CS4410

Operating Systems

Lecture 25: The OS Network Stack

Rachit Agarwal

Context for today's lecture

- A quick overview of the OS network stack
  - Has evolved over decades
  - Many different components
  - Many different protocols
  - Today is just a brief overview (details in 4450)
- What shall we focus on?
  - Interaction with other components that we have studied in 4410

Recap: what do computer networks do?

A computer network delivers data between the end points

- **One and only one task**: Delivering the data
- This delivery is done by:
    - Chopping the data into **packets**
    - Sending individual packets across the network
    - Reconstructing the data at the end points
- **That is all!**

Recap: Data delivery as a fundamental goal

- **Support the logical equivalence of Interprocess Communication (IPC)**
  - Mechanism for "processes on the same host" to exchange messages

- **Computer networks allow "processes on two different hosts" to exchange messages**

- **Clean separation of concerns**
  - Computer networks deliver data
  - Applications running on end hosts decide what to do with the data

- **Keeps networks simple, general and application-agnostic**

End-to-end story

Four fundamental problems!

- **Naming, addressing:** Locating the destination, receiver app
- **Routing:** Finding a path to the destination host
- **Forwarding:** Sending data from the sender app to the receiver app
- **Reliability:** Handling failures, packet drops, etc.

Fundamental problem #1: Naming and Addressing

- **Network Address: where host is located**
  - Requires an address for the destination host

- **Host Name: which host it is**
  - Consider when you access a website
    - URL is **user-level name** (eg, www.cornell.edu)
    - Network needs address (eg, where is www.cornell.edu)?

- Must map names to addresses
  - Just like we use an address book to map human names to addresses

Must be done at the end-host;

The source knows the name—

Maps that name to an address using DNS!

(Done only once, when establishing a connection—low overhead)

Fundamental problem #2

Routing packets through network elements (eg, routers) to destination

- Given destination address (and name), how does each switch/router know where to send the packet so that the packet reaches its destination

- When a packet arrives at a router
  - a **routing table** determines which outgoing link the packet is sent on
  - Computed using **routing protocols**

Mostly done within the network switches/routers;

Has little to do with the OS

Fundamental problem #3

Queueing and Forwarding of packets at <u>switches/routers</u>

- **Queueing:** When a packet arrives, store it in “input queues”
  - When a packet arrives:
    - Look up its destination’s address (how?)
    - Find the link on which the packet will be forwarded (how?)

- **Forwarding:** When the outgoing link free
  - Pick a packet from the corresponding virtual output queue
  - forward the packet!

Done at switches/routers;

Has little to do with the OS

Fundamental problem #3

Queueing and Forwarding of packets at the host

- When a process wants access to the network, it opens a **socket**, which is associated with a **port**
- **Socket:** an OS mechanism that connects processes to the network stack
- **Port:** number that identifies that particular socket
    - used by the OS to direct incoming packets
- There is a sender-side socket/port, and a receiver-side socket/port

Done at the host OS;

Lot of interesting tradeoffs

What must packets carry to enable forwarding?

- **Packets must describe where it should be sent**
  - Requires an address for the destination host
  - A port number (socket identifier) for the destination application

- **Packets must describe where its coming from**
  - For handling failures, etc.
  - Requires an address for the source host
  - A port number (socket identifier) for the source application

- **Packets must carry data**
  - can be bits in a file, image, whatever

Must be done at the host;

Network just delivers the packet

Fundamental problem #4

How do you deliver packets reliable?

- Packets can be dropped along the way
  - Buffers in router can overflow
  - Routers can crash while buffering packets
  - Links can garble packets

- How do you make sure packets arrive safely on an unreliable network?
  - Or, at least, know if they are delivered?
  - Want no false positives, and high change of success

Mostly implemented at the host (end-to-end principle)

Using a protocol called TCP

There is also an unreliable transmission mechanism: UDP

The end-to-end story

- Application opens a **socket** that allows it to connect to the **network stack**
- Maps **name** of the web site to its **address** using **DNS**
- The network stack at the source embeds the address and **port** for both the source and the destination in **packet header**
- Each **router** constructs a **routing table** using a distributed algorithm
- Each router uses destination address in the packet header to look up the **outgoing link** in the routing table
    - And when the link is free, forwards the packet
- When a packet arrives the destination:
    - The network stack at the destination uses the port to forward the packet to the right application

Four fundamental problems!—what does the OS do?

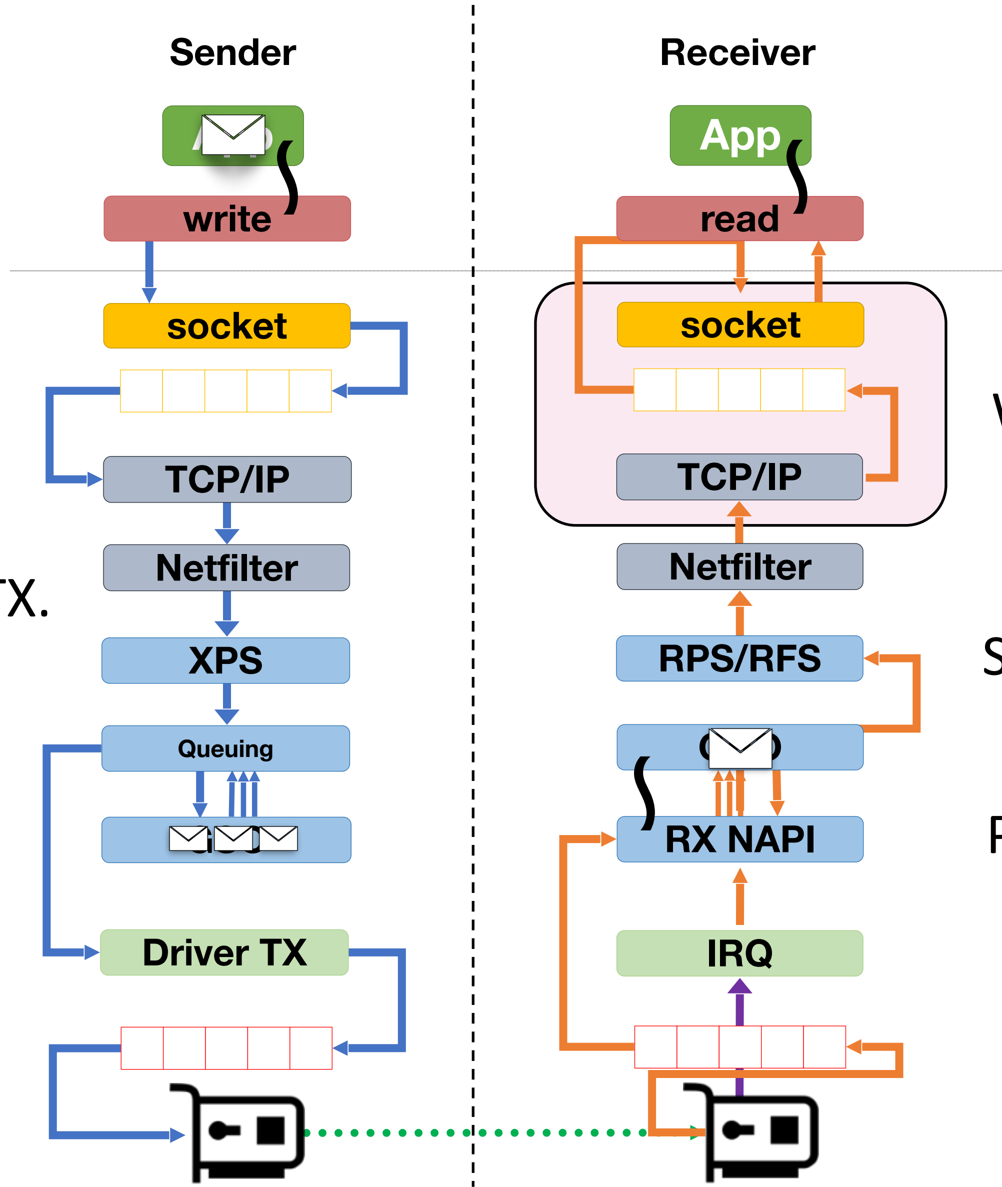
- **Naming, addressing:** Locating the destination
  - Setting up connection (name resolution, etc.)—low overhead
- **Routing:** Finding a path to the destination
  - Little or nothing
- **Forwarding:** Sending data to the destination
  - Create/insert packet headers—high overhead
  - Move data around based on sockets/ports—high overhead
  - Enable applications to read/write data—very high overheads
- **Reliability:** Handling failures, packet drops, etc.
  - Protocol-level processing—high overhead

Questions?

End-host network stack: Questions to ask

- **Where is the sender-side socket located?**
  - Depends on where the application is currently scheduled (CPU scheduler)
- **How does the OS move packets from the sender-side socket to the network hardware?**
  - Data is in some memory location specified by the socket (virtual memory)
  - The OS performs sender-side processing (create packets, headers, TCP processing, ..)
  - Data moved to the network hardware using DMA (studied in IO and devices)
- **Where is the receiver-side socket located?**
  - Depends on where the application is currently scheduled (CPU scheduler)
- **How does the OS move packets from the network hardware to the receiver-side socket?**
  - OS gives hardware some memory addresses to write the data (virtual memory)
  - Network hardware copies data using DMA (studied in IO and devices)
  - The OS knows where the data is copied (virtual memory)
  - The OS performs receiver-side processing (reliability, strip off headers, etc…)
  - Tells the application where to read the data from (virtual memory)

Network Stack Data Path



Wake-up Application Thread.

