UNIVERSITY OF WATERLOO

CS 856: Programmable Networks

Lecture 5: Programming Software Network Stacks

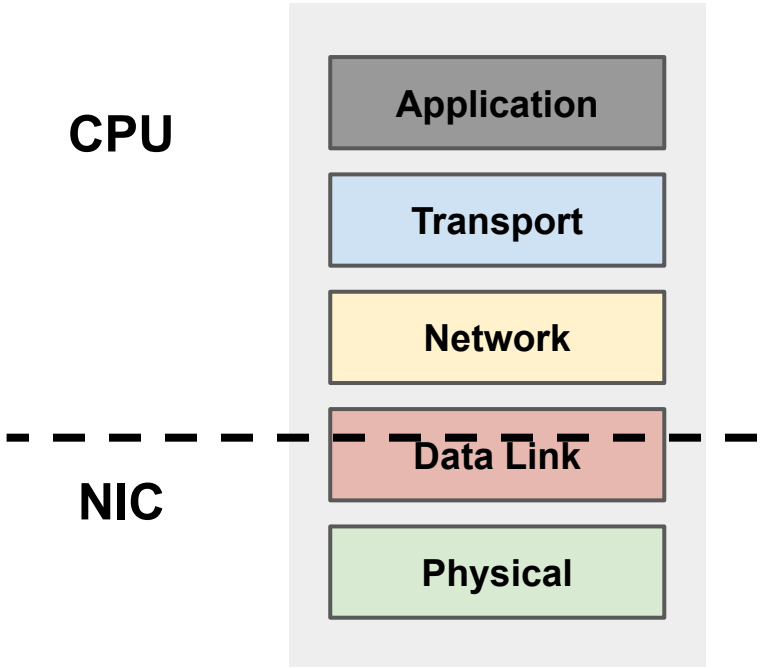
Mina Tahmasbi Arashloo

Winter 2024

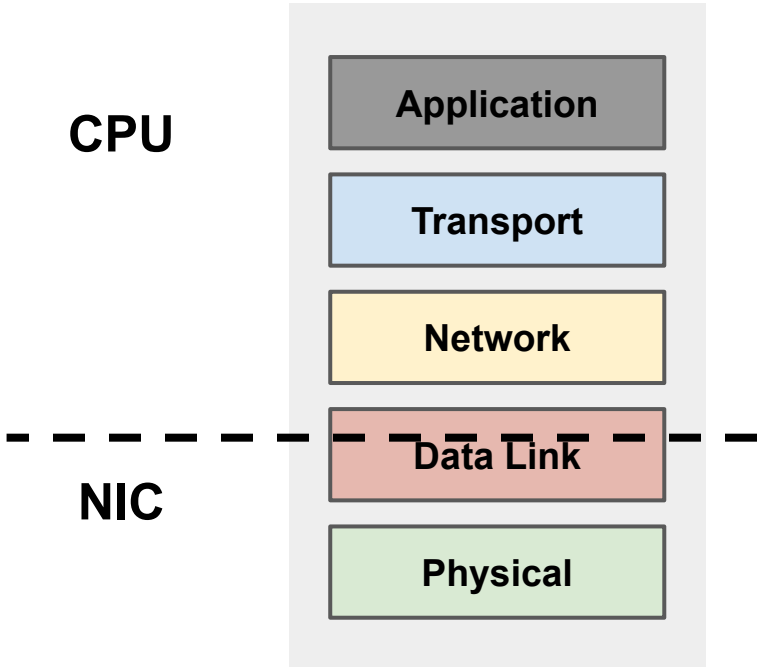
Logistics

- Reviews are due **Monday, Feb 12, at 5pm.**
- Assignment 1 is due **Monday, Feb 26th.**

End-Point network stack



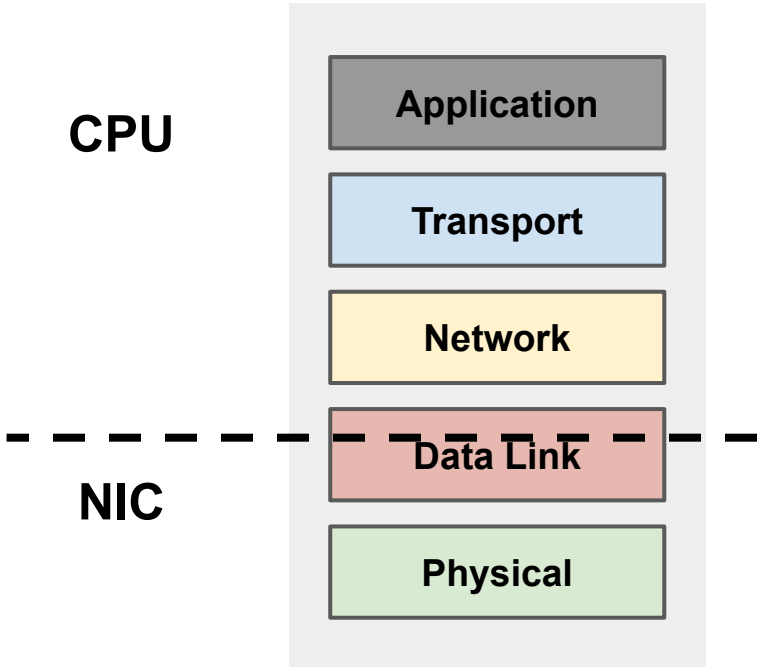
End-Point network stack



On 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 sends them over the link

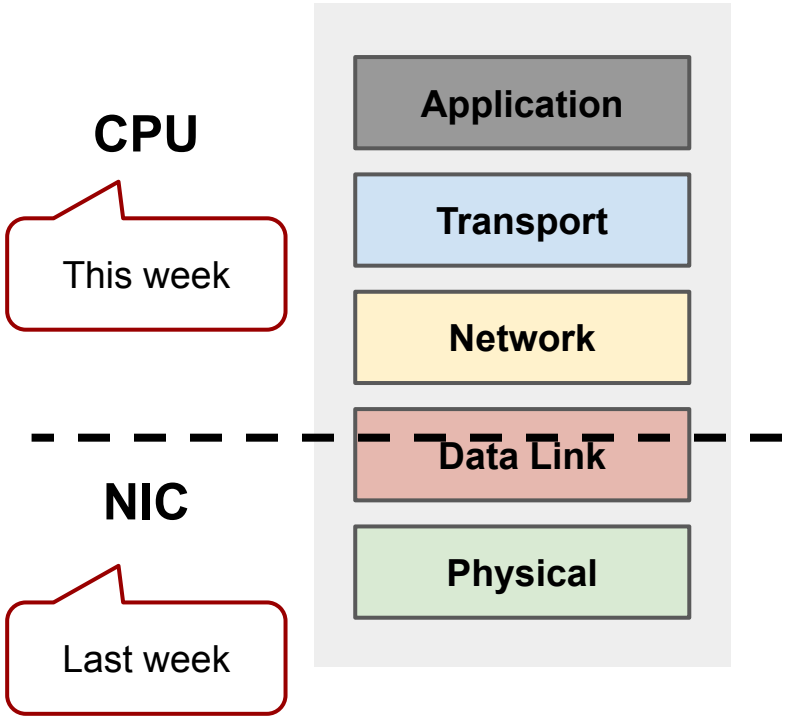
End-Point network stack



On 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

End-Point network stack



On receive (ingress)

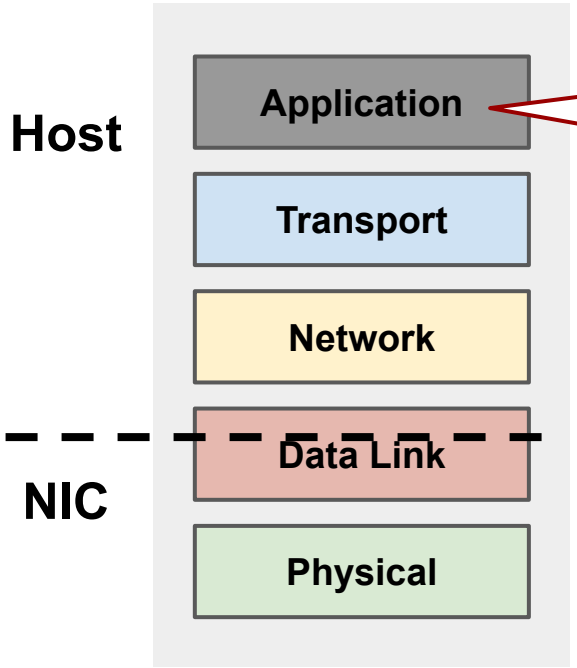
- The NIC turns bits into packets
- Packets are sent to the host over PCIe
- The host CPU processes packets and delivers them to applications

Host Networking

- Changing/customizing end-point packet processing was always technically *possible*.
  - Unlike network switches/routers
  - because it's software
  - no need to go convince a switch vendor to change their hardware/switch OS
- But that doesn't mean it's *easy*.
- Even without programmable NICs, packet processing on end-hosts has grown into a diverse and complex ecosystem.

