ECE391- Computer System Engineering

Lecture 3.14

Networking

University of Illinois at Urbana-Champaign

Fall 2021

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Office of the Provost

Dear Colleagues,

We have shared several updates about Fall 2021 in the past several weeks as we monitor the progress with managing the pandemic and respond to new science-based COVID-19 guidance from the CDC, IDPH, CUPHD and our own SHIELD team. Below is a condensed version of what we need to know as we prepare for in-person classes in a few weeks.

Face Coverings

- Everyone (faculty, staff, students, visitors) is required to wear a face covering in university facilities. Read more about face coverings and face shields here.
- If a student is not wearing a face covering in your class, ask them to put one on. If they refuse, dismiss the class and report the student to the Office for Student Conflict Resolution for further discipline by filling out this form. Call UIPD, 217-333-1216, only if an individual becomes belligerent, disruptive and threatening.

Announcements

- MP3
  - Checkpoint 4: 5:59PM on Tuesday 11/16
    - CP4 Demo: **11/16 Tuesday, 6:30pm**: all teams
  - Checkpoint 5: 5:59PM on Sunday 12/5
    - Final Demo: 12/7 - 12/8 (Time TBD)
- No Class 12/08/2021 (Demos)
- Final Exam
  - ECE 391 AL 12/14/2021 T 1:30 PM 4:30 PM 1002 Electrical & Computer Eng Bldg
  - ECE 391 ALO 12/14/2021 T 1:30 PM 4:30 PM Exam Online
  - Review Session in class 12/02/2021

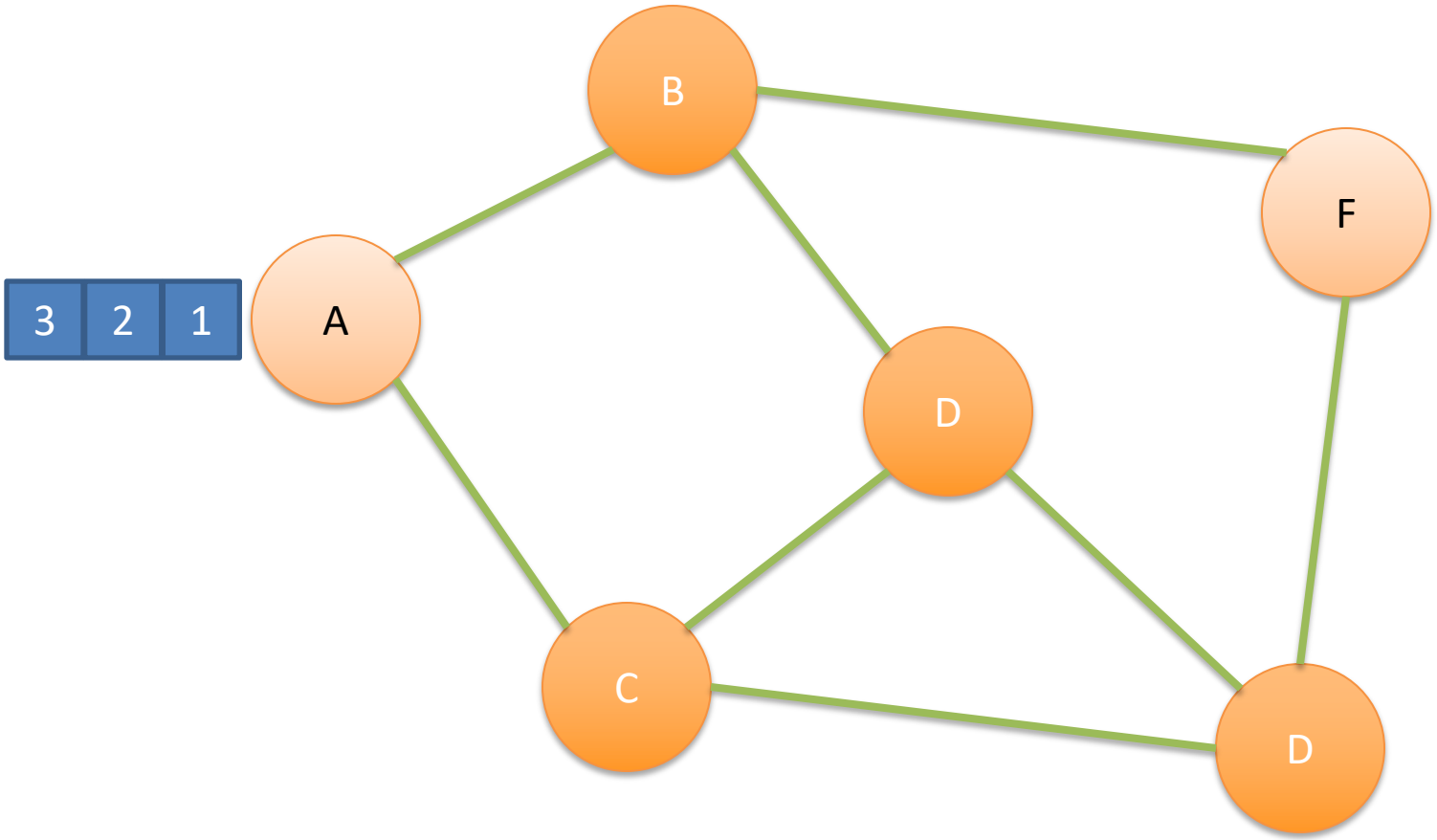
Networking

BACKGROUND

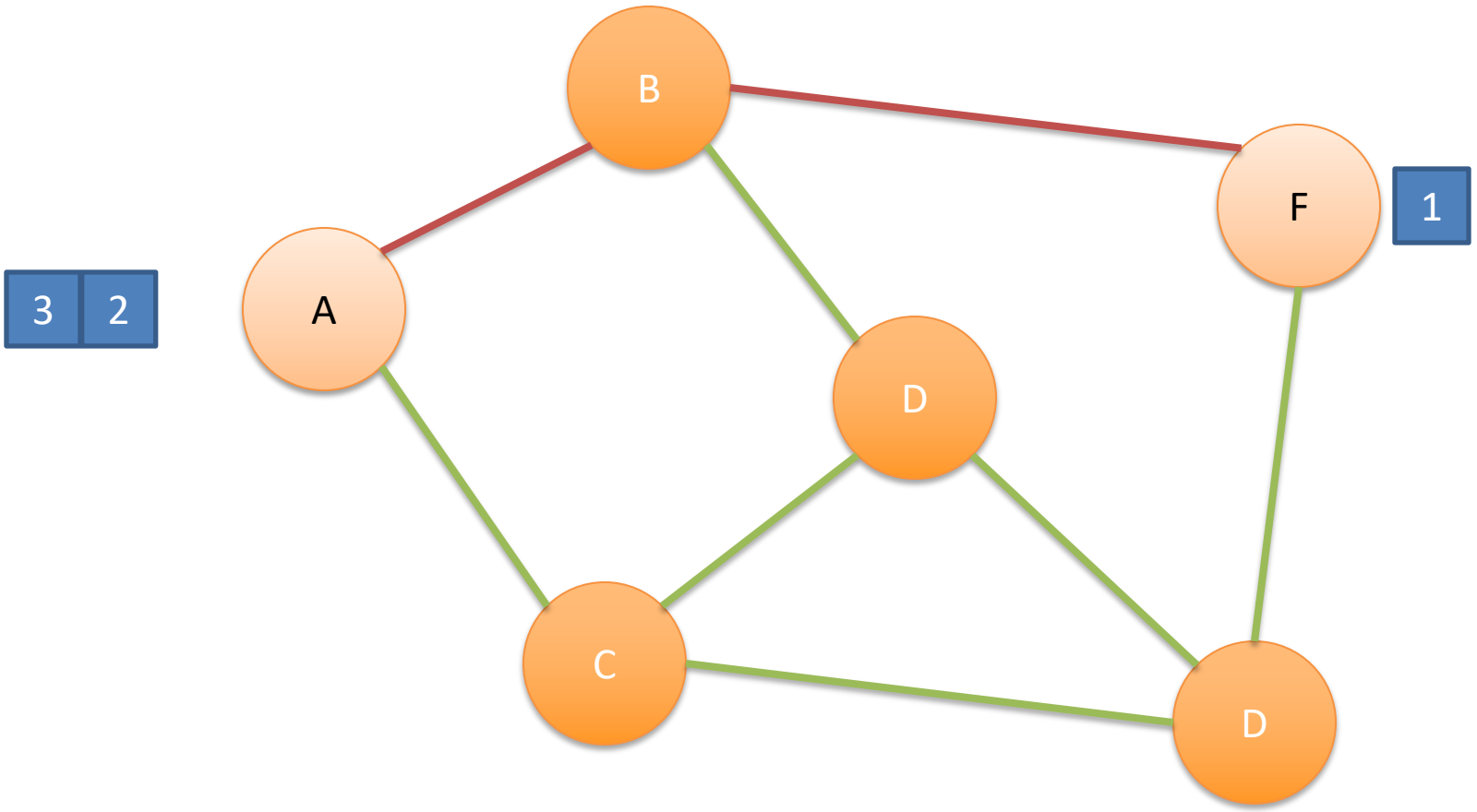
Circuit and Packet Switching

- Circuit switching
  - Legacy phone network
  - Single route through sequence of hardware devices established when two nodes start communication
  - Data sent along route
  - Route maintained until communication ends
- Packet switching
  - Internet
  - Data split into packets
  - Packets transported independently through network
  - Each packet handled on a best efforts basis
  - Packets may follow different routes

