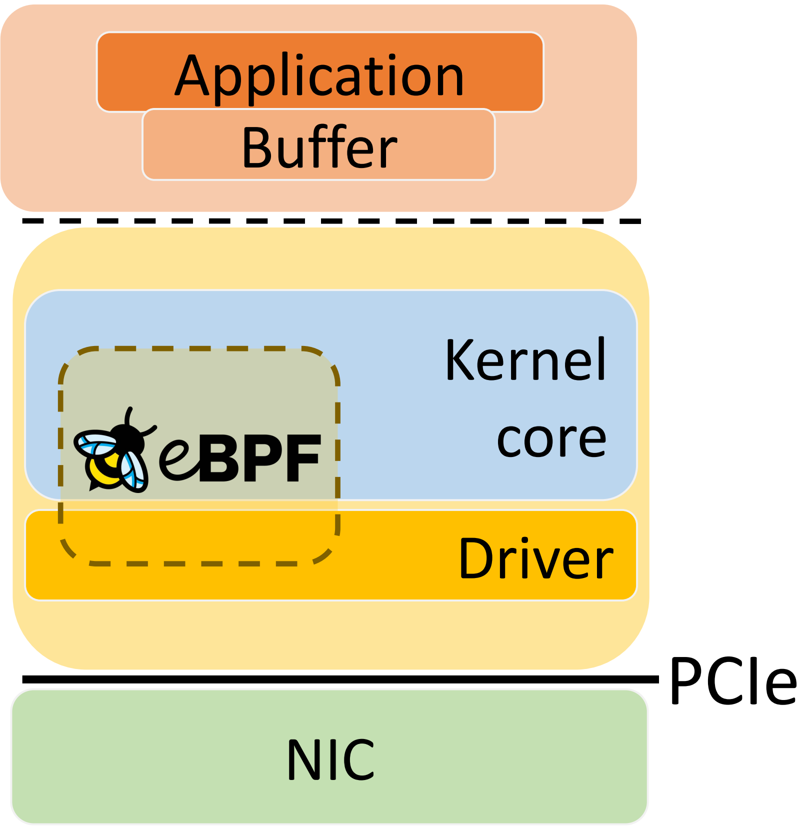


eBPF key features

- **<u>Feature 2:</u>** Safetiness

- Linux kernel must be protected from erroneous or malicious injected programs
  - Achieved with a sandbox that prevents critical conditions at run-time

- A verifier checks the code before it gets sandboxed to ensure
  - No invalid memory accesses
  - Bounded program size
  - Bounded max number of instructions



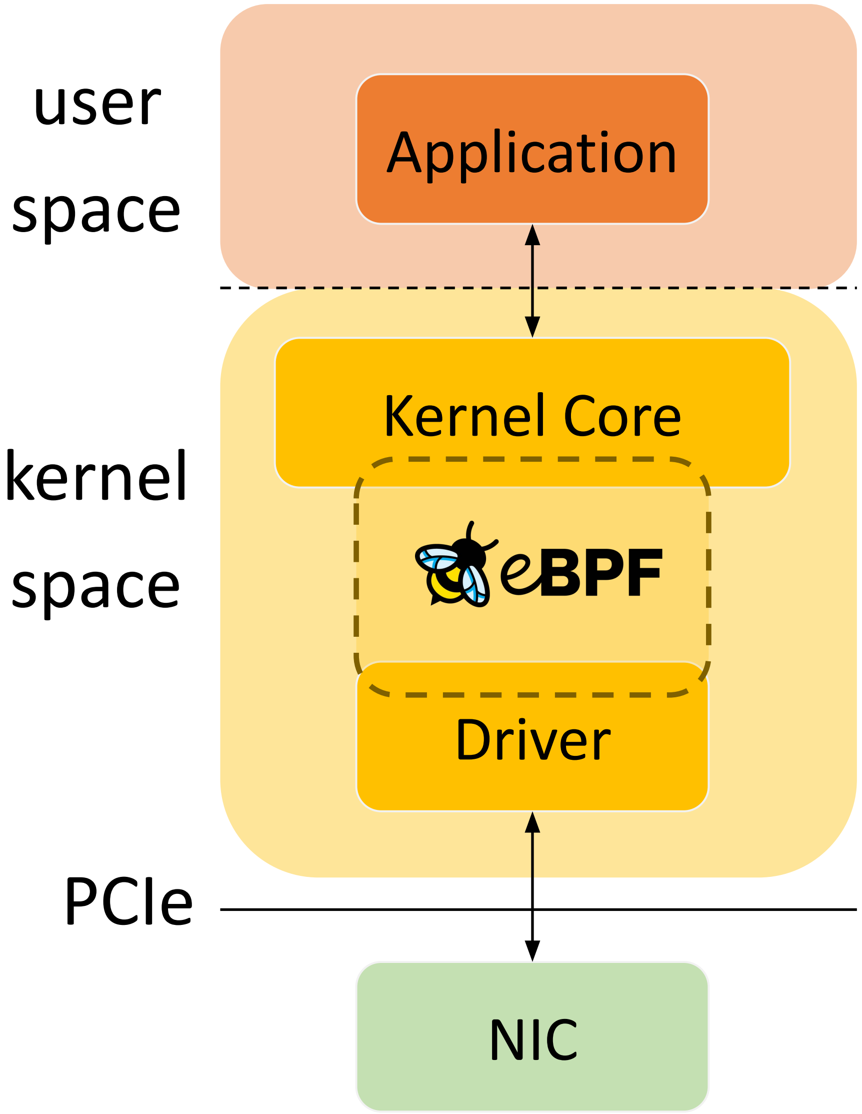
<u>Consequence</u>: we cannot push arbitrary programs in the kernel!

eBPF features

- **Feature 3:** Efficiency

- eBPF programs consumes a little amount of resources

- They executes in kernel space, potentially close to when packets are received (no need to copy packets as when we move them from kernel to user)

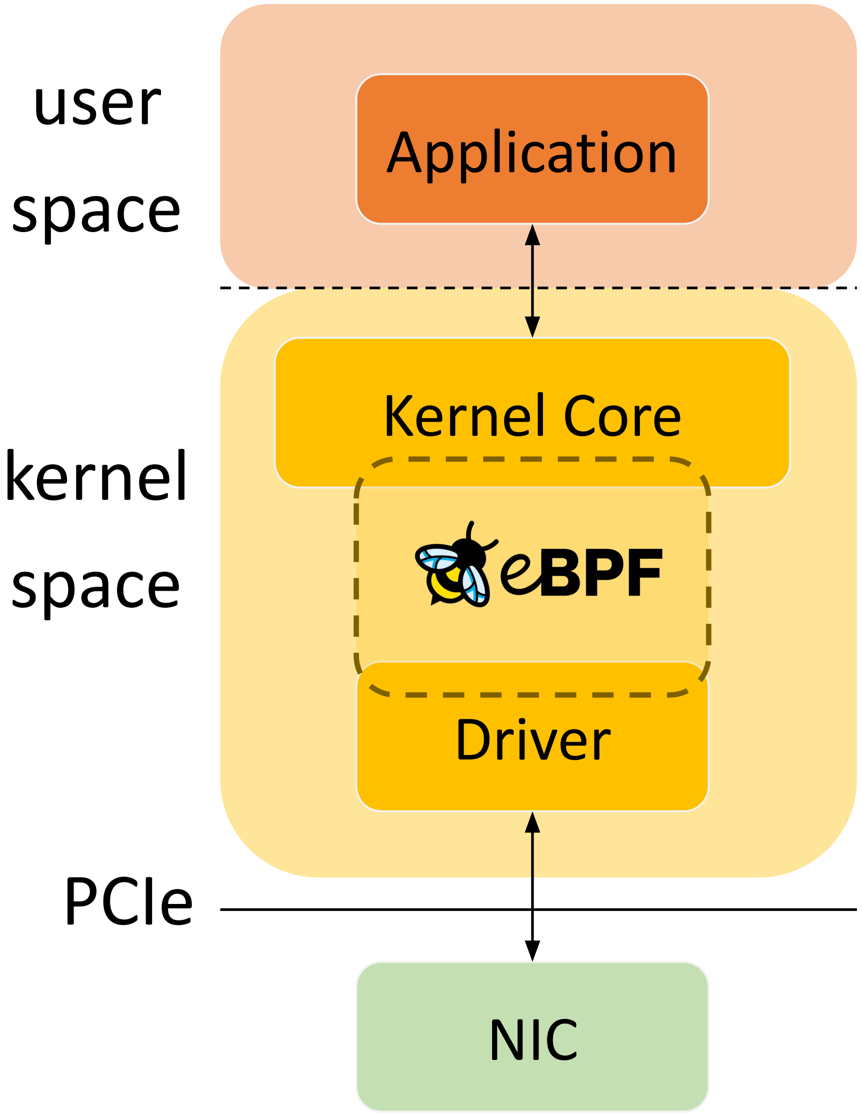


eBPF features

- **<u>Feature 4:</u>** Kernel events reaction

- eBPF code is hooked to a kernel event
  - When fired, your code (associated to an event handler) is executed

