

Advanced Networked Systems SS24

Host Networking

Prof. Lin Wang, Ph.D.

Computer Networks Group

Paderborn University

<https://cs.uni-paderborn.de/cn>

This material is (partially) inspired by the slides from Mina Tahmasbi Arashloo (University of Waterloo)



Learning objectives

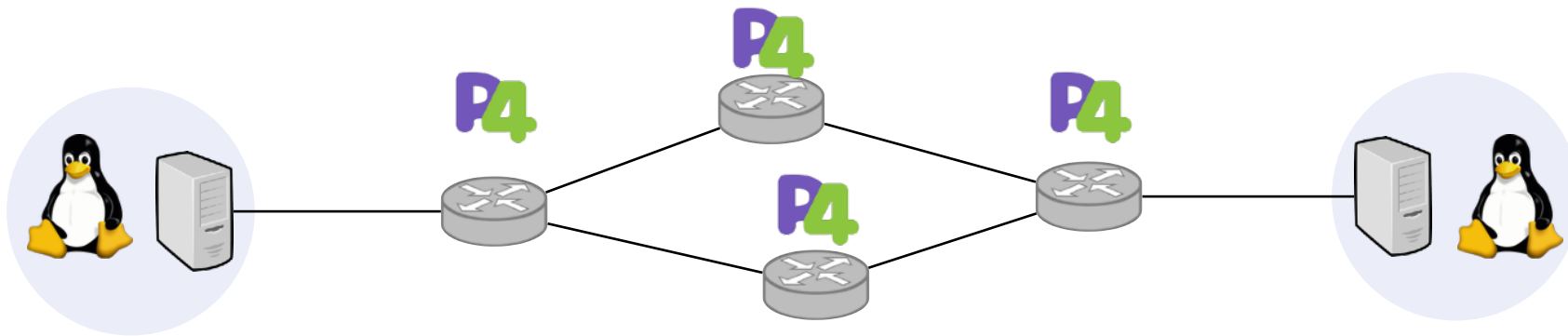
Why and how to **offload** to NIC?

How to make the **Linux kernel programmable** for networking?

How **not to use the Linux kernel** for networking?

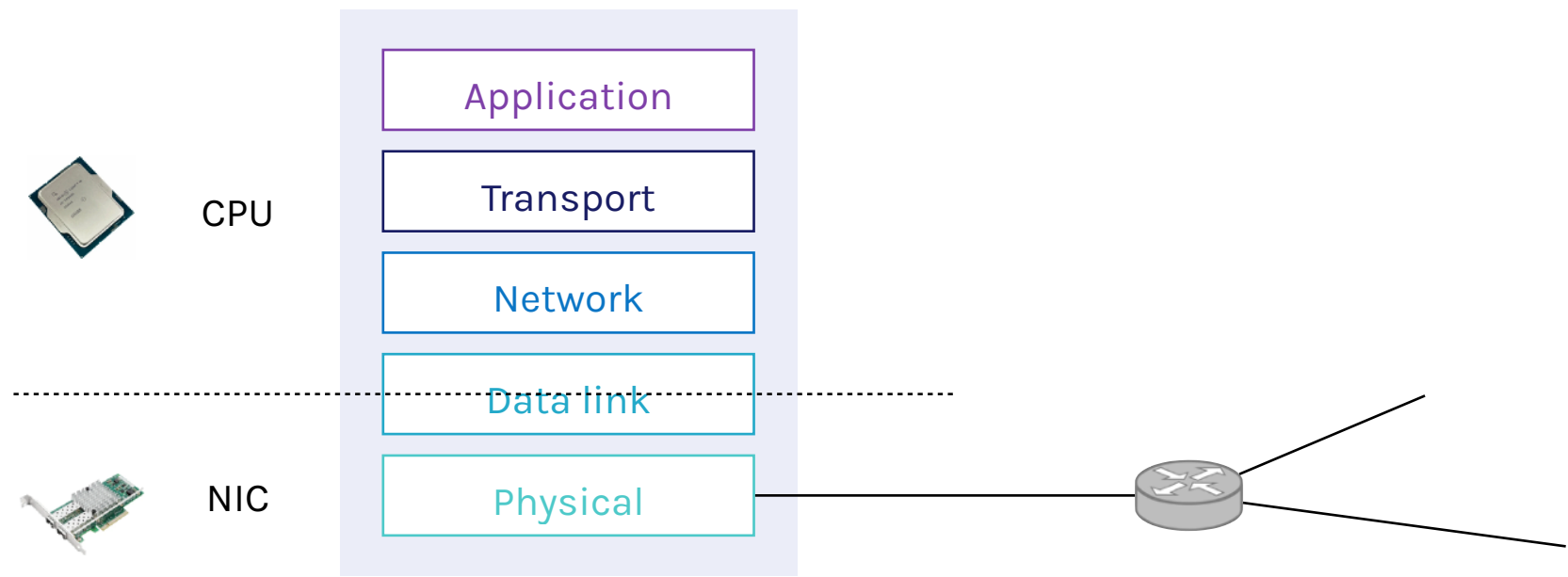
Network programmability on end hosts

Network stack for preparing and processing packets at the end hosts

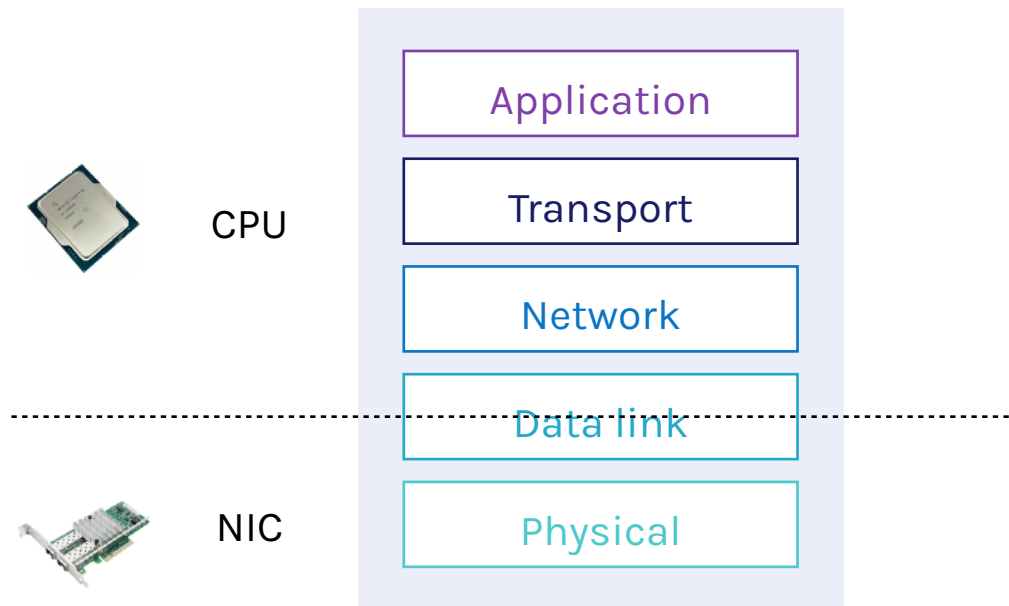


Programmability inside the network via SDN and P4 programmable data planes

End-host network stack



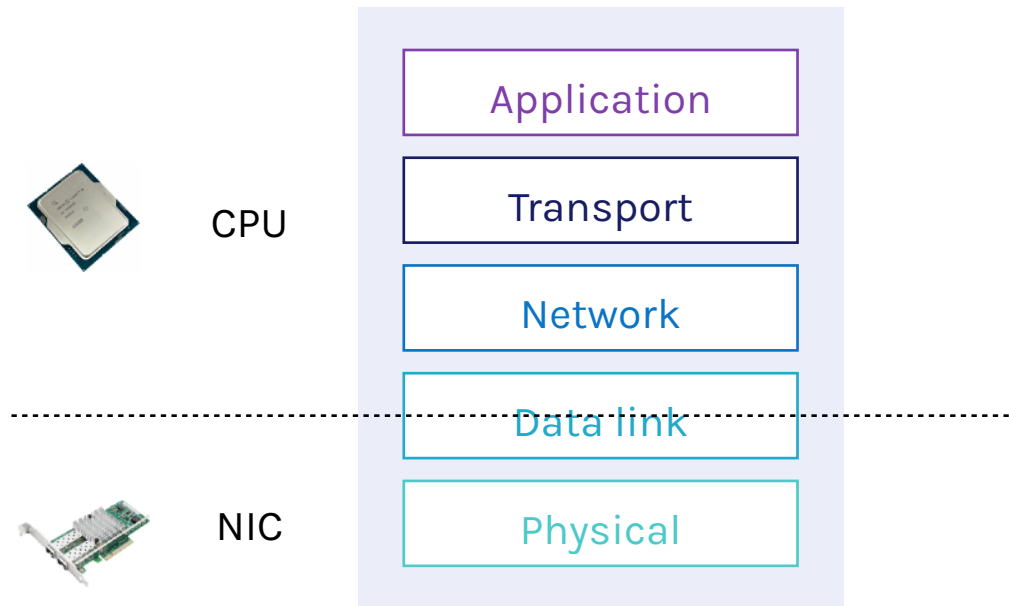
Division of labor



On packet transmit (egress):

- The host CPU generates packets on application request
- Packets are sent to the NIC over PCIe
- The NIC transforms packets to bits and then over the link

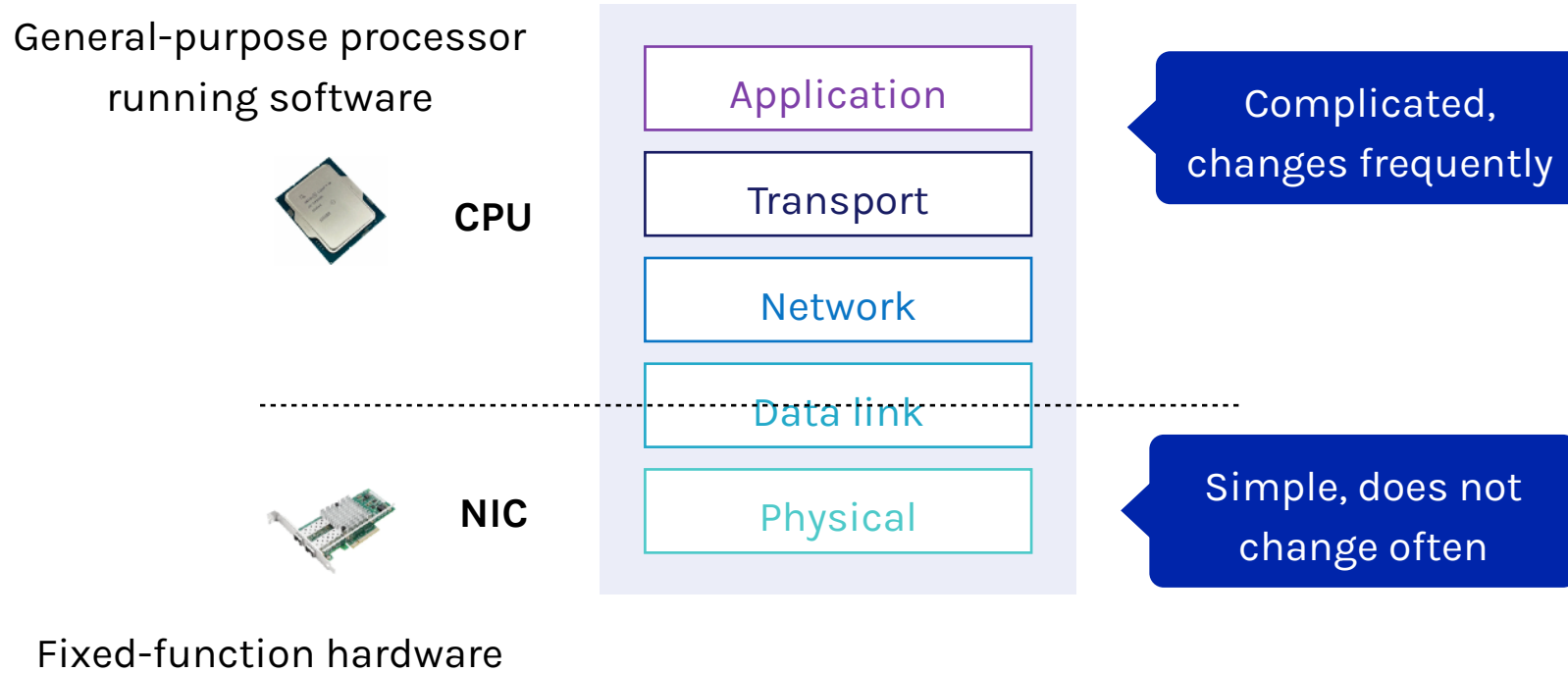
Division of labor



On packet receive (ingress):

- The NIC turns bits into packets
- Packets are sent to the host over PCIe
- The host CPU processes packets and delivers them to applications

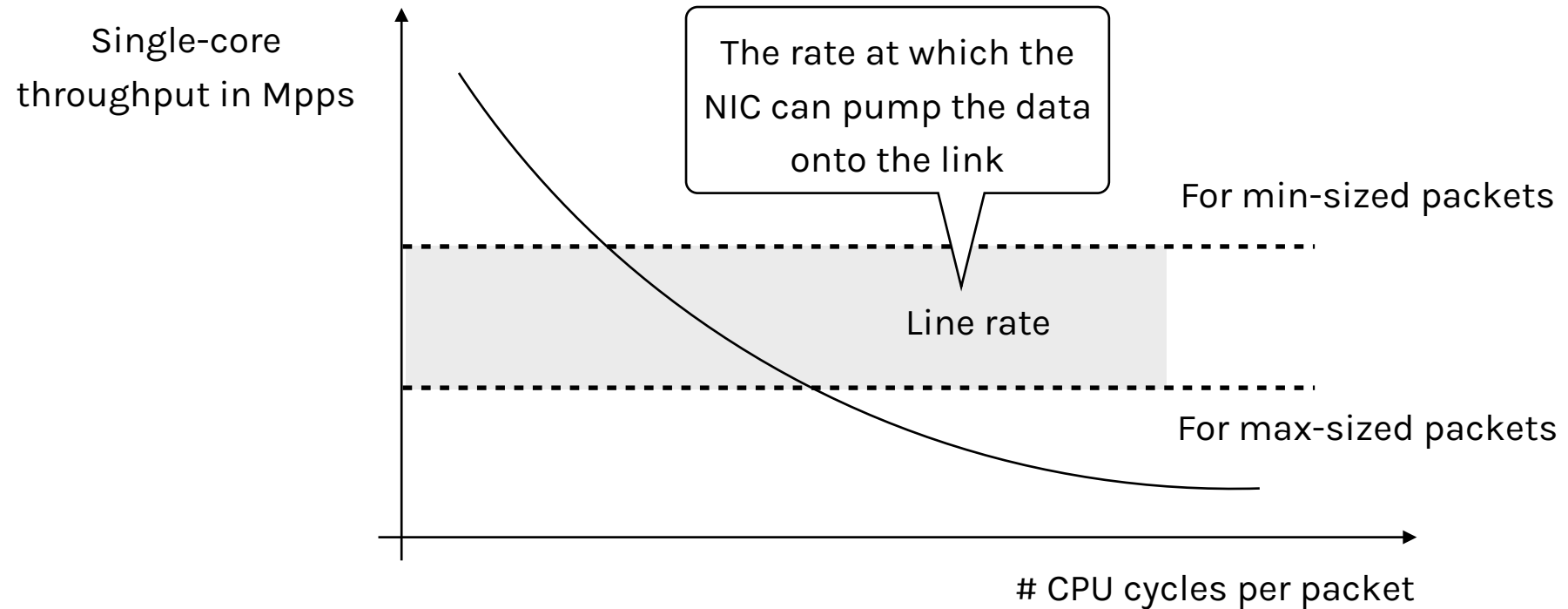
Division of labor



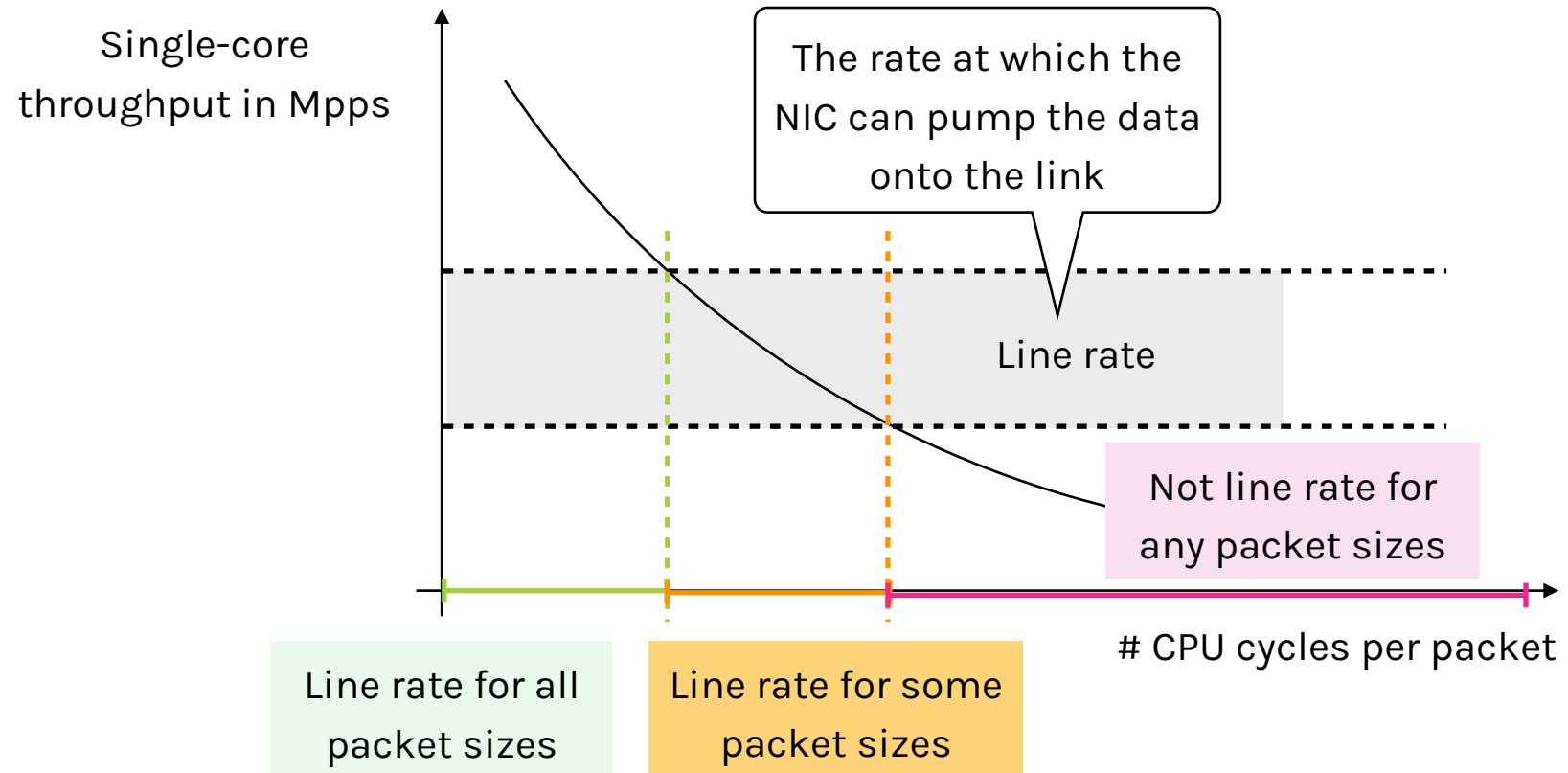
Can CPU keep up with the NIC, i.e., process packets at line rate in reasonable #cycles?

