(Linux) Networking

Nima Honarmand

Network Layer Diagrams

- OSI and TCP/IP Stacks (From *Understanding Linux Network Internals*)

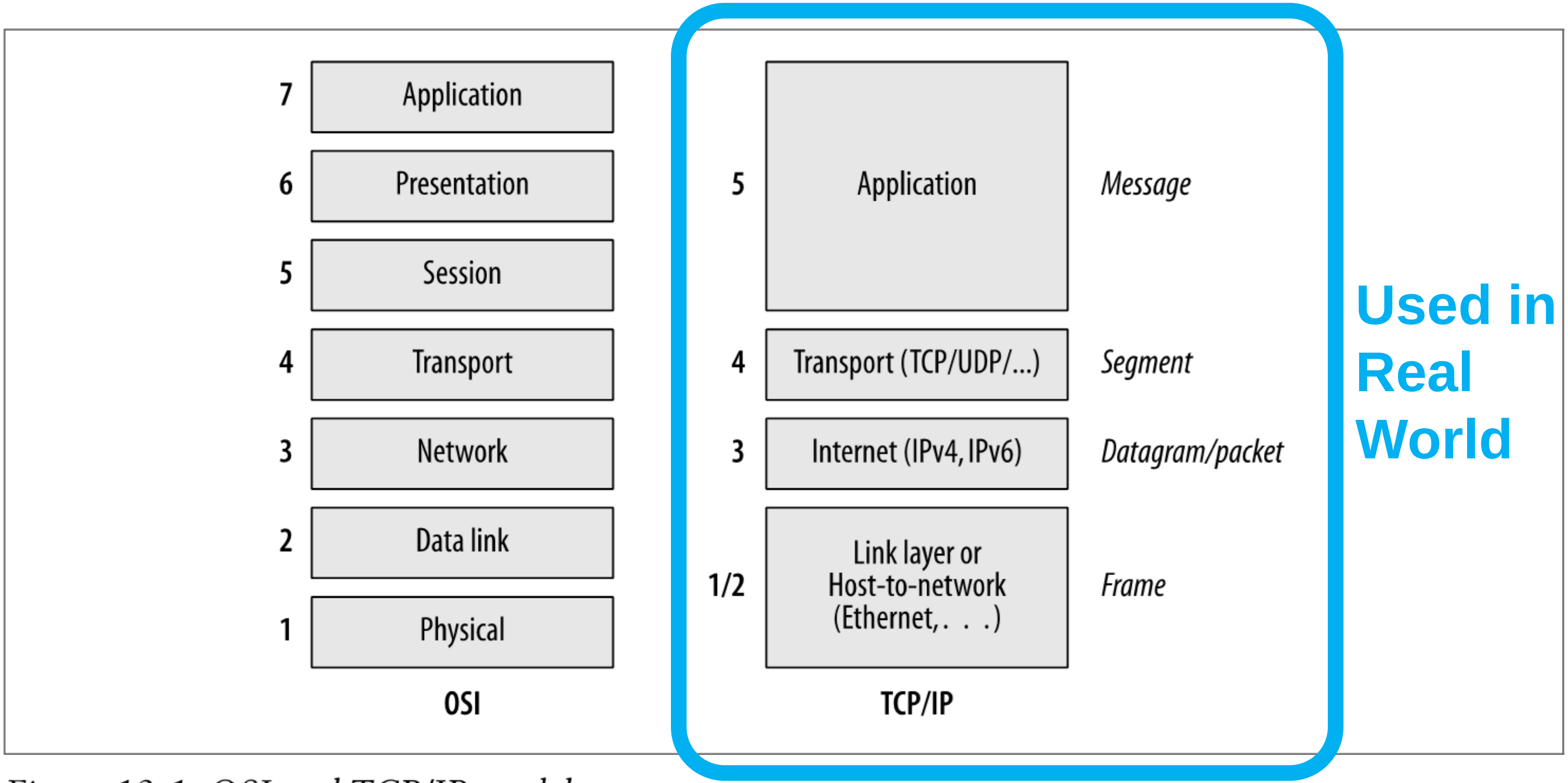


Figure 13-1. OSI and TCP/IP models

Ethernet (IEEE 802.3)

- LAN (Local Area Network) connection
- Simple packet layout:
  - Header
    - Type (e.g., IPv4)
    - source MAC address
    - destination MAC address
    - length (up to 1500 bytes)
    - …
  - Data block (payload)
  - Checksum
- Higher-level protocols “wrapped” inside payload
- “Unreliable” – no guarantee packet will be delivered