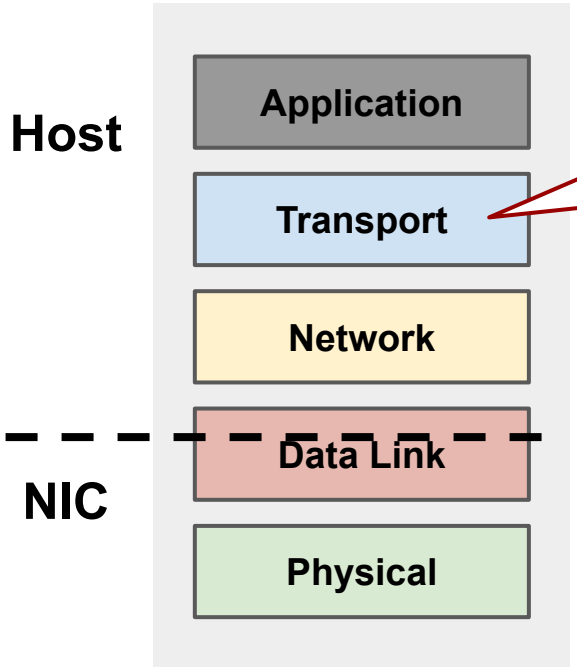
Kernel Packet Processing

The (Linux) kernel network stack (simplified)



Uses system calls to create sockets, write data to them to send to another end-point, read the received data.

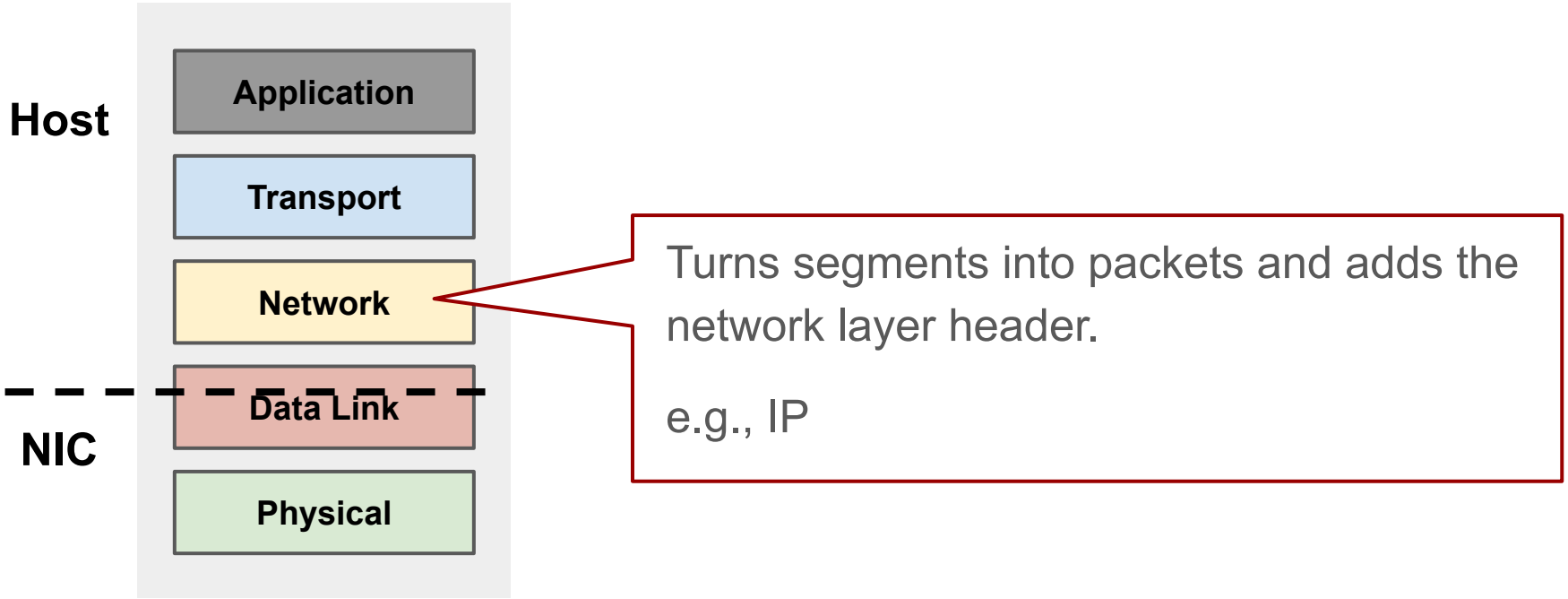
The (Linux) kernel network stack (simplified)



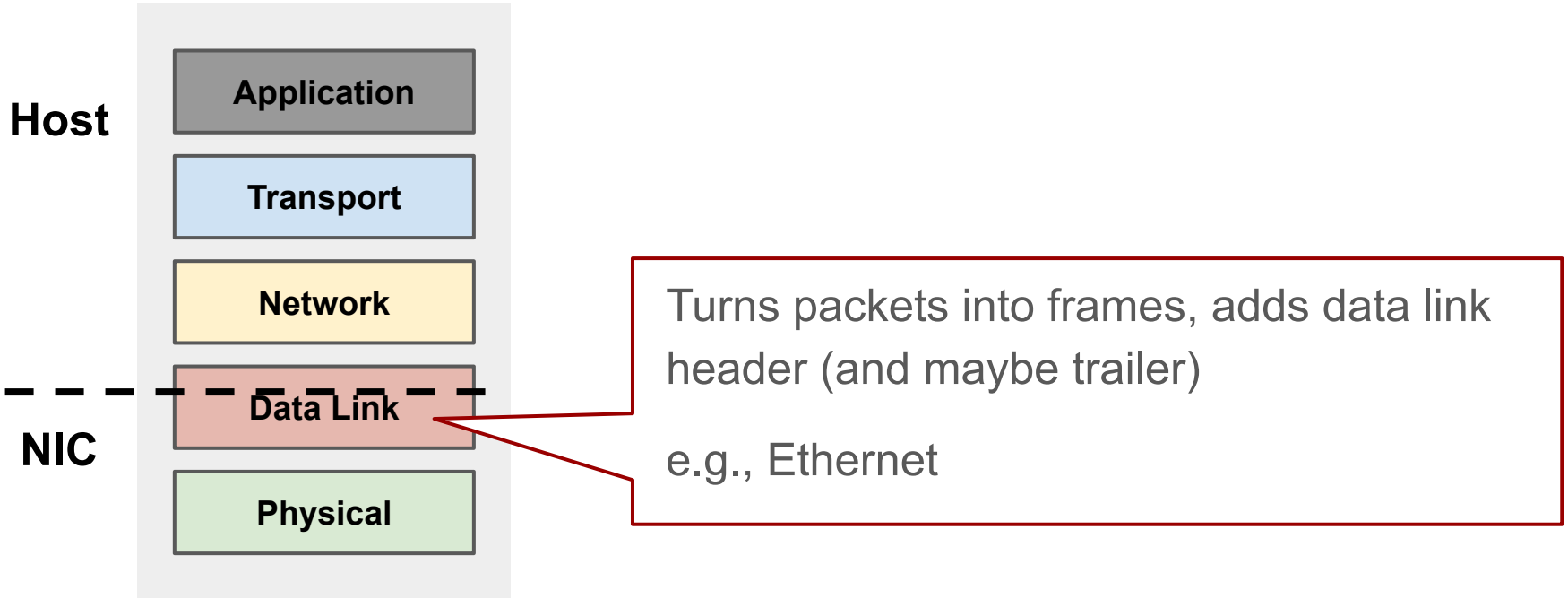
Breaks up the socket data into *segments*, adds the transport layer header