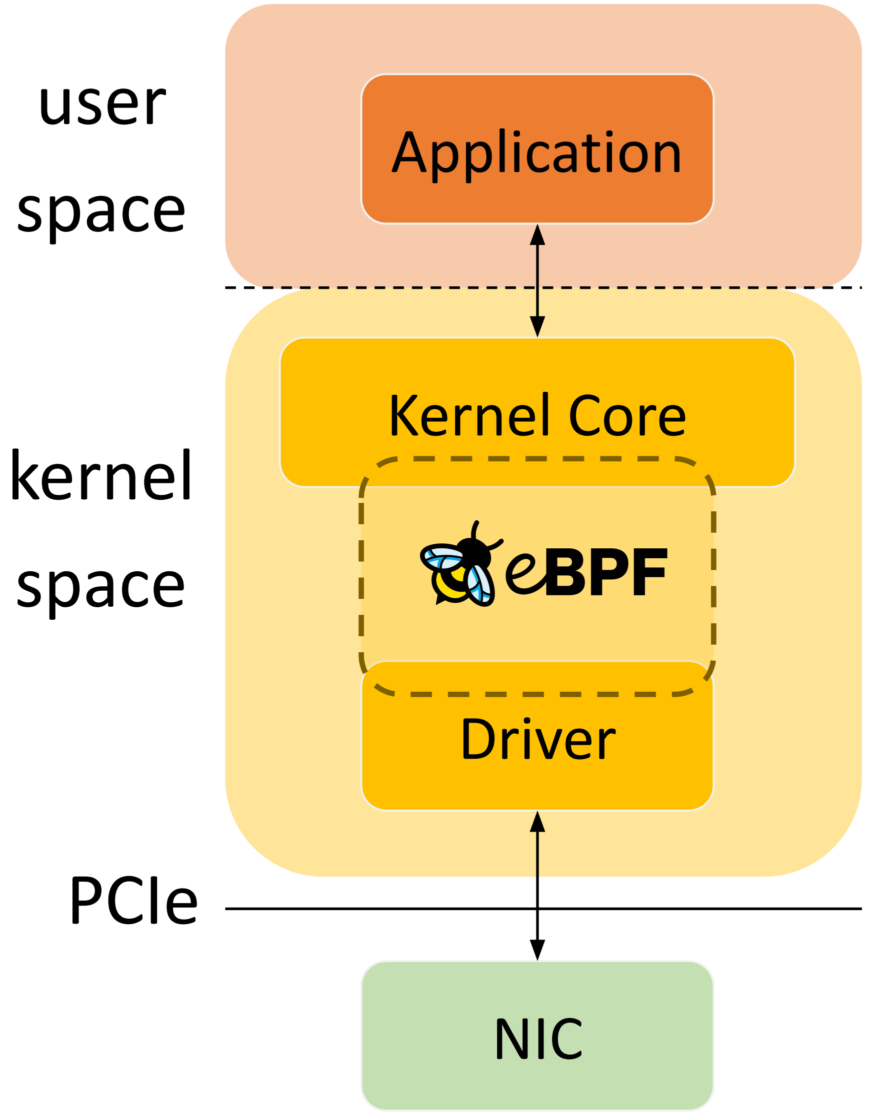
- Examples of possible events:
  - Network packet received
  - Message (socket-layer) received
  - Data written to disk
  - Page fault in memory
  - File in /etc folder being modified



eBPF features

- In general, any event can be potentially intercepted

- Depending on the event where the program is attached to, the context change



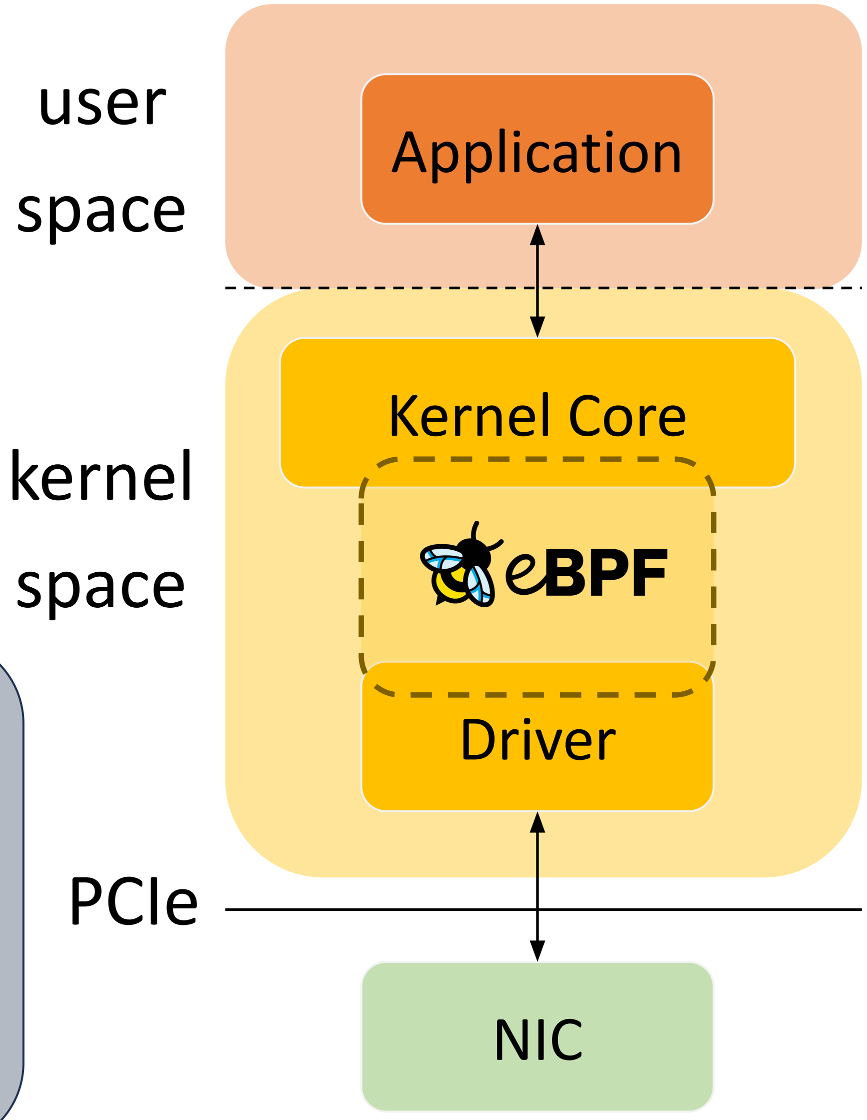
<u>Example 1</u>: a program attached to the TCP/IP stack is executed when a packet is received, and the context is the packet buffer

eBPF features

- In general, any event can be potentially intercepted
- Depending on the event where the program is attached to, the context change

Example 1: a program attached to the TCP/IP stack is executed when a packet is received, and the context is the packet buffer

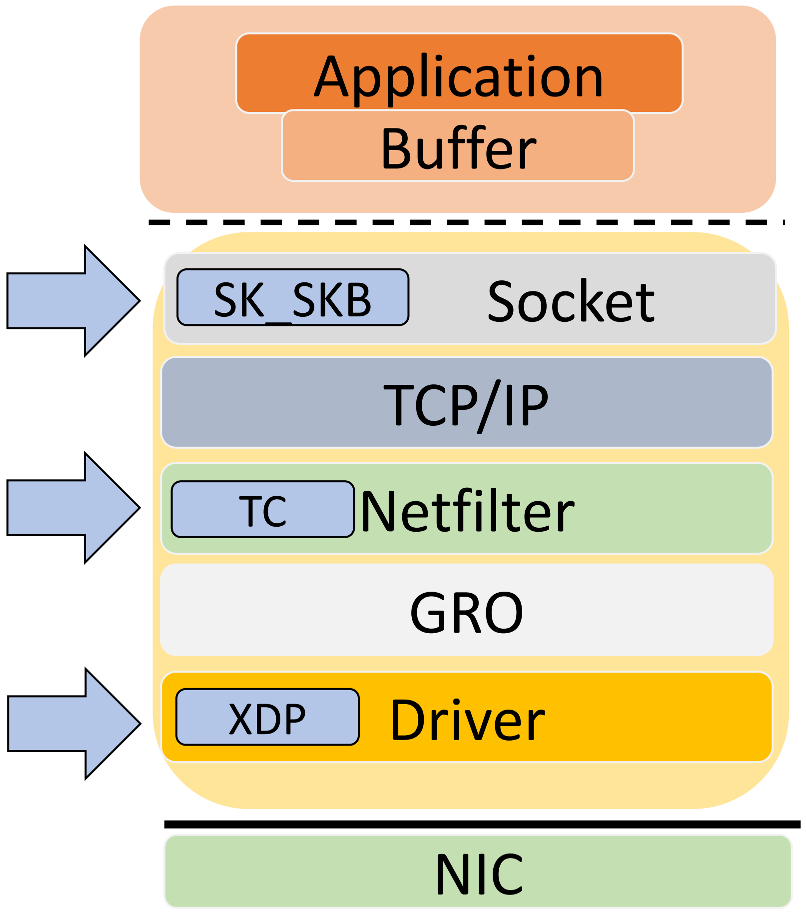
Example 2: a program attached to a syscall can be executed before or after the syscall execution and the context is the list of parameters of the syscall



Hook points

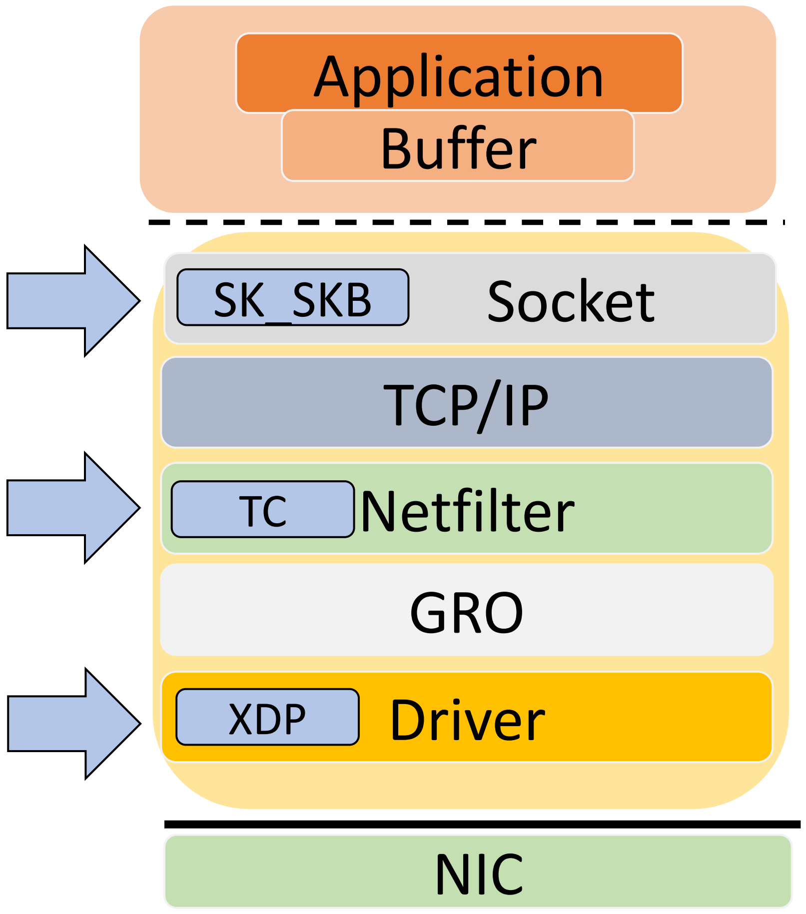
- Several hook points (kernel events) for networking
  - Located at different levels of the stack
  - Opens up the possibility to implement packet-processing programs at different layers of the stack