Internet Protocol (IP)

- 2 flavors: Version 4 and 6
  - Version 4 widely used in practice
  - Version 6 should be used in practice – but isn't
    - Public IPv4 address space is practically exhausted (see arin.net)
- Provides a network-wide unique address (IP address)
  - Along with netmask
  - Netmask determines if IP is on local LAN or not
- If destination not on local LAN
  - Packet sent to LAN's ***gateway***
  - At each gateway, payload sent to next hop

Address Resolution Protocol (ARP)

- IPs are logical (set in OS with ***ifconfig*** or ***ipconfig***)
- OS needs to know where (physically) to send packet
  - And switch needs to know which port to send it to
- Each NIC has a MAC (Media Access Control) address
  - "physical" address of the NIC
- OS needs to translate IP to MAC to send
  - Broadcast "who has 10.22.17.20" on the LAN
  - Whoever responds is the physical location
    - Machines can cheat (spoof) addresses by responding
  - ARP responses cached to avoid lookup for each packet

User Datagram Protocol (UDP)

- Applications on a host are assigned a port number
  - A simple integer
  - Multiplexes many applications on one device
  - Ports below 1k reserved for privileged applications
- Simple protocol for communication
  - Send packet, receive packet
  - No association between packets in underlying protocol
    - Application is responsible for dealing with…
      - Packet ordering
      - Lost packets
      - Corruption of content
      - Flow control
      - Congestion

Transmission Control Protocol (TCP)

- Same port abstraction (1-64k)
  - But different ports
  - i.e., TCP port 22 isn't the same port as UDP port 22
- Higher-level protocol providing end-to-end reliability
  - Transparent to applications
  - Lots of features
    - packet acks, sequence numbers, automatic retry, etc.
  - Pretty complicated

