# CS4450

Computer Networks:
Architecture and Protocols

**Lecture 21**
**Host Network Stack**

**Midhul Vuppalapati**

# Goals for today's lecture

- **Deep dive into Host Network Stack**
  - Detailed overview of its functionality
  - Architecture of the Linux Network Stack
  - Overheads of network stack processing
  - Common optimizations used to minimize overheads

# Recap: Sockets and Ports

- 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
- Sender/destination addresses/names established before creating a socket

# Recap: End-to-end story

- **Application opens a <u>socket</u> 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 <u>packet header</u>**
- 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 <u>forward the packet to the right application</u>**

# Recap: Four fundamental problems

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

# Four fundamental problems — Role of Network Stack

- **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

# Why care about the Host Network Stack?

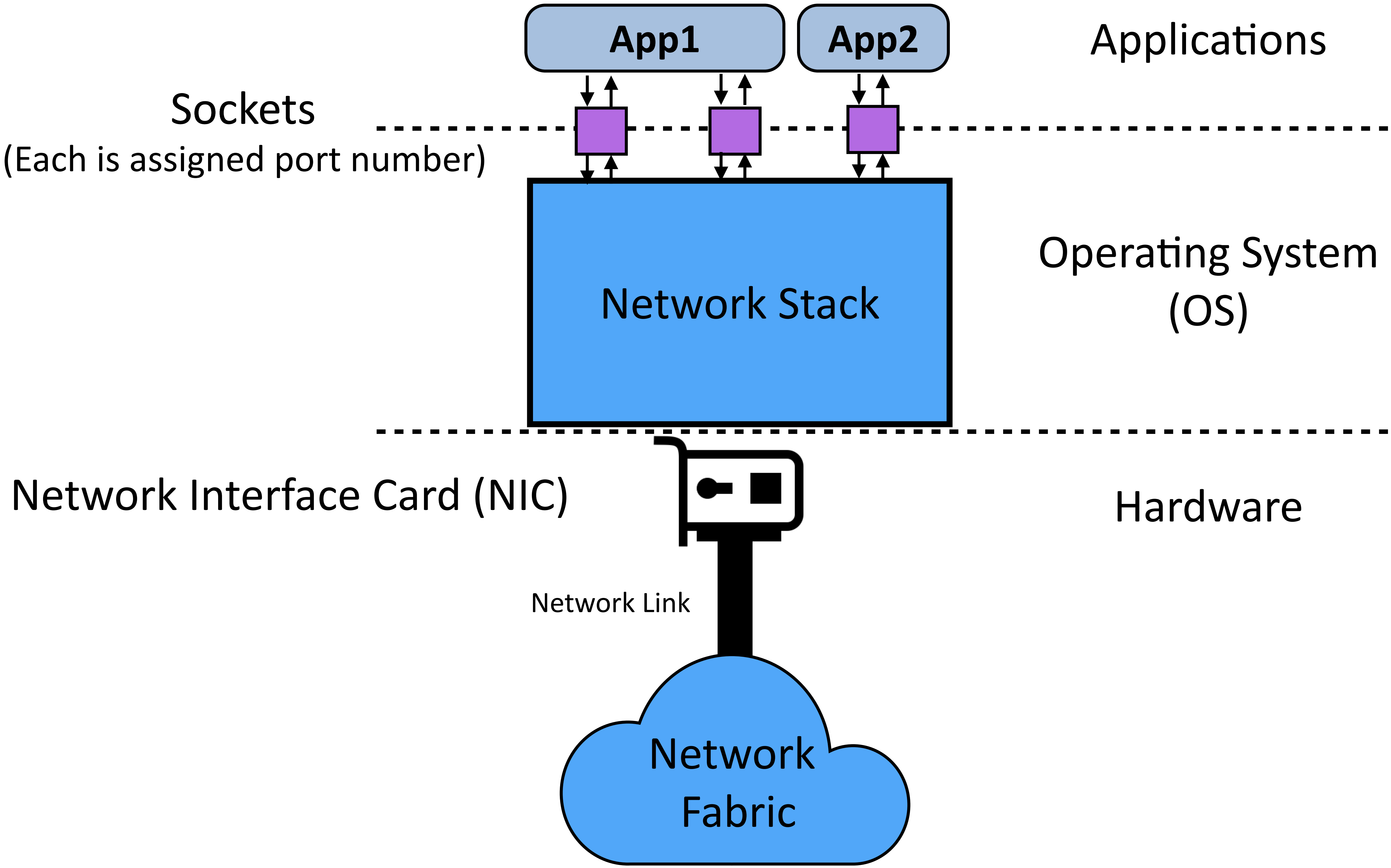
- Network stack processing consumes CPU resources

- Every CPU cycle consumed by Network Stack
  - is a CPU cycle taken away from Applications

- **Challenge:** Designing an **efficient** Host Network Stack
  - Minimize overheads of Network Stack processing

- Recent Technology Trends (more details in later lecture)
  - Network link bandwidths are growing rapidly (esp. in Datacenters)
  - CPU speeds are not growing
  - **Host network stack is becoming a bottleneck**
  - Even more important to design efficient Host Network Stack

# Big picture of a Host



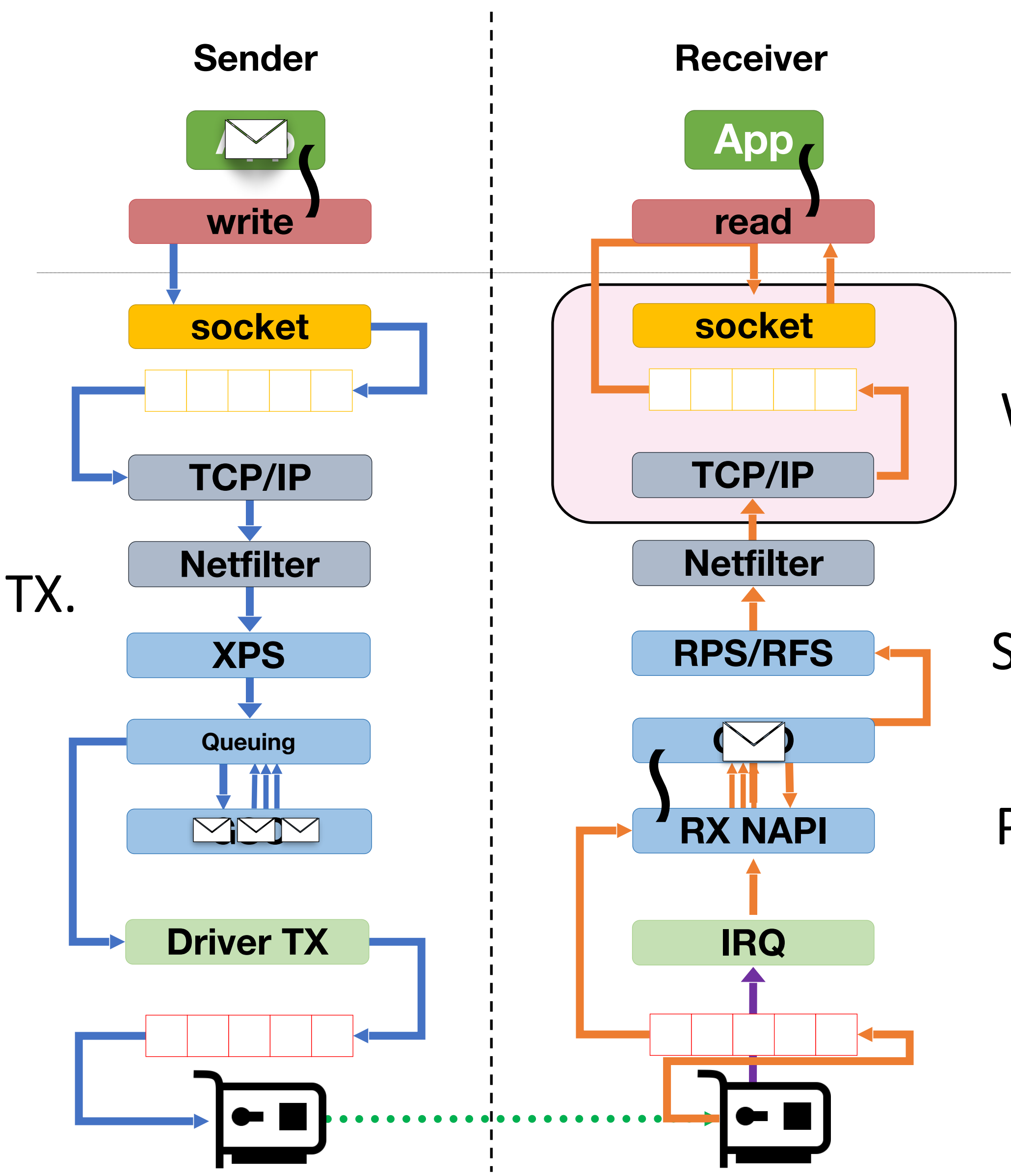
# Background: Network Interface Card (NIC)

- Input / Output (I/O) Device that connects Host to the network
- Implements Data Link & Physical Layer functionality
- **Modern NICs expose multiple hardware queues**
  - Transmit (Tx) Queues: for transmitting data over network link
  - Receive (Rx) Queues: for receiving data from network link
- **How do NIC and Network Stack interact with each other?**
  - **Data Transfer: <u>Direct Memory Access (DMA)</u>**
    - NIC reads/write data from/to memory
  - **Signaling:**
    - Network stack signals NIC: **<u>Doorbells</u>**
    - NIC signals Network Stack: **<u>Interrupts (IRQs)</u>**

# Linux Network Stack

- One of the most widely used Network Stacks today
- Has evolved over multiple decades
- Many different components
- Many different protocols (our focus: TCP/IP)
- **Heads up:** Some of the terminology may seem overwhelming
    - But the key ideas are simple

# Network Stack Data Path




Select the Hardware Queue for TX.

