UNIVERSITY OF WATERLOO

CS 856: Programmable Networks

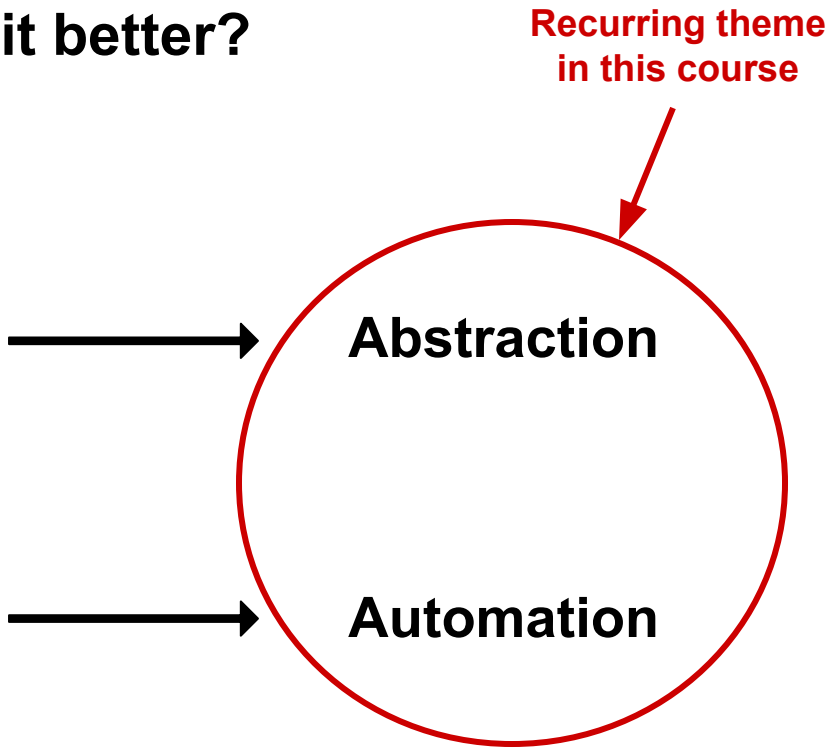
Mina Tahmasbi Arashloo

Winter 2025

How can we make it better?

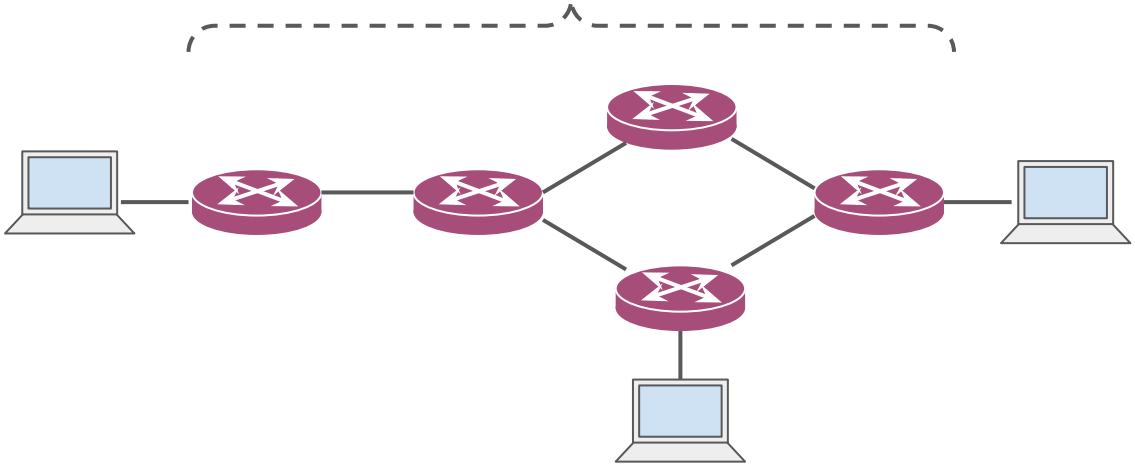
Separate *what* you want the network to do from *how* it is implemented

Don't implement in *manually* 🙂



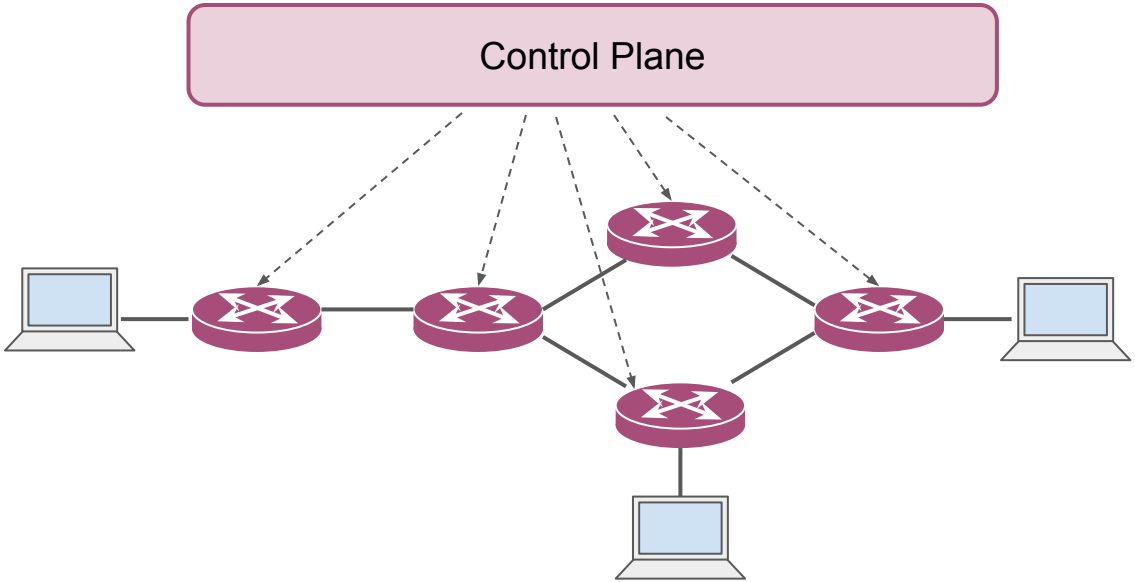
Here are some examples…

Configure a pre-defined set of distributed protocols (e.g., OSPF, BGP, etc.) to pick your desired forwarding paths.



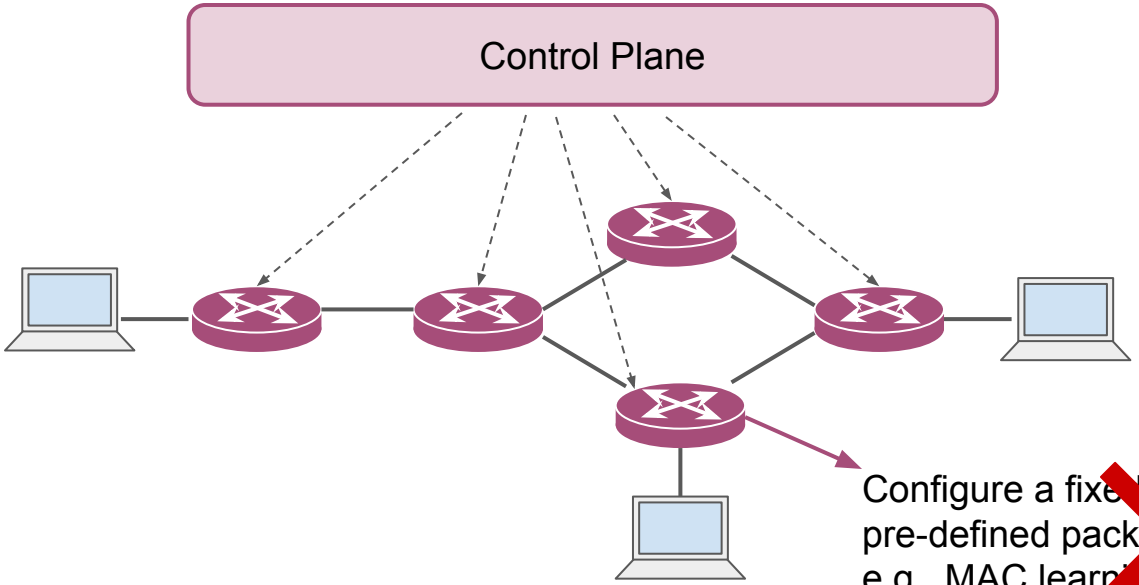
Here are some examples…

- Write a program that decides the forwarding paths.
- Have a runtime compute and communicate proper configurations to network devices.



Here are some examples…

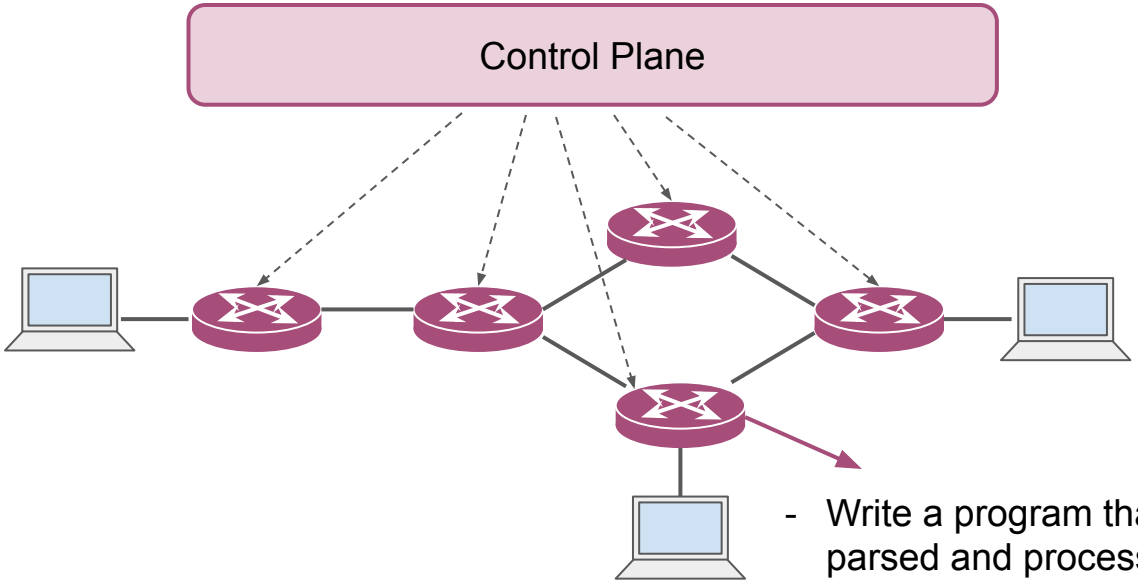
- Write a program that decides the forwarding paths.
- Have a runtime compute and communicate proper configurations to network devices.



Configure a fixed-function hardware with pre-defined packet processing steps, e.g., MAC learning → GRE-Tunnel Processing → IP forwarding

Here are some examples…

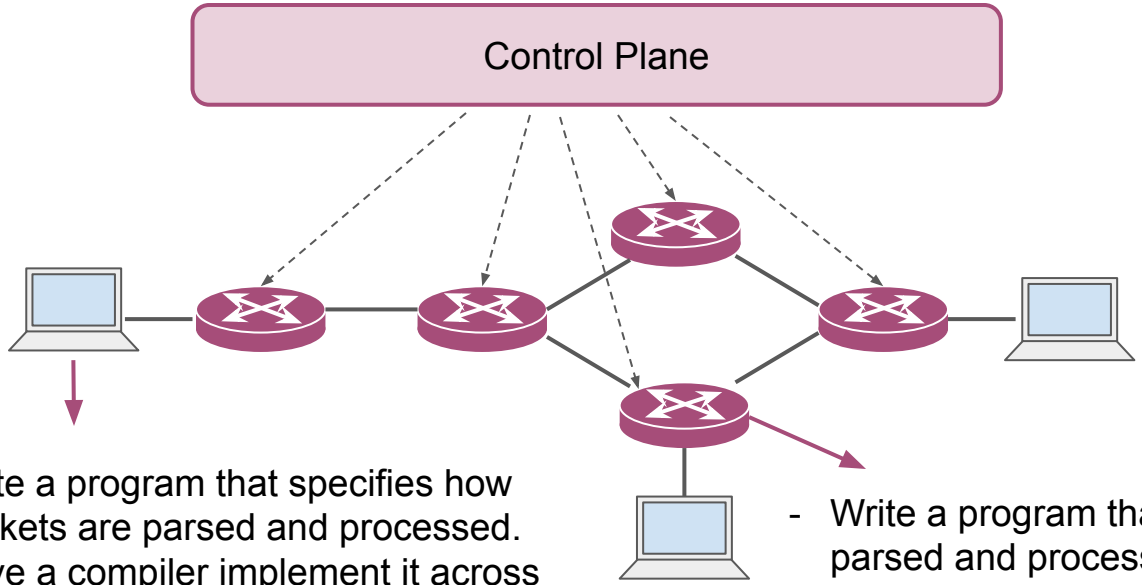
- Write a program that decides the forwarding paths.
- Have a runtime compute and communicate proper configurations to network devices.



- Write a program that specifies how packets are parsed and processed.
- Have a compiler translate that into instructions for switch hardware.

Here are some examples…

- Write a program that decides the forwarding paths.
- Have a runtime compute and communicate proper configurations to network devices.

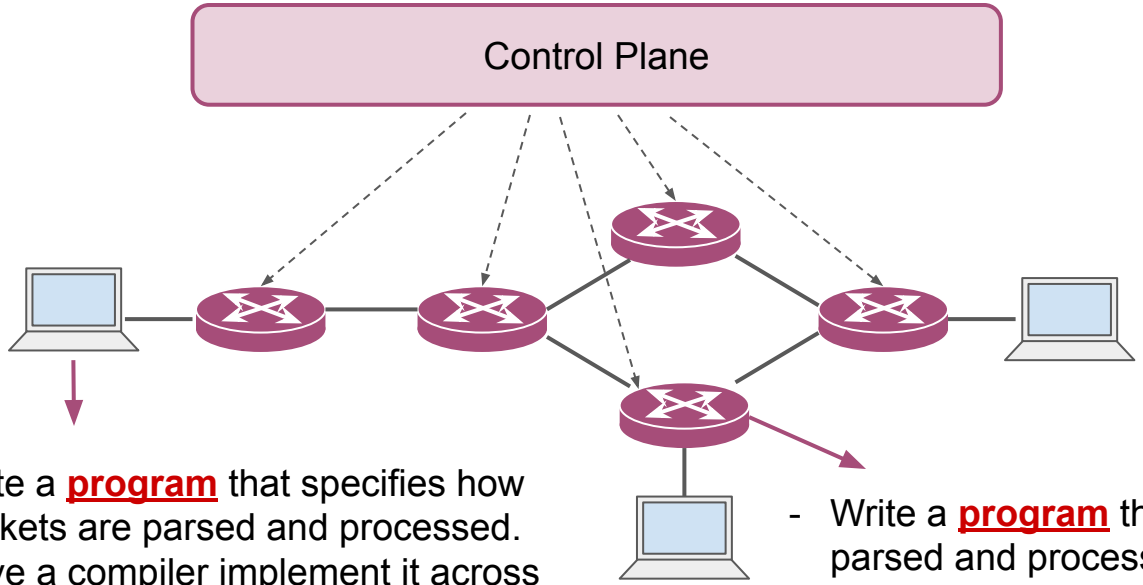


- Write a program that specifies how packets are parsed and processed.
- Have a compiler implement it across user-space, the Kernel, and hardware accelerators.

- Write a program that specifies how packets are parsed and processed.
- Have a compiler translate that into instructions for switch hardware.

Here are some examples…

- Write a **program** that decides the forwarding paths.
- Have a runtime compute and communicate proper configurations to network devices.

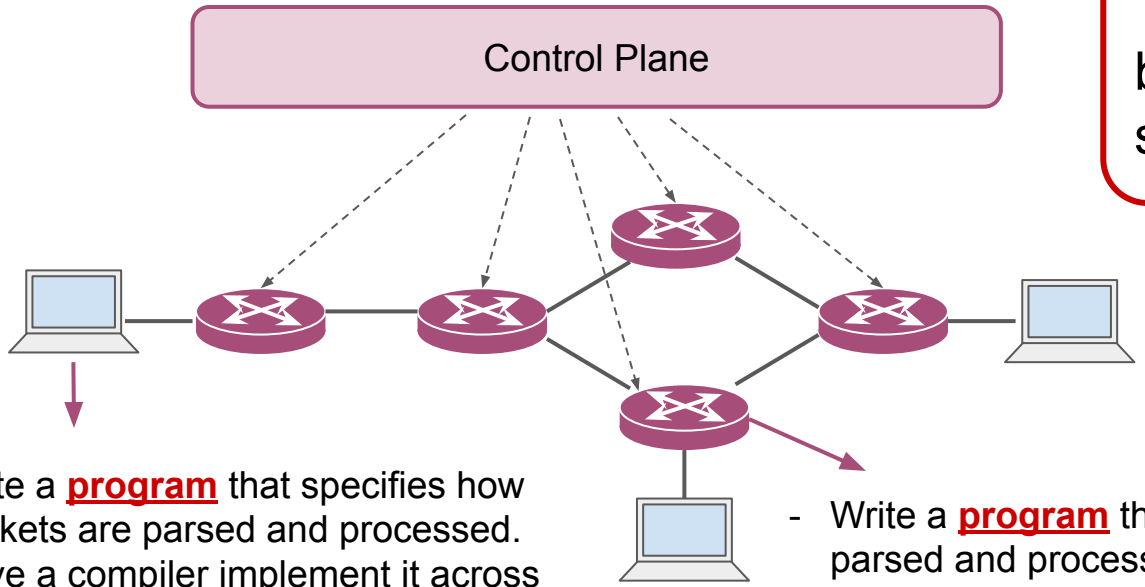


- Write a **program** that specifies how packets are parsed and processed.
- Have a compiler implement it across user-space, the Kernel, and hardware accelerators.

- Write a **program** that specifies how packets are parsed and processed.
- Have a compiler translate that into instructions for switch hardware.

Here are some examples…

- Write a **program** that decides the forwarding paths.
- Have a runtime compute and communicate proper configurations to network devices.



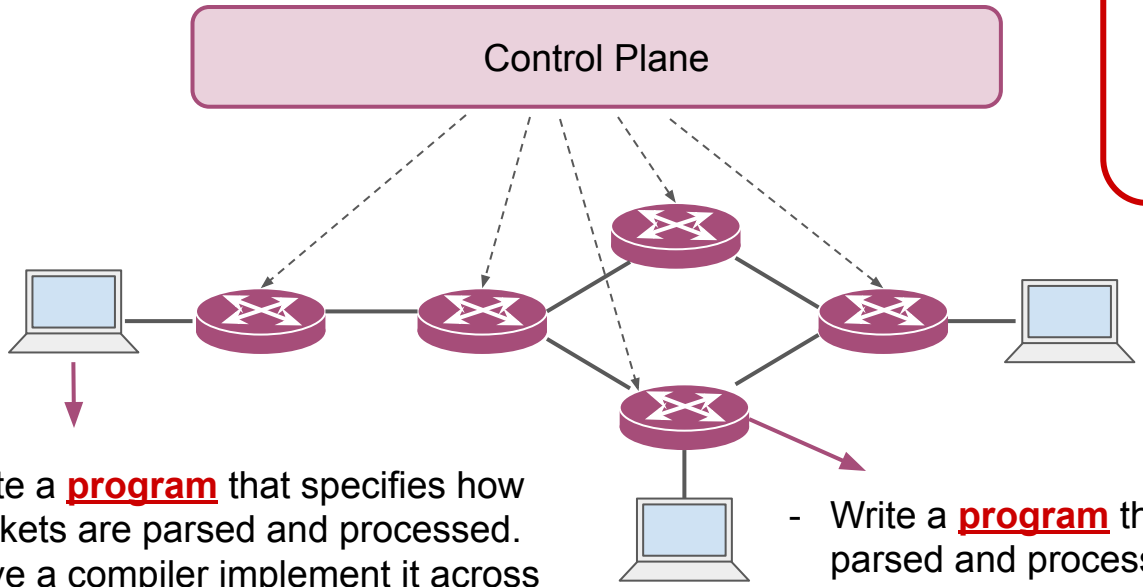
Treat the network as a big, distributed, and specialized computer

- Write a **program** that specifies how packets are parsed and processed.
- Have a compiler implement it across user-space, the Kernel, and hardware accelerators.

- Write a **program** that specifies how packets are parsed and processed.
- Have a compiler translate that into instructions for switch hardware.

Here are some examples…

- Write a **program** that decides the forwarding paths.
- Have a runtime compute and communicate proper configurations to network devices.



Programmable Networks

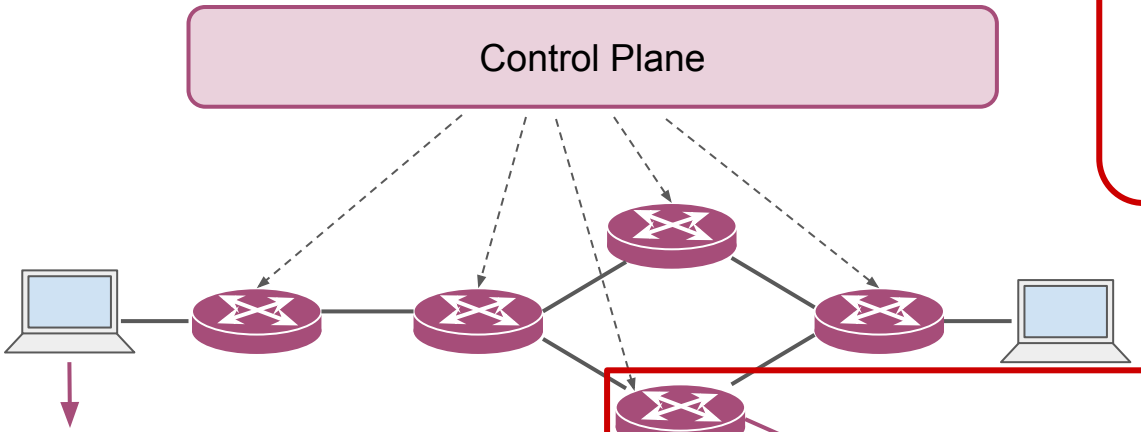
- Write a **program** that specifies how packets are parsed and processed.
- Have a compiler implement it across user-space, the Kernel, and hardware accelerators.

- Write a **program** that specifies how packets are parsed and processed.
- Have a compiler translate that into instructions for switch hardware.

Here are some examples…

- Write a **program** that decides the forwarding paths.
- Have a runtime compute and communicate proper configurations to network devices.

Programmable Networks



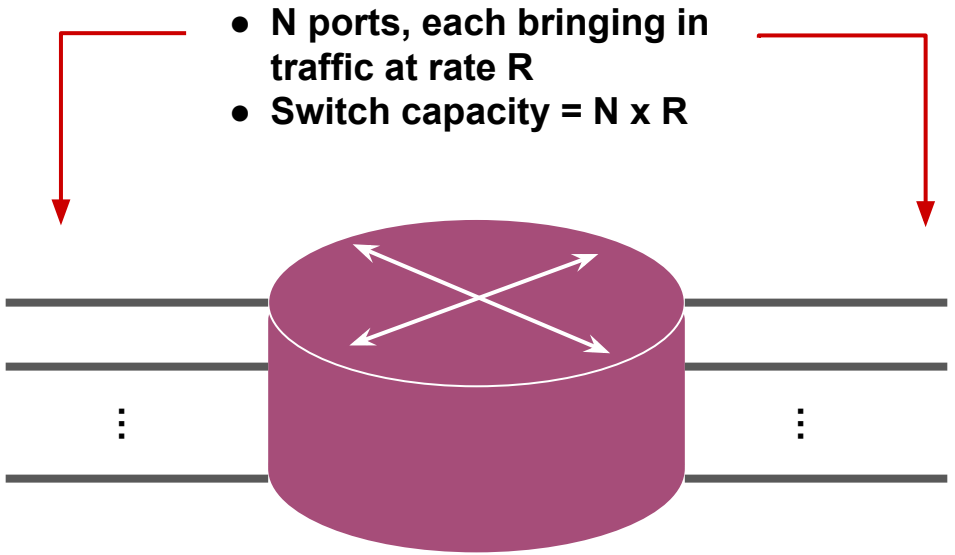
- Write a **program** that specifies how packets are parsed and processed.
- Have a compiler implement it across user-space, the Kernel, and hardware accelerators.

- Write a **program** that specifies how packets are parsed and processed.
- Have a compiler translate that into instructions for switch hardware.

Programmable Switches

Challenge: High-Speed Reconfigurable Data Plane

- Switch data planes need to process packets very fast



Challenge: High-Speed Reconfigurable Data Plane

- Switch data planes need to process packets very fast

R = 100 Gbps

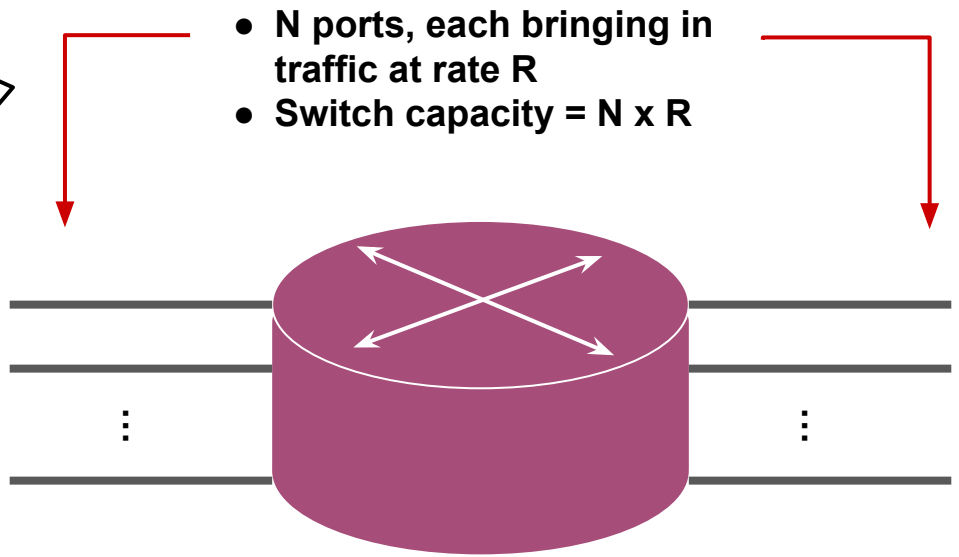
For back-to-back 64B packets, we have a packet every ~5ns.

For back-to-back 1500B packets, we have a packet every ~120ns.

N = 16

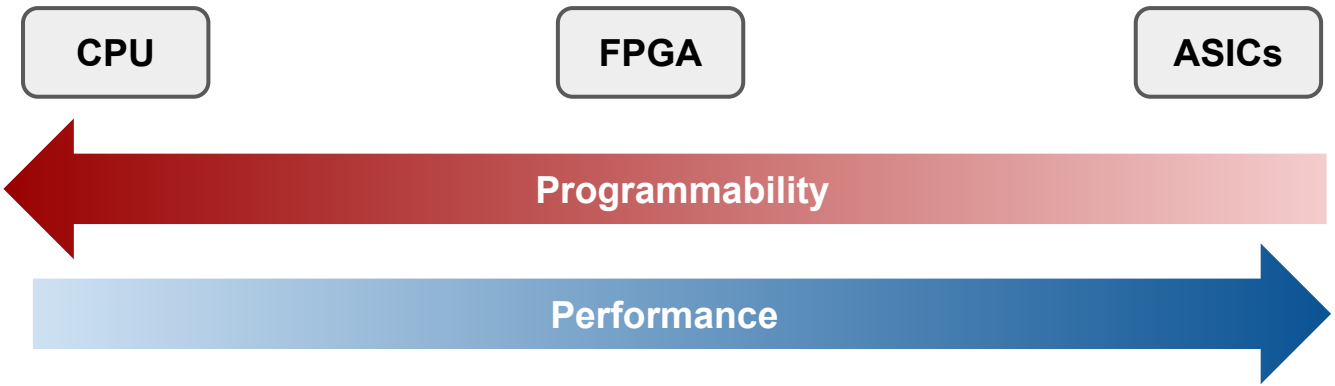
This happens concurrently on 16 ports…

N x R = 1.6 Tbps!