Web Request Example

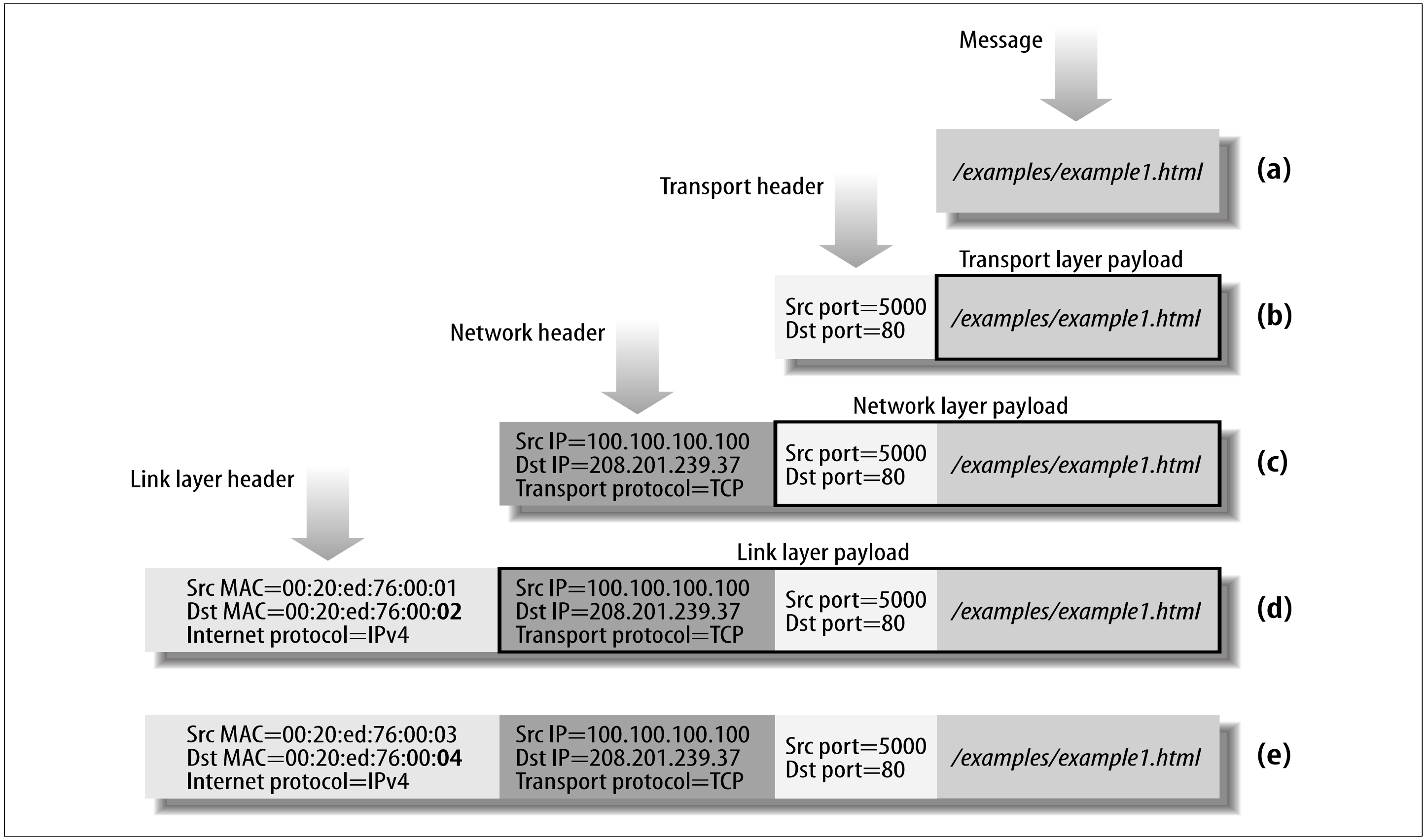


Figure 13-4. Headers compiled by layers: (a…d) on Host X as we travel down the stack; (e) on Router RT1

Source: Understanding Linux Network Internals

User-Level Networking APIs

- Programmers rarely create Ethernet frames
  - Or IP or TCP packets
- Most applications use the ***socket*** abstraction
  - Stream of messages or bytes between two applications
  - Applications specify protocol (TCP or UDP), remote IP address and port number

POSIX interface

- `socket()`: create a socket; returns associated file descriptor