- Some of interest:
  - eXpress Data Path (XDP)
  - Traffic Control (TC)
  - Socket SKB (SK_SKB)
  - There are more…



Hook points

- The eBPF program returns a verdict to tell the OS how to process the packet next
  - **Redirect** (redirect the packet to another net device of index ifindex)
  - **Pass** (continue the path of the packet in the stack)
  - **Drop** (drop the packet)

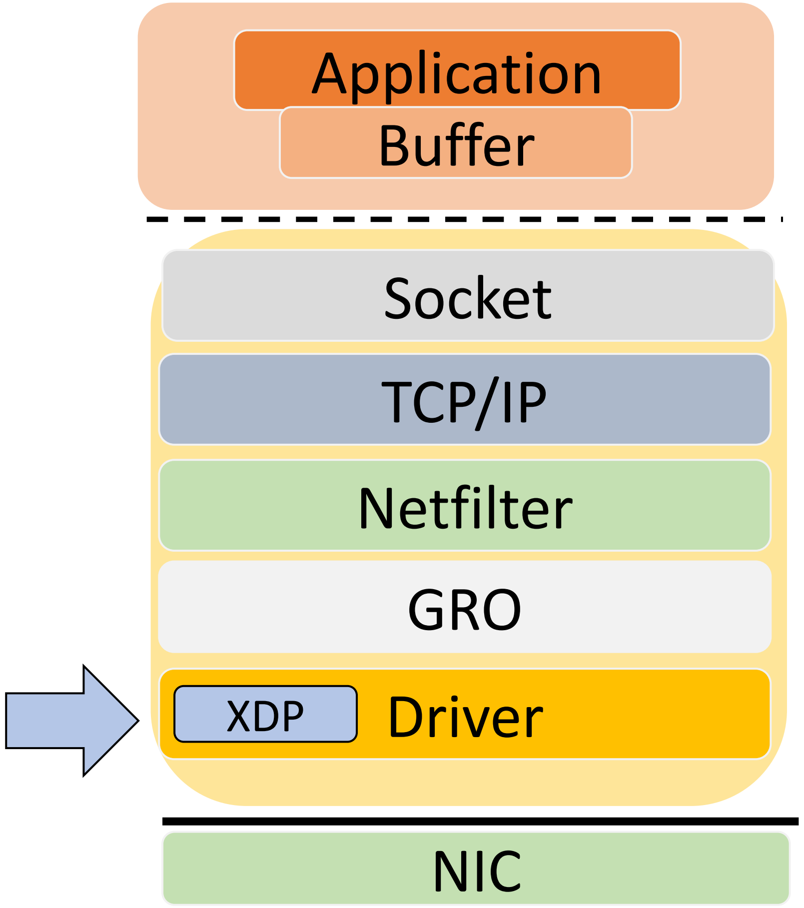


XDP Hook

- XDP allows to run eBPF programs at the driver level, before skb allocation

- This is the earliest hook-point, suitable for high-performance processing

why?



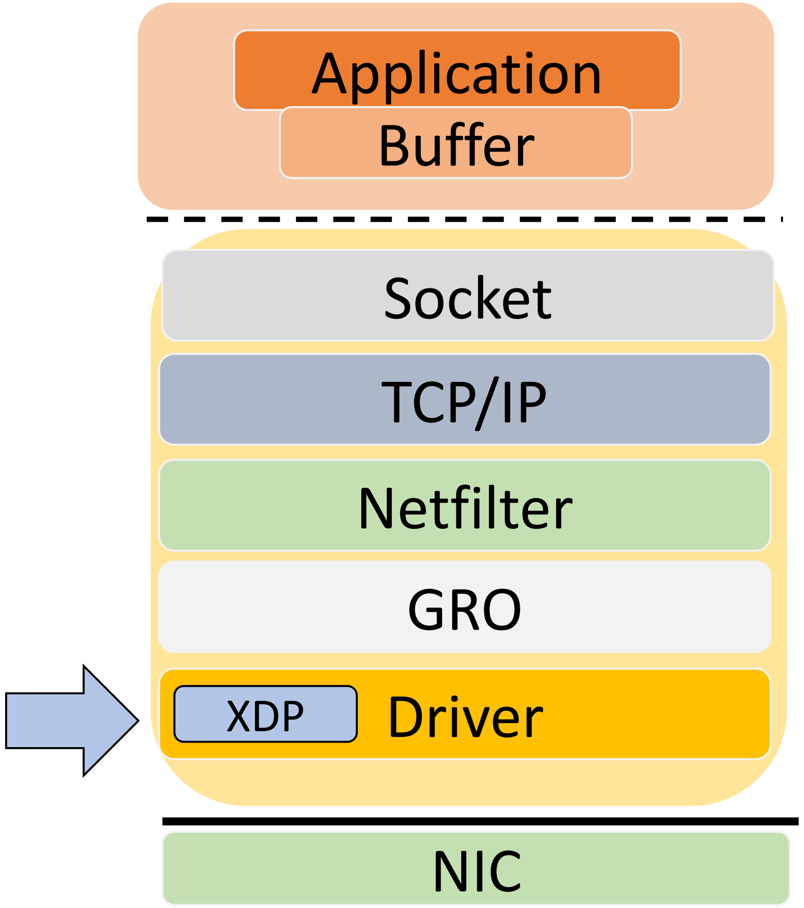
XDP Hook

- XDP allows to run eBPF programs at the driver level, before skb allocation

- This is the earliest hook-point, suitable for high-performance processing

why?

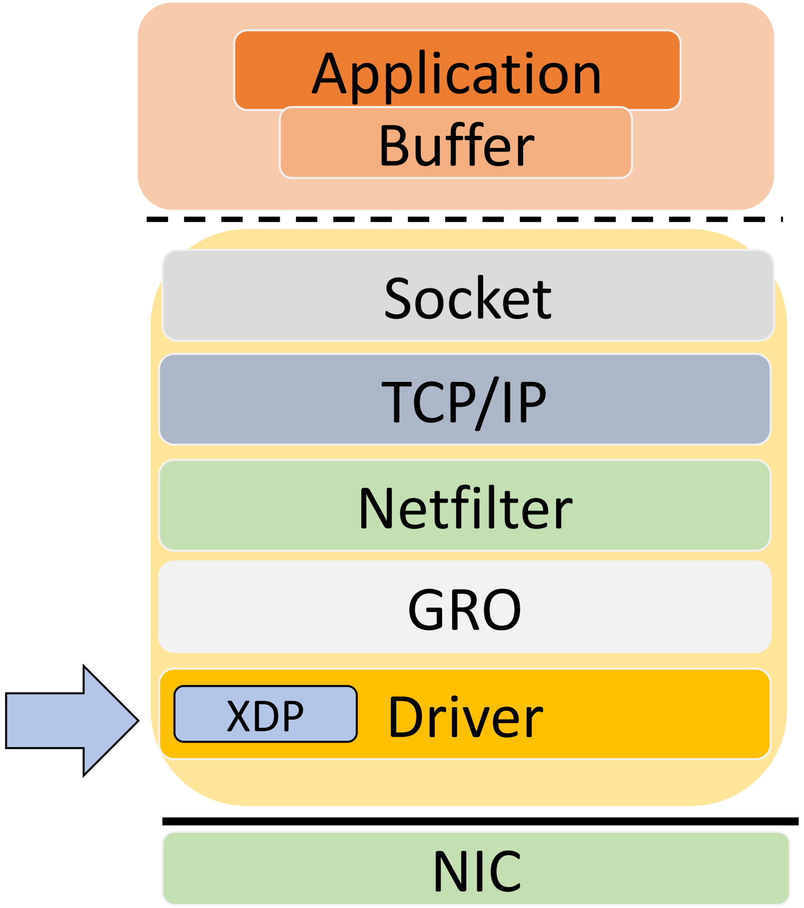
skb allocation is expensive and kernel processing too



XDP Hook

- XDP allows to run eBPF programs at the driver level, before skb allocation
- This is the earliest hook-point, suitable for high-performance processing
- XDP is good for offloading stateless protocols like UDP

why?

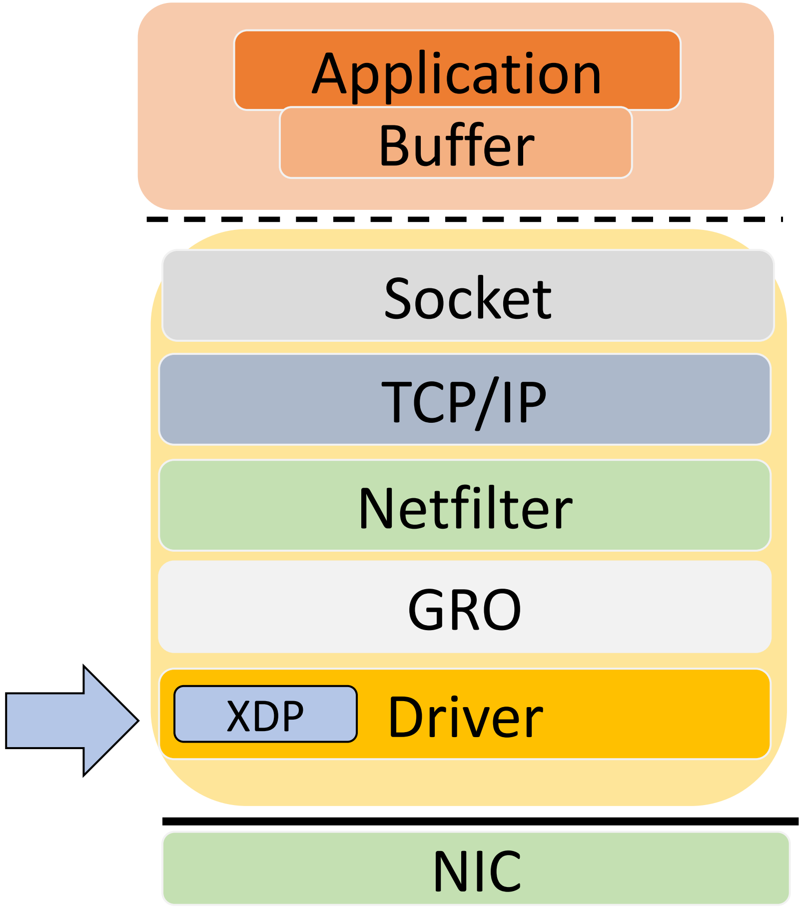


XDP Hook

- XDP allows to run eBPF programs at the driver level, before skb allocation
- This is the earliest hook-point, suitable for high-performance processing
- XDP is good for offloading stateless protocols like UDP

why?