Select the Hardware Queue for TX.

Select the CPU for TCP/IP Processing.

Packet Scheduling.

Poll for RX Packets.

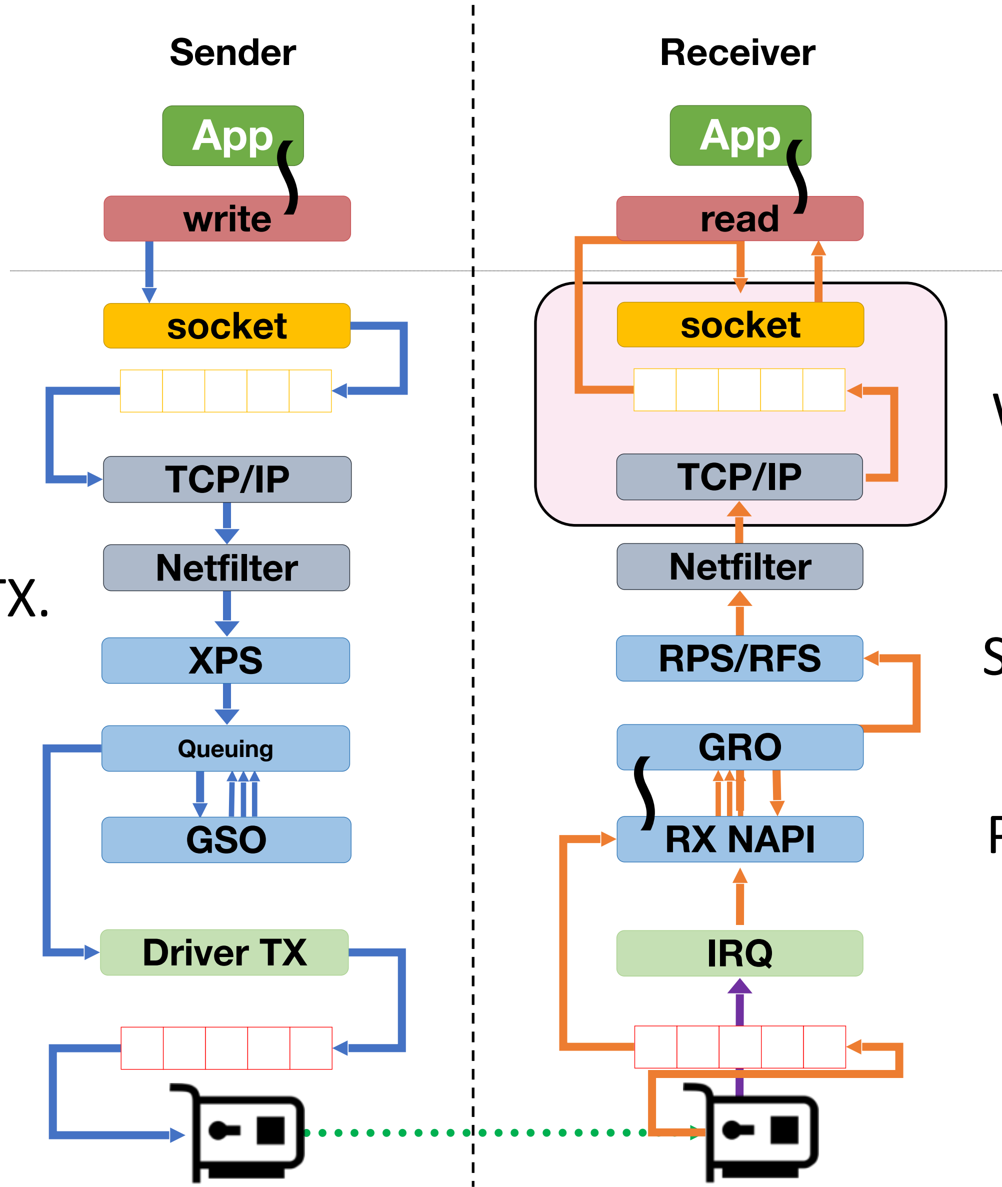
Write system call

- **Initiates data copy**
  - From the application buffers (address space) to kernel buffers

- **High CPU overheads**
  - Just moving data around (read from one buffer, write to another buffer)
  - All kinds of caching and page replacement issues come up

- **Packets are constructed at this point**
  - Push data to socket’s write queue until the queue is full
  - Block until queue is empty

Network Stack Data Path



Select the Hardware Queue for TX.

Packet Scheduling.

Wake-up Application Thread.

Select the CPU for TCP/IP Processing.

Poll for RX Packets.

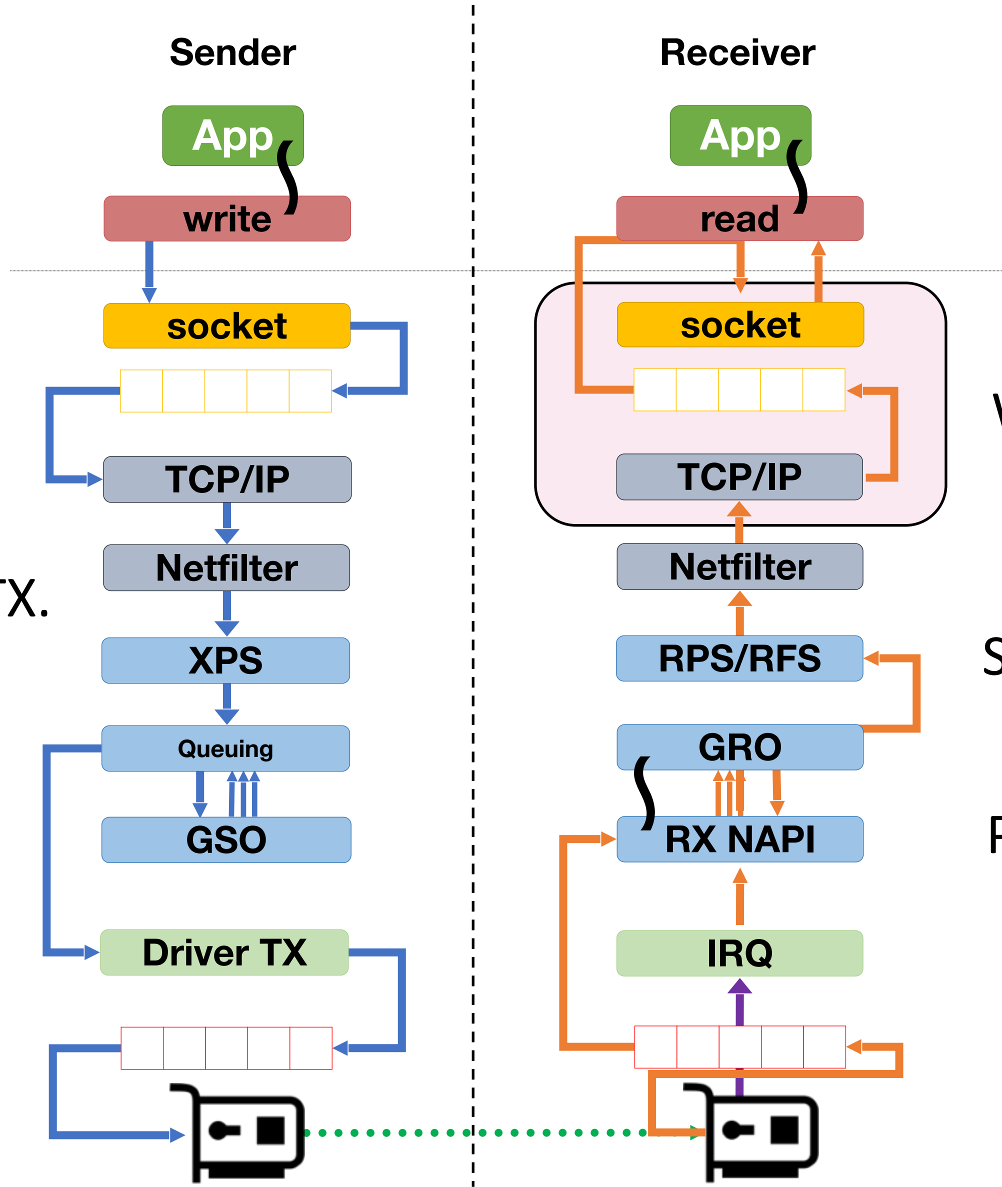
TCP/IP processing

- **All reliability-specific operations**

- **If protocol says okay to send data**
  - Pop packets from socket's write queue and push to the next layer
  - Must not delete packets yet, in case the packet gets lost in the network

- **Delete packets once ack-ed by the receiver**
  - A lot of book keeping (could be complicated)

Network Stack Data Path



Select the Hardware Queue for TX.

Packet Scheduling.

Wake-up Application Thread.