Challenge: High-Speed Reconfigurable Data Plane

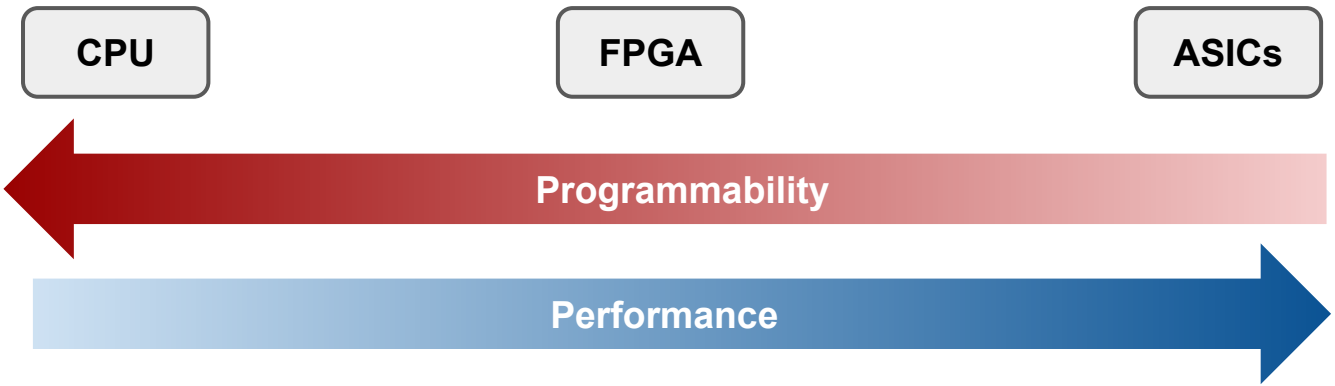
- There is a trade-off between programmability and performance



Challenge: High-Speed Reconfigurable Data Plane

- programmability and performance

General-purpose processors like CPUs can be programmed to execute any logic.

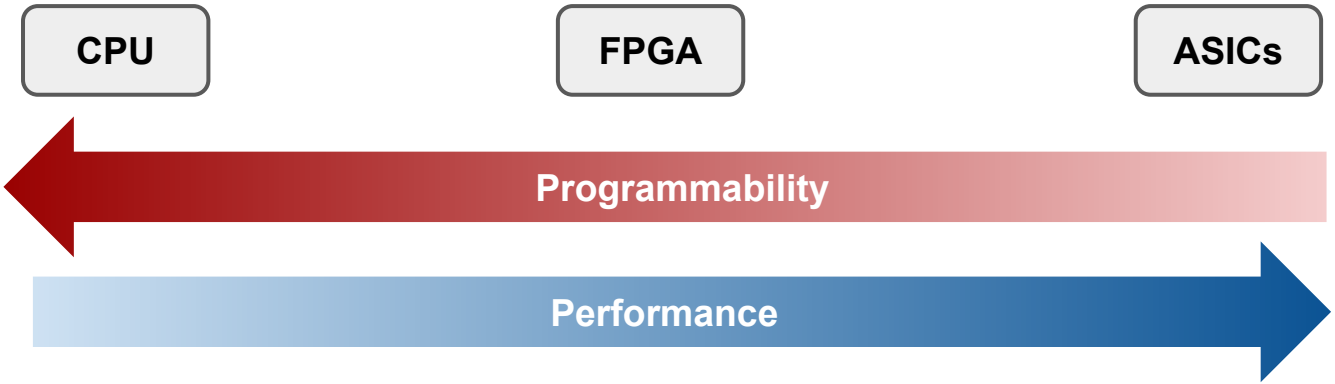


Challenge: High-Speed Reconfigurable Data Plane

- programmal

General-purpose processors like CPUs can be programmed to execute any logic.

Fixed-function ASICs are customized and optimized to for a certain kind of computation.

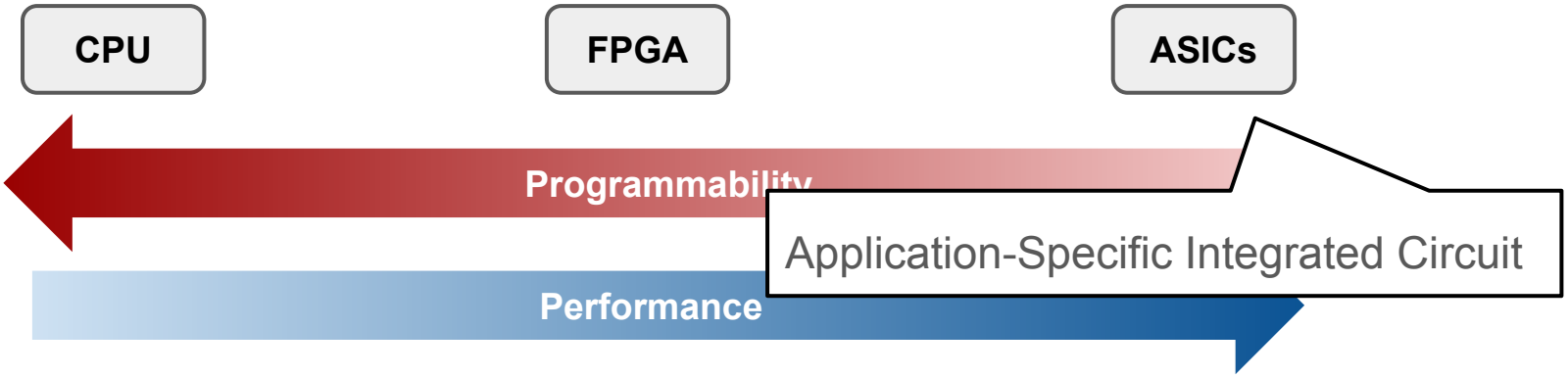


Challenge: High-Speed Reconfigurable Data Plane

- programmable

General-purpose processors like CPUs can be programmed to execute any logic.

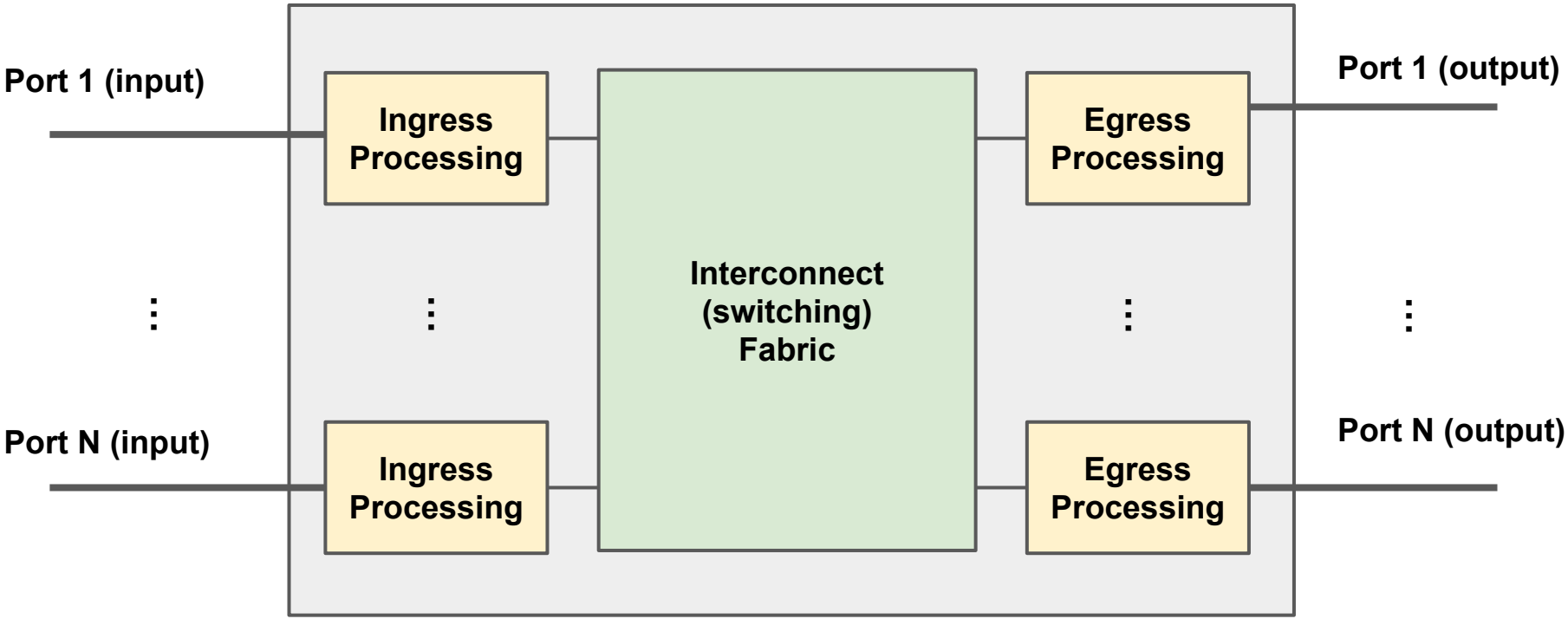
Fixed-function ASICs are customized and optimized to for a certain kind of computation.



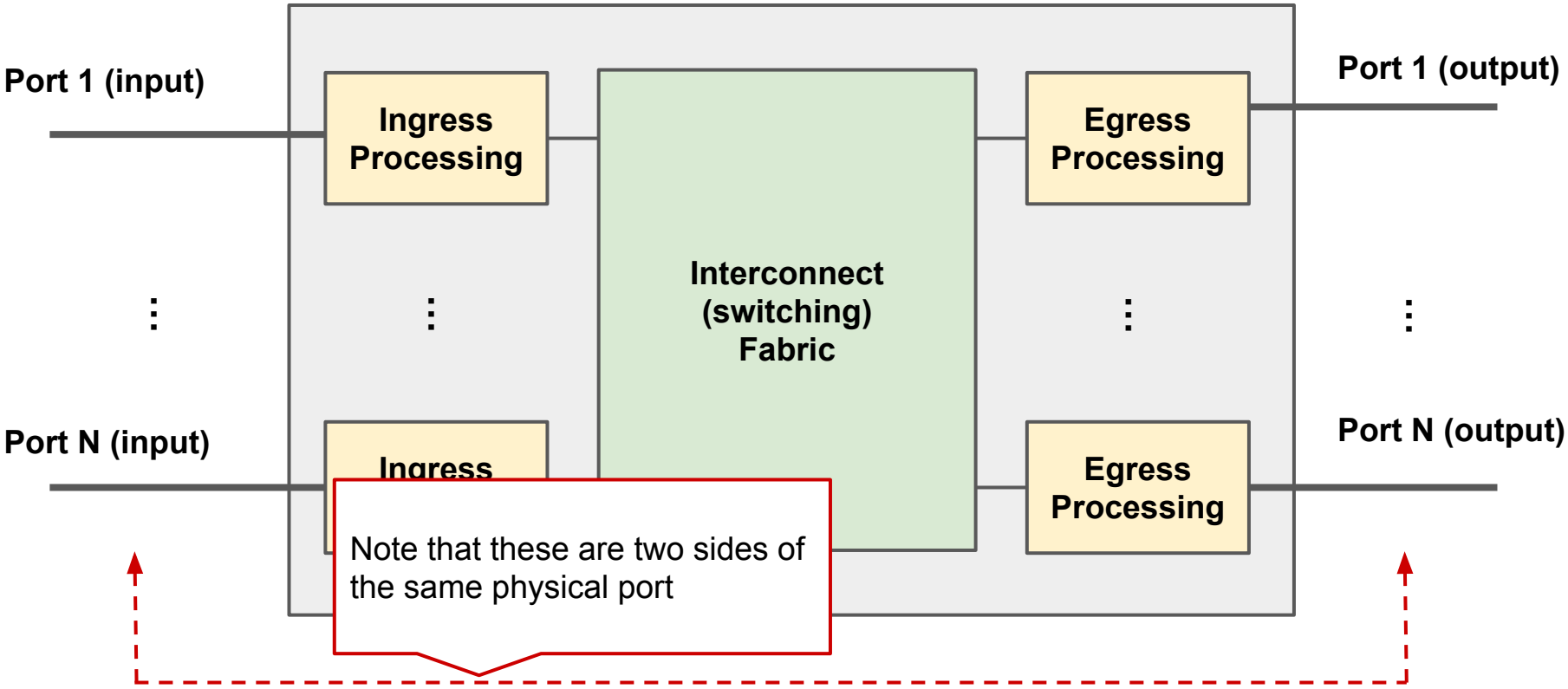
Challenge: High-Speed Reconfigurable Data Plane

- **Traditionally:** switching chips were ASICs
  - customized for packet processing, e.g., packet parsing, forwarding tables, etc.
- **The "programmability" trend:**
  - **Q1:** Is it possible to have a high-speed reconfigurable switch data plane?
  - **Q2:** How much reconfigurability can we add to the switch data plane and still be able to perform high-speed packet processing?

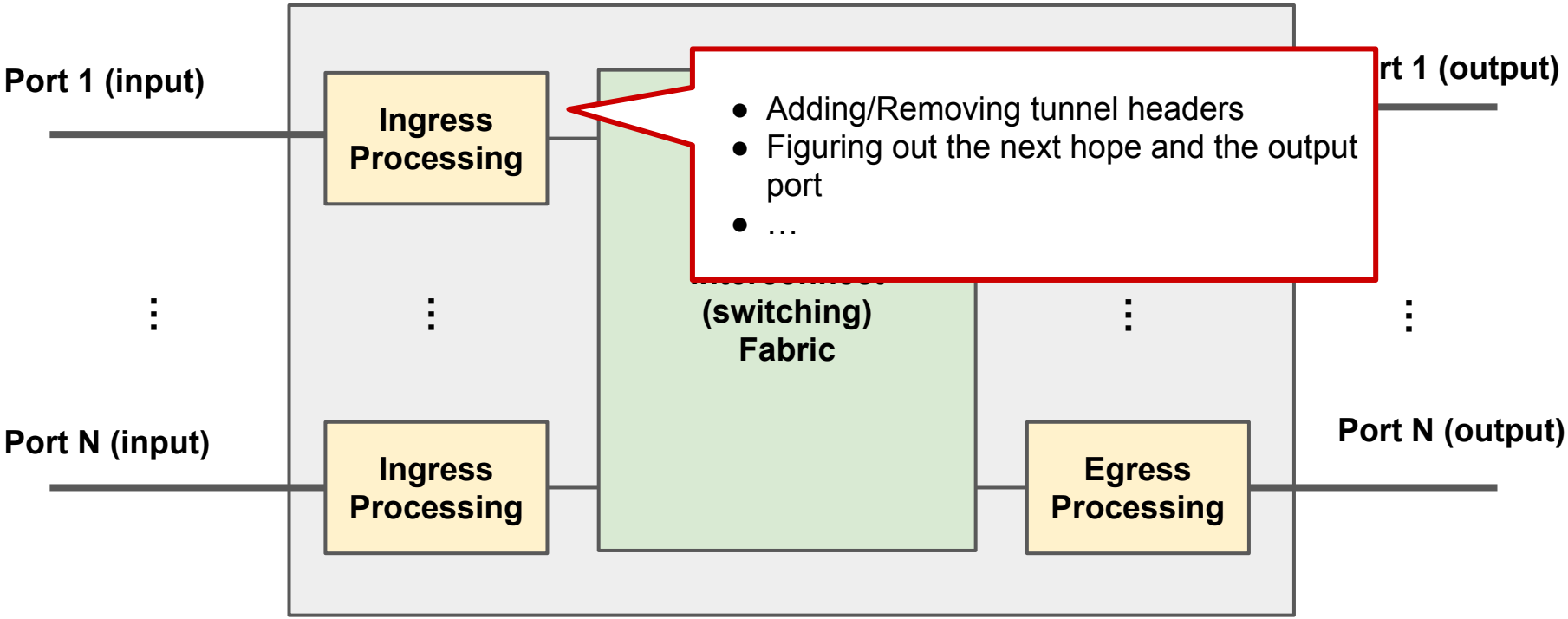
Inside a (output-queued) switch



Inside a (output-queued) switch

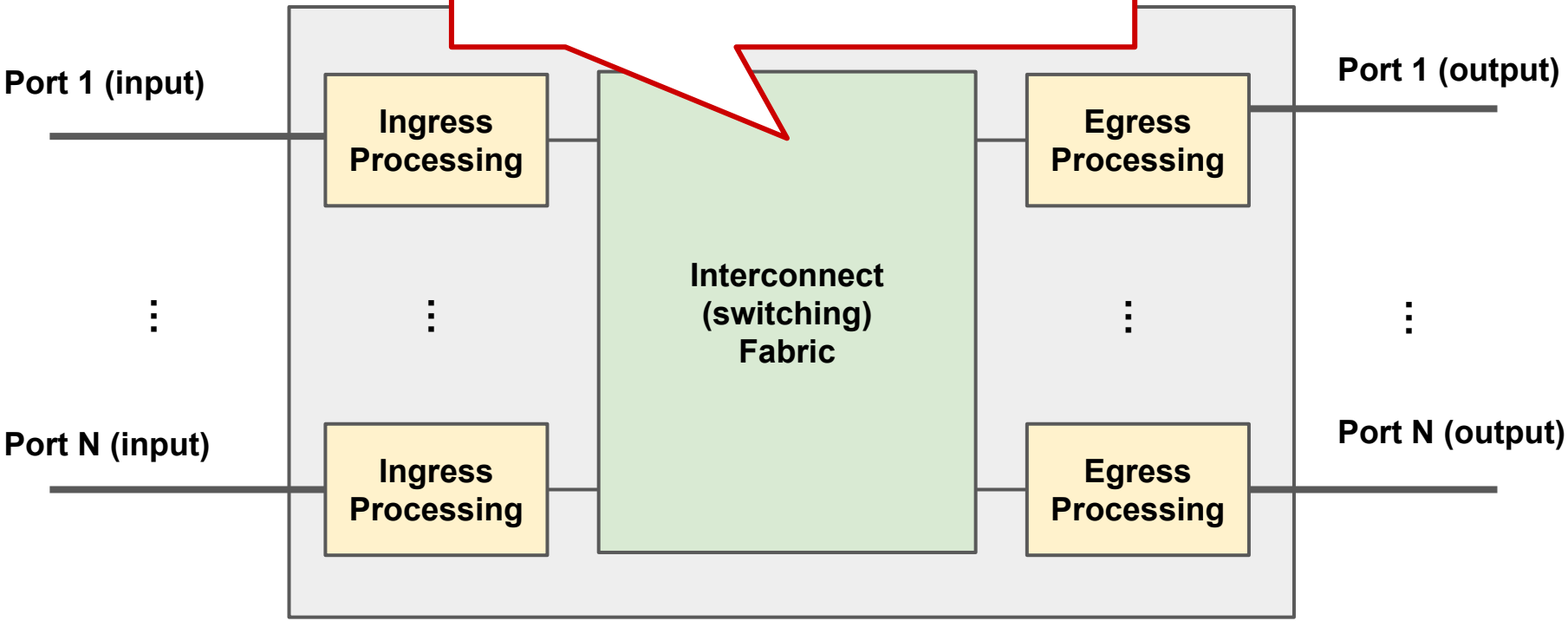


Inside a (output-queued) switch



Inside a (output-c

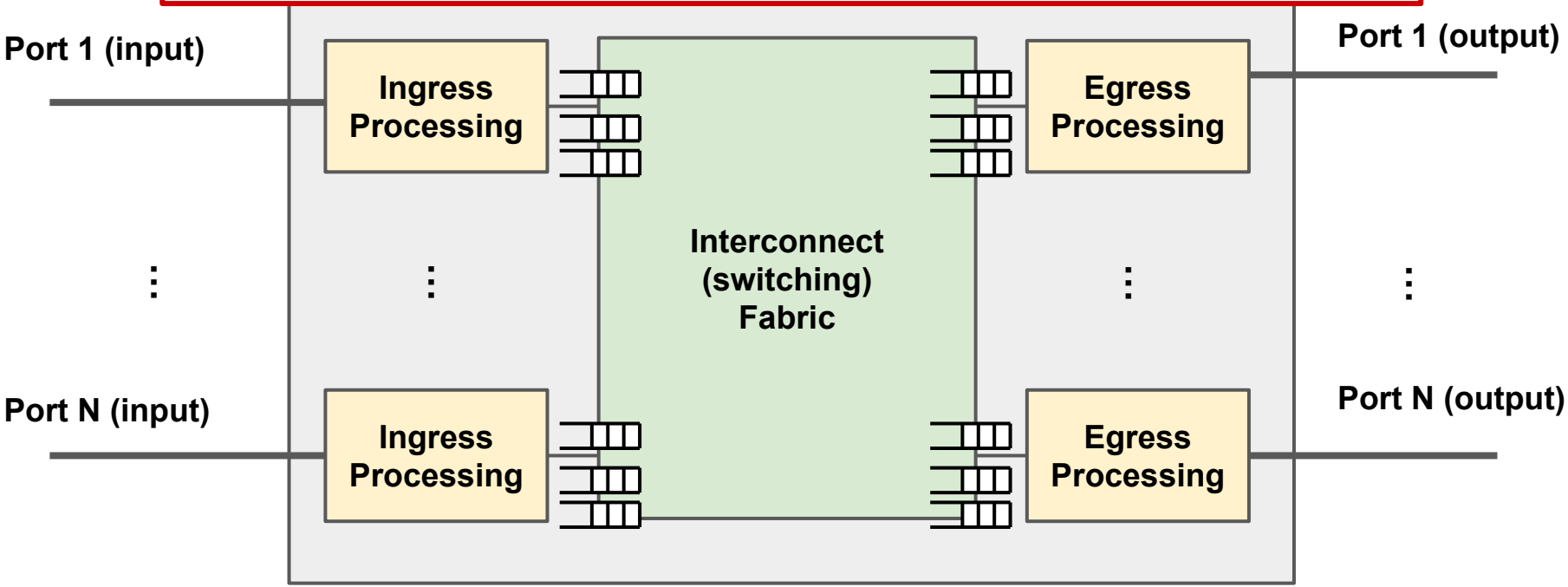
- Connects input ports to output ports
- Needs to operate at high speed (~ N times the speed of an individual port)



Inside

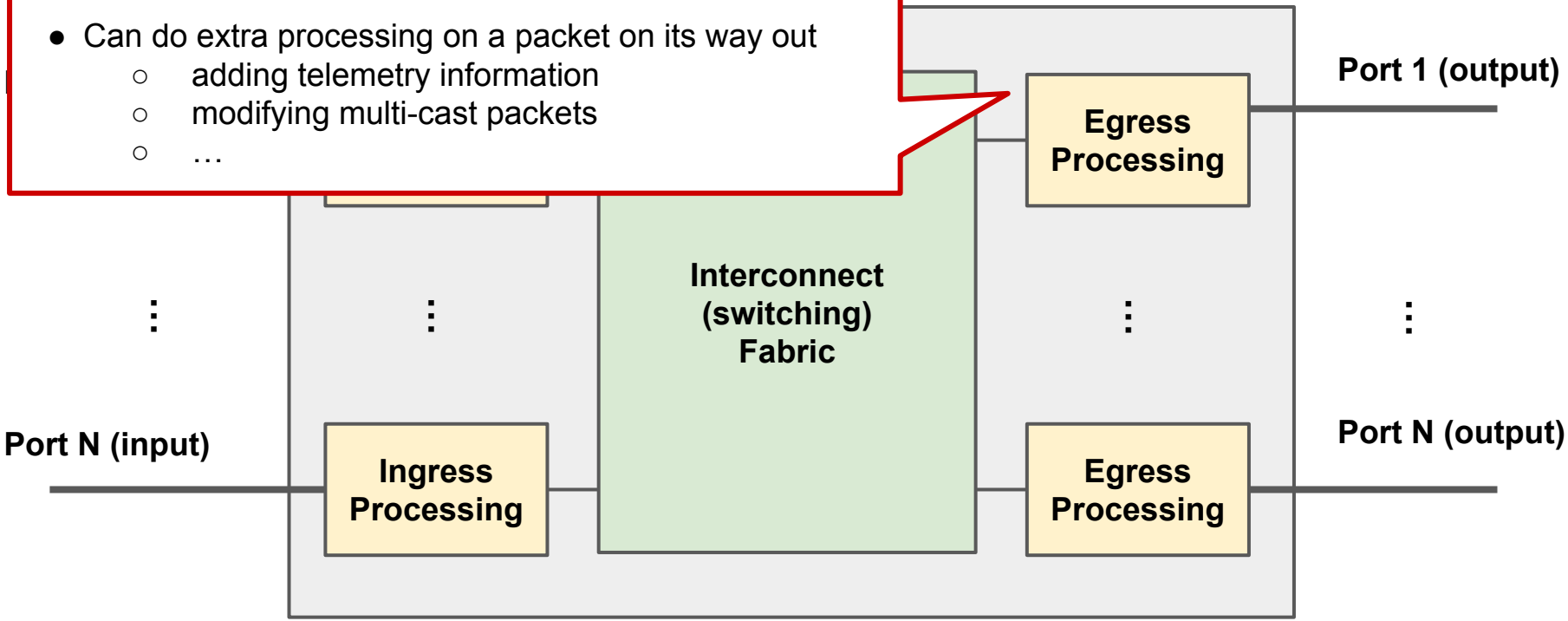
Traffic manager:

- Packets going to the same output will be buffered in a queue
  - In ingress and/or egress.
- Packet scheduling algorithms decide which packets will go out of that port next

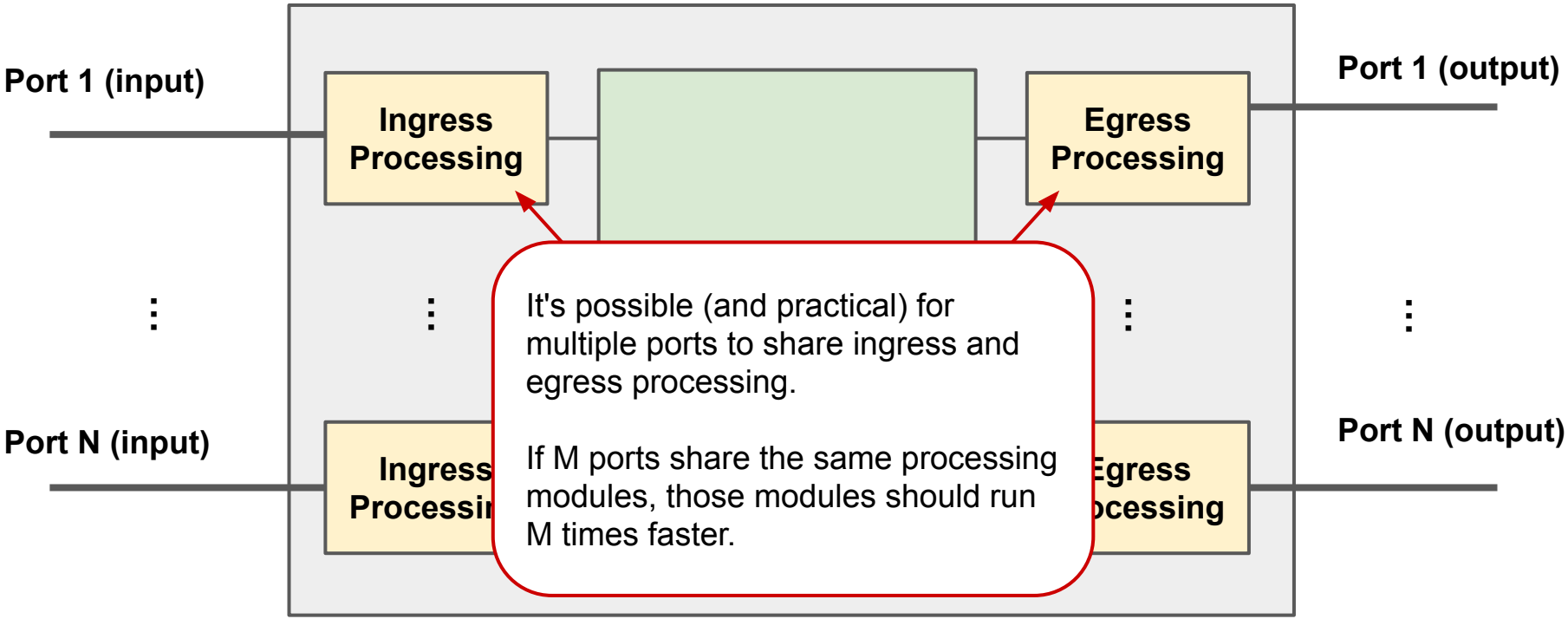


Inside a (output-queued) switch

- Can do extra processing on a packet on its way out
  - adding telemetry information
  - modifying multi-cast packets
  - …



Inside a (output-queued) switch



What should a "programmable" switch look like?

- We can't make everything programmable
  - the programmability-performance trade-off
- How do we decide what should be fixed and what programmable?
  - Which parts are subject to more innovation?
  - The logic of which part do we want to change more frequently?
  - Where can we afford to pay the overhead of programmability?