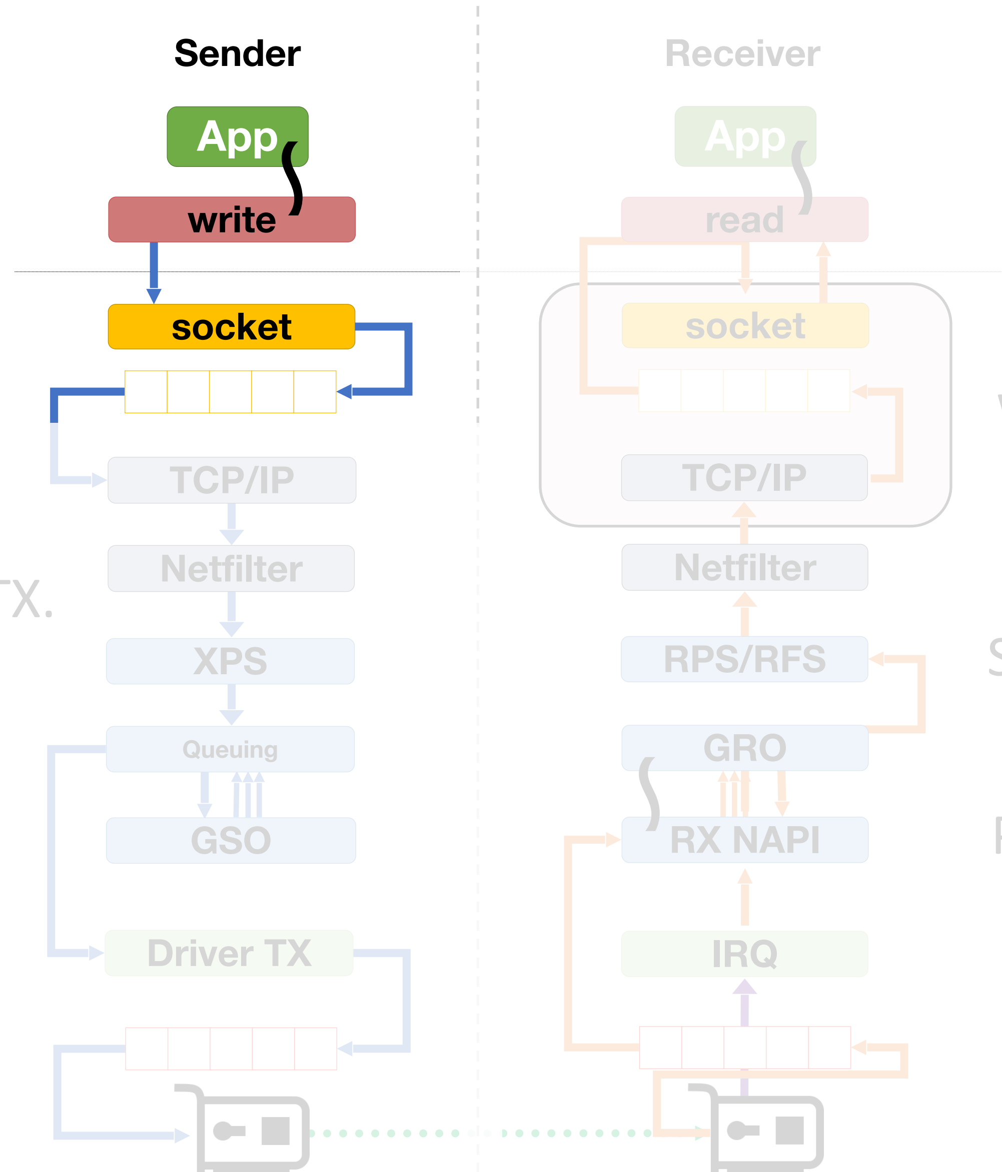
Packet Scheduling.

Wake-up Application Thread.

Select the CPU for TCP/IP Processing.

Poll for RX Packets.

# 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.

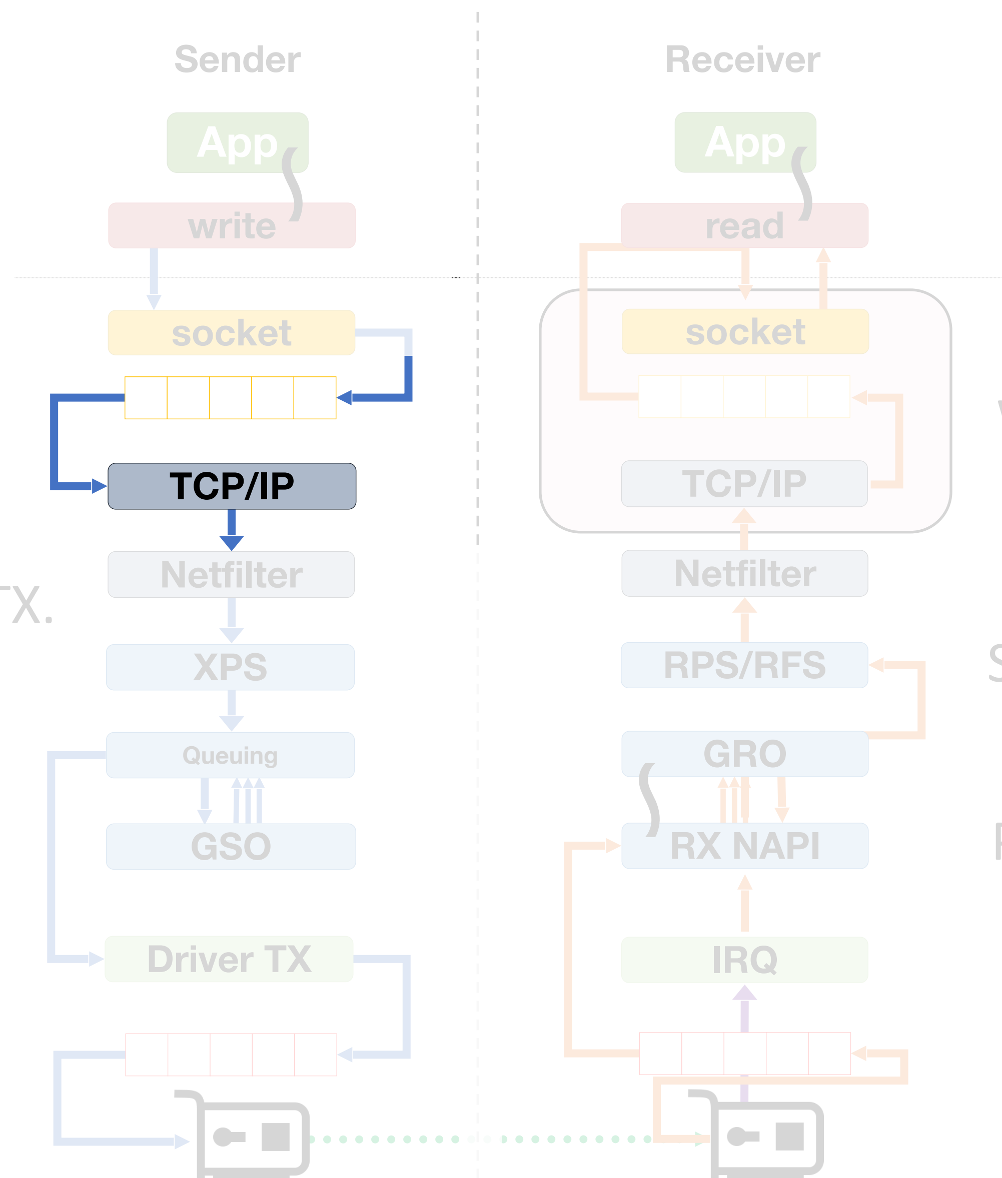
# Write system call

- **Initiates data copy**
  - From the application buffers to OS 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.