- `bind()`/`listen()`/`accept()`: wait for connection (*server*)
- `connect()`: connect to remote end (*client*)
- `send()`/`recv()`: send and receive data
  - All headers are added/stripped by OS

Linux Implementation

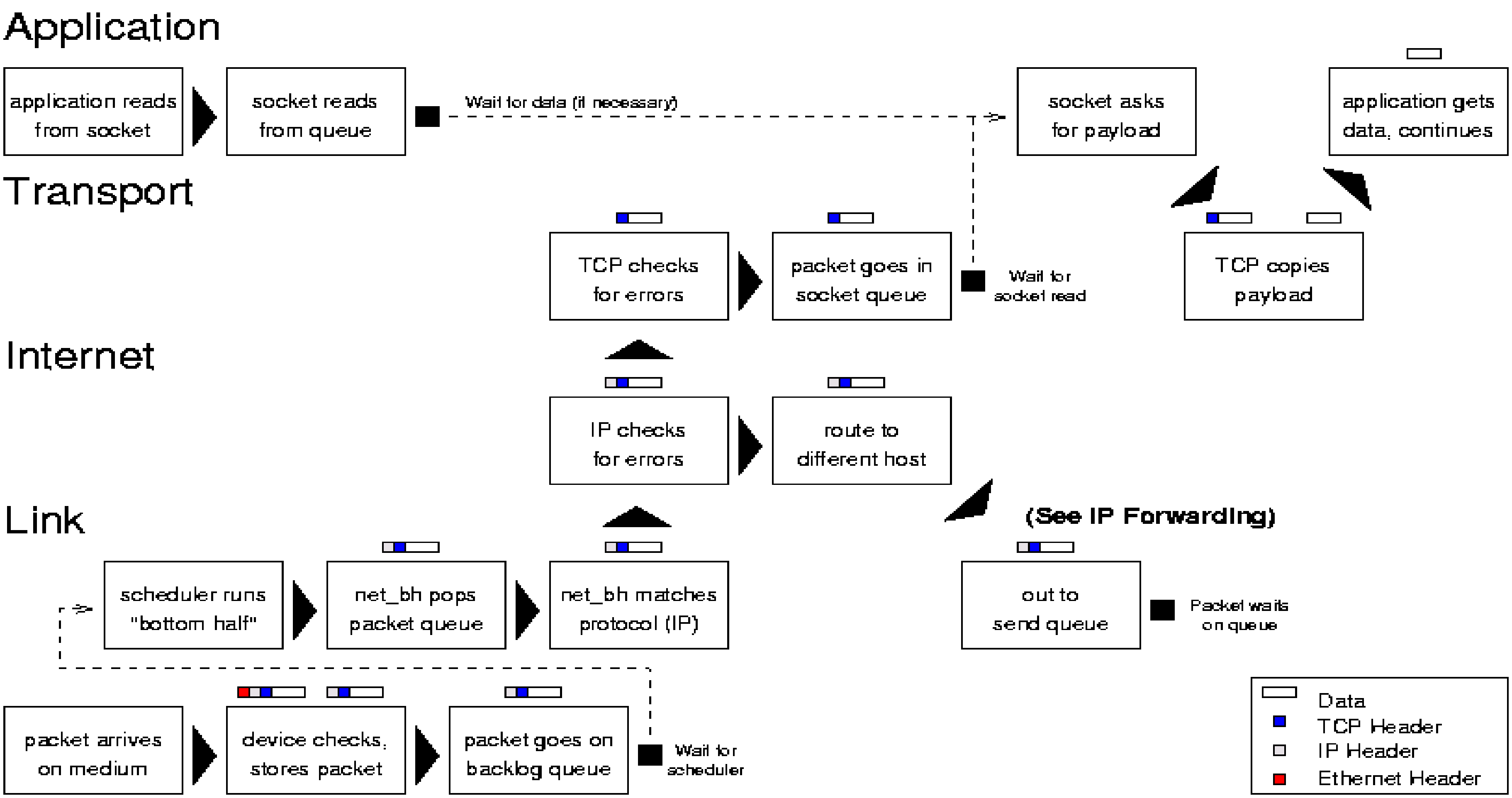
- Sockets implemented in the kernel
  - So are TCP, UDP, and IP and all other protocols
- Benefits:
  - Application not involved in TCP ACKs, retransmit, etc.
    - If TCP is implemented in library, app wakes up for timers
  - Kernel trusted with correct delivery of packets