Network Interface Card (NIC)

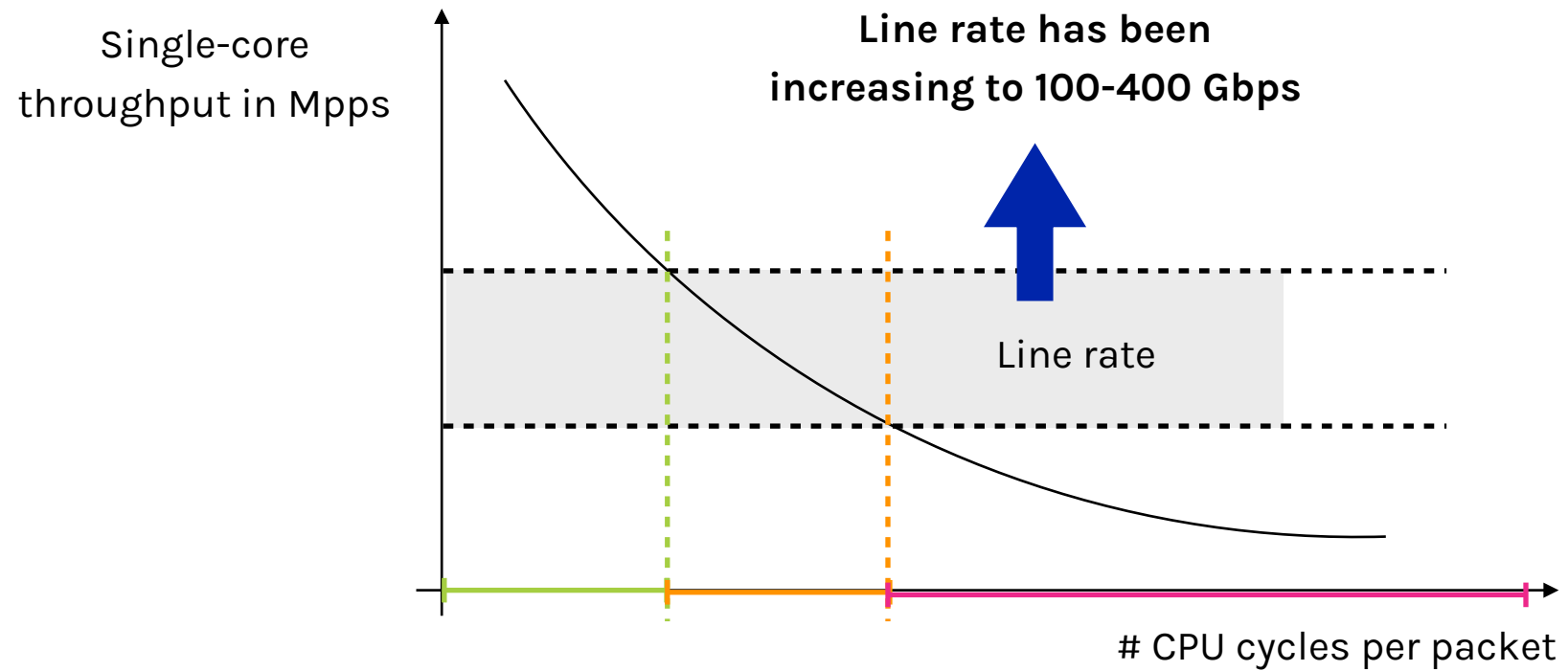
Limits of software packet processing



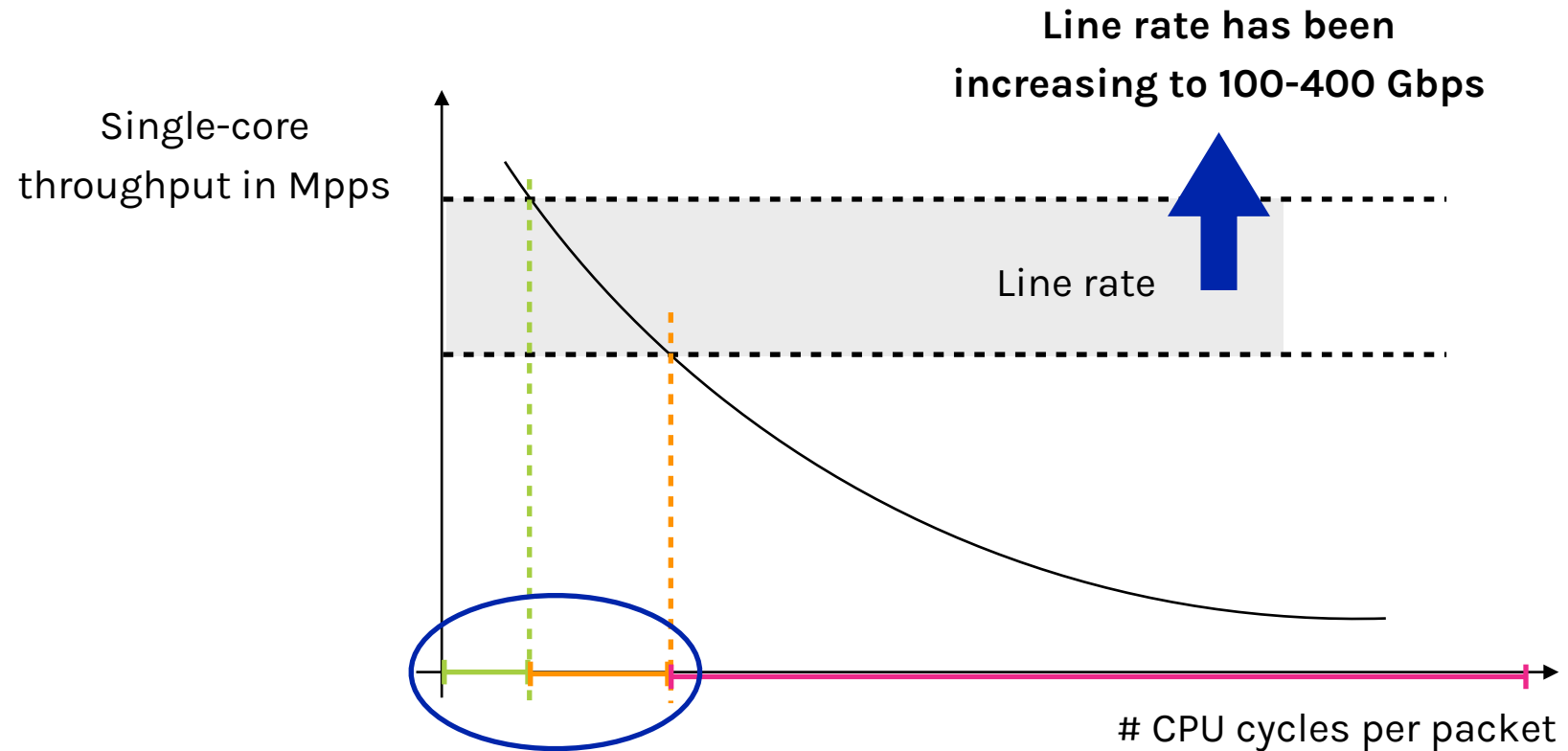
Limits of software packet processing



Limits of software packet processing

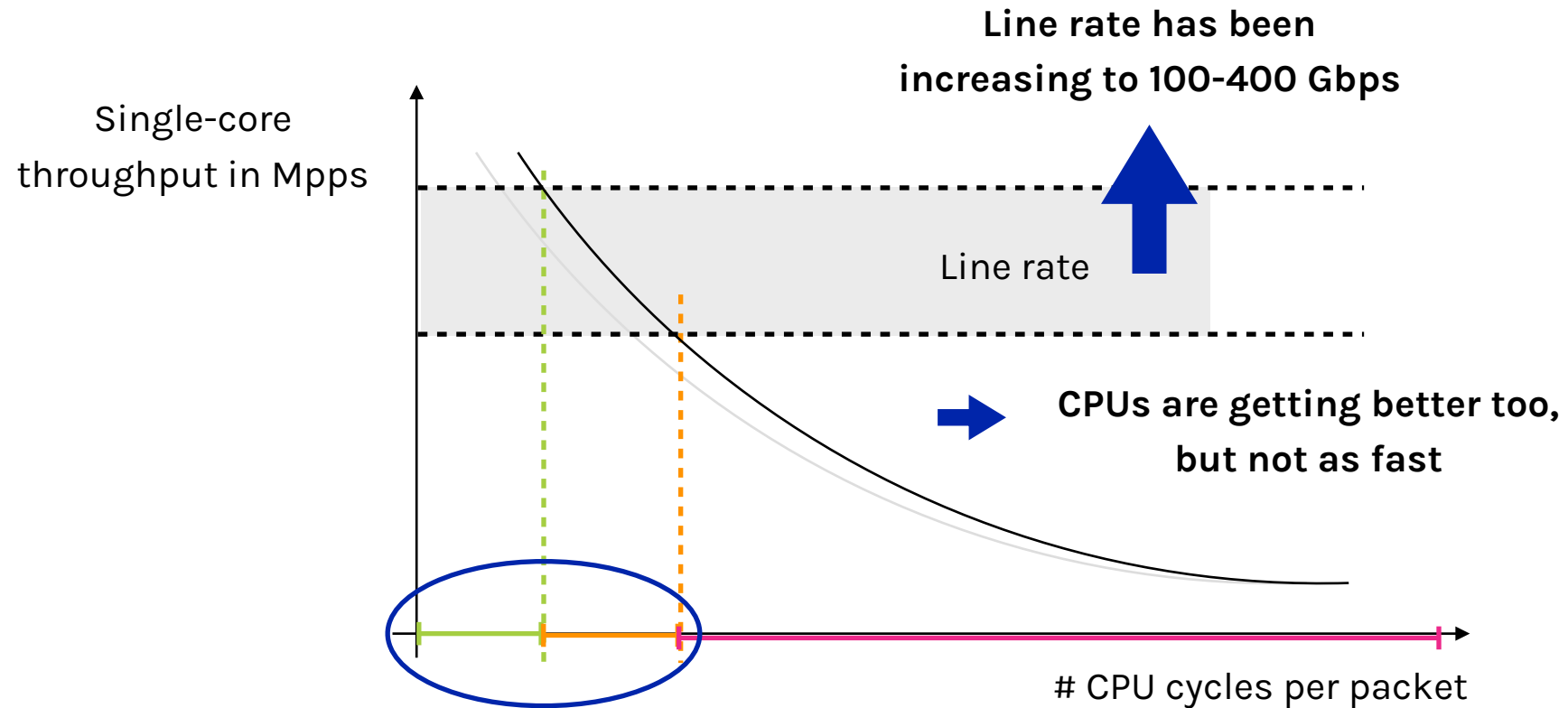


Limits of software packet processing

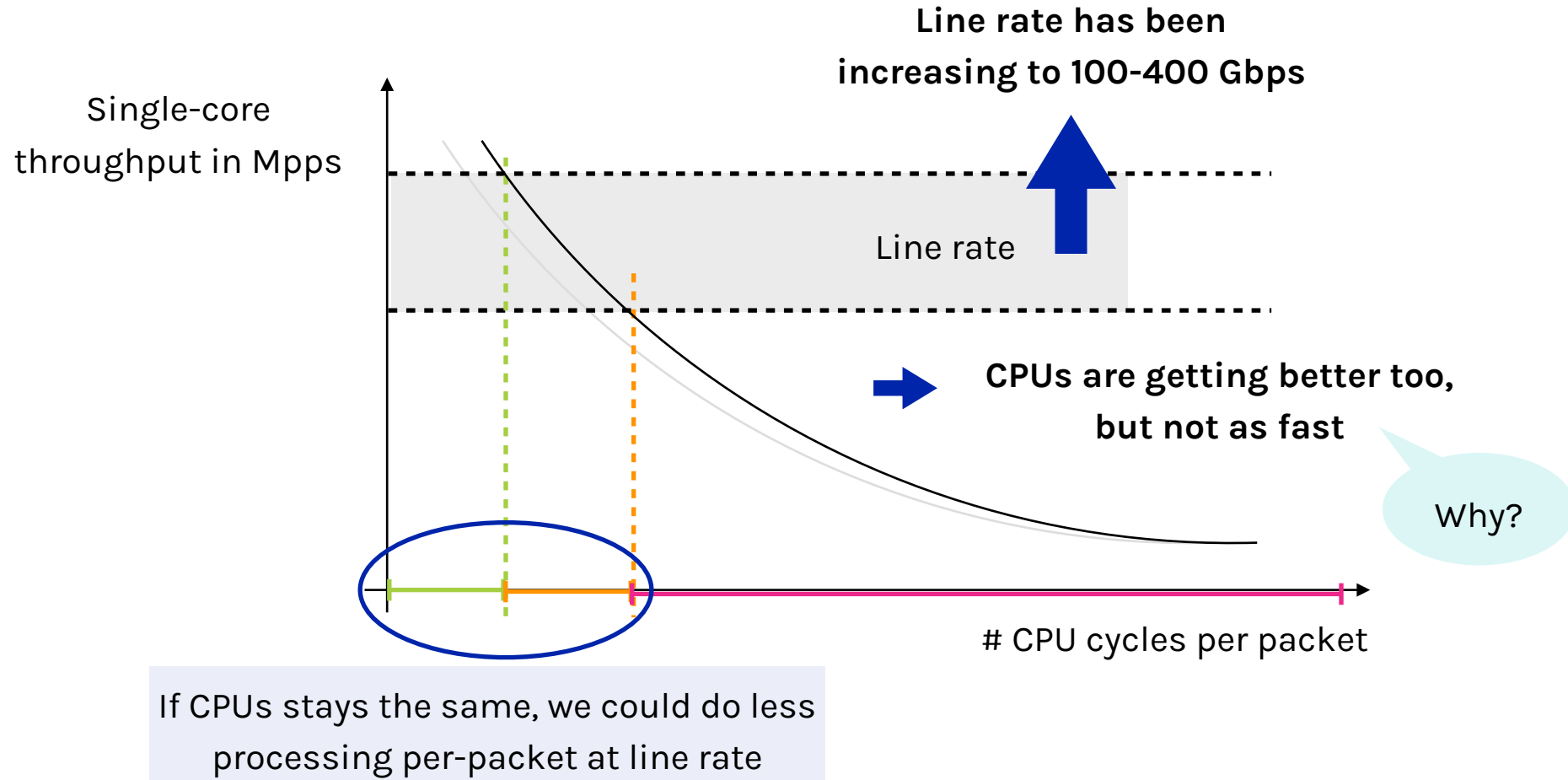


If CPUs stays the same, we could do less processing per-packet at line rate

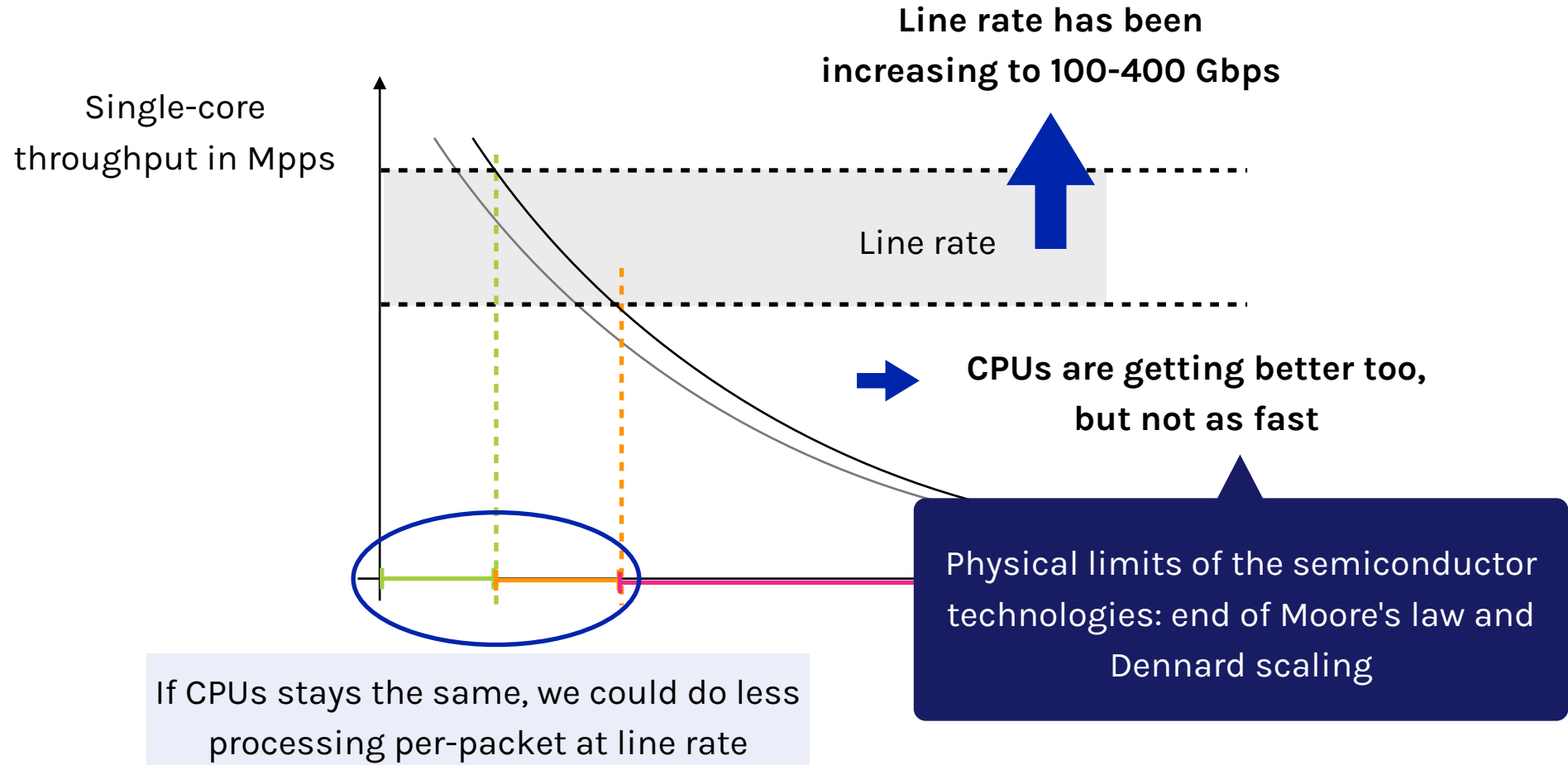
Limits of software packet processing



Limits of software packet processing



Limits of software packet processing



What options do we have?

Processor

Takes over part (or even all) of packet processing that is currently done by CPUs

What options do we have?

