CSE 506: Operating Systems

Networking & NFS

4 to 7 layer diagram

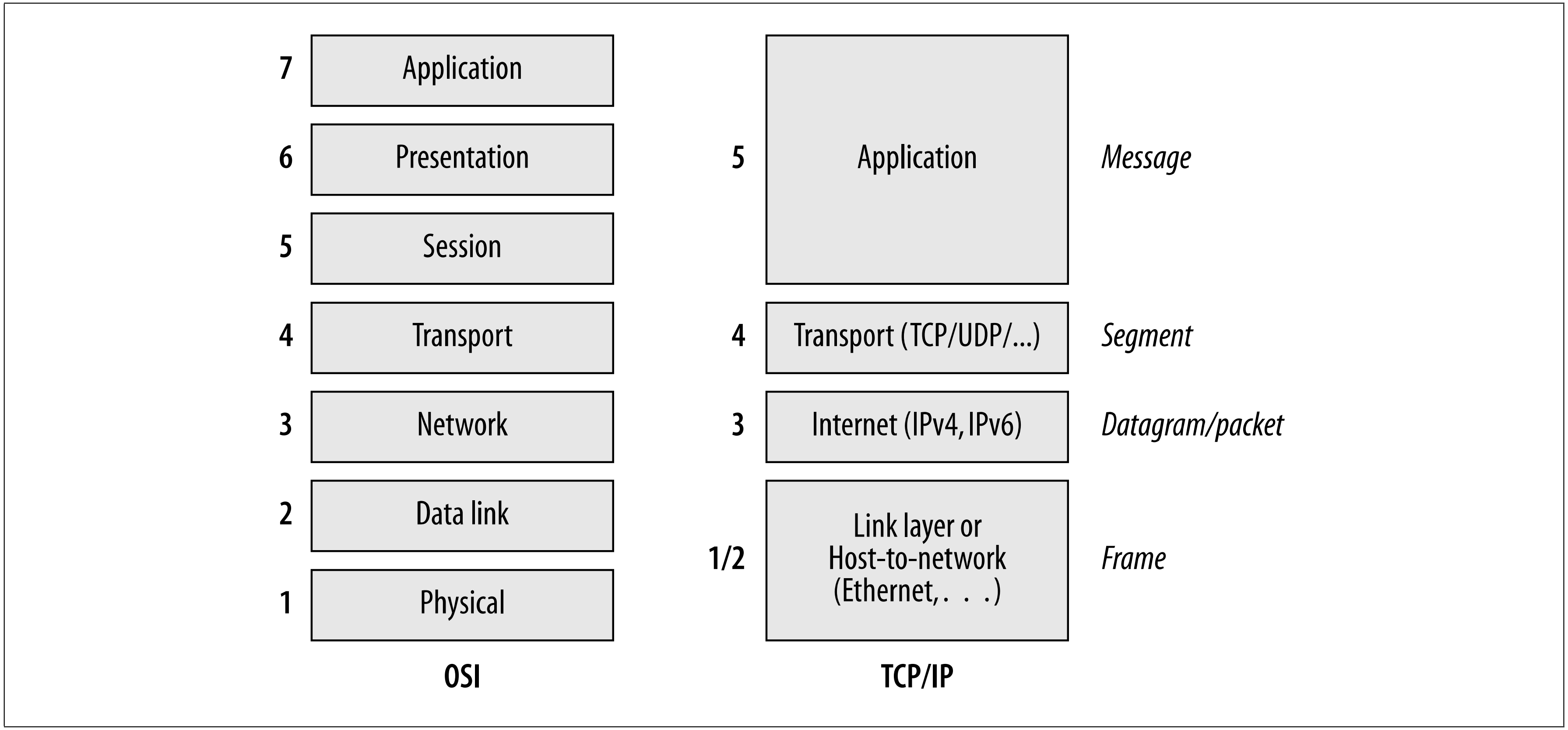


Figure 13-1. OSI and TCP/IP models

TCP/IP Reality

- The OSI model is great for undergrad courses
- TCP/IP (or UDP) is what the majority of world uses