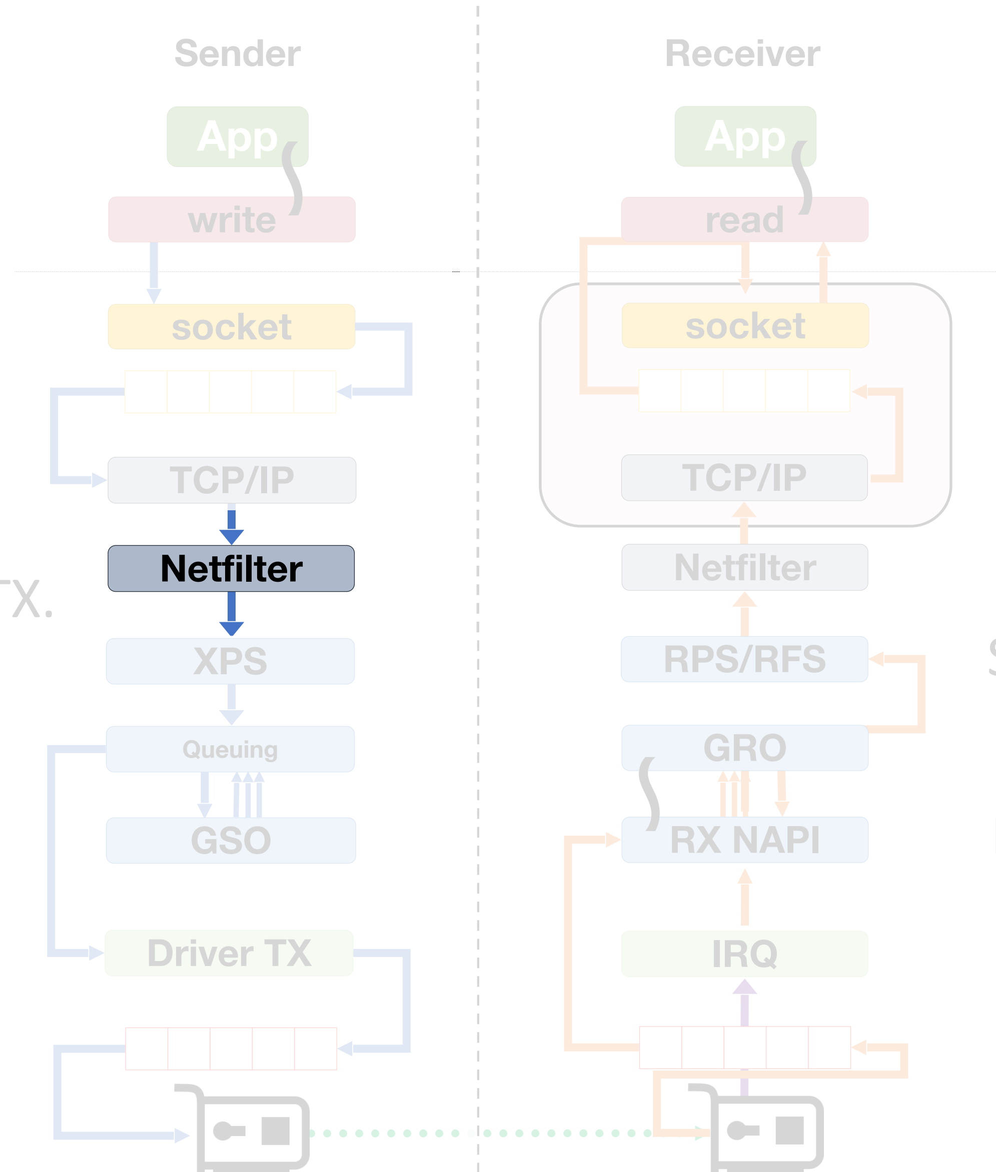
# TCP/IP processing

- **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 keep packets, 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.

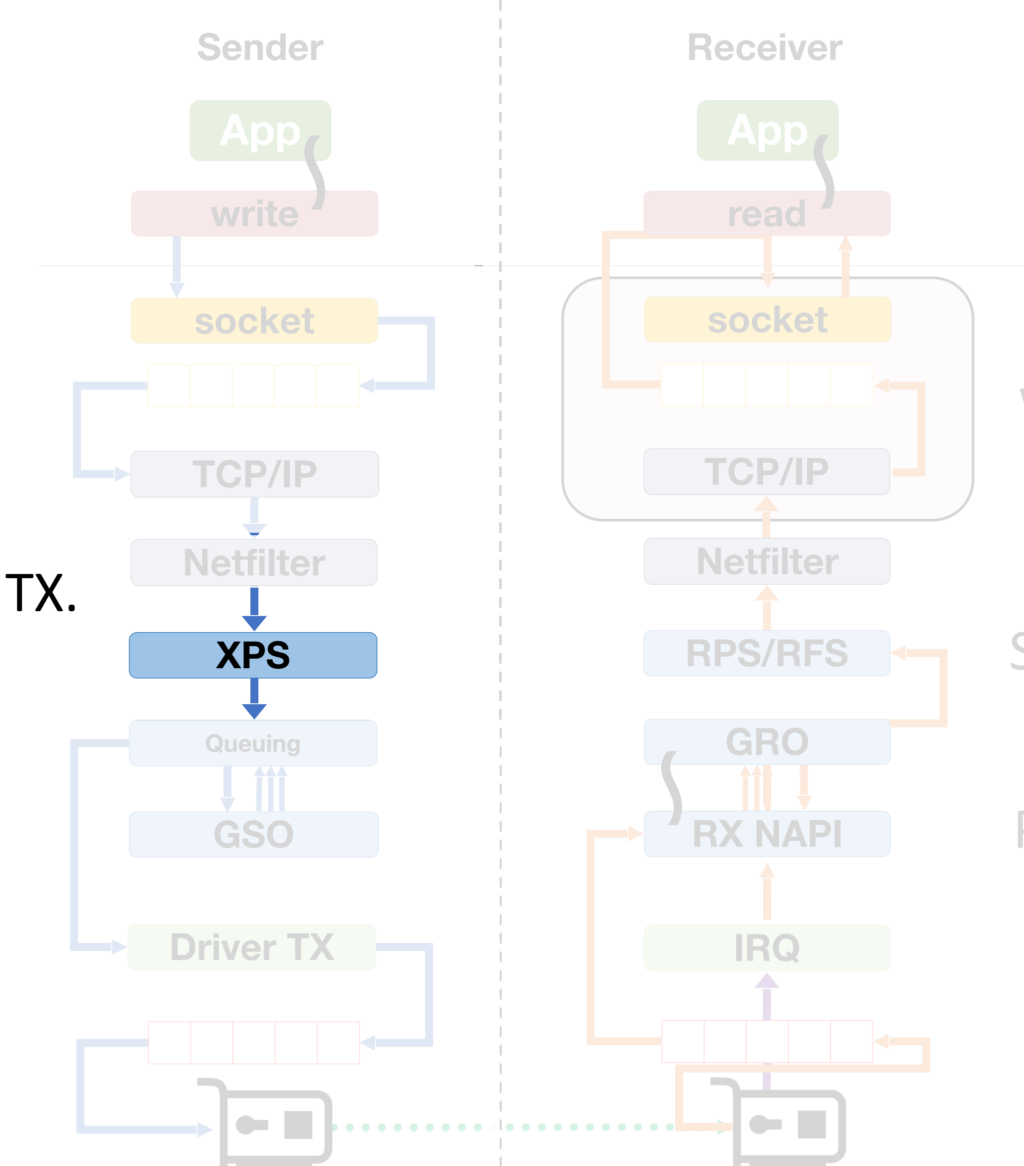
# NetFilter

- **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

Select the Hardware Queue for TX.

Packet Scheduling.




Wake-up Application Thread.

Select the CPU for TCP/IP Processing.

Poll for RX Packets.

# XPS

- **NIC has multiple Transmit (TX) Queues**

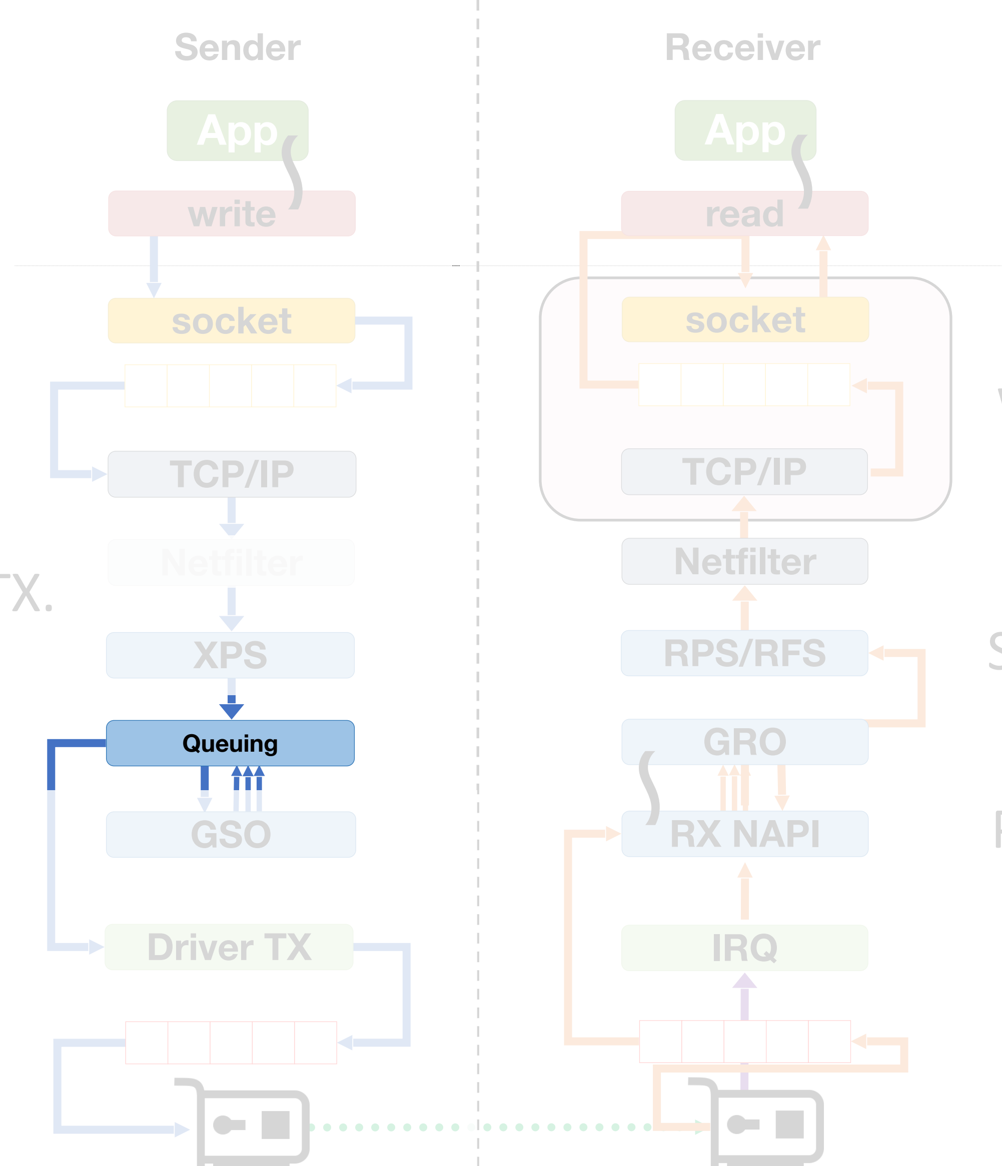
- **To which queue should one forward packets from a particular socket?**
  - How should the mapping work?
    - All sockets forward to one queue?
    - Each socket is assigned its own queue?
    - If many-to-many mapping, how to map sockets to queues?