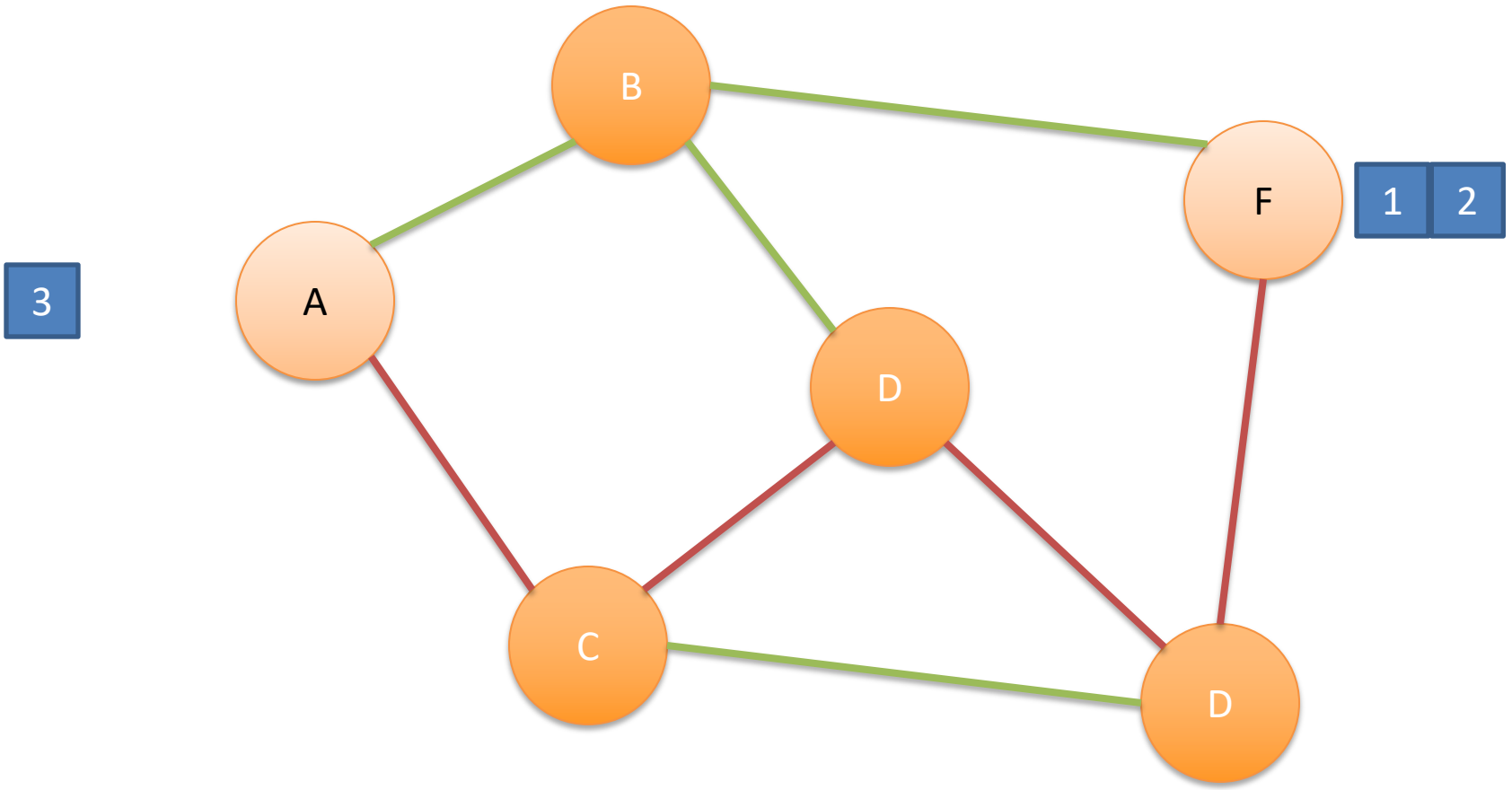
Packet Switching



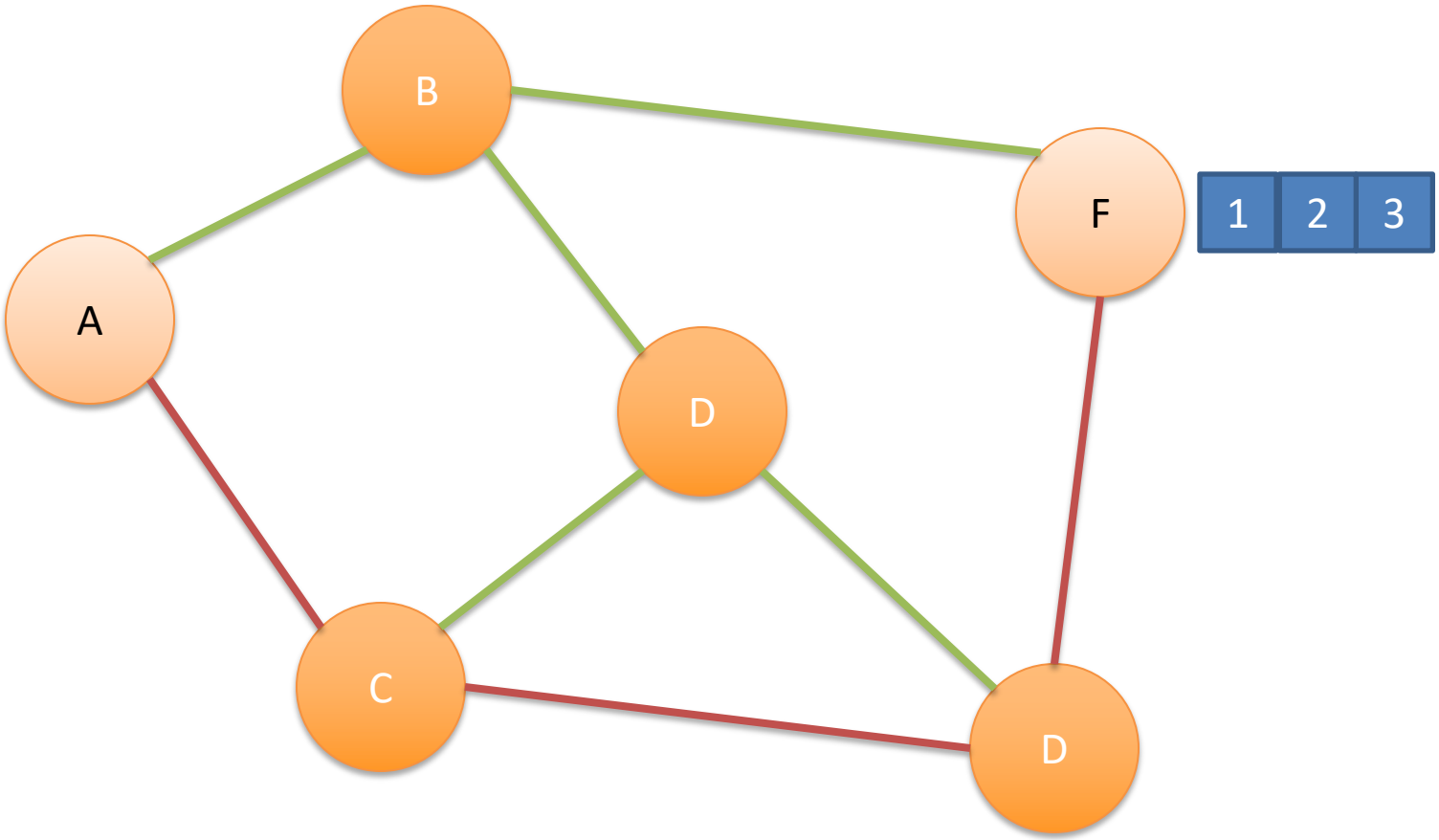
Packet Switching



Packet Switching



Packet Switching



Network Layers

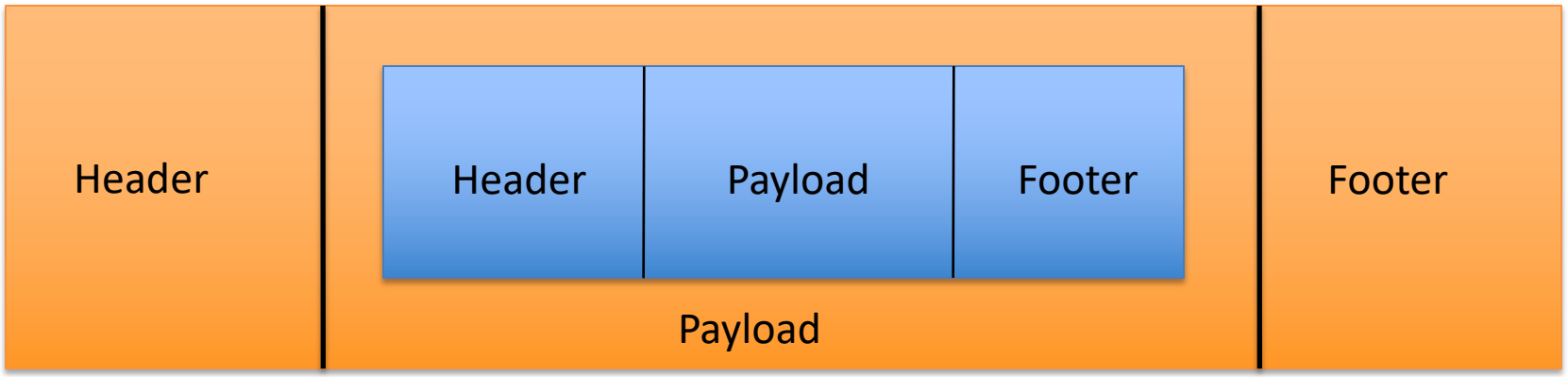
- Network models typically use a stack of layers
  - Higher layers use the services of lower layers via encapsulation
  - A layer can be implemented in hardware or software
  - The bottom-most layer must be in hardware
- A network device may implement several layers
- A communication channel between two nodes is established for each layer
  - Actual channel at the bottom layer
  - Virtual channel at higher layers

Protocols

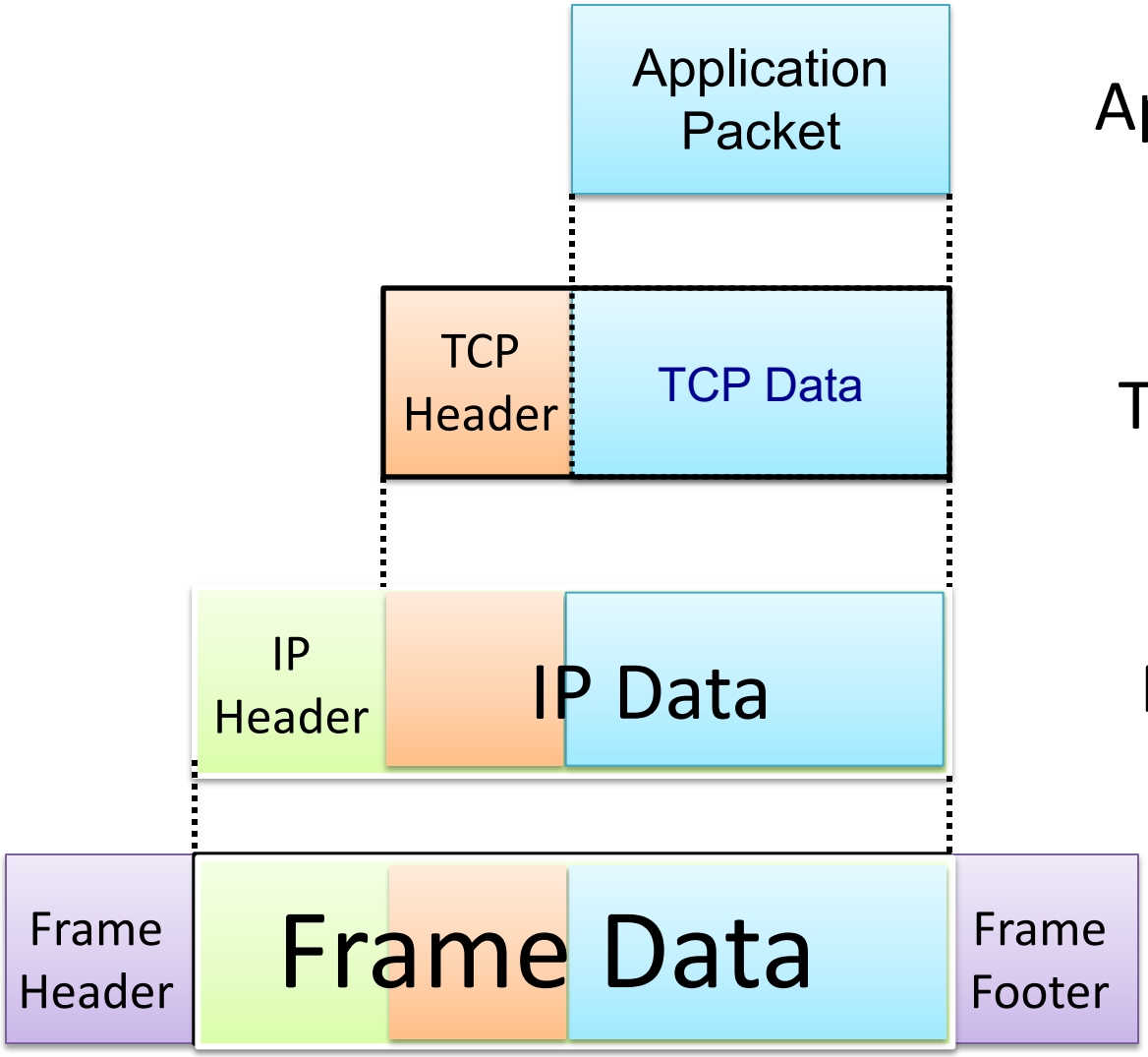
- A protocol defines the rules for communication between computers
- Protocols are broadly classified as connectionless and connection oriented
- Connectionless protocol
  - Sends data out as soon as there is enough data to be transmitted
  - E.g., user datagram protocol (UDP)
- Connection-oriented protocol
  - Provides a reliable connection stream between two nodes
  - Consists of set up, transmission, and tear down phases
  - Creates virtual circuit-switched network
  - E.g., transmission control protocol (TCP)

Encapsulation

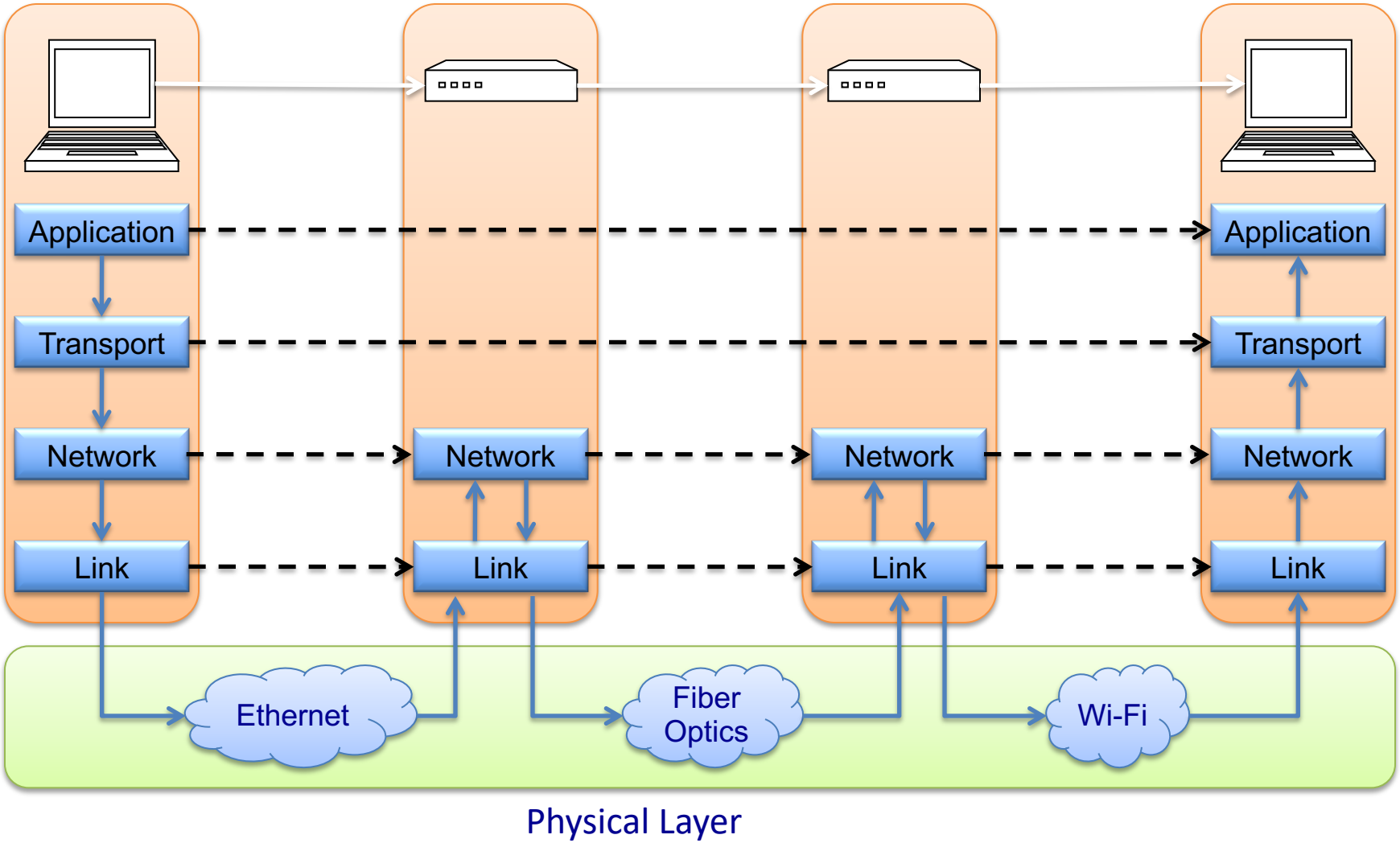
- A packet typically consists of
  - Control information for addressing the packet: header and footer
  - Data: payload
- A network protocol N1 can use the services of another network protocol N2
  - A packet p1 of N1 is encapsulated into a packet p2 of N2
  - The payload of p2 is p1
  - The control information of p2 is derived from that of p1