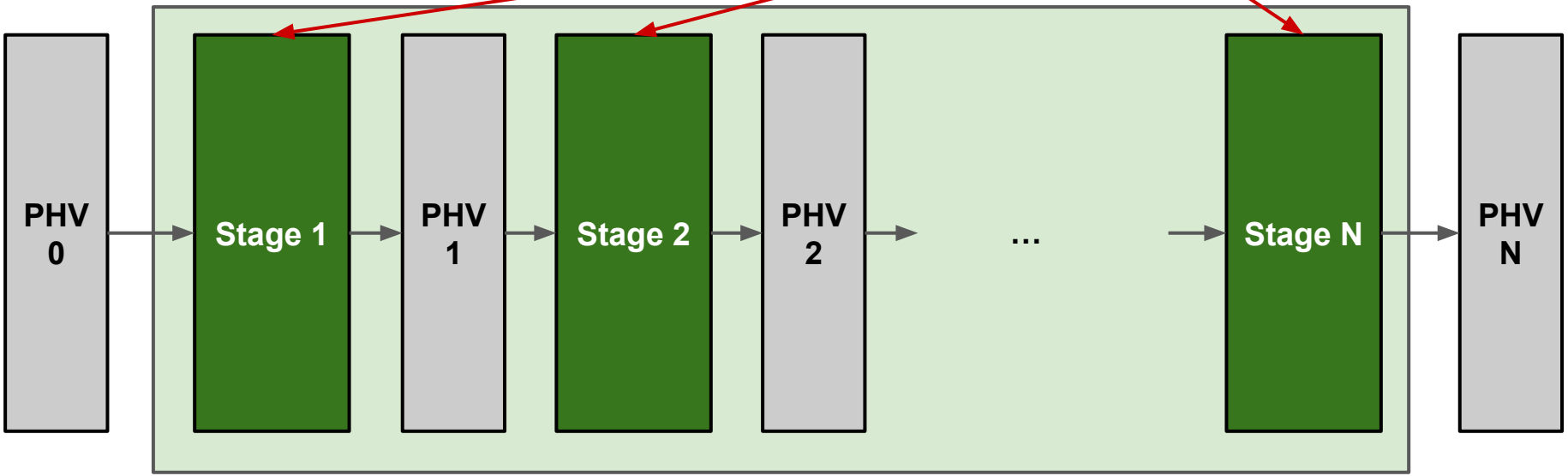
Proposals for programmable switch architectures (not exhaustive)

- 2013: Reconfigurable Match-Action Tables (RMT)
  - Evolved into Protocol-Independent Switch Architecture (PISA)
  - There was a successful startup (Barefoot Networks) and a commercial switching chip based on it (Tofino).
  - Acquire by Intel, and unfortunately discontinued ~2 years ago.
  - Why are we still talking about this then?
- 2017: dRMT = disaggregated RMT
- 2022: Trio by Juniper Networks
- 2022: FlexCore
- 2024: OptimusPrime

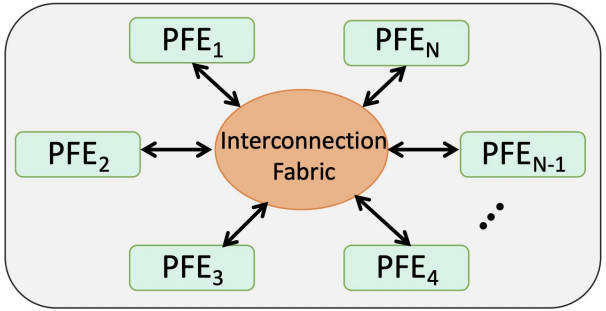
Pipelines vs. Run-to-completion on cores

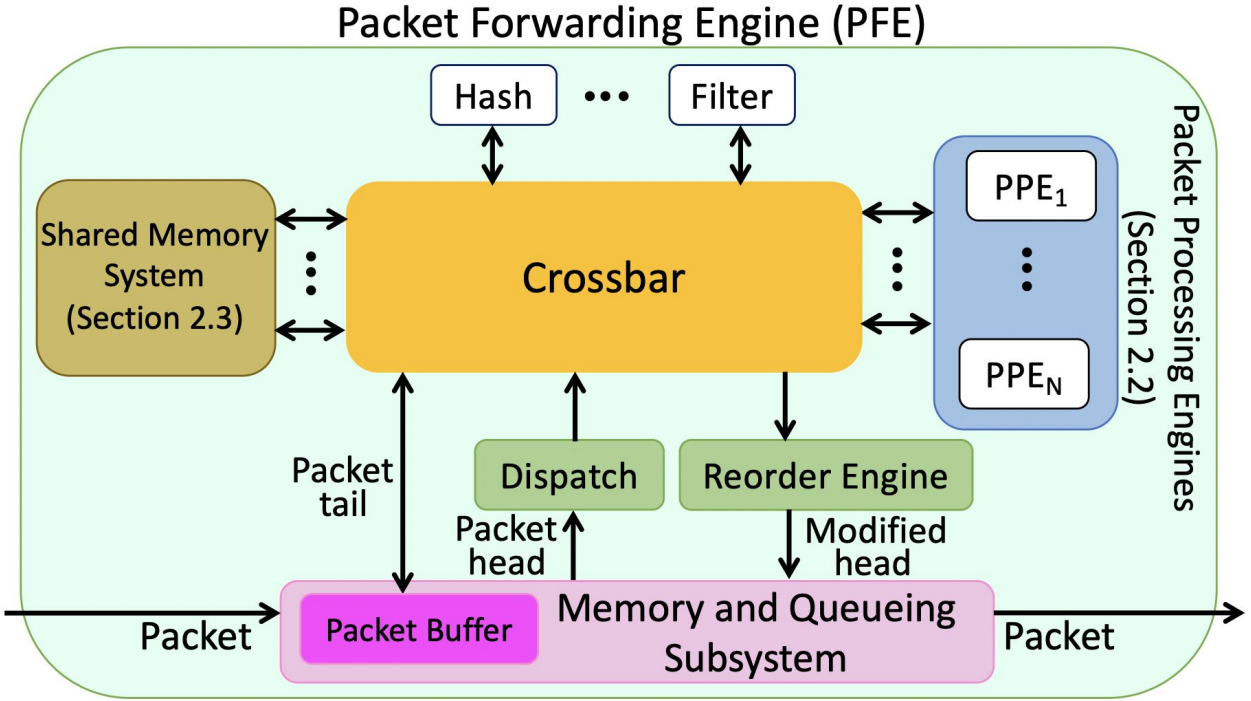
PHV = Packet Header Vector =
the collection of all the header fields that are parsed from the packet and can be used later for processing

Once PHV for a packet is past Stage 1, Stage 1 can start processing the PHV of the next packet → Parallel Processing!



Pipelines vs. Run-to-completion on cores





How do we program these switches?

- The P4 language is the de-facto at the moment
  - Came out of the research on RMT switches
  - Is the language used for programming Tofino chips
  - Has an active and large community (academic and industry)
  - checkout https://p4.org/
- Its benefits and use cases have extend beyond programmable switching chips
  - Programming other components of the network
  - Testing and verification of fixed-function switches (e.g., at Google)
  - …
- Other language/extensions have been proposed as well
  - NPL (Broadcom)
  - Domino, Mantis, MicroP4, P4All, …

What are some research questions to explore?

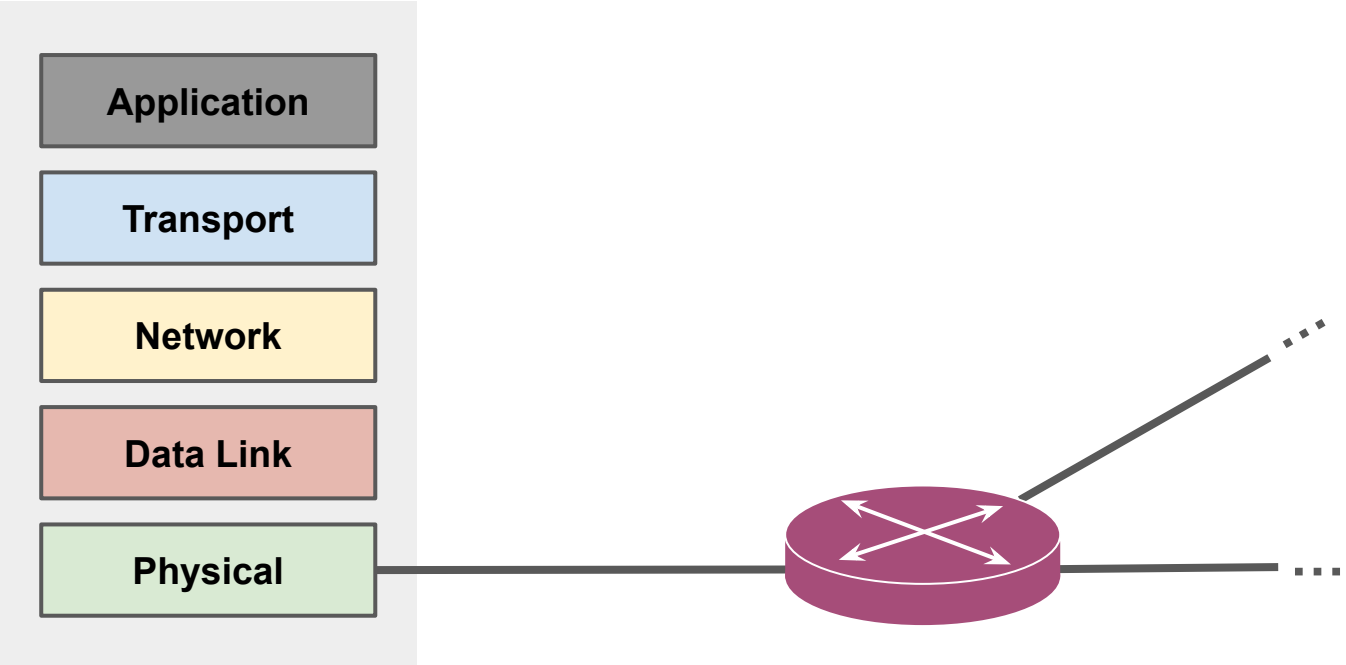
- What is the set of functionality that, if placed in the switch, will significantly benefit the network (and the applications using it) as a whole?
  - The answer could change from network to network
  - Are there some common sets of primitives?
- Can current switch architectures support them at high-speed?
  - If not, what changes are necessary?
- Do we have the right programming abstraction for implementing them?
- Heads-up: this has been studied quite a bit in the past ten years.
  - That doesn't mean all the problems are solved though.

What are some research questions to explore?

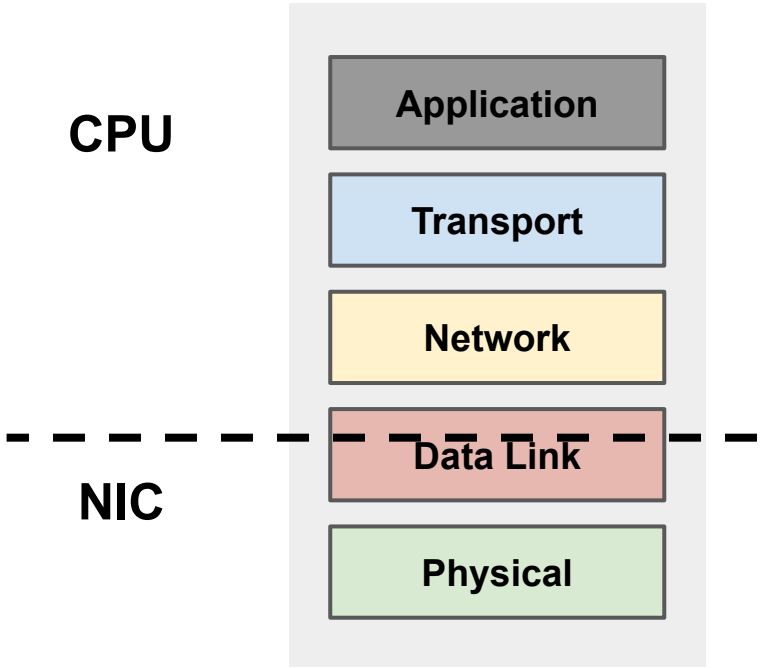
- Runtime programmability
  - Can you re-program the switch while it is still processing traffic?
  - Otherwise, you'll have to drain the switch, change the program, and put the switch back on the path.
- Has lead to re-thinking the hardware architecture and programming abstractions.

Programmable Network Interface Cards (NICs)

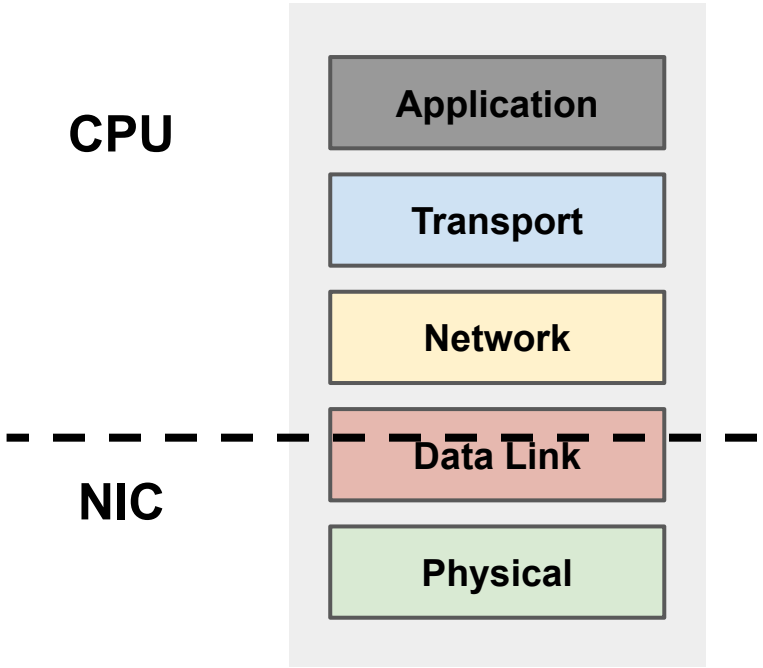
End-Point network stack



Network interface cards (NICs)



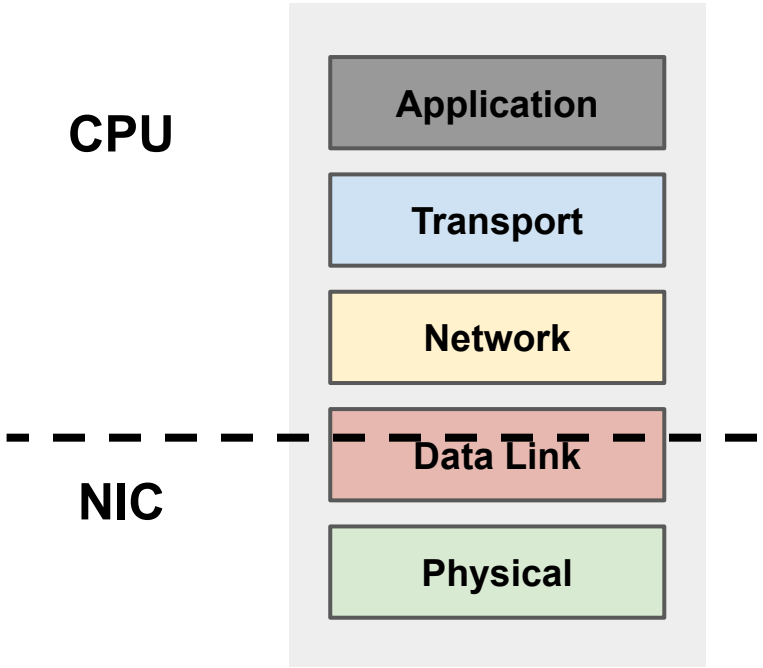
Network interface cards (NICs)



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

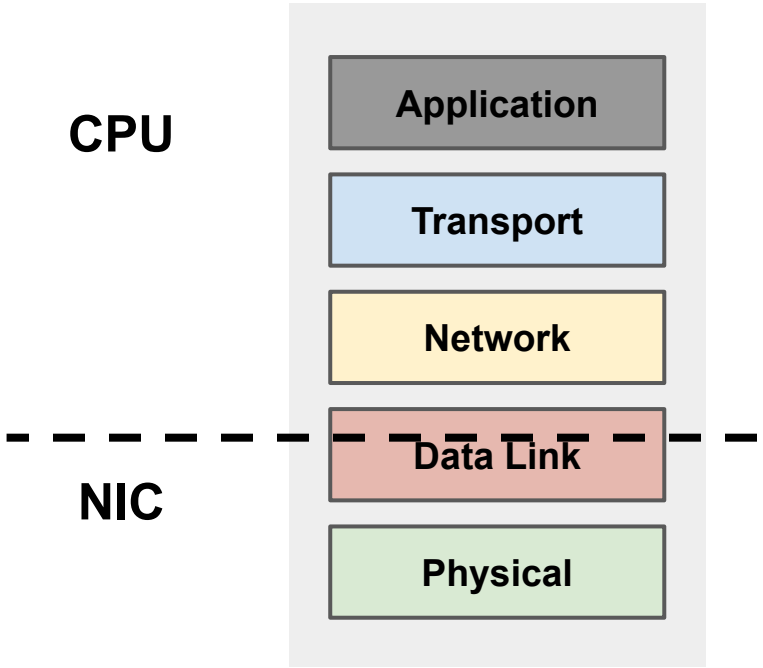
Network interface cards (NICs)



On receive (ingress)

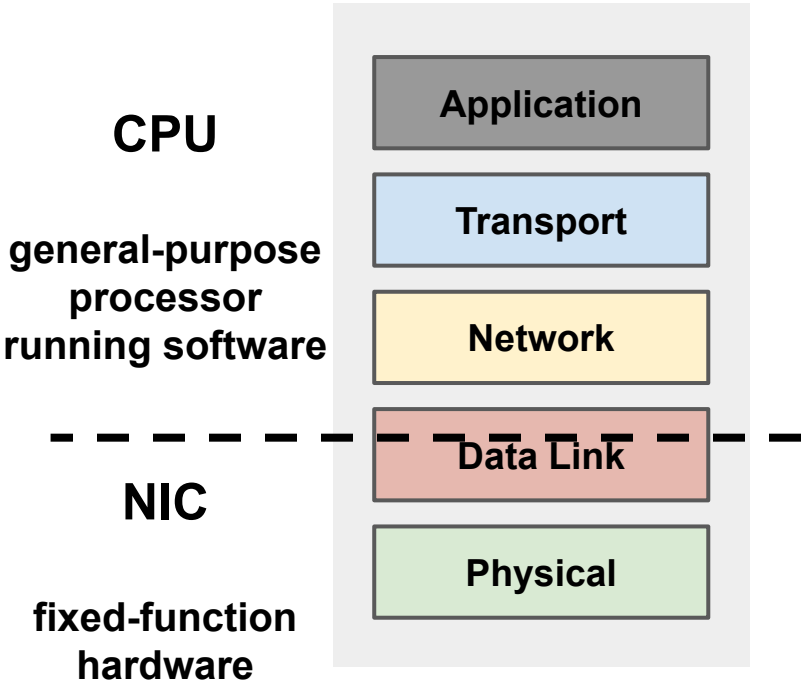
- The NIC turns bits into packets
- Packets are sent to the host over PCIe
- The host CPU processes packets and delivers them to applications

Network interface cards (NICs)



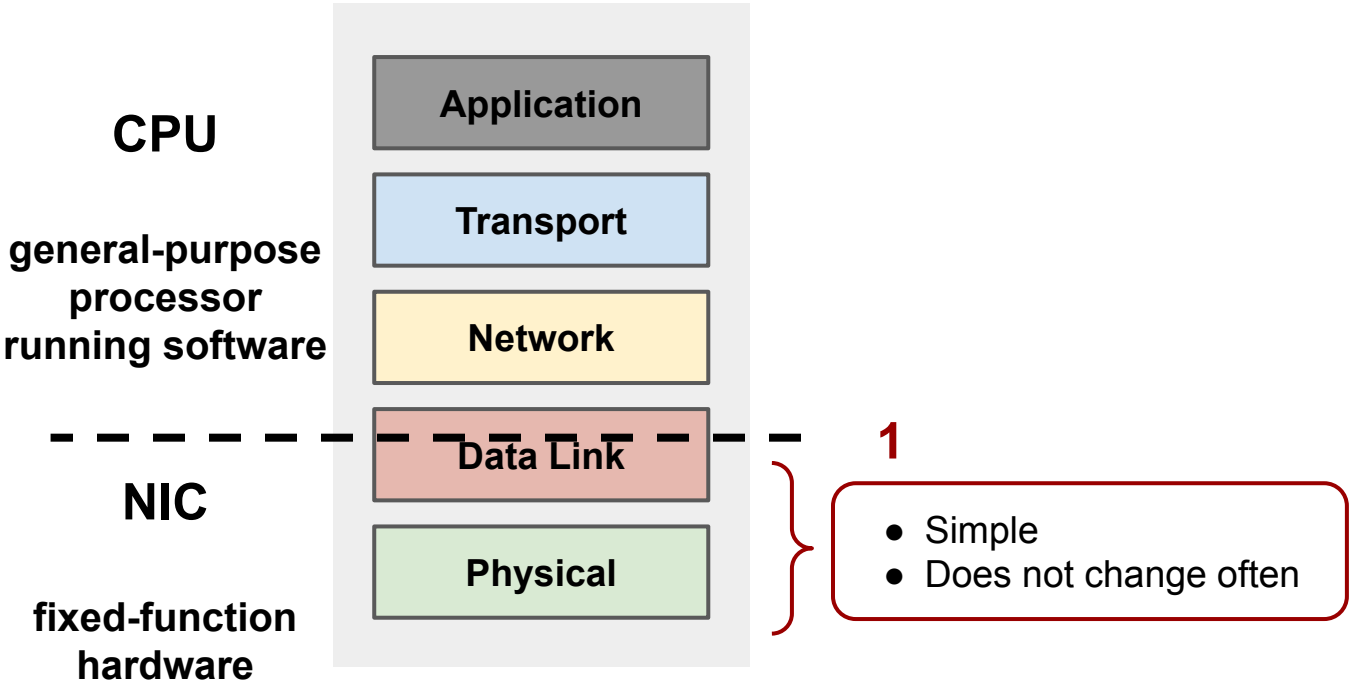
Great division of labor!

Network interface cards (NICs)

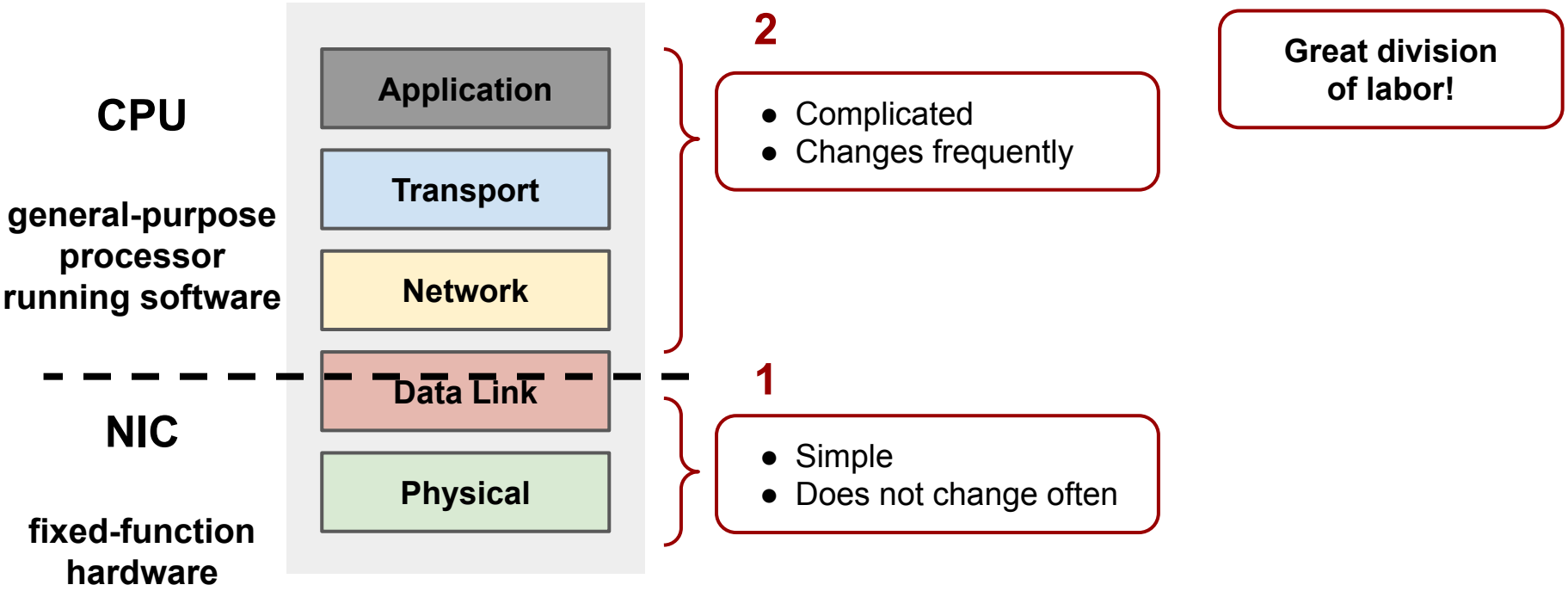


Great division of labor!