e.g., TCP and UDP

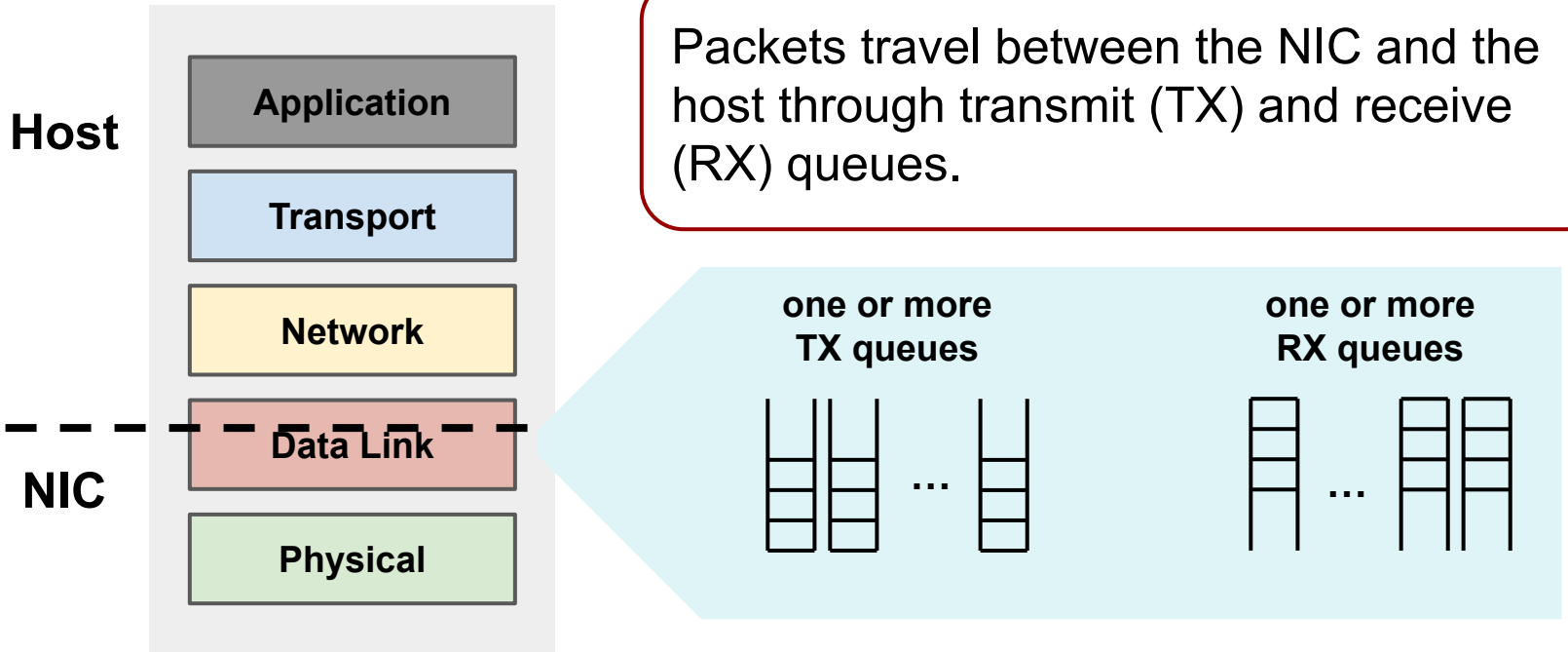
The (Linux) kernel network stack (simplified)



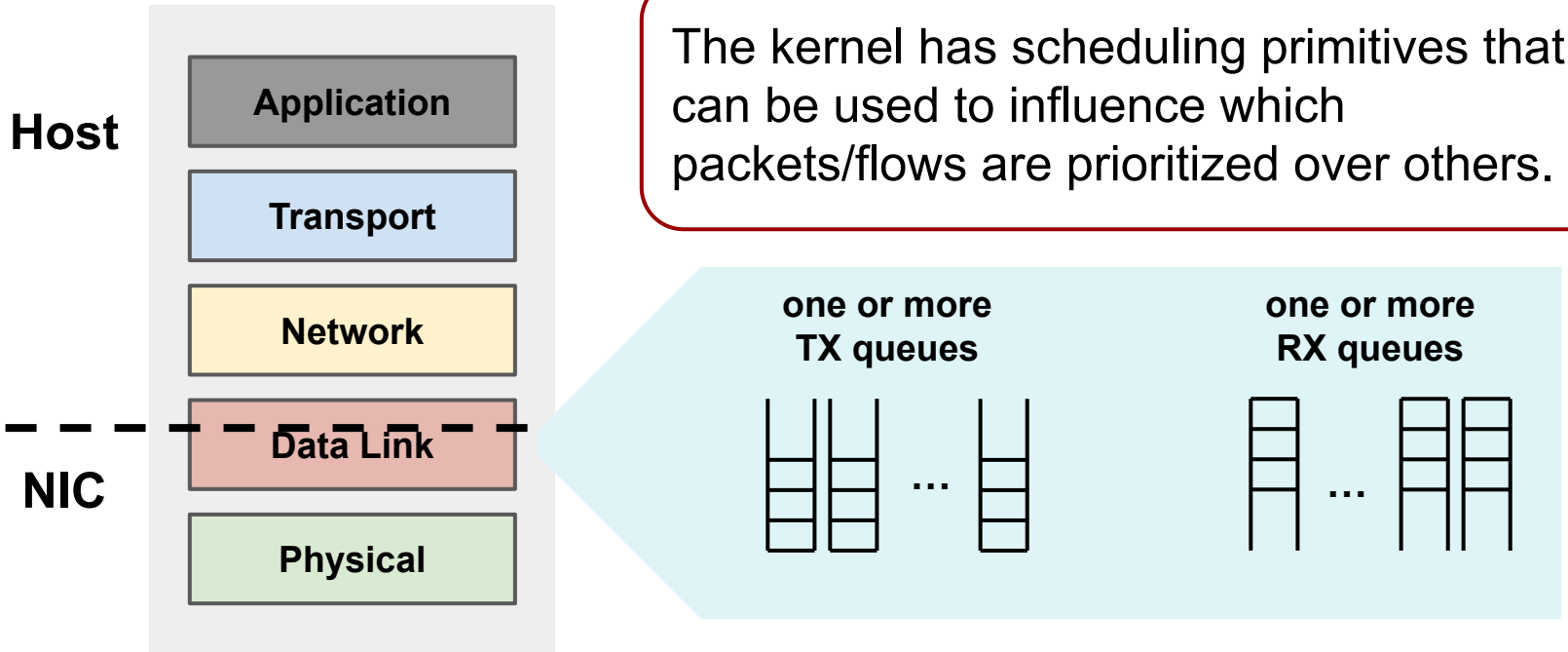
The (Linux) kernel network stack (simplified)



The (Linux) kernel network stack (simplified)