Hardware

Accelerator, specialized for network processing

Processor

Takes over part (or even all) of packet processing that is currently done by CPUs

What options do we have?

Programmable

Customizable offloading (what and how)

Hardware

Accelerator, specialized for network processing

Processor

Takes over part (or even all) of packet processing that is currently done by CPUs

What options do we have?

Programmable

Customizable offloading (what and how)

Hardware

Accelerator, specialized for network processing

Processor

Takes over part (or even all) of packet processing that is currently done by CPUs

On the NIC

Co-locating with the NIC provides extra benefits

SmartNICs



A regular NIC



Programmable domain-specific hardware

Field Programmable Gate Arrays (FPGA)
Multi-Core Systems on Chip (SoC)
P4-programmable pipelines
Or combinations of the above...

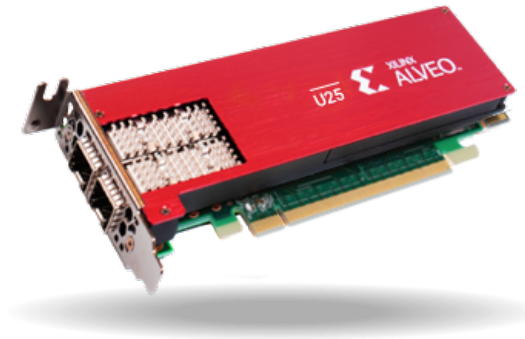
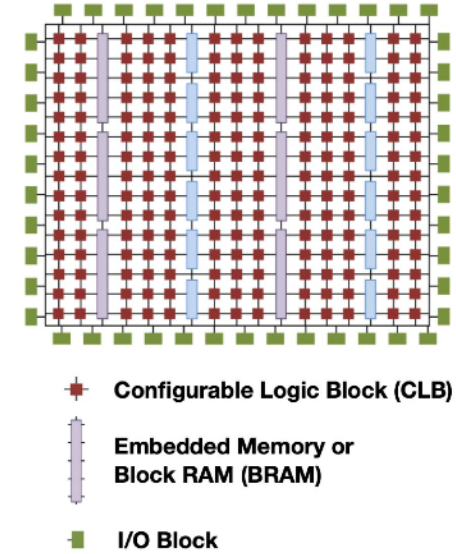
FPGA-based SmartNICs

FPGA

- A collection of small reconfigurable logic and memory blocks
- Programmers can write code to assemble these blocks to perform desired processing

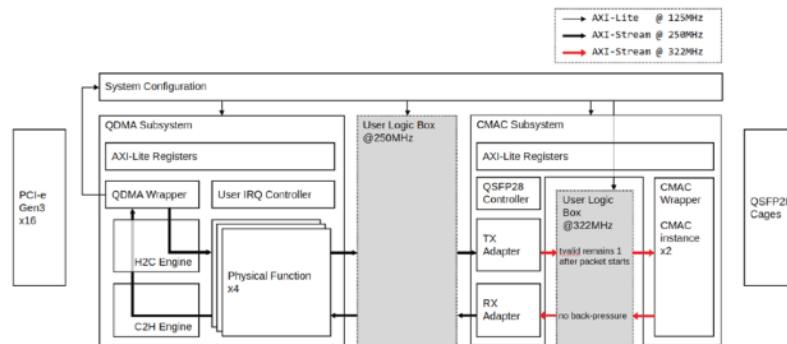
Why good for SmartNIC?

- FPGA hardware resources (logic and memory) can be highly customized for the intended computation
- Good fit for highly-parallelizable computation



AMD OpenNIC Project

The OpenNIC project provides an FPGA-based NIC platform for the open source community. It consists of multiple components: a NIC shell, a Linux kernel driver, and a DPDK driver. The NIC shell contains the RTL sources and design files for targeting several of the AMD-Xilinx Alveo boards featuring UltraScale+ FPGAs. It delivers a NIC implementation supporting up to four PCI-e physical functions (PFs) and two 100Gbps Ethernet ports. The shell is equipped with well-defined data and control interfaces and is designed to enable easy integration of user logic into the shell. A block diagram of the OpenNIC shell follows:



Intel® P4 Suite for FPGA

The Intel® P4 Suite for FPGA automates the generation of packet-processing IP and adapts the P4 Architecture to reflect the flexibility of FPGAs using networking hardware and software.



Overview

Key Features

Development Tool Flow

Applications

Overview

The Intel® P4 Suite for FPGA is a high-level design tool that:

- uses P4, an open-source, domain-specific language that describes how a networking data plane device processes a packet.
- automates the generation of packet-processing RTL IP.
- can be used across a wide range of networking hardware and software.

The tool consists of:

- A compiler that generates RTL from a P4 program.
- An FPGA Software Framework that provides a software Application Programming Interface (API) that controls the P4-generated RTL at runtime.

Feedback

Why not creating our own NIC or switches to support in-network computing?

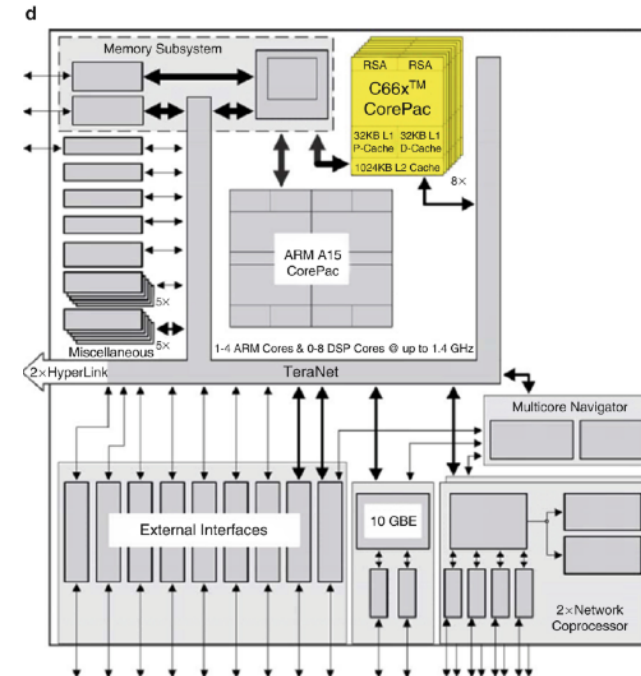
SoC-based SmartNICs

SoC

- A “small” computer on a single chip
- Includes (light-weight) processing cores and a memory hierarchy

Why good for SmartNIC?

- Programming model is close to software
- Cores (and the architecture) can be specialized for network processing



AMD Pensando™ DPU Accelerators

AMD Pensando™ DPU Accelerators

At least a generation ahead of the competition¹



Overview Portfolio Customer Stories Test Drive Resources

Bring Hyperscale DPU Technology to Your Data Center

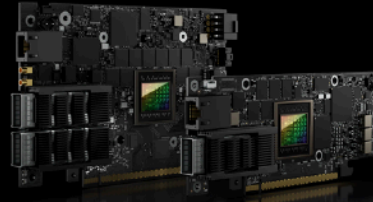
Built on the same technology that the hyperscalers use, the heart of the AMD Pensando™ platform is our fully programmable P4 data processing unit (DPU). Optimized to execute a software stack delivering cloud, compute, network, storage, and security services at cloud scale - with minimal latency, jitter, and power requirements. Where other vendors have roadmaps, the AMD Pensando™ platform has an established solution in production today, with leading cloud service providers and enterprise customers.

Networking Products Solutions Industries Support Buy

NVIDIA BlueField Networking Platform

Advanced infrastructure computing platform for powering the world's data centers.

Latest News



FPGA vs SoC

	FPGA	SoC
Hardware architecture	Reconfigurable hardware and therefore can be highly customized for the intended packet processing	The cores' instruction set and memory architecture are fixed and are therefore less customizable
Programming model	Hardware description languages (e.g., Verilog) → hard to program	C-like languages → easier to program
Performance	High throughput, low latency	Lower throughput, higher latency

Notes on SmartNIC

SmartNICs can (and do) have fixed-function blocks

- These blocks are optimized hardware implementations of common packet processing functionality: encryption, hashing, certain common protocols

A fully ASIC-based NIC can still be considered a SmartNIC

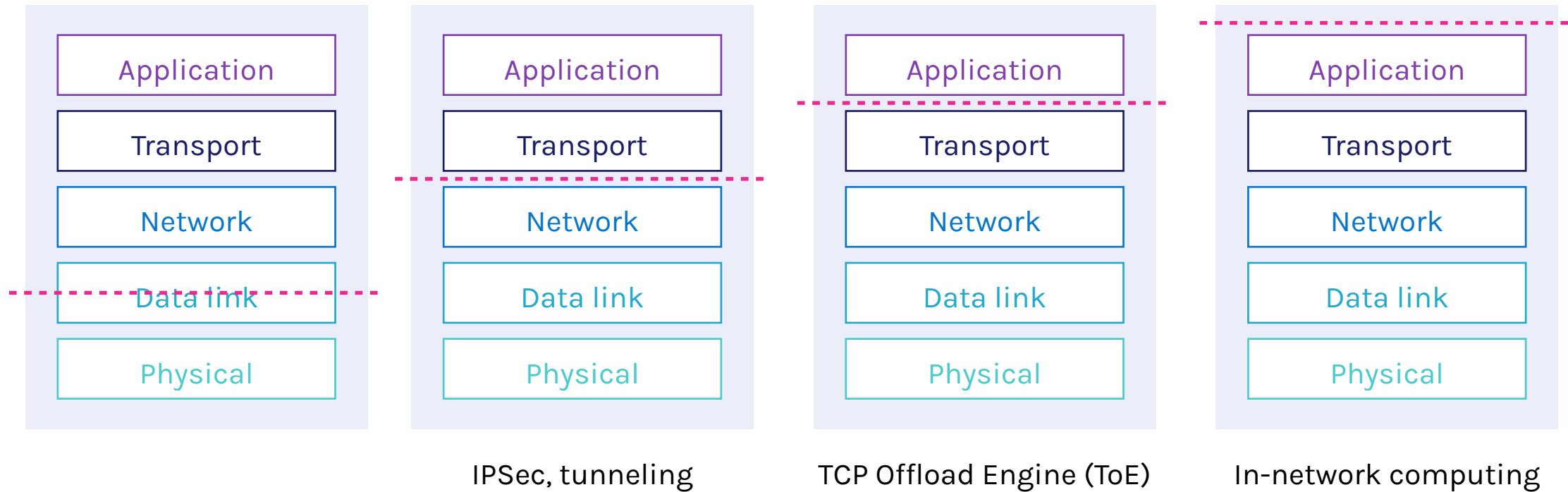
- As long as it supports more complex functionality than a traditional NIC

In industry, SmartNICs have various names

- Nvidia Bluefield Data Processing Unit (DPU), Intel Infrastructure Processing Unit (IPU)
- A combination of FPGAs, SoCs, P4 programmable pipelines, fixed function accelerators

What can we do with SmartNICs?

We can move the line for the division of labor



Programming abstractions for SmartNICs

There is such a wider range of functionality people can and are interested in implementing on the NICs

There are many different SmartNIC architectures

- FPGAs, different kinds of SoCs, P4 pipelines, fixed-function blocks, combinations of these

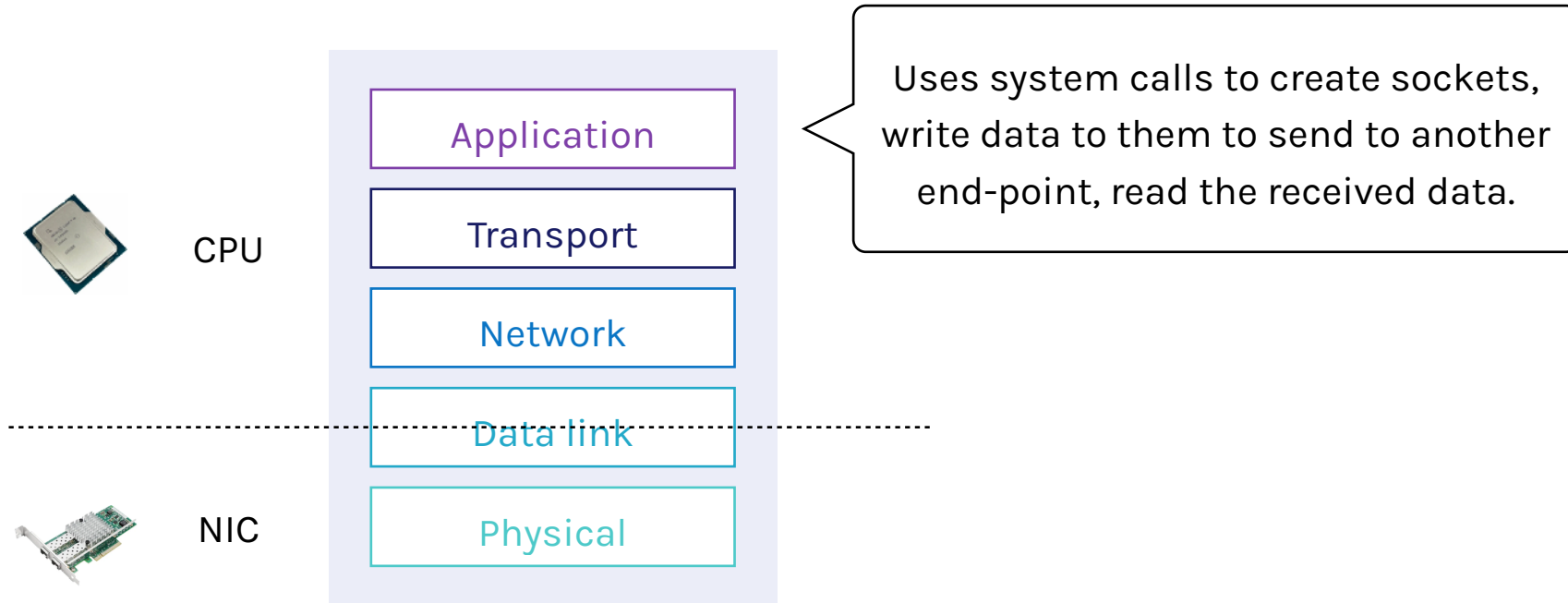
Do we keep P4 and rely on architectures and their externs for all the extra functionality?

- There are efforts on creating a portable NIC architecture in p4.org

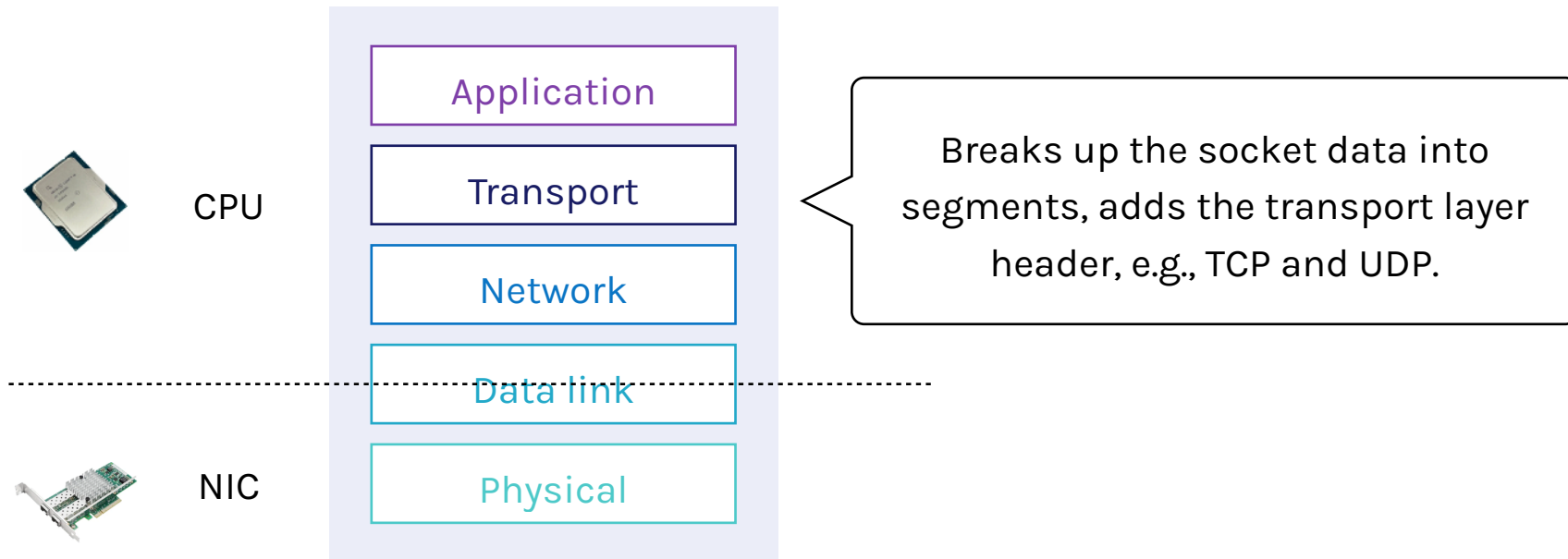
How do we partition/distribute the functionality over different kinds of hardware? What is the best offloading strategy?