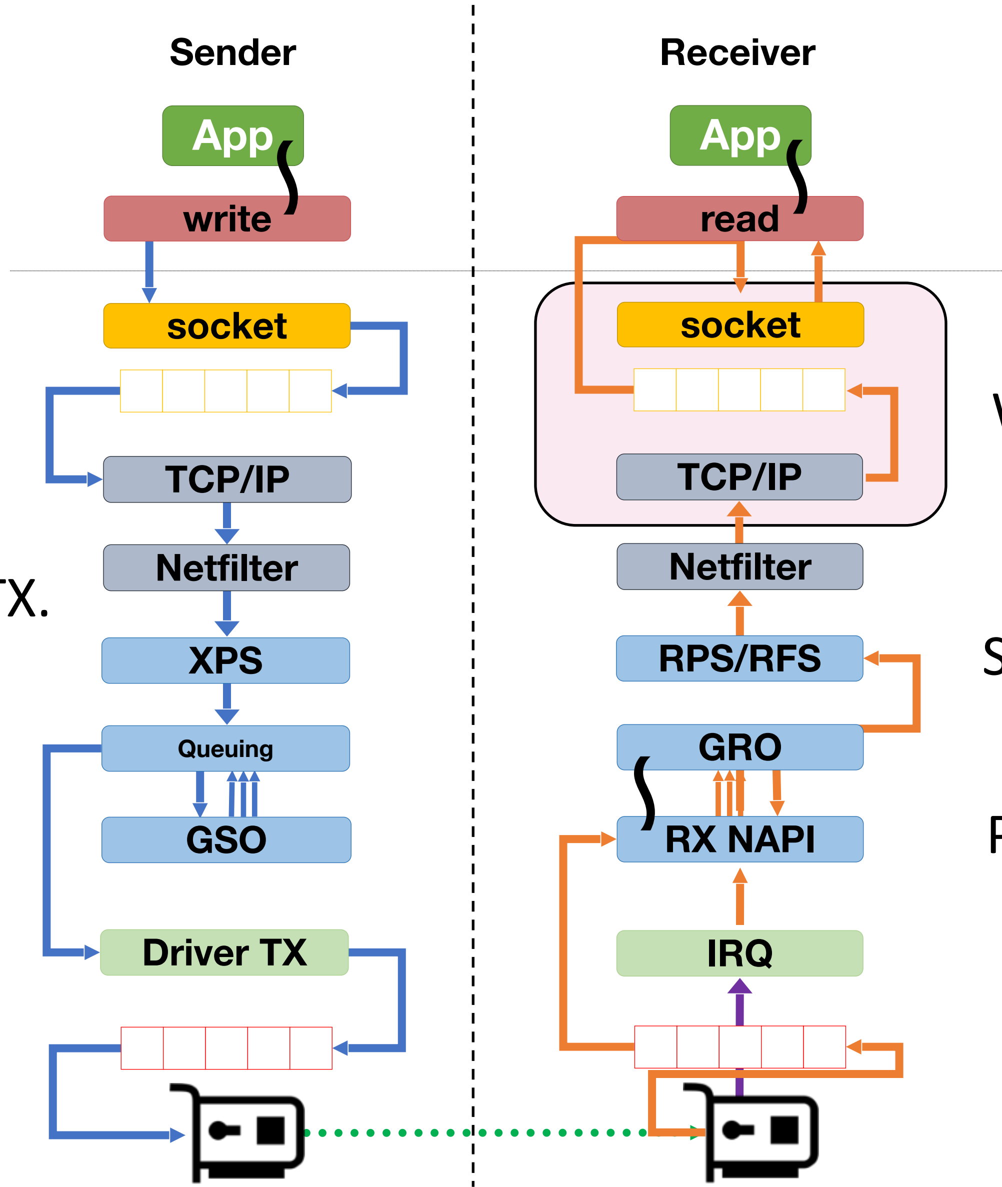
Select the CPU for TCP/IP Processing.

Poll for RX Packets.

NetFilter

- **Performs “filtering” of packets**
  - e.g., firewall
- **Network address/port translation**
  - E.g., when one wants to hide sender port/addresses from other servers
- In Linux, iptable and nftable commands are used for filtering
  - Lightweight

Network Stack Data Path



Wake-up Application Thread.

Select the Hardware Queue for TX.

Select the CPU for TCP/IP Processing.

Packet Scheduling.

Poll for RX Packets.

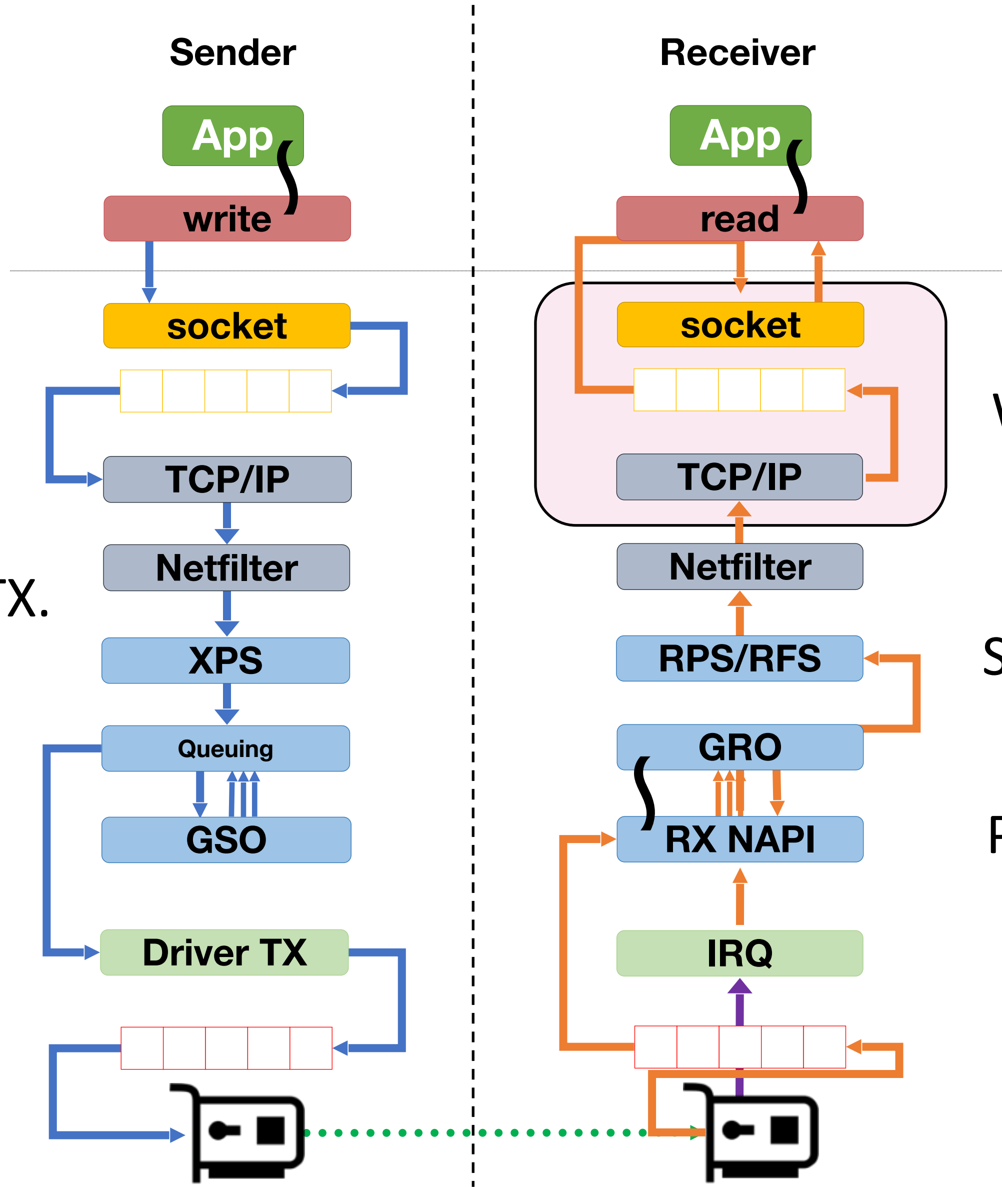
XPS

- **Network Hardware (NIC) has multiple queues**
    - Just like other storage hardware that we have discussed
- **To which queue should one forward packets from a particular socket?**
    - How should the mapping work?
        - All sockets forward to one queue?
        - Each socket is assigned its own queue?
        - If many-to-many mapping, how to map sockets to queues?
- Linux XPS layer is used to define/perform this mapping
    - Usually maps all sockets running on the same core to the same NIC queue
    - But can define any mapping

Network Stack Data Path

Select the Hardware Queue for TX.

Packet Scheduling.



Wake-up Application Thread.