- 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.

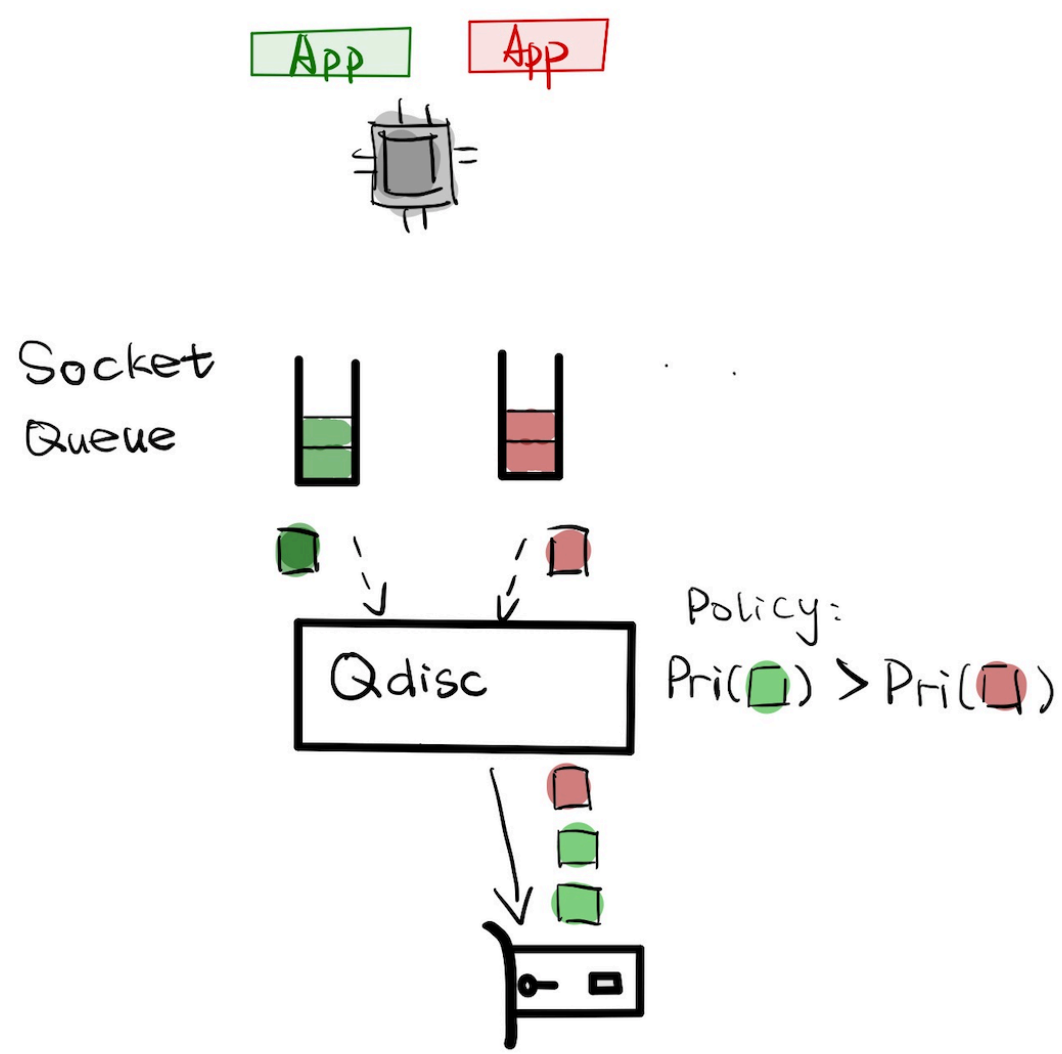
Select the CPU for TCP/IP Processing.

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 NIC queue basis**

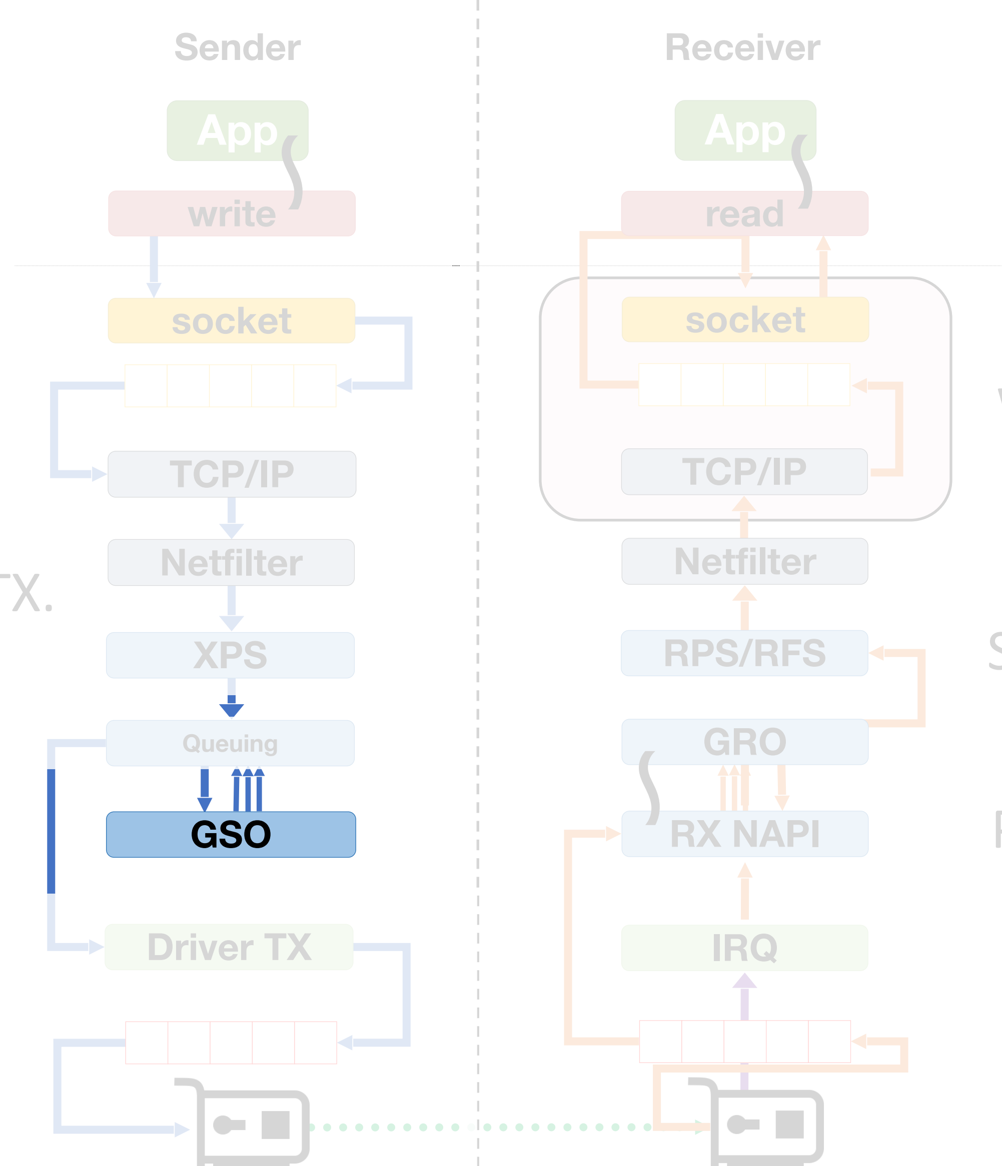
- **Each transmit queue has its own queueing discipline (qdisc) in the OS**
  - In Linux, **tc** command is used for managing qdisc



# 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
    - 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 (GSO)**
  - Software-based solution to batch packet processing
  - But packets transmitted at 1500byte granularity
  - Thus, once processed by the OS, we must “segment” packets before transmission