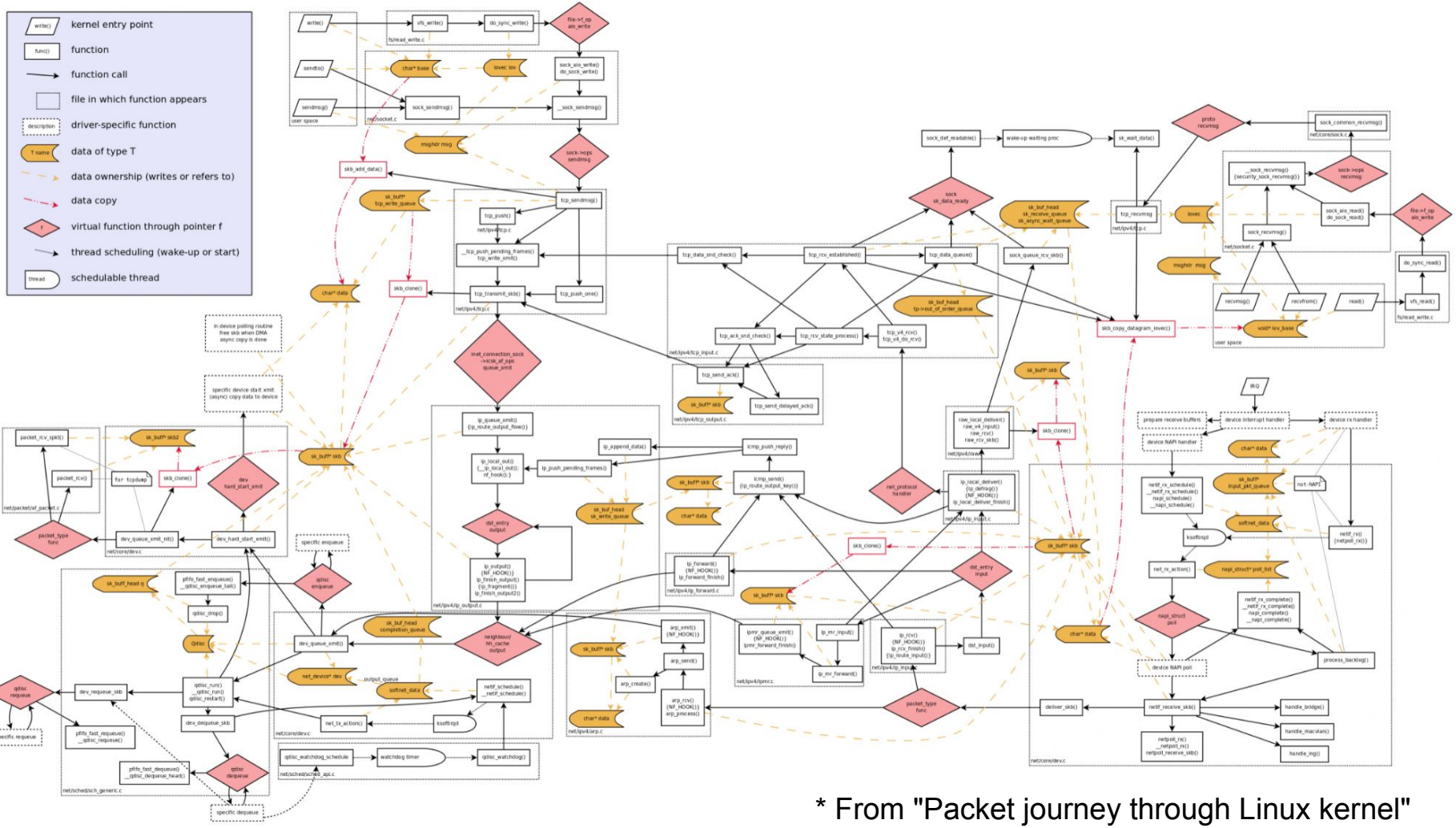


The (Linux) kernel network stack (simplified)



The (Linux) kernel network stack (slightly more realistic)

- The previous slides presented a simplified view
- The reality looks a bit different
- The following figure is a high-level (🙂) diagram of a packet's journey through the Linux kernel.



* From "Packet journey through Linux kernel"

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."

How do we make the kernel "more programmable"?

Solution #1: make it more modular

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

Example 1: Pluggable TCP Congestion Control

```
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 */
};
```

Example 1: Pluggable TCP Congestion Control

```
void tcp_reno_cong_avoid(struct sock *sk, u32 ack, u32 acked)
{ /* … */}


/* Slow start threshold is half the congestion window (min 2) */
u32 tcp_reno_ssthresh(struct sock *sk)
{ /* … */}

u32 tcp_reno_undo_cwnd(struct sock *sk)
{ /* … */}

struct tcp_congestion_ops tcp_reno = {
       .flags        = TCP_CONG_NON_RESTRICTED,
       .name         = "reno",
       .owner        = THIS_MODULE,
       .ssthresh     = tcp_reno_ssthresh,
       .cong_avoid   = tcp_reno_cong_avoid,
       .undo_cwnd    = tcp_reno_undo_cwnd,
};
```

Example 1: Pluggable TCP Congestion Control

```
void tcp_reno_cong_avoid(struct sock *sk, u32 ack, u32 acked)
{ /* … */}

/* Slow start threshold is half th
u32 tcp_reno_ssthresh(struct so
{ /* … */}

u32 tcp_reno_undo_cwnd(struct
{ /* … */}

struct tcp_congestion_ops tcp_
        .flags          = TCP_CONG
        .name           = "reno",
        .owner          = THIS_MOD
        .ssthresh       = tcp_reno_
        .cong_avoid     = tcp_reno_
        .undo_cwnd      = tcp_reno_
};
```

```
void tcp_reno_cong_avoid(struct sock *sk, u32 ack, u32 acked)
{
        struct tcp_sock *tp = tcp_sk(sk);

        if (!tcp_is_cwnd_limited(sk))
                return;

        /* In "safe" area, increase. */
        if (tcp_in_slow_start(tp)) {
                acked = tcp_slow_start(tp, acked);
                if (!acked)
                        return;
        }
        /* In dangerous area, increase slowly. */
        tcp_cong_avoid_ai(tp, tcp_snd_cwnd(tp), acked);
}
```