Networking Services in Linux

- In addition to the socket interface and TCP/IP handling, the kernel provides a ton of other services
  - Address resolution
  - Bridging (Layer-2 switching)
  - Loopback and virtual network devices
  - Routing (L3 switching)
  - Firewall and filtering
  - Packet sniffing
  - …
- <u>Here, we only focus on general packet processing for application send and receives</u>

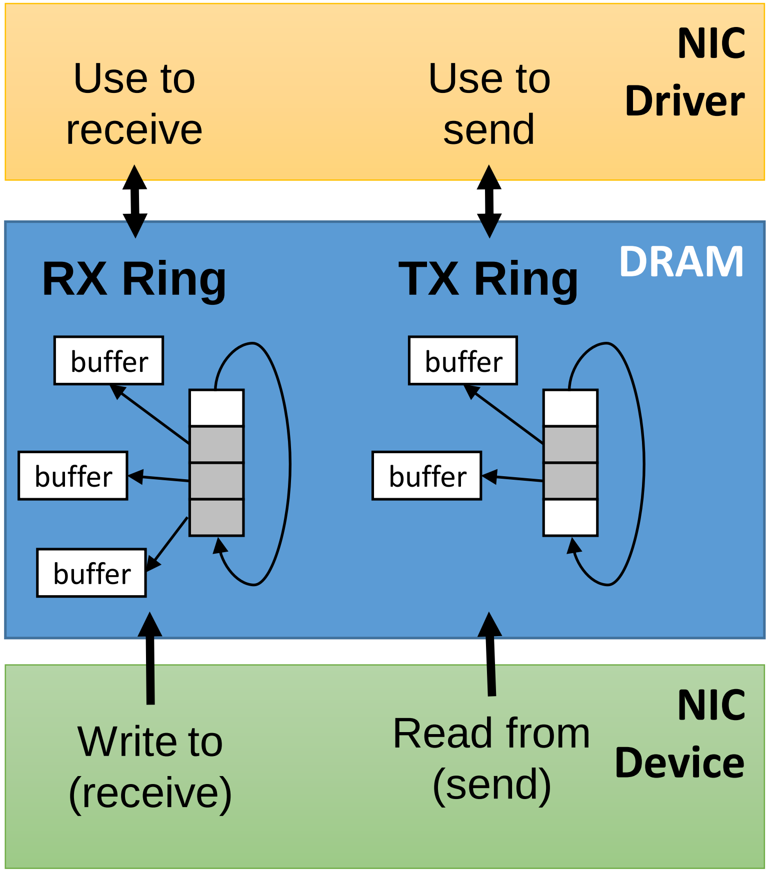
(Part of) Received Packet Processing

Source: *http://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html*



NIC Interface: Ring Buffers (1)