Kernel Packet Processing

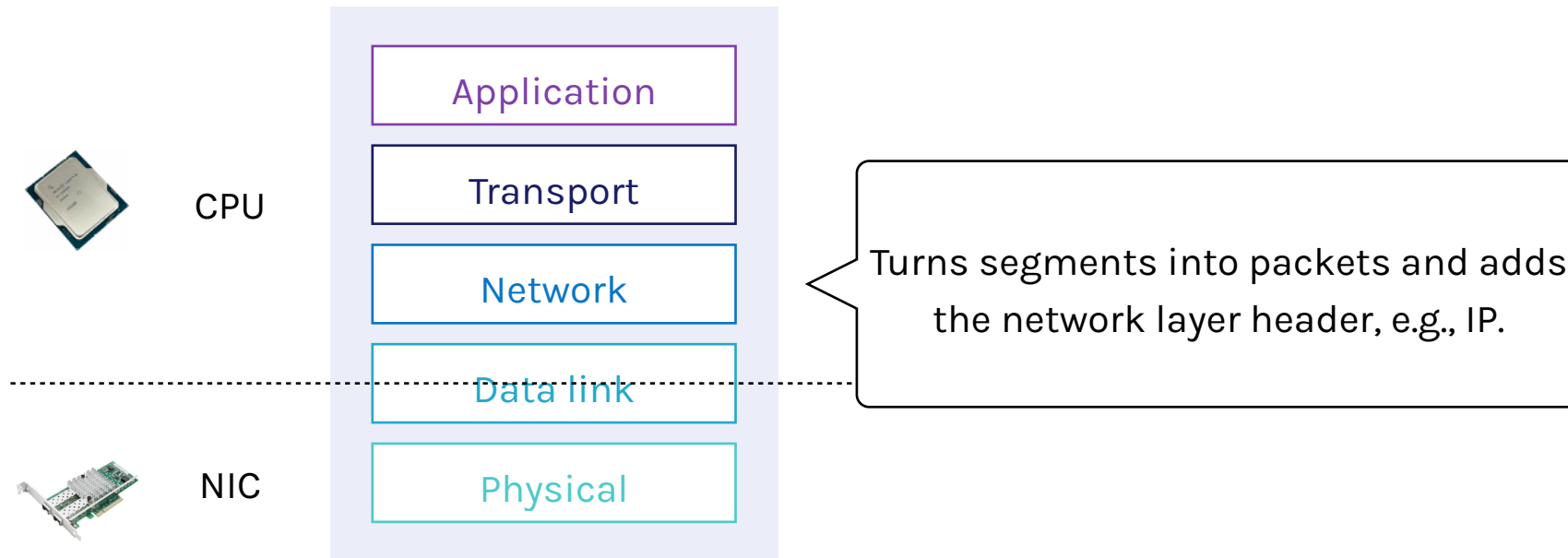
The Linux kernel network stack (simplified)



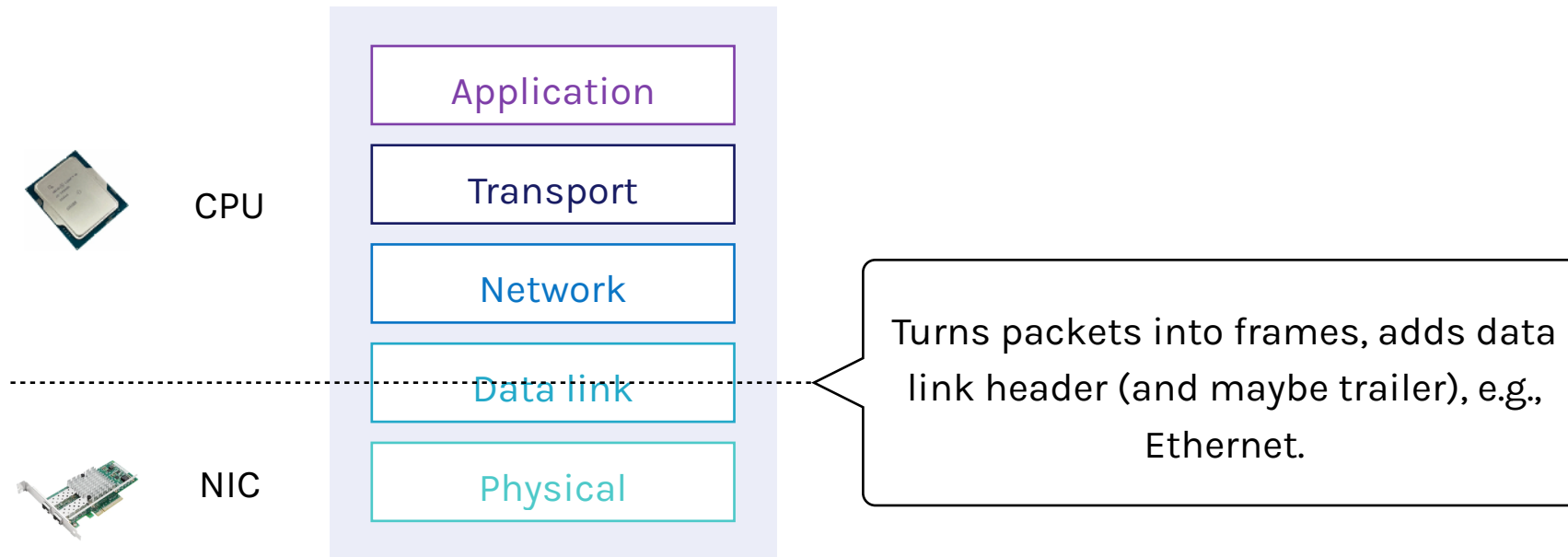
The Linux kernel network stack (simplified)



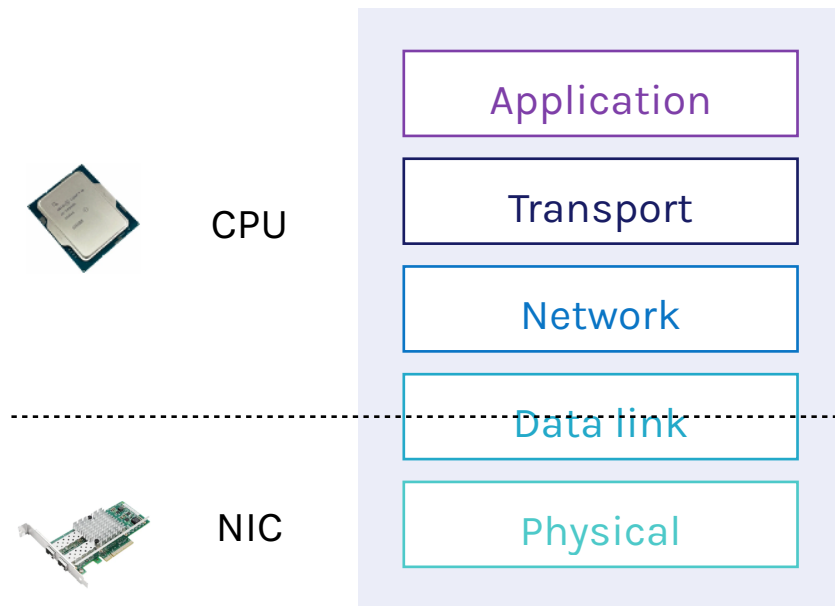
The Linux kernel network stack (simplified)



The Linux kernel network stack (simplified)

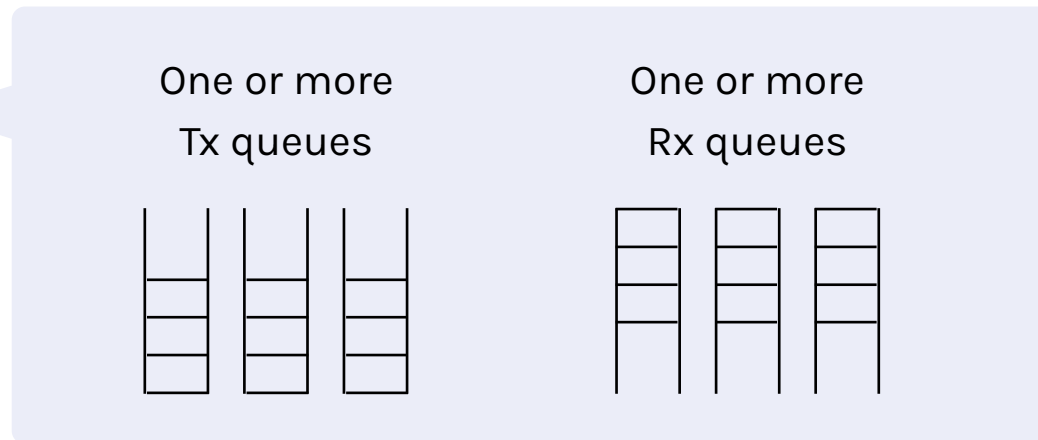


The Linux kernel network stack (simplified)



Packets travel between the NIC and the host through transmit (TX) and receive (RX) queues.

The kernel has scheduling primitives that can be used to influence which packets/flows are prioritized over others.



Modifying the kernel is challenging

Understanding and optimizing the linux kernel network stack is not an easy feat

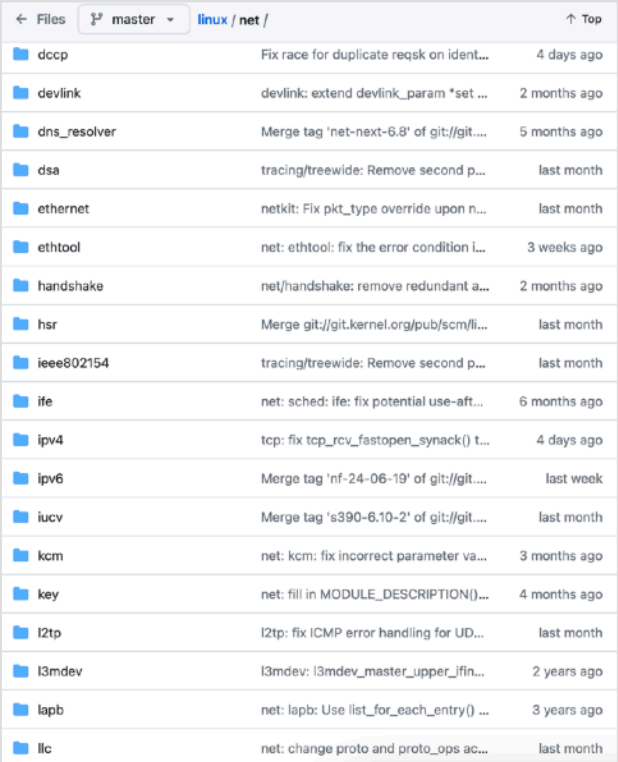
Let alone modifying it to implement new functionality

Even if you figure out where to make changes without breaking anything else, the actual implementation can get challenging

- "computing the cube root function [...] requires using a table lookup and a Newton-Raphson iteration instead of a simple function call."

New functionalities must be approved by Linus Torvalds

<https://github.com/torvalds/linux/tree/master/net>



Files	master	linux / net /	Top
dccp	Fix race for duplicate reqsk on ident...	4 days ago	
devlink	devlink: extend devlink_param *set ...	2 months ago	
dns_resolver	Merge tag 'net-next-6.8' of git://git...	5 months ago	
dsa	tracing/treewide: Remove second p...	last month	
ethernet	netkit: Fix pkt_type override upon n...	last month	
ethtool	net: ethtool: fix the error condition l...	3 weeks ago	
handshake	net/handshake: remove redundant a...	2 months ago	
hsr	Merge git://git.kernel.org/pub/scm/li...	last month	
ieee802154	tracing/treewide: Remove second p...	last month	
ife	net: sched: ife: fix potential use-aft...	6 months ago	
ipv4	tcp: fix tcp_rcv_fastopen_synack() t...	4 days ago	
ipv6	Merge tag 'nf-24-06-19' of git://git...	last week	
iucv	Merge tag 's390-6.10-2' of git://git...	last month	
kcm	net: kcm: fix incorrect parameter va...	3 months ago	
key	net: fill in MODULE_DESCRIPTION()...	4 months ago	
l2tp	l2tp: fix ICMP error handling for UD...	last month	
l3mdev	l3mdev: l3mdev_master_upper_ffin...	2 years ago	
lapb	net: lapb: Use list_for_each_entry() ...	3 years ago	
llc	net: change proto and proto_ops ac...	last month	

How to make the kernel more programmable?

Solution #1: make the kernel more modular

- Identify which parts of the stack need to change more frequently and separate out those parts of the code as a standalone "module"
- Define interfaces for these modules to interact with the rest of the stack/kernel

Examples

- Pluggable TCP congestion control
- Packet scheduling with QDiscs

Problems: reliability and security

```
struct tcp_congestion_ops {
    unsigned long flags;

    /* return slow start threshold (required) */
    u32 (*ssthresh)(struct sock *sk);
    /* lower bound for congestion window (optional) */
    u32 (*min_cwnd)(const struct sock *sk);
    /* do new cwnd calculation (required) */
    void (*cong_avoid)(struct sock *sk, u32 ack, u32 in_flight);
    /* call when cwnd event occurs (optional) */
    void (*cwnd_event)(struct sock *sk, enum tcp_ca_event ev);
    /* new value of cwnd after loss (optional) */
    u32 (*undo_cwnd)(struct sock *sk);
    /* hook for packet ack accounting (optional) */
    void (*pkts_acked)(struct sock *sk, u32 num_acked, s32 rtt_us);

    char name[TCP_CA_NAME_MAX];
    struct module *owner;

    /* plus some other functions and fields */
};
```

```
static int bfifo_enqueue(struct sk_buff *skb, struct Qdisc *sch,
                       struct sk_buff **to_free){
    if (likely(sch->qstats.backlog + qdisc_pkt_len(skb) <= sch->limit))
        return qdisc_enqueue_tail(skb, sch);

    return qdisc_drop(skb, sch, to_free);
}

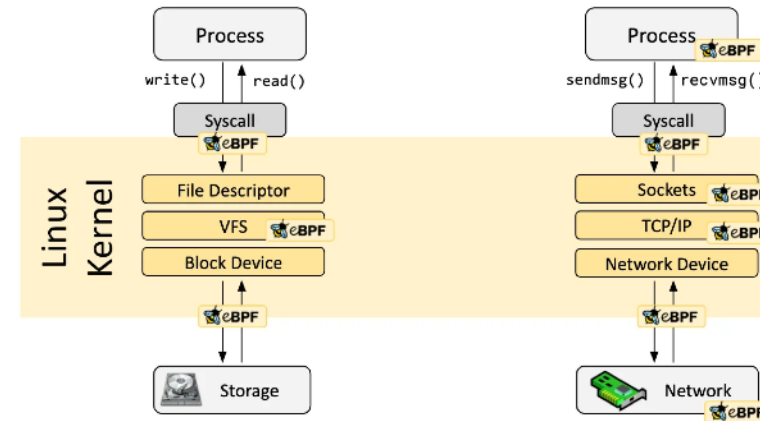
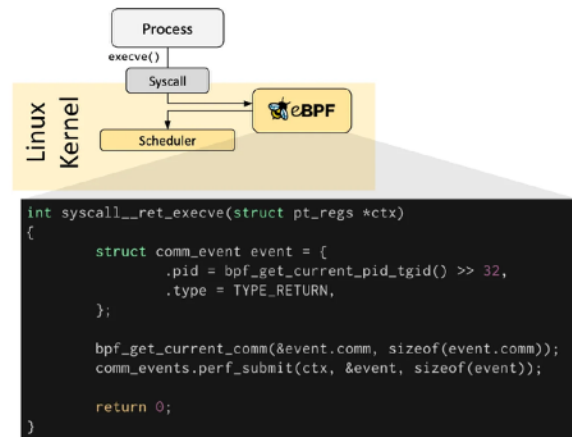
/* definitions of other functions */

struct Qdisc_ops bfifo_qdisc_ops __read_mostly = {
    .id = "bfifo",
    .priv_size = 0,
    .enqueue = bfifo_enqueue,
    .dequeue = qdisc_dequeue_head,
    .peek = qdisc_peek_head,
    .init = fifo_init,
    .destroy = fifo_destroy,
    .reset = qdisc_reset_queue,
    .change = fifo_init,
    .dump = fifo_dump,
    .owner = THIS_MODULE,
};
```

How to make the kernel more programmable?

Solution #2: allow modifications from user space

- eBPF (extended Berkeley Packet Filter)
- Allows you to run your user-space programs in a "sandbox" in certain locations in the kernel
- You can safely and efficiently extend kernel capabilities without having to change the kernel



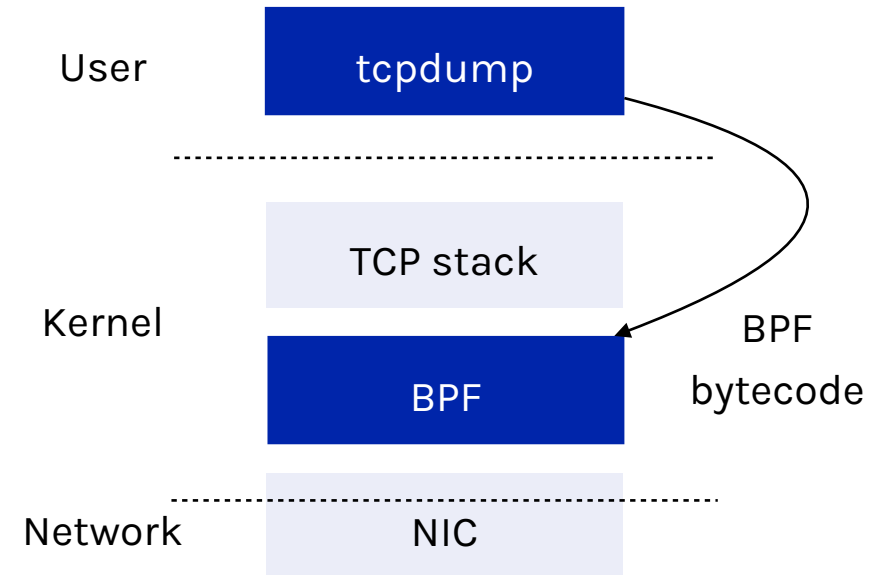
Berkeley Packet Filter (BPF)

A small virtual machine that can run programs injected from the user space in the kernel space without changing/recompiling the kernel code

- First implementation (BPF) → Linux Kernel 3.15 (1992)
- Better known as the packet filter language for **tcpdump**

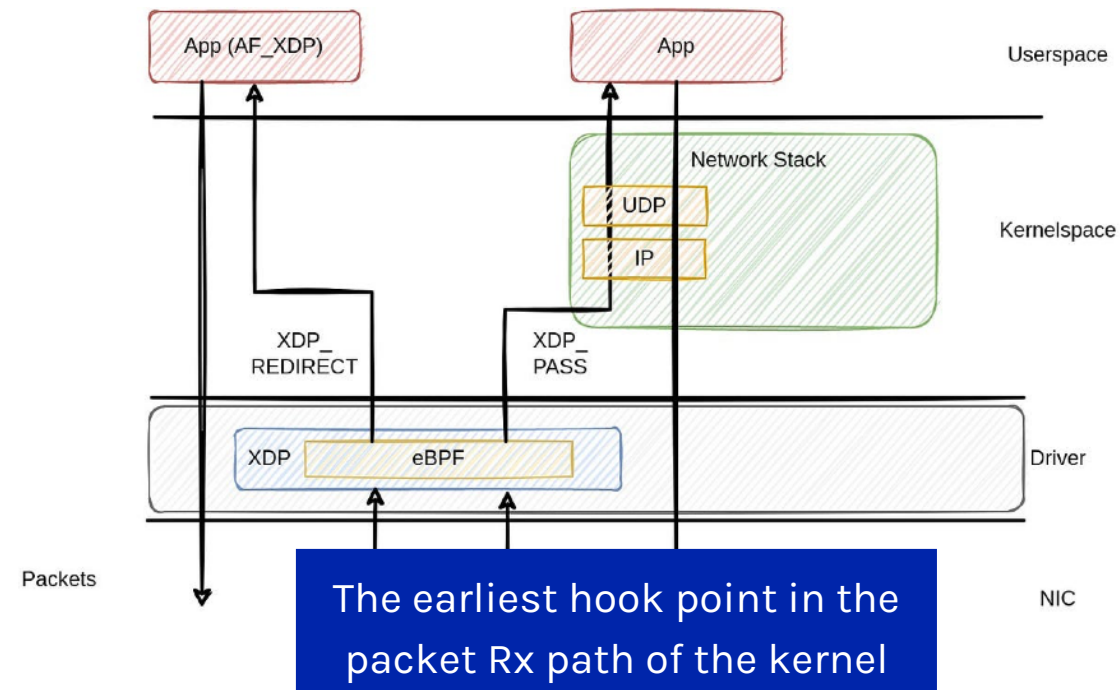
eBPF extends BPF

- 64 bits, 512 bytes stack, Maps (key/value)
- Flexible enough for general processing