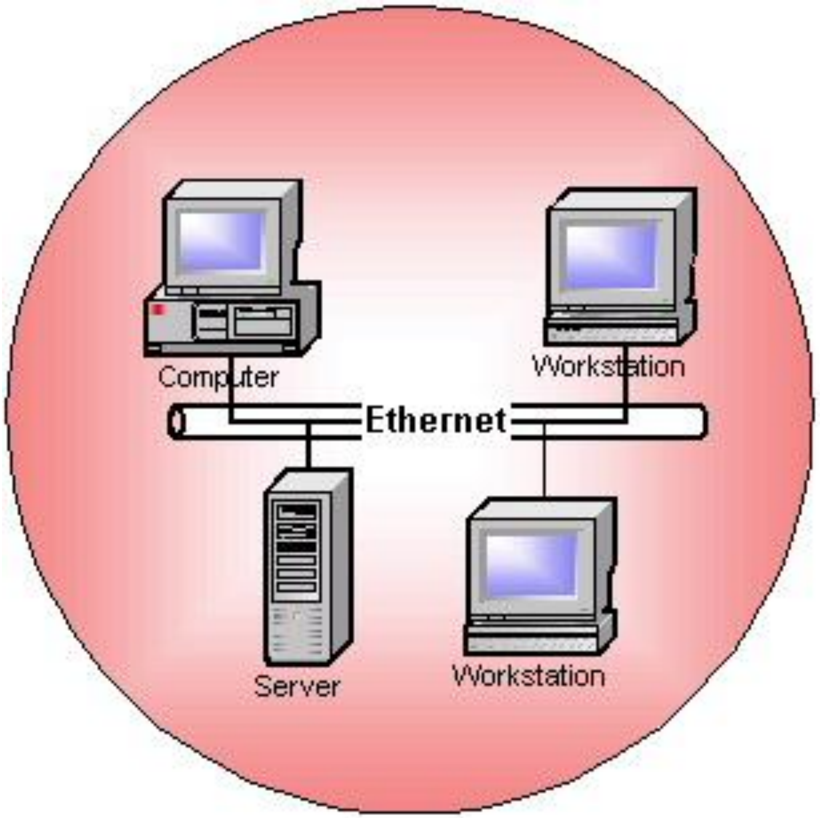
Ethernet (or 802.2 or 802.3)

- LAN (Local Area Network) connection
- Simple packet layout:
  - Header
    - type
    - source MAC address
    - destination MAC address
    - length (up to 1500 bytes regular, up to 9000 bytes “jumbo”)
    - …
  - Data block (payload)
  - Checksum
- Higher-level protocols “nested” inside payload
- “Unreliable” – no guarantee packet will be delivered

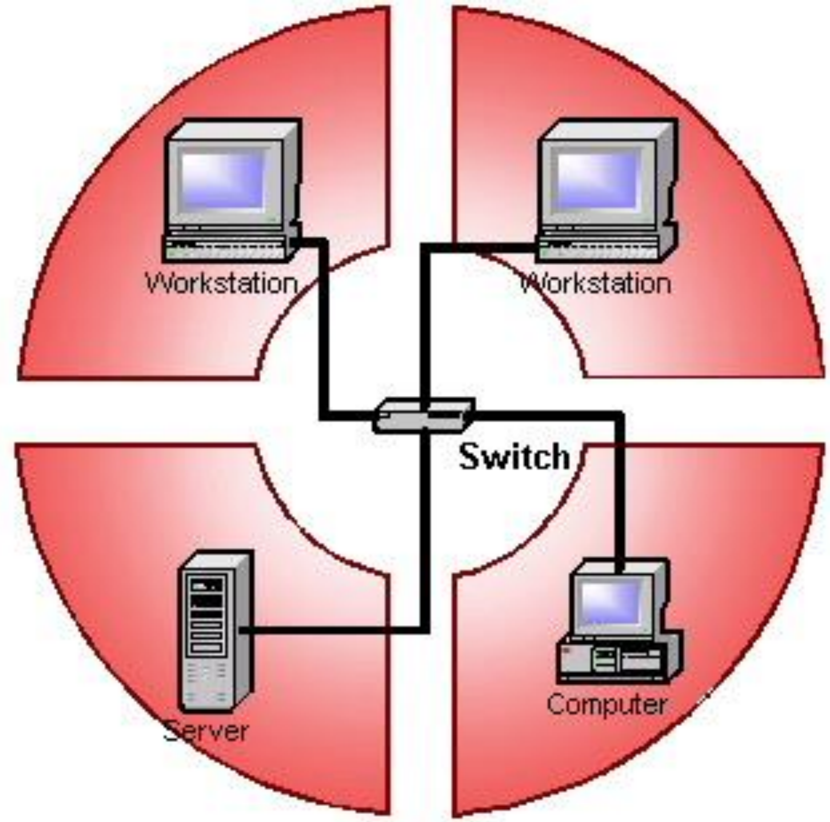
Ethernet Details

- Each device listens to all traffic
  - Hardware filters out traffic intended for other hosts
    - i.e., different destination MAC address
  - Can be put in "promiscuous" mode
    - Accept everything, even if destination MAC is not own
- If multiple devices talk at the same time
  - Hardware automatically retries after a random delay

Shared vs Switched



Shared Ethernet: : 1 collision domain for multiple nodes. The possibility of collisions. Non-deterministic



Switched Full Duplex Ethernet: 1 collision domain per node. Use of switch. No possibility of collisions. Deterministic.