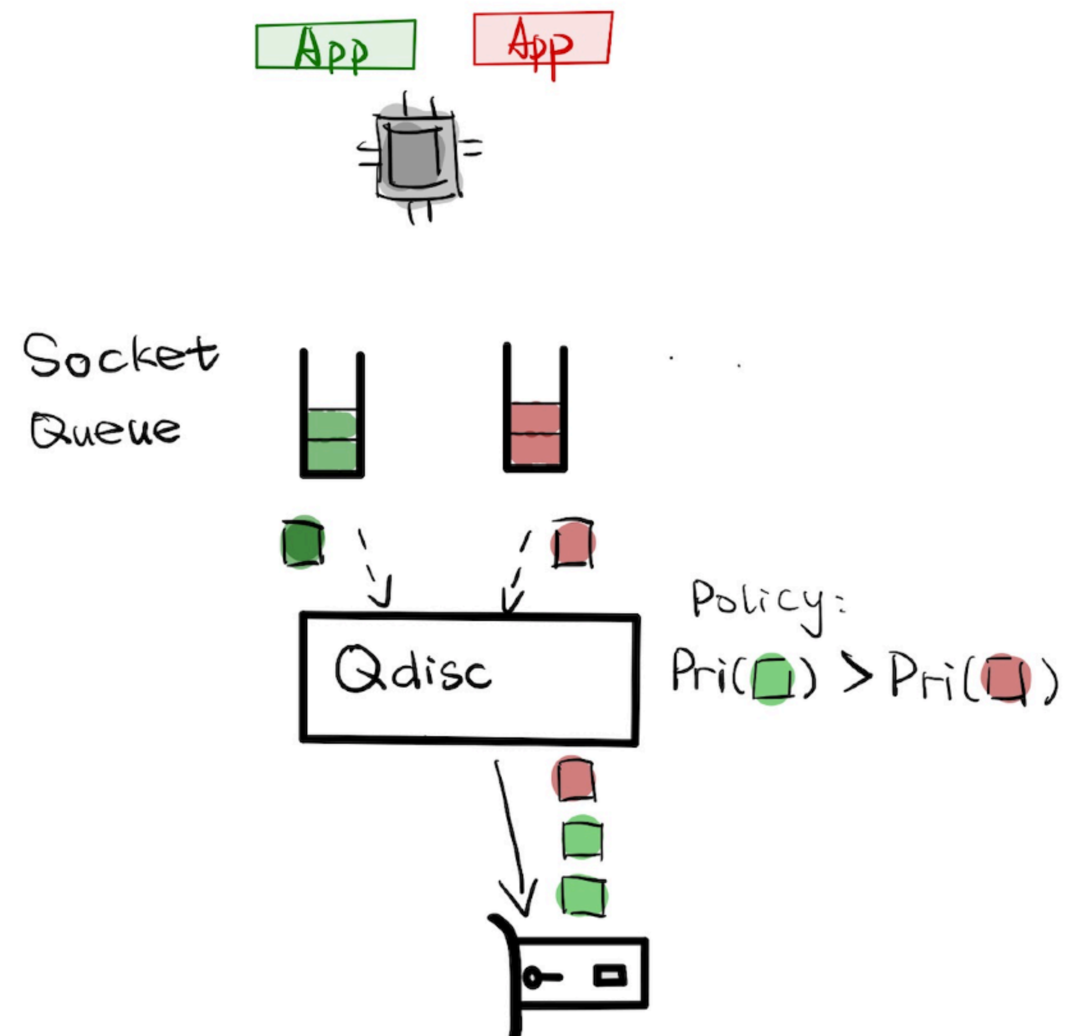
Select the CPU for TCP/IP Processing.

Poll for RX Packets.

Queueing Discipline

- **Performs "traffic shaping" and packet scheduling**
    - **Shaping:** how much bandwidth to give to each socket
    - **Scheduling:** among sockets mapped to a queue, which packet to choose next?
    - **Performed on a per-queue basis**

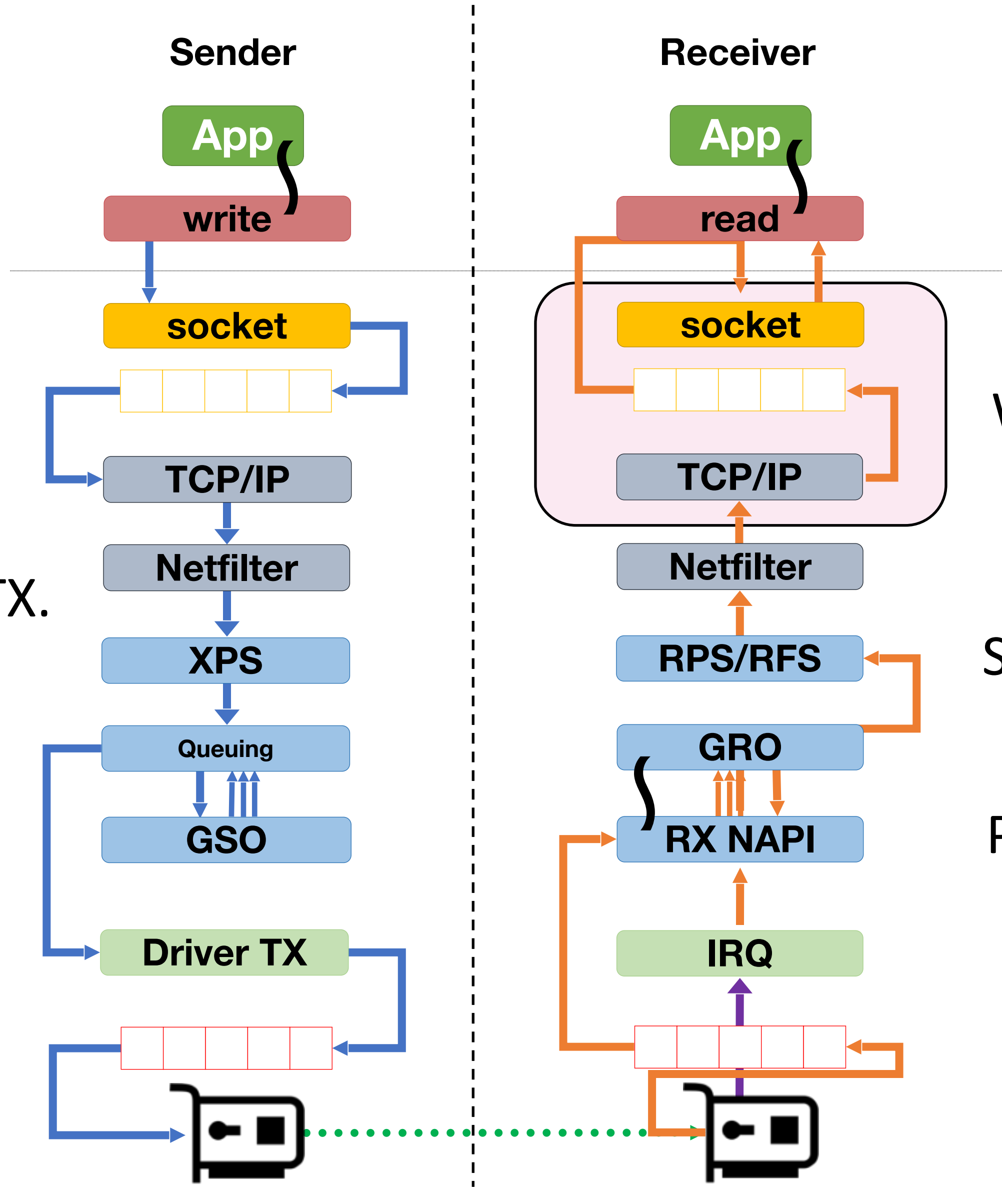
- **Each transmit queue has its own queueing discipline (qdisc) in the OS**
    - In Linux, **tc** command is used for managing qdisc



Network Stack Data Path

Select the Hardware Queue for TX.

Packet Scheduling.



Wake-up Application Thread.

Select the CPU for TCP/IP Processing.

Poll for RX Packets.

Segmentation

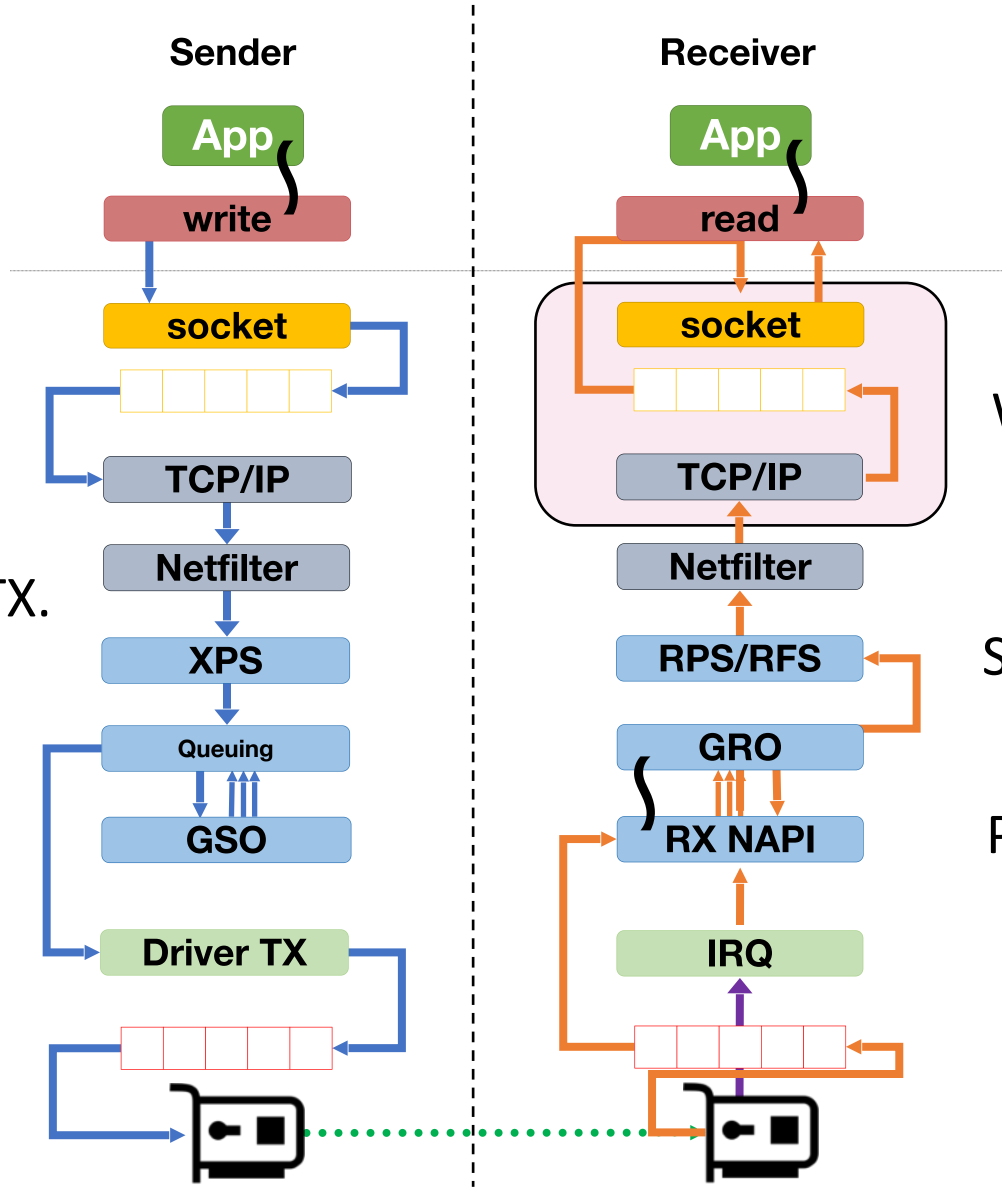
- **Traditionally, data processed and transmitted at 1500byte granularity**
  - But, if the application has a lot of data to send (on the same socket)
    - Many of the previous processing steps will be similar for all packets
    - Individual processing unnecessarily wastes CPU cycles
  - High packet processing overheads