Example 2: Packet scheduling with QDiscs

```
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 do we 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
- So, you can safely and efficiently extend the capabilities of the kernel without having to change the kernel.

eBPF - Benefits and Challenges

- Much easier to use (compared to kernel programming)!
  - eBPF is like a virtual machine with its own instruction set.
  - You can write C programs, compile them to eBPF, and use the bpf() system call to load them into the kernel.
- Several restrictions on the program to ensure it can run safely in the kernel
  - e.g., on program size, data structures, available libraries and functions, etc.

Example eBPF "hook": XDP

- XDP stands for eXpress Data Path.
- The hook is right after packets are received by the NIC and right before they enter the kernel network stack.
- After processing packets, you can make one of several decisions about the packet, including but not limited to
  - drop (early filtering)
  - send through the kernel stack (pre-processing)
  - send directly to the user-space buffers (kernel bypass)
  - …

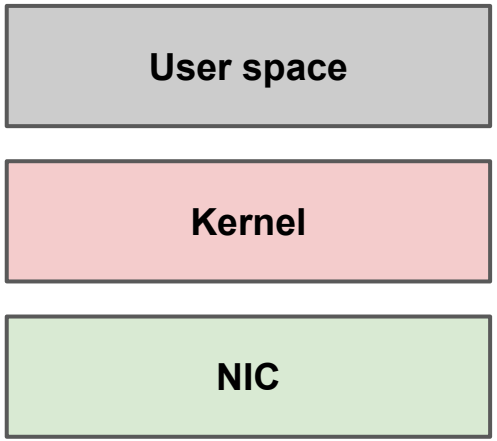
Looking Forward

- Can we design higher level abstractions and/or better tool chains for "programming" the kernel stack?
  - Writing kernel modules is not easy.
  - Writing C programs that would satisfy all the constraints of eBPF is not easy.
- Can we design higher level abstractions for end-host networking, not necessarily tied to the kernel as the data path?

User-Space Packet Processing

Kernel Bypass

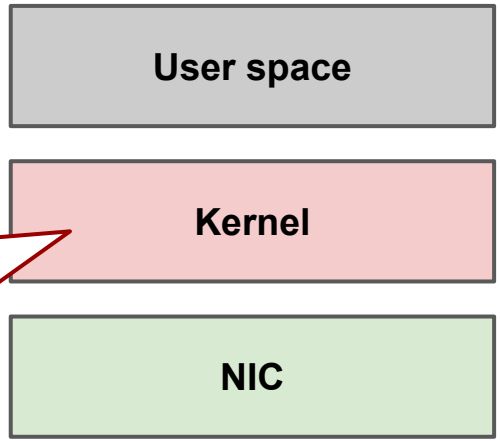
- What if we could write all the packet processing code in a regular program in user space?



Kernel Bypass

- What if we could write all the packet processing code in a regular program in user space?