Switched Networks

- Modern Ethernets are point-to-point and switched
- What is a hub vs. a switch?
  - Both are boxes that link multiple computers together
  - Hubs broadcast to all plugged-in computers
    - Let NICs figure out what to pass to host
      - Promiscuous mode sees everyone's traffic
  - Switches track who is plugged in
    - Only send to expected recipient
      - Makes sniffing harder ☹

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

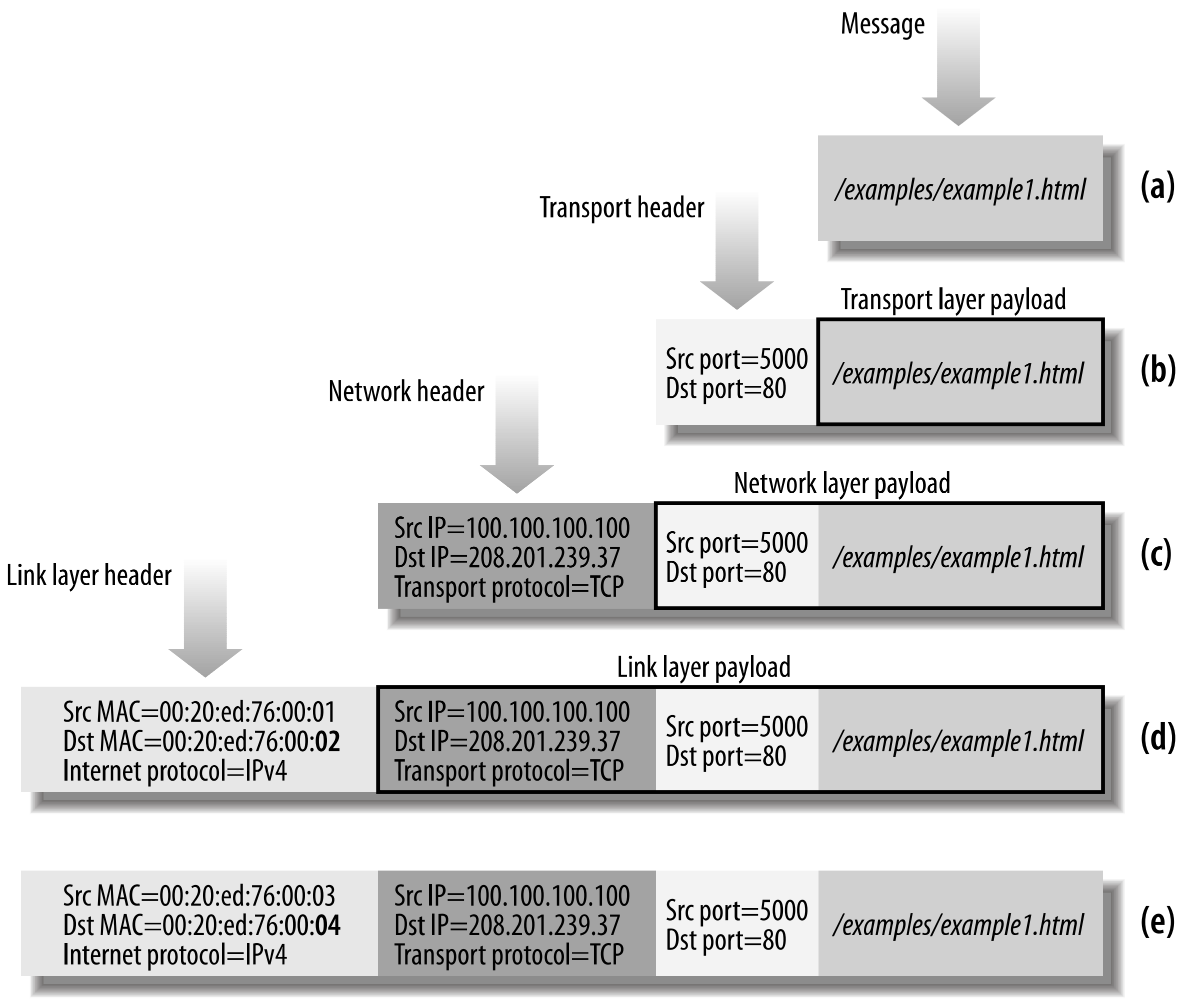
User Datagram Protocol (UDP)

- 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
- 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

Transmission Control Protocol (TCP)

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

Web Request Example



Networking APIs

- Programmers rarely create Ethernet frames
- Most applications use the socket abstraction
  - Stream of messages or bytes between two applications
  - Applications specify protocol (TCP or UDP), remote IP
- `bind()/listen()`: waits for incoming connection
- `connect()/accept()`: connect to remote end
- `send()/recv()`: send and receive data
  - All headers are added/stripped by OS