Internet Packet Encapsulation



Internet Layers

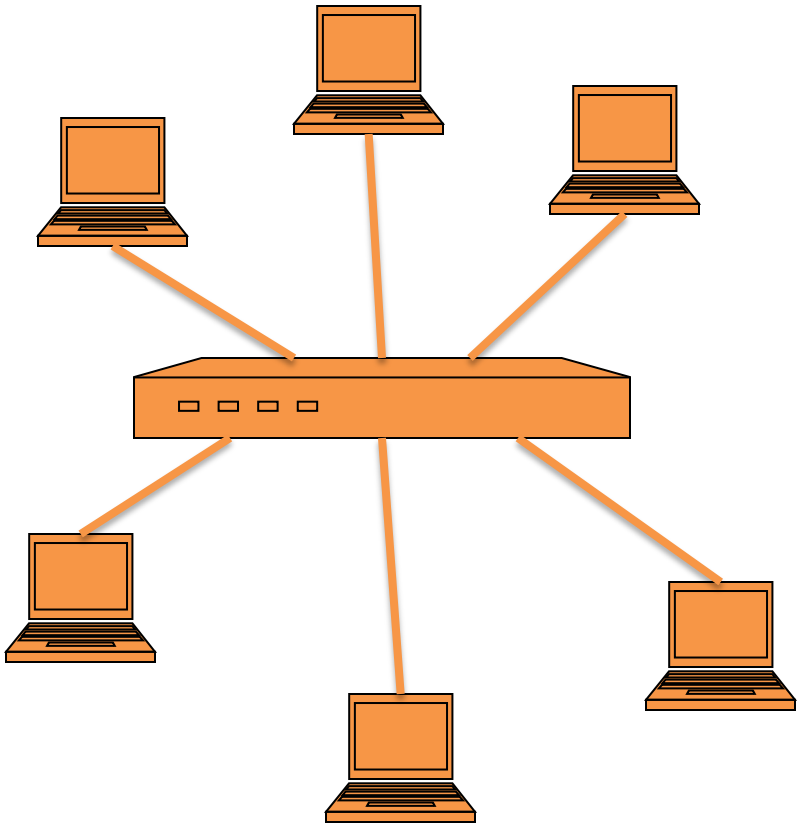


Network Interfaces

- Network interface: device connecting a computer to a network
  - Ethernet card
  - WiFi adapter
- A computer may have multiple network interfaces
- Packets transmitted between network interfaces
- Most local area networks, (including Ethernet and WiFi) broadcast frames
- In regular mode, each network interface gets the frames intended for it
- Traffic sniffing can be accomplished by configuring the network interface to read all frames (promiscuous mode)

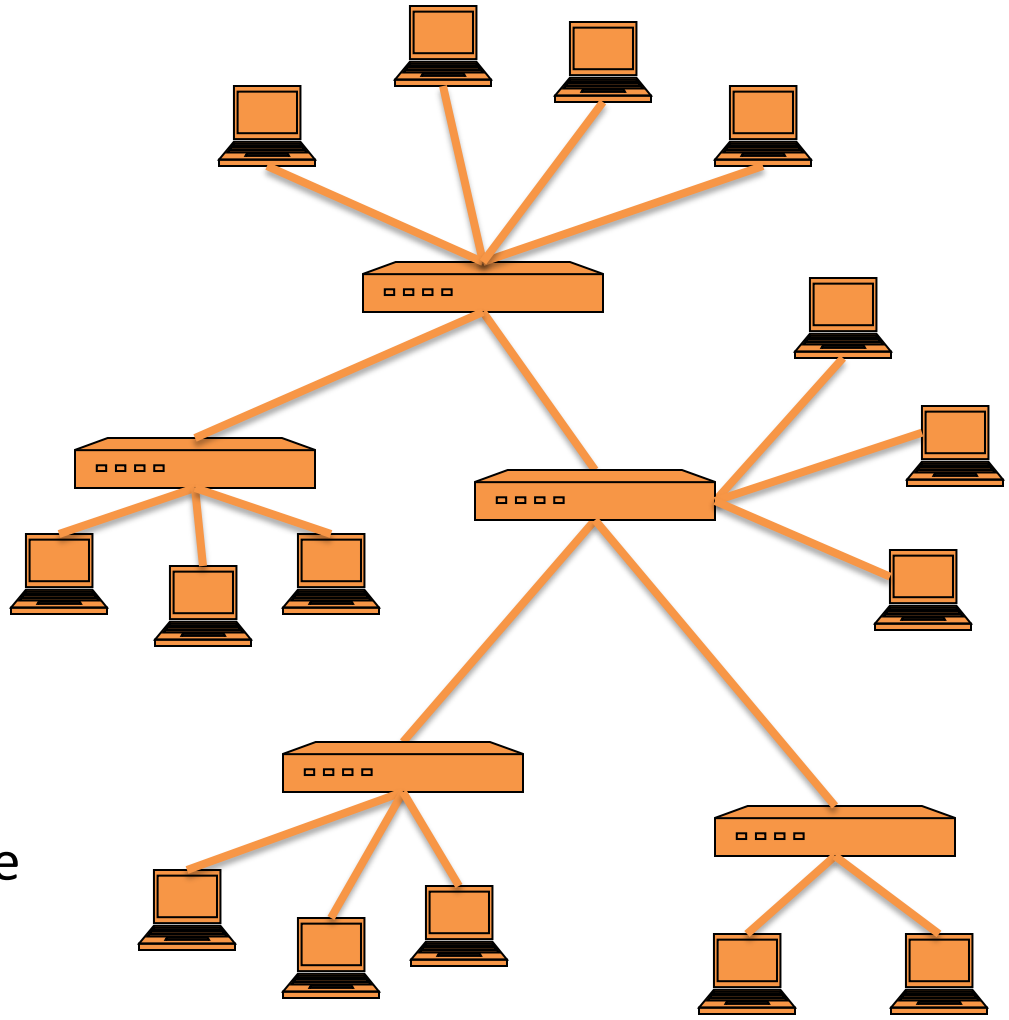
Switch

- A switch is a common network device
  - Operates at the link layer
  - Has multiple ports, each connected to a computer
- Operation of a switch
  - Learn the MAC address of each computer connected to it
  - Forward frames only to the destination computer



Combining Switches

- Switches can be arranged into a tree
- Each port learns the MAC addresses of the machines in the segment (subtree) connected to it
- Fragments to unknown MAC addresses are broadcast
- Frames to MAC addresses in the same segment as the sender are ignored

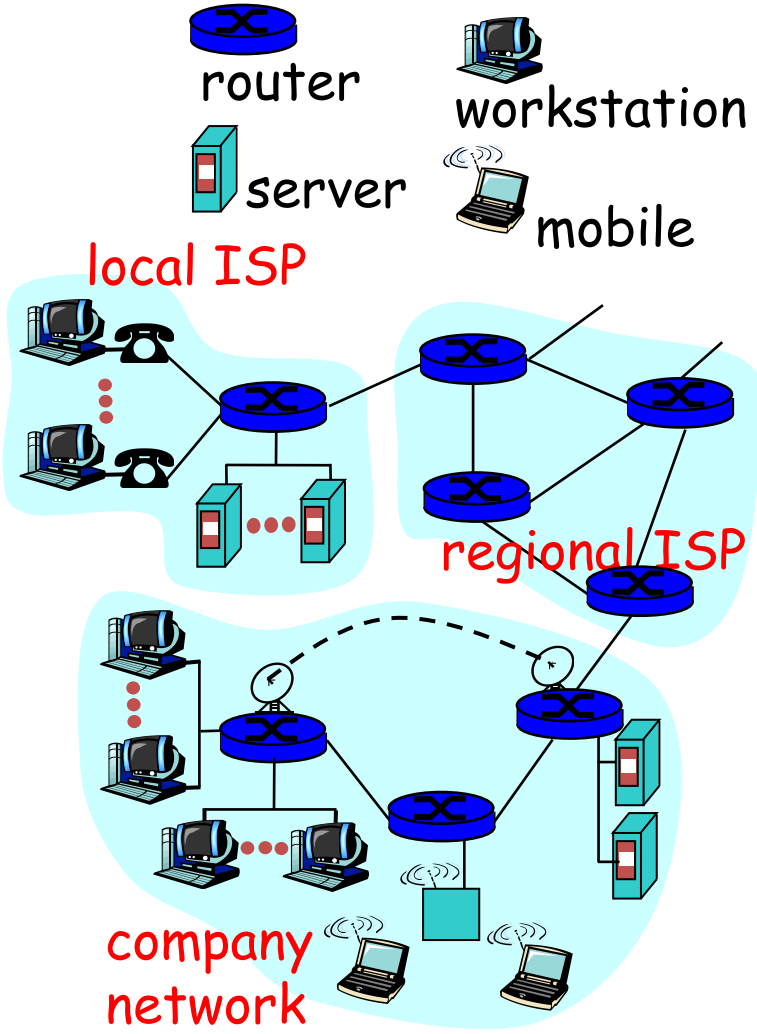


Inter-networking and the Internet

What's the Internet: "nuts and bolts" view

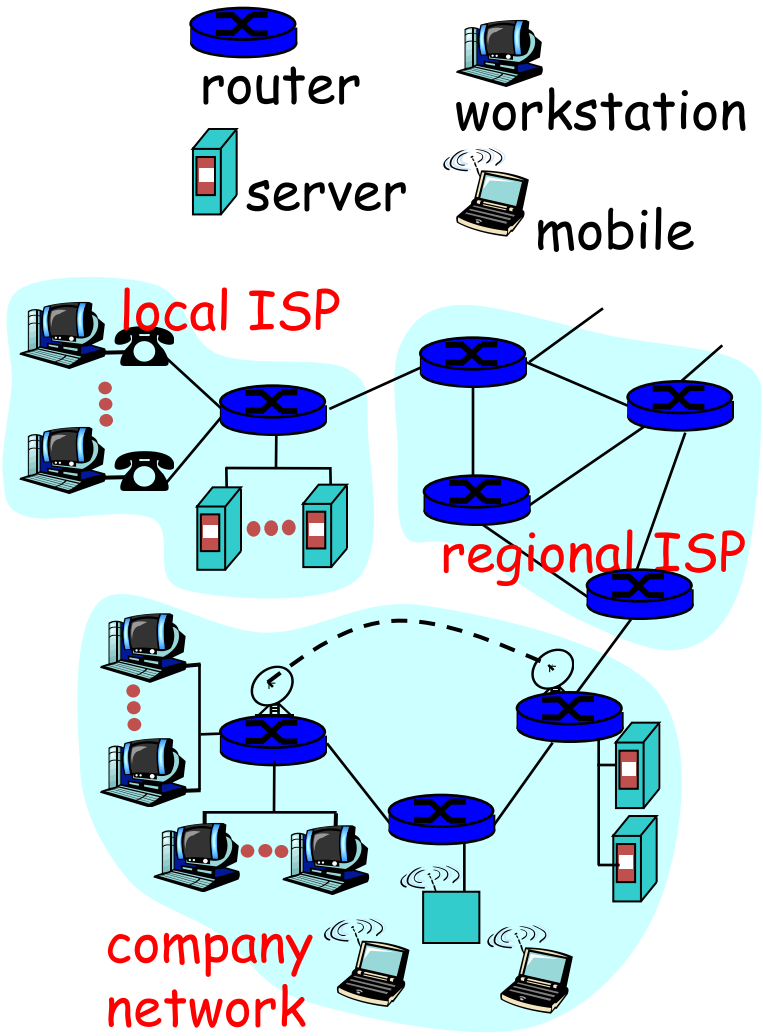
- millions of connected computing devices: *hosts, end-systems*
  - PCs workstations, servers
  - PDAs phones, toasters

  running *network apps*
- *communication links*
  - fiber, copper, radio, satellite
  - transmission rate = ***bandwidth***
- *routers:* forward packets (chunks of data)



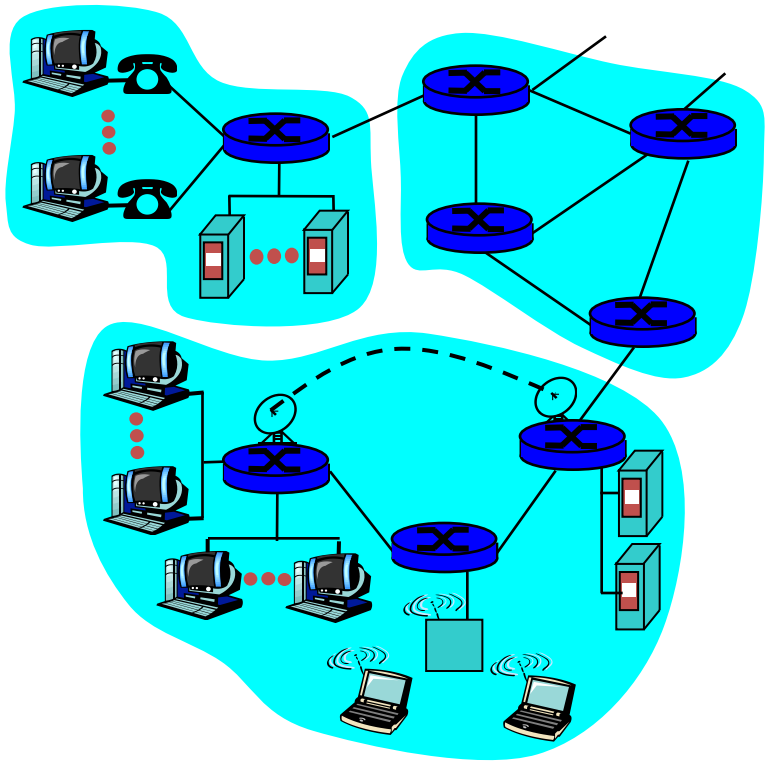
What's the Internet: "nuts and bolts" view

- *protocols* control sending, receiving of msgs
  - e.g., TCP, IP, HTTP, FTP, PPP
- *Internet:* "network of networks"
  - loosely hierarchical
  - public Internet versus private intranet
- Internet standards
  - RFC: Request for comments
  - IETF: Internet Engineering Task Force



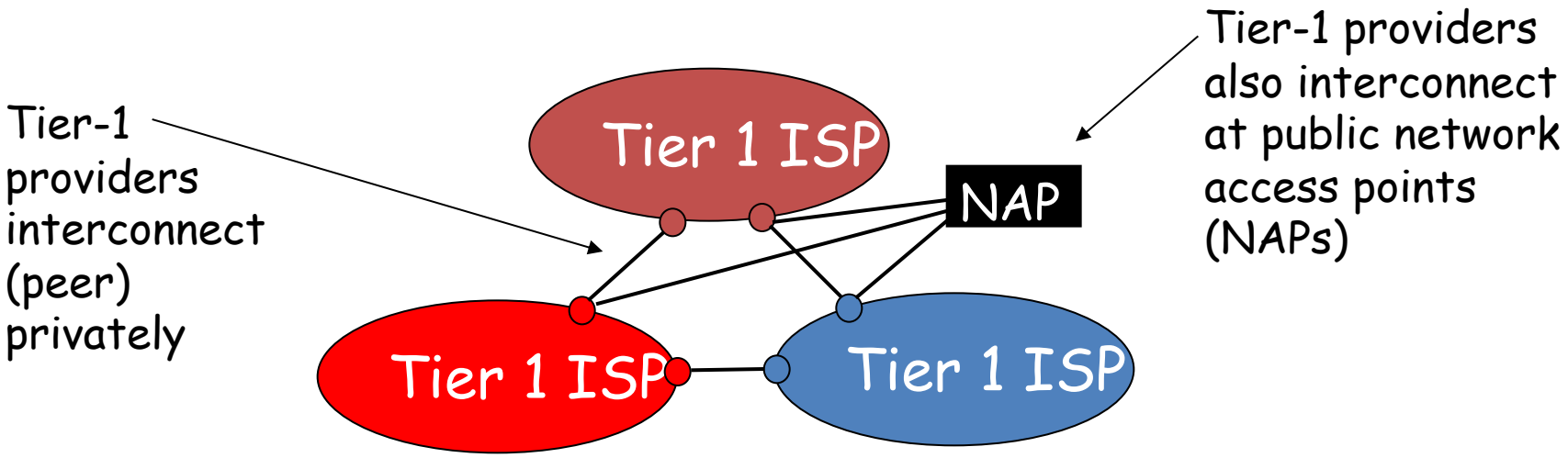
What's the Internet: a service view

- communication *infrastructure* enables distributed applications:
  - Web, email, games, e-commerce, database., voting, file (MP3) sharing
- communication services provided to apps:
  - connectionless
  - connection-oriented