Helps a program in user space coordinate memory regions with the NIC for incoming and outgoing packets.



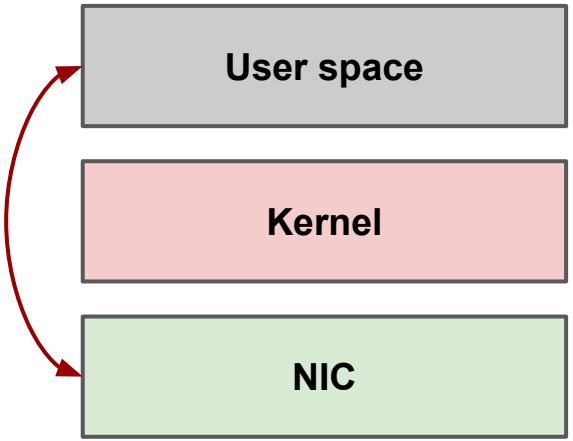
Kernel Bypass

- What if we could write all the 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.

Hence the name, kernel bypass

Example frameworks: DPDK, Netmap



Kernel Bypass - Pros

You are in complete control!

- Fully customizable
- High performance
  - You can optimize your processing to match your traffic and application
  - You don't have to deal with the kernel's overhead for the functionality that you don't necessarily need
- Easier software to develop
  - compared to kernel programming
- Provides an opportunity to rethink how we design the network stack

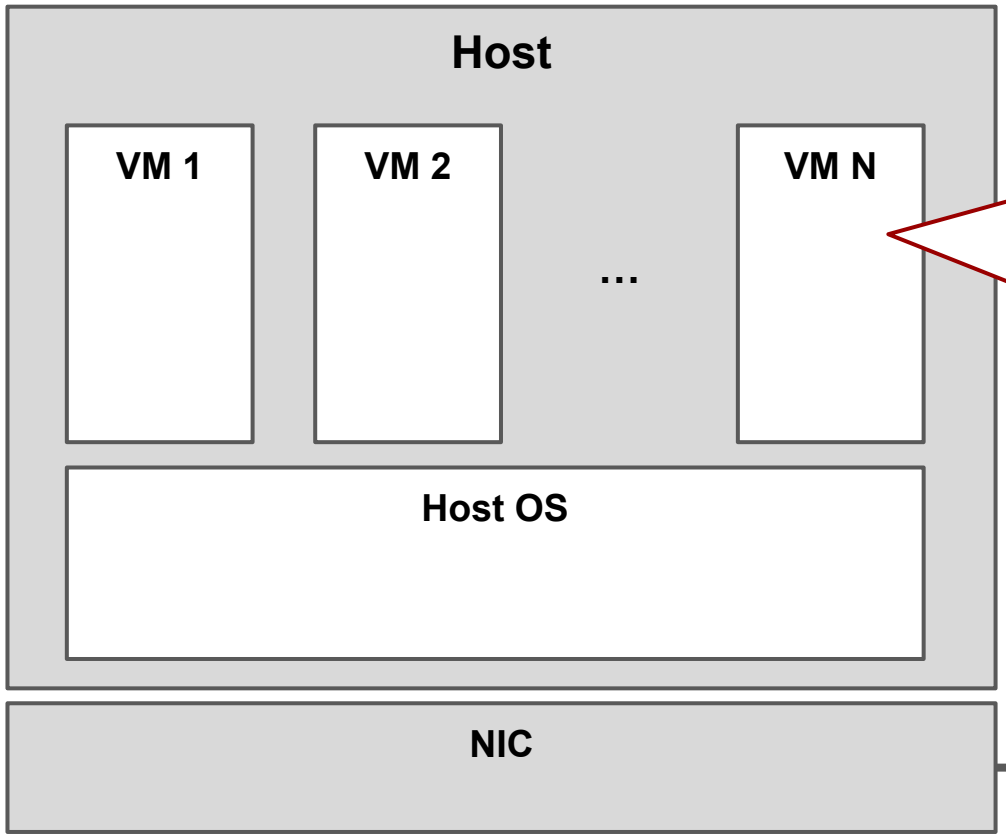
Kernel Bypass - Challenges

You are in complete control :)

- The user-space program takes over the entire NIC.
- Have to re-implement *all* of network processing yourself, from scratch
- Can't take advantage of the Kernel benefits
  - e.g., resource management, security, etc.
- Busy polling to get packets locks up CPU resources

Network Processing in Virtualized Platforms

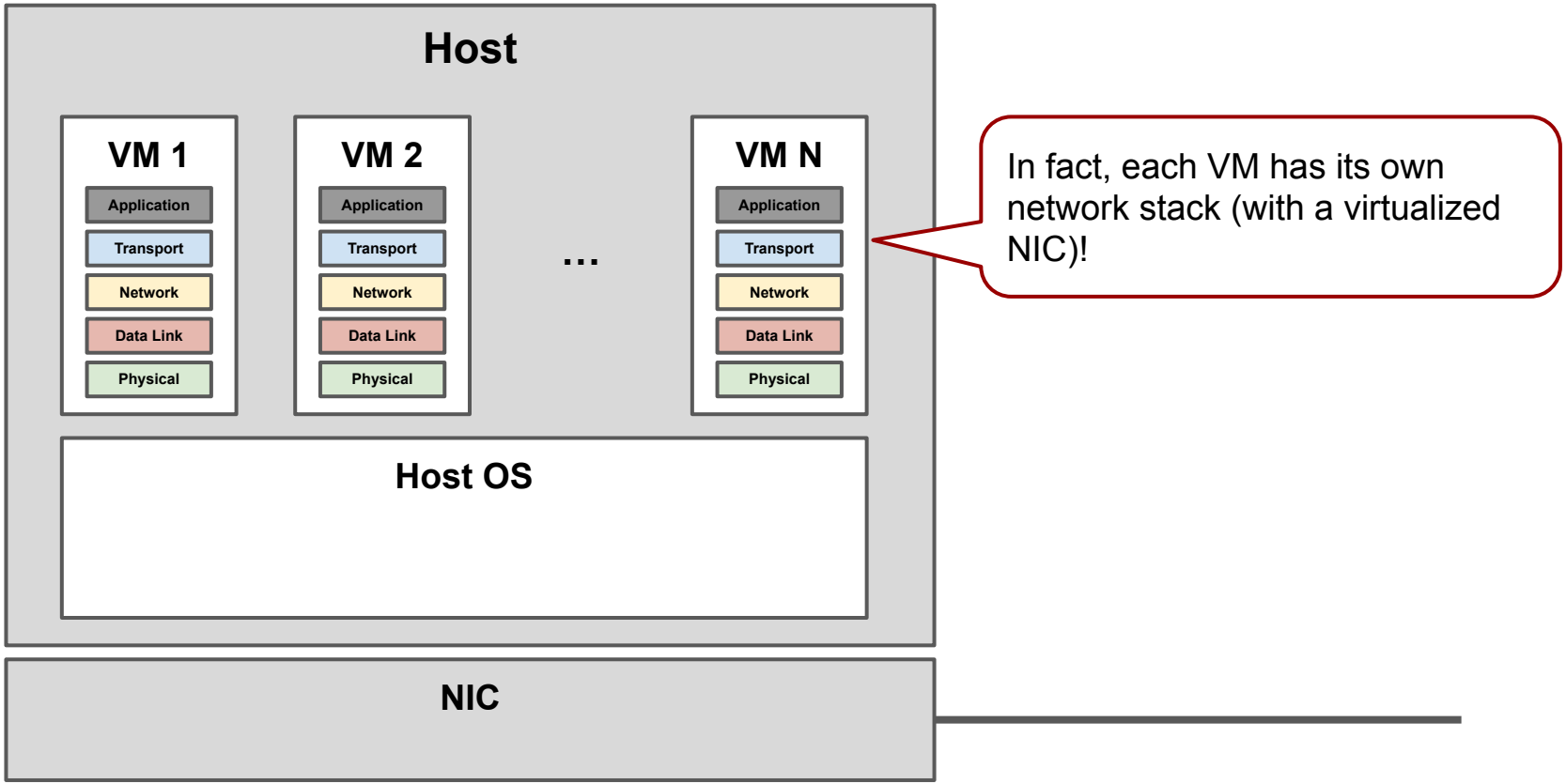
Server Virtualization



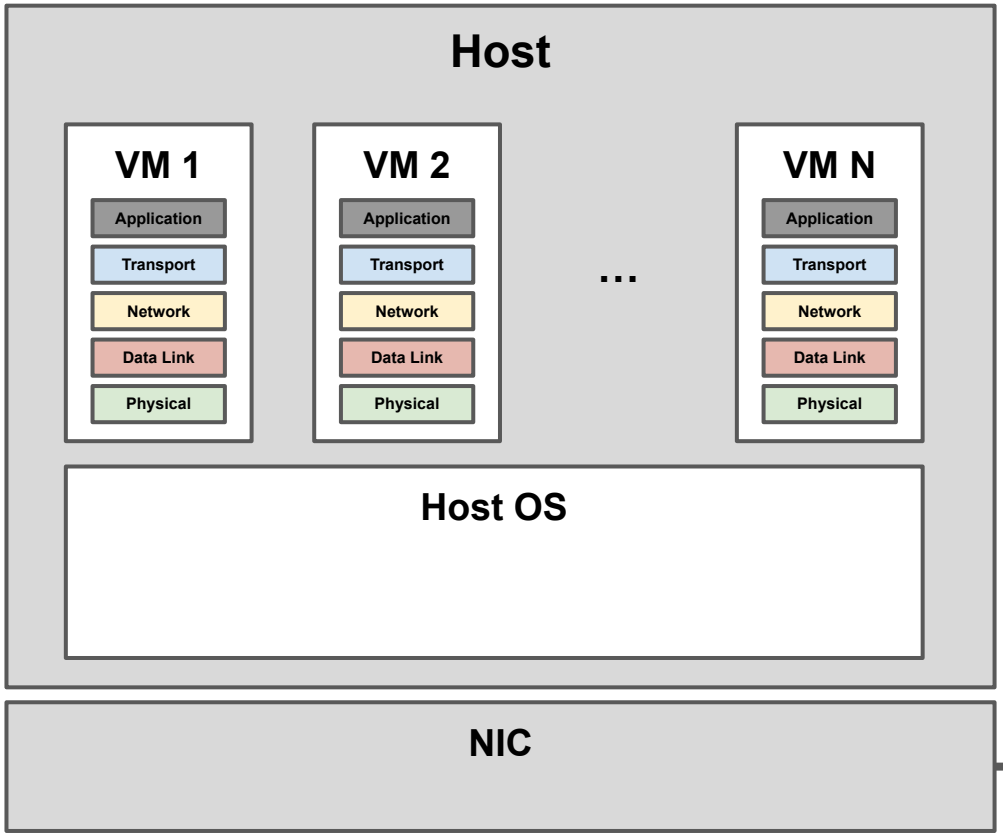
Each VM provides an illusion of having a standalone server.

You can run your operating system of choice, configure/change it however you want, run any application you choose, etc.

Server Virtualization

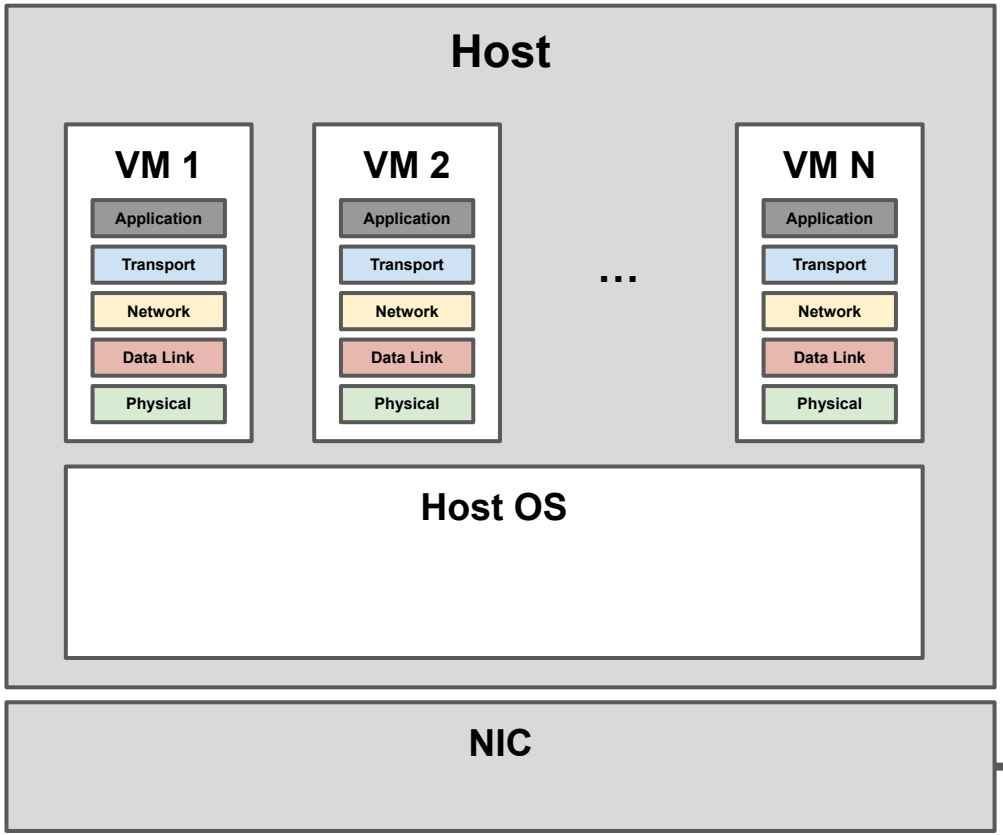


Server Virtualization



The VMs share the link to the network and can run any application and/or network processing they like!

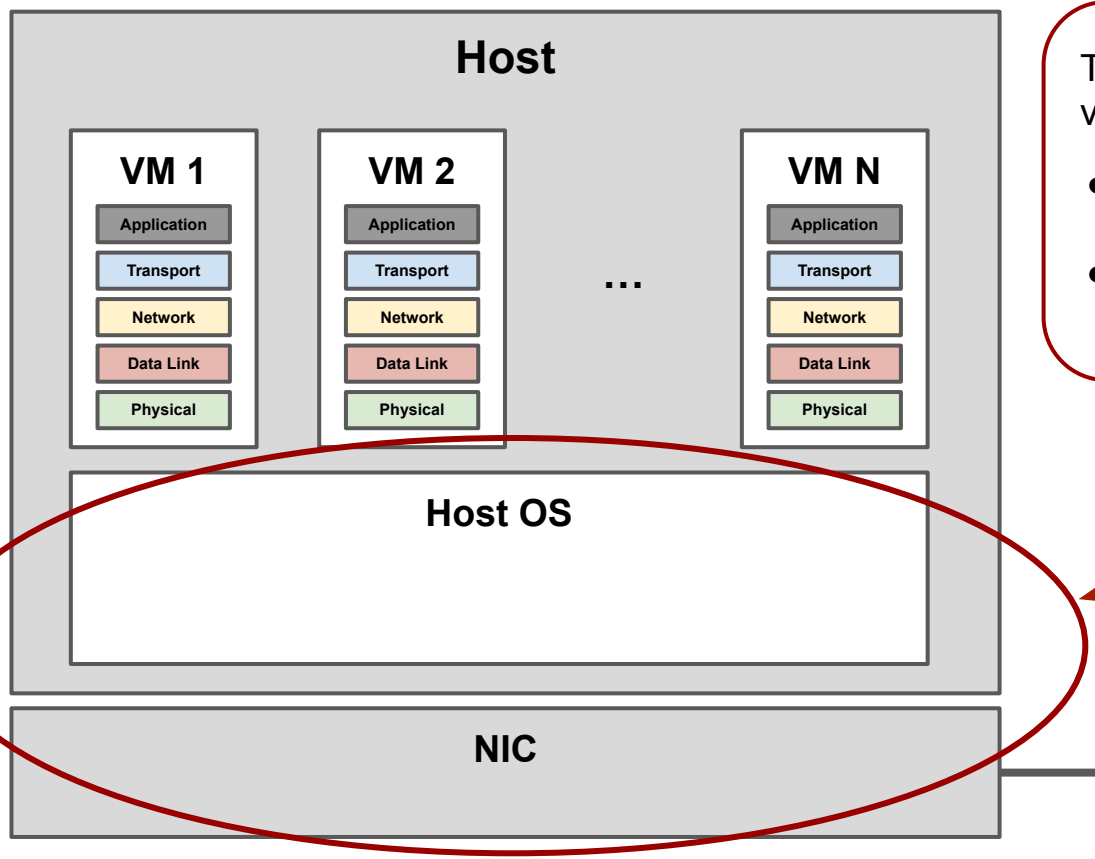
Server Virtualization



Typically, the provider of the virtualized platform needs to

- forward traffic between VMs or VMs and the NIC.
- manage how to share network resource between VMs.

Server Virtualization