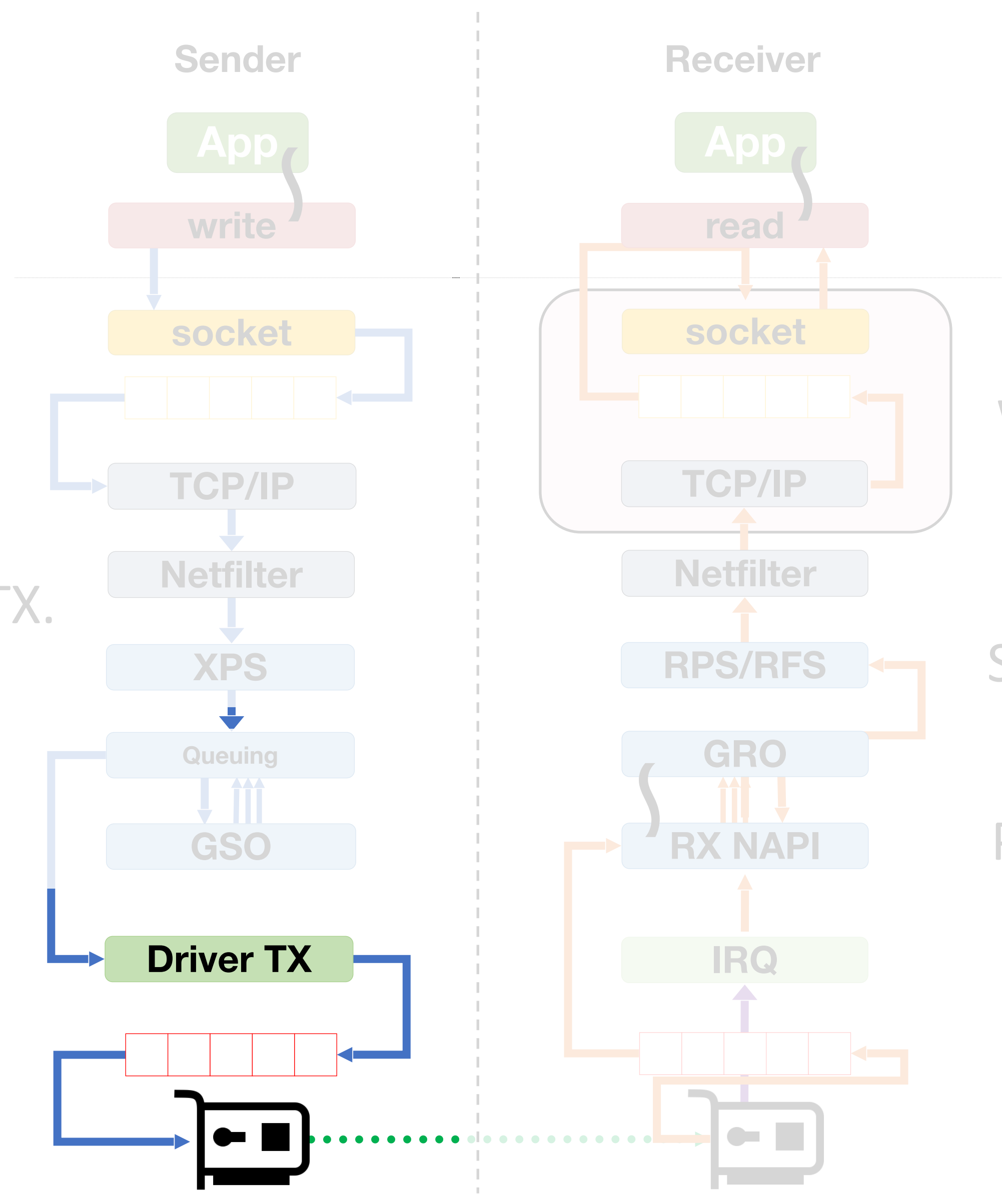
- GSO saves cycles for packet processing using batches of packets (~64KB)
  - But has overheads (implemented in software, after all): perform segmentation

- TCP Segmentation offload (TSO)
  - Always 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.



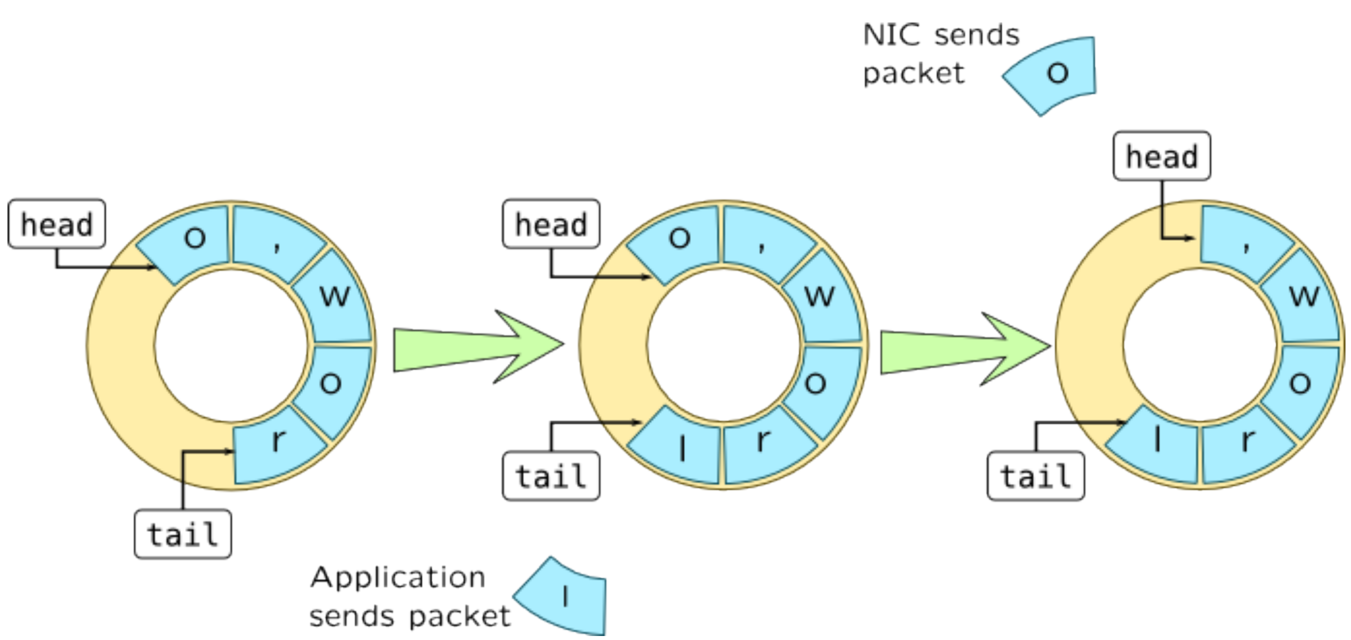

Wake-up Application Thread.

Select the CPU for TCP/IP Processing.

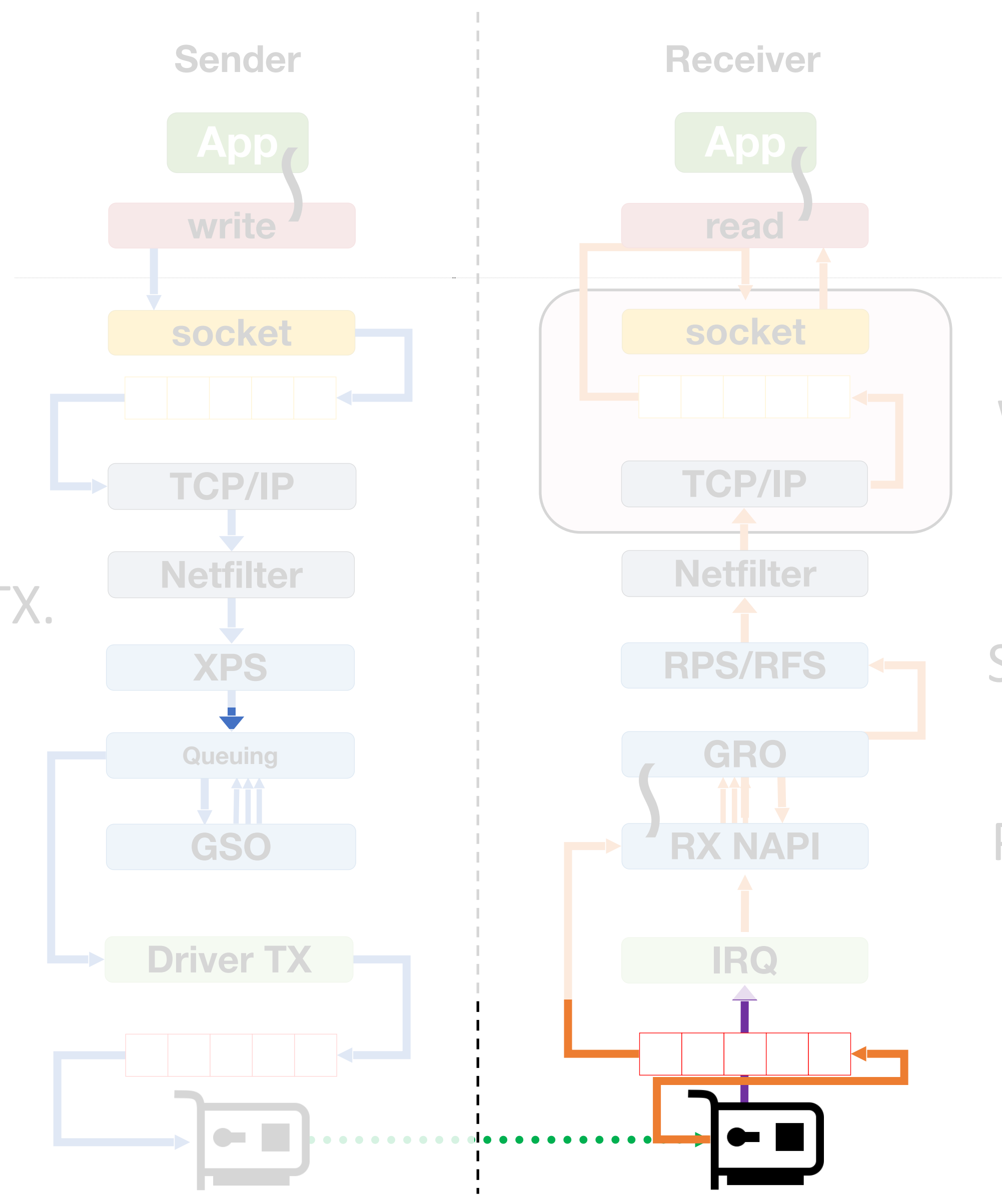
Poll for RX Packets.