- **General Segmentation Offload (GRO)**
  - Software-based solution to batch packet processing (e.g., 64KB granularity)
  - But packets transmitted at 1500byte granularity
  - Thus, once processed by the OS, we must "segment" packets before transmission

- GSO saves cycles for packet processing using batches of packets (~64KB)
  - But has overheads (implemented in software, after all): perform segmentation

- TCP Segmentation offload (TSO)
  - Perform packet processing in batches in the OS
  - Offload segmentation of packet batches to the hardware
  - Most NICs support TSO

Network Stack Data Path



Select the Hardware Queue for TX.

Packet Scheduling.

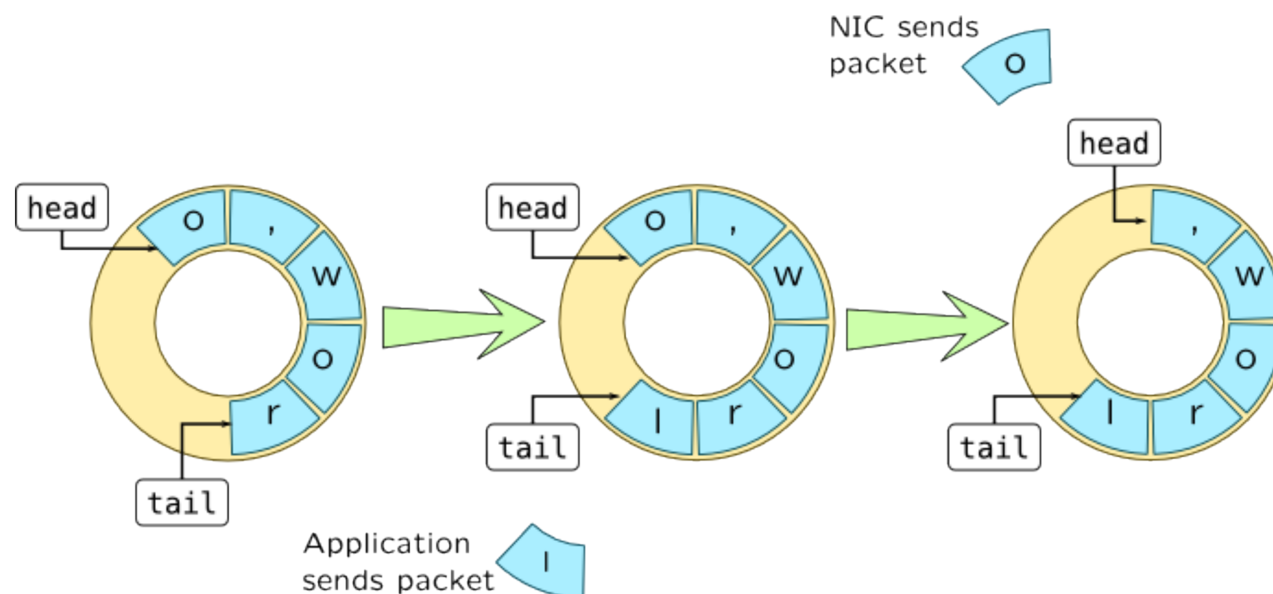
Wake-up Application Thread.

Select the CPU for TCP/IP Processing.

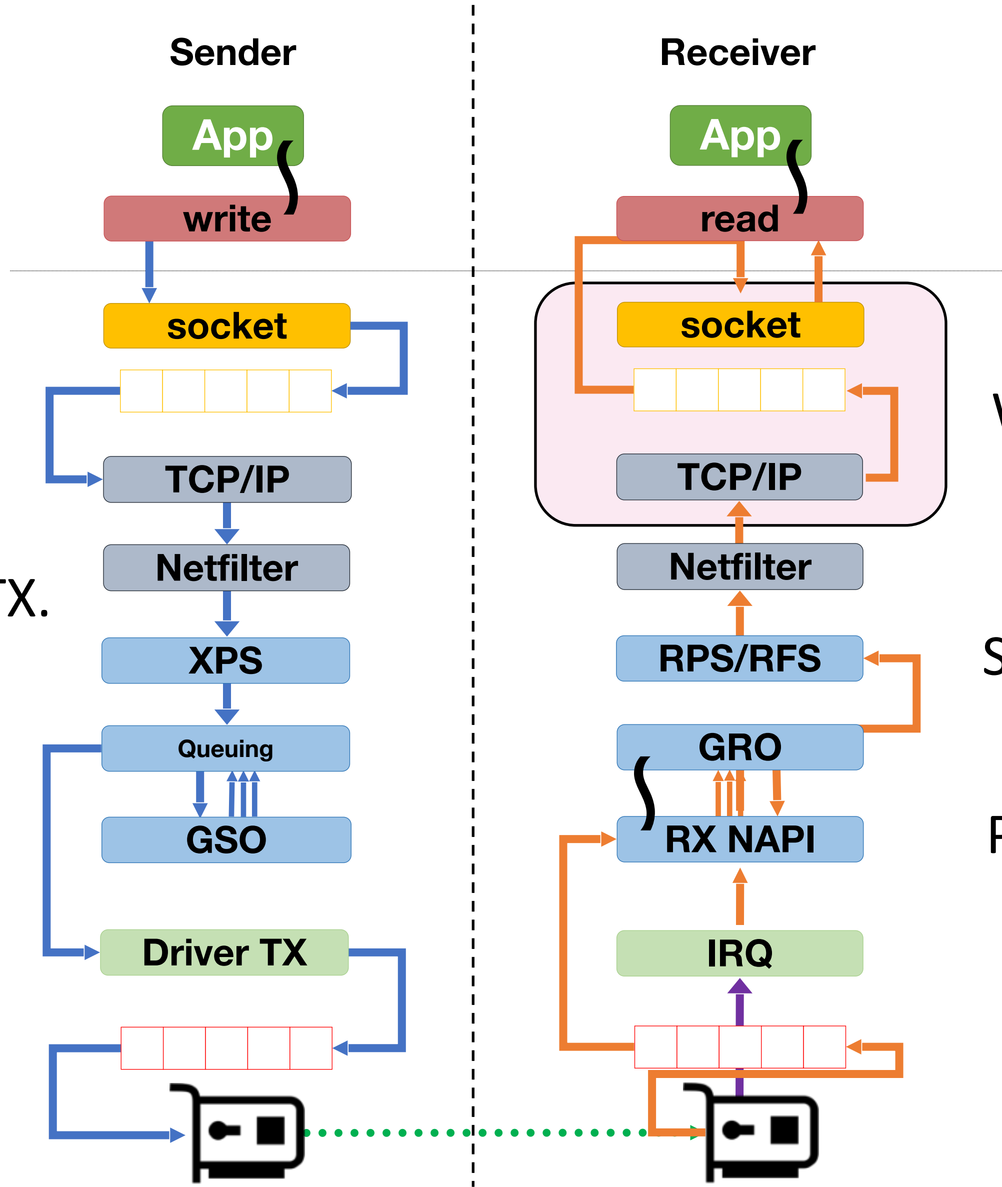
Poll for RX Packets.

Driver Tx

- **Manage "shared memory" between the NIC and the OS**
  - Share memory region: a ring (circular) buffer
  - Each element in the buffer referred to as a "packet descriptor"
    - Memory address where data in a particular packet will be copied
- **Operations:**
  - Write data into one of the descriptors
  - Signal to the NIC that data is ready to be transmitted (ring doorbell)
    - Ring doorbell in per-descriptor basis has high CPU overheads
  - NIC then fetches packets from DRAM pointed by the packet descriptor
  - Descriptors reinserted into the ring buffer as data in a descriptor is transmitted



Network Stack Data Path



Select the Hardware Queue for TX.

Packet Scheduling.

Wake-up Application Thread.