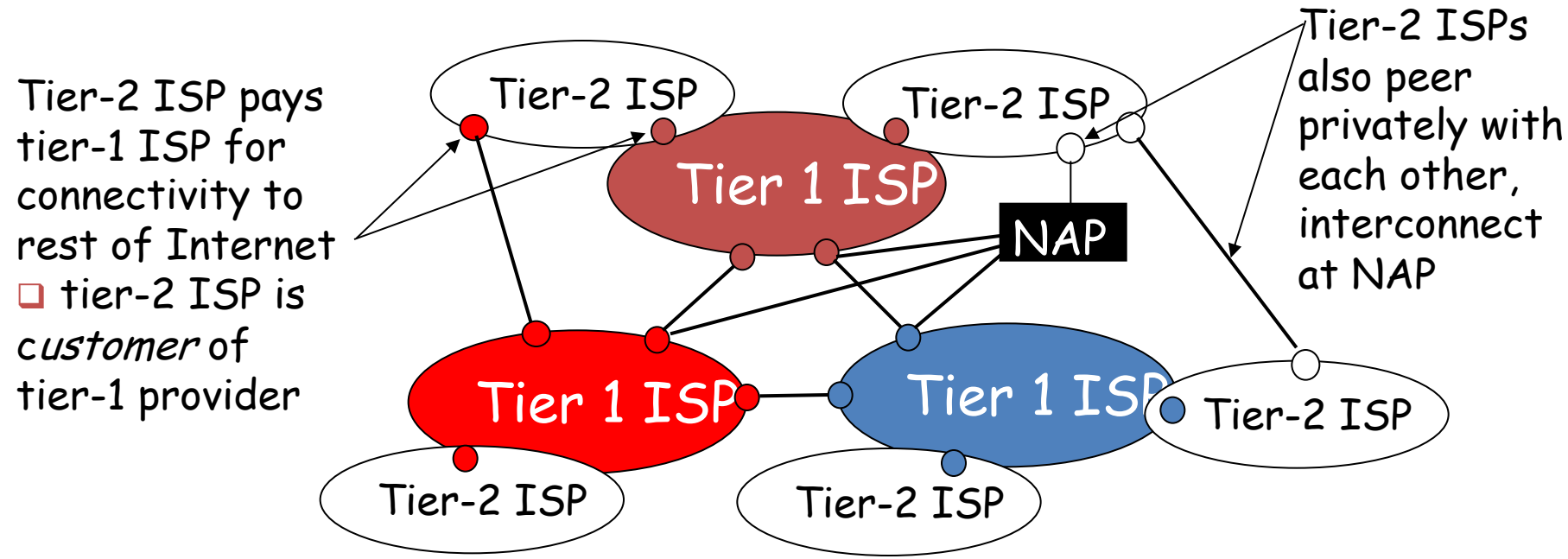
Internet structure: network of networks

- roughly hierarchical
- at center: "tier-1" ISPs (e.g., UUNet, BBN/Genuity, Sprint, AT&T), national/international coverage
  - treat each other as equals



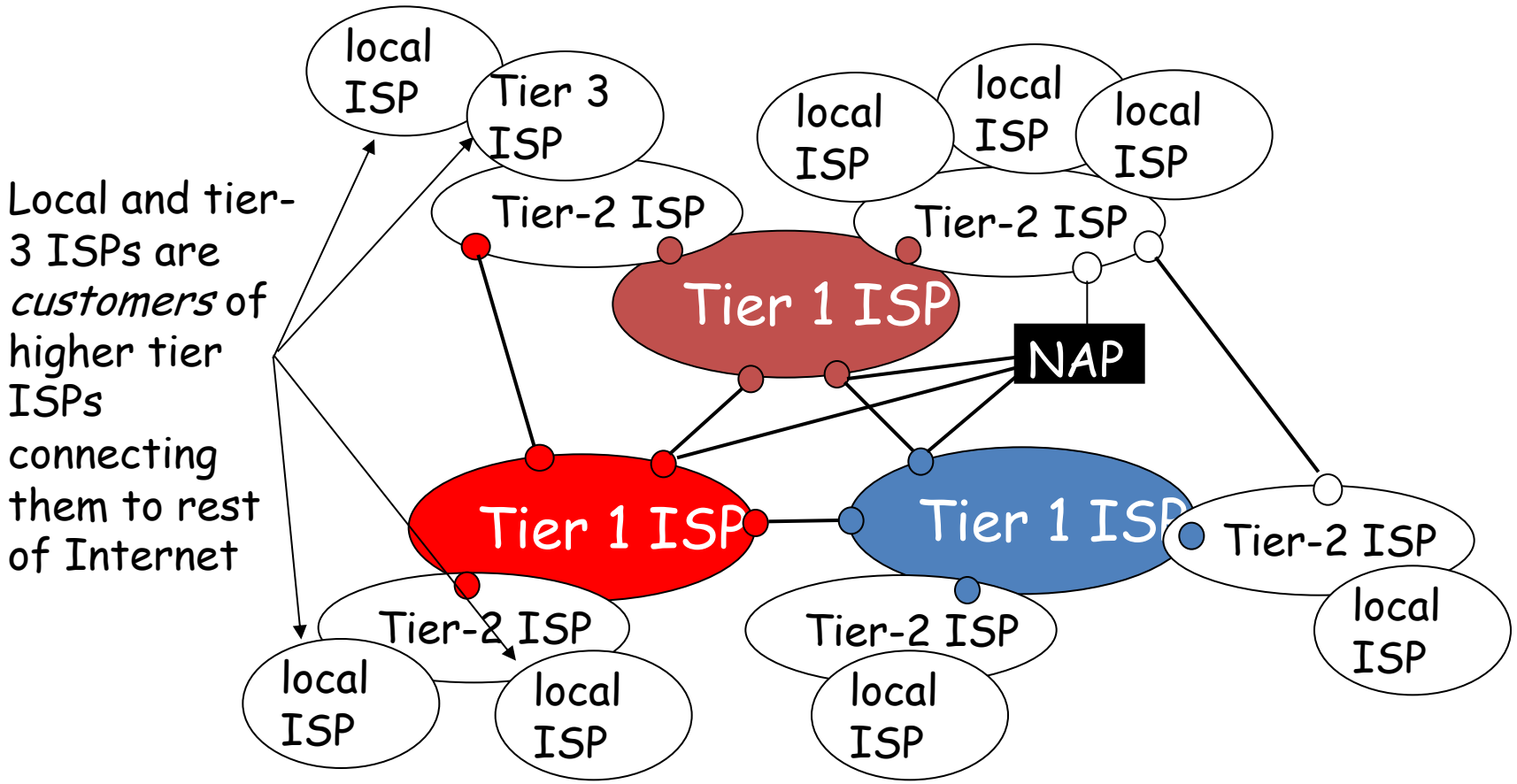
Internet structure: network of networks

- "Tier-2" ISPs: smaller (often regional) ISPs
  - Connect to one or more tier-1 ISPs, possibly other tier-2 ISPs

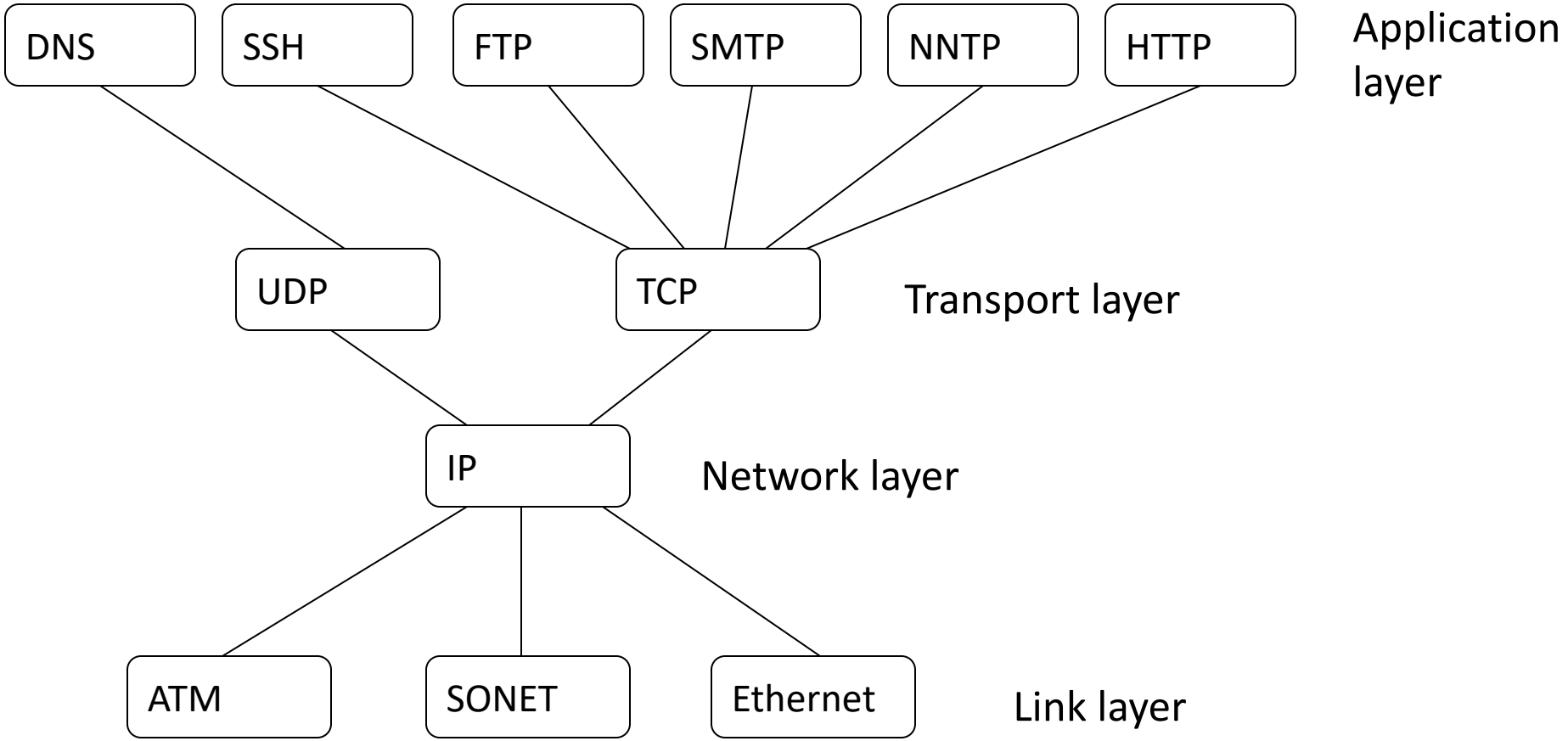


Internet structure: network of networks

- "Tier-3" ISPs and local ISPs
  - last hop ("access") network (closest to end systems)



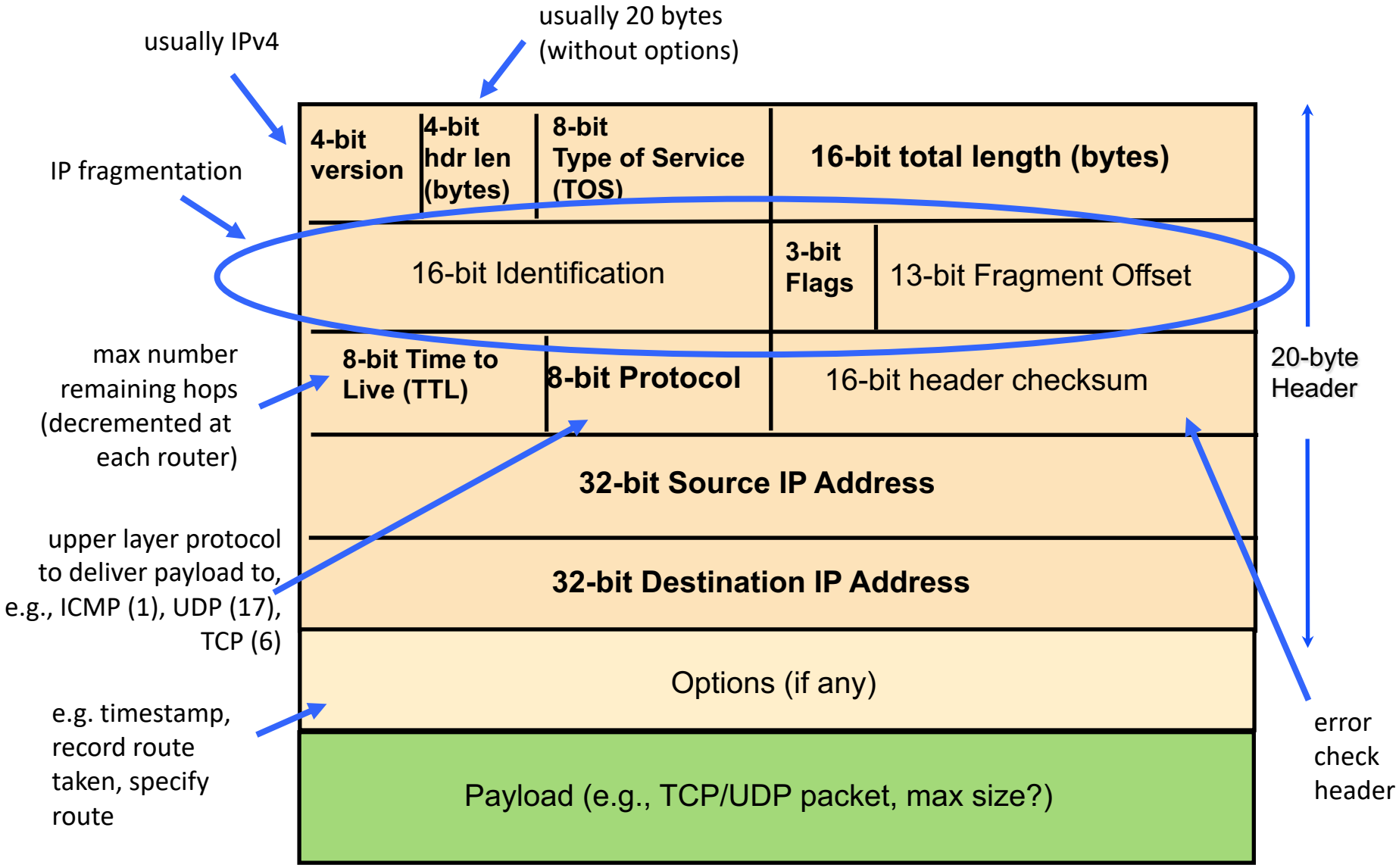
Layering of protocols



Internet Protocol

- Connectionless
  - Each packet is transported independently from other packets
- Unreliable
  - Delivery on a best effort basis
  - No acknowledgments
  - Packets may be lost, reordered, corrupted, or duplicated
- IP packets
  - Encapsulate TCP and UDP packets
  - Encapsulated into link-layer frames

IPv4 Packet Header Format



IP Addressing

- IP address used to route datagrams through network.
- IPv4 32 bit address, IPv6 128 bit address.
- Divided into two parts: network and host.
- Network part: Used to route packets. (ZIP code)
- Host part: Used to identify an individual host. (House number)
- Usually represented in dotted decimal notation: 141.211.144.212
- Each number represents 8 bits: 0-255.