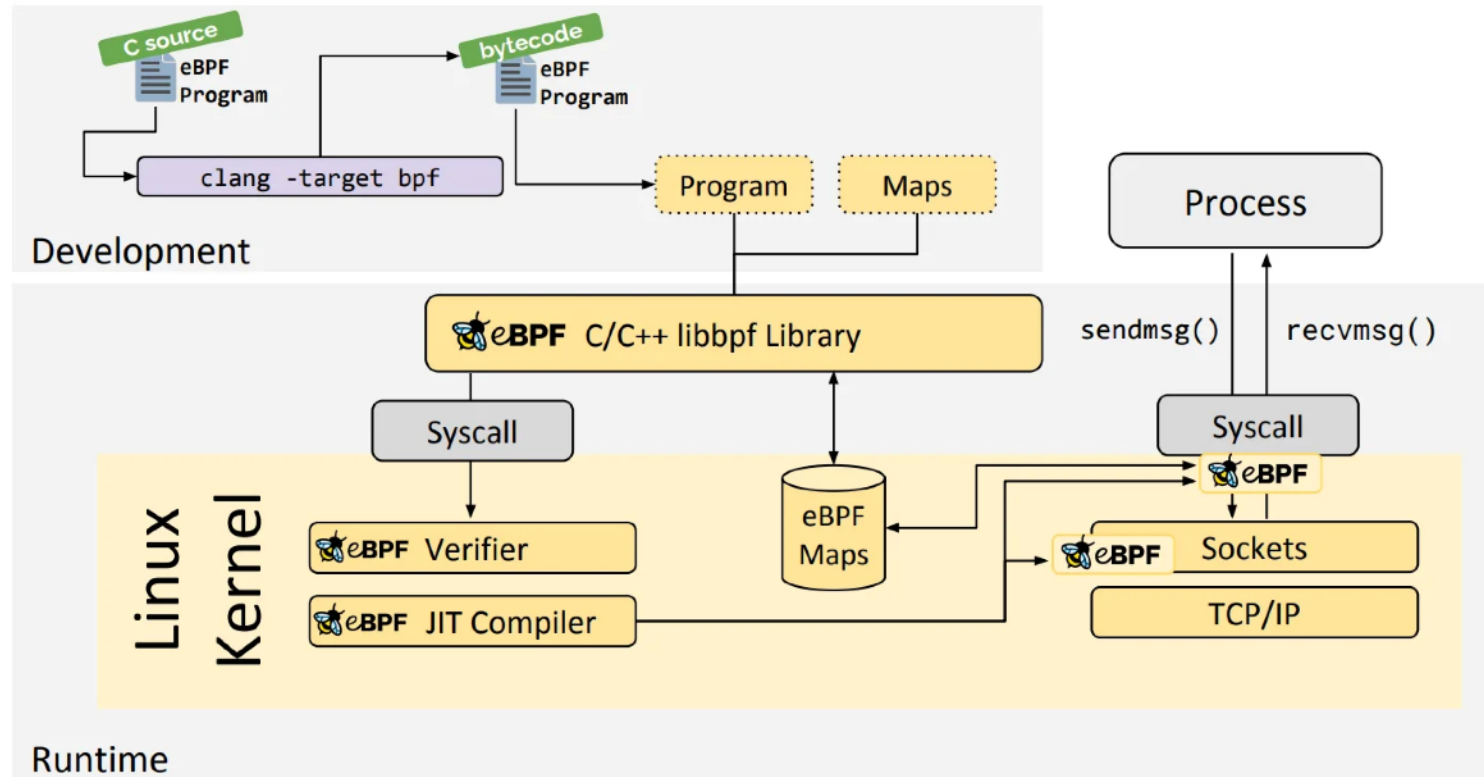


Express Data Path (XDP)

Suitable for high-performance customized packet processing

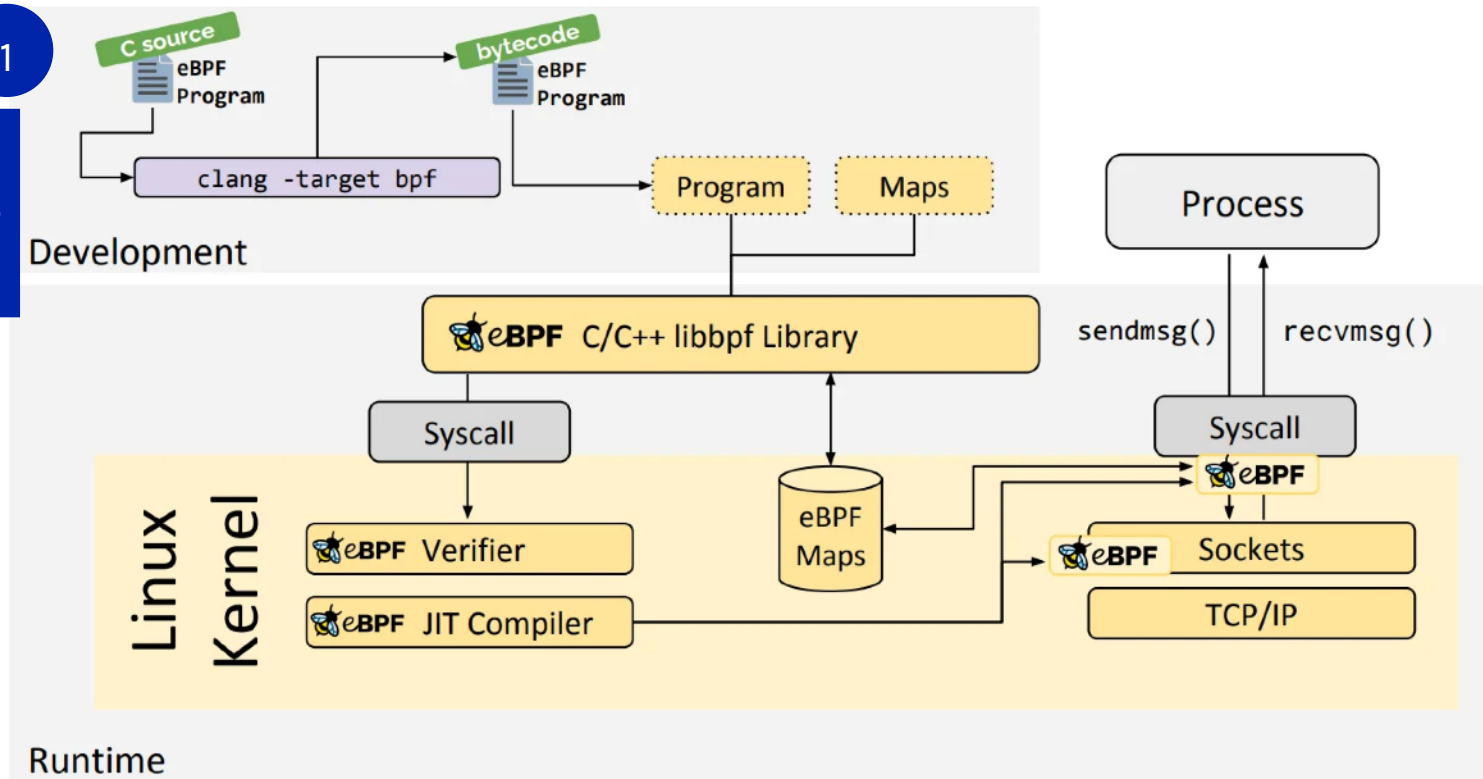


eBPF workflow

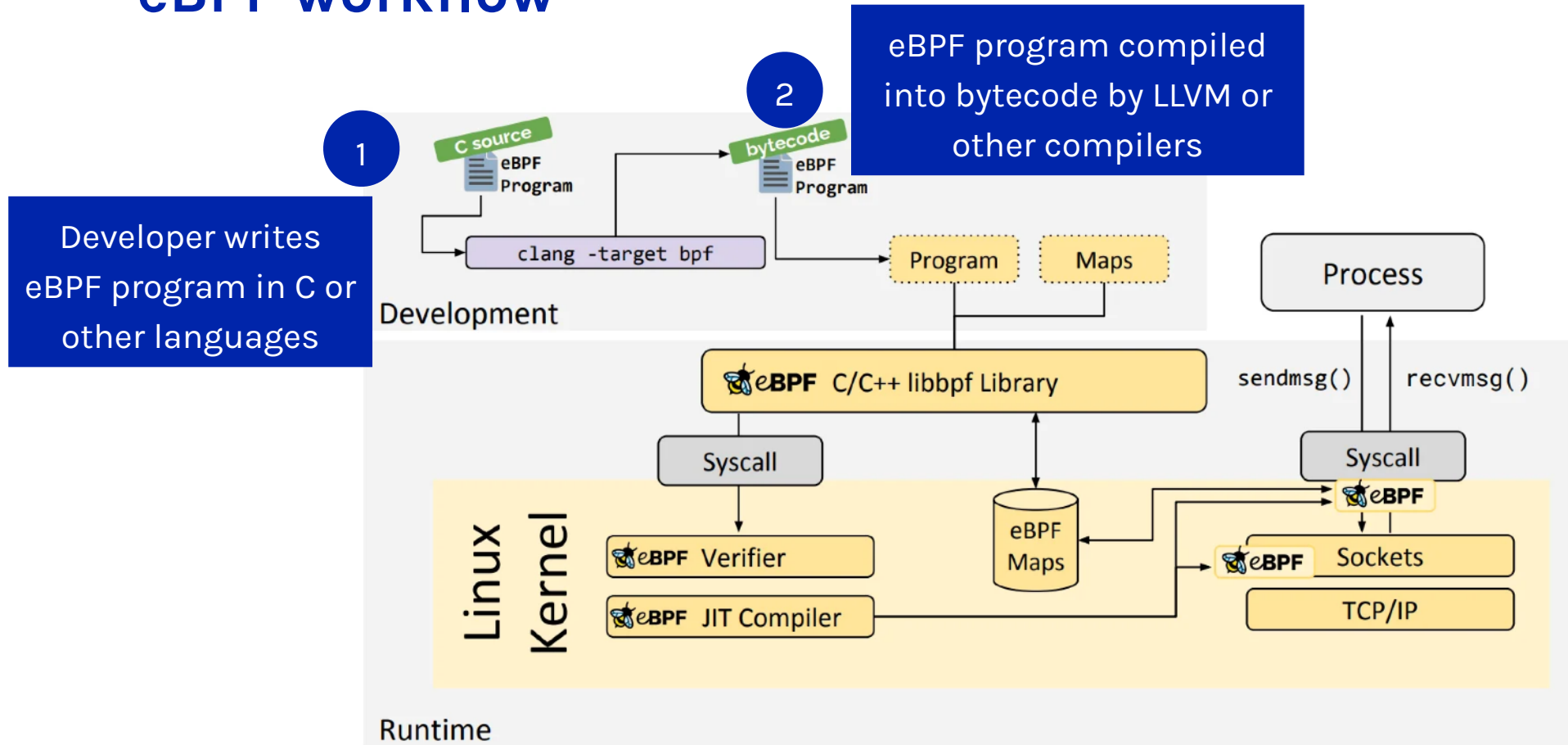


eBPF workflow

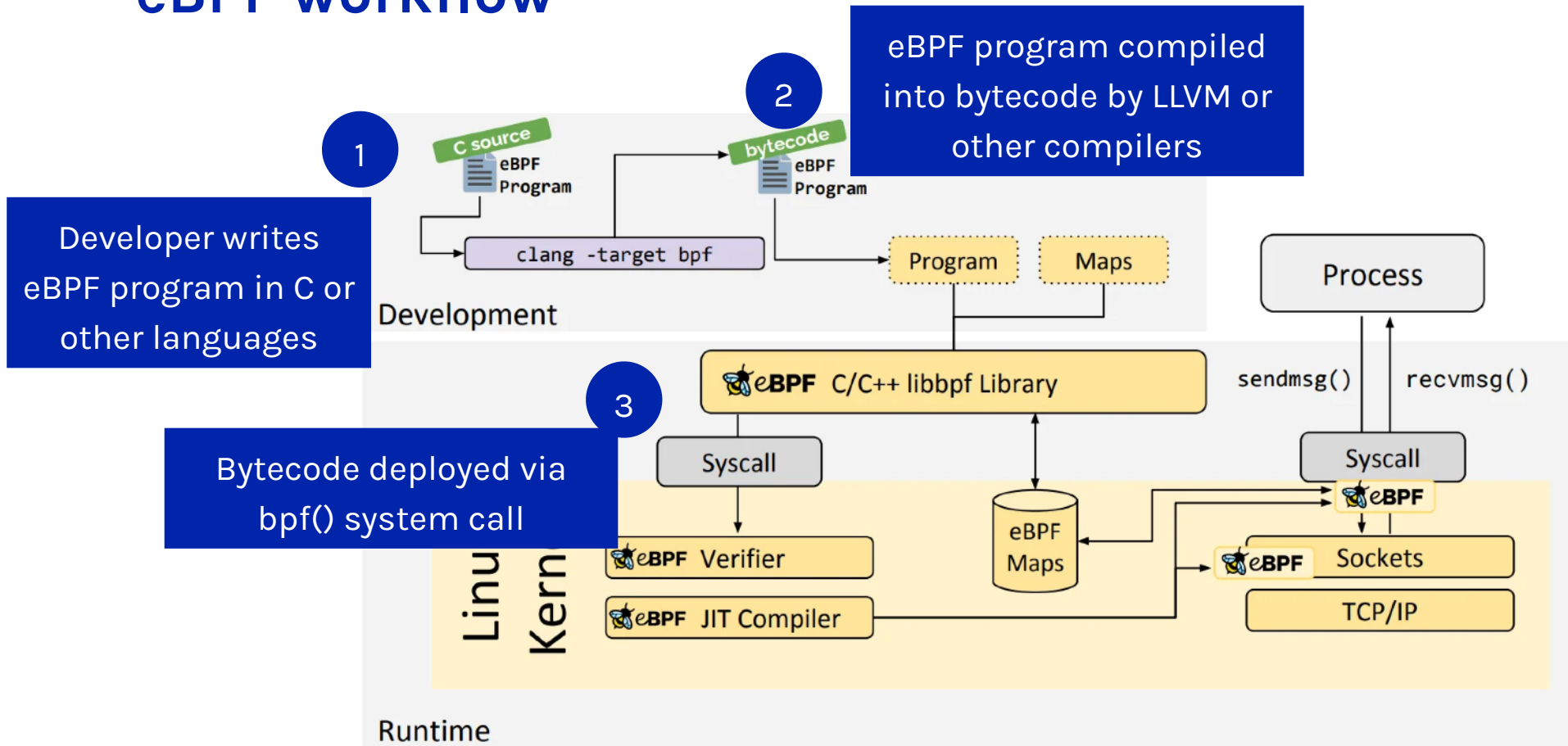
1
Developer writes eBPF program in C or other languages