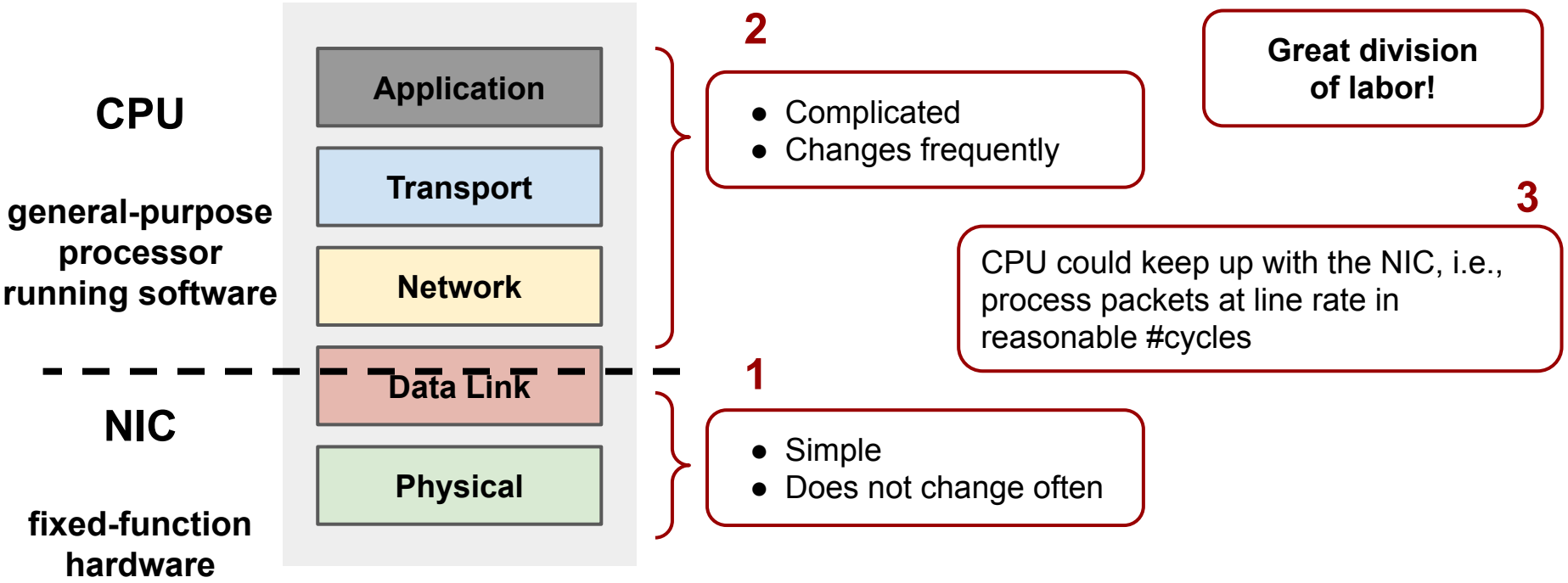
Network interface cards (NICs)



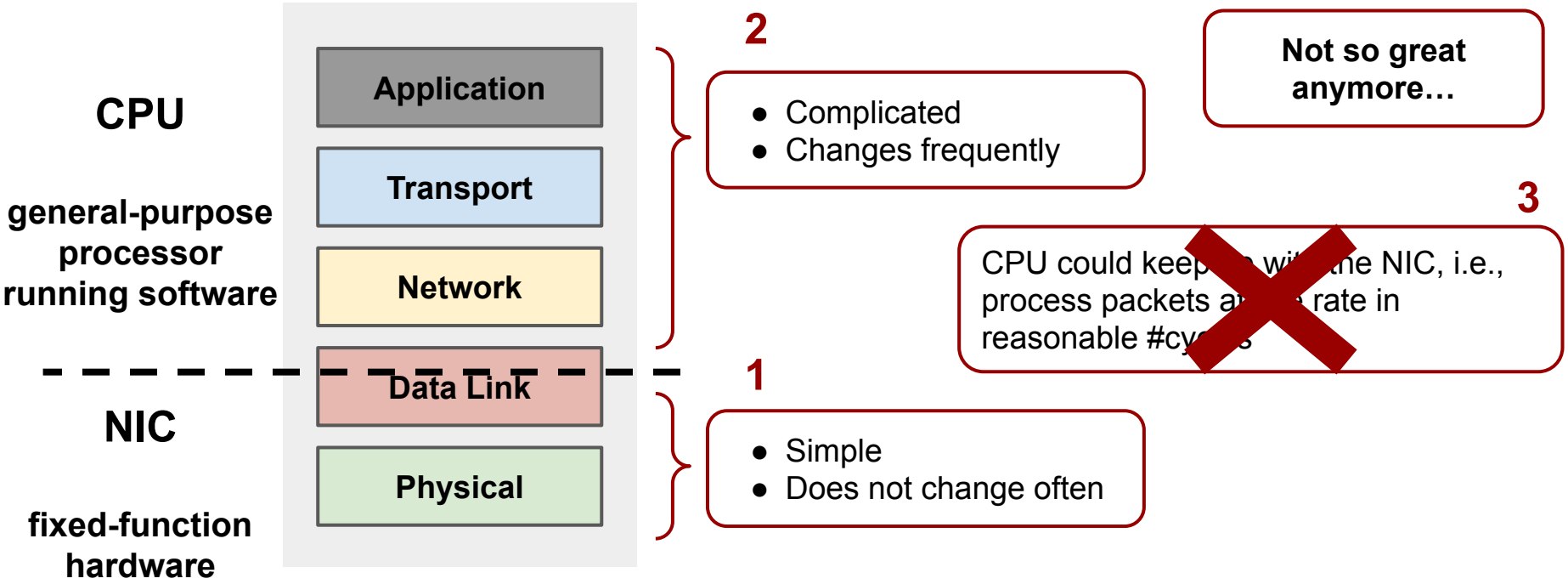
Network interface cards (NICs)



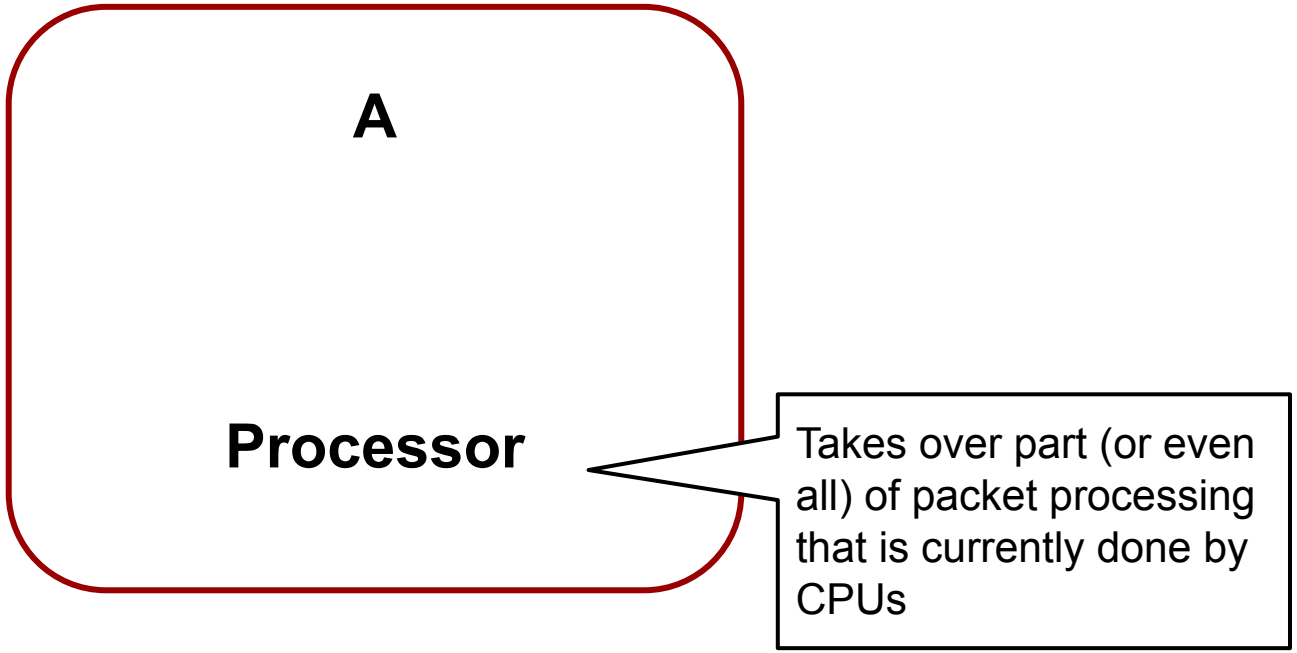
Network interface cards (NICs)



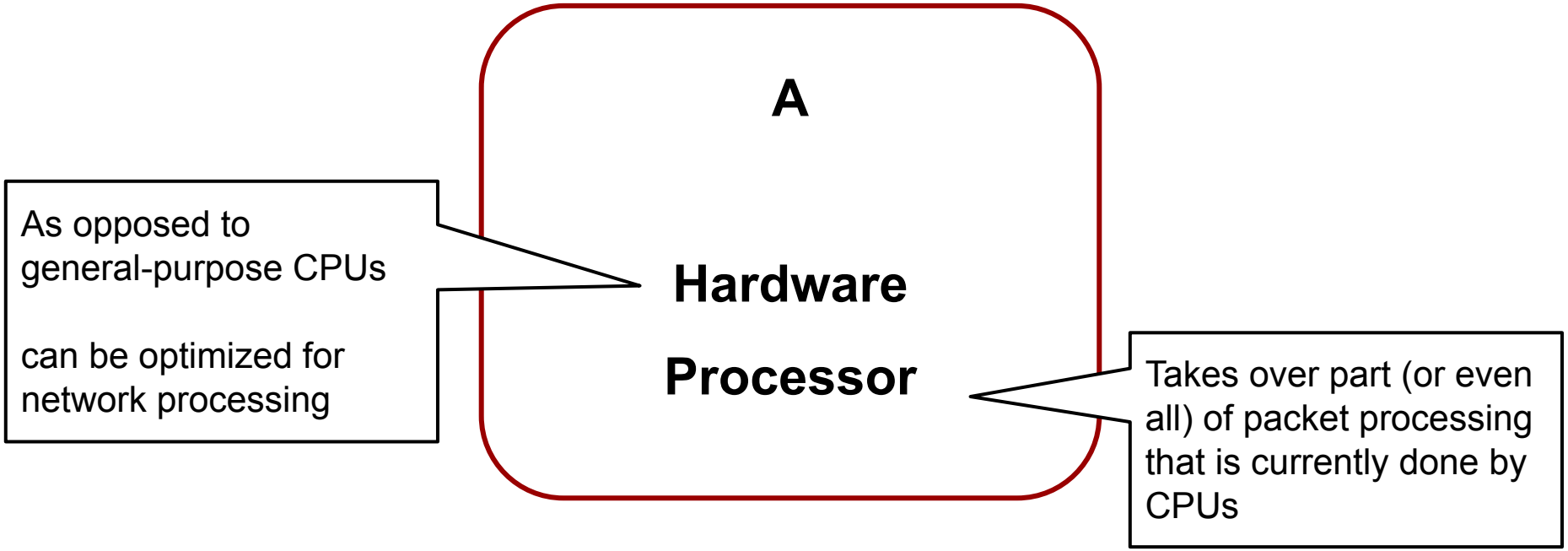
Network interface cards (NICs)



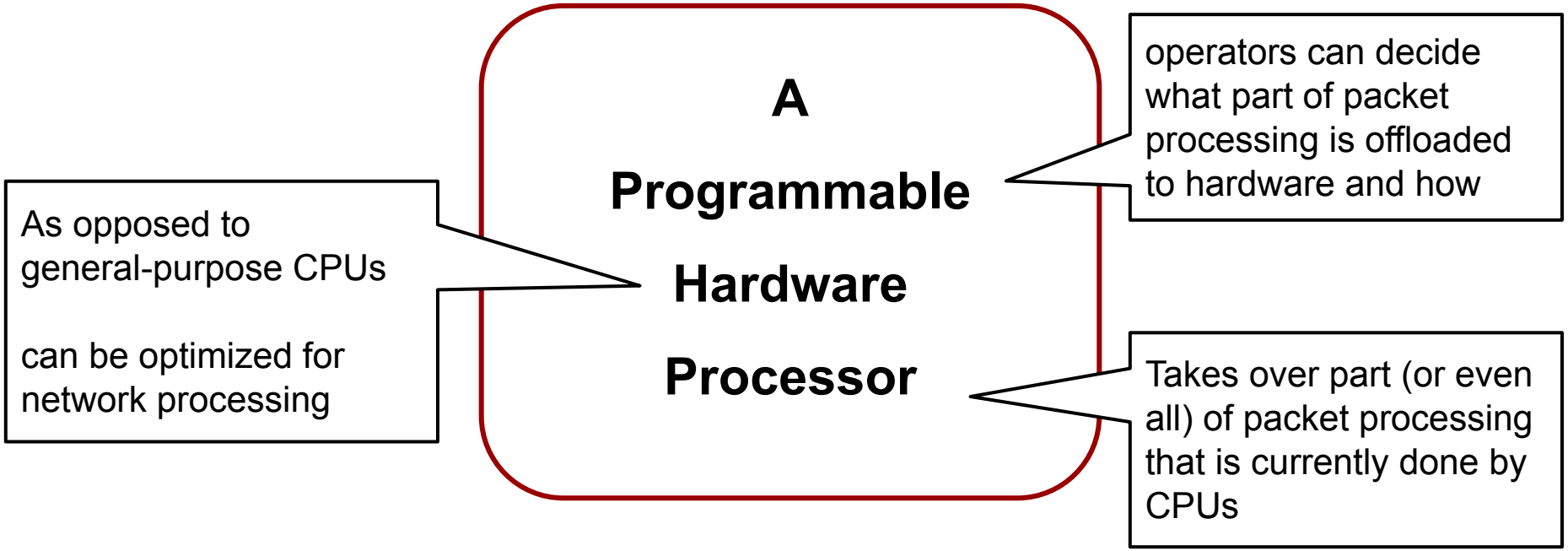
Solution?


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

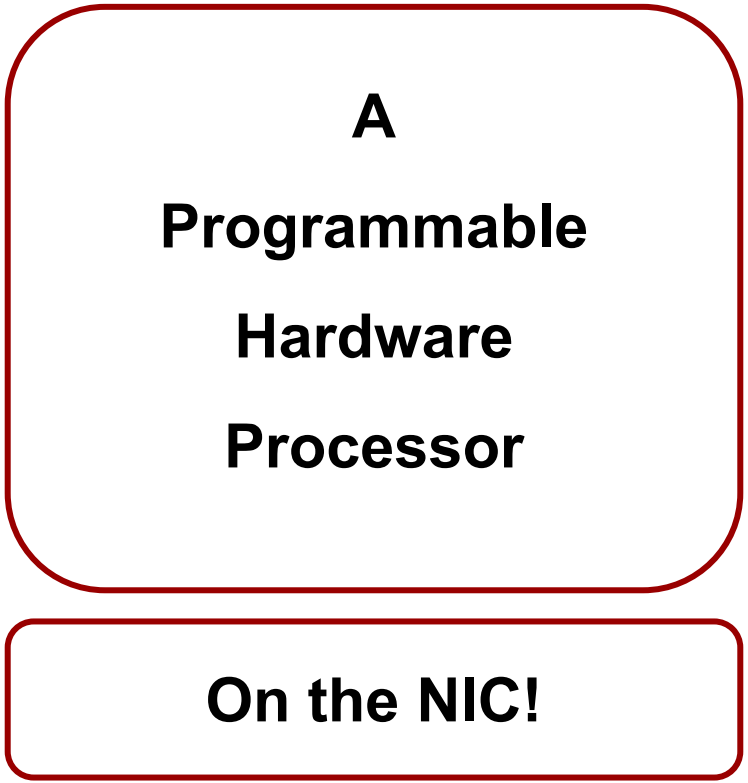
Solution?


A
Hardware
Processor
As opposed to general-purpose CPUs
can be optimized for network processing
Takes over part (or even all) of packet processing that is currently done by CPUs

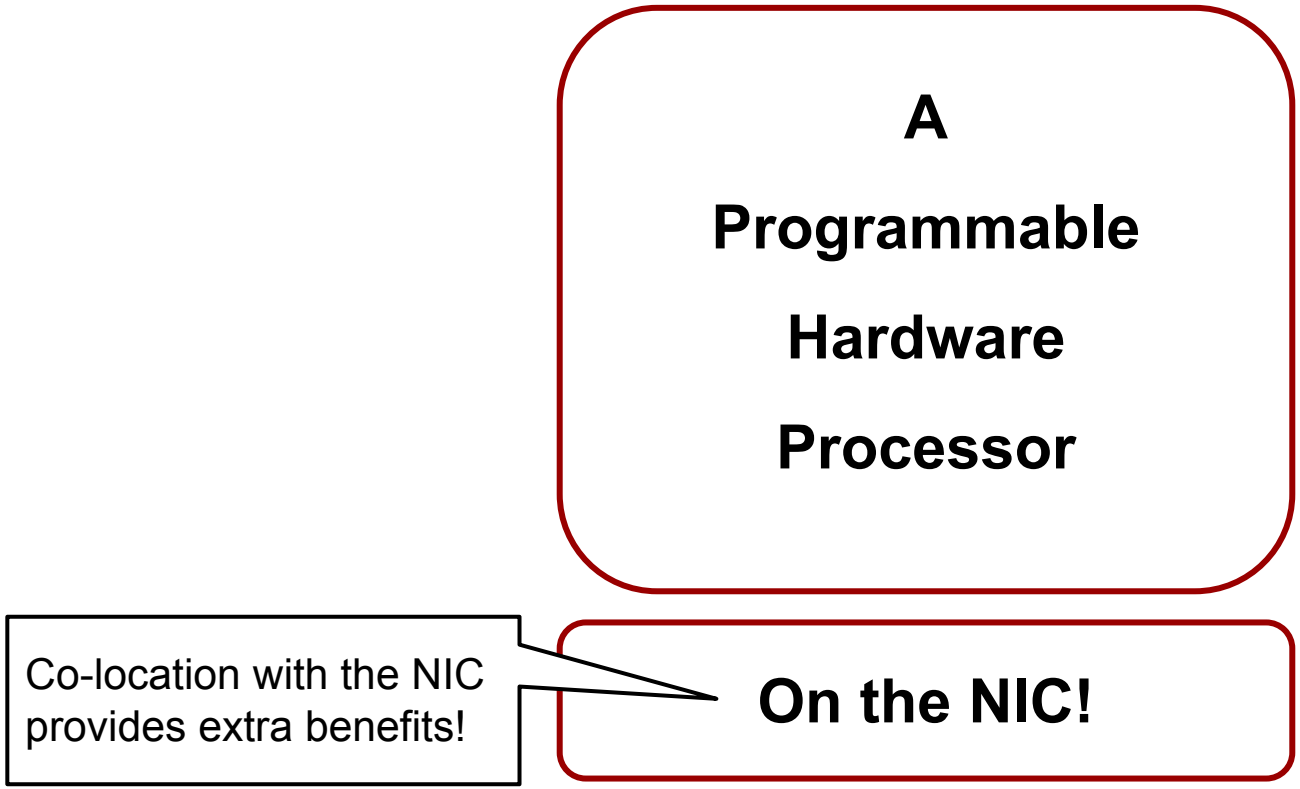
Solution?


A
Programmable
Hardware
Processor
As opposed to general-purpose CPUs
can be optimized for network processing
operators can decide what part of packet processing is offloaded to hardware and how
Takes over part (or even all) of packet processing that is currently done by CPUs

Solution?



Solution?


A
Programmable
Hardware
Processor
On the NIC!
Co-location with the NIC provides extra benefits!

Smart NICs!

A regular NIC

+

A programmable domain-specific hardware

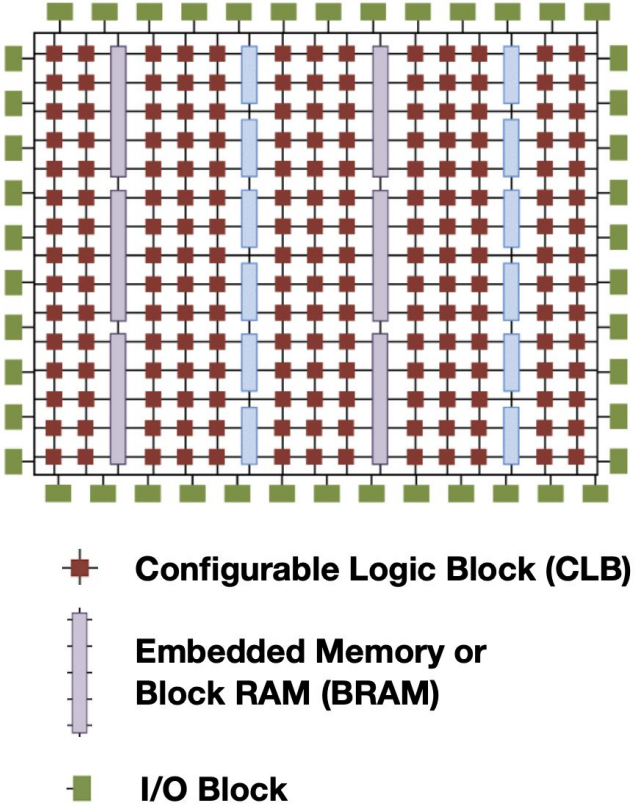
A closer look at the hardware

- Field Programmable Gate Arrays (FPGAs)
- Multi-Core Systems on Chip (SoCs)
- P4-Programmable pipelines
- Or combinations of the above …

FPGAs

- An FPGA is a collection of small configurable logic and memory blocks

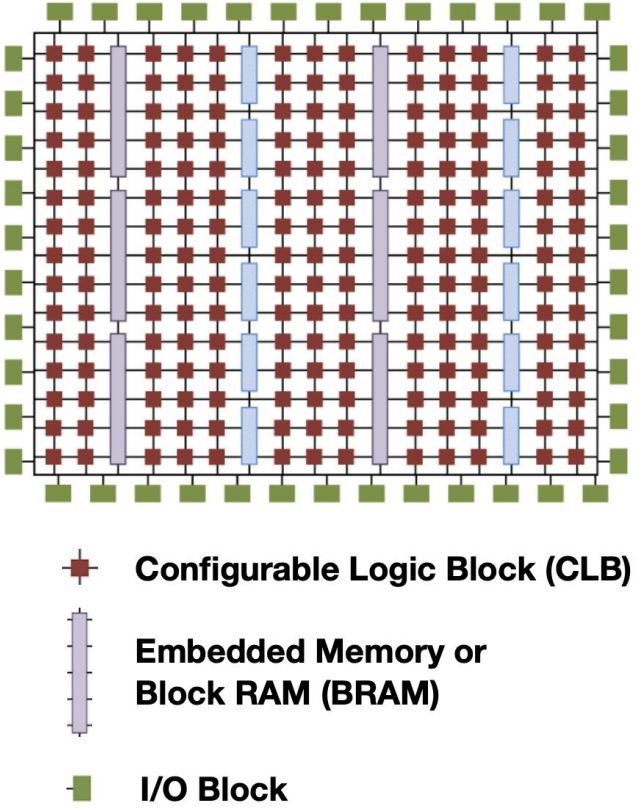
- Programmers can write code to assemble these blocks to perform their desired processing



FPGAs

Why is an FPGA a popular hardware choice for smart NICs?

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



Multi-Core Systems on Chip

- A “small” computer on a single chip
- Includes (light-weight) processing cores and a memory hierarchy
- Why is it a popular hardware choice for smart NICs?
  - Programming model is close to software
  - Cores (and the architecture) can be specialized for network processing

FPGAs vs SoCs for network processing

	FPGAs	SoCs
Hardware Architecture	**Reconfigurable hardware** and therefore can be highly customized for the intended packet processing	The cores' instruction set and memory architecture is fixed and is therefore less customizable
Programming Model	Hardware description languages (e.g., Verilog) ↓ Harder to program	C-like languages ↓ Easier to program
Performance	Higher throughput lower latency *	Lower throughput higher latency *

* For most kinds of network processing

Side note #1

- Smart NICs can (and do) have fixed-function blocks
- These blocks are optimized hardware implementations of common packet processing functionality.
  - e.g., encryption, hashing, certain common protocols
- A fully ASIC-based NIC can still be considered a "Smart NIC"
  - as long as it supports more complex functionality than a traditional NIC

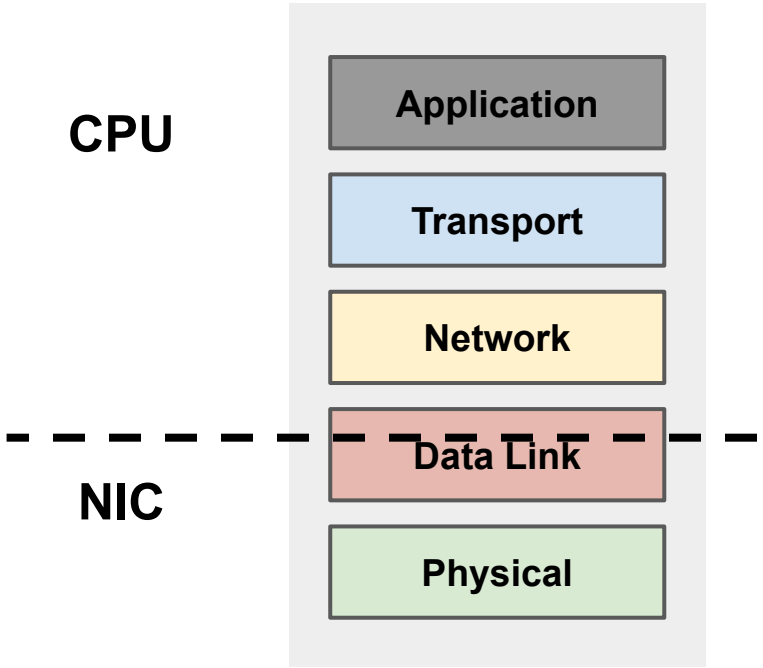
Side note #2

- In industry, Smart NICs have various names
  - Data Processing Unit (DPU)
  - Infrastructure Processing Unit (IPU)
  - …
- They are all conceptually the same.
  - Accelerators of compute and communication at the interface card.

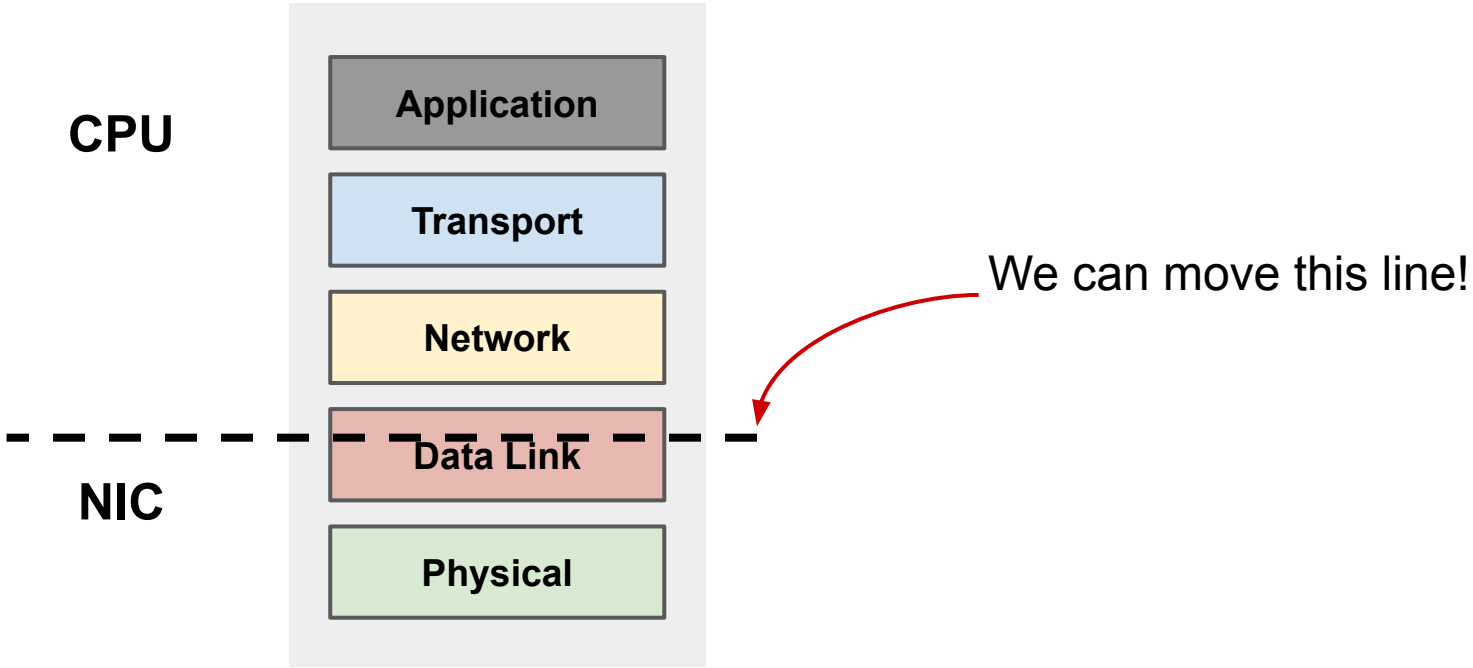
Today's Smart NICs / DPUs/ IPUs /…

- A combination of the following kinds of hardware
  - FPGA
  - SoC
  - P4-programmable pipelines
  - Fixed-function accelerators

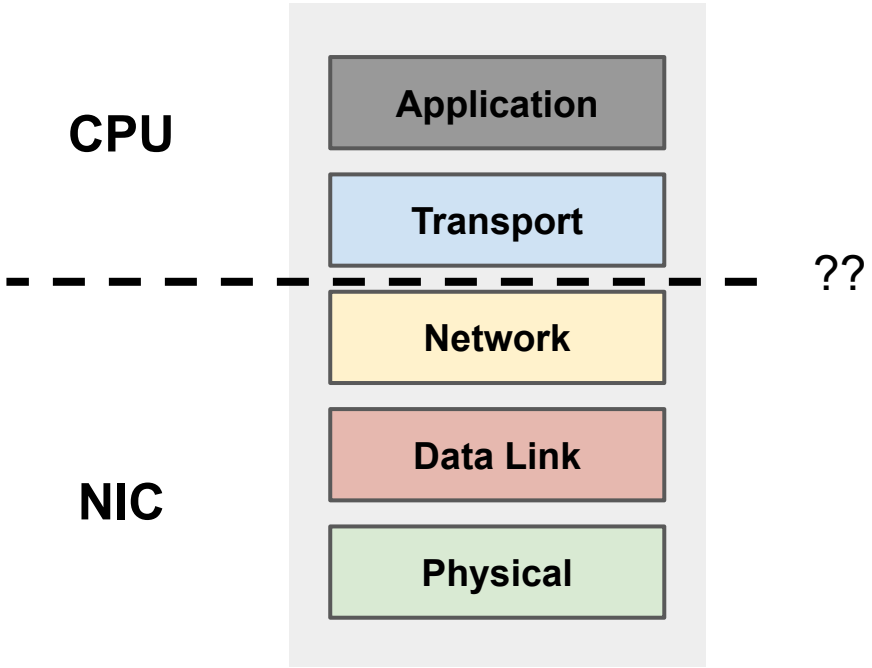
What can we do with our "Smart" NICs?



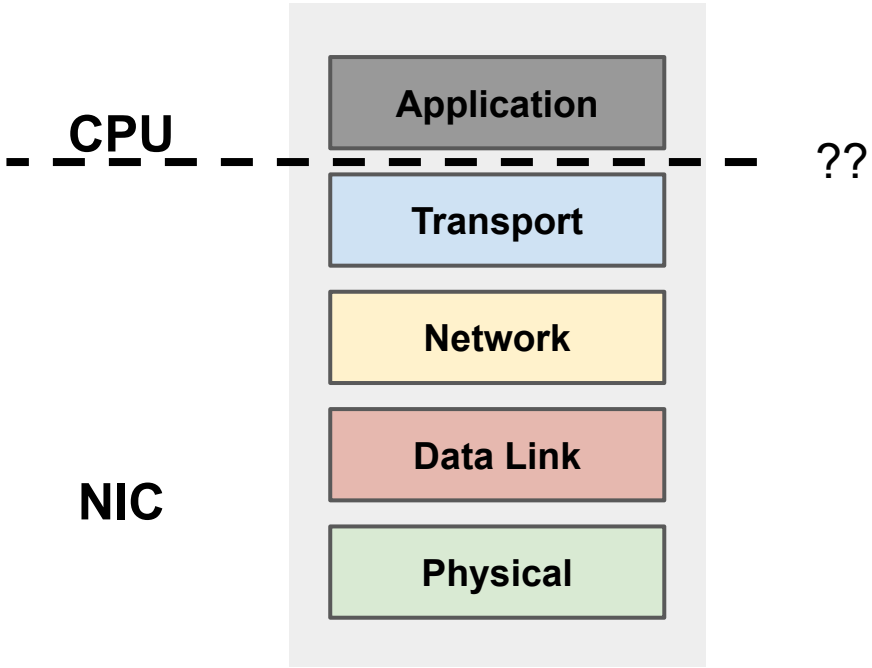
What can we do with our "Smart" NICs?