Classless Interdomain Routing (CIDR)

- Allow division between network and host portion on any bit boundary.
  - More efficient use of address space.
  - Allows division/aggregation of sub-assignments.
- Networks now identified by network address and the length of the network portion: 141.213.8.0/24
- Hosts identified by address and network mask: 141.213.8.1, 255.255.255.0.
- WHY? Rapid depletion of class B address space and poor utilization of the assigned address space

Connection-oriented service

Goal: data transfer between end systems

- *handshaking:* setup (prepare for) data transfer ahead of time
  - Hello, hello back human protocol
  - *set up "state"* in two communicating hosts
- TCP - Transmission Control Protocol
  - Internet's connection-oriented service

TCP service [RFC 793]

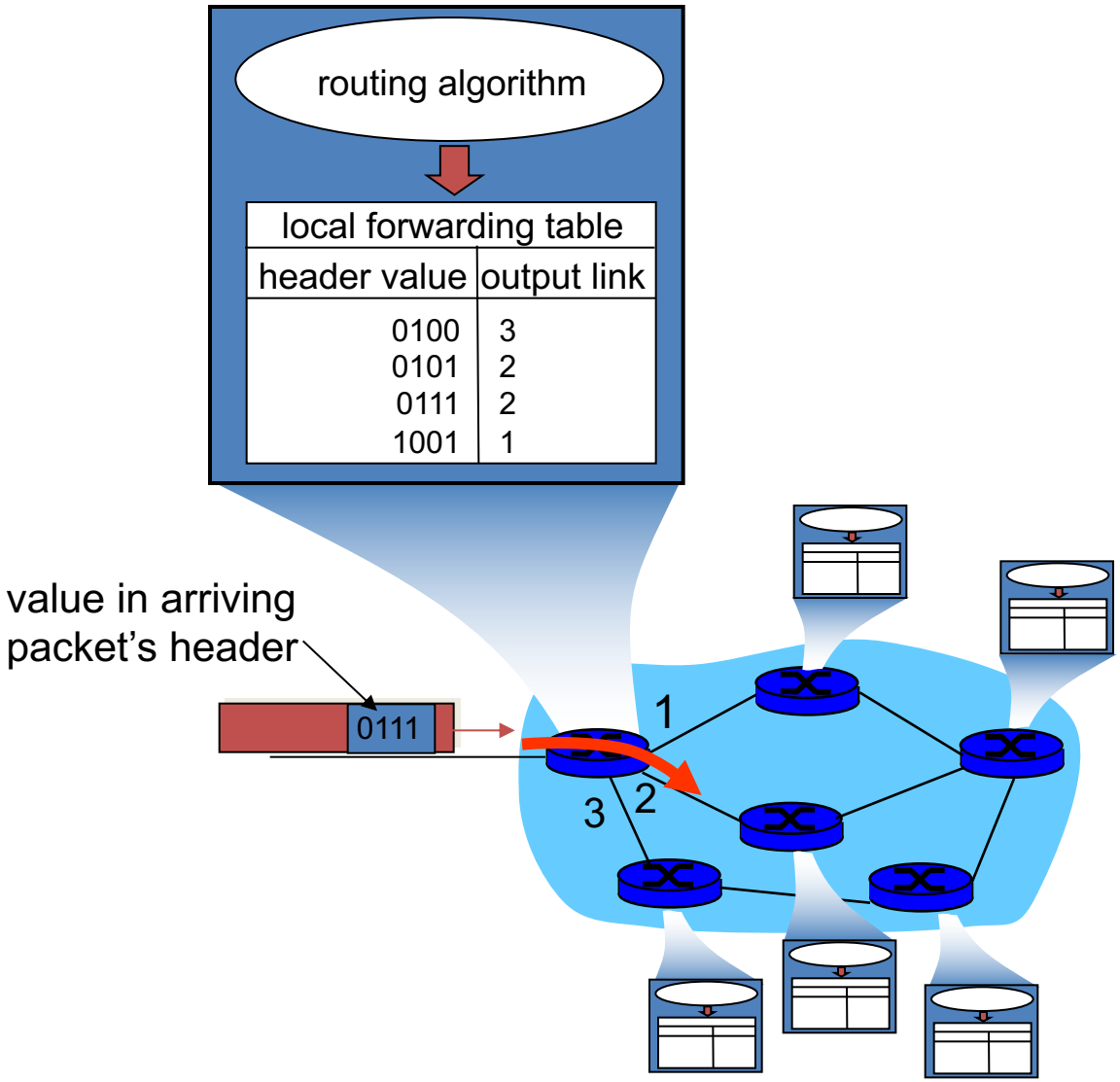
- Byte-steam abstraction: reliably delivering a stream of bytes from S to R
- *reliable, in-order* byte-stream data transfer
  - loss: acknowledgements and retransmissions
- *flow control:*
  - sender won't overwhelm receiver
- *congestion control:*
  - senders "slow down sending rate" when network congested

Network Services

BGP: IP Routing

- A **router** bridges two or more networks
  - Operates at the network layer
  - Maintains tables to forward packets to the appropriate network
  - Forwarding decisions based solely on the destination address
- Routing table
  - Maps ranges of addresses to LANs or other gateway routers

Interplay between routing and forwarding



Internet inter-AS routing: BGP

- BGP (Border Gateway Protocol): *the* de facto standard
- BGP provides each AS a means to:
  1. Obtain subnet reachability information from neighboring ASs.
  2. Propagate the reachability information to all routers internal to the AS.
  3. Determine "good" routes to subnets based on reachability information and policy.
- Allows a subnet to advertise its existence to rest of the Internet: *"I am here"*

DNS: Domain Name System

People: many identifiers:

- SSN, name, passport #

Internet hosts, routers:

- IP address (32 bit) - used for addressing datagrams
- "name", e.g., www.eecs.umich.edu - used by humans

Q: map between IP addresses and name ?

Domain Name System:

- *distributed database* implemented in hierarchy of many *name servers*
- *application-layer protocol* host, routers, name servers to communicate to *resolve* names (address/name translation)
  - note: core Internet function, implemented as application-layer protocol
  - complexity at network's "edge"

DNS name servers

Why not centralize DNS?

- single point of failure
- traffic volume
- distant centralized database
- Maintenance

doesn't *scale!*

Record Type	Million Queries	Percentage
A	1,220	70.4%
AAAA	206	11.9%
MX	152	8.8%
DS	69	4.0%
NS	25	1.4%
ANY	19	1.1%
TXT	18	1.0%
SOA	6	0.4%
A6	5	0.3%
SPF	4	0.3%
Other	8	0.5%
Total	1,732	

Table 8: Top 10 Resource Records Requested

- no server has all name-to-IP address mappings

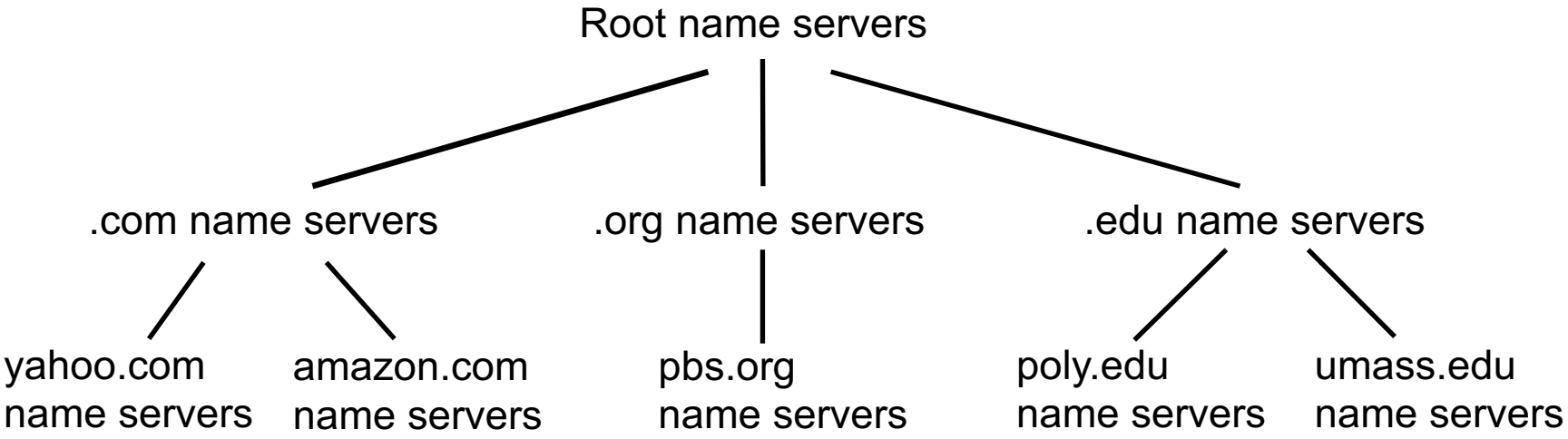
local name servers:

- each ISP, company has *local (default) name server*
- host DNS query first goes to local name server

authoritative name server:

- for a host: stores that host's IP address, name
- can perform name/address translation for that host's name

Distributed Hierarchical Database (1st Approx)



- Client wants IP for `www.amazon.com`:
- Client queries a root server to find `.com` name server
- Client queries `.com` name server to get `amazon.com` name server
- Client queries `amazon.com` name server to get IP address for `www.amazon.com`

ARP

- The address resolution protocol (ARP) connects the network layer to the data layer by converting IP addresses to MAC addresses
- ARP works by broadcasting requests and caching responses for future use
- The protocol begins with a computer broadcasting a message of the form

  who has <IP address1> tell <IP address2>

- When the machine with <IP address1> or an ARP server receives this message, its broadcasts the response

  <IP address1> is <MAC address>

- The requestor's IP address <IP address2> is contained in the link header
- The Linux and Windows command arp - a displays the ARP table

```
  Internet Address      Physical Address      Type
  128.148.31.1          00-00-0c-07-ac-00     dynamic
  128.148.31.15         00-0c-76-b2-d7-1d     dynamic
  128.148.31.71         00-0c-76-b2-d0-d2     dynamic
  128.148.31.75         00-0c-76-b2-d7-1d     dynamic
  128.148.31.102        00-22-0c-a3-e4-00     dynamic
  128.148.31.137        00-1d-92-b6-f1-a9     dynamic
```

Networking

USER/APPLICATION PERSPECTIVE

Startup

- When Linux boots as an operating system, it loads its image from the disk into memory, unpacks it, and establishes itself by installing the file systems and memory management and other key systems.
- As the kernel's last (initialization) task, it executes the init program.
- This program reads a configuration file (/etc/inittab) which directs it to execute a startup script.
- This in turn executes more scripts, eventually including the network script (/etc/rc.d/init.d/network).

https://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html

ifconfig

```
ifconfig ${DEVICE} ${IPADDR} netmask ${NMASK} broadcast ${BCAST}

ifconfig

eth0  Link encap:Ethernet  HWaddr 00:C1:4E:7D:9E:25
      inet addr:172.16.1.1  Bcast:172.16.1.255  Mask:255.255.255.0
      UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
      RX packets:389016 errors:16534 dropped:0 overruns:0 frame:24522
      TX packets:400845 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:100
      Interrupt:11 Base address:0xcc00
```

https://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html

Useful examples

- ifconfig eth0 down - shut down eth0
- ifconfig eth1 up - activate eth1
- ifconfig eth0 arp - enable ARP on eth0
- ifconfig eth0 -arp - disable ARP on eth0
- ifconfig eth0 netmask 255.255.255.0 - set the eth0 netmask
- ifconfig lo mtu 2000 - set the loopback maximum transfer unit
- ifconfig eth1 172.16.0.7 - set the eth1 IP address

https://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html

Route program

```
route add -net ${NETWORK} netmask ${NMASK} dev ${DEVICE} -or-
route add -host ${IPADDR} ${DEVICE}

route

Kernel IP routing table
Destination   Gateway         Genmask         Flags Metric Ref Use Iface
172.16.1.4    *               255.255.255.255 UH    0      0     0 eth0
172.16.1.0    *               255.255.255.0   U     0      0     0 eth0
127.0.0.0     *               255.0.0.0       U     0      0     0 lo
default       viper.u.edu     0.0.0.0         UG    0      0     0 eth0
```

The *route* program simply adds predefined routes for interface devices to the Forwarding Information Base (FIB).