- High performance devices (such as NICs) use pre-allocated FIFOs of descriptors as device interface
  - E.g., network cards use send (TX) and receive (RX) rings
- Each descriptor in the queue usually points to a "buffer" where NIC should read data from (for send) or written data to (for recv)



NIC Interface: Ring Buffers (2)

- Both rings and buffers allocated in DRAM by driver
  - Device uses DMA to access descriptors and buffers
- Ring structured like a circular FIFO queue
  - Device has registers for ring *base*, *end*, *head* and *tail*
    - **Head**: the first HW-owned (ready-to-consume) DMA buffer
    - **Tail**: location after the last HW-owned DMA buffer
  - Device advances head pointer to get the next valid buffer
  - Driver advances tail pointer to add a valid buffer
- No dynamic buffer allocation or device stalls if ring is well-sized to the load
  - Trade-off between device stalls (or dropped packets) & memory overheads

NIC Interface: Interrupts & Doorbells (1)

- Ring buffers used for both sending and receiving
- **Receive**: <u>device</u> copies data into next empty buffer in RX ring and advances <u>head</u> pointer
- How would driver know about the new buffer?
  - Option 1: driver **polls** head pointer to see if changed
  - Option 2: Device sends an **interrupt**
- How would device know when there is a new empty buffer?
  - When the driver writes to RX <u>tail</u> register
  - Sometimes, referred to as *ringing the **doorbell***

NIC Interface: Interrupts & Doorbells (2)

- **Send**: driver prepares a full buffer & appends it to the TX ring tail
- How would device know about the new buffer?
  - When the driver writes to TX tail register
  - Again, a doorbell operation
- How would driver know there is room for new buffers in the ring?
  - Same options as before: driver polling or device interrupting

Handling Interrupts

- Recall: interrupts disabled while in interrupt handler
  → Need to avoid spending much time in there
- But processing received packets can take a long time
- Solution: split interrupt processing into two steps
  - ***Top half***: acknowledge interrupt, queue work somewhere
  - ***Bottom half***: take work from queue and do it
- Only top half needs to run with interrupts disabled
- NOTE: This is a general interrupt processing scheme for all devices, not just for network

Top and Bottom Halves

- "Top half":
  - acknowledges device interrupt by writing to a special register
  - sets a flag in kernel memory to <u>activate</u> the corresponding bottom half
- "Bottom half" does the actual processing of the device interrupt
- Terminology: ***Hard-*** vs. ***Soft-IRQ***
  - A hard-IRQ is the hardware interrupt line (triggers the top half handler from IDT)
  - Soft-IRQ is the actual interrupt handling code (bottom half)

Linux Implementation

- There is a per-cpu bitmask of pending Soft-IRQs
  - One bit per Soft-IRQ
    - e.g., NET_RX_SOFTIRQ and NET_TX_SOFTIRQ for network
  - There is a function associated with each Soft-IRQ
- Hard IRQ service routine sets the bit in the bitmask
  - bit can also be set by other code in kernel including Soft IRQ code itself
- At the right time, the kernel checks the bitmask and calls the function for pending Soft-IRQs

Linux Implementation

- Right time: when about to return to usermode from exceptions/interrupts/syscalls
- Each CPU also has a kernel thread ***ksoftirqd<CPU#>***
  - Processes pending bottom halves for that CPU
  - ***ksoftirqd*** is nice +19: Lowest priority—only called when nothing else to do
- Only process a few (e.g., 10) packets before returning to user mode
  - To avoid delaying user-mode program indefinitely
  - Remaining packets will be processed when ksoftirqd runs

Benefits of Separate Halves

1) Minimizes time in an interrupt handler with interrupts disabled

2) Simplifies service routines (defer complicated operations to a more general processing context)
   - E.g., what if you need to wait for a lock?
     - No Problem
   - or, be put to sleep until your `kmalloc()` succeeds?
     - No Problem