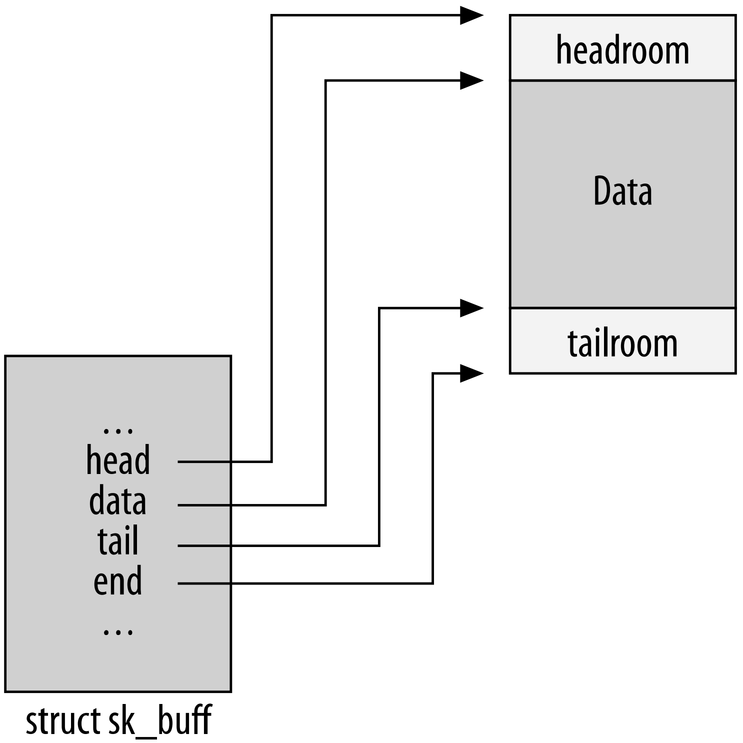
Linux implementation

- Sockets implemented in the kernel
  - So are TCP, UDP, and IP
- 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
- A single system call:
  - sys_socketcall(call, args)
    - Has a sub-table of calls, like bind, connect, etc.

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 is the device itself (e.g., NIC driver)
  - Sends/receives packets on the wire

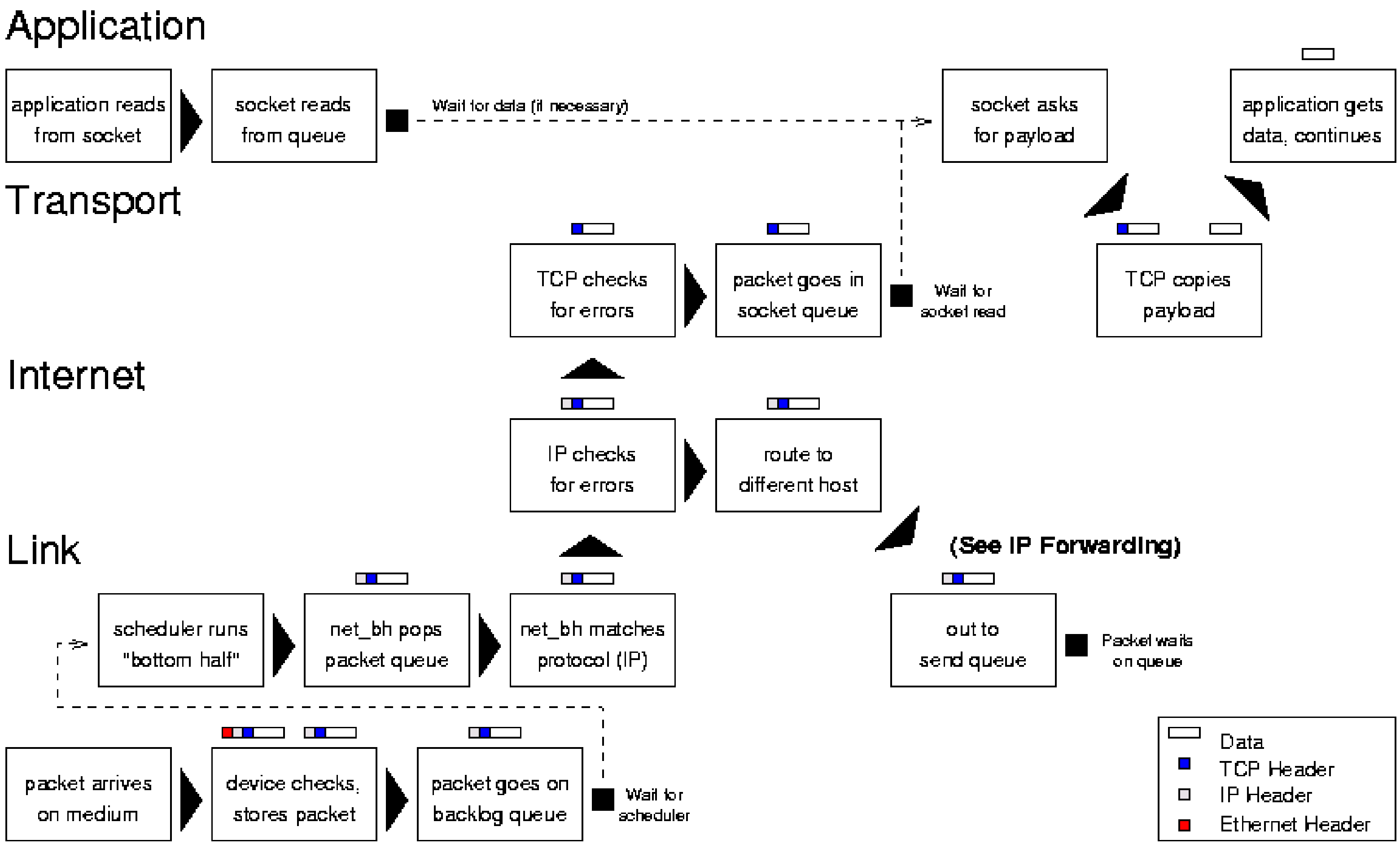
sk_buff



Efficient packet processing

- Moving pointers is better than removing headers
- Appending headers is more efficient than re-copy