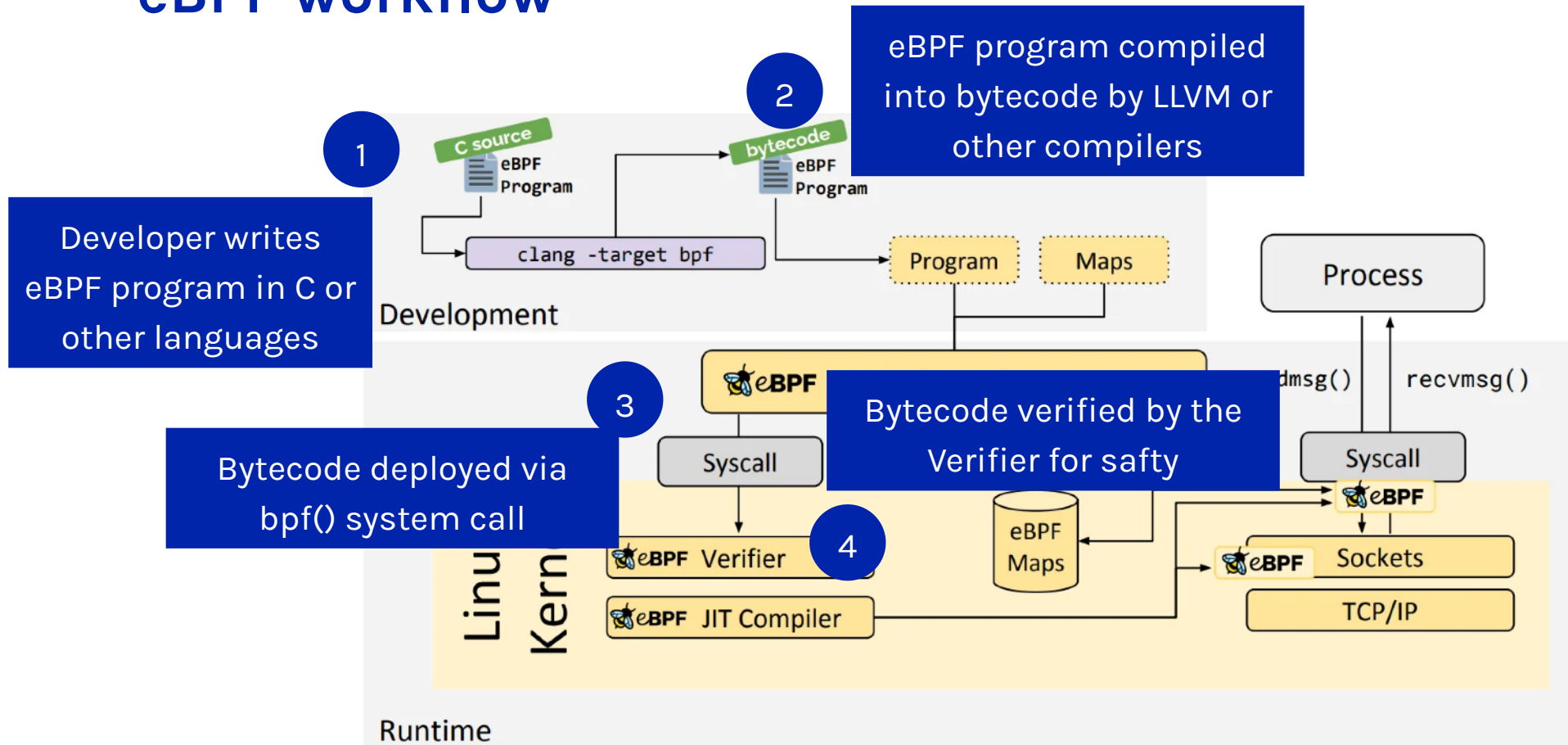
eBPF workflow



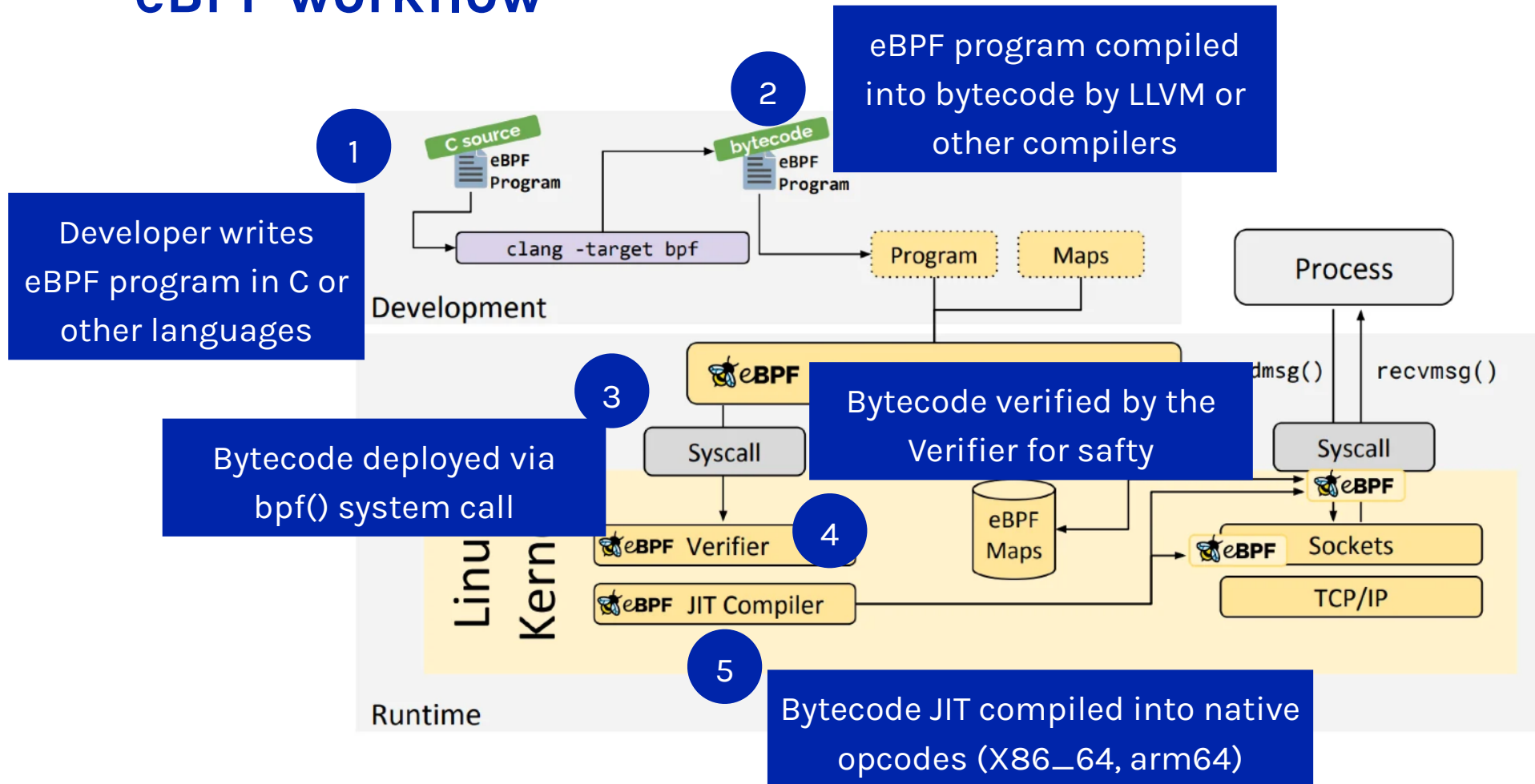
eBPF workflow



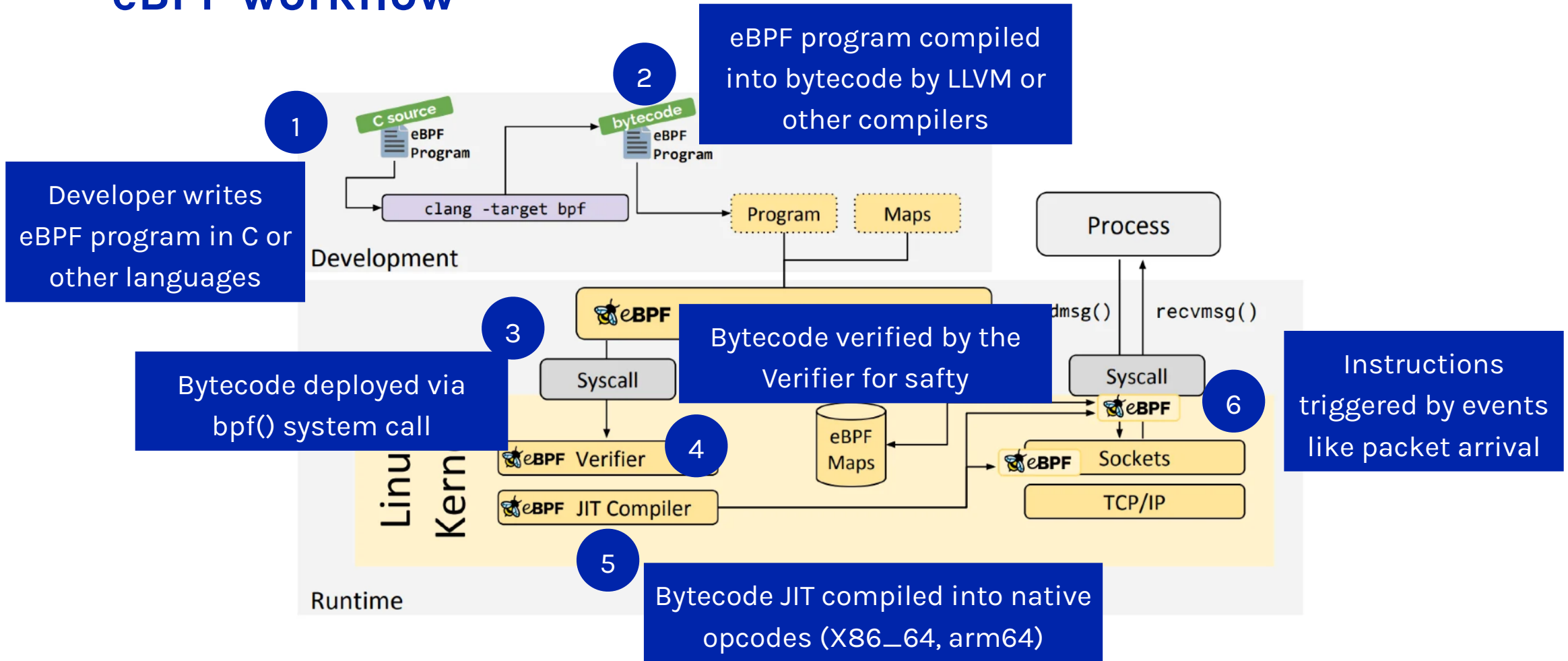
eBPF workflow



eBPF workflow



eBPF workflow



An example eBPF program

```
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <linux/ip.h>
#include <linux/in.h>
#include <bpf/bpf_helpers.h>

SEC("filter")
int xdp_filter_prog(struct xdp_md *ctx) {
    void *data_end = (void *) (long) ctx->data_end;
    void *data = (void *) (long) ctx->data;
    struct ethhdr *eth = data;

    // Ensure the packet data is valid
    if (eth + 1 > data_end)
        return XDP_DROP;

    // Check if the packet is IPv4
    if (eth->h_proto == htons(ETH_P_IP)) {
        return XDP_PASS; // Pass the packet
    } else {
        return XDP_DROP; // Drop the packet
    }
}

char _license[] SEC("license") = "GPL";
```

Compile the program

```
clang -O2 -target bpf -c xdp_filter_prog.c
    -o xdp_filter_prog.o
```

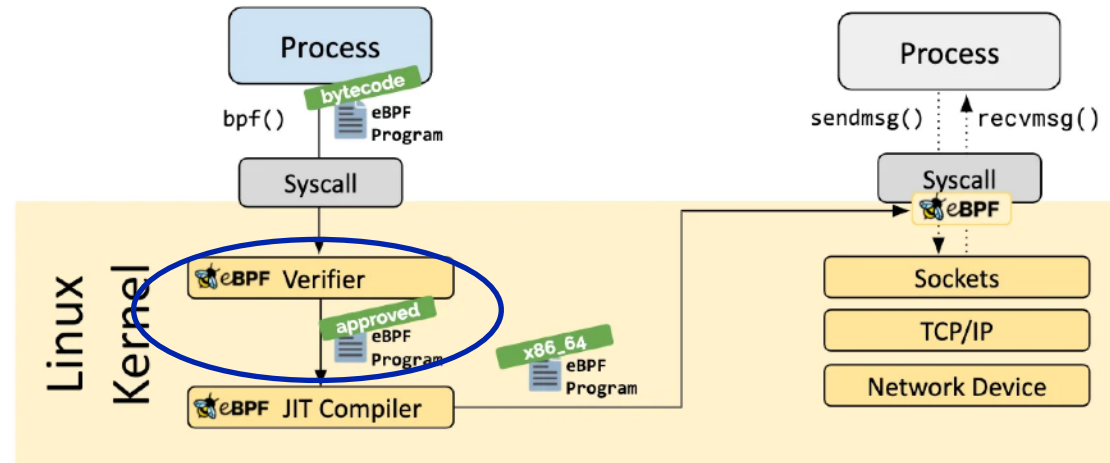
Load the program (e.g., with the bpftool loader or libbpf loader)

```
ip link set dev eth0 xdpgeneric obj
    xdp_filter_prog.o sec filter
```

Check the state of the program

```
bpftool prog show
```


Bytecode verification

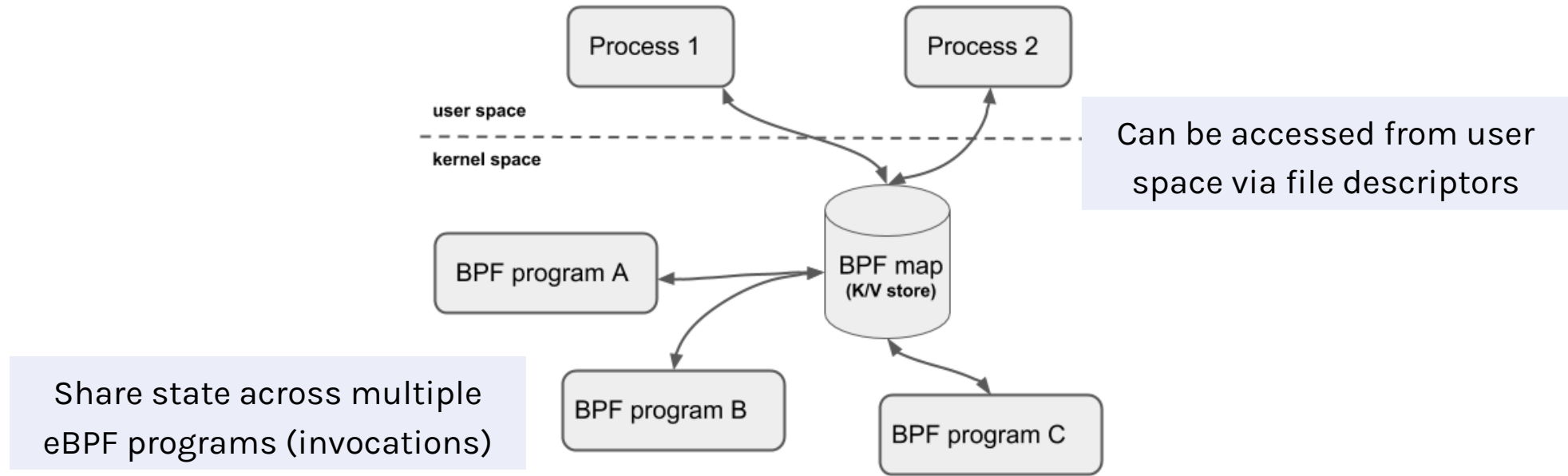


Verification goals

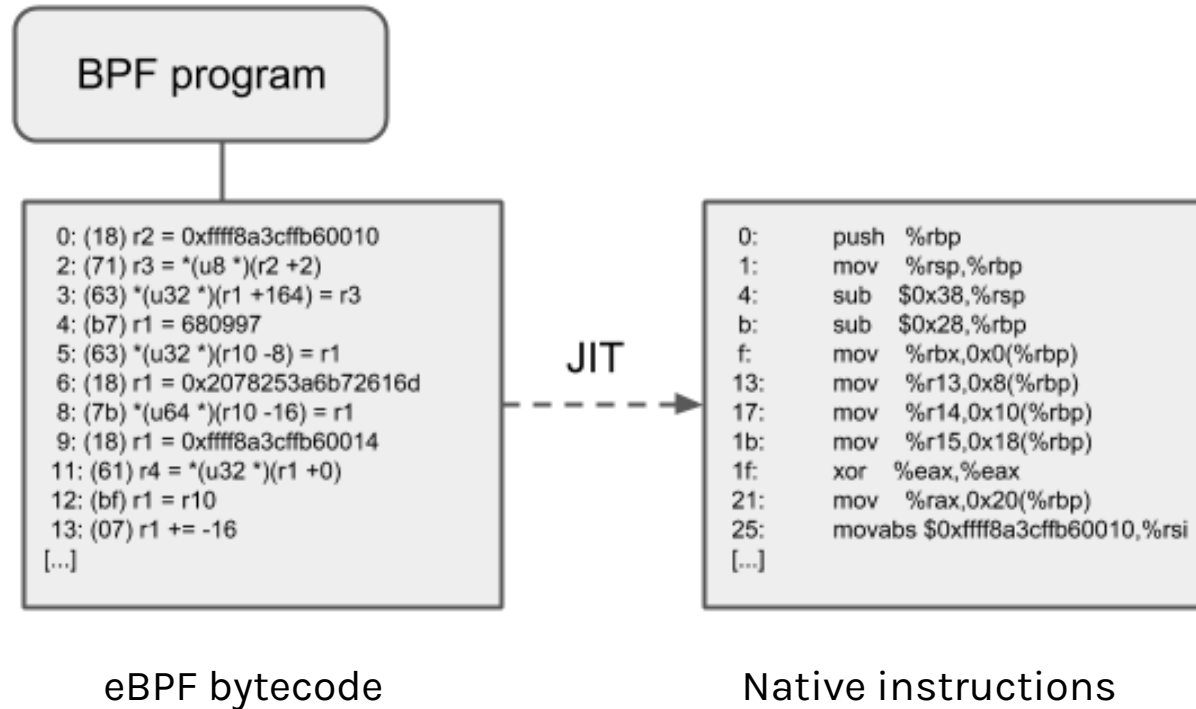
- The process loading the eBPF program has the required capabilities (CAP_BPF) or privileges (root)
- The program does not crash or otherwise harm the system (out-of-bound jumps, memory access)
- The program always runs to completion (i.e., no loop, limited number of instructions)

Maps

Efficient key/value stores in kernel space to keep eBPF state, protected with the read-copy-update (RCU) mechanism for thread safety

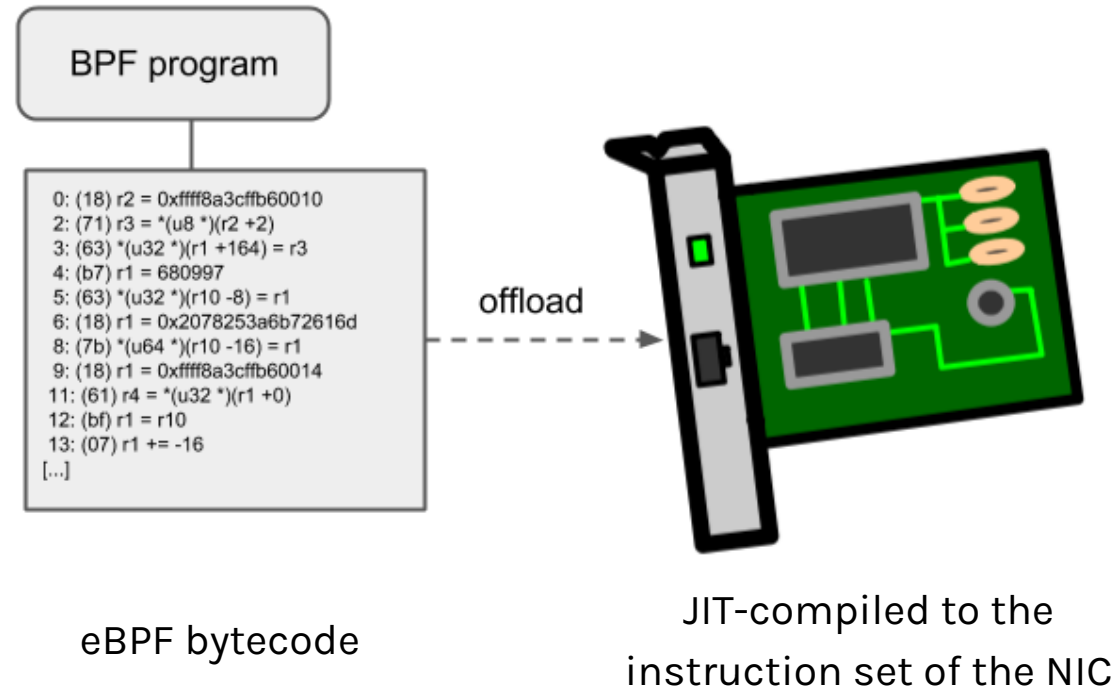


JIT



Reduced per-instruction cost compared to an interpreter, reduced executable image size (more instruction cache friendly), optimized opcodes

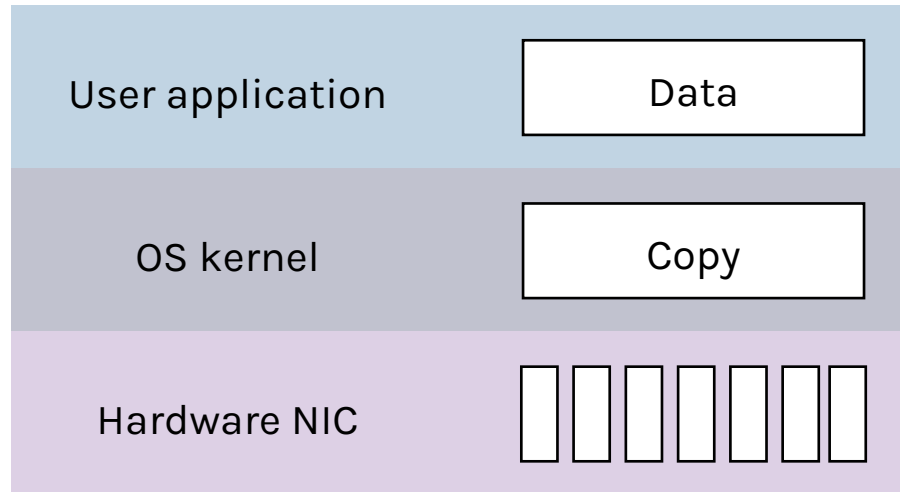
Offloading



Further optimize the performance of eBPF programs by running them on the NIC hardware directly