# Driver Tx

- **Manage "shared memory" between the NIC and OS**
  - Shared memory region: a ring (circular) buffer (per NIC TX queue)
  - Each element in the buffer referred to as a "packet descriptor"
    - Memory address where data for a particular packet is present
- **Operations:**
  - Write data into one of the descriptors
  - Signal to the NIC that data is ready to be transmitted (ring doorbell)
  - NIC fetches packets from host memory pointed to by the packet descriptor
  - Descriptors re-inserted into the ring buffer once data in a descriptor is transmitted



# 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.

# Driver Rx

- **NIC maintains multiple Rx ring buffers (one per NIC RX queue)**

- **For each Rx ring buffer, network stack does the following operations:**
  - Allocate empty OS buffers for NIC to do DMA
  - Prepare new descriptors pointing to these OS buffers
  - Push descriptors to the ring buffer
  - Replenish the ring buffer with new descriptors so NIC can continue to do DMA

# Network Stack Data Path

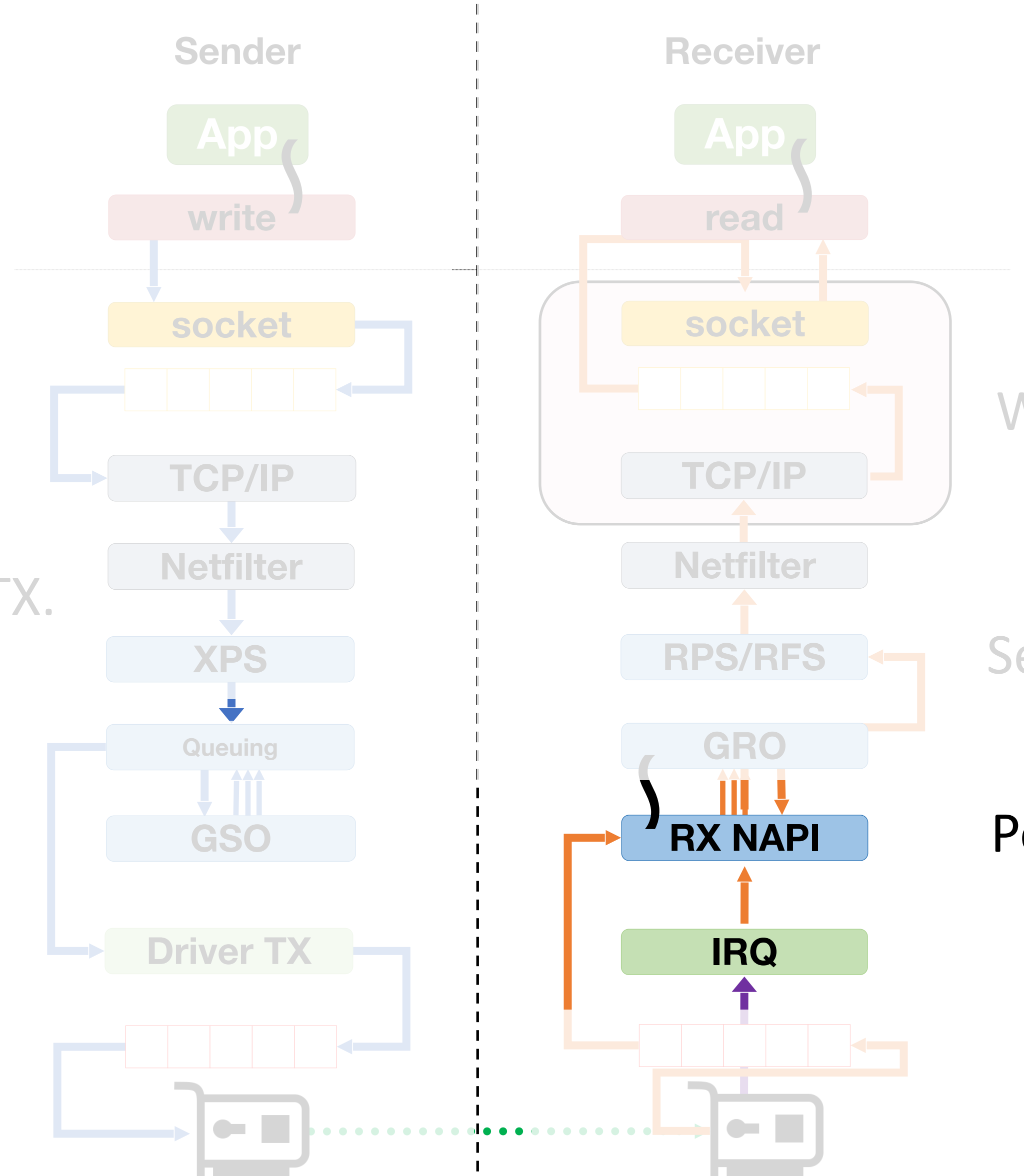
Wake-up Application Thread.

Select the Hardware Queue for TX.

Select the CPU for TCP/IP Processing.

Packet Scheduling.

Poll for RX Packets.



# IRQ Handling and NAPI

- **Packets are DMA-ed to OS memory buffers (based on descriptors in Rx ring buffer)**

- **NIC triggers interrupt (IRQ) to wake up OS for handling packets**
  - Downside: per-packet interrupts have very high overheads

- **NAPI (new API): disable the interrupt and start the poll loop for handling packets**
  - Reduces # of interrupts => lower overheads
  - Only the first packet triggers an interrupt

# Network Stack Data Path

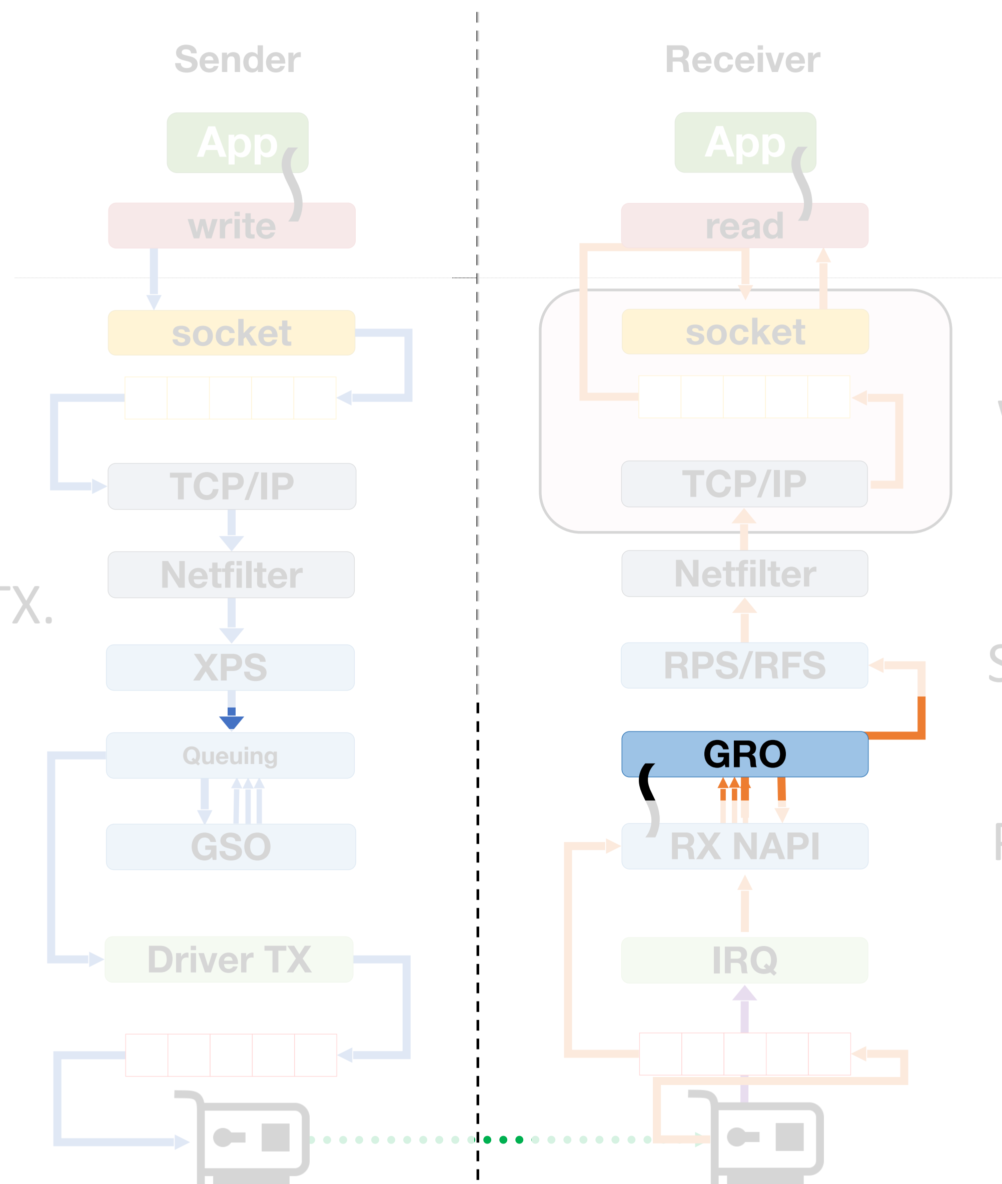
Wake-up Application Thread.

Select the Hardware Queue for TX.

Select the CPU for TCP/IP Processing.

Packet Scheduling.

Poll for RX Packets.