Received Packet Processing



Source = http://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html#tth_sEc6.2

Interrupt Handler

- "Top half" responsible to:
  - Allocate/get a buffer (`sk_buff`)
  - Copy received data into the buffer
  - Initialize a few fields
  - Call "bottom half" handler
- In reality:
  - Systems allocate ring of `sk_buffs` and give to NIC
  - Just "take" the buff from the ring
    - No need to allocate (was done before)
    - No need to copy data into it (DMA already did it)

SoftIRQs

- A hardware IRQ is the hardware interrupt line
  - Use to trigger the “top half” handler from IDT
- SoftIRQ is the big/complicated software handler
  - Or, “bottom half”
- How are these implemented in Linux?
  - Two canonical ways: SoftIRQ and Tasklet
  - More general than just networking

SoftIRQs

- Kernel's view: per-CPU work lists
  - Tuples of <function, data>
- At the right time, call `function(data)`
  - Right time: Return from exceptions/interrupts/sys. calls
  - Each CPU also has a kernel thread ksoftirqd_CPU#
    - Processes pending requests
    - In case softirq can't handle them quickly enough

SoftIRQs

- Device programmer's view:
  - Only one instance of SoftIRQ will run on a CPU at a time
    - Doesn't need to be reentrant
      - If interrupted by HW interrupt, will not be called again
        - Guaranteed that invocation will be finished before start of next
  - One instance can run on each CPU concurrently
    - Must use spinlocks to avoid conflicting on data structures

Tasklets

- For the faint of heart (and faint of locking prowess)
- Constrained to only run one at a time on any CPU
  - Useful for poorly synchronized device drivers
    - Those that assume a single CPU in the 90’s
  - Downside: All bottom halves are serialized
    - Regardless of how many cores you have
    - Even if processing for different devices of the same type
      - e.g., multiple disks using the same driver

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 handler 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 task is blocked on input for this socket
    - If yes, wake it up corresponding process
- Read/recv system calls copy data into application

Socket sending

- Send/write system calls copy 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
  - Receive handling not counted toward app
- Last protocol handler enqueues packet for transmit

Receive livelock

- Condition when system never makes progress
  - Spends all time starting to process new packets
- 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