dropping or modifying TCP packets can corrupt the connection state

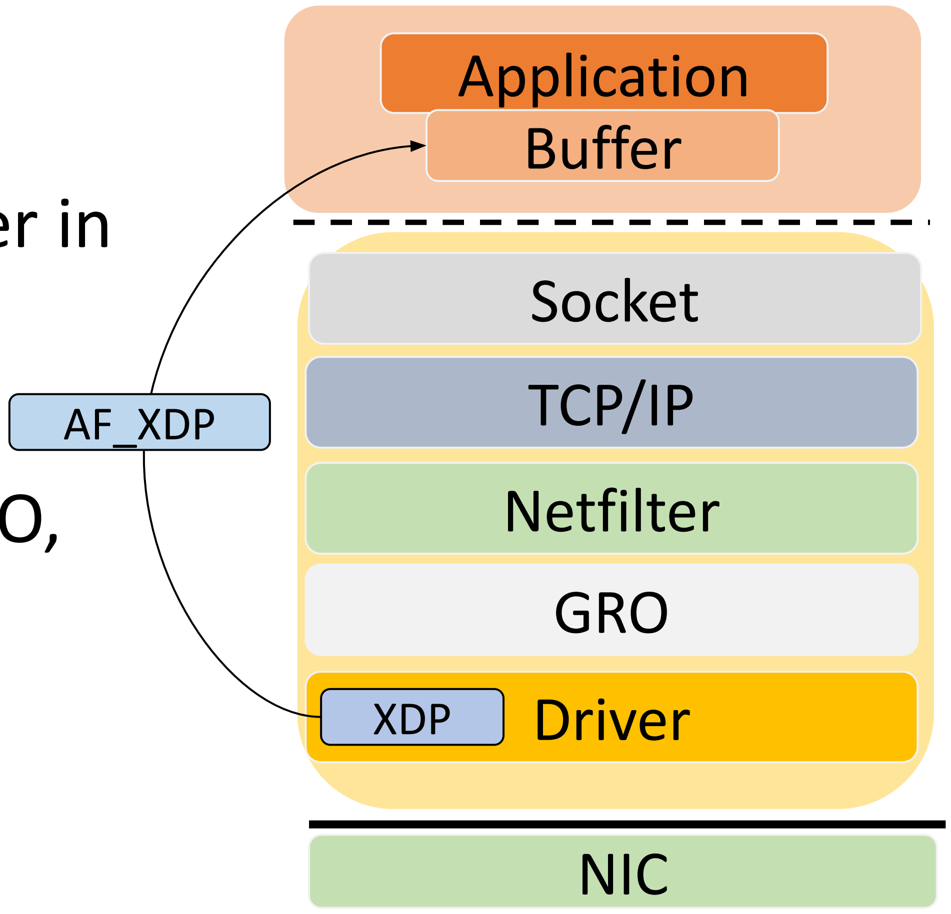


AF_XDP

- AF_XDP sockets enable the possibility for XDP programs to redirect frames to a memory buffer in user-space

- Consequence: with AF_XDP you can bypass GRO, Netfilter and TCP/IP

- This is basically the Linux's answer to kernel bypass (such as DPDK)



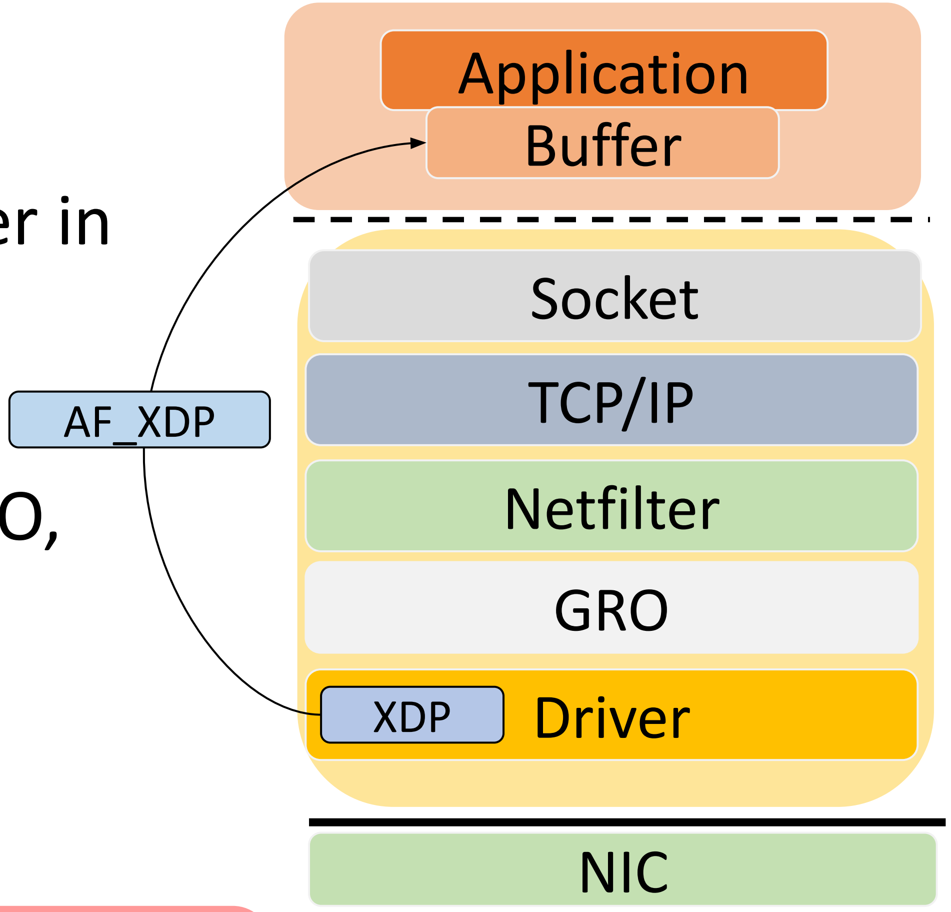
AF_XDP

- AF_XDP sockets enable the possibility for XDP programs to redirect frames to a memory buffer in user-space

- Consequence: with AF_XDP you can bypass GRO, Netfilter and TCP/IP

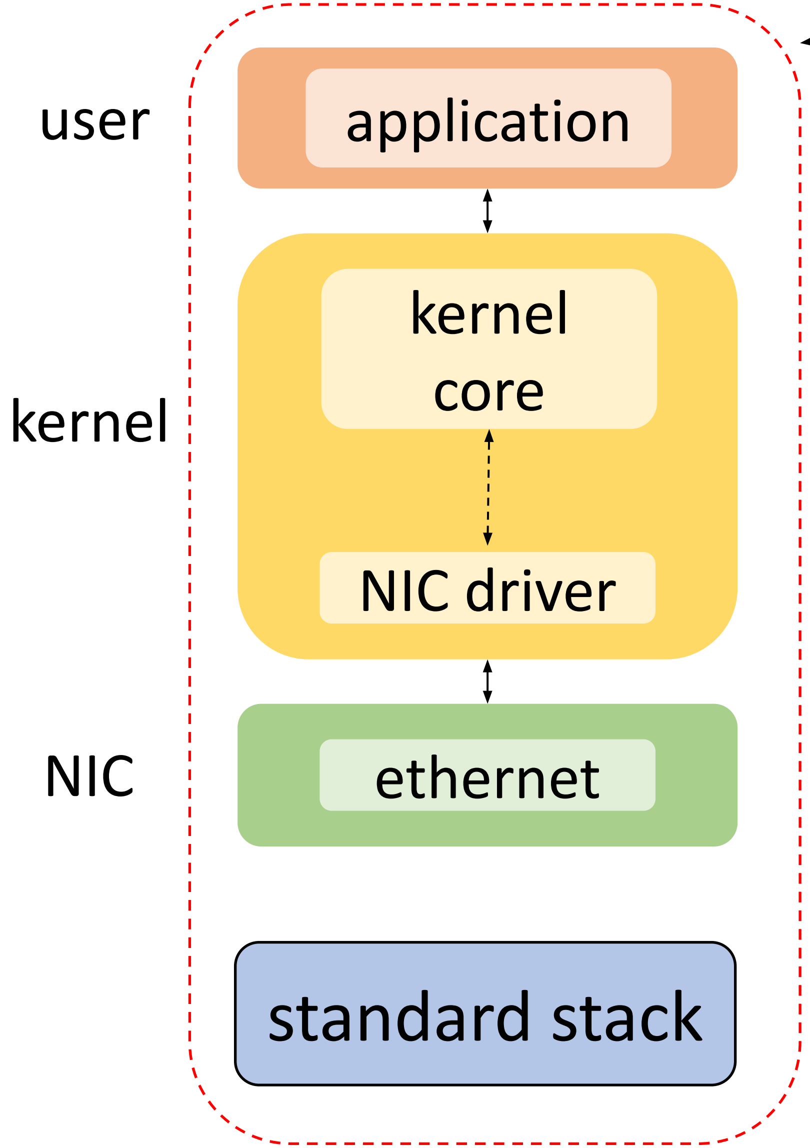
- This is basically the Linux's answer to kernel bypass (such as DPDK)

with DPDK we saw that we give up all the kernel features, is this the same with AF_XDP?