https://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html

Useful examples

- route add -host 127.16.1.0 eth1 - adds a route to a host
- route add -net 172.16.1.0 netmask 255.255.255.0 eth0 - adds a network
- route add default gw jeep - sets the default route through jeep
- (Note that a route to jeep must already be set up)
- route del -host 172.16.1.16 - deletes entry for host 172.16.1.16

https://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html

Overview

- On each end, an application gets a socket, makes the transport layer connection, and then sends or receives packets.
- In Linux, a socket is actually composed of two socket structures (one that contains the other). When an application creates a socket, it is initialized but empty.
- When the socket makes a connection (whether or not this involves traffic with the other end) the IP layer determines the route to the distant host and stores that information in the socket.
- From that point on, all traffic using that connection uses that route - sent packets will travel through the correct device and the proper routers to the distant host, and received packets will appear in the socket's queue.

https://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html

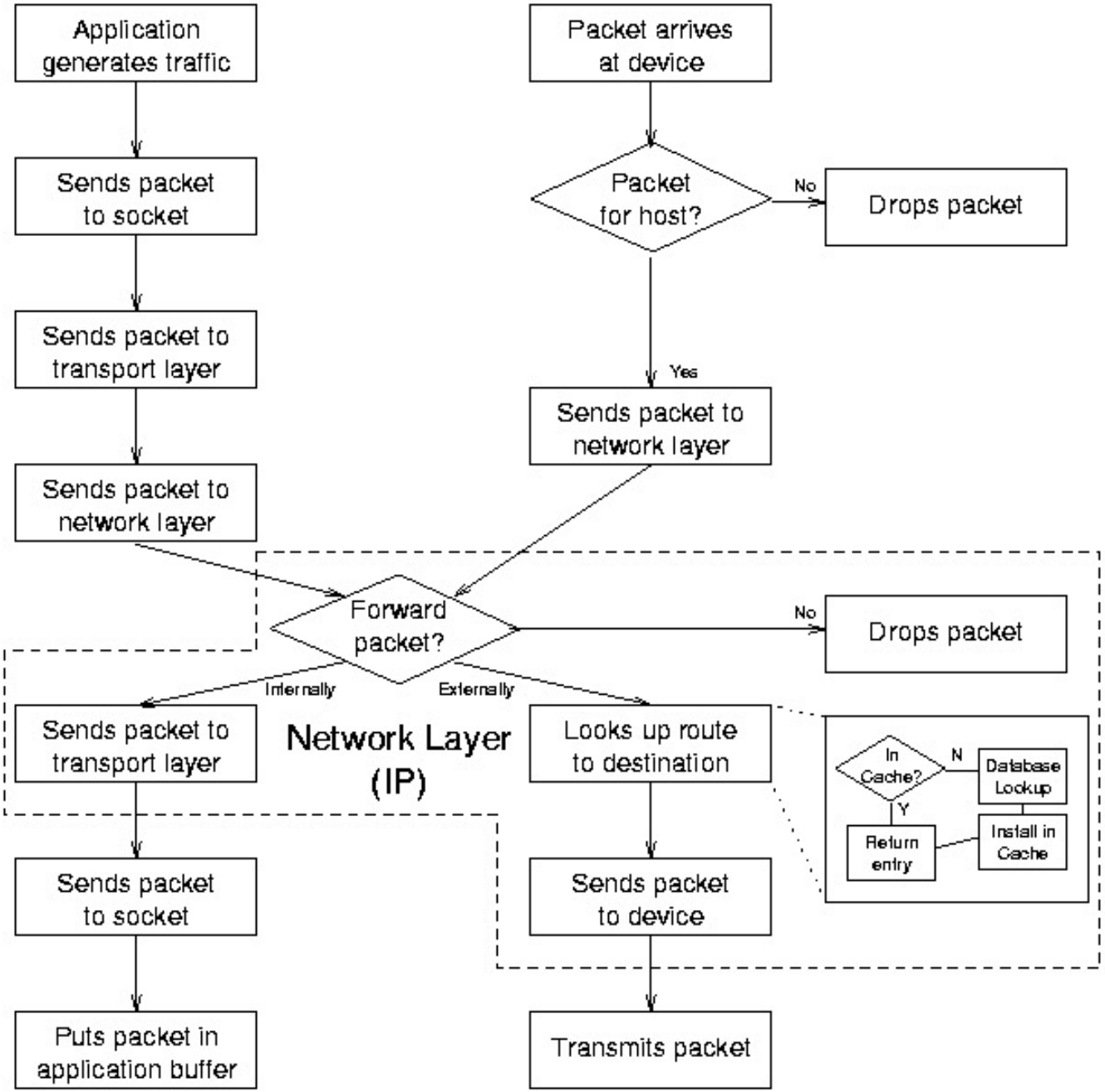
Establishing Connections

```
/* look up host */
server = gethostbyname(SERVER_NAME);
/* get socket */
sockfd = socket(AF_INET, SOCK_STREAM, 0);
/* set up address */
address.sin_family = AF_INET;
address.sin_port = htons(PORT_NUM);
memcpy(&address.sin_addr,server->h_addr,server->h_length);
/* connect to server */
connect(sockfd, &address, sizeof(address));
```

https://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html

Networking

KERNEL


Application generates traffic
Sends packet to socket
Sends packet to transport layer
Sends packet to network layer
Packet arrives at device
Packet for host?
No
Drops packet
Yes
Sends packet to network layer
Forward packet?
No
Drops packet
Internally
Externally
Network Layer (IP)
Sends packet to transport layer
Looks up route to destination
In Cache?
N
Database Lookup
Y
Return entry
Install in Cache
Sends packet to socket
Sends packet to device
Puts packet in application buffer
Transmits packet

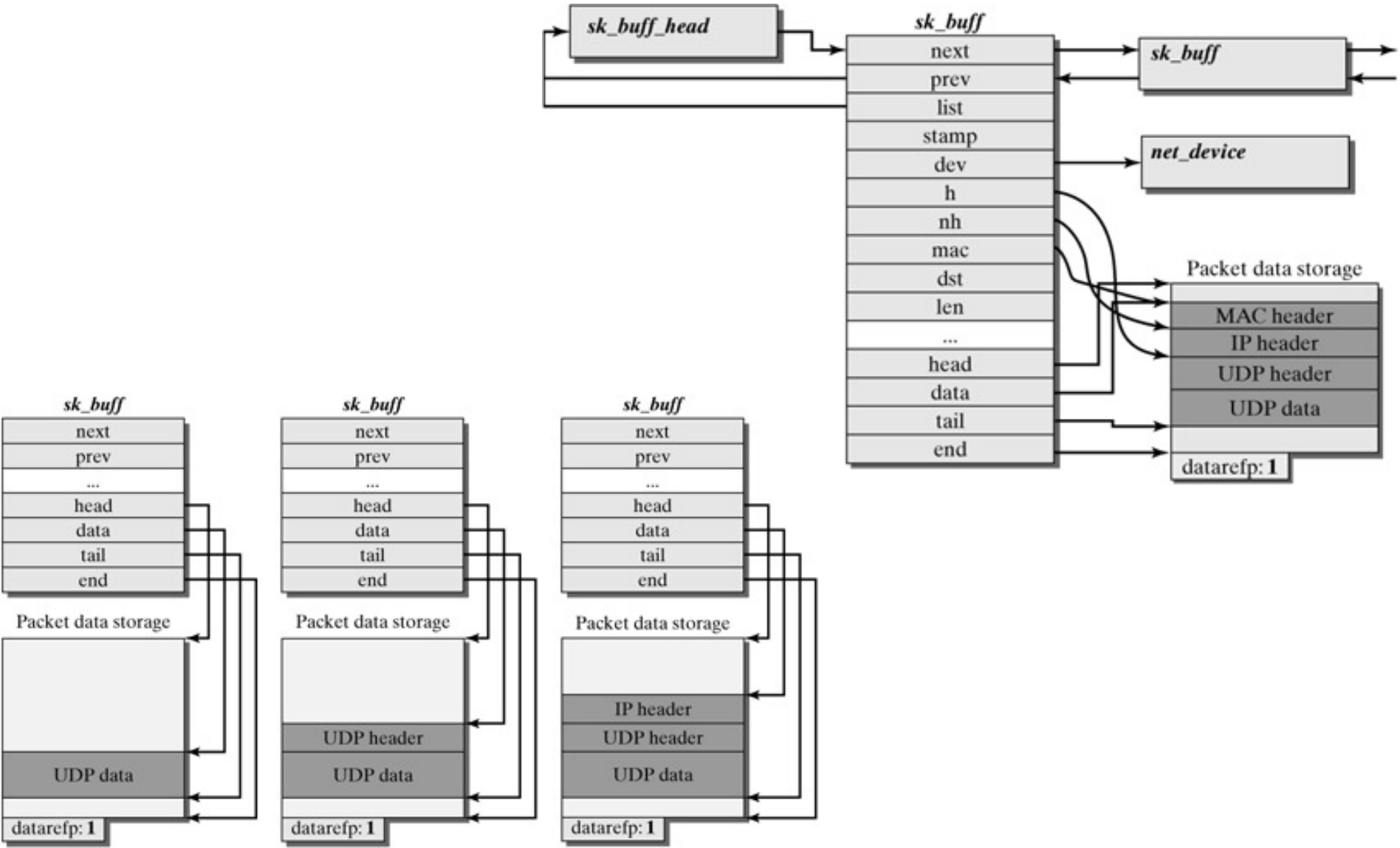
Internet routing

- The IP layer handles routing between computers. It keeps two data structures;
  - a Forwarding Information Base (FIB) that keeps track of all of the details for every known route,
  - and a faster routing cache for destinations that are currently in use.
- The FIB is the primary routing reference
  - it contains up to 32 zones (one for each bit in an IP address) and entries for every known destination.
  - Each zone contains entries for networks or hosts that can be uniquely identified by a certain number of bits
    - a network with a netmask of 255.0.0.0 has 8 significant bits and would be in zone 8, while a network with a netmask of 255.255.255.0 has 24 significant bits and would be in zone 24.
  - When IP needs a route, it begins with the most specific zones and searches the entire table until it finds a match (there should always be at least one default entry).
- The routing cache is a hash table that IP uses to actually route packets.
  - It contains up to 256 chains of current routing entries, with each entry's position determined by a hash function.
  - When a host needs to send a packet, IP looks for an entry in the routing cache. If there is none, it finds the appropriate route in the FIB and inserts a new entry into the cache.
- These tables perform all the routing on a normal system.

Packet structure (sk_buff)

- The key to maintaining the strict layering of protocols without wasting time copying parameters and payloads back and forth is the common packet data structure (a socket buffer, or sk_buff).
- Throughout all of the various function calls as the data makes it way through the protocols, the payload data is copied only twice; once from user to kernel space and once from kernel space to output medium (for an outbound packet)

sk_buff


sk_buff_head
sk_buff
next
prev
list
stamp
dev
h
nh
mac
dst
len
...
head
data
tail
end
sk_buff
net_device
Packet data storage
MAC header
IP header
UDP header
UDP data
datarefp: 1
sk_buff
next
prev
...
head
data
tail
end
Packet data storage
UDP data
datarefp: 1
sk_buff
next
prev
...
head
data
tail
end
Packet data storage
UDP header
UDP data
datarefp: 1
sk_buff
next
prev
...
head
data
tail
end
Packet data storage
IP header
UDP header
UDP data
datarefp: 1

Socket Structures

- There are two main socket structures in Linux: general BSD sockets and IP specific INET sockets.
- They are strongly interrelated; a BSD socket has an INET socket as a data member and an INET socket has a BSD socket as its owner.

https://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html

BSD sockets

- BSD sockets are of type struct socket as defined in include/linux/socket.h.
- Some fields include:
  - struct proto_ops *ops
    - this structure contains pointers to protocol specific functions for implementing general socket behavior.
    - For example, ops- > sendmsg points to the inet_sendmsg() function.
  - struct inode *inode
    - this structure points to the file inode that is associated with this socket.
  - struct sock *sk
    - this is the INET socket that is associated with this socket.

https://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html

INET sockets

- INET sockets are of type struct sock as defined in include/net/sock.h.
- Some fields include:
  - struct sock *next, *pprev
    - all sockets are linked by various protocols, so these pointers allow the protocols to traverse them.
  - struct dst_entry *dst_cache
    - this is a pointer to the route to the socket's other side (the destination for sent packets).
  - struct sk_buff_head receive_queue
    - this is the head of the receive queue.
  - struct sk_buff_head write_queue
    - this is the head of the send queue.
  - __u32 saddr
    - the (Internet) source address for this socket.
  - struct sk_buff_head back_log,error_queue
    - extra queues for a backlog of packets (not to be confused with the main backlog queue) and erroneous packets for this socket.
  - struct proto *prot
    - this structure contains pointers to transport layer protocol specific functions. For example, prot- > recvmsg may point to the tcp_v4_recvmsg() function.
  - union struct tcp_op af_tcp; tp_pinfo
    - TCP options for this socket.
  - struct socket *sock
    - the parent BSD socket