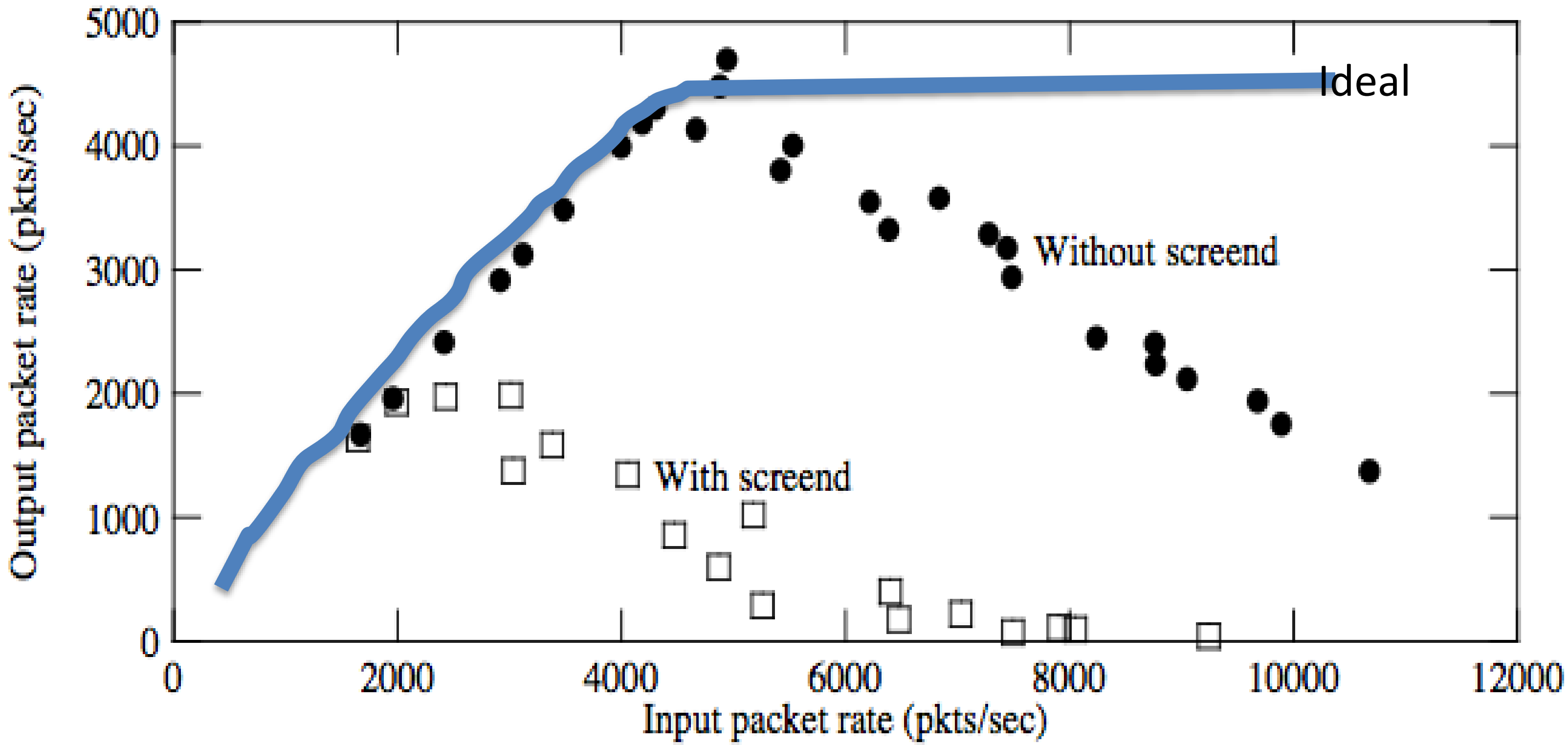


Fig. 2. Forwarding performance of unmodified kernel.

Source: Mogul & Ramakrishnan, ToCS 96

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

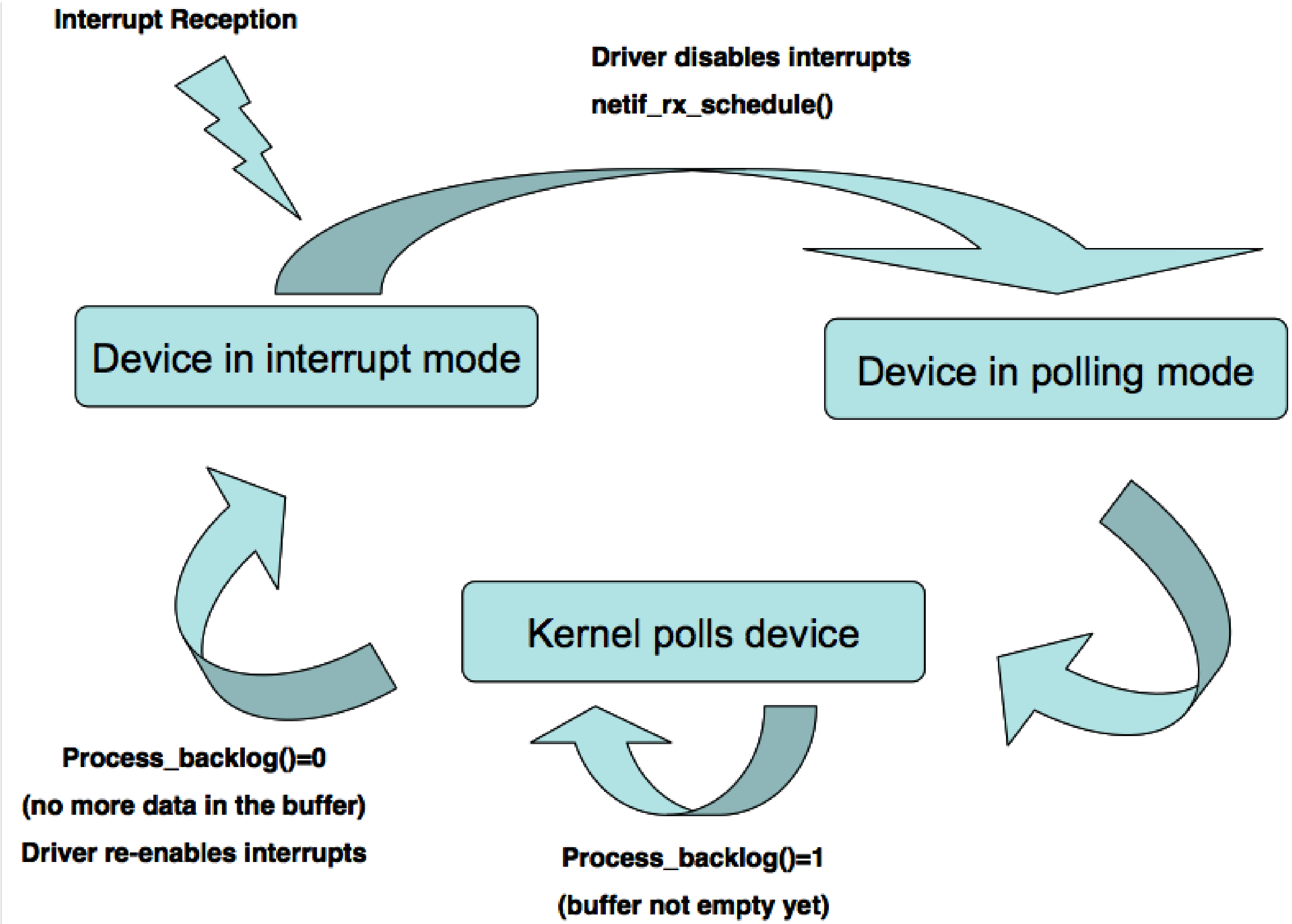
Why not poll all the time?