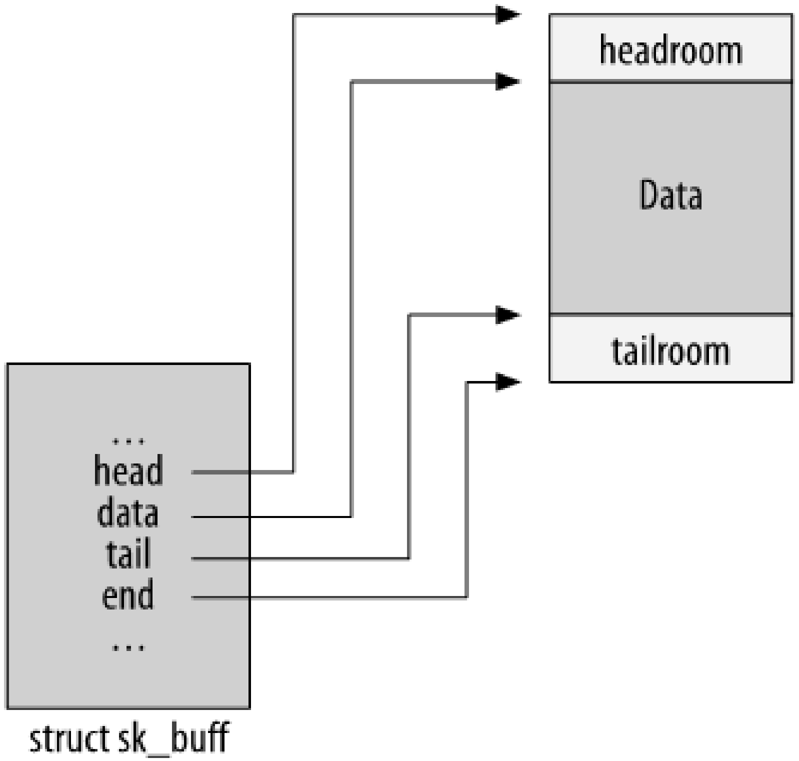
3) Gives kernel more scheduling flexibility
   - Can mix processing of device interrupts (using ksoftirqd) with application threads

Linux Plumbing

- Each message is put in a `sk_buff` structure
  - Passed through a stack of protocol handlers
  - Handlers update bookkeeping, wrap headers, etc.
- At the bottom are the device rings
  - Device sends/receives packets according to `sk_buff`s on its TX and RX rings

Efficient Packet Processing

- Receive side:
  Moving pointers is better than removing headers
- Send side:
  Prepending headers is more efficient than re-copy



head/end vs. data/tail pointers in sk_buff

Source: Understanding Linux Network Internals

Back to Receive: Bottom Half

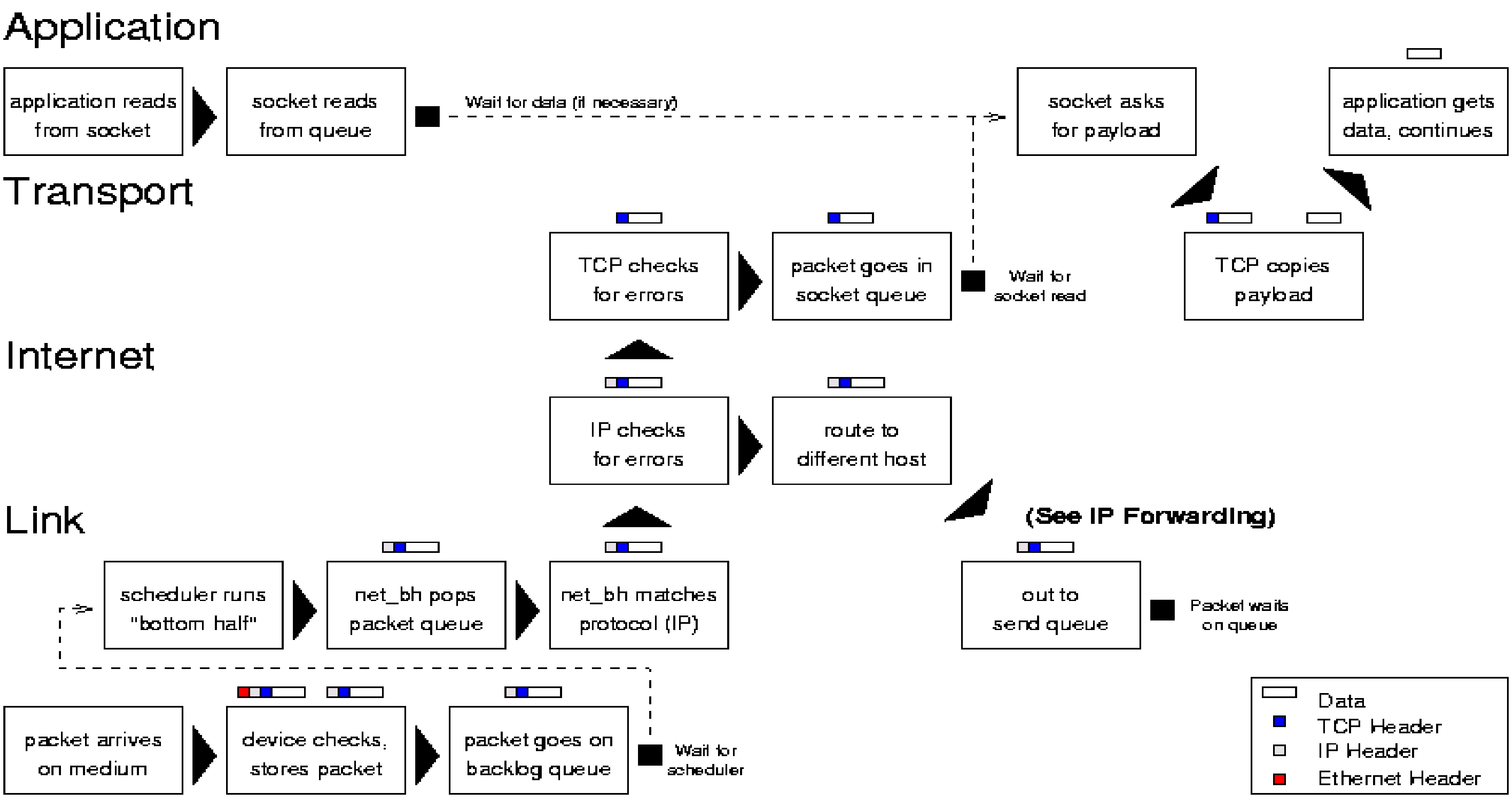
- For each pending `sk_buff`:
  - Pass a copy to any taps (sniffers)
  - Do any MAC-layer processing, like bridging
  - Pass a copy to the appropriate protocol handler (e.g., IP)
    - Recur on protocol handlers until you get to a port number
      - Perform some handling transparently (filtering, ACK, retry)
    - If good, deliver to associated socket
    - If bad, drop