# Generic Receive Offload (GRO)

- **Receiver-side optimization similar to GSO/TSO**

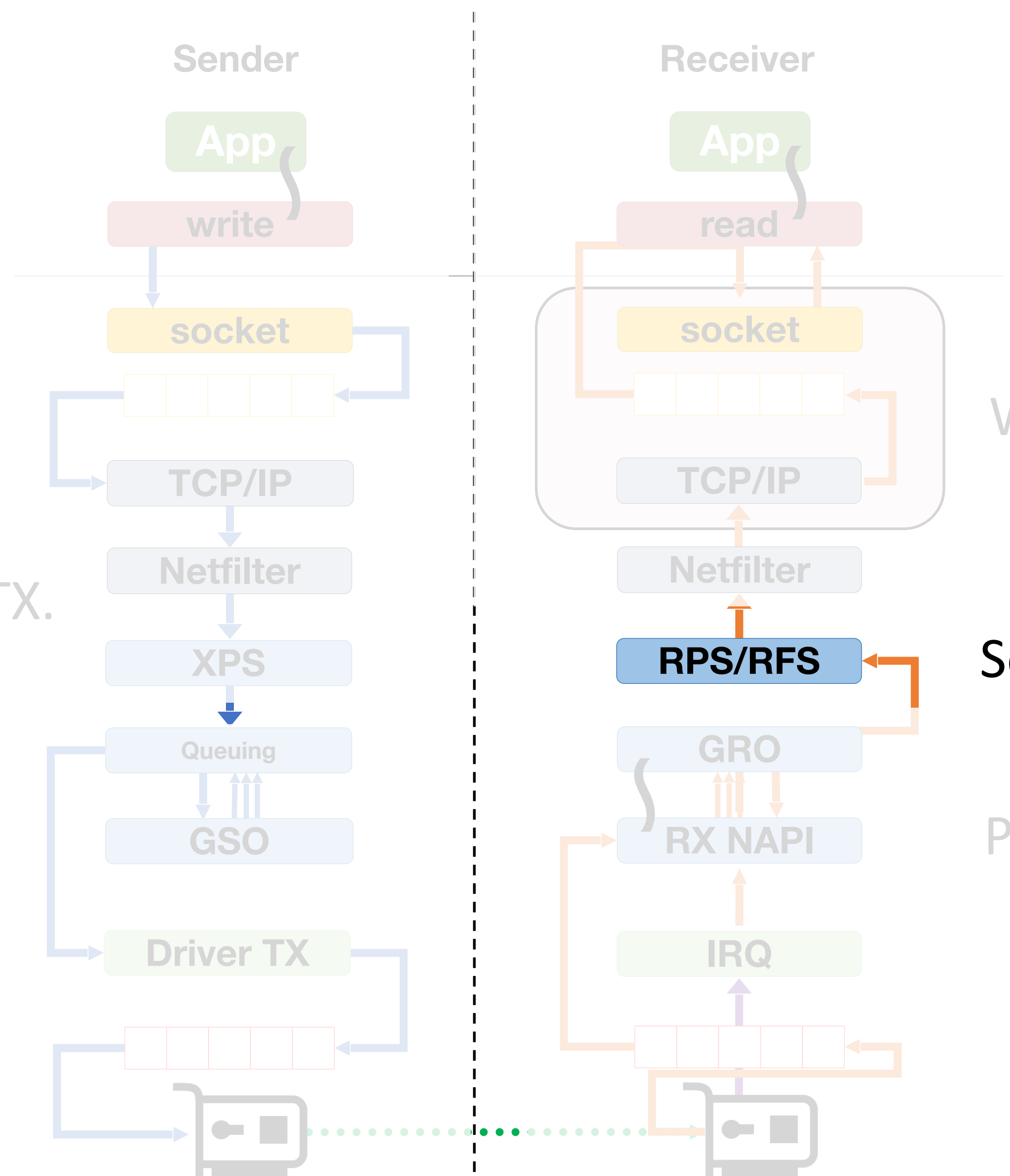
- **Aggregate packets of the same connection before passing it upper layers**
  - Reduces processing overheads of upper layers
  - Aggregation is software-based
    - Cost: Extra CPU overheads (similar to GSO)

- **LRO: Offload GRO to hardware (NIC)**
  - Can get the benefits of GRO without extra CPU overheads
  - 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.

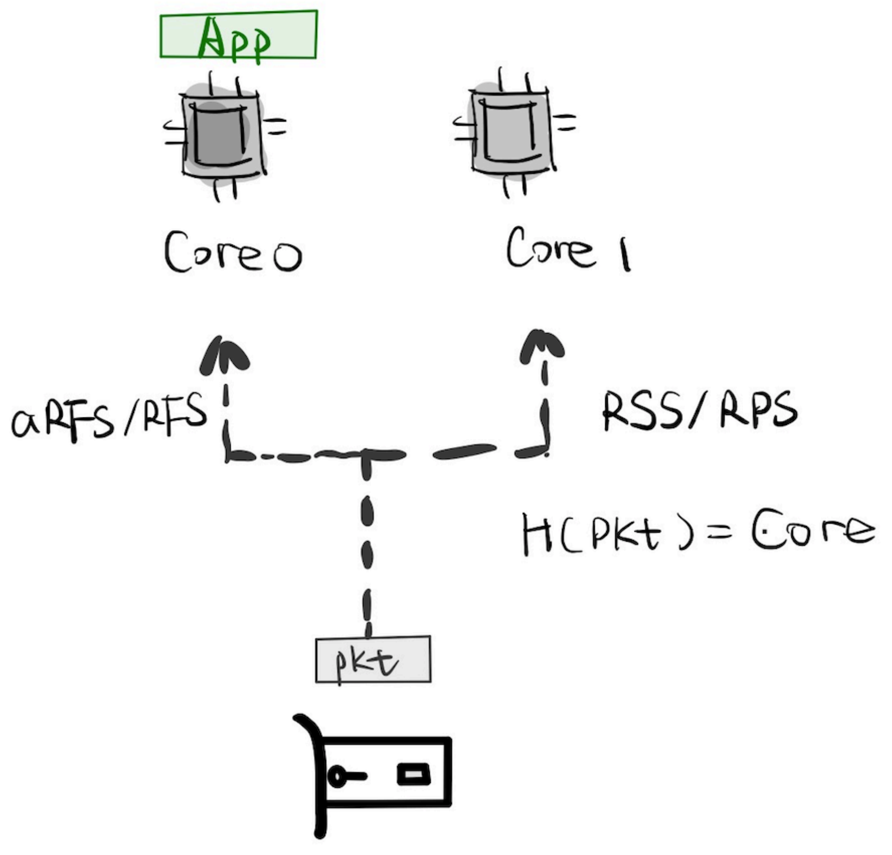
Poll for RX Packets.

# Packet and flow steering

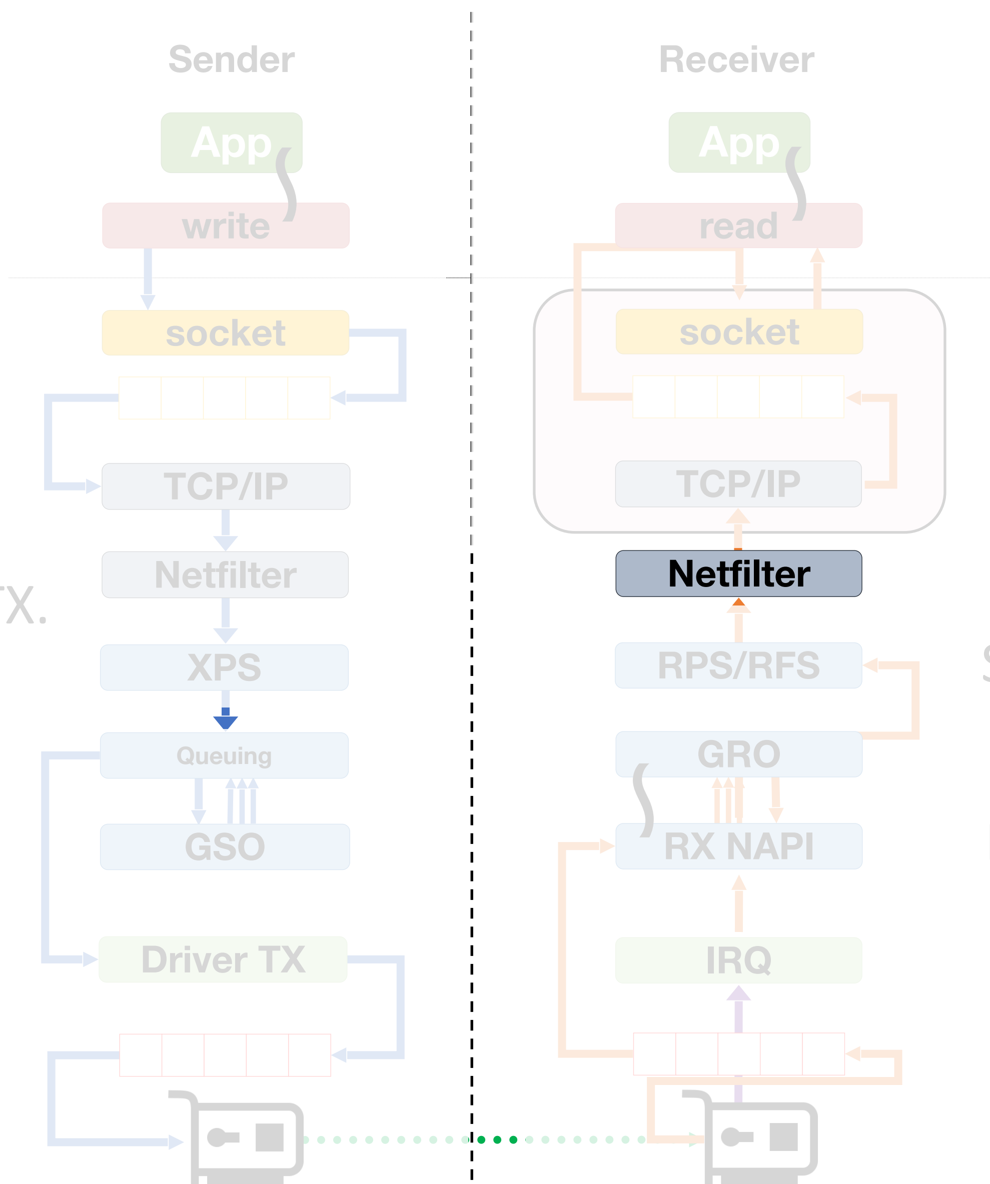
- **Which CPU core should NIC forward packets to?**

- **RPS/RSS: Choose CPU core based on hash of packet header**
  - RPS: software-based, RSS: hardware-based
  - Enables scalability via parallelized packet processing
  - Downside: Cache and NUMA issues

- **RFS/aRFS: Choose CPU core based on where the application is running**
  - RFS: software-based, aRFS: hardware-based
  - Benefits: Local cache/memory locality
  - Downside: Poor scalability when # of apps running on same core increases



# Network Stack Data Path




Select the Hardware Queue for TX.