Kernel Bypassing

Recall traditional network stack

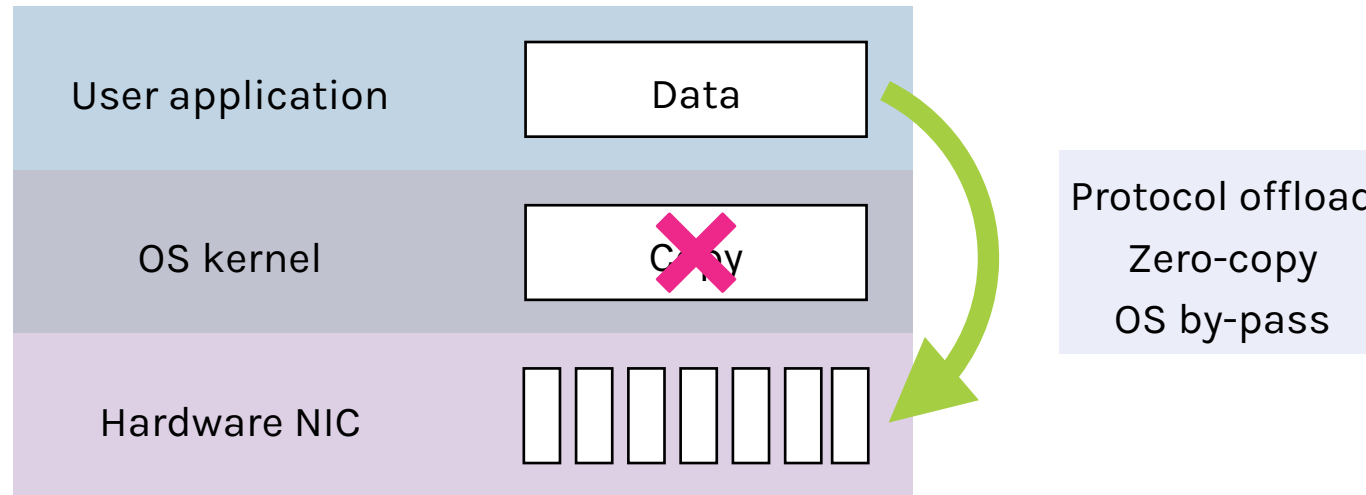


Packet processing in OS incurs high latency, cannot support high throughput, and leads to high CPU utilization

Not acceptable in today's data centers

- Few microseconds of latency
- 10s to 100s Gbps bandwidth
- CPU is costly

Remote direct memory access (RDMA)



Traditionally used in Infiniband clusters for HPC: achieves low latency, high throughput, and negligible CPU utilization

RDMA

Properties

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

Main highlights

- Zero-copy data
- Bypasses the CPU and OS kernel
- Message based transactions

RDMA use cases in data centers

Distributed storage

- Distributed key-value stores
- Distributed file systems
- NVMe over Fabric



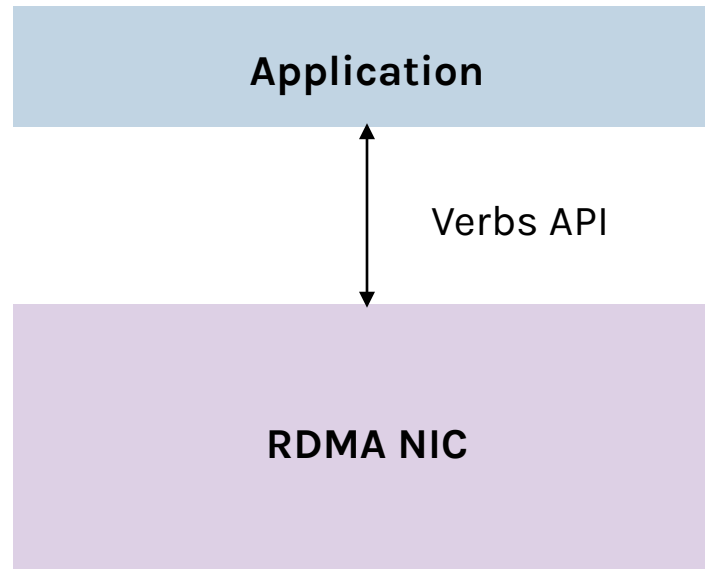
Applications requiring low latency (e.g., search queries)

GPU direct communication (by-pass CPU): machine learning training

Other proposals

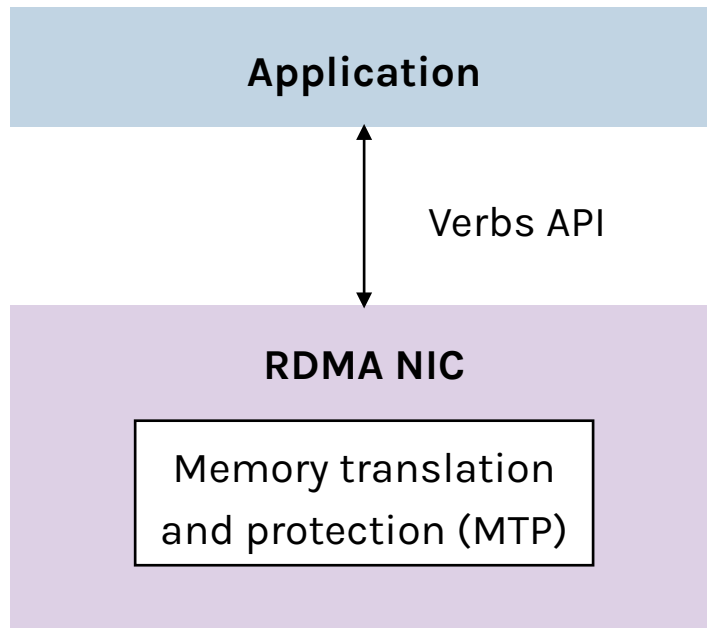
- Resource disaggregation, remote swapping, CPU-free computing

RDMA overview and components



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

Memory translation and protection

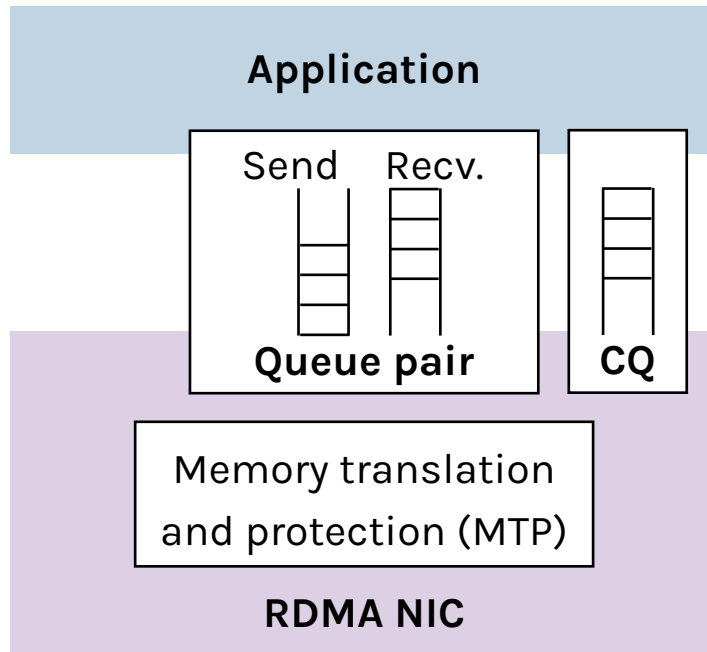


Applications register memory regions with the NIC

Translation: MTP maintains virtual address to physical address mapping

Protection: MTP assigns local and remote access keys to memory region

Queue pairs (QPs)



QPs are interfaces between the application and the NIC

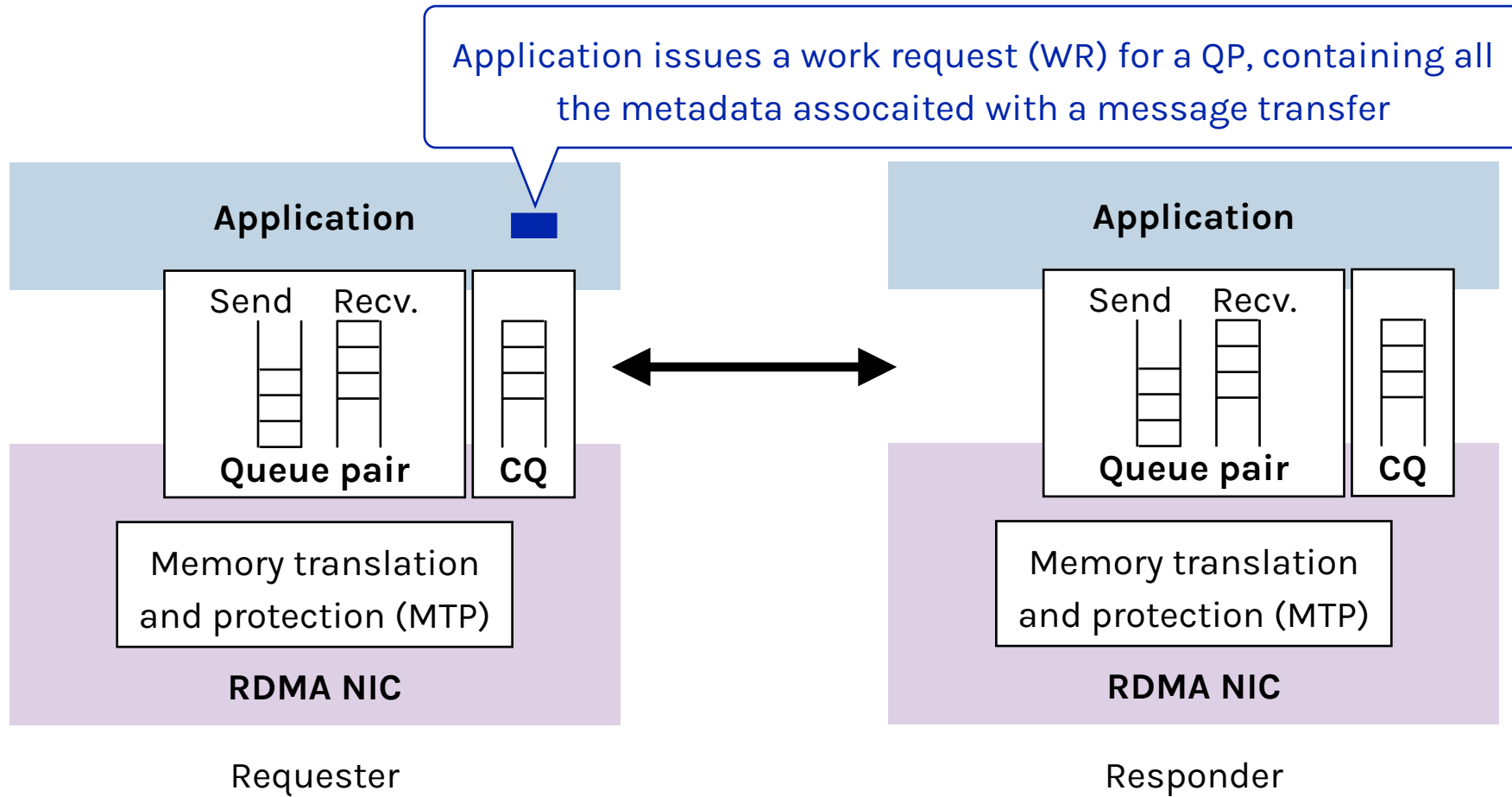
Different types

- Connection-oriented vs. datagram
- Reliable vs. unreliable

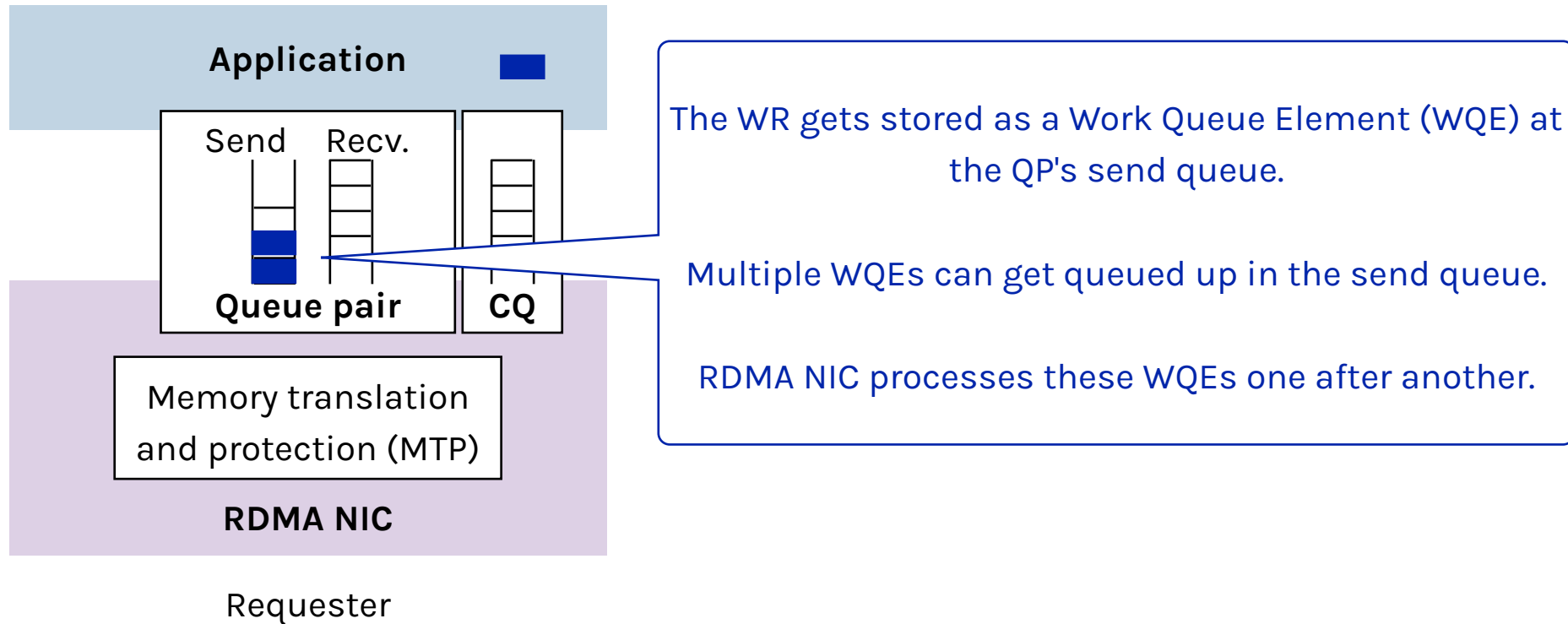
Completion queue

- Notify when the work has been completed

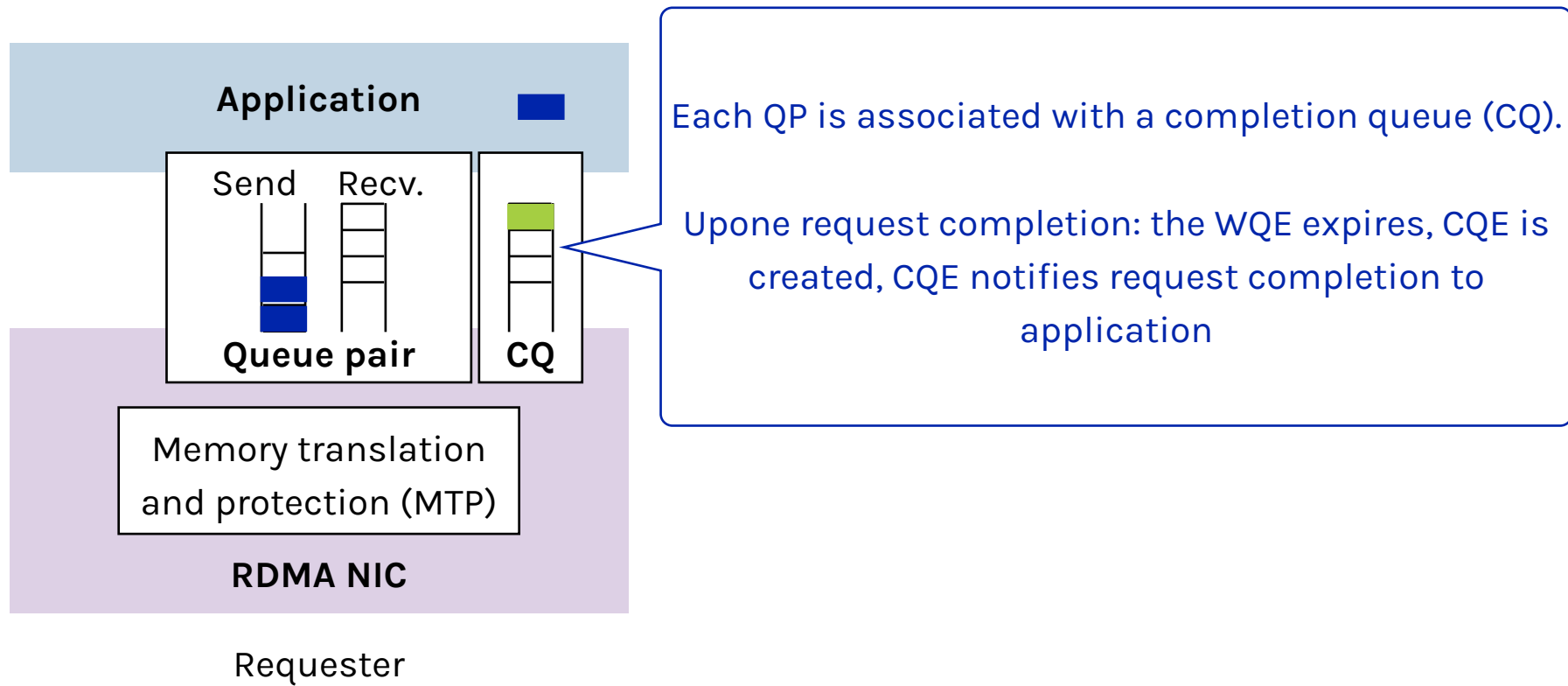
Work requests



Work requests



Work requests



RDMA is just a mechanism

RDMA does not specify the semantics of a data transfer

Two types of memory access models

- One-sided: RDMA read and write + atomic operations
- Two-sided: RDMA send and receive

RDMA send and receive

Traditional message passing where both the source and the destination processes are actively involved in the communication

Both need to have created their queues

- A queue pair of a send and a receive queue, a completion queue for the queue pair