- 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
    - Ex.: 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

Pictorially



Source: *download.intel.com/design/intarch/**PAPERS**/323704.pdf*

Why is this only relevant to networks?

- Why don’t disks have this problem?
  - Inherently rate limited
- If CPU is too busy processing previous disk requests
  - It can’t issue more
- External CPU can generate all sorts of network inputs

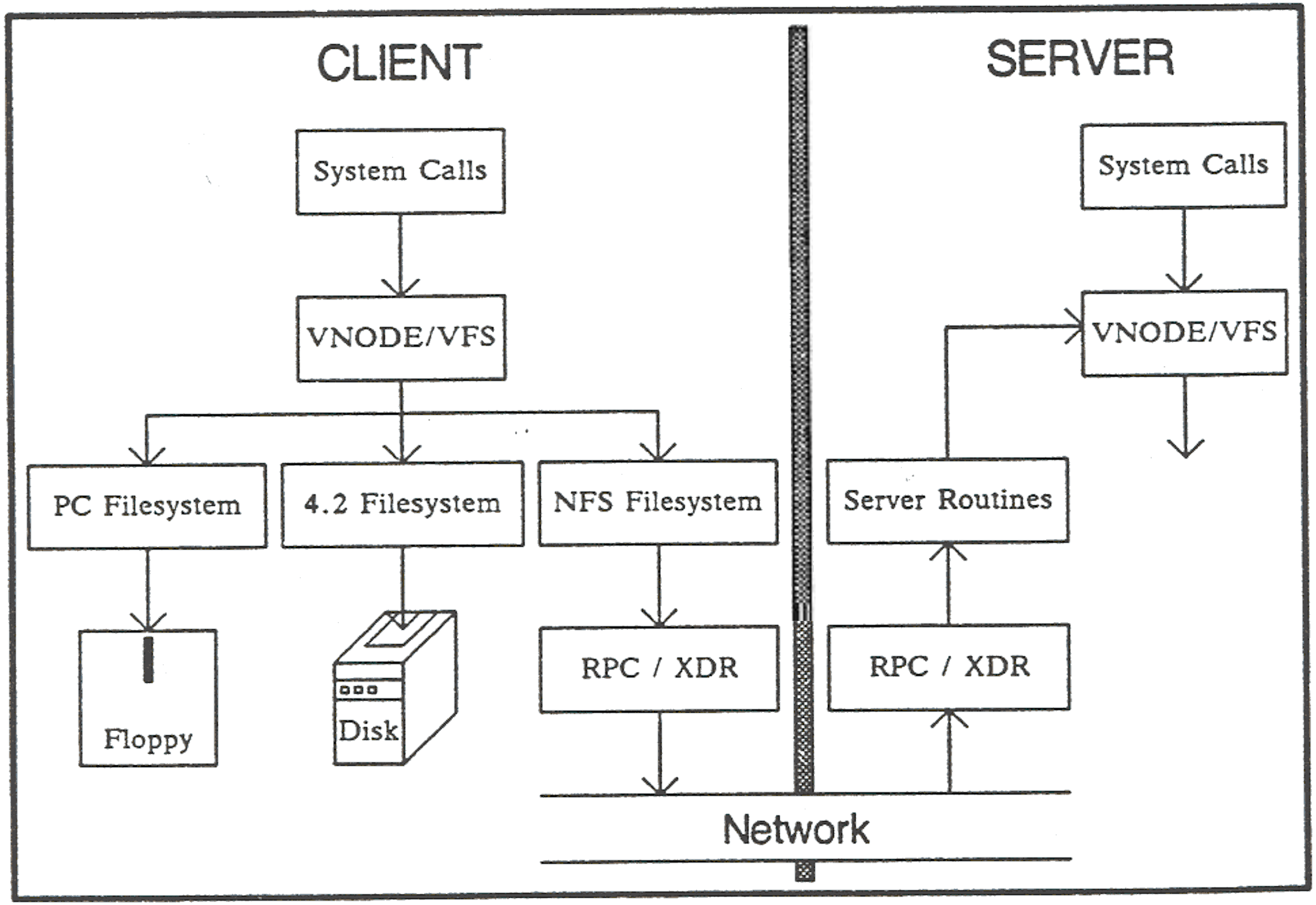
Linux NAPI

- ***"New API"***
- Drivers provides `poll()` for low-level receive
  - Called in first step of softirq RX function
- Top half schedules `poll()` to do receive as softirq
  - Can disable the interrupt under heavy loads
    - Use timer interrupt to schedule a poll
  - 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

Linux NAPI and Legacy Drivers

- Slow adoption – drivers need to be rewritten
- Backwards compatibility solution:
  - Old top half creates sk_buffs and puts them in a queue
  - Queue assigned to a fake “backlog” device
  - Backlog poll device is scheduled by NAPI softirq
  - Interrupts can still be disabled on NIC

NFS


CLIENT
SERVER
System Calls
VNODE/VFS
PC Filesystem
4.2 Filesystem
NFS Filesystem
Floppy
Disk
RPC / XDR
System Calls
VNODE/VFS
Server Routines
RPC / XDR
Network

Intuition

- Instead of translating VFS requests into disk accesses
  - Translate them into remote procedure calls to server
- Easy, right?