Establishing Connections

```
/* look up host */
server = gethostbyname(SERVER_NAME);
/* get socket */
sockfd = socket(AF_INET, SOCK_STREAM, 0);
/* set up address */
address.sin_family = AF_INET;
address.sin_port = htons(PORT_NUM);
memcpy(&address.sin_addr,server->h_addr,server->h_length);
/* connect to server */
connect(sockfd, &address, sizeof(address));
```

https://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html

Socket Call Walk-Through

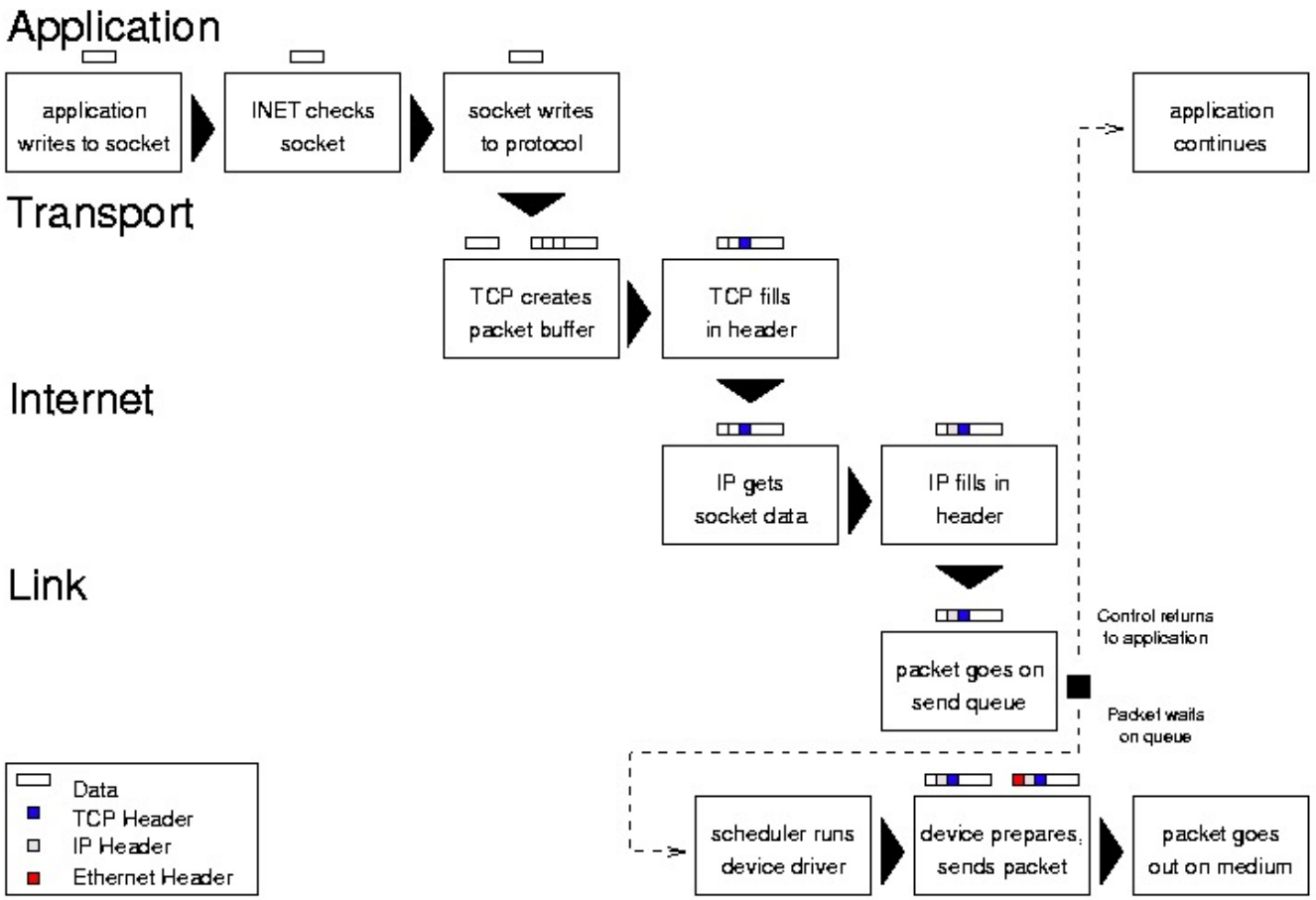
- Check for errors in call
- Create (allocate memory for) socket object
- Put socket into INODE list
- Establish pointers to protocol functions (INET)
- Store values for socket type and protocol family
- Set socket state to closed
- Initialize packet queues

https://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html

Connect Call Walk-Through

- Check for errors
- Determine route to destination:
  - Check routing table for existing entry (return that if one exists)
  - Look up destination in FIB
  - Build new routing table entry
  - Put entry in routing table and return it
- Store pointer to routing entry in socket
- Call protocol specific connection function (e.g., send a TCP connection packet)
- Set socket state to established

Sending


Application
application writes to socket
INET checks socket
socket writes to protocol
application continues
Transport
TCP creates packet buffer
TCP fills in header
Internet
IP gets socket data
IP fills in header
Link
packet goes on send queue
Control returns to application
Packet waits on queue
scheduler runs device driver
device prepares, sends packet
packet goes out on medium
Data
TCP Header
IP Header
Ethernet Header

Writing to a Socket

- Write data to a socket (application)
- Fill in message header with location of data (socket)
- Check for basic errors - is socket bound to a port? can the socket send messages? is there something wrong with the socket?
- Pass the message header to appropriate transport protocol (INET socket)

Creating a Packet with UDP

- Check for errors - is the data too big? is it a UDP connection?
- Make sure there is a route to the destination (call the IP routing routines if the route is not already established; fail if there is no route)
- Create a UDP header (for the packet)
- Call the IP build and transmit function

Creating a Packet with TCP

- Check connection - is it established? is it open? is the socket working?
- Check for and combine data with partial packets if possible
- Create a packet buffer
- Copy the payload from user space
- Add the packet to the outbound queue
- Build current TCP header into packet (with ACKs, SYN, etc.)
- Call the IP transmit function

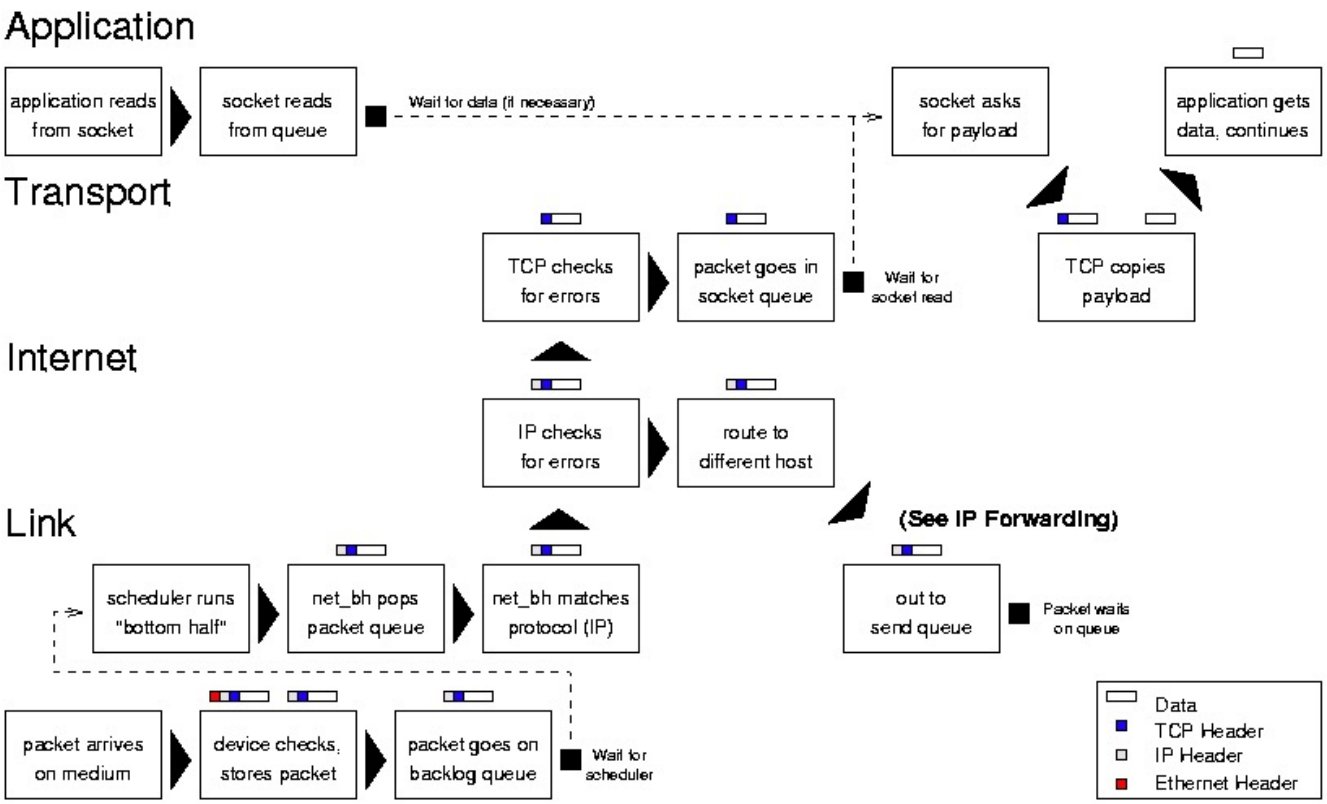
Wrapping a Packet in IP

- Create a packet buffer (if necessary - UDP)
- Look up route to destination (if necessary - TCP)
- Fill in the packet IP header
- Copy the transport header and the payload from user space
- Send the packet to the destination route's device output funtion

Transmitting a Packet

- Put the packet on the device output queue
- Wake up the device
- Wait for the scheduler to run the device driver
- Test the medium (device)
- Send the link header
- Tell the bus to transmit the packet over the medium

Receiving


Application
application reads from socket
socket reads from queue
Wait for data (if necessary)
socket asks for payload
application gets data, continues
Transport
TCP checks for errors
packet goes in socket queue
Wait for socket read
TCP copies payload
Internet
IP checks for errors
route to different host
(See IP Forwarding)
Link
scheduler runs "bottom half"
net_bh pops packet queue
net_bh matches protocol (IP)
out to send queue
Packet waits on queue
packet arrives on medium
device checks, stores packet
packet goes on backlog queue
Wait for scheduler
Data
TCP Header
IP Header
Ethernet Header

Reading from a Socket (Part I)

- Try to read data from a socket (application)
- Fill in message header with location of buffer (socket)
- Check for basic errors - is the socket bound to a port? can the socket accept messages? is there something wrong with the socket?
- Pass the message header with to the appropriate transport protocol (INET socket)
- Sleep until there is enough data to read from the socket (TCP/UDP)

Receiving a Packet

- Wake up the receiving device (interrupt)
- Test the medium (device)
- Receive the link header
- Allocate space for the packet
- Tell the bus to put the packet into the buffer
- Put the packet on the backlog queue
- Set the flag to run the network bottom half when possible
- Return control to the current process

Running the Network ``Bottom Half''

- Run the network bottom half (scheduler)
- Send any packets that are waiting to prevent interrupts (bottom half)
- Loop through all packets in the backlog queue and pass the packet up to its Internet reception protocol - IP
- Flush the sending queue again
- Exit the bottom half

Unwrapping a Packet in IP

- Check packet for errors - too short? too long? invalid version? checksum error?
- Defragment the packet if necessary
- Get the route for the packet (could be for this host or could need to be forwarded)
- Send the packet to its destination handling routine (TCP or UDP reception, or possibly retransmission to another host)

Accepting a Packet in UDP

- Check UDP header for errors
- Match destination to socket
- Send an error message back if there is no such socket
- Put packet into appropriate socket receive queue
- Wake up any processes waiting for data from that socket

Accepting a Packet in TCP

- Check sequence and flags; store packet in correct space if possible
- If already received, send immediate ACK and drop packet
- Determine which socket packet belongs to
- Put packet into appropriate socket receive queue
- Wake up and processes waiting for data from that socket

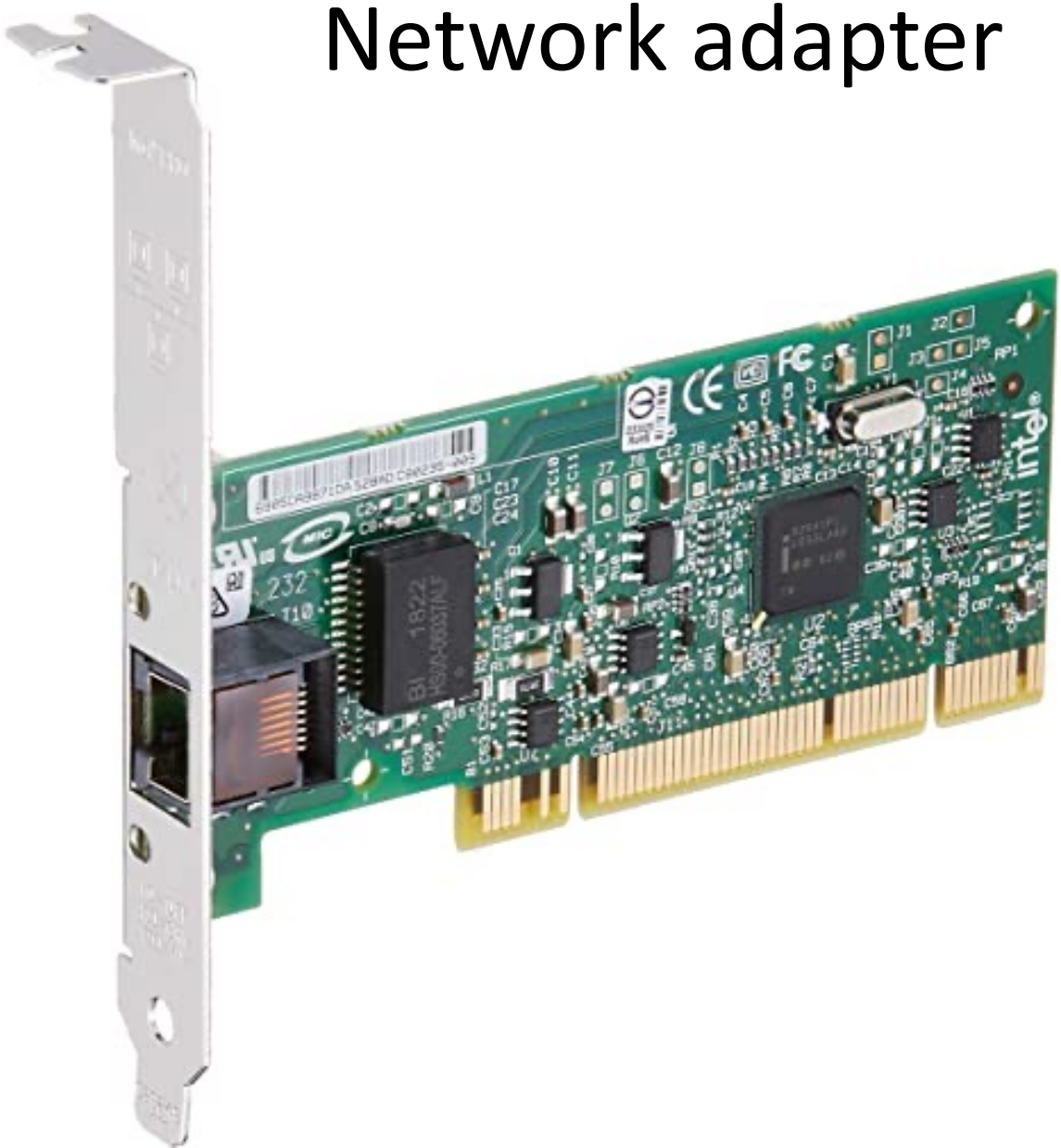
Reading from a Socket (Part II)

- Wake up when data is ready (socket)
- Call transport layer receive function
- Move data from receive queue to user buffer (TCP/UDP)
- Return data and control to application (socket)

Networking

HARDWARE

Intel PRO/1000 GT Desktop Adapter - Network adapter



Datasheet

- The Intel® 82541ER Gigabit Ethernet is a single, compact component with an integrated Gigabit Ethernet Media Access Control (MAC) and physical layer (PHY) functions.
  - IEEE 802.3 Ethernet interface for 1000BASE-T, 100BASE_TX, and 10BASE-T applications (802.3, 802.3u, and 802.3ab).
  - The controller is capable of transmitting and receiving data at rates of 1000 Mbps, 100 Mbps, or 10 Mbps.
- In addition to managing MAC and PHY layer functions, the controller provides a 32-bit wide direct Peripheral Component Interconnect (PCI) 2.3 compliant interface capable of operating at 33 or 66 MHz.