Sender's work request has a pointer to a buffer that it wants to send to

- WQE is enqueued in the send queue

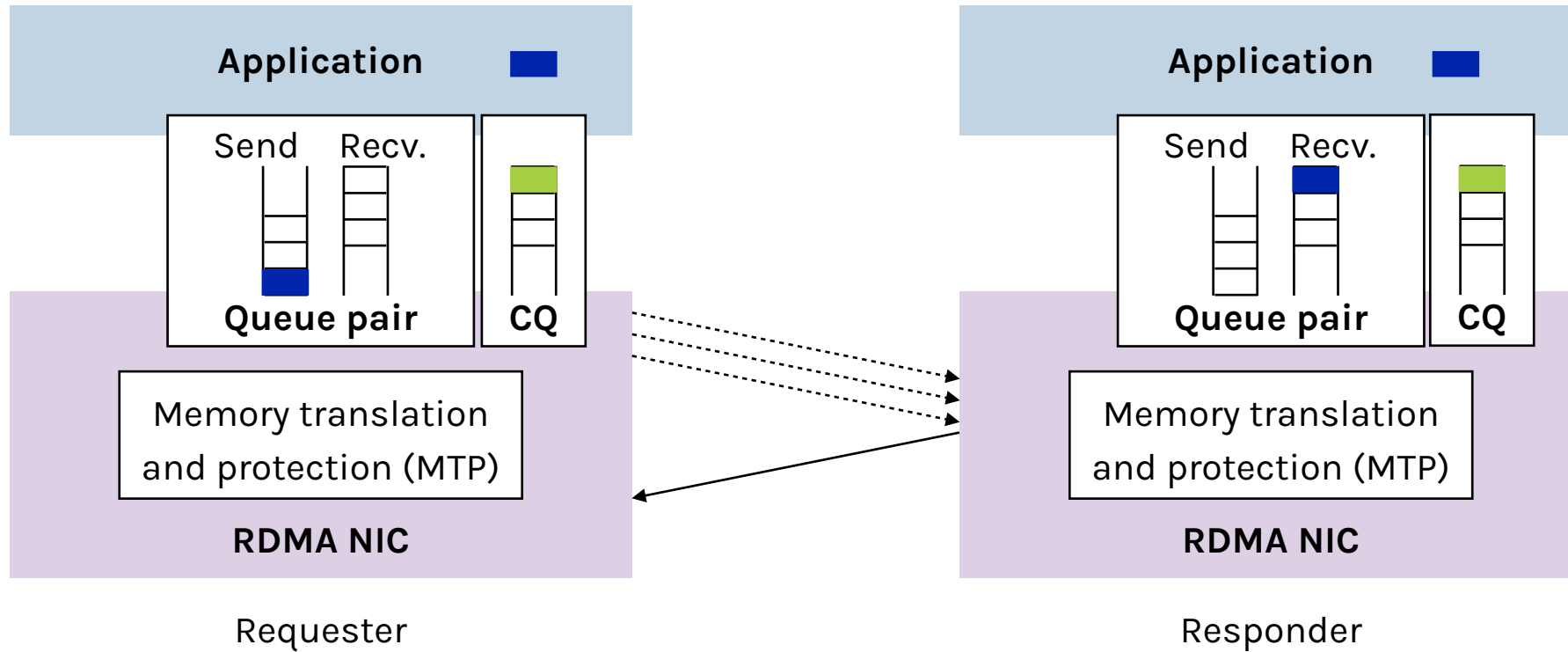
Receiver's work request has a pointer to an empty buffer for receiving the message

- WQE is enqueued in the receive queue

RDMA send and receive

Send WQE metadata: local source virtual address, local key, data length

Receive WQE metadata: local sink virtual address



RDMA read and write

Only the sender side is active; the receiver is passive

- The passive side issues no operation, uses no CPU cycles, gets no indication that a "read" or a "write" happen

To issue an RDMA read and write, the work request must include

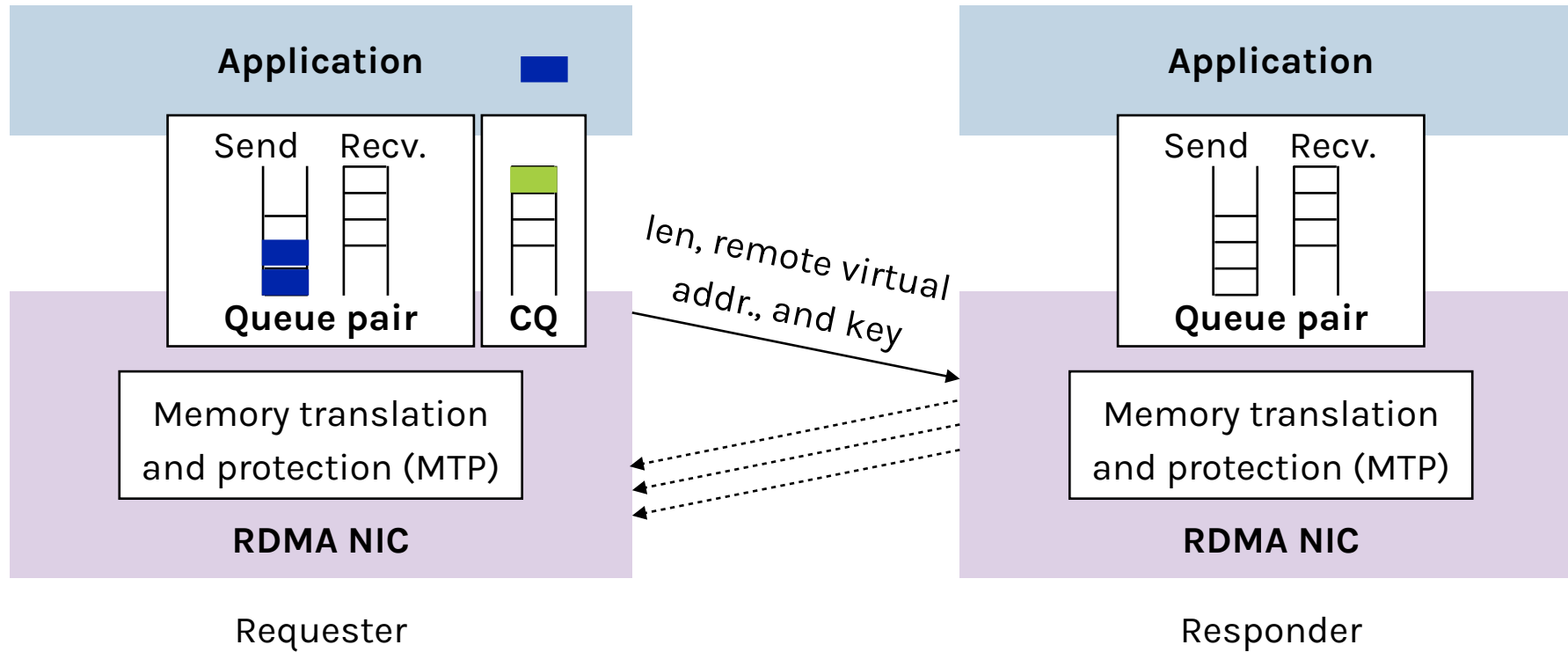
- The remote side's virtual memory address
- The remote side's memory registration key

The active side must obtain the address and key of the passive side beforehand

- Typically, the traditional RDMA send/receive mechanisms are used

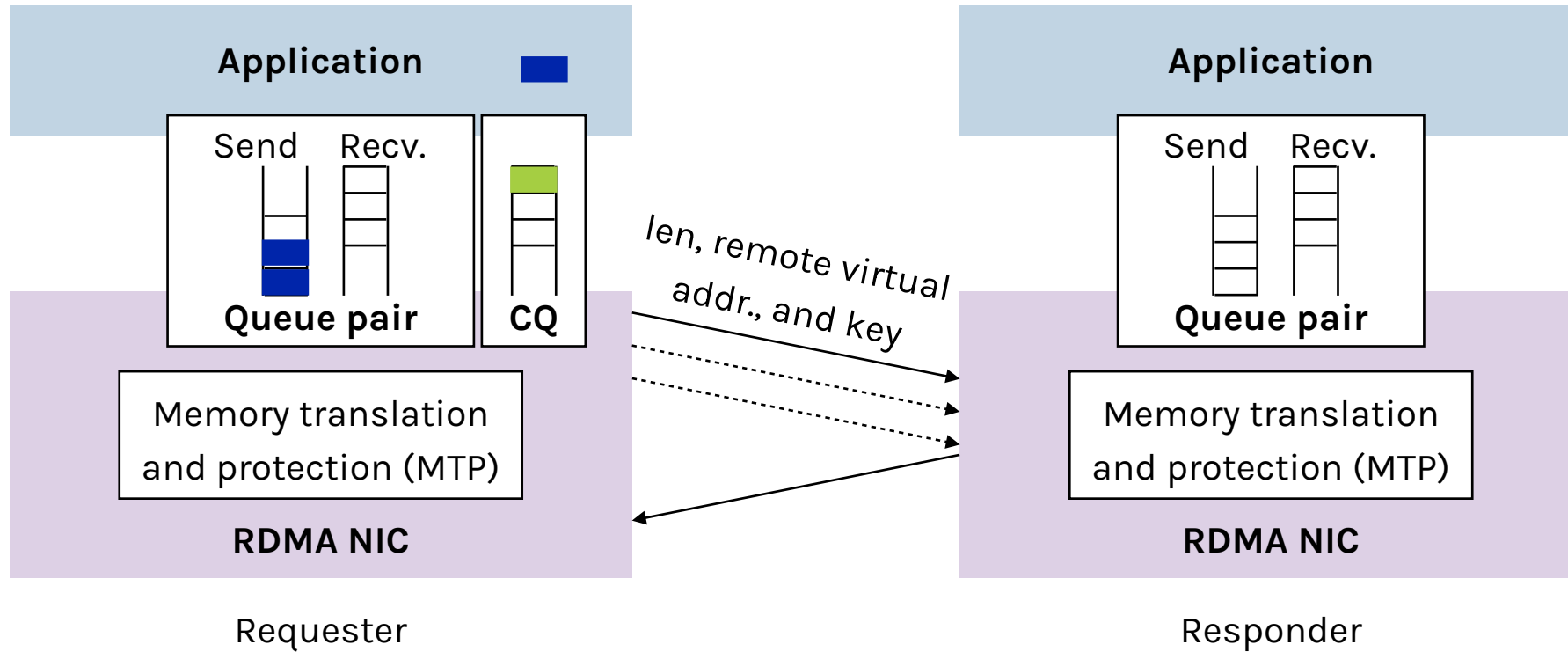
RDMA read

WR/WQE metadata: local sink virtual address, local key, data length, remote source virtual address, remote key



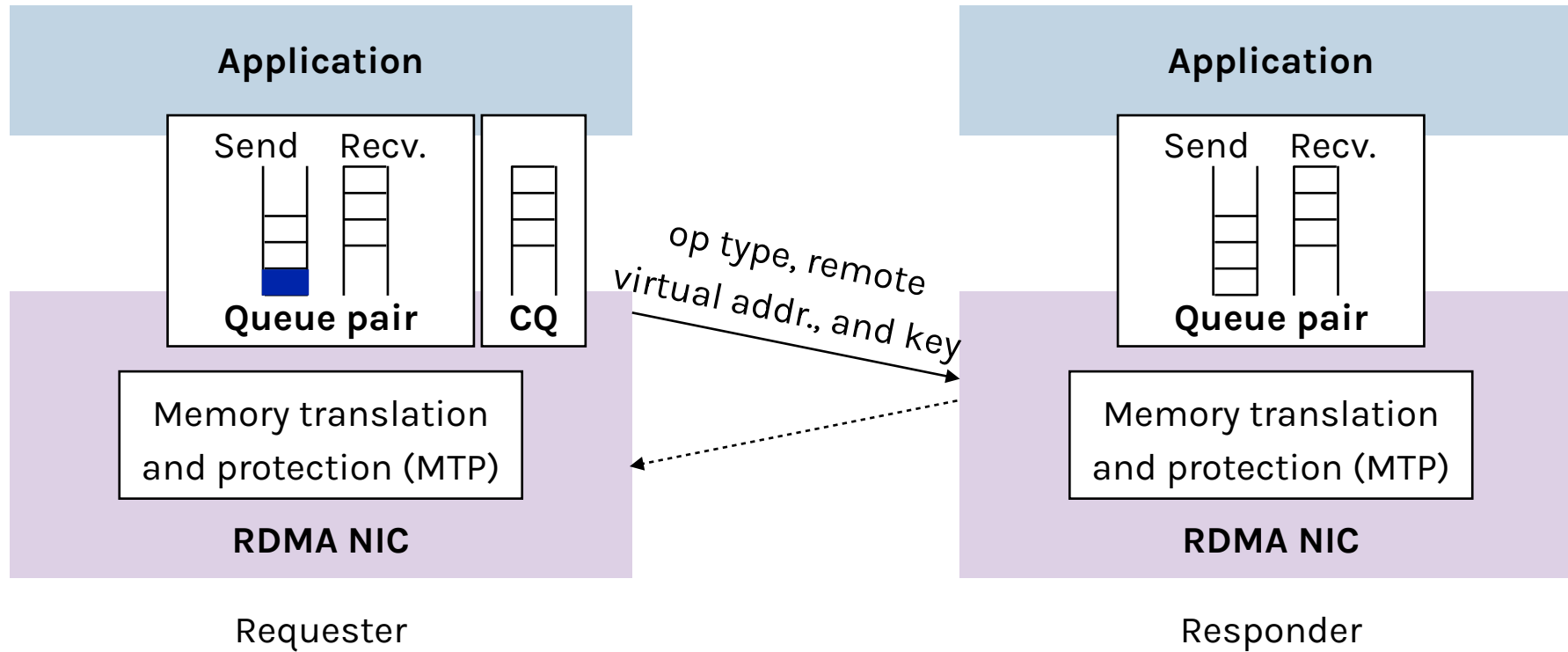
RDMA write

WR/WQE metadata: local source virtual address, local key, data length, remote sink virtual address, remote key



RDMA atomic

WR/WQE metadata: local sink virtual address, local key, atomic operation, remote source virtual address, remote key



Network protocols supporting RDMA

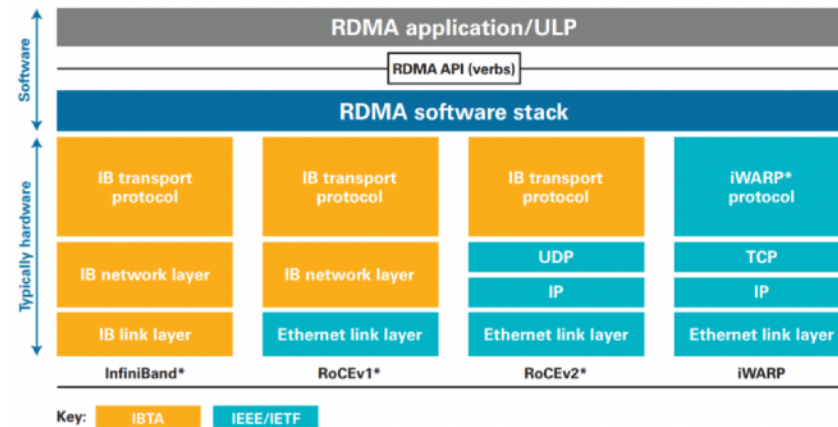
InfiniBand (IB): interconnects for high-performance computers

- Credit-based flow control: PFC (priority flow control), losses are rare
- Transport: discarding out-of-order packets, go-back-N on packet loss

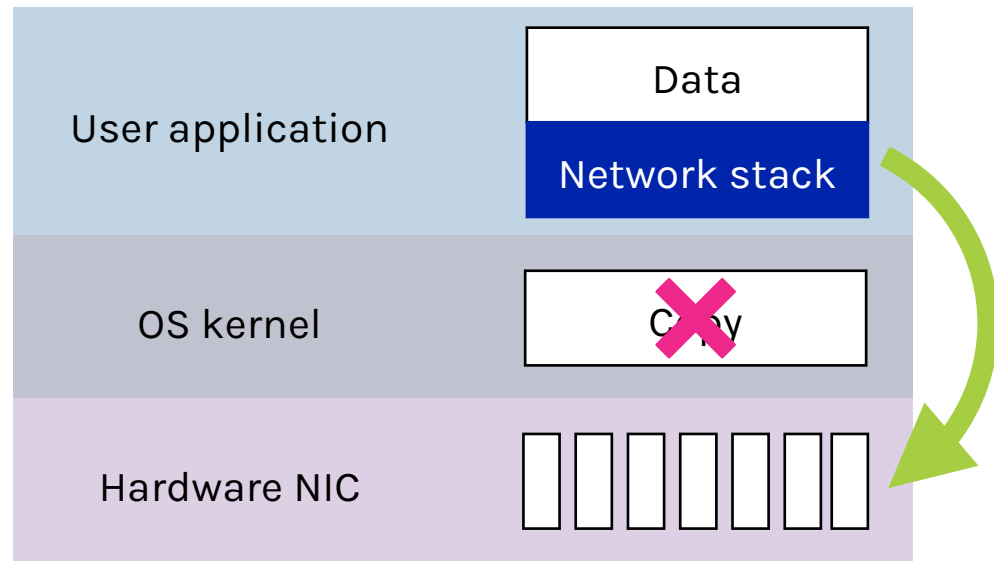
RoCE: RDMA over Converged Ethernet

- Allows running RDMA over Ethernet and IP

iWARP: Internet wide area RDMA protocol



User-space networking



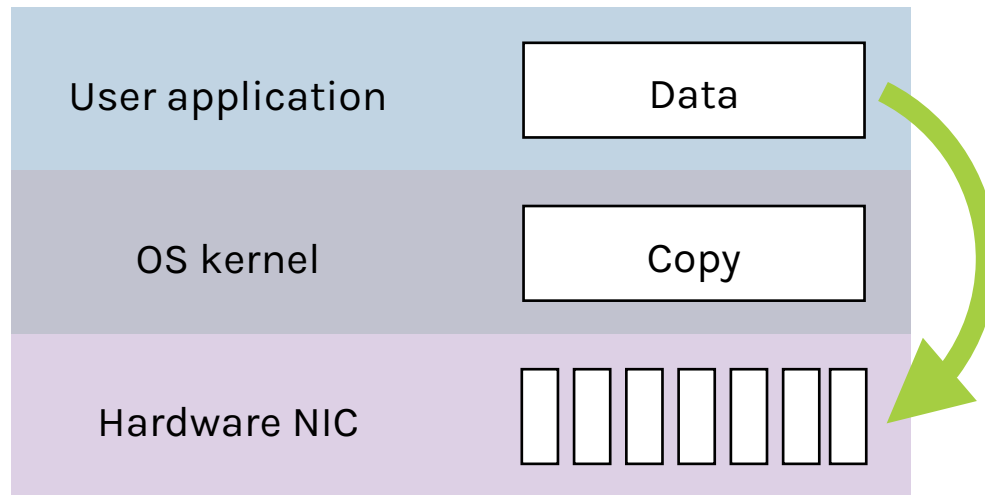
Write all **packet processing** code in a regular program **in user space**

Packets go directly from the NIC to user space (and vice versa) without any interference from the kernel

Kernel-bypass via user-space networking

Example frameworks: DPDK, netmap

Summary

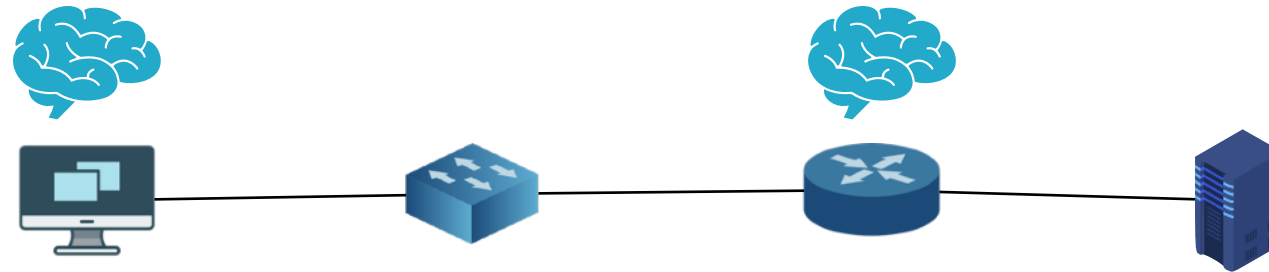


Kernel bypassing: RDMA,
user-space networking

kernel programmability:
eBPF/XDP

Hardware offloading:
SmartNICs

Next time: machine learning for networking



How can we leverage machine learning in networking?