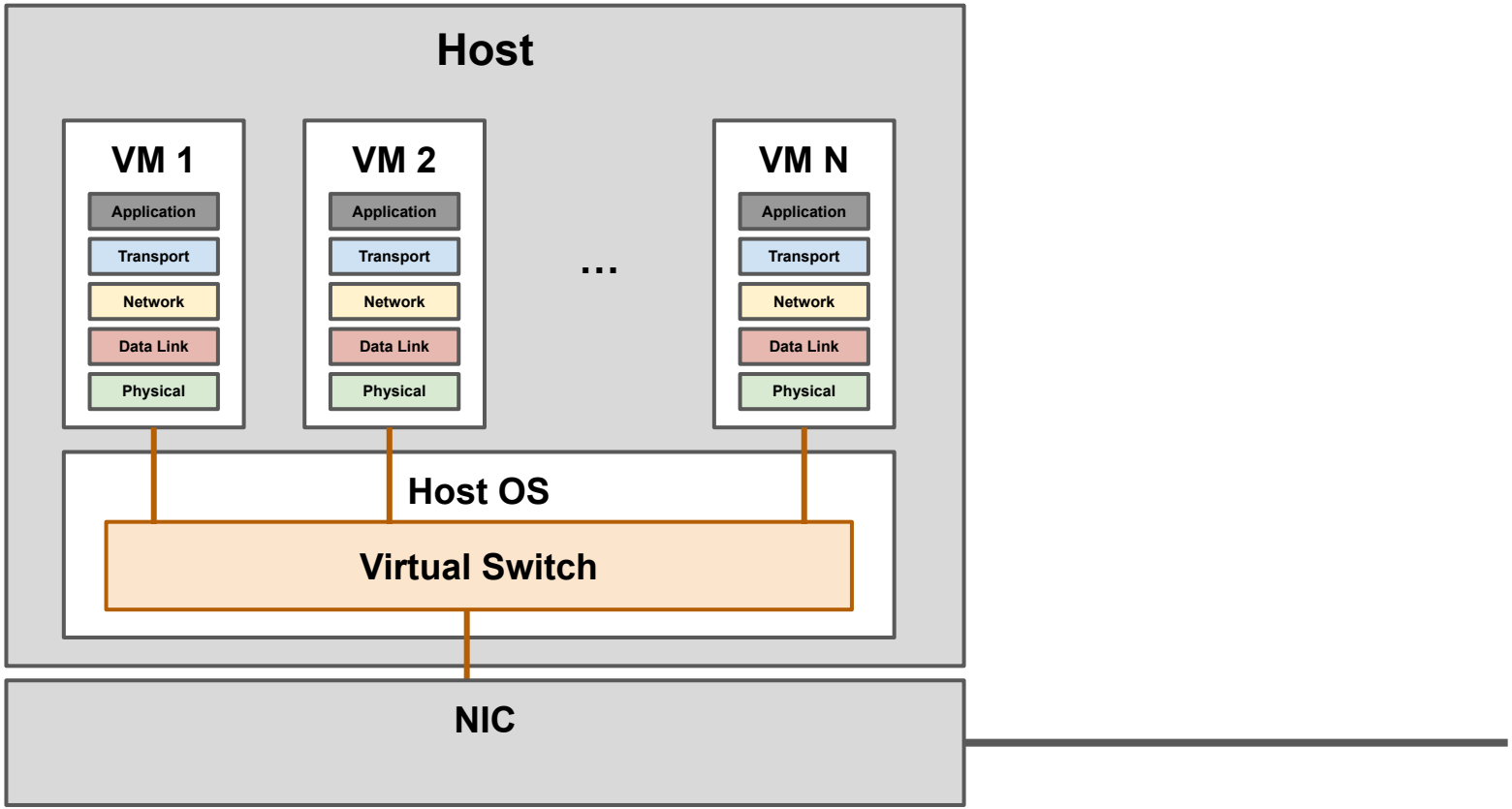


Typically, the provider of the virtualized platform needs to

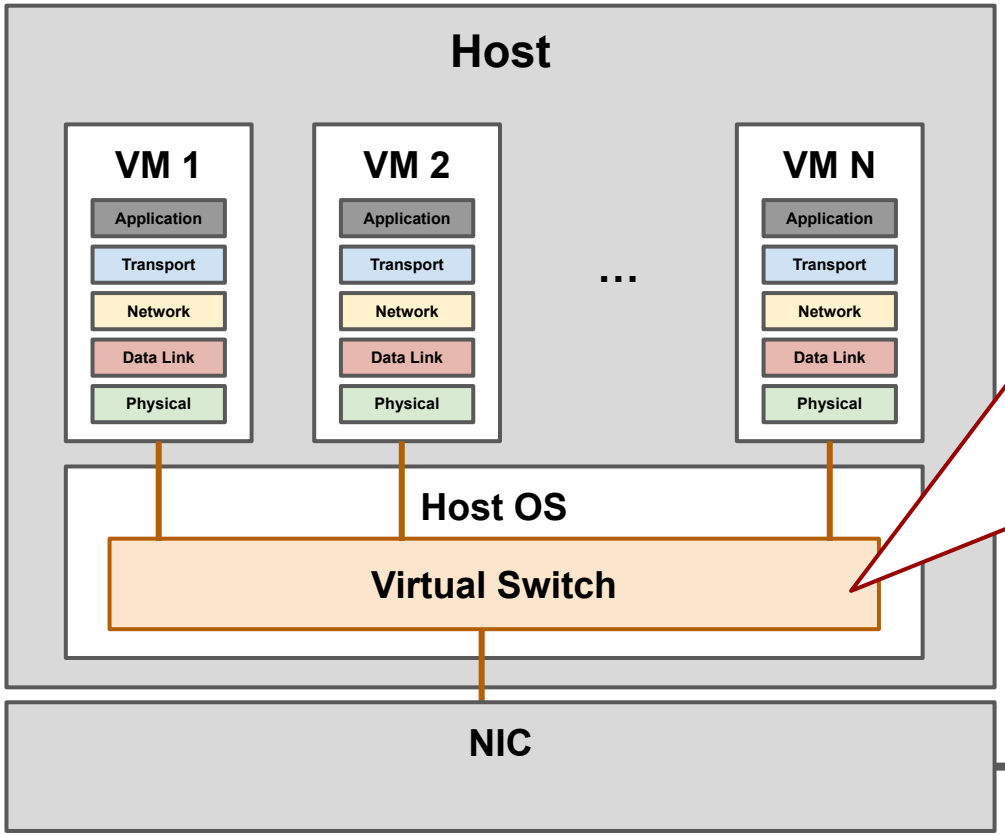
- forward traffic between VMs or VMs and the NIC.
- manage how to share network resource between VMs.

They have to do it somewhere here

Virtual Switch (vSwitch)



Virtual Switch (vSwitch)

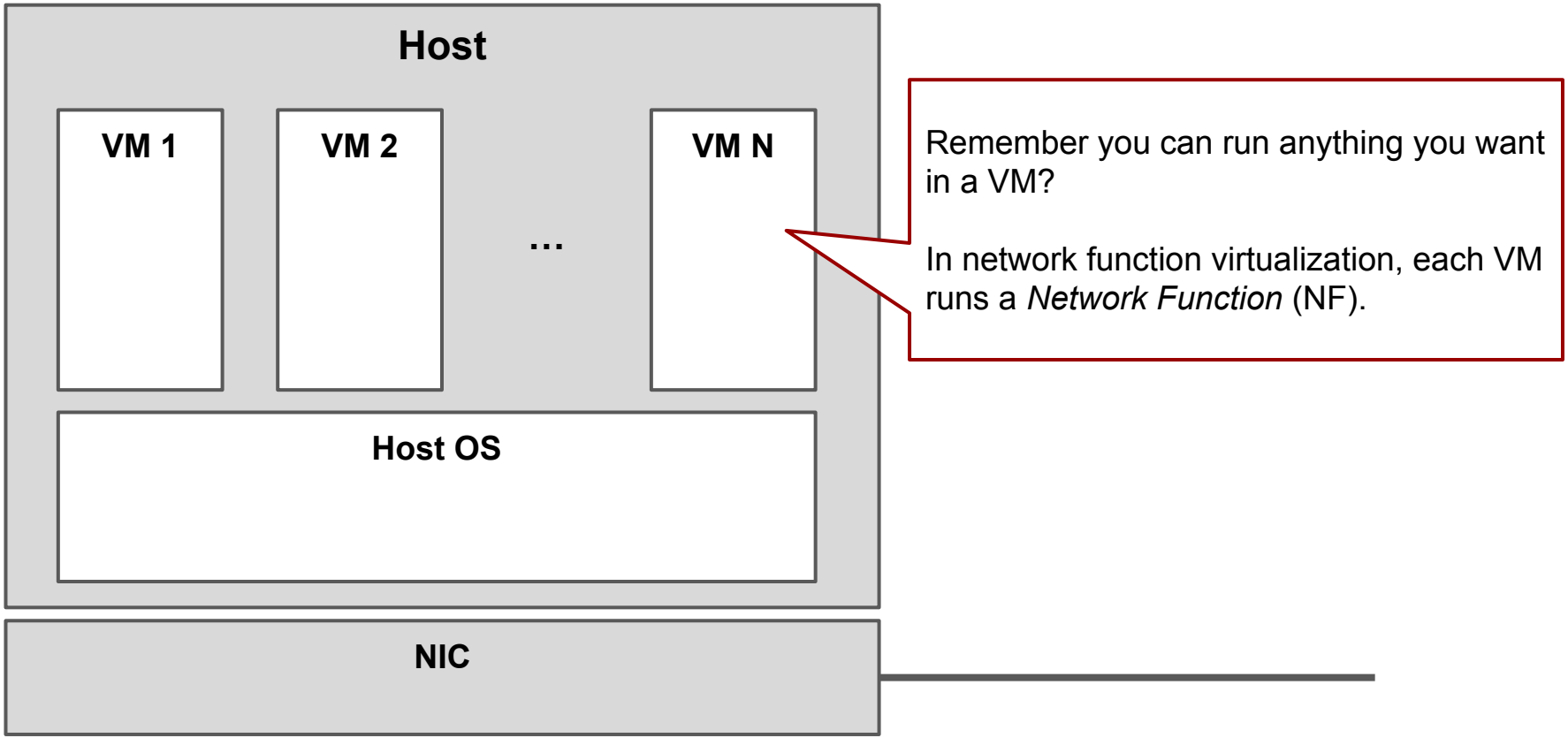


It is a switch! But it can (and needs to) do much more than a switch in the middle of the network (e.g., connection tracking)

It is a large complex piece of software that needs to run fast → possible to change but not easy

Do we use P4 to program it? Do we use OpenFlow (e.g., the start of Open vSwitch)? Or do we need something else? (e.g., Microsoft VFP)

Network Function Virtualization (NFV)



What is a network function?

- Traditionally, switches and routers only do packet processing up to and including layer 3 (the network layer) to do forwarding.
- But soon, it became apparent we may need to do more than just forwarding in the middle of the network and may need to look further into packets (i.e., high layers of the stack)
    - Network address translation (NAT)
    - Stateful firewalls
    - Load balancers
    - Proxies
    - Intrusion detection and prevention
    - …

What is a network function?

- Specialized devices were designed and customized to do these more "advanced" kinds of packet processing.
- They were called *middleboxes*.

What is a network function?

- Network function is a generic term to describe any kind of network processing, specially the more advanced middlebox-like packet processing.
- If network function virtualization (NFV), network functions are as software inside VMs instead of each having a separate (specialized) physical device.

Programming platforms for software network functions

- Should we use a generic server virtualization platform and run network functions in VMs?