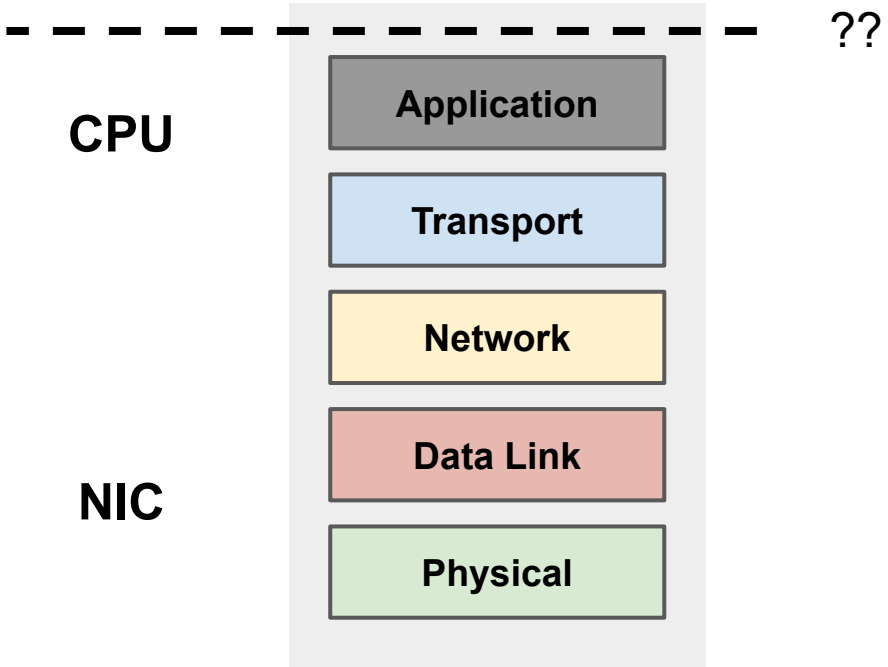
What can we do with our "Smart" NICs?



What can we do with our "Smart" NICs?



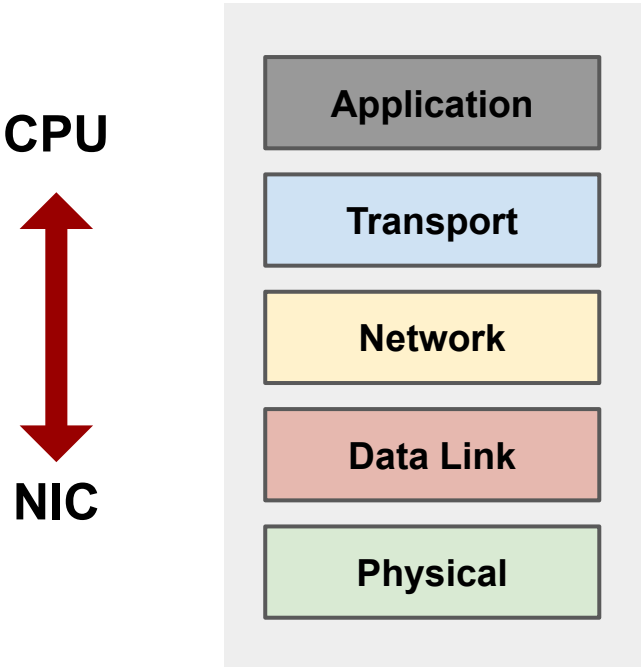
What can we do with our "Smart" NICs?



Offloading the network stack (and beyond!) to the NIC

- Hypervisor vSwitch: AccelNet (NSDI’18)
- Packet scheduling: PIEO (SIGCOMM’19), Loom (NSDI’19)
- Network functions: ClickNP (SIGCOMM’16), FlowBlaze (NSDI’19)
- Transport: Tonic (NSDI’20)
- Even applications: iPipe (SIGCOMM’19), KV-Direct (SOSP’17), Bing web search ranking (ISCA’14)

What can we do with our "Smart" NICs?



We can also optimize the movement of packets between the CPU and the NIC (FlexNIC, ASPLOS'16)

Can we use the same programming model as switches?

- Maybe, but NICs and switches are quite different
- **Speed**: switches have to be faster
  - Switches process traffic for multiple end-points → **Tbps**
  - NICs process traffic for one end point → (10s to 100s of) **Gbps**
- **Functionality:** switches have more limited functionality
  - limited visibility (e.g., don't see both directions of a connection)
  - have to process packets faster.
  - more resource constraints (in contrast, NIC has access to host memory)

Programming abstractions for Smart NICs

- Still an open question!
- There is such a wider range of functionality people can and are interested in implementing on the NICs
- There are many different Smart NIC architectures
    - FPGAs, different kinds of SoCs, P4 pipelines, fixed-function blocks, combinations of these

Programming abstractions for Smart NICs

- Do we keep P4 and extend it?
- Or are there more common constructs specific to NIC processing that we can pull out and define a different programming language?

Compilation challenges

- Suppose we have a program describing the network processing we want to happen at the end point.
- We can have many different kinds of hardware at our disposal!
  - CPU, all the different hardware on the NIC, even GPUs
- How do we partition/distribute the functionality over these different kinds of hardware? What is the best offloading strategy? How do we know what kind of performance to expect from a certain offloading strategy?

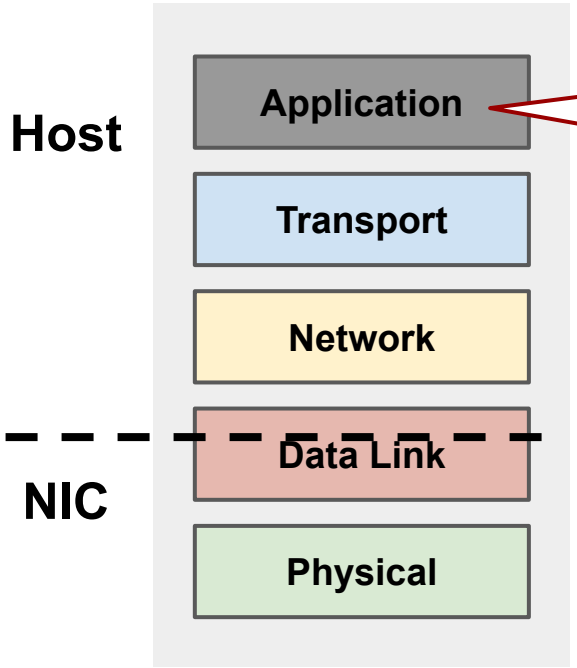
Software Packet Processing

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.

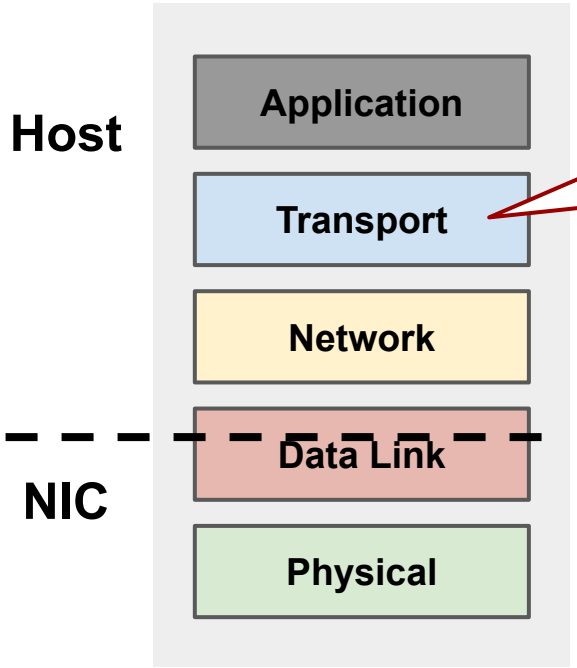
Software Packet Processing: in the Kernel

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.

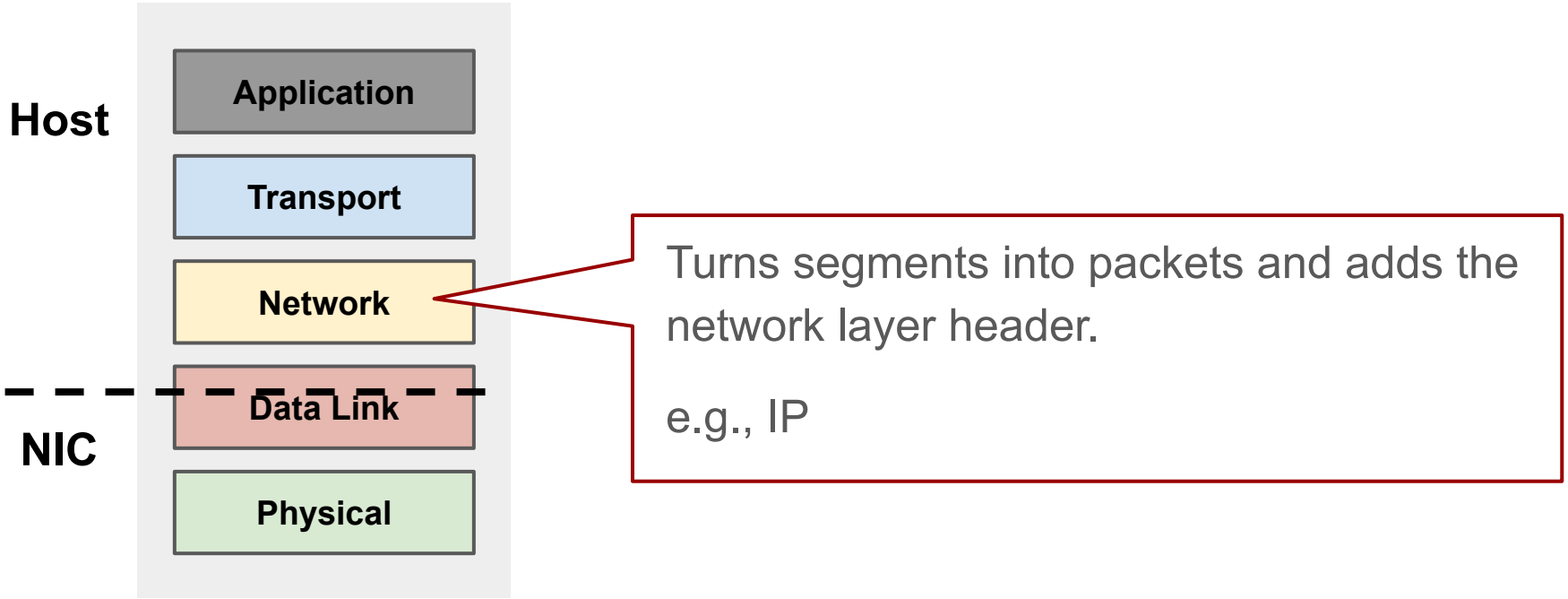
The (Linux) kernel network stack (simplified)



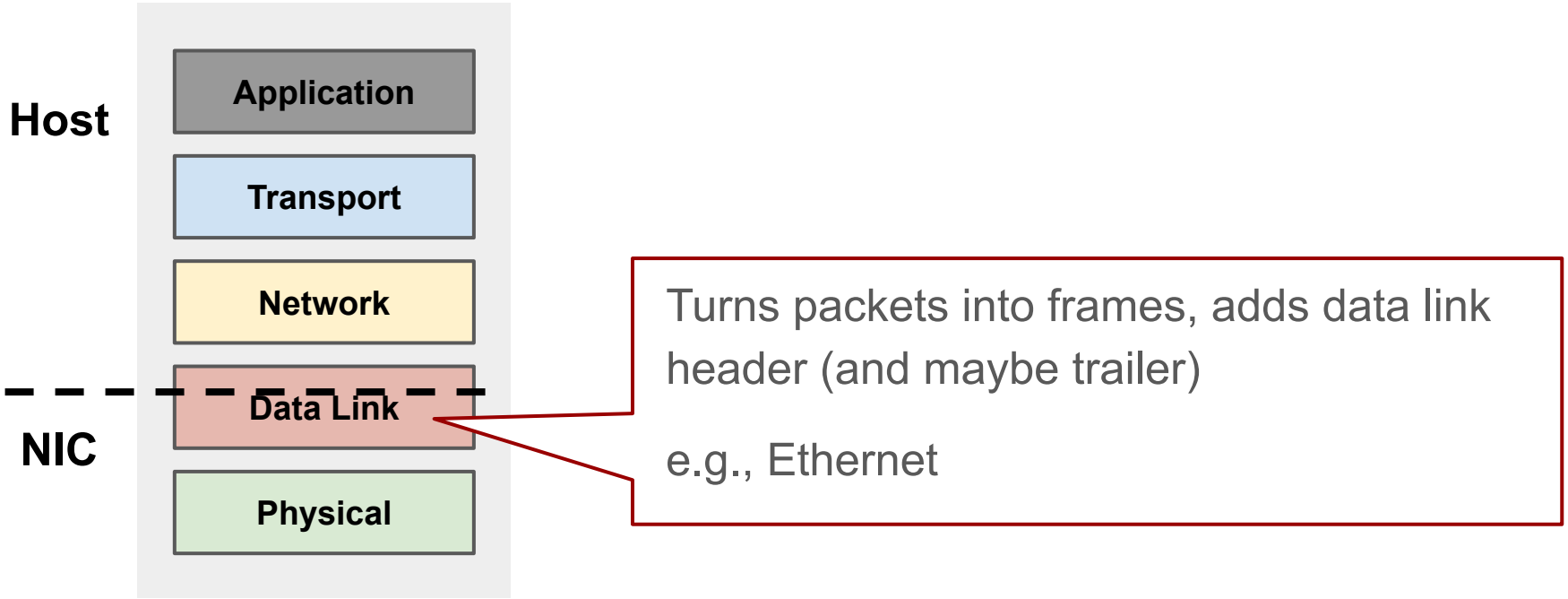
Breaks up the socket data into *segments*, adds the transport layer header

e.g., TCP and UDP

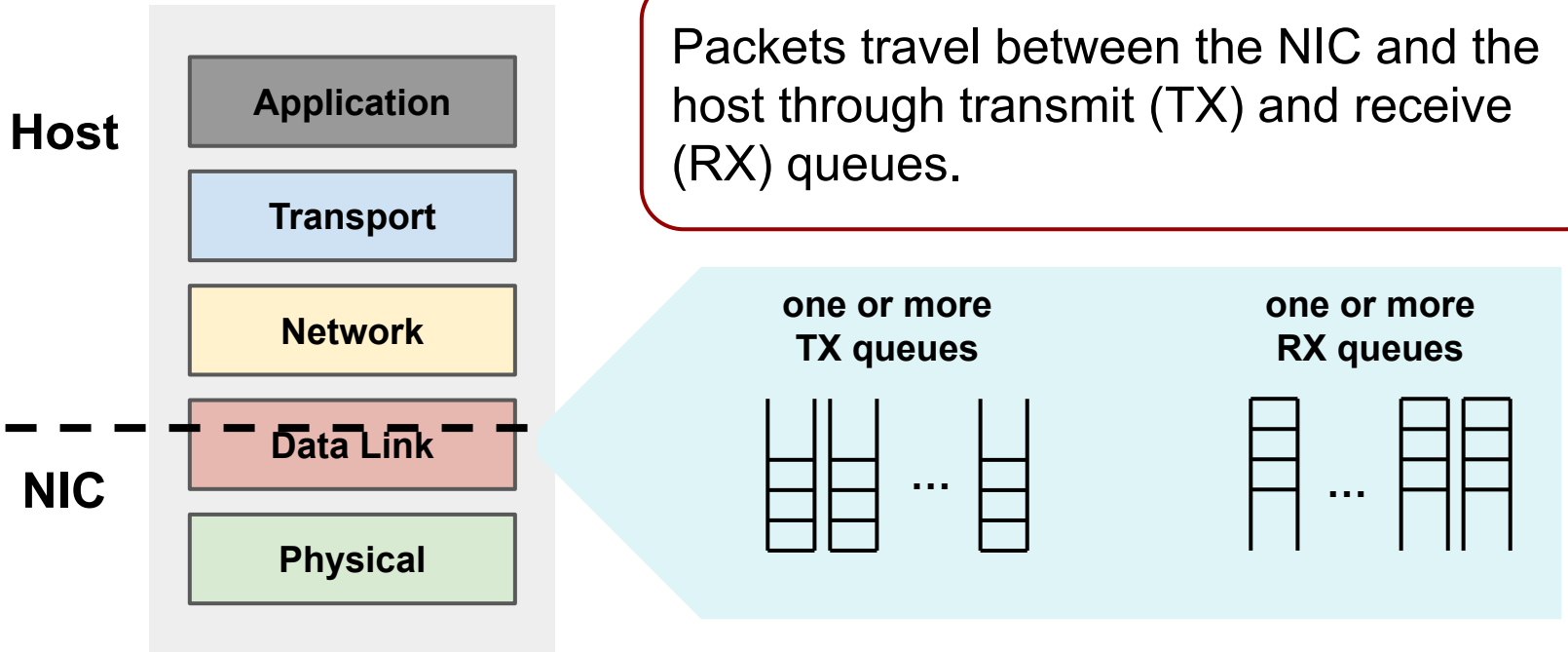
The (Linux) kernel network stack (simplified)



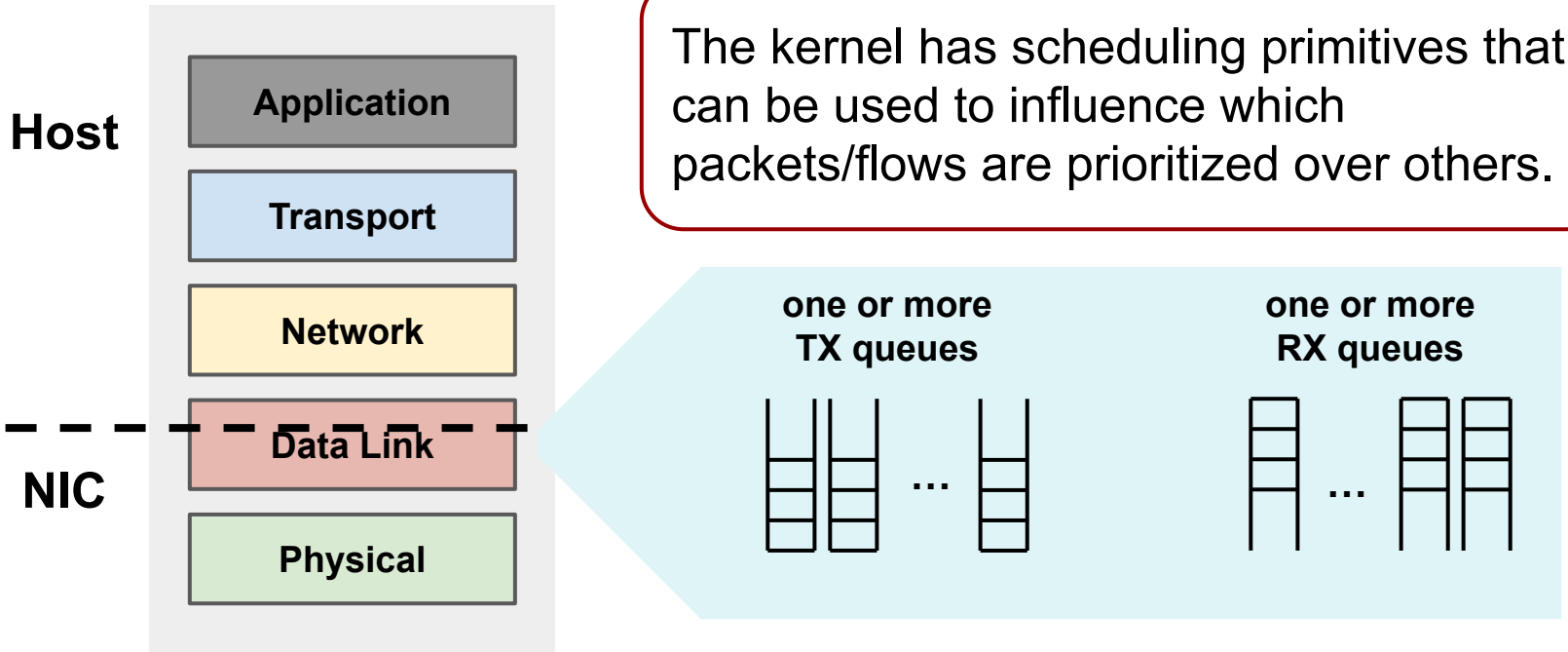
The (Linux) kernel network stack (simplified)



The (Linux) kernel network stack (simplified)

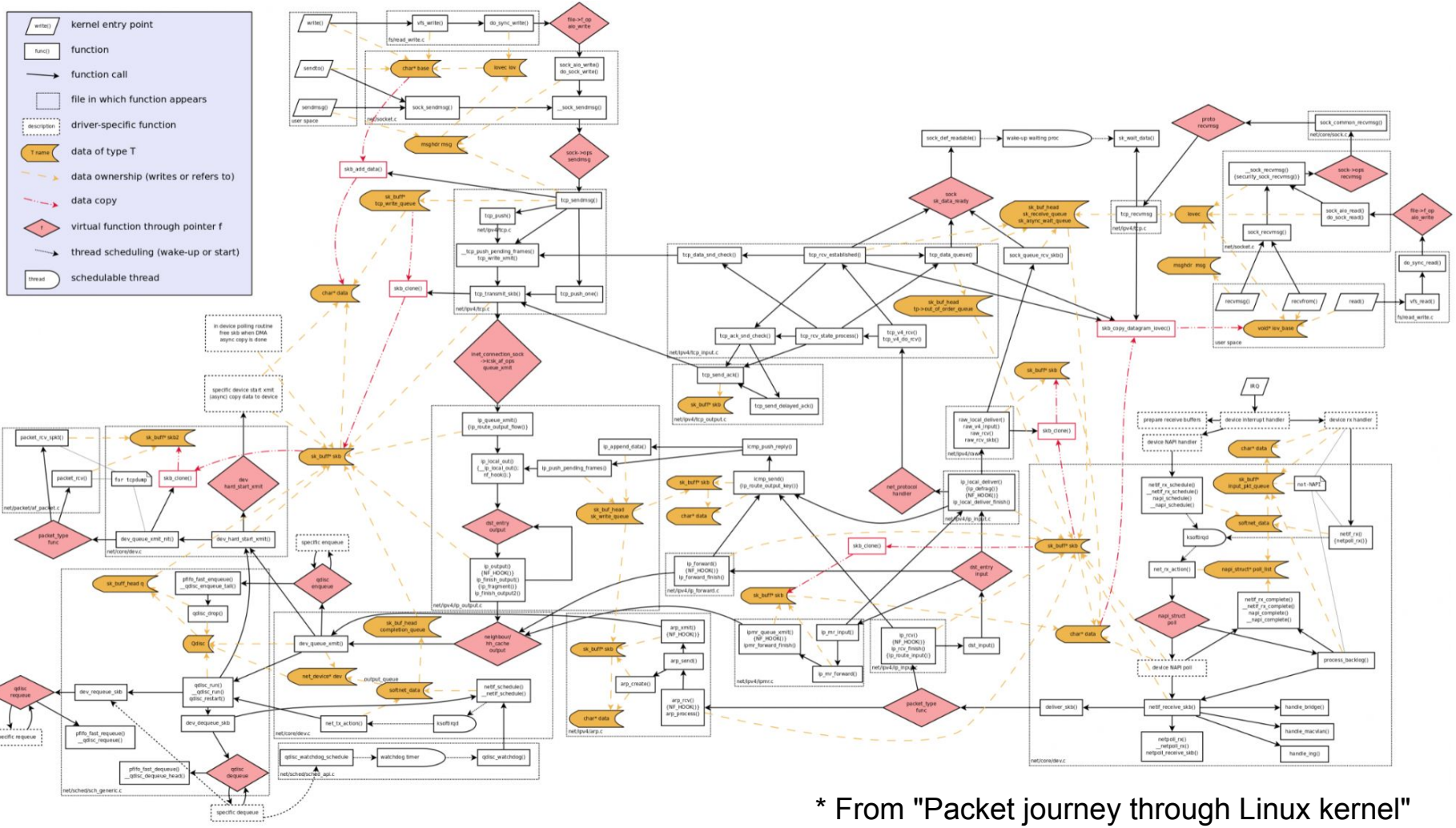


The (Linux) kernel network stack (simplified)



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 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)
  - …

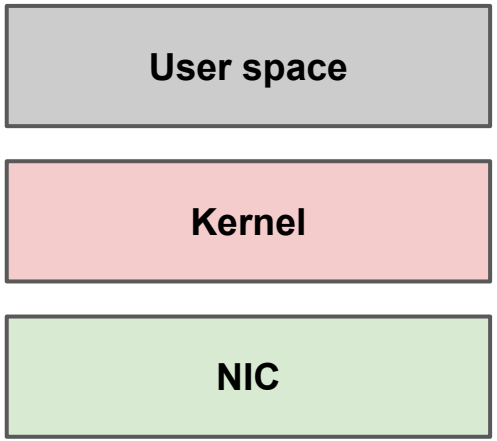
Looking Forward

- 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?

Software Packet Processing: Kernel-Bypass

Kernel Bypass

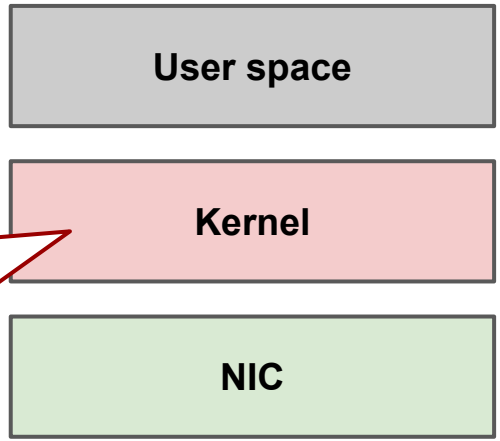
- What if we could write all the packet processing code in a regular program in user space?



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.