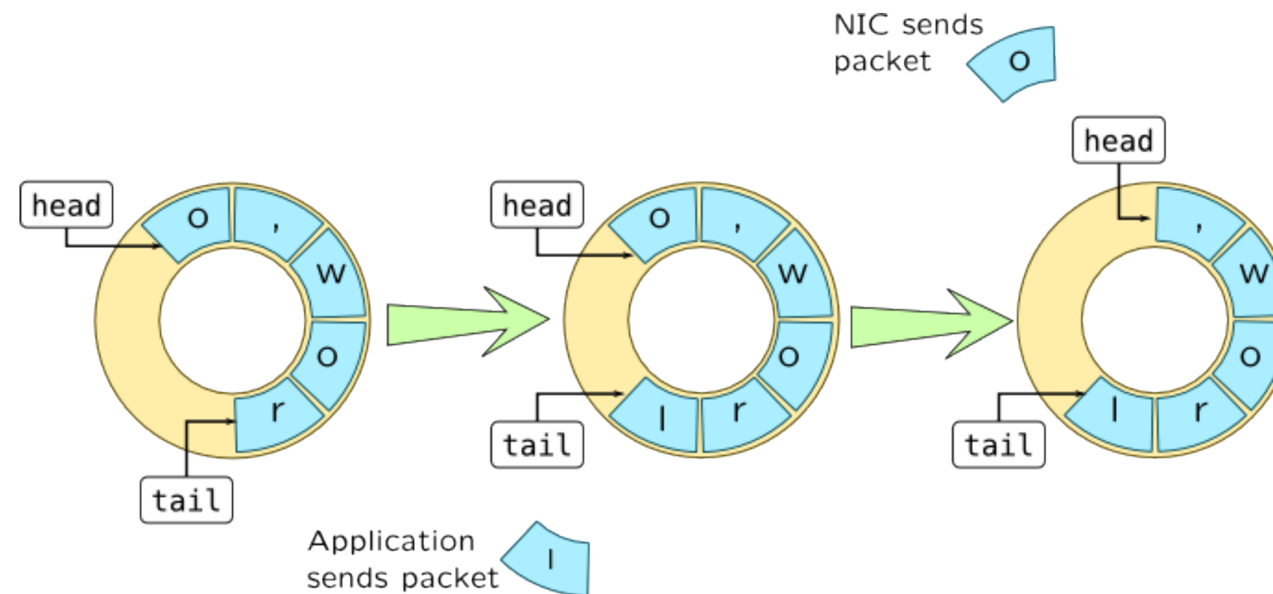
Select the CPU for TCP/IP Processing.

Poll for RX Packets.

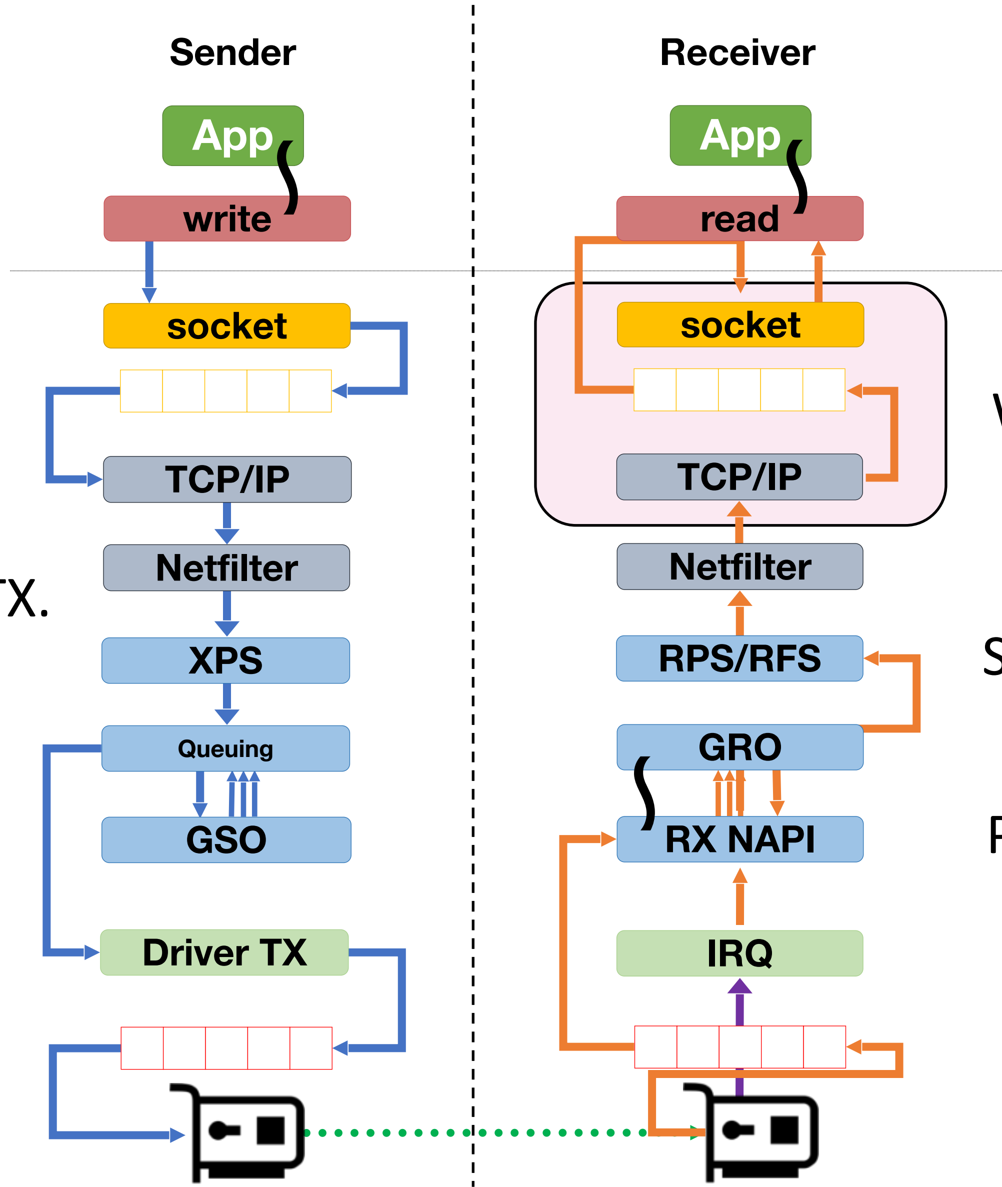
Driver Rx

- **NICs maintain multiple Rx ring buffers**

- **For each buffer, OS does the following operations:**
  - Prepare new descriptors for the NIC to do DMA
  - Push new descriptors containing empty pages to the ring buffer
  - Once NIC finishes DMA, unmap DMA mappings



Network Stack Data Path



Select the Hardware Queue for TX.

Packet Scheduling.

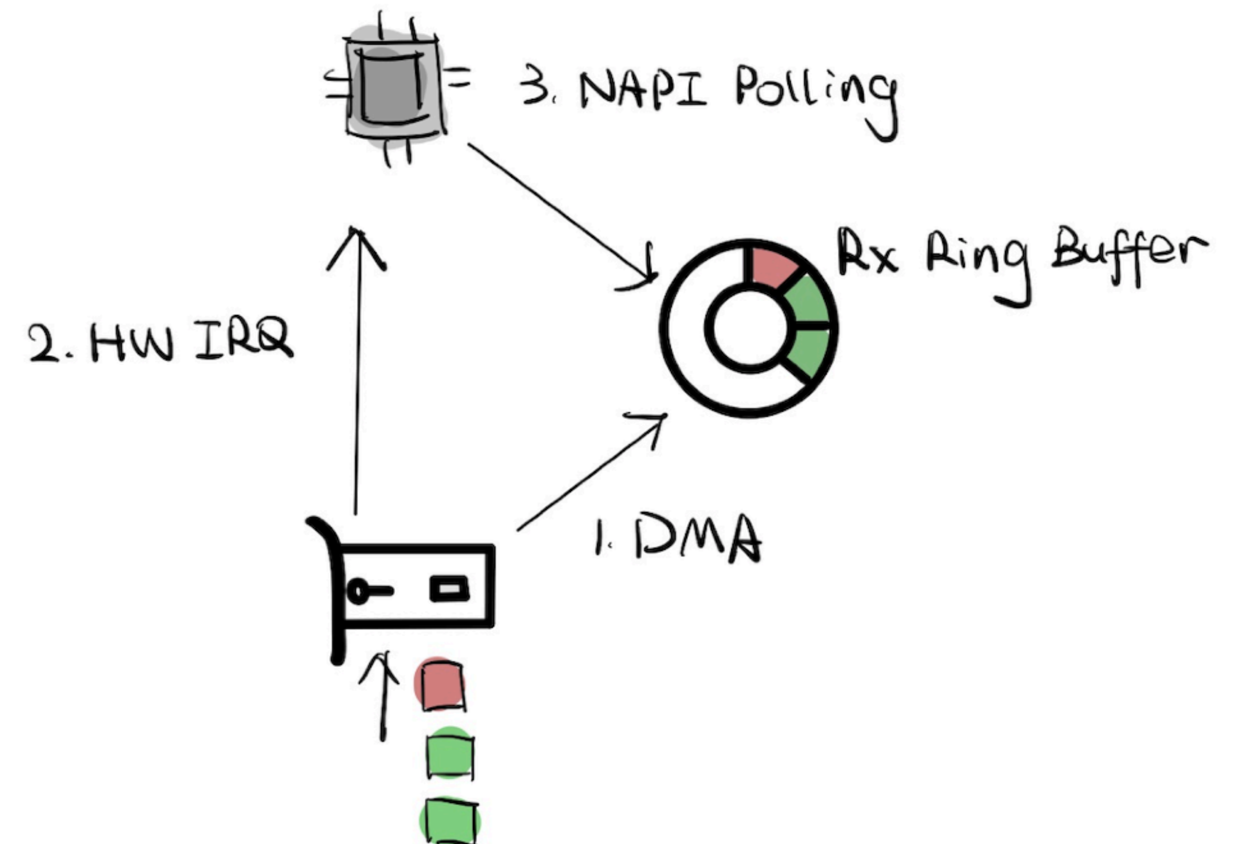
Wake-up Application Thread.