Packet Scheduling.

Wake-up Application Thread.

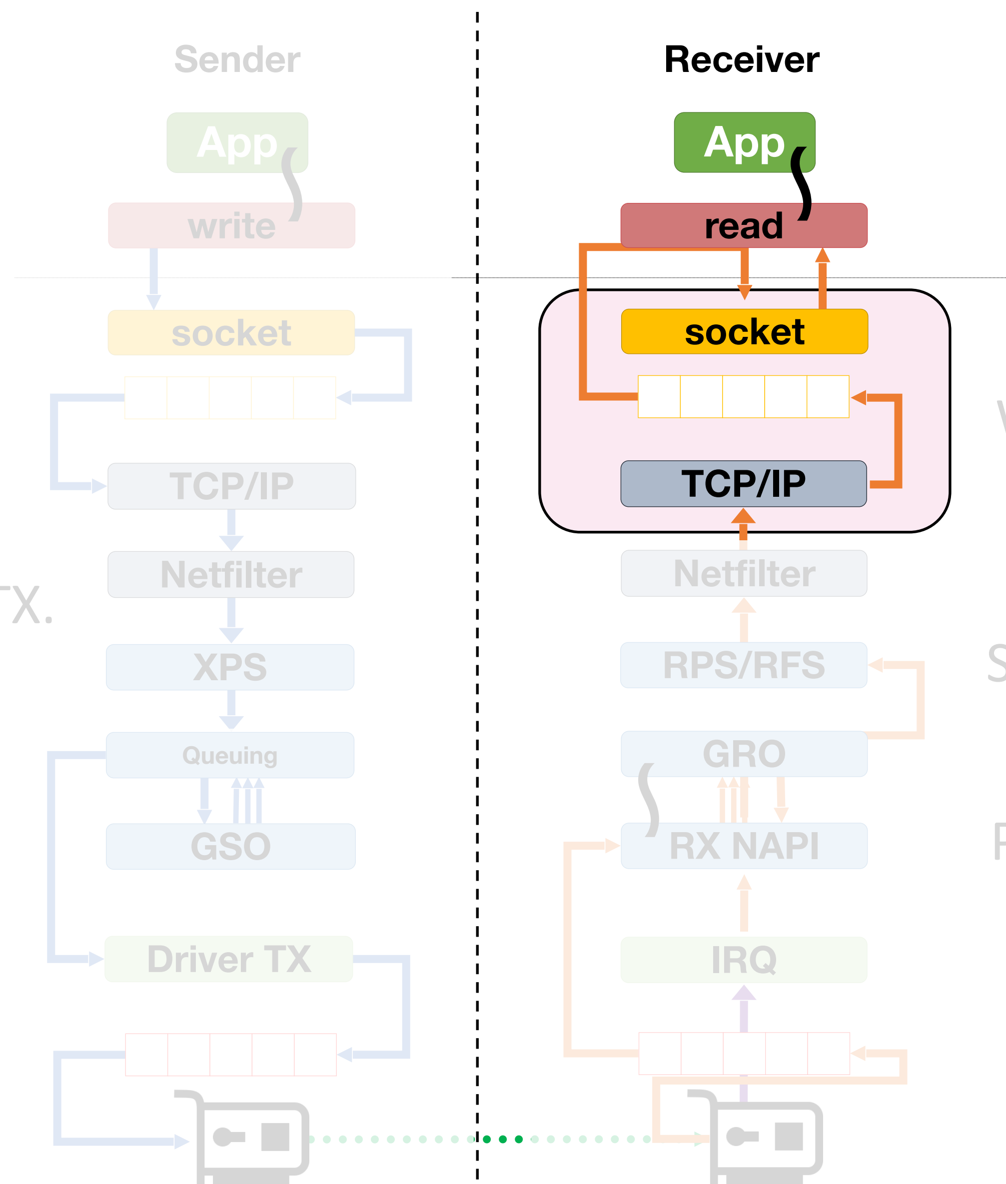
Select the CPU for TCP/IP Processing.

Poll for RX Packets.

# Network Stack Data Path

Select the Hardware Queue for TX.

Packet Scheduling.




Wake-up Application Thread.

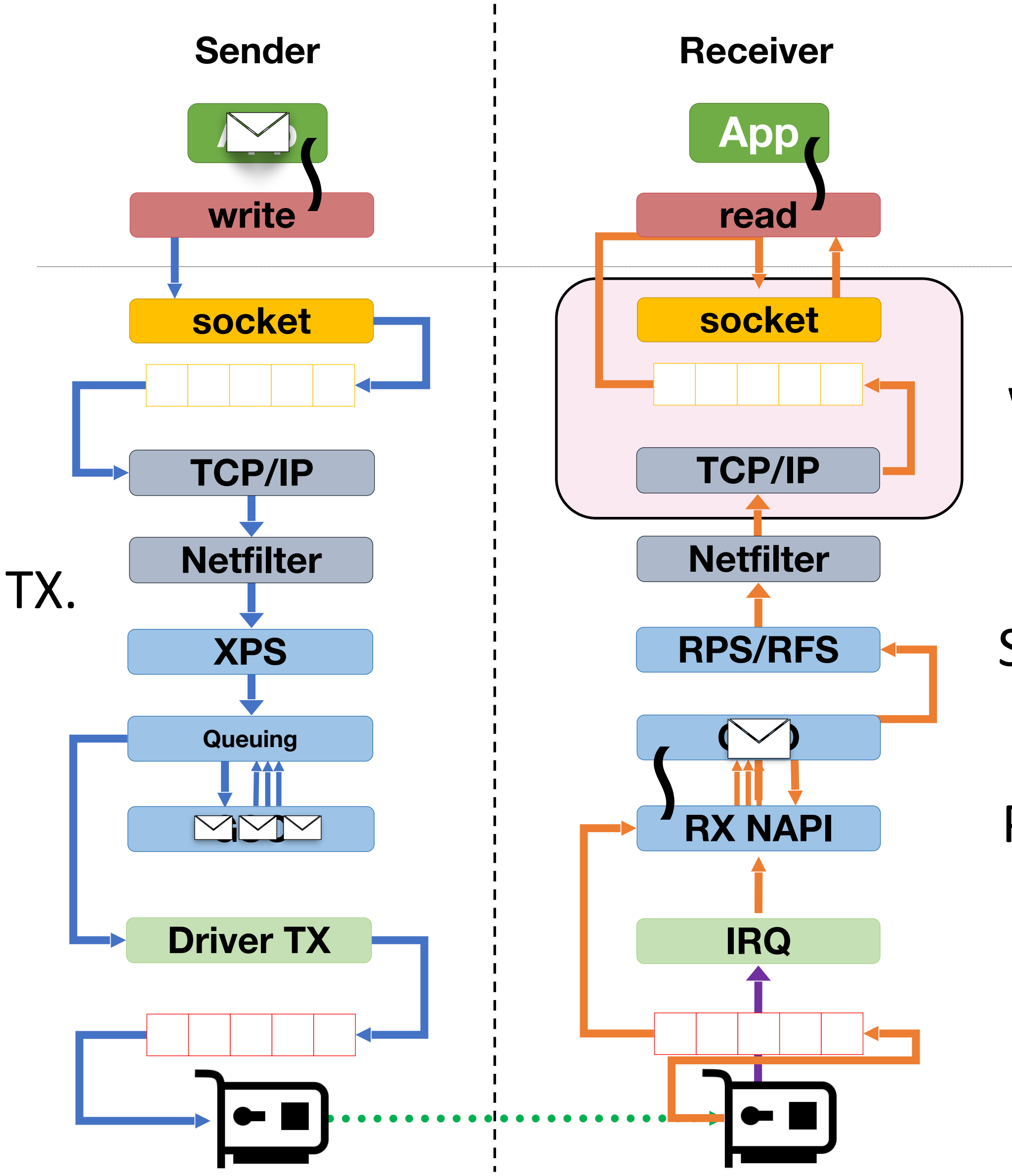
Select the CPU for TCP/IP Processing.

Poll for RX Packets.

# TCP/IP and read system call

- **Push packets to socket read queue**
- **Generate and send Acknowledgements (ACKs)**
  - Sender can clear out packets that have been delivered
- **Wake up application thread for copying data to application buffers**
  - Extra CPU scheduling overhead/delay
  - Once woken up, data is copied from OS buffers to application buffers

# End-to-end 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.

# Questions?