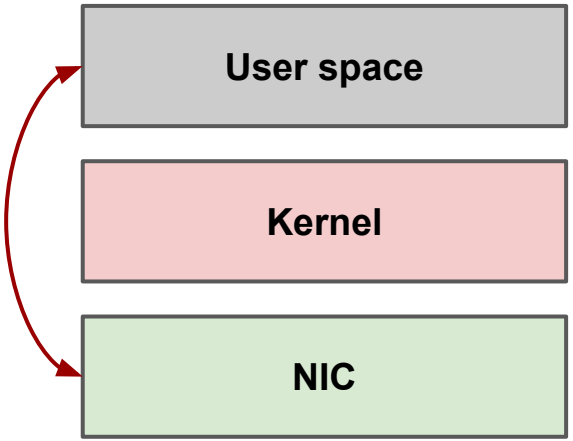
Kernel Bypass

- 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

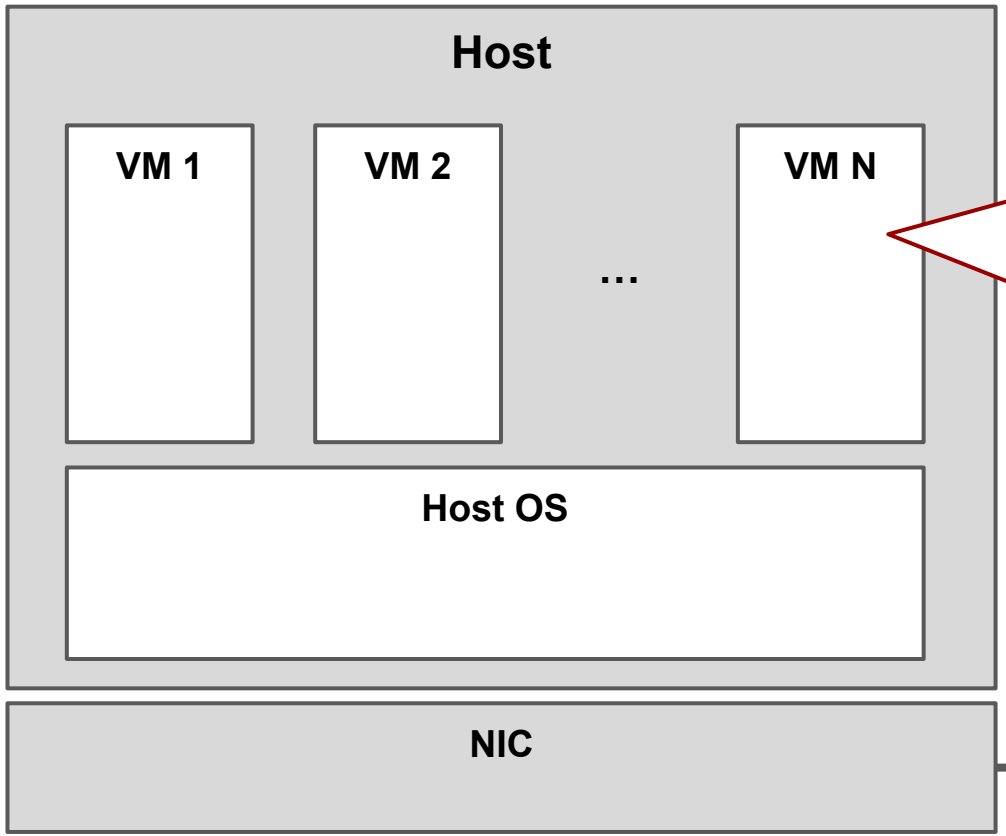
Kernel Bypass - Challenges

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

Software Packet Processing: In Virtualized Platforms

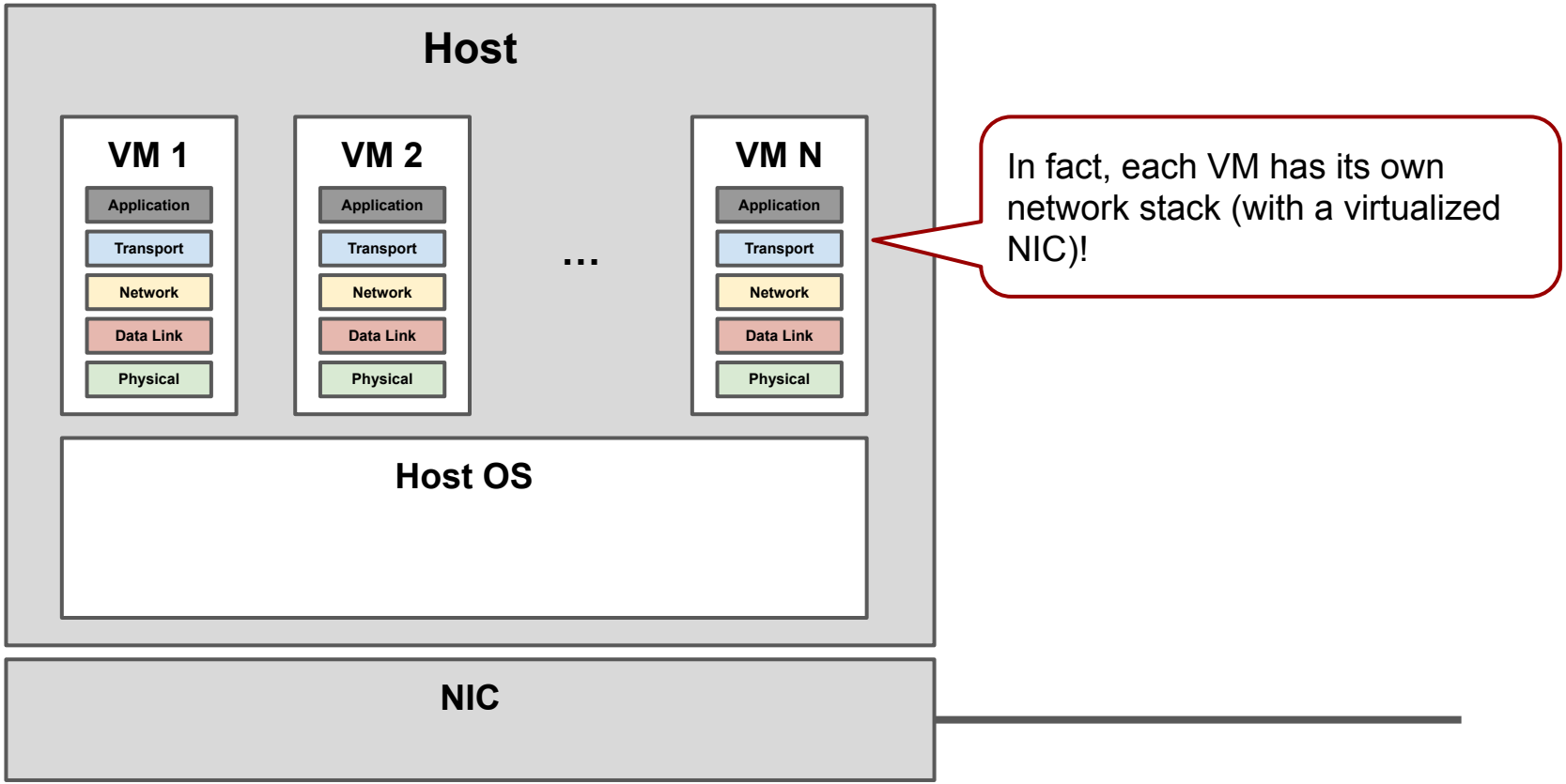
Server Virtualization



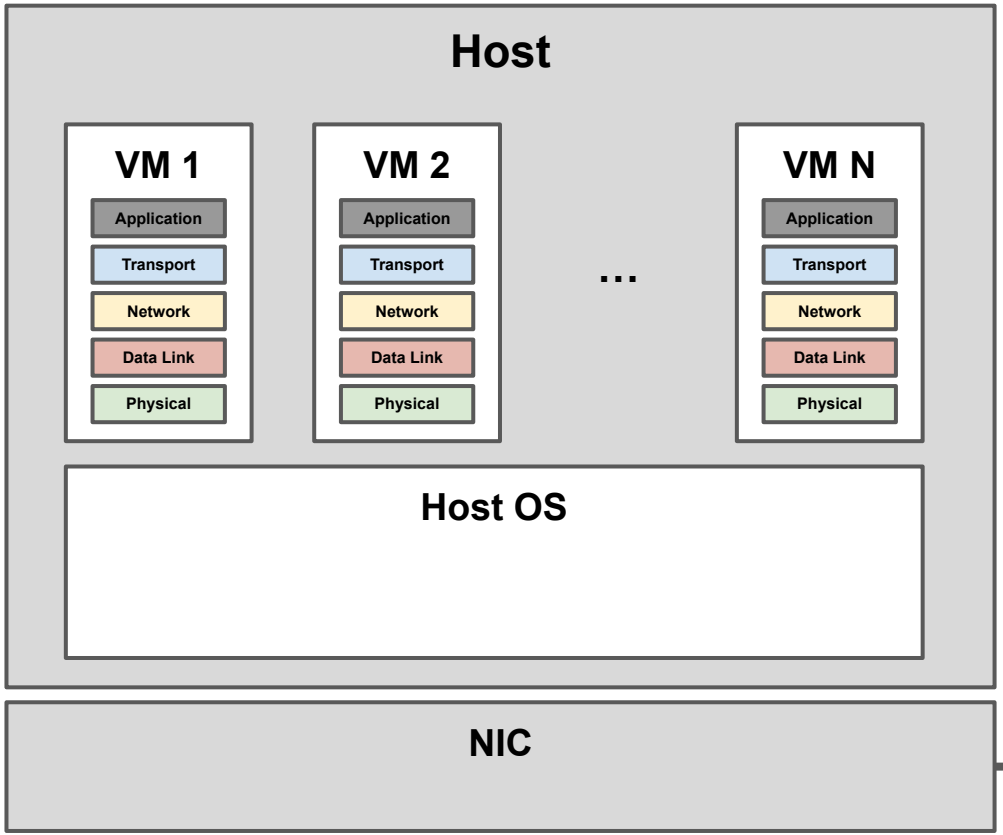
Each VM provides an illusion of having a standalone server.

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

Server Virtualization

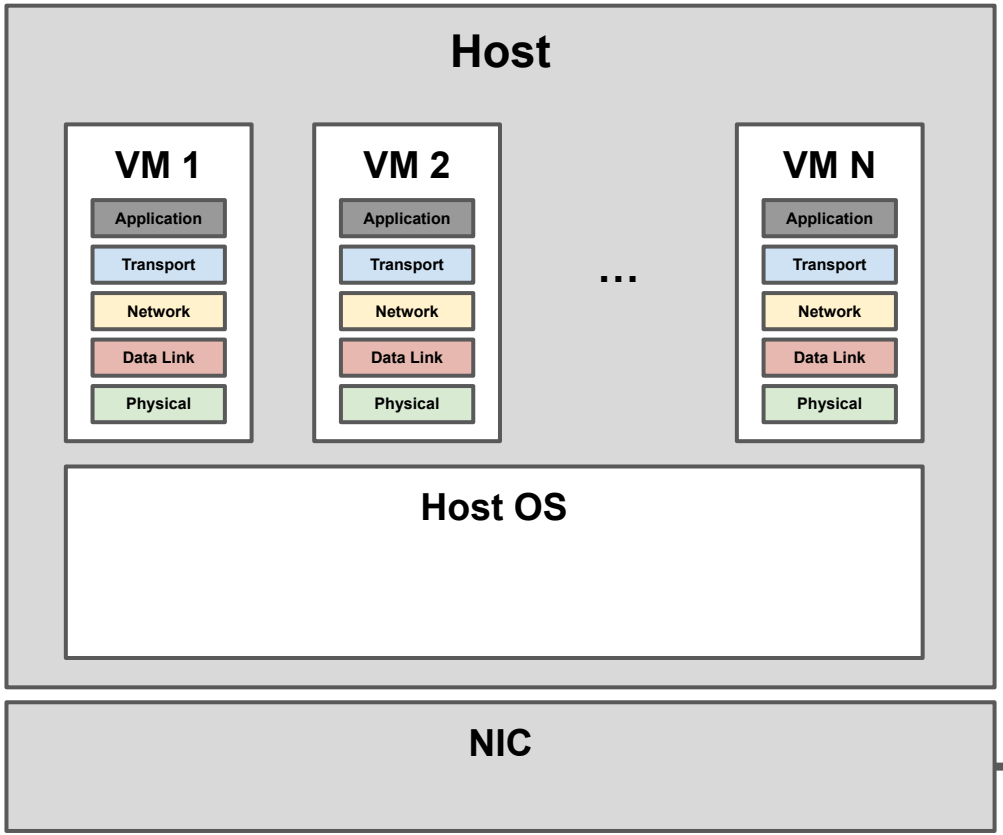


Server Virtualization



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

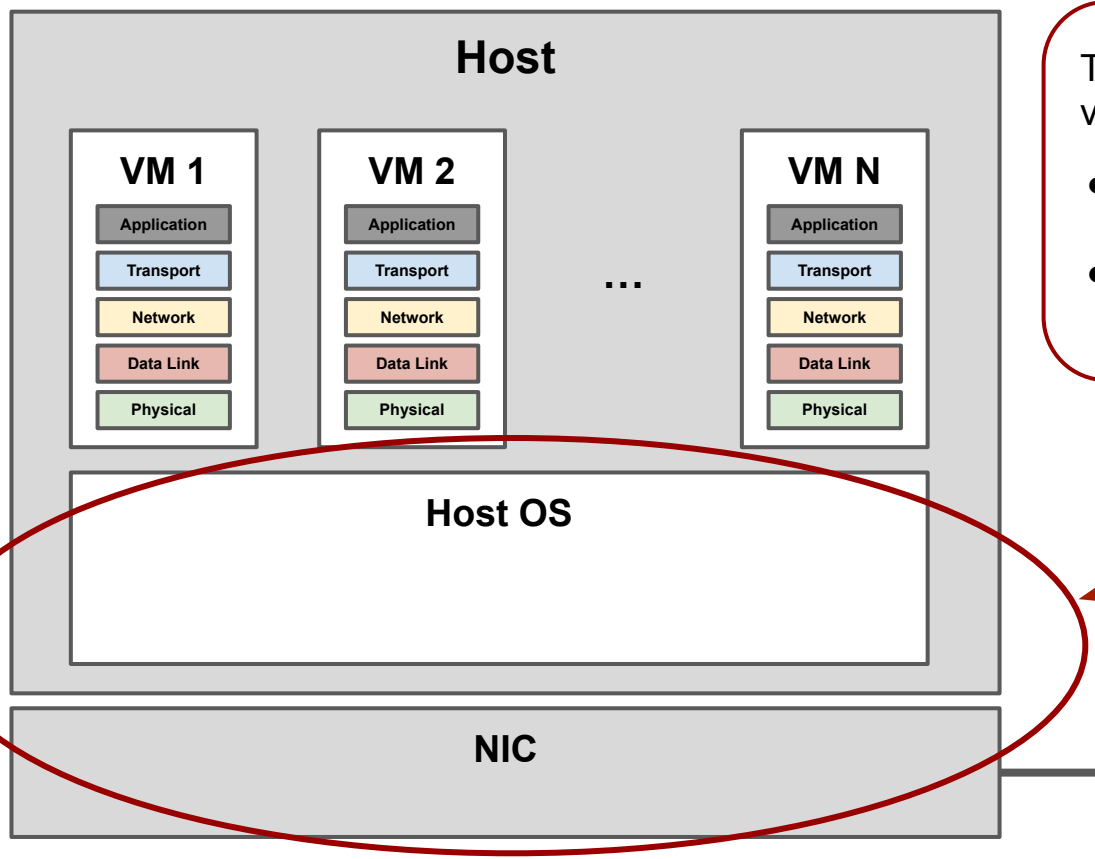
Server Virtualization



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

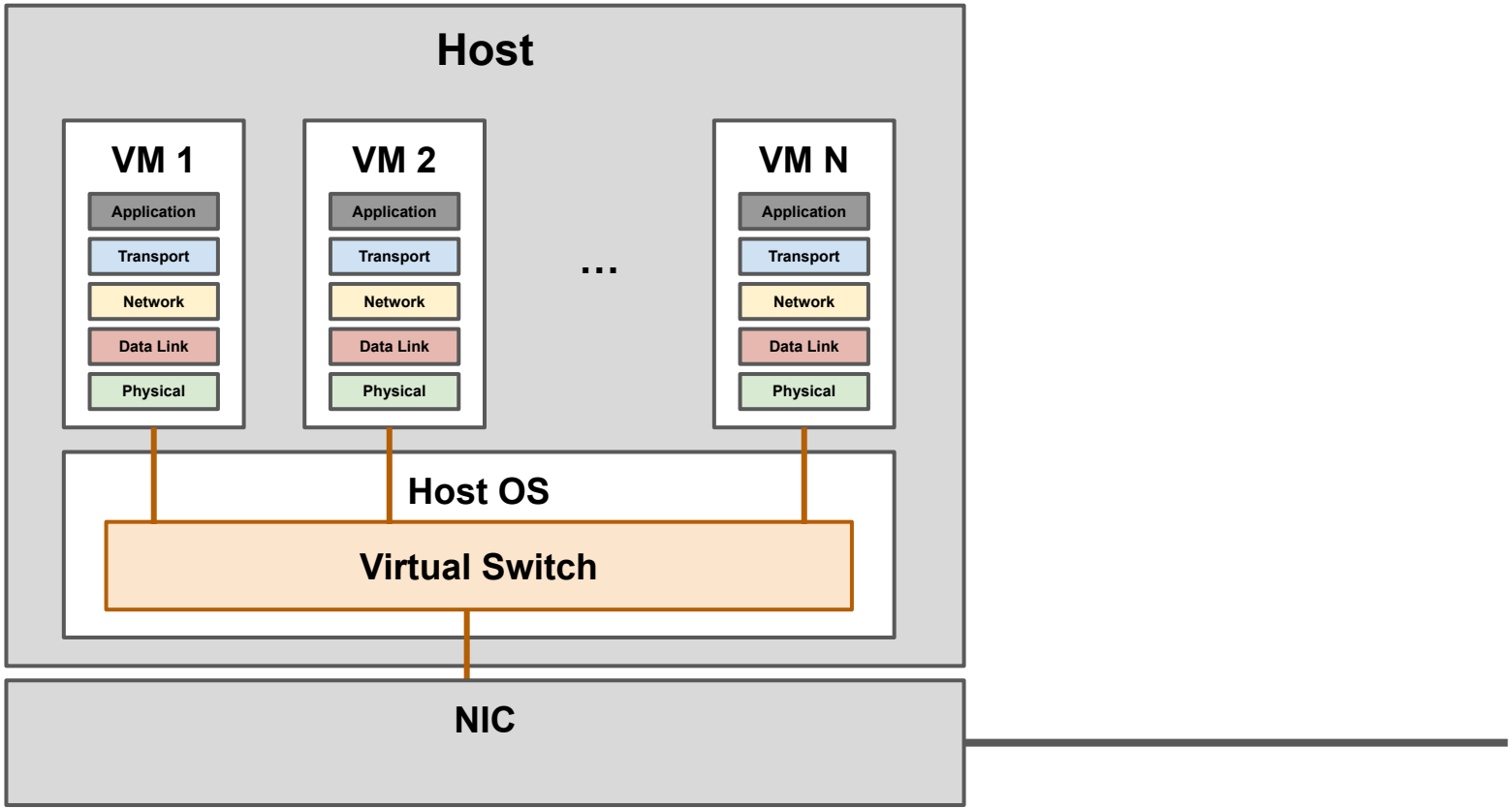


Typically, the provider of the virtualized platform needs to

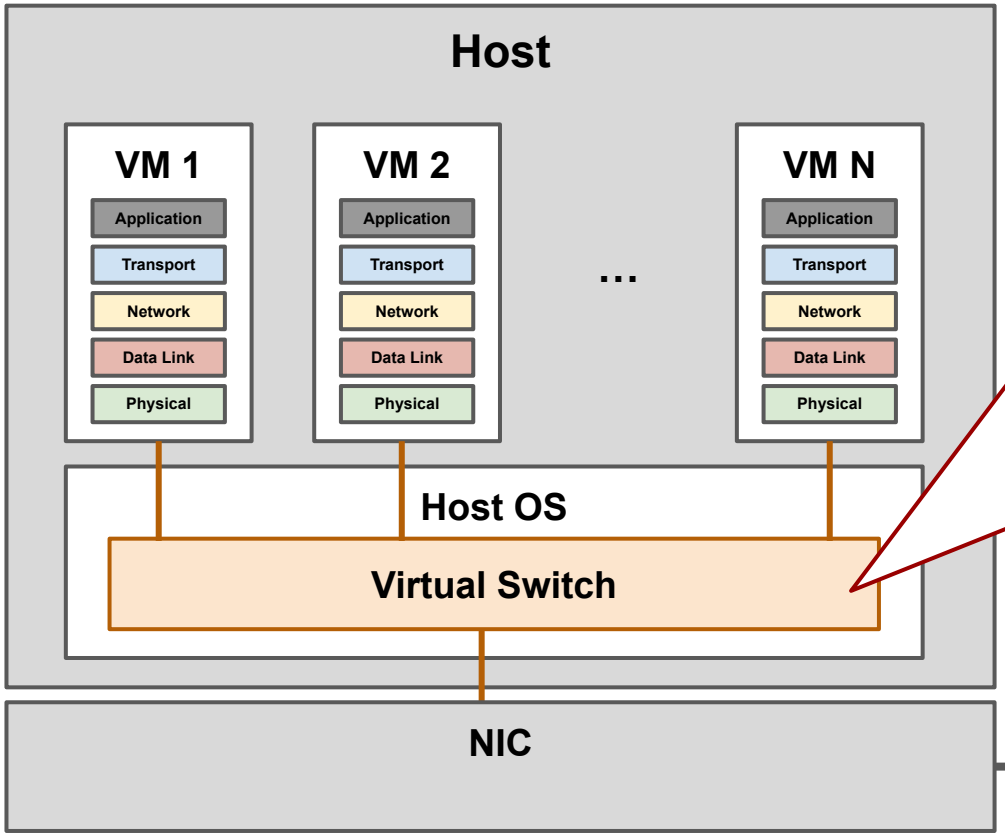
- forward traffic between VMs or VMs and the NIC.
- manage how to share network resource between VMs.

They have to do it somewhere here

Virtual Switch (vSwitch)



Virtual Switch (vSwitch)

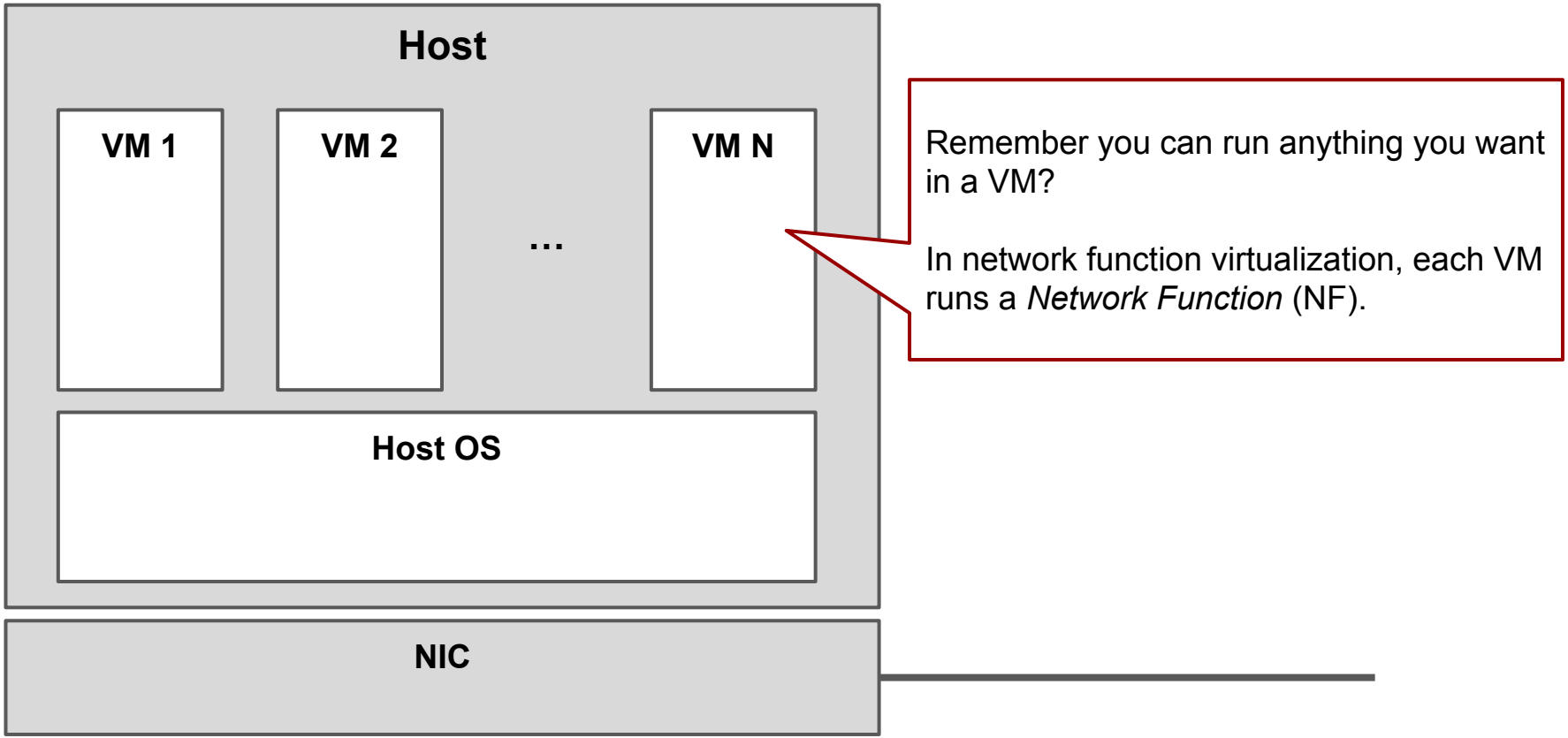


It is a switch! But it can (and needs to) do much more than a switch in the middle of the network (e.g., connection tracking)

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?

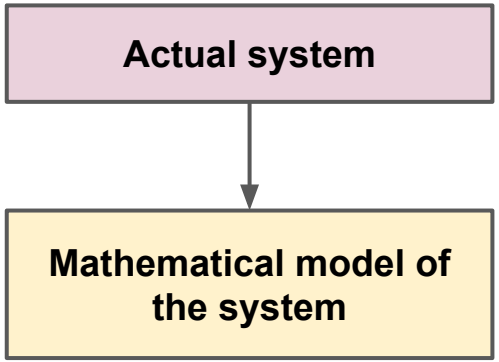
Applications to Network Verification

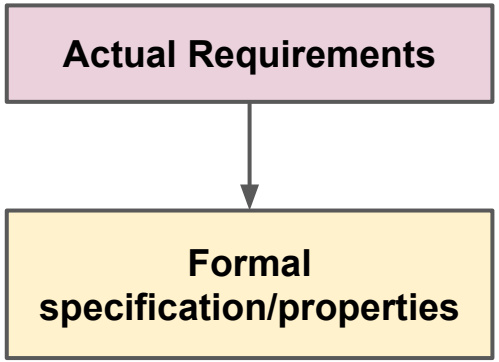
How do we go about verifying a system?

Actual system

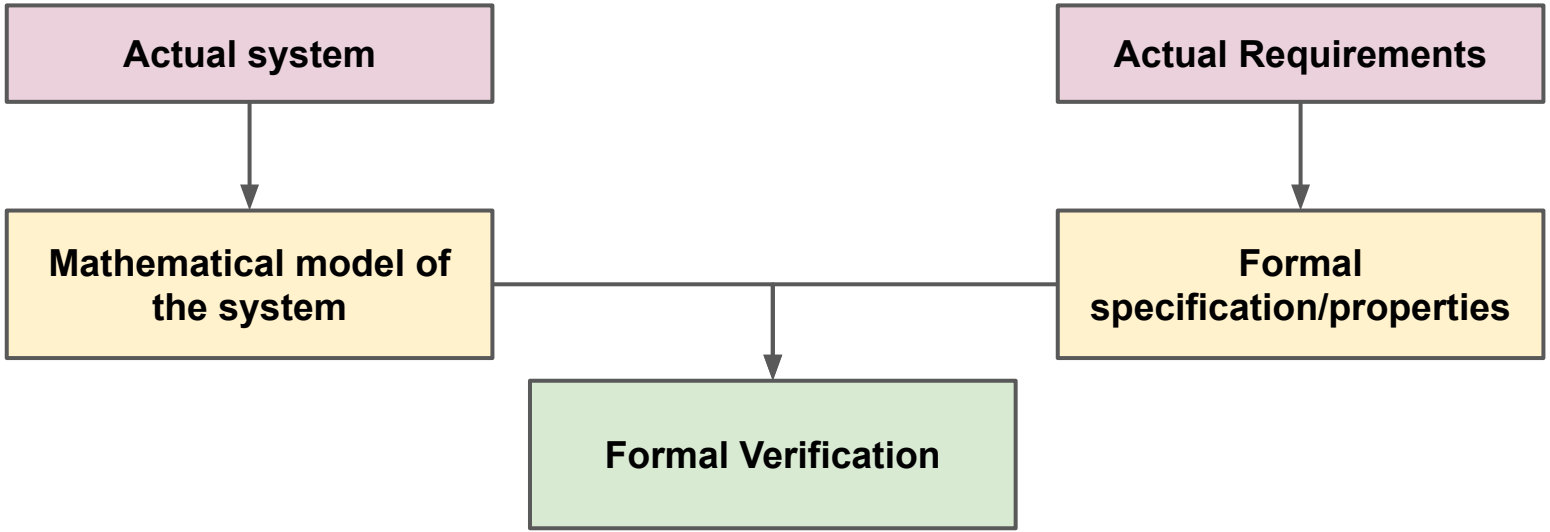
Actual Requirements

How do we go about verifying a system?