Select the CPU for TCP/IP Processing.

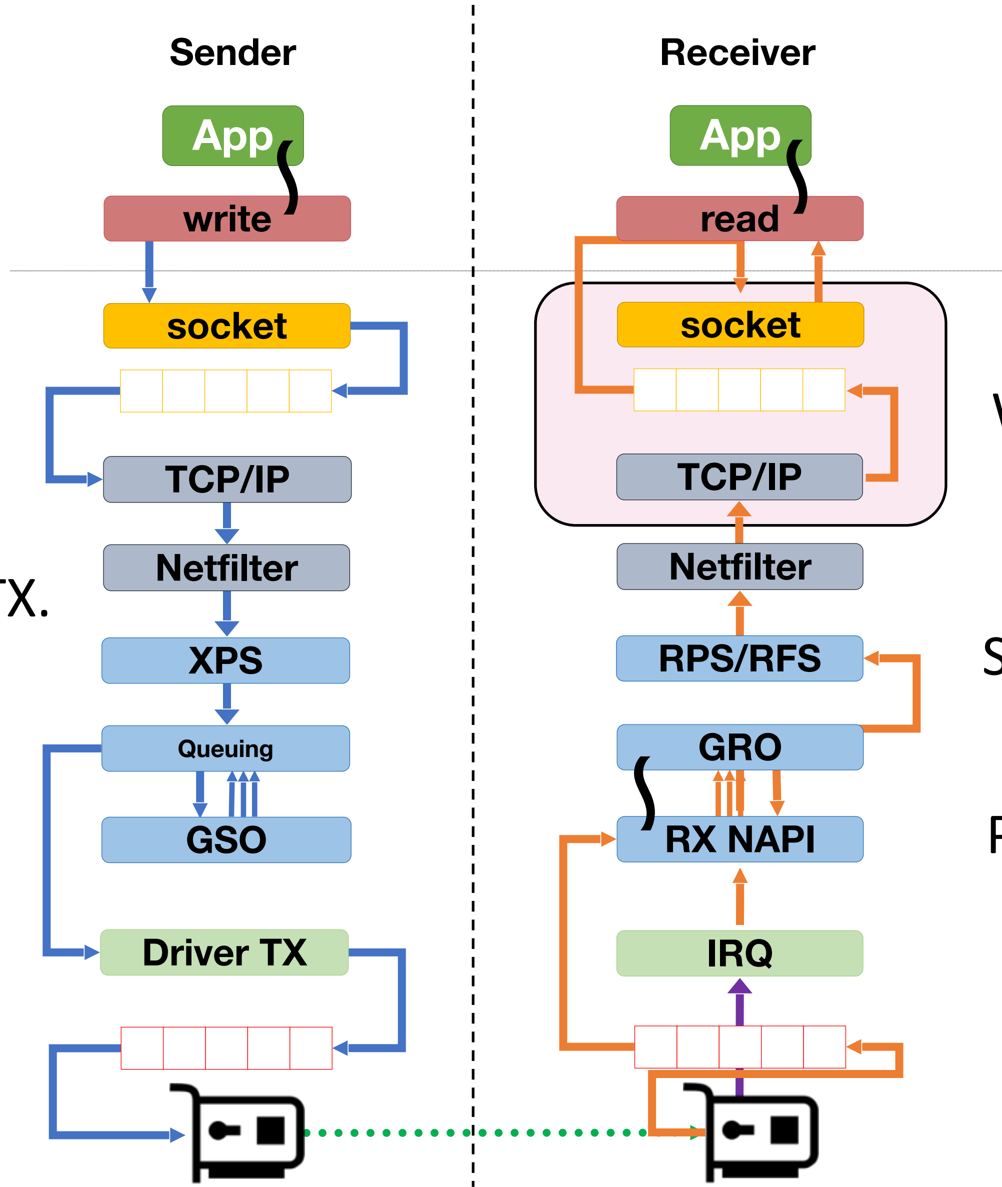
Poll for RX Packets.

IRQ Handling and NAPI

- **Packets are DMA-ed to Rx ring buffer in shared (kernel) memory**
- **NIC triggers an interrupt to wake up OS for handling packets**
  - Downside: per-packet interrupt has very high overheads
- **NAPI (new API): disable the interrupt and start a poll loop for handling packets**
  - Reduce #interrupts
  - Only the first packet triggers an interrupt



Network Stack Data Path



Wake-up Application Thread.

Select the Hardware Queue for TX.

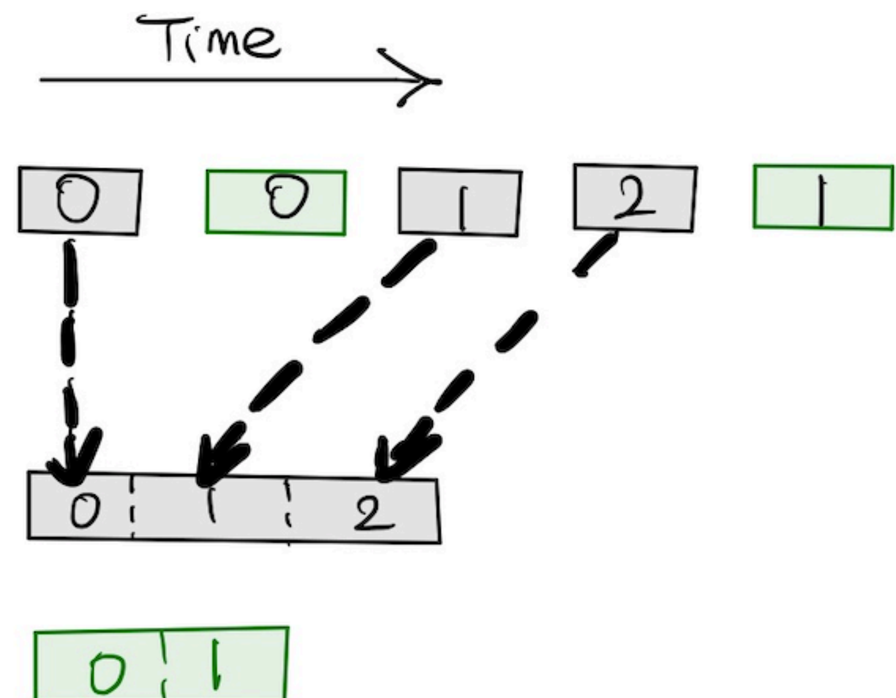
Select the CPU for TCP/IP Processing.

Packet Scheduling.

Poll for RX Packets.

Generic Receiver Offload (GRO)

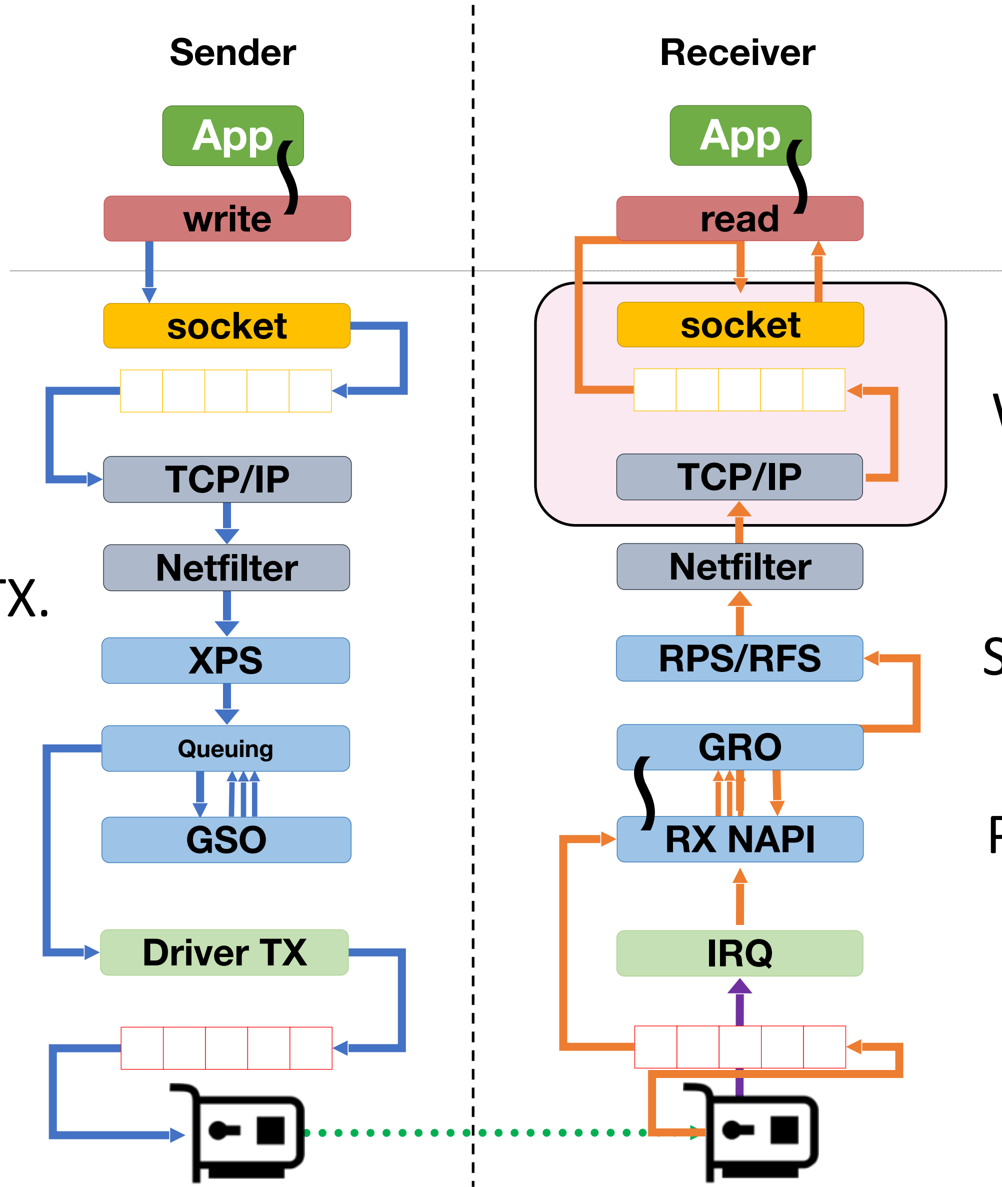
- **Receiver-side optimization similar to GSO/TSO**
- **Aggregate packets of the same flow before TCP/IP processing**
  - software-based
  - Extra CPU overheads (similar to GSO)
- **LRO: offload GRO to the hardware (NIC)**
  - Downside: NIC has limited memory to store packets