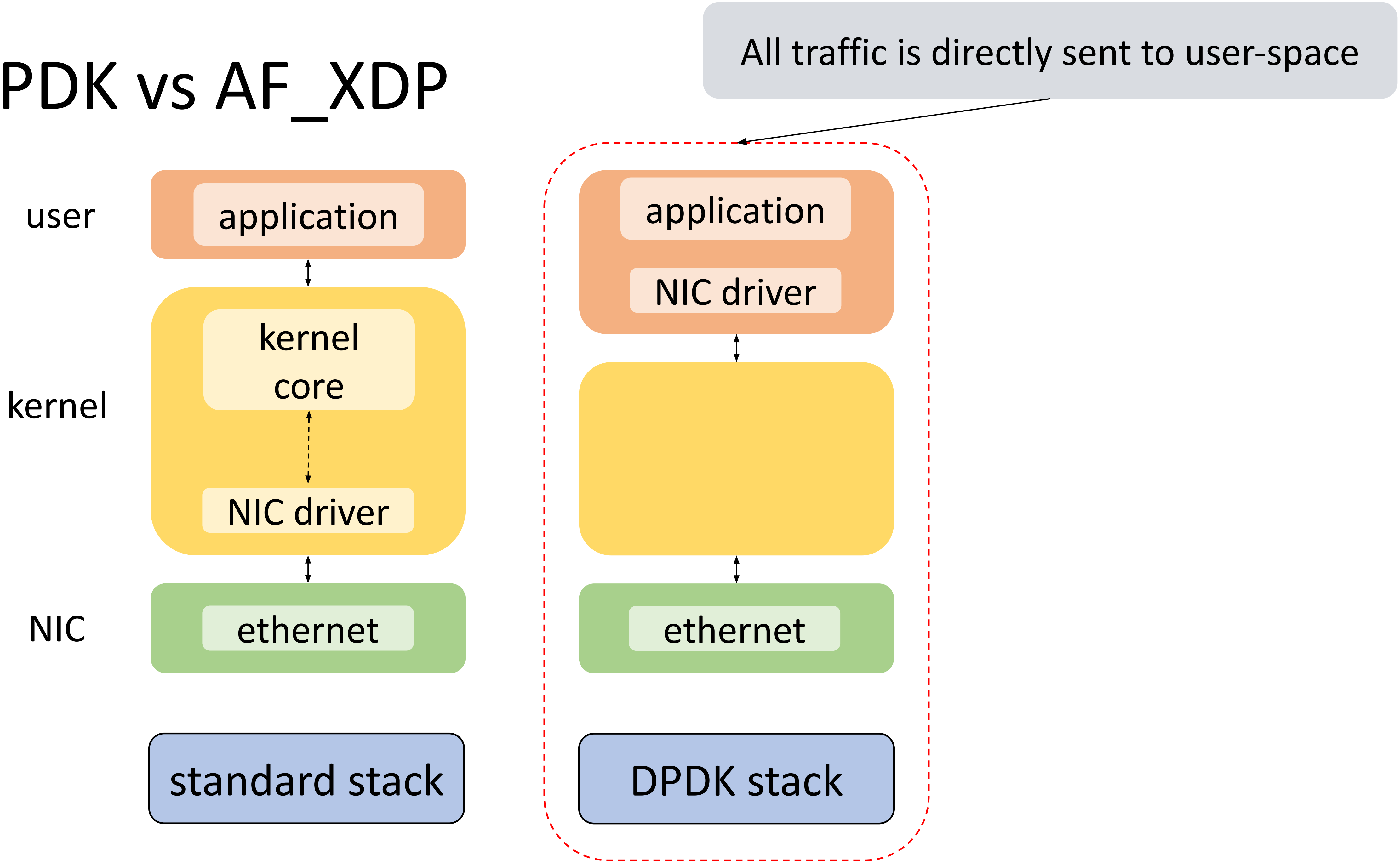


DPDK vs AF_XDP

All traffic goes through the kernel

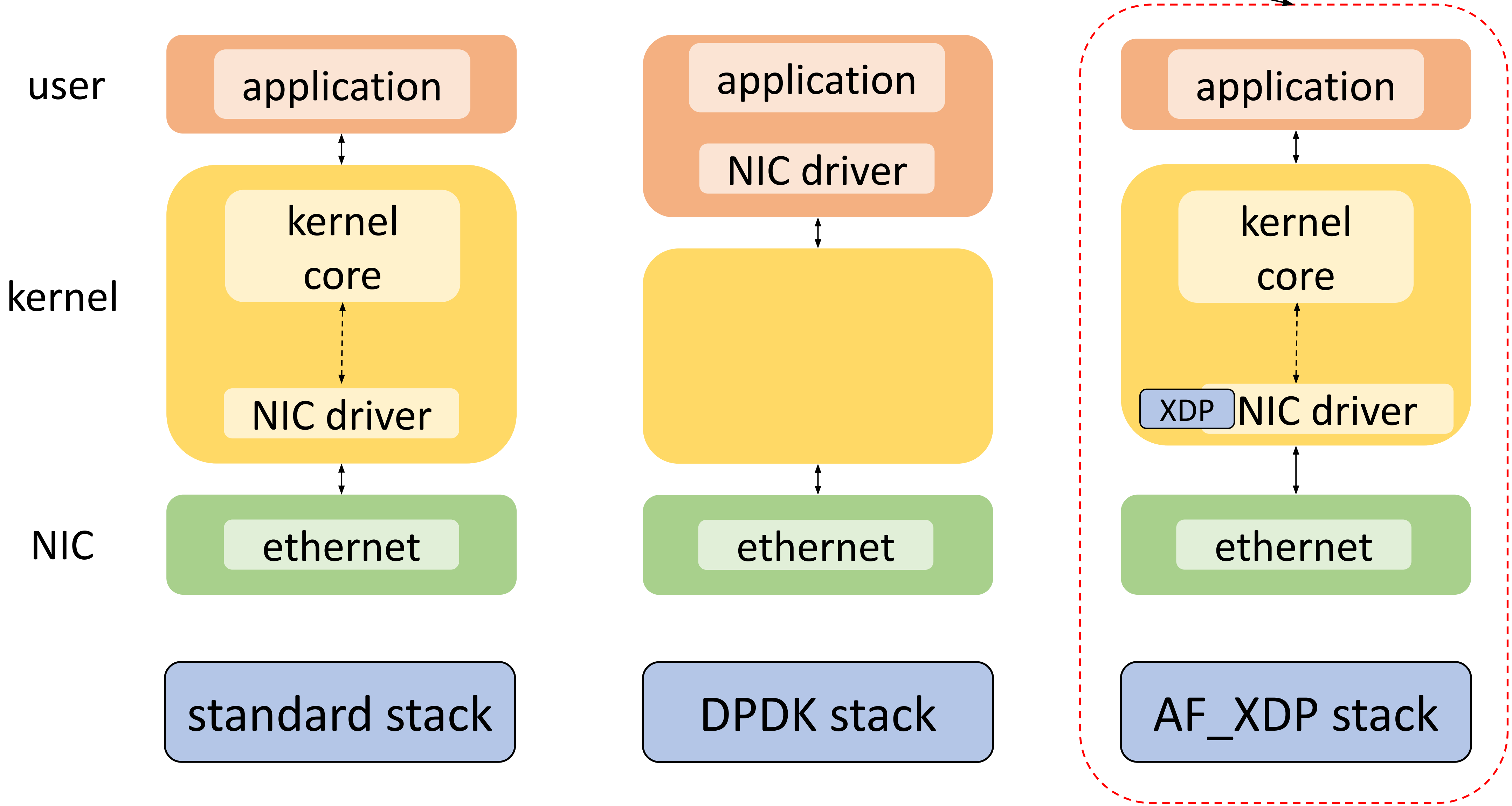


DPDK vs AF_XDP

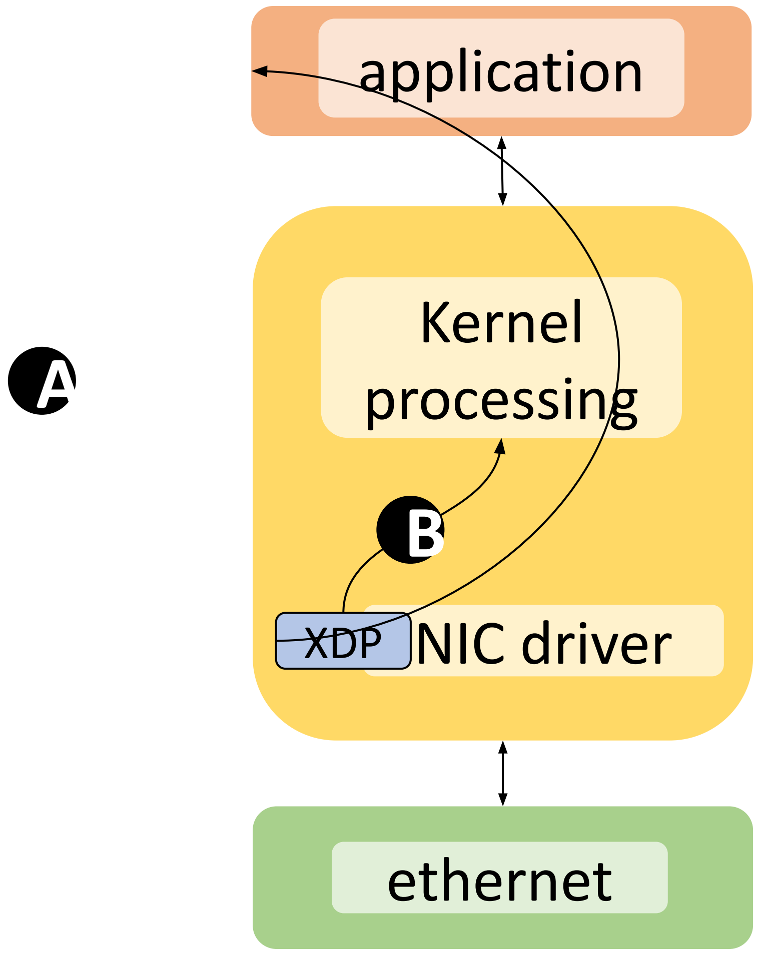


DPDK vs AF_XDP

Depending on the XDP program, traffic can go either through the kernel or directly sent to userspace