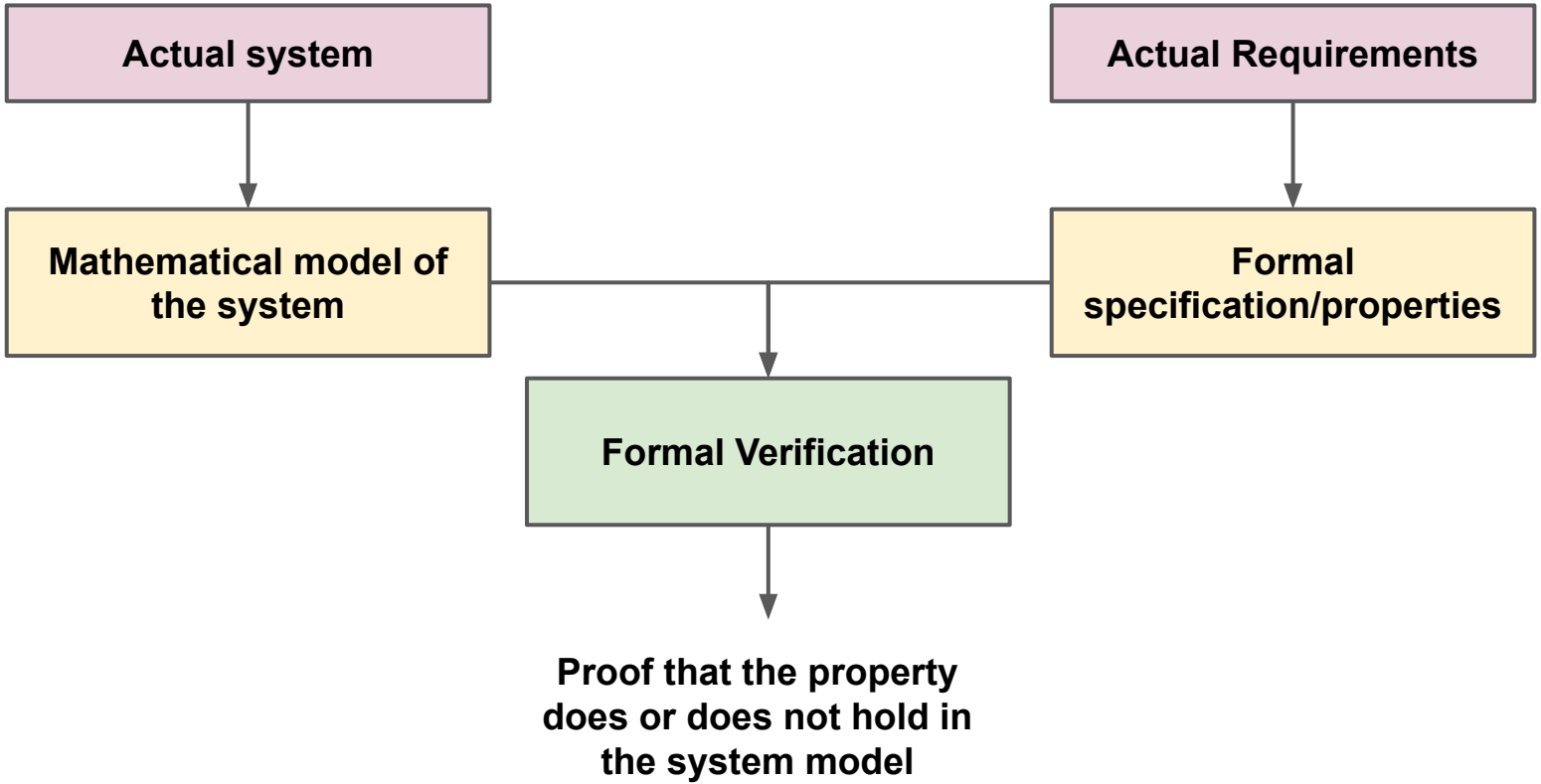




How do we go about verifying a system?



How do we go about verifying a system?



Why use formal verification in networking?

- Networks are growing increasingly complex.
  - They can have hundreds or thousands of interacting components
  - The functionality running in each component is getting more complex
  - configurations files can grow as large as thousands of lines
- Networks are becoming a critical infrastructure
  - Bugs can take down the network or reduce its performance.
  - Network problems can affect thousands if not millions of people
- We need to catch bugs (or prove lack thereof) proactively before going into production

Formal verification in networking

- Started with verifying the forwarding properties of the data plane and control plane.
- Now expanding into more complex functionalities and properties
  - DNS, network performance, …

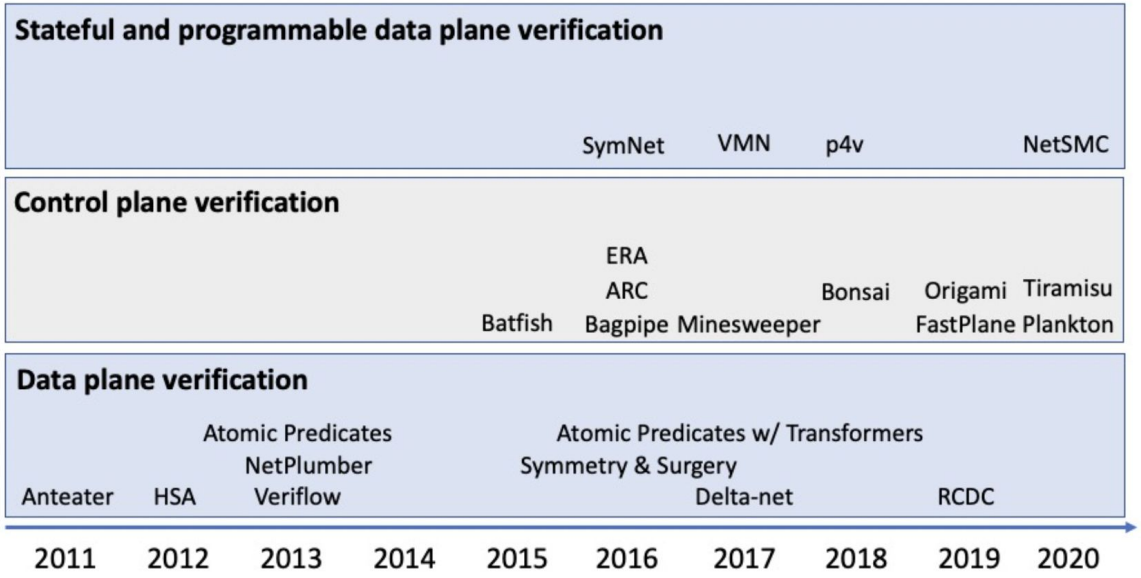


Figure taken from netverify.fun

Formal verification in networking

- Started with verifying the forwarding properties of the data plane and control plane.
- Now expanding into more complex functionalities and properties
  - DNS, network performance, …

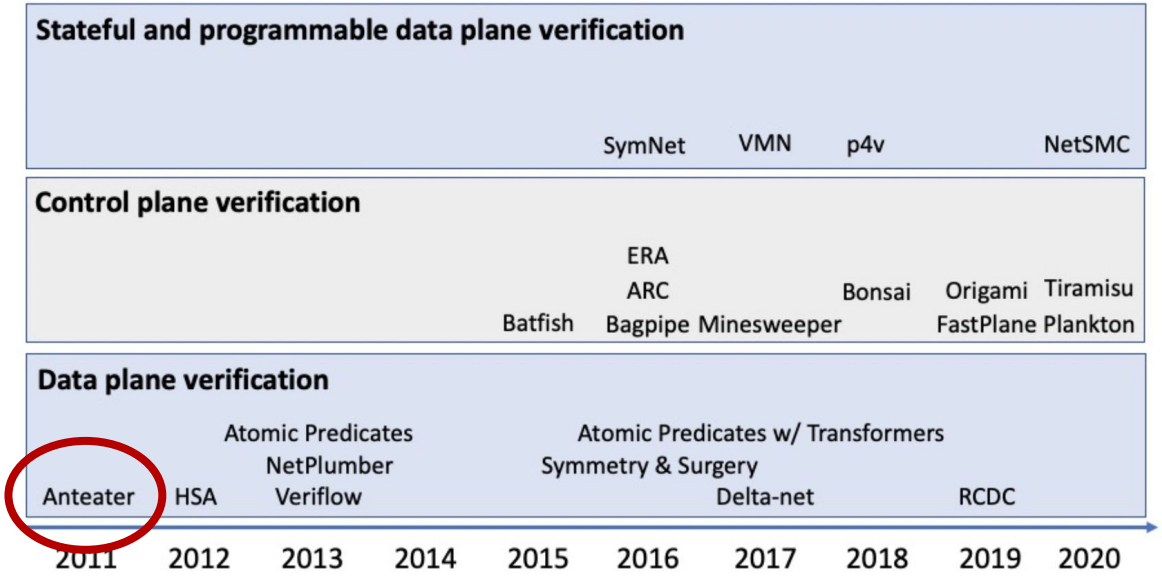


Figure taken from netverify.fun

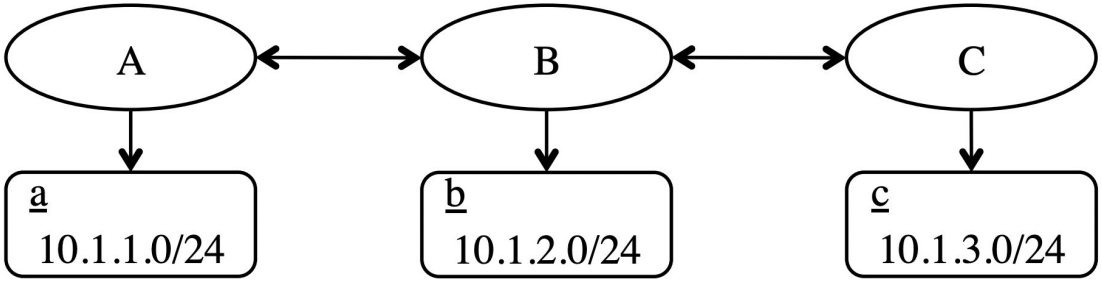
Example - Anteater (SIGCOMM'11)

A:
10.1.1.0/24 -> DIRECT
10.1.2.0/24 -> B
10.1.3.0/24 -> B

B:
10.1.1.0/24 -> A
10.1.2.0/24 -> DIRECT
10.1.3.0/24 -> C

C:
10.1.1.0/24 -> B
10.1.2.0/24 -> B
10.1.3.0/24 -> DIRECT

B->C:
10.1.3.128/25 -> DROP



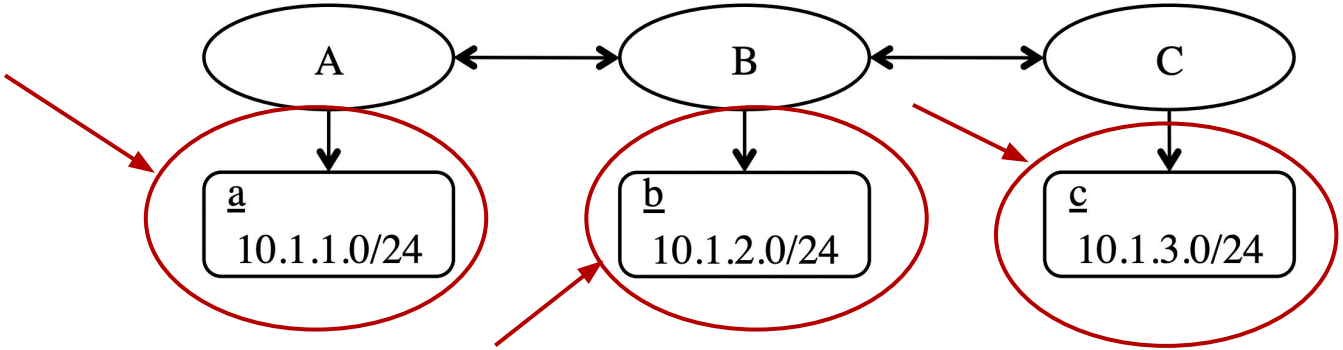
Example - Anteater (SIGCOMM'11)

A:
10.1.1.0/24 -> DIRECT
10.1.2.0/24 -> B
10.1.3.0/24 -> B

B:
10.1.1.0/24 -> A
10.1.2.0/24 -> DIRECT
10.1.3.0/24 -> C

C:
10.1.1.0/24 -> B
10.1.2.0/24 -> B
10.1.3.0/24 -> DIRECT

B->C:
10.1.3.128/25 -> DROP



Example - Anteater (SIGCOMM'11)

A:
10.1.1.0/24 -> DIRECT
10.1.2.0/24 -> B
10.1.3.0/24 -> B

B:
10.1.1.0/24 -> A
10.1.2.0/24 -> DIRECT
10.1.3.0/24 -> C

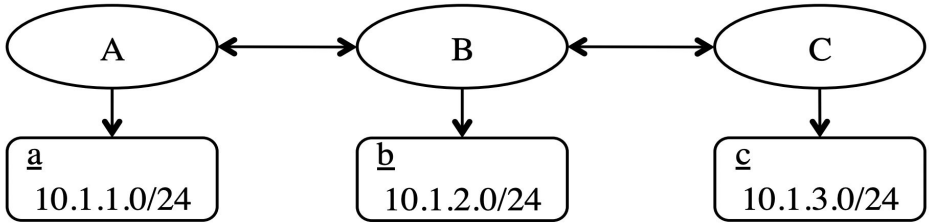
C:
10.1.1.0/24 -> B
10.1.2.0/24 -> B
10.1.3.0/24 -> DIRECT

B->C:
10.1.3.128/25 -> DROP

Model each bit in the packet as a boolean variable.

- The rules only use destination IP, so we only model the 32 bits in the destination IP address.

`P(x, y)`: boolean formula describing which packets can go from **`x`** to **`y`**.



Example - Anteater (SIGCOMM'11)

A:
10.1.1.0/24 -> DIRECT
10.1.2.0/24 -> B
10.1.3.0/24 -> B

B:
10.1.1.0/24 -> A
10.1.2.0/24 -> DIRECT
10.1.3.0/24 -> C

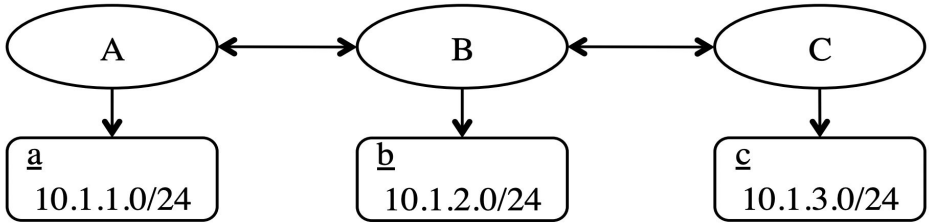
C:
10.1.1.0/24 -> B
10.1.2.0/24 -> B
10.1.3.0/24 -> DIRECT

B->C:
10.1.3.128/25 -> DROP

P(x, y): boolean formula describing which packets can go from **x** to **y**.

$$P(A, a) = \text{dst ip} =_{24} 10.1.1.0$$

$$P(A, B) = \text{dst ip} =_{24} 10.1.2.0 \vee \text{dst ip} =_{24} 10.1.3.0$$



Example - Anteater (SIGCOMM'11)

A:
10.1.1.0/24 -> DIRECT
10.1.2.0/24 -> B
10.1.3.0/24 -> B

B:
10.1.1.0/24 -> A
10.1.2.0/24 -> DIRECT
10.1.3.0/24 -> C

C:
10.1.1.0/24 -> B
10.1.2.0/24 -> B
10.1.3.0/24 -> DIRECT

B->C:
10.1.3.128/25 -> DROP

P(x, y): boolean formula describing which packets can go from **x** to **y**.

$P(A, a) = \text{dst ip} =_{24} 10.1.1.0$

$P(A, B) = \text{dst ip} =_{24} 10.1.2.0 \vee \text{dst ip} =_{24} 10.1.3.0$

$\text{dst ip} =_{w} \text{prefix}$

is a shorthand for

$\bigwedge_{32-w \leq i \leq 32} (\text{dst ip}[i] = \text{prefix}[i])$

Example - Anteater (SIGCOMM'11)

A:
10.1.1.0/24 -> DIRECT
10.1.2.0/24 -> B
10.1.3.0/24 -> B

B:
10.1.1.0/24 -> A
10.1.2.0/24 -> DIRECT
10.1.3.0/24 -> C

C:
10.1.1.0/24 -> B
10.1.2.0/24 -> B
10.1.3.0/24 -> DIRECT

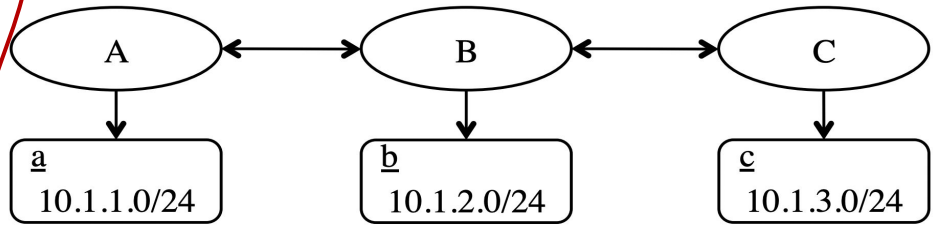
B->C:
10.1.3.128/25 -> DROP

P(x, y): boolean formula describing which packets can go from **x** to **y**.

$P(B,A) = \text{dst ip} =_{24} 10.1.1.0$

$P(B, b) = \text{dst ip} =_{24} 10.1.2.0$

$P(B, C) = \text{dst ip} =_{24} 10.1.3.0 \wedge \text{dst ip} \neq_{25} 10.1.3.128$



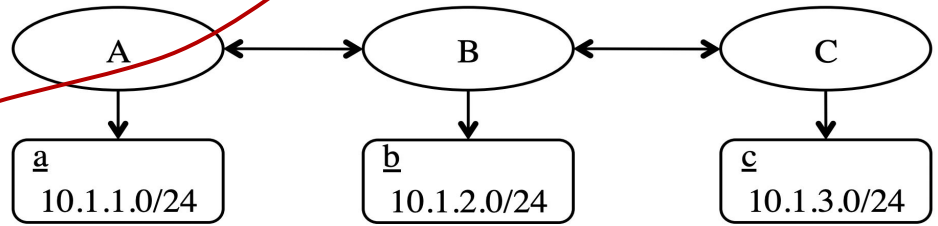
Example - Anteater (SIGCOMM'11)

A:
10.1.1.0/24 -> DIRECT
10.1.2.0/24 -> B
10.1.3.0/24 -> B

B:
10.1.1.0/24 -> A
10.1.2.0/24 -> DIRECT
10.1.3.0/24 -> C

C:
10.1.1.0/24 -> B
10.1.2.0/24 -> B
10.1.3.0/24 -> DIRECT

B->C:
10.1.3.128/25 -> DROP



P(x, y): boolean formula describing which packets can go from **x** to **y**.

$$P(C, B) = dst\ ip =_{24} 10.1.1.0 \vee dst\ ip =_{24} 10.1.2.0$$

$$P(C, c) = dst\ ip =_{24} 10.1.3.0$$

Example - Anteater (SIGCOMM'11)

- Can A reach C?
- Anteater uses a simple graph algorithm to construct the boolean formula that describe all the packets that can reach C from A using P(x, y)
- That formula is **`P(A, B)`**$\wedge$ **`P(B, C)`**
- The formula is given to a SAT solver to check if any assignment to the boolean variables, i.e., any destination IP address, exists that can go from A to C
- If no, no packets can reach C from A

Reasoning about network forwarding behavior

- Since Anteater, there has been several other proposals for other ways for both modeling and analysis
- Header Space Analysis (HSA) (NSDI'12)
  - models sets of K-bit packets as subspaces in a K-dimensional space
  - uses set operations for analysis
- Veriflow (NSDI'13)
  - uses a trie to find equivalence classes (ECs) of packets
  - models the forwarding behavior of ECs using a forwarding graph
  - analyzes the network behavior using graph algorithms
- There has been a lot more! (see netverify.fun for a survey)

Formal methods in networking

- Data-plane verification
  - Model and analyze the forwarding rules on the data plane
  - Anteater, HSA, Veriflow, …
- Control-plane verification
  - Model and analyze the control-plane protocols that configure the data plane
- Stateful and programmable data planes

Formal methods in networking

- Analyzing DNS
  - Is there a query under our domain that is sent for resolution to a name server, not under our domain?
- Analyzing performance
  - Is there an input traffic pattern under which the network provides high latency?

Formal methods in networking industry

- Large cloud providers are integrating formal methods into their network operations
  - Microsoft, Amazon, Google, Alibaba, …
  - "Be sure before shipping – the need for safety in clouds" - Dave Maltz keynote in the netverify'21 workshop organized by Microsoft and Google
- Several startup companies
  - Forward Networks, Veriflow, Intentionet, …

How does this all relate to programmable networks?

- Automated testing and verification did not start with and is not limited to programmable networks.
- But, programming abstractions for a single device or collection of devices provides extra opportunities.
  - We can reuse so much of the existing knowledge, expertise, and tools for program verification in the formal methods and PL community
  - In our "network" programs, we already have accurate well-defined specifications of network functionality.
  - We can verify the compilers (or their output) to provide end-to-end verified tool chains
  - …

Applications to Network Monitoring

Why is network monitoring challenging?

We have to observe network traffic and analyze it in real-time.

Why is network monitoring challenging?

We have to observe network traffic and analyze it in real-time.

- Terabits of traffic per second on a single switch
- 100s or 1000s of switches in a network

Why is network monitoring challenging?

- Many different statistics and properties to monitor
- What you need to monitor can change over time.

We have to observe network traffic and analyze it in real-time.

- Terabits of traffic per second on a single switch
- 100s or 1000s of switches in a network

Why is network monitoring challenging?

- Many different statistics and properties to monitor
- What you need to monitor can change over time.

We have to observe network traffic and analyze it in real-time.

- Terabits of traffic per second on a single switch
- 100s or 1000s of switches in a network

Have to observe and analyze traffic quite fast

- Data plane: fast but has limited resources.
- Control plane: more resources but slower.

Monitoring in "traditional" networks

- It is up to the vendors what kind of monitoring data they collect on the switches and how it can be reported to a monitoring server.
  - Typically limited to coarse-grained information every few seconds.
  - e.g., NetFlow
- Sounds familiar? :)

Network programmability → Flexible and fine-grained monitoring

- Program the data plane to gather the data that you want
- Program the data plane (and the run-time) to have the data pushed to/pulled form a central monitor when you want.
- Create top-down programmable monitoring frameworks:
  - Users specify the information they are interested as queries
  - The compiler and runtime figure out how to configure each device to collect and report information according to the query.

Network programmability → Flexible and fine-grained monitoring

- Program the data plane to gather the data that you want
- Program t… ed to/pulled …
- Create to…
  - Users…
  - The compiler and runtime figure out how to configure each device to collect and report information according to the query.

Monitoring is one of the "killer" apps for programmable data planes

Network programmability → Flexible and fine-grained monitoring

- Program the data plane to gather the data that you want
- Program the data plane (and the run-time) to have the data pushed to/pulled form a central monitor when you want.
- Create top-down programmable monitoring frameworks:
  - Users specify the information they are interested as queries
  - The compiler and runtime figure out how to configure each device to collect and report information according to the query.

Network programmability → Flexible and fine-grained monitoring

- Program the data plane to gather the data that you want
- Program the data plane (and the run-time) to control the data pushed to/pulled form a central monitor when
- Create top-down programmable moni
  - Users specify the information they are interested as queries
  - The compiler and runtime figure out how to configure each device to collect and report information according to the query.

- Sketches
- In-Band Network Telemetry (INT)
- …

Sketches

- Modern high-speed switches can observe terabits of traffic every second.
- but have limited computational resources
  - Specially memory, which is essential for monitoring purposes, e.g., to keep track of statistics
- Typically, if we have N flows going through a switch, we don't have O(N) memory in the switch to keep information about them.
- So, what do we do?

Sketches

- Sketches are approximate data structures that
  - keep information about a large amount of data in a substantially smaller amount of space
  - and can answer certain queries about it in an approximate way.
- They typically provide a trade-off between resource usage and accuracy.
- If you give them more space, they'll provide a more accurate result.
- Extensive line of research on designing sketches that can run on programmable switches
  - NICs?

Connections to streaming algorithms

- There are lots of synergies between network monitoring and streaming algorithms.
- Algorithms in the streaming setting have more constraint than "regular" algorithms
  - They see the input as a sequence of items examined in a few passes, typically one → packets passing through the switch
  - Operate within limited memory (sublinear) and sometimes limited processing per item → computational constraints of programmable switches

In-Band Network Telemetry (INT)

- In programmable data planes, you can define custom headers and process them however you want.
- INT proposes to add a "telemetry" header and have switches populate it with information that will help network monitoring
  - How long did the packet spend in the switch? How much of it was waiting in a queue?
  - switch id, to help figure out what paths packets take in the network
  - …

In-Band Network Telemetry (INT)

- Once the packet gets to its destination, the information in the INT header can be analyzed there and/or sent to a central monitor.
- Having fine-grained information about what happened to the packet as it traverses a network is extremely useful.
- Specially for detecting and debugging transient and subtle problems.
- There is no free lunch!
    - If every switch adds a large INT header to the packet, that can create throughput overheads.

Network programmability → Flexible and fine-grained monitoring

- Program the data plane to gather the data that you want
- Program the data plane (and the run-time) to have the data pushed to/pulled form a central monitor when you want.
- Create top-down programmable monitoring frameworks:
  - Users specify the information they are interested as queries
  - The compiler and runtime figure out how to configure each device to collect and report information according to the query.

Network programmability → Flexible and fine-grained monitoring

- Program the data plane to gather the data that you want
- Program the data plane (and the run-time) to have the data pushed to/pulled form a central monitor when you want.
- Create top-down programmable monitoring frameworks:
  - Users specify the information they are interested as queries
  - The compiler and runtime figure out how to configure each device to collect and report information according to the query.

Connections to network verification

- It is likely that we can't model *everything* in the network in detail and analyze it proactively.
- To ensure our networks satisfy our desired properties, we need
  - scalable proactive analysis to catch as many violating scenarios as possible before production
  - flexible and fine-grained run-time monitoring to continuously watch for property violations at runtime.

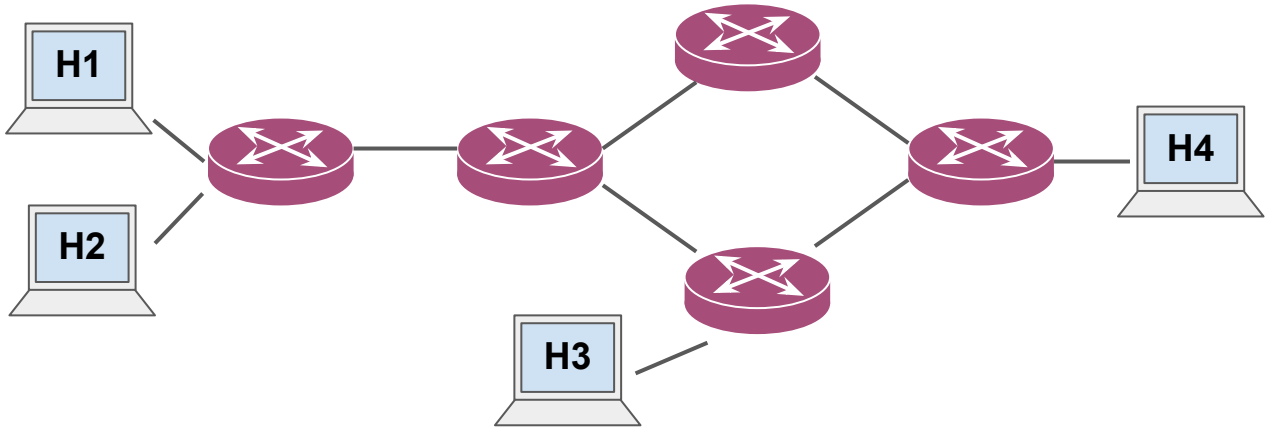
Applications to Transport and Network Quality of Servcie (QoS)

Networks are shared infrastructure

IP only provides *best-effort packet delivery*

There are other mechanisms to control/customize how different flows share network resources.

- end-to-end congestion control
- packet scheduling
- active queue management
- …



Traditional networks mostly rely on end-to-end congestion control

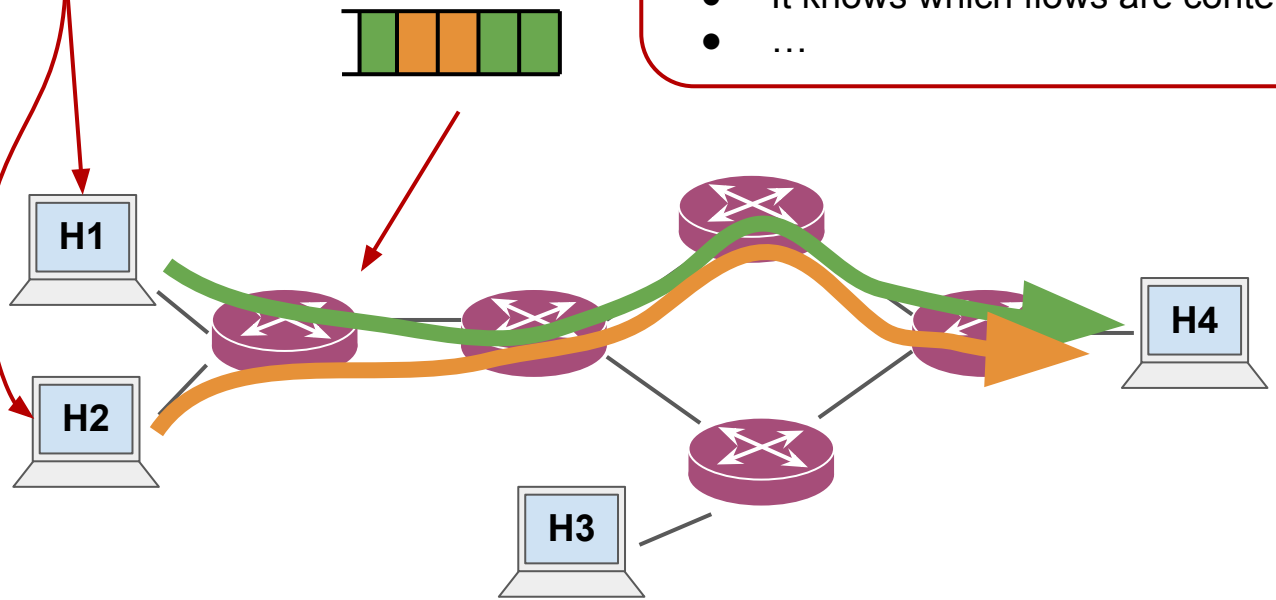
- keeping the functionality in the network quite simple
  - No explicit signals or a simple fixed set of signals for end-to-end congestion control algorithms
  - A few FIFO queues and a few schedulers (e.g., priorities, DRR, etc.)
  - No AQM or a simple fixed set of AQM algorithms
- Why?
  - end-to-end principle
  - Keeping network devices simple and fast

But a little help from the network can go a long way

These hosts don't know their traffic is going to collide.