Network Stack Data Path

Select the Hardware Queue for TX.

Packet Scheduling.



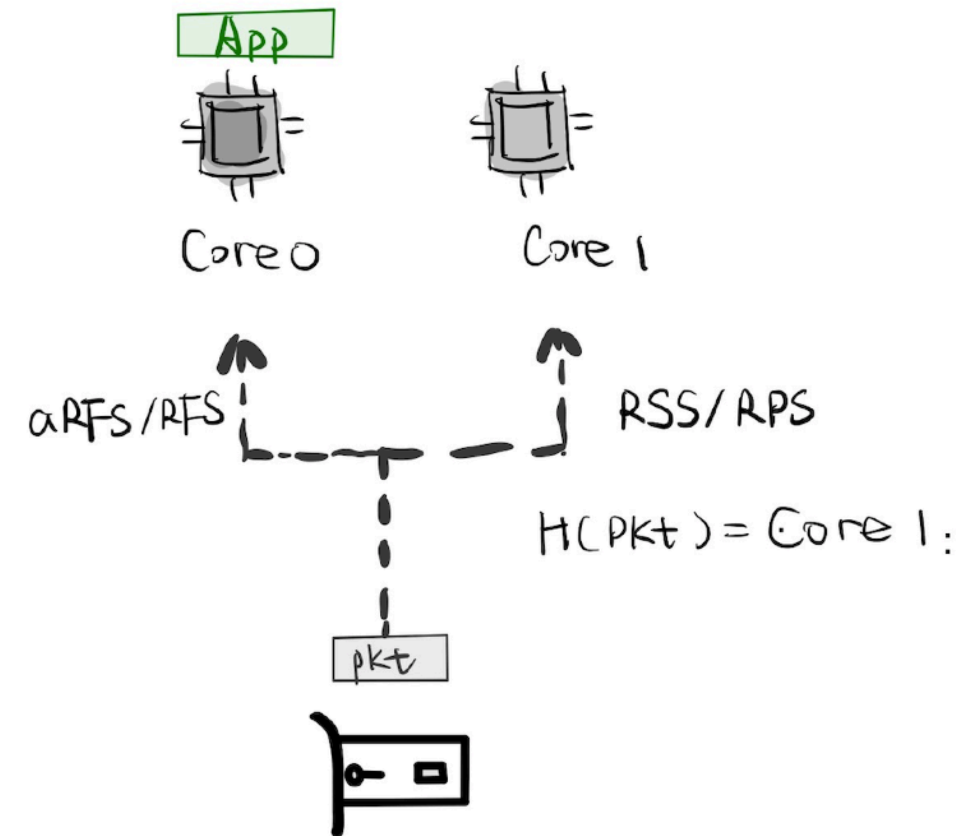
Wake-up Application Thread.

Select the CPU for TCP/IP Processing.

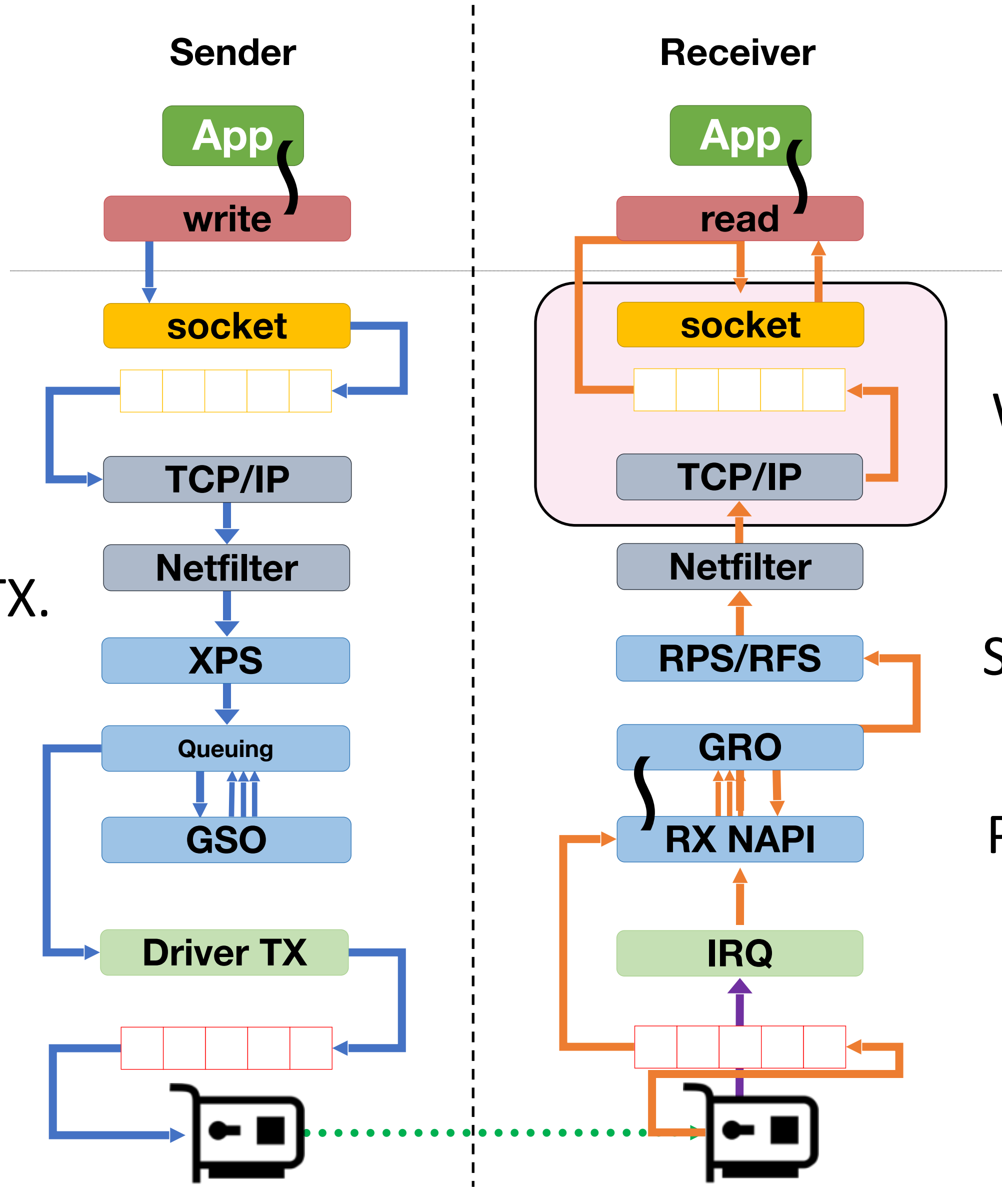
Poll for RX Packets.

Packet and flow steering

- **Which core should NIC forward packets to?**
- **RSS/RPS: choose core based on the hash of the packet header**
    - RSS: hardware-based; RPS: software-based
    - Scale packet processing with bandwidth
    - Downside: NUMA, cache unawareness
- **aRFS/RFS: choose core based on where the application is running**
    - aRFS: hardware-based; RFS: software-based
    - Benefits: Read/write data from local cache/memory
    - Downside: poor scalability when #apps in the same core increases



Network Stack Data Path



Wake-up Application Thread.

Select the Hardware Queue for TX.

Select the CPU for TCP/IP Processing.

Packet Scheduling.

Poll for RX Packets.

TCP/IP and read system call

- **Push packets to socket read queue**
- **Wake up application thread for copying data to the application buffers**
  - Downside: extra scheduling overhead/delay
- **Send ACK packets**
  - Sender can clear out packets that have been delivered