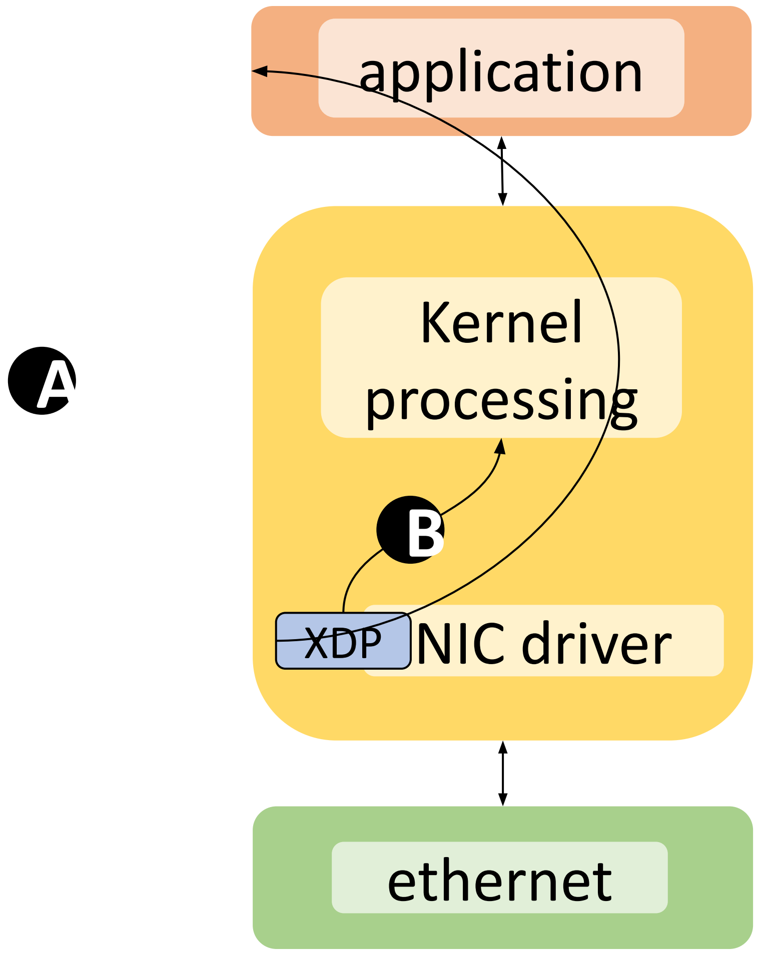


AF_XDP split processing



- The XDP program can steer packets through two different paths
- Path A: packets bypass the kernel and show at the application through the AF_XDP socket interface
- Path B: packets will go through the kernel
- Consequence: I can get kernel features for some packets and high-speed processing for others

AF_XDP split processing



- The XDP program can steer packets through two different paths
- Path A: packets bypass the kernel and show at the application through the AF_XDP socket interface
- Path B: packets will go through the kernel
- Consequence: I can get kernel features for some packets and high-speed processing for others

Example: I can steer ICMP packets through the kernel and get ping working 😊

eBPF helpers

- eBPF assembly instructions are limited for safety reasons
  - We may need to perform some tasks not natively allowed in eBPF
- Solution: helpers!
  - Functions that are implemented natively in the Linux kernel, which are available as an assembly call
  - https://github.com/iovisor/bcc/blob/master/src/cc/export/helpers.h
- Examples: hash, longest-prefix-match tries, printk

Storing data in eBPF

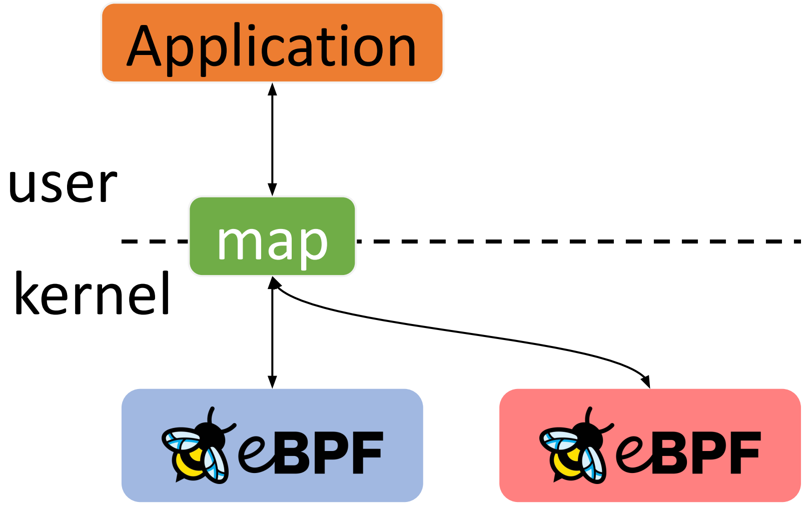
- eBPF adopts preformatted memory, instead of unstructured vanilla memory pages

- Data access arbitrated by structures called *maps*
  - Key/value storage of different types
  - Array, HashMap, LRUMap..

- Interaction with maps is possible through helpers, such as
  - *bpf_map_lookup_elem()*
  - *bpf_map_update_elem()*
  - *bpf_map_delete_elem()*

Storing data in eBPF

- Maps can be used to:
  - Export data from kernel to user space (e.g., kernel updates statistics about network traffic)
  - Data pushed by user space to kernel (e.g., user application configures and eBPF program behavior, for example a routing table)
  - Data shared between different eBPF programs (e.g., packet parser, sharing pointers to relevant protocol fields to all following module)



- Note: there are other shared memory communication channels (i.e., ring buffers, perf buffers)

eBPF toolchain

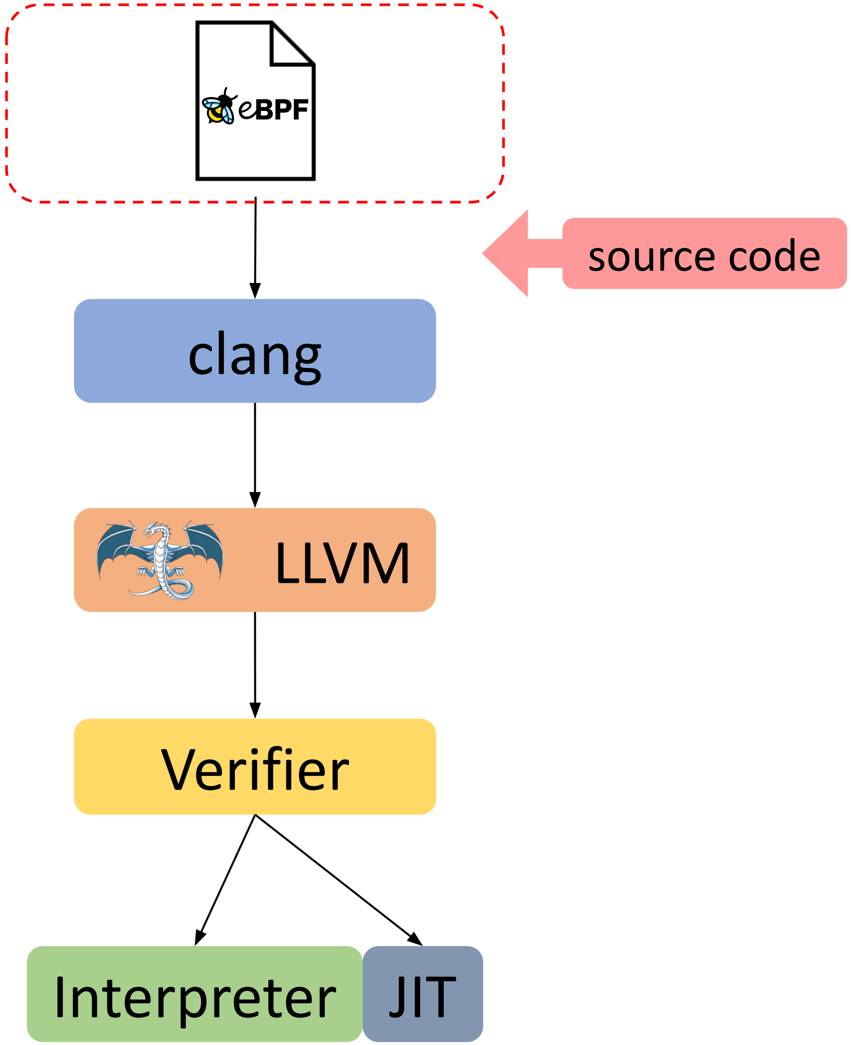
- eBPF code is natively written in restricted C

```
static __always_inline int parse_ethhdr(void *data, void *data_end,
                                        __u16 *nh_off,
                                        struct ethhdr **ethhdr) {
    struct ethhdr *eth = (struct ethhdr *)data;
    int hdr_size = sizeof(*eth);

    /* Byte-count bounds check; check if current pointer + size of header
     * is after data_end.
     */
    if ((void *)eth + hdr_size > data_end)
        return -1;

    *nh_off += hdr_size
    *ethhdr = eth;

    return eth->h_proto; /* network-byte-order */
}
```