- Network functions are special kinds of software
  - They are heavily network-bound
  - They need optimized packet I/O
  - May need more "VM to VM" communication (e.g., for NF chaining)
- Should we use the knowledge that we are running special packet processing software to customize/optimize things more?

This Week's Reading

Paper 1: The design and implementation of Open vSwitch

- A very popular virtual switch
  - Open source
  - Programmable (was based on OpenFlow from the start, but has evolved over the years
  - Production quality
- Uses caching to achieve high performance
  - first packet of a flow goes through a "slower" path with multiple tables and complex actions
  - Once we know the actions we want to take for the packet, a simpler rules with simpler actions is installed in the fast cache for next packets.
  - Sounds familiar?

Paper 2: Restructuring endpoint congestion control

- Remember the pluggable TCP congestion control interface in the linux kernel? This paper takes it a step further.
- They propose a congestion control plane (CCP)
  - think of it as applying the SDN principle to congestion control
- The main logic of the congestion control algorithm runs in the user space out of the main packet processing data path
- The packet processing data path is configured with programs written in a domain specific language to collects statistics for making congestion control decisions
- CCP receives statistics reports from the data path and sends back congestion control decisions (e.g., rate, window size, etc.)

Additional Resources

- Revisiting the Open vSwitch Dataplane Ten Years Later (SIGCOMM'21)
- The eXpress Data Path (XDP) (CoNEXT'18)
- K2, a compiler that optimizes BPF bytecode with formal correctness and safety guarantees (SIGCOMM'21)
- NetBricks: Taking the V out of NFV (OSDI'16)