Socket Delivery

- Once bottom half moves payload into a socket:
  - Check to see if a task is blocked on input for this socket
  - If yes, copy data to awaken the thread
- Once awoken, `recv()` reads data from socket buffer and copies to user-mode buffer and returns to user mode

Revisiting Received Packet Processing

Source: *http://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html*



Socket Sending

- `send()` copies data into socket
  - Allocate `sk_buff` for data
  - Be sure to leave plenty of head and tail room!
- System call handles protocol in application's timeslice
- Last protocol handler enqueues packet for transmit
  - If there is space in the TX ring
- Interrupt usually signals completion
  - Bottom half has very little to do
  - Usually, just add pending packets to the TX ring if previously full

Receive Livelock

- What happens when packets arrive at a very high frequency?
  - You spend all of your time handling interrupts!
- Receive Livelock: Condition when system never makes progress
  - Because spends all of its time starting to process new packets
  - Bottom halves never execute
    - Hard to prioritize other work over interrupts
- Better process one packet to completion than to run just the top half on a million

Receive Livelock in Practice

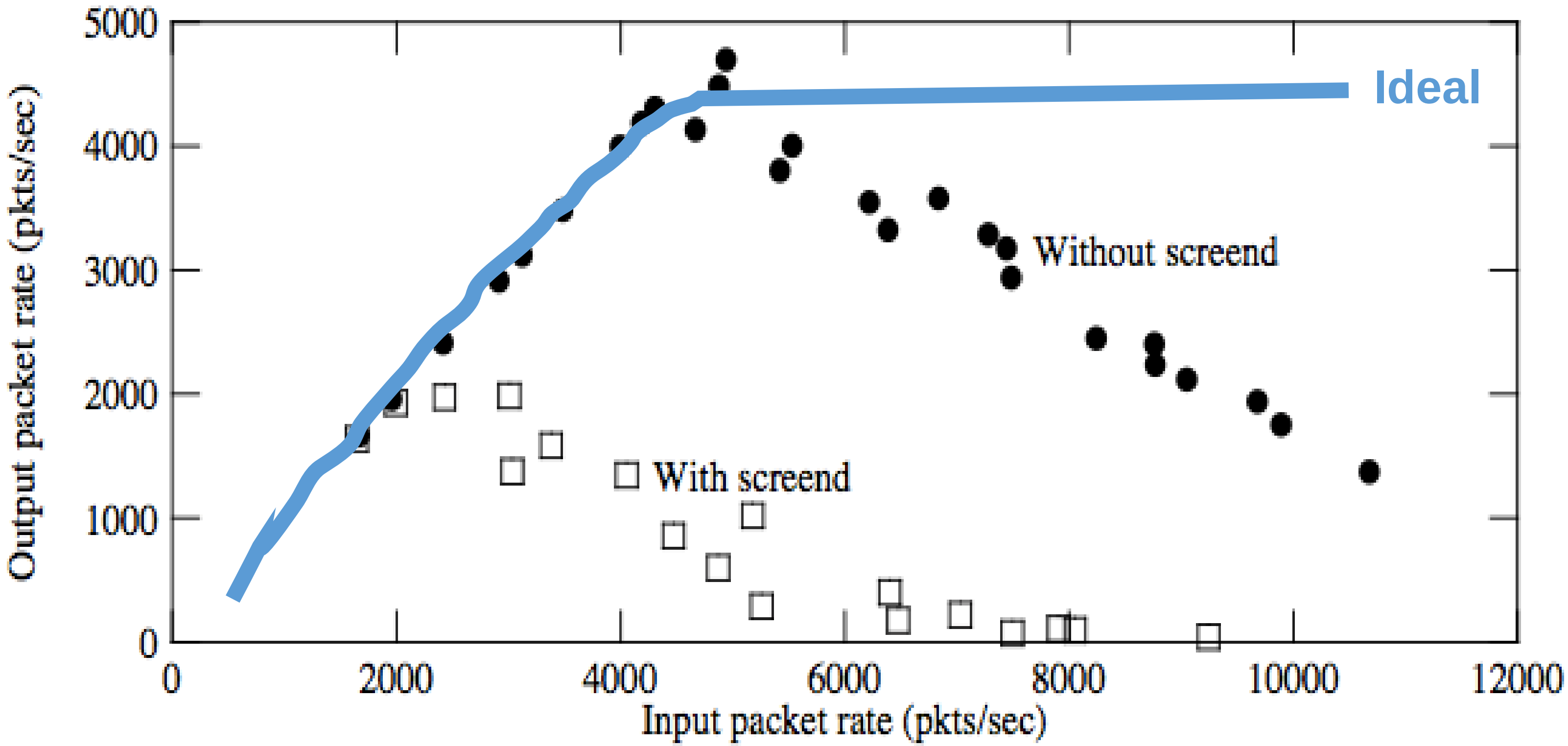


Fig. 2. Forwarding performance of unmodified kernel.

Source: Mogul & Ramakrishnan, ToCS, Aug 1997

Shedding Load

- If can't process all incoming packets, must drop some
- If going to drop some packets, better do it early!
  - Stop taking packets off of the network card
  - NIC will drop packets once its buffers get full on its own

Polling Instead of Interrupts

- Under heavy load, disable NIC interrupts
- Use polling instead
  - Ask if there is more work once you've done the first batch
- Allows packet go through bottom half processing
  - And the application, and then get a response back out
- Ensures some progress

Why not Poll All the Time?

- If polling is so great, why bother with interrupts?
- Latency
  - If incoming traffic is rare, want high-priority
    - Latency-sensitive applications get their data ASAP
    - Example: annoying to wait at ssh prompt after hitting a key

General Insight on Polling

- If the expected input rate is low
  - Interrupts are better
- When expected input rate is above threshold
  - Polling is better
- Need way to dynamically switch between methods

Livelock Only Relevant to Networks?

- Why don’t other devices (e.g., disks) have this problem?

1) For disk, if CPU is too busy processing previous disk requests, it can’t issue more

2) For network, external CPU can generate all sorts of network inputs

Linux NAPI (*New API*)

- Drivers provides `poll()` method for low-level receive
  - Passes packets received by the device to kernel
- Bottom half calls driver's `poll()` to get pending packets from the device
- Bottom half can disable the interrupt under heavy loads
  - Or uses a timer interrupt to schedule a poll (instead of per-packet interrupts)
  - Bonus: Some NICs have a built-in timer
    - Can fire an interrupt periodically, only if something to say!
- Gives kernel control to throttle network input
  - Under heavy-load, device will overwrite some buffers in the ring → Packets dropped in the device itself without involving the CPU
- Once load drops can enable per-packet interrupts back again

Conclusion

- Networking in OS a humongous piece of code
  - We just covered socket send/recv
- High performance devices (like NICs) use ring buffers as their interfaces
- Livelock is a real problem for NICs
  - Use combination of polling and interrupts
  - Use polling when there is heavy load
  - Once load drops, enable interrupts again