eBPF toolchain

- eBPF code is natively written in restricted C

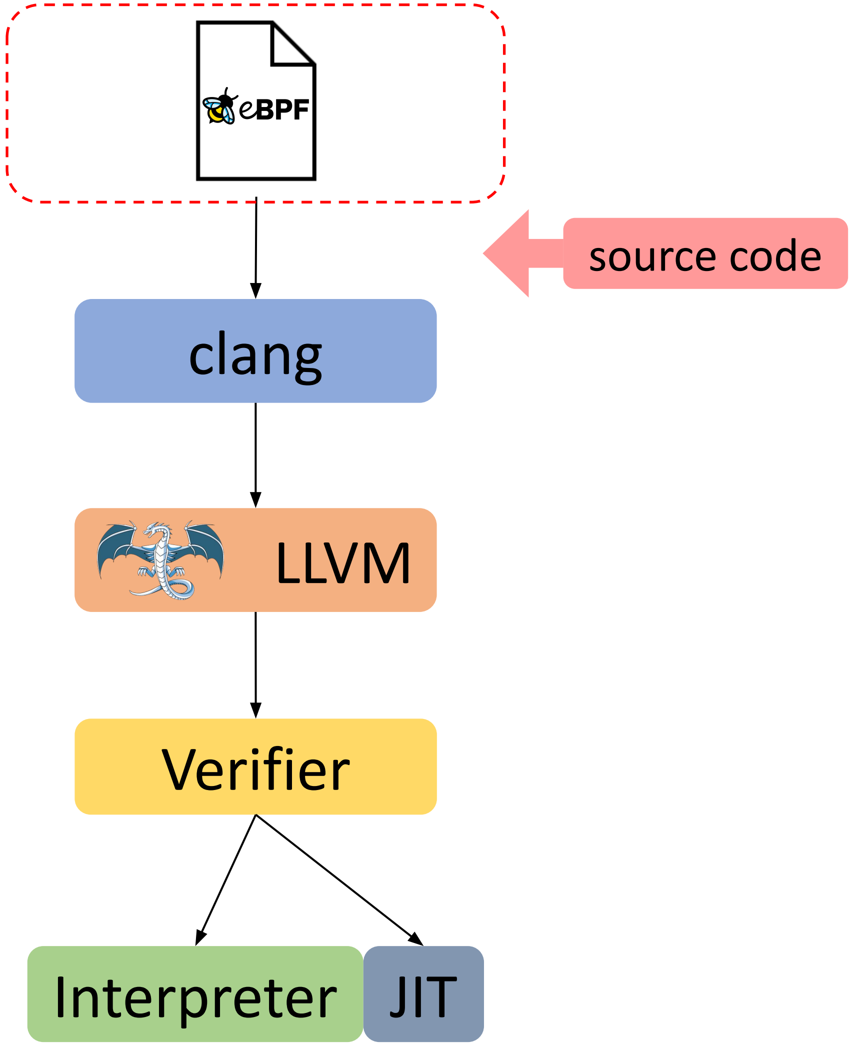
- Compilers for other languages exists
  - To generate eBPF bytecode
  - To interact with the running eBPF program

libbpf-rs | redbpf | Aya
rust

bcc

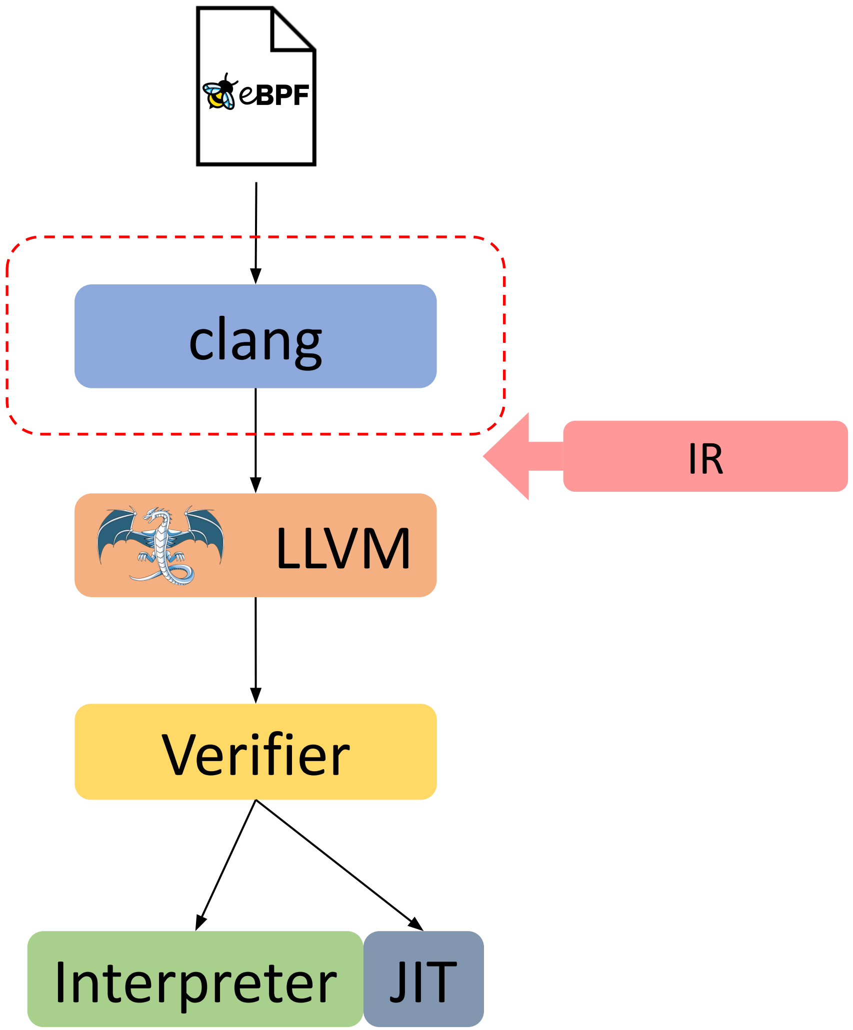
libbpf

cilium/ebpf



eBPF toolchain

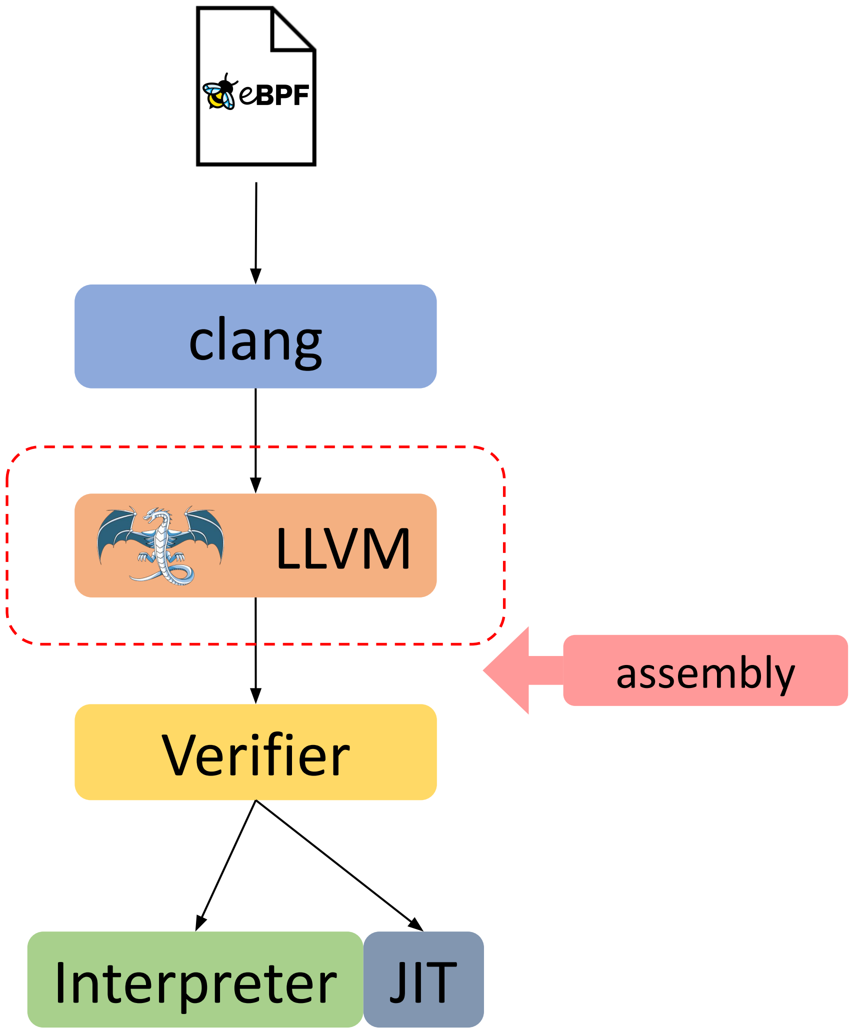
- The code gets processed by Clang, a compiler front end for C-style programming languages
  - clang is a drop-in replacement of GCC
  - eBPF is also supported by GCC, but clang is the most commonly used



eBPF toolchain

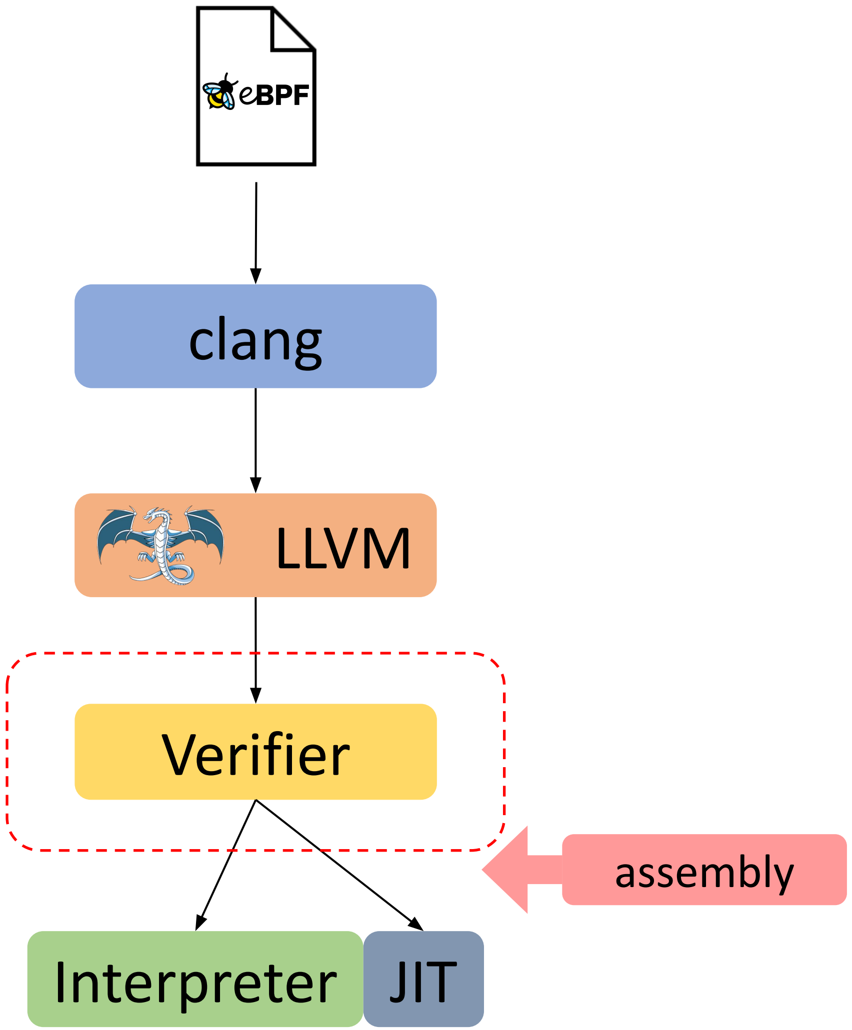
- The code gets processed by Clang, a compiler front end for C-style programming languages
  - clang is a drop-in replacement of GCC
  - eBPF is also supported by GCC, but clang is the most commonly used