The switch knows a lot more about the contention

- It is where the contention is happening
- It sees the queue building up
- It knows which flows are contending
- …



But a little help from the network can go a long way

- In traditional networks, the sender has to infer what is happening at the switch from indirect signals (delays, loss, marked packets).
- Why not have the the switch provide the information more directly and explicitly to senders?
- Why not have the switch play a more active role in handling contention with more sophisticated scheduling and AQM algorithms?

How has network programmability helped?

- Customizing the signals to e2e congestion control, scheduling, AQM, etc. to the each network and the requirements of its applications.
- Motivating new signaling, scheduling, AQM, etc. techniques
  - Implement it in a programmable switch
  - show it can run at line rate
  - show it provides significant benefits
  - so you can convince vendors to include it in their switches

How has network programmability helped?

- Better signals for congestion control algorithms
  - e.g., use INT to add information about queue lengths to the packets (HPCC, SIGCOMM 2019)
- More complex (and flexible) packet scheduling
  - e.g., fair queuing is hard to implement at line-rate but you can implement and approximation on programmable switches (AFQ, NSDI'18).
  - a programmable hardware architecture for packet scheduling, so we can configure the switch for different scheduling algorithms (PIFO, SIGCOMM'16)

How has network programmability helped?

- Targeted fine-grained measurements
  - can help provide better signals to congestion control algorithms
  - can help create more effective AQM schemes
  - e.g., if we could detect which flow(s) contribute most to the queue build up, we can mark/drop those packets in our AQM scheme (Conquest, CoNEXT'19)

In-Network Computing

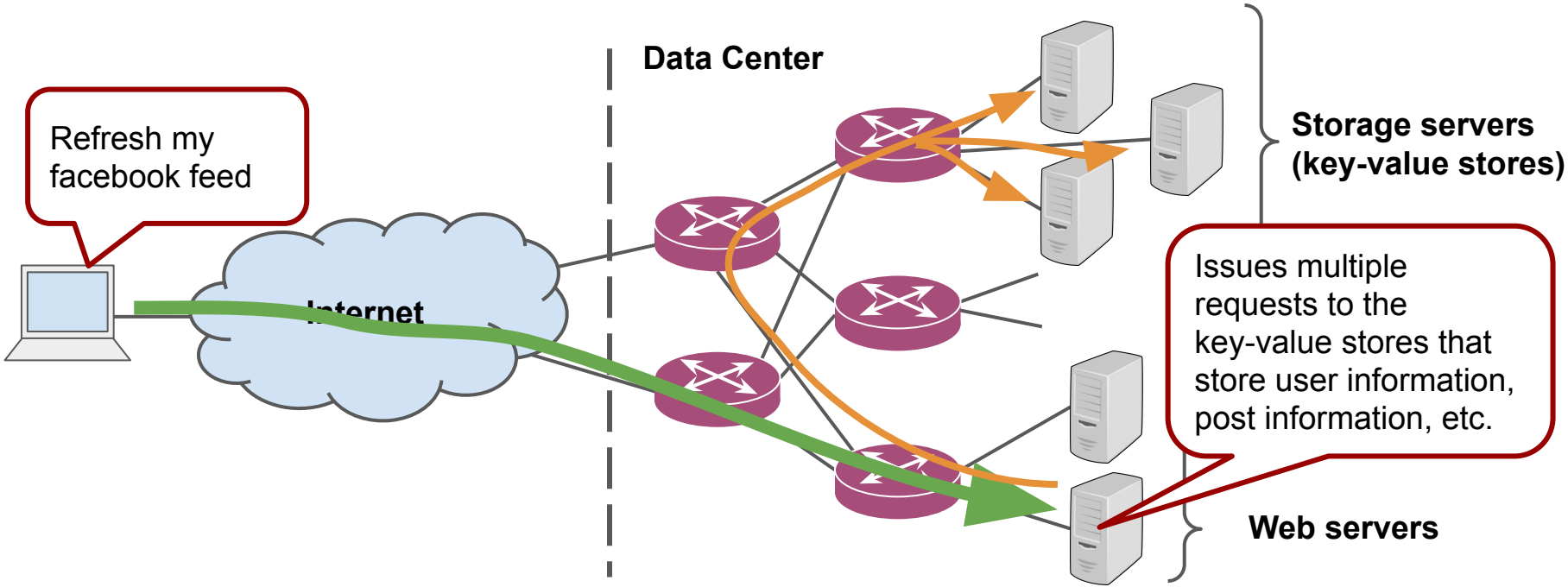
In-Network Computing

Offloading part of the application processing (i.e., compute) to the network

Example 1: In-network caching

- Key-value stores can get millions if not billions of requests every second.
- To handle such load, there are usually several storage servers, each taking care of part of the key-value store.
- Requests are load-balanced across storage servers.
- Problem?
    - Hot items change all the time
    - This can create load imbalance.
    - That is, one server (or a subset of them) can get overwhelmed and not be able to answer queries fast enough for good user quality of experience.

Example 1: In-network caching

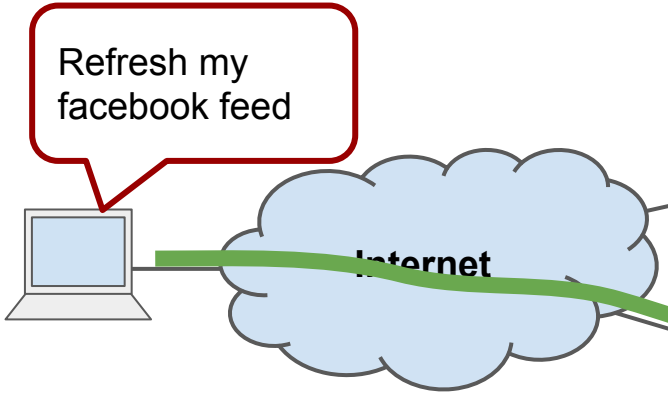


Example 1: In-network caching

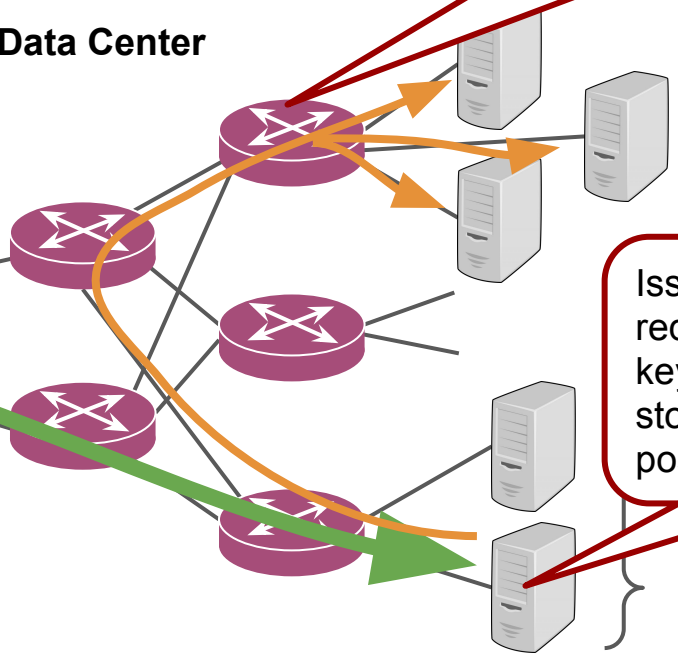
All the requests are going through the top of rack switch!

Can we store (i.e., cache) some of the "hot items" there?

Refresh my facebook feed



Data Center



Storage servers (key-value stores)

Issues multiple requests to the key-value stores that store user information, post information, etc.

Web servers

Example 1: In-network caching

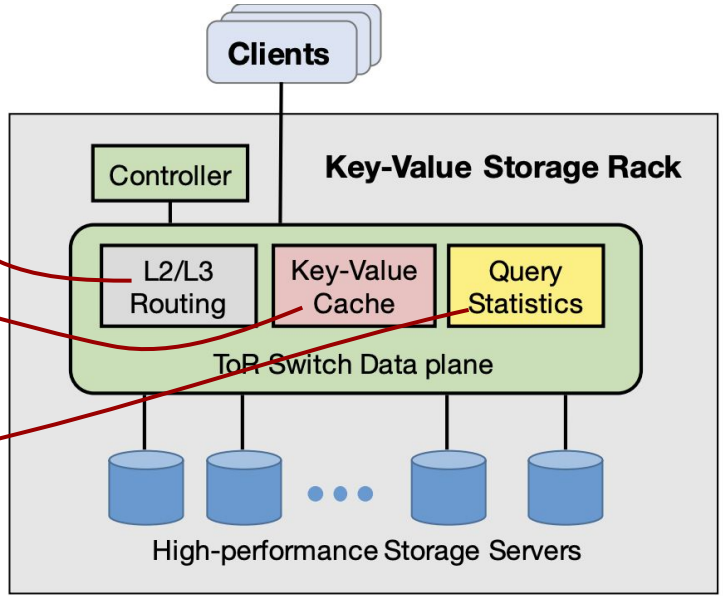
- NetCache (SOSP'17) proposes to do just that!

Regular switch functionality

Maintains "hot" items

Gather statistics about the queries.

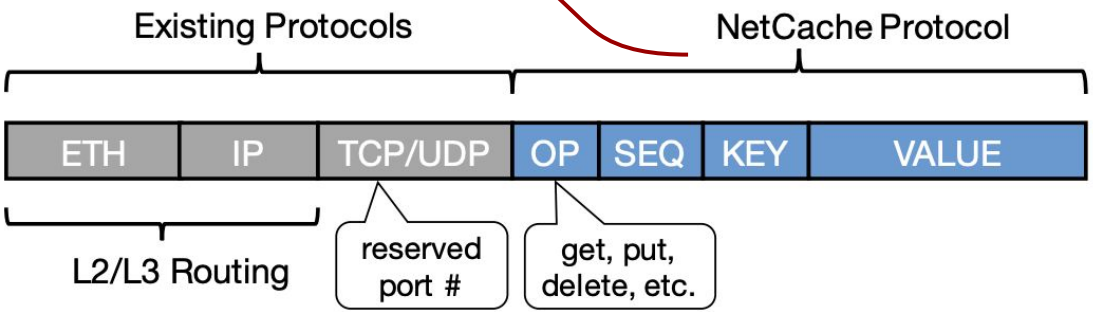
so the controller can update the cache as query patterns change.



Example 1: In-network caching

- NetCache (SOSP'17) proposes to do just that!

with a programmable parser, NetCache can define its own header.

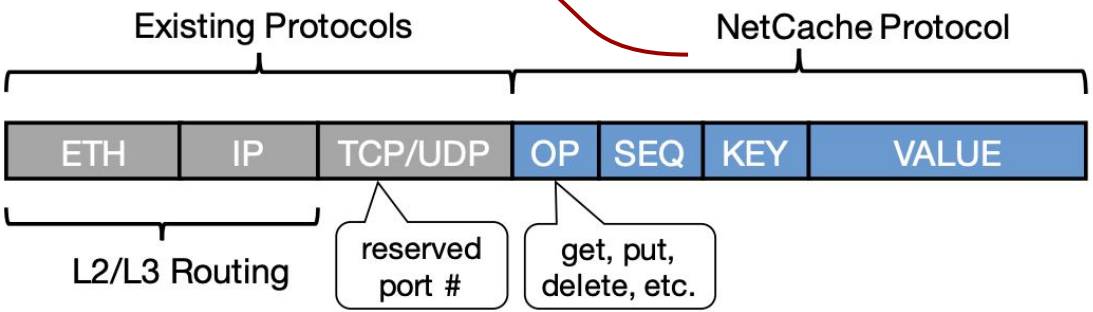


Example 1: In-network caching

- NetCache (SOSP'17) proposes to do just that!

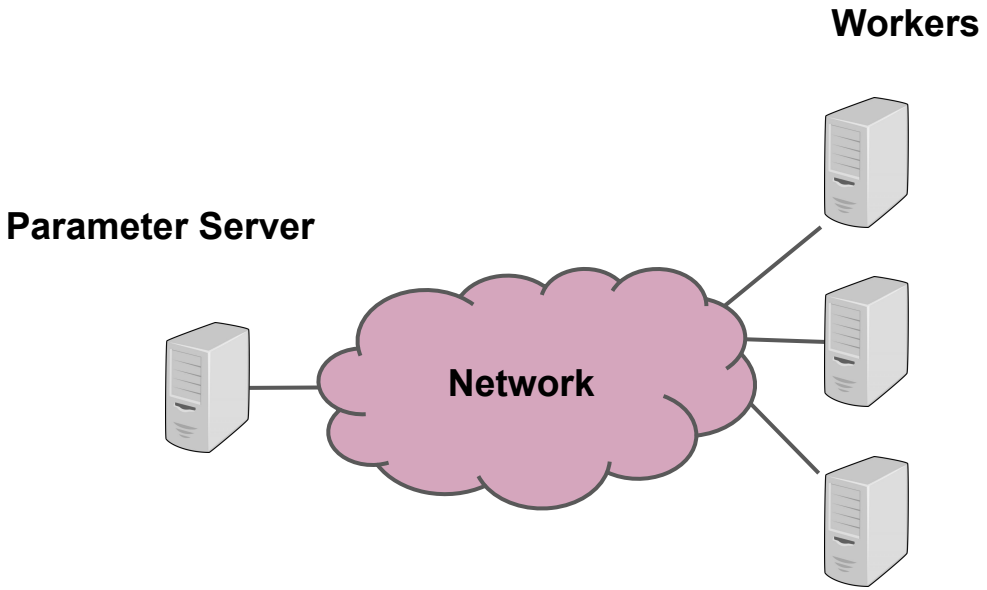
with a programmable parser, NetCache can define its own header.

Applications are provided with a library that translates their requests to packets with NetCache headers.



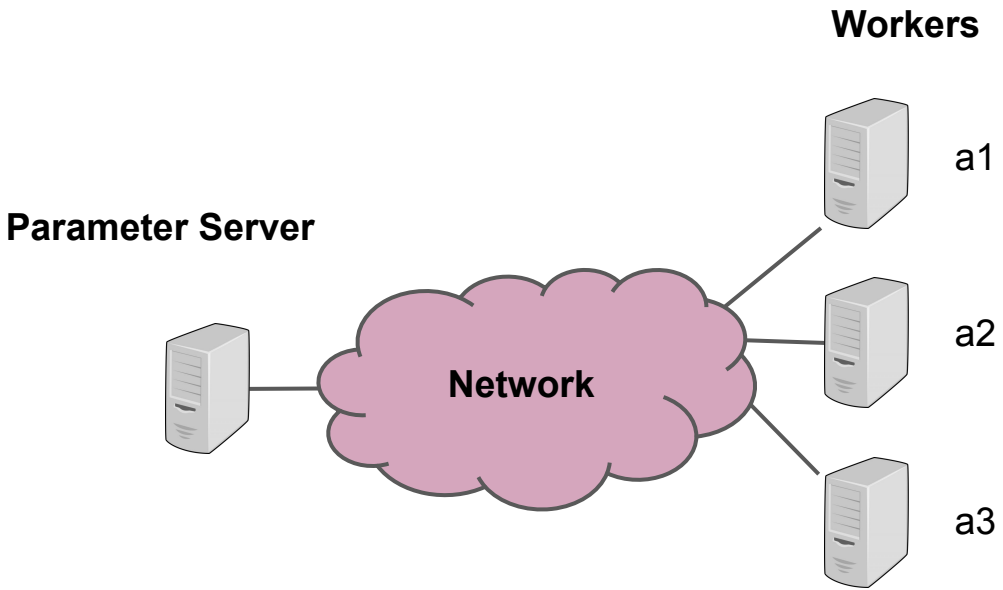
Example 2: Accelerating ML Training

- Distributed training of ML models can require a lot of network communication.



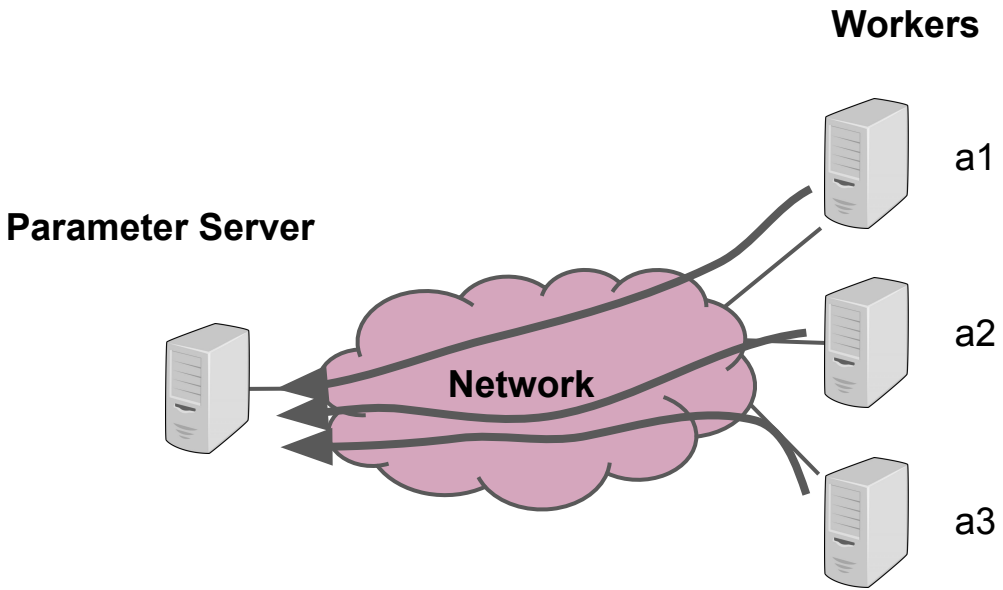
Example 2: Accelerating ML Training

- Distributed training of ML models can require a lot of network communication.



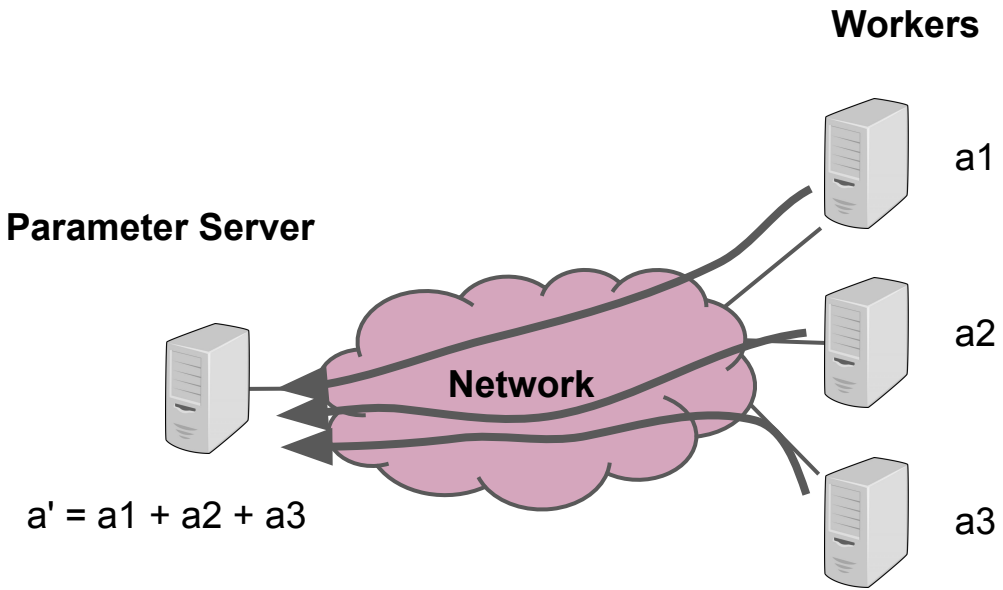
Example 2: Accelerating ML Training

- Distributed training of ML models can require a lot of network communication.



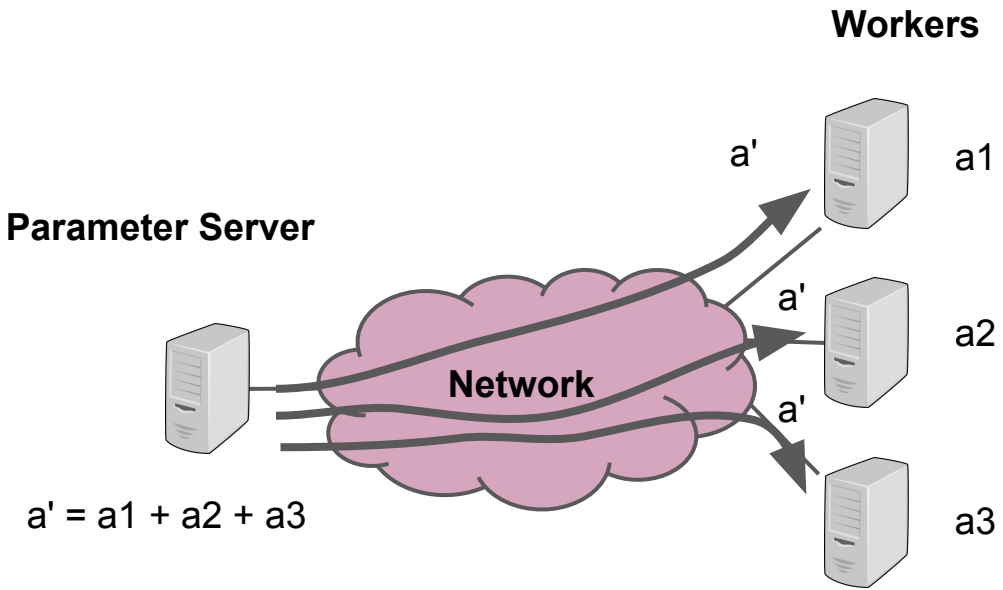
Example 2: Accelerating ML Training

- Distributed training of ML models can require a lot of network communication.



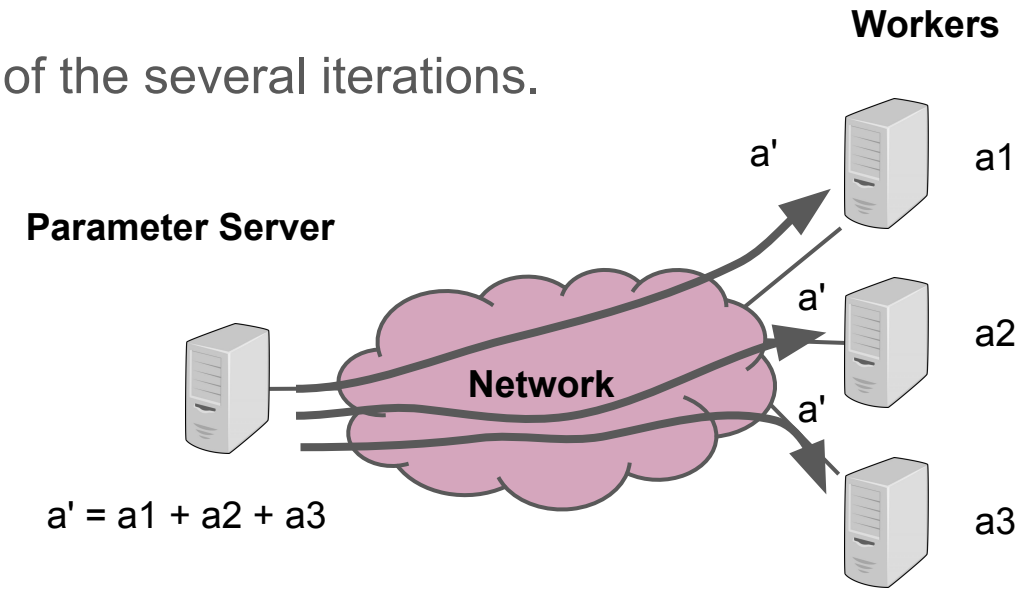
Example 2: Accelerating ML Training

- Distributed training of ML models can require a lot of network communication.

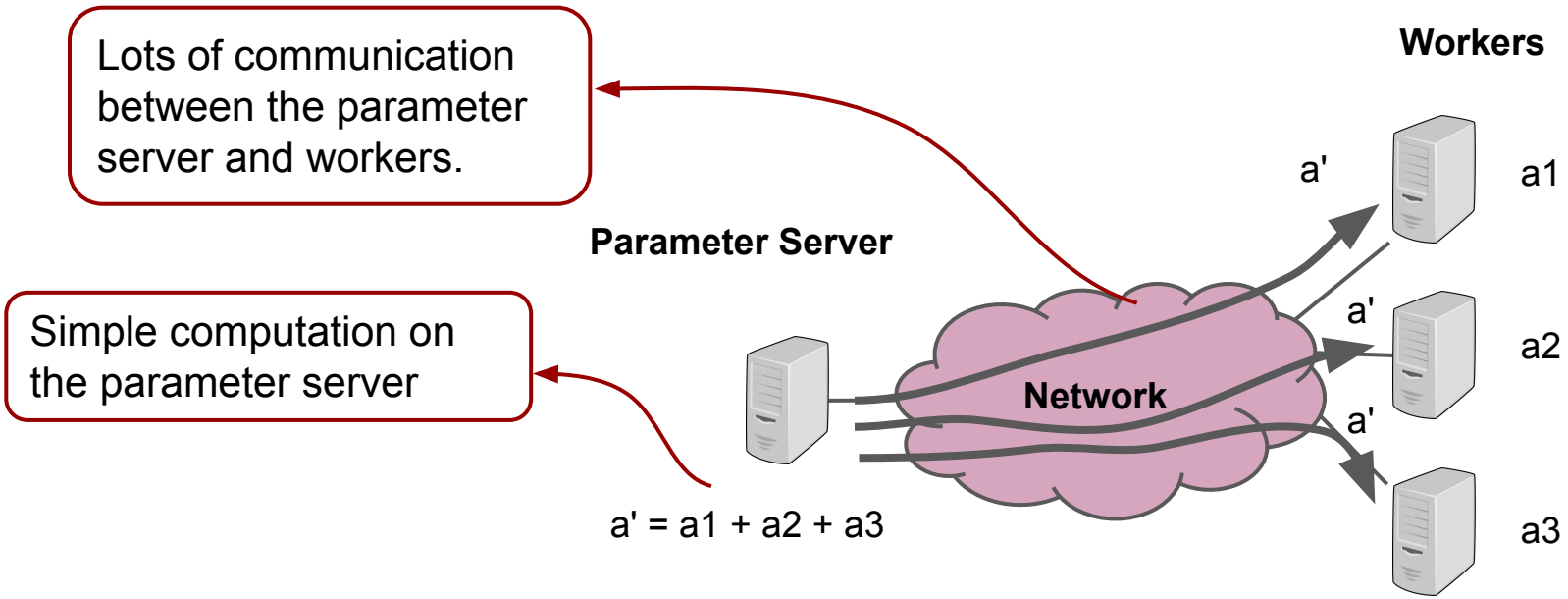


Example 2: Accelerating ML Training

- Distributed training of ML models can require a lot of network communication.
- This happens in every of the several iterations.



Example 2: Accelerating ML Training



Example 2: Accelerating ML Training

Implement the parameter server in network switches

- The switch can keep track of the sum (aggregate) in a register.
- As packets come from the workers, it can retrieve values from packets and update the sum.
- Once the switch receives values from all workers, it can send the sum back to the workers.
- Benefits? Same as before
    - Higher throughput and lower communication latency

Challenges of in-network computing

- What if the information we need from the applications spans multiple packets?
  - e.g., in Netcache, what if the value for a key-value pair doesn't fit into one packet?
- It is difficult to reconstruct a stream in the switch
  - reconstruct = put together packet contents from multiple packets

Challenges of in-network computing

- Application logic is typically stateful.
- Switches have limited memory, and only allow limited access to it
- Application logic can be more complex than network processing
- Switches have limited computational capabilities.

Challenges of in-network computing

- You can see these constraints play out in current applications of in-network computing
  - NetCache caches hot items with small-ish values.
  - ML training parameter aggregation doesn't need lots of data to be stored
  - In all cases, computation is quite simple.
- There have been proposals for switches with computational resources and capabilities that are more suited for application acceleration
  - e.g., Trio, or Tofino + FPGA

Challenges of in-network computing

- What should the API be for the applications?
- Suppose you are writing a distributed/networked application.
- How should you specify which part should be "offloaded" and executed in the network?

Challenges of in-network computing

- There is a higher abstraction bar here for programming abstractions.
- If someone is implementing a new network protocol, you can assume they have networking knowledge.
- We don't want application developers to have to learn all the details about network processing (packets, headers, protocols, etc.) to be able to accelerate their application.
- There are recent proposals that try to extend familiar programming abstractions like connections and RPCs for this purpose.