Challenges

- Server can crash or be disconnected
- Client can crash or be disconnected
- How to coordinate multiple clients on same file?
- Security

Disconnection

- Machine can crash between writes to the hard drive
  - Client can crash between writes to the server
- Server must recover if client fails between requests
  - Simple protocols (e.g., send block updates) won’t work
    - Client disconnects after marking block in use, before referencing it
  - When is it safe to reclaim the block?
    - What if, 3 months later, the client tries to use the block?

Stateful protocols

- Stateful protocols persist state across requests
  - Like the example on previous slide
- Server Challenges:
  - Knowing when a connection has failed (timeout)
  - Tracking state that needs to be cleaned up on a failure
- Client Challenges:
  - If server thinks we failed (timeout)
    - Must recreate server state to make progress

Stateless protocol

- The (potentially) simpler alternative:
  - All necessary state is sent with a single request
  - Server implementation much simpler!
- Downside:
  - May introduce more complicated messages
    - And more messages in general

NFS is stateless

- Every request sends all needed info
  - User credentials (for security checking)
  - File identifier and offset
- Each request matches VFS operation
  - e.g., write, delete, stat

Challenge: Lost request?

- Request sent to NFS server, no response received
  - Did the message get lost in the network (UDP)?
  - Did the server die?
  - Is the server slow?
    - Don’t want to do things twice
      - Bad idea: write data at the end of a file twice
- Idea: Make all requests idempotent
  - Requests have same effect when executed multiple times
    - Ex: write() has an explicit offset, same effect if done 2x

Challenge: Inode reuse

- Process A opens file ‘foo’
  - Maps to inode 30
- Process B unlinks file ‘foo’
  - On local system, OS holds reference to the inode
    - Blocks belonging to file ‘foo’ not reused
  - NFS is stateless, server doesn’t know about open handle
    - The file can be deleted and the inode reused
    - Next request for inode 30 will go to the wrong file
- Idea: Generation numbers
  - If inode in NFS is recycled, generation number is incremented
  - Client requests include an inode + generation number
    - Enables detecting attempts to access an old inode

Challenge: Security

- Local UID/GID passed as part of the call
  - UIDs must match across systems
  - Yellow pages (yp) service; evolved to NIS
  - Replaced with LDAP or Active Directory
- Root squashing: “root” (UID 0) mapped to “nobody”
  - Ineffective security
    - Can send any UID in the NFS packet
    - With root access on NFS client, “su” to another user to get UID

Challenge: File locking

- Must have way to change file without interference
  - Get a server-side lock
    - What happens if the client dies?
    - Lots of options (timeouts, etc), mostly bad
  - Punted to a separate, optional locking service
    - With ugly hacks and timeouts

Challenge: Removal of open files

- Unix allows accessing deleted files if still open
  - Reference in in-memory inode prevents cleanup
    - Applications expect this behavior
      - How to deal with it with NFS?
- On client, check if file is open before removing it
  - If yes, rename file instead of deleting it
    - `.nfs*` files in modern NFS
  - When file is closed, delete temp file
    - If client crashes, garbage file is left over ☹

Challenge: Time synchronization

- Each CPU's clock ticks at slightly different rates
  - These clocks can drift over time
- Tools like 'make' use timestamps
  - Clock drift can cause programs to misbehave

```
make[2]: warning: Clock skew detected.
Your build may be incomplete.
```

- Systems using NFS must have clocks synchronized
  - Usually with external protocol like NTP
    - Synchronization depends on unknown communication delay
      - Very complex protocol
      - Works pretty well in practice

Challenge: Caches and Consistency

- Clients A and B have file in their cache
- Client A writes to the file
  - Data stays in A's cache
  - Eventually flushed to the server
- Client B reads the file
  - Does B see the old contents or the new file contents?
    - Who tells B that the cache is stale?
      - Server can tell
        - » But only after A actually wrote/flushed the data

Consistency/Performance Tradeoff

- Performance: cache always, write when convenient
  - Other clients can see old data, or make conflicting updates
- Consistency: write everything immediately
  - And tell everyone who may have it cached
  - Much more network traffic, lower performance
  - Common case: accessing an unshared file

Close-to-Open Consistency

- NFS Model: Flush all writes on a close
- When opening file, get latest version on the server
  - Copy entire file from server into local cache
  - Odd behavior when multiple clients use the same file
    - Probably a reasonable compromise
  - What if the file is really big?
    - How big is “really big”?

NFS Evolution

- The simple protocol was version 2
- Version 3 (1995):
  - 64-bit file sizes and offsets (large file support)
  - Bundle attributes with other requests to eliminate stat()
  - Other optimizations
  - Still widely used today

NFS V4 (2000)

- Attempts to address many of the problems of v3
  - Security (eliminate homogeneous UID assumptions)
  - Performance
- Becomes a stateful prototocol
- pNFS –extensions for parallel distributed accesses
- Too advanced for its own good
  - Much more complicated then v3
    - Slow adoption
  - Barely being phased in now
    - With hacks that lose some of the features (looks more like v3)