- The Intermediate Representation (IR) generated by Clang gets goes to the Low Level Virtual Machine (LLVM), a backend for assembly code generation and optimization



eBPF toolchain

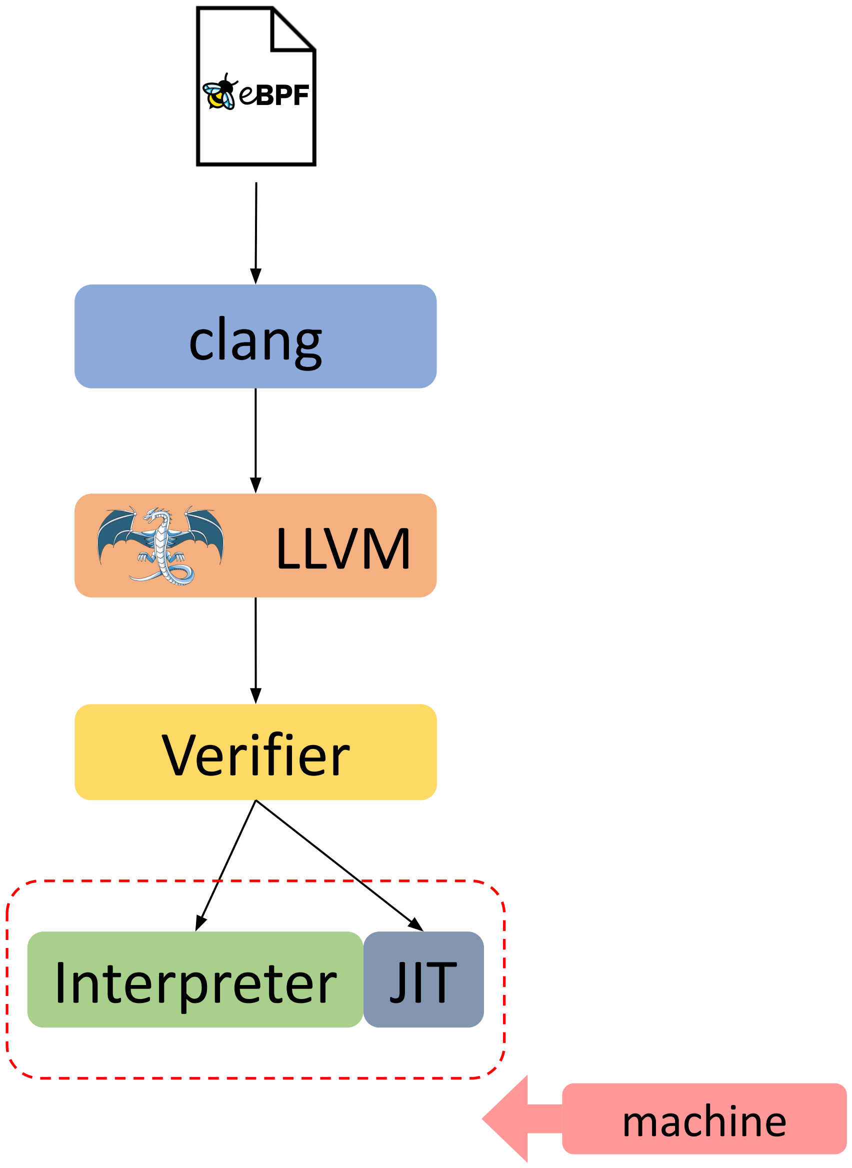
- The assembly code is then checked for safety (we do not want the code to harm kernel behavior)



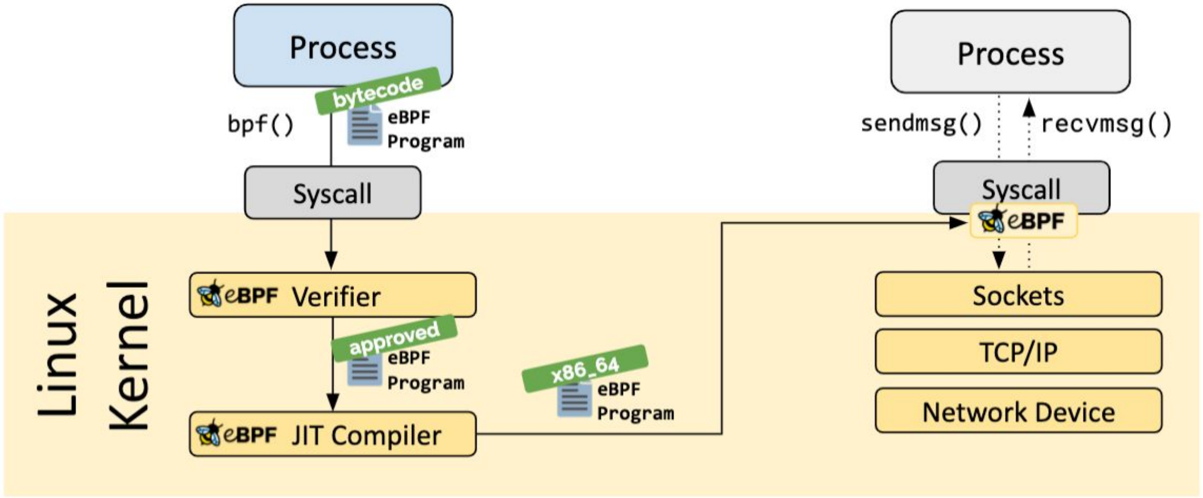
eBPF toolchain

- The assembly code is then checked for safety (we do not want the code to harm kernel behavior)

- As a final step, the verified assembly code goes through either an interpreter or a Just-In-Time compiler for machine code generation



eBPF toolchain



The runtime accepts the bytecode, verifies it, just-in-time compiles it, and runs it at the requested hook point

Facts for the most curious ones

- The eBPF foundation is the entity that control eBPF

Members

Platinum

Meta | Google | ISOVALENT | Red Hat

HUAWEI | CROWDSTRIKE | intel

Silver

NETFLIX | FUTUREWEI | DATADOG

Facts for the most curious ones

- The eBPF foundation is the entity that control eBPF

- There is also an Android BPF loader

Android

Android BPF loader

During Android boot, all eBPF programs located at `/system/etc/bpf/` are loaded. These programs are binary objects built by the Android build system from C programs and are accompanied by `Android.bp` files in the Android source tree. The build system stores the generated objects at `/system/etc/bpf`, and those objects become part of the system image.

https://source.android.com/docs/core/architecture/kernel/bpf

Facts for the most curious ones

- The eBPF foundation is the entity that control eBPF

- There is also an Android BPF loader

- BPF is not just a Linux thing

eBPF
For windows

https://github.com/microsoft/ebpf-for-windows

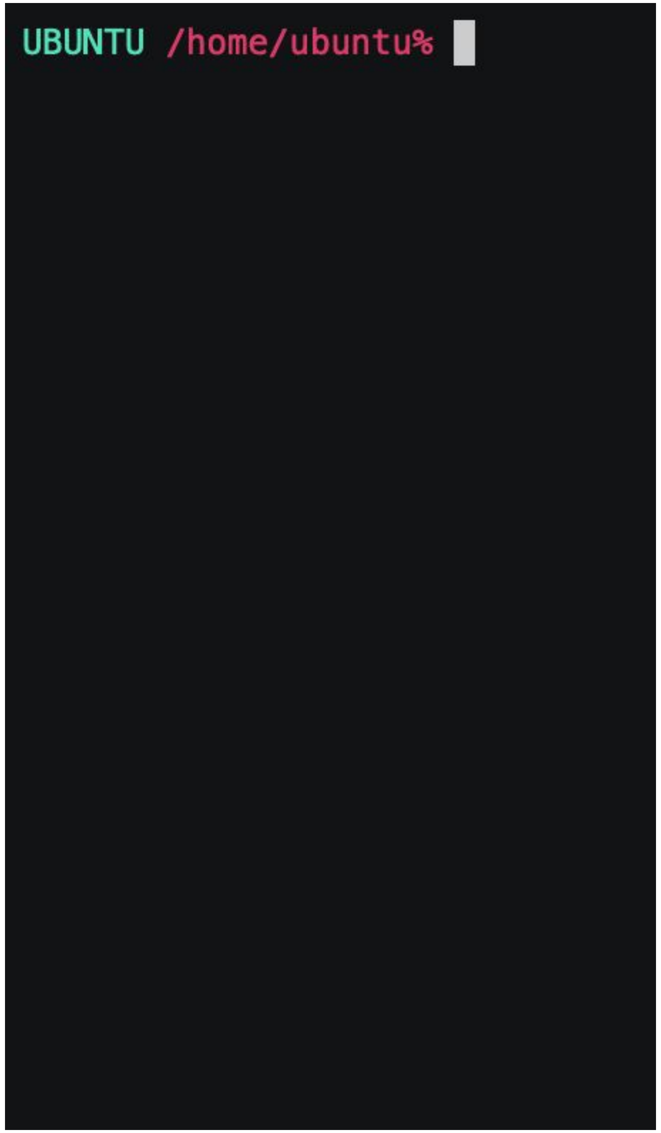
Facts for the most curious ones

- The eBPF foundation is the entity that control eBPF

- There is also an Android BPF loader

- BPF is not just a Linux thing

- You can also build Tetris in BPF!



https://github.com/mmisono/bpftrace-tetris