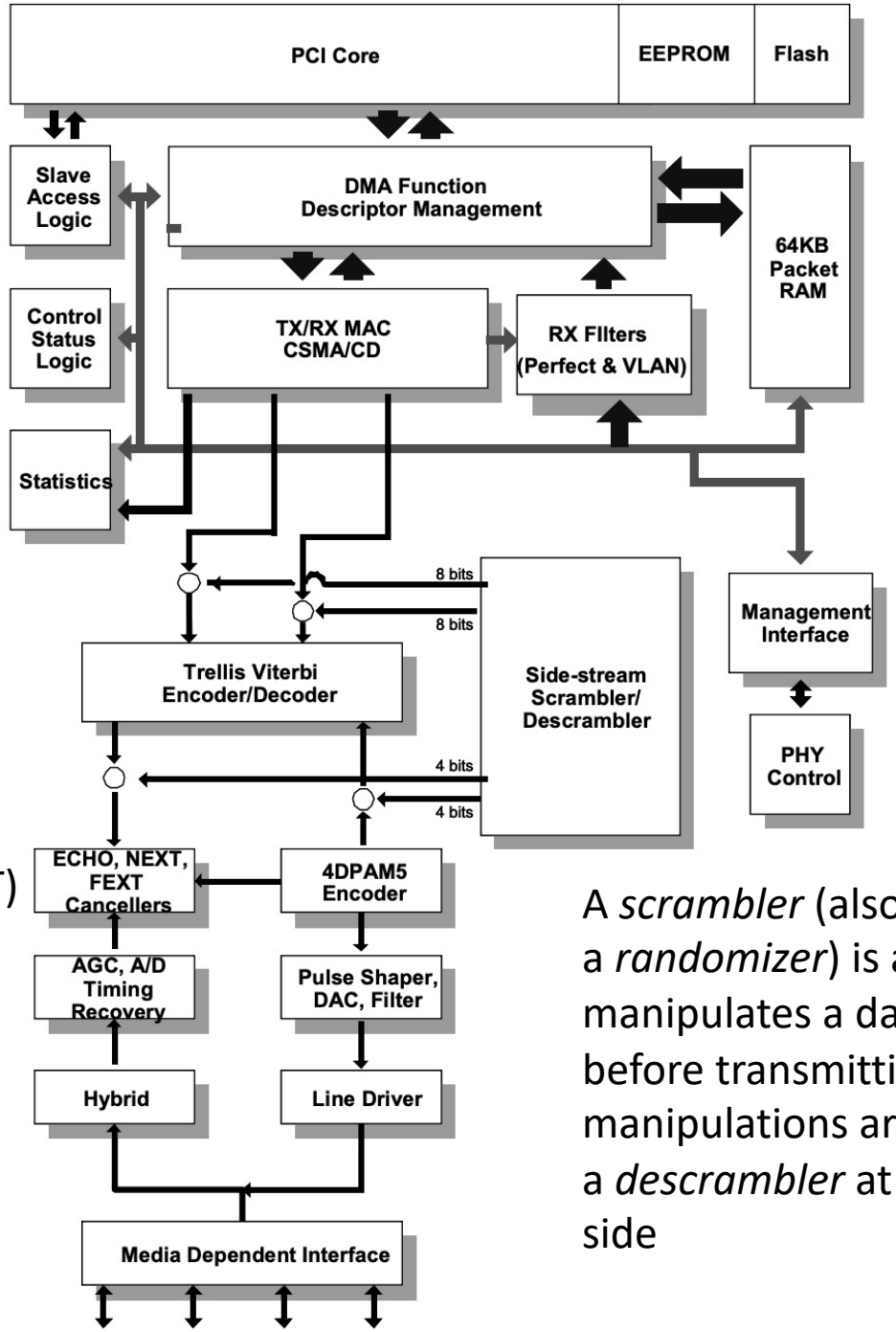
Datasheet

- PCI Bus
  - PCI revision 2.3, 32-bit, 33/66 MHz
  - Algorithms that optimally use advanced PCI, MWI, MRM, and MRL commands
  - 3.3 V (5 V tolerant PCI signaling)
- MAC Specific
  - Low-latency transmit and receive queues
  - IEEE 802.3x-compliant flow-control support with software-controllable thresholds
  - Caches up to 64 packet descriptors in a single burst
  - Programmable host memory receive buffers (256 B to 16 KB) and cache line size (16 B to 256 B)
  - Wide, optimized internal data path architecture
  - 64 KB configurable Transmit and Receive FIFO buffers
- PHY Specific
  - Integrated for 10/100/1000 Mb/s operation
  - IEEE 802.3ab Auto-Negotiation support
  - IEEE 802.3ab PHY compliance and compatibility
  - State-of-the-art DSP architecture implements digital adaptive equalization, echo cancellation, and cross-talk cancellation
  - Automatic polarity detection
  - Automatic detection of cable lengths and MDI vs. MDI-X cable at all speeds
- Host Off-Loading
  - Transmit and receive IP, TCP, and UDP checksum off-loading capabilities
  - Transmit TCP segmentation
  - Advanced packed filtering
  - Jumbo frame support up to 16 KB
  - Intelligent Interrupt generation (multiple packets per interrupt)
- Manageability
  - Network Device Class Power Management Specification 1.1
  - Compliance with PCI Power Management 1.1 and ACPI 2.0
  - SNMP and RMON statistic counters
  - D0 and D3 power states
- Additional Device Four programmable LED outputs
  - On-chip power control circuitry
  - BIOS LAN Disable pin
  - JTAG (IEEE 1149.1) Test Access Port built in silicon
- Lead-free a 196-pin Ball Grid Array (BGA). Devices that are lead-free are marked with a circled “e1” and have the product code: LUxxxxxx.

Convolutional coding is a coding method which is not based on blocks of bits but rather by logic operations on bit windows.

Near-end crosstalk (NEXT)
Far-end crosstalk (FEXT)
Echo cancellation

Automatic gain control (AGC)



Carrier-sense multiple access with collision detection (CSMA/CD)

A *scrambler* (also referred to as a *randomizer*) is a device that manipulates a data stream before transmitting. The manipulations are reversed by a *descrambler* at the receiving side

Networking

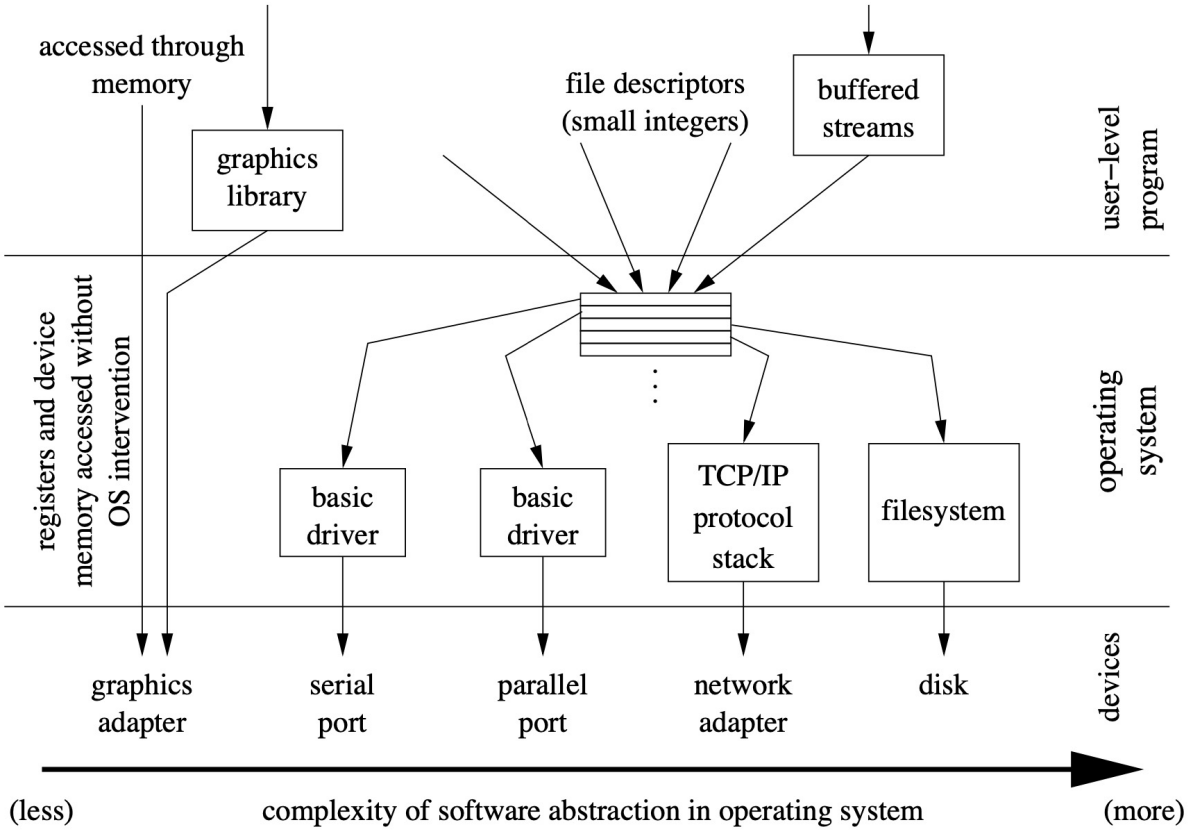
KERNEL NETWORK DEVICE DRIVERS

Device Drivers

- Kernel interacts with I/O devices by means of device drivers
  - operate at the kernel level
  - include data structures and functions that control one or more devises, e.g., keyboard, monitors, network interfaces
- Advantages:
  - device code can be encapsulated in a module
  - easy addition of new devices – no need to know the kernel source code
  - dynamic load/unload of device drivers

Example of Range of Device Driver Complexity

- Complexity grows from left to right



Range of Device Driver Complexity

- Leftmost example
  - no driver used (other than for obtaining/release access rights)
  - reads/writes go directly to device without OS intervention
  - lack of system calls improves performance
  - often used for graphics, and often with user-level libraries
- Remaining examples use file descriptor abstraction
  - sometimes with user-level libraries
  - all reduced to using file descriptors to do system calls
  - operating system
    - looks up file in array
    - uses file operations structure to hand off system call appropriately

Range of Device Driver Complexity

- Simple drivers such as serial port/parallel port
- Drivers for network adapters add state for protocol, kernel-side buffering
- Drivers for disks support block transfer, readahead, ….
  - usually used indirectly by file systems (inside kernel)
  - some user code (e.g., databases) uses raw disk commands

Kernel Abstraction for Devices – Device Files

- I/O devices are treated as special files called device files
  - same system calls used to interact with files on disk can be used to work with I/O devices
  - e.g., same write() used to
    - write to a regular file or
    - send data to a printer by writing to /dev/lp0 device file
- According to the characteristics of the underlying device drivers, device files can be of two types
  - block
  - character

Kernel Abstractions for Devices: Block Devices

- Block devices (underlies file systems)
  - data accessible only in blocks of fixed size, with size determined by device
  - can be addressed randomly
  - transfers to/from device are usually buffered (to) and cached (from) for performance
  - examples: disks, CD ROM, DVD

Kernel Abstractions for Devices: Character Devices

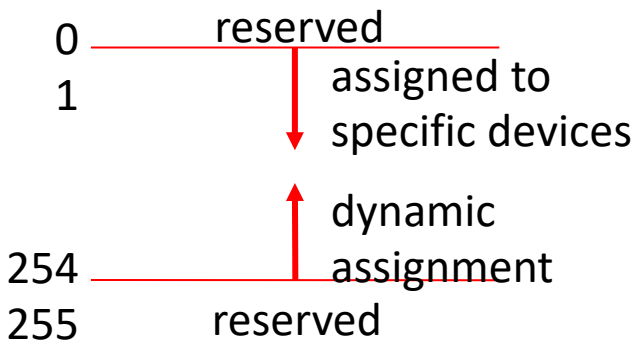
- Character device:
  - Almost everything else, except network cards
    - Network cards are not directly associated with device files
- A more constructive definition
  - contiguous space (or spaces) of bytes
  - some allow random access (e.g., magnetic tape driver)
  - others available only sequentially (e.g., sound card)
  - examples: keyboard, terminal, printers

Kernel Abstractions for Devices

- A device file is usually stored as a real file
  - Inode includes an identifier of the hardware device corresponding to the character or block device file
- Devices identified by major and minor numbers (traditionally both 8-bit)
  - major number is device type (e.g., specific model of disk, not just “disk”)
  - minor number is instance number (if driver allows you to have more than one of a given model attached to your computer)

Kernel Abstractions for Devices

- Major numbers each have
  - associated fops structure
  - name (see **`/proc/devices`**)



Example of device files:

Name	Type	Major	Minor	Description
/dev/hda	block	3	0	First IDE disk
/dev/hda2	block	3	2	Secondary/primary partion of first IDE disk
/dev/tty0	char	3	0	Terminal

Kernel Abstractions for Devices

- Registering and unregistering a device (see linux/fs.h)

```
int register_chrdev (unsigned int major,
     const char* name, struct file_operations* fops);
```

  - to request a specific major #, pass it as input argument
    - returns 0 on success
    - returns negative value on failure
  - for a dynamically assigned major #, pass 0 as input argument (major)
    - returns assigned major # on success (not 0!)
    - returns negative value on failure

Kernel Abstractions for Devices

```
int unregister_chrdev (unsigned int major,
                       const char* name);
```

- both parameters must match those of registration call
- returns 0 on success
- returns negative value on failure

Kernel Abstractions for Devices

To create a special device file for accessing a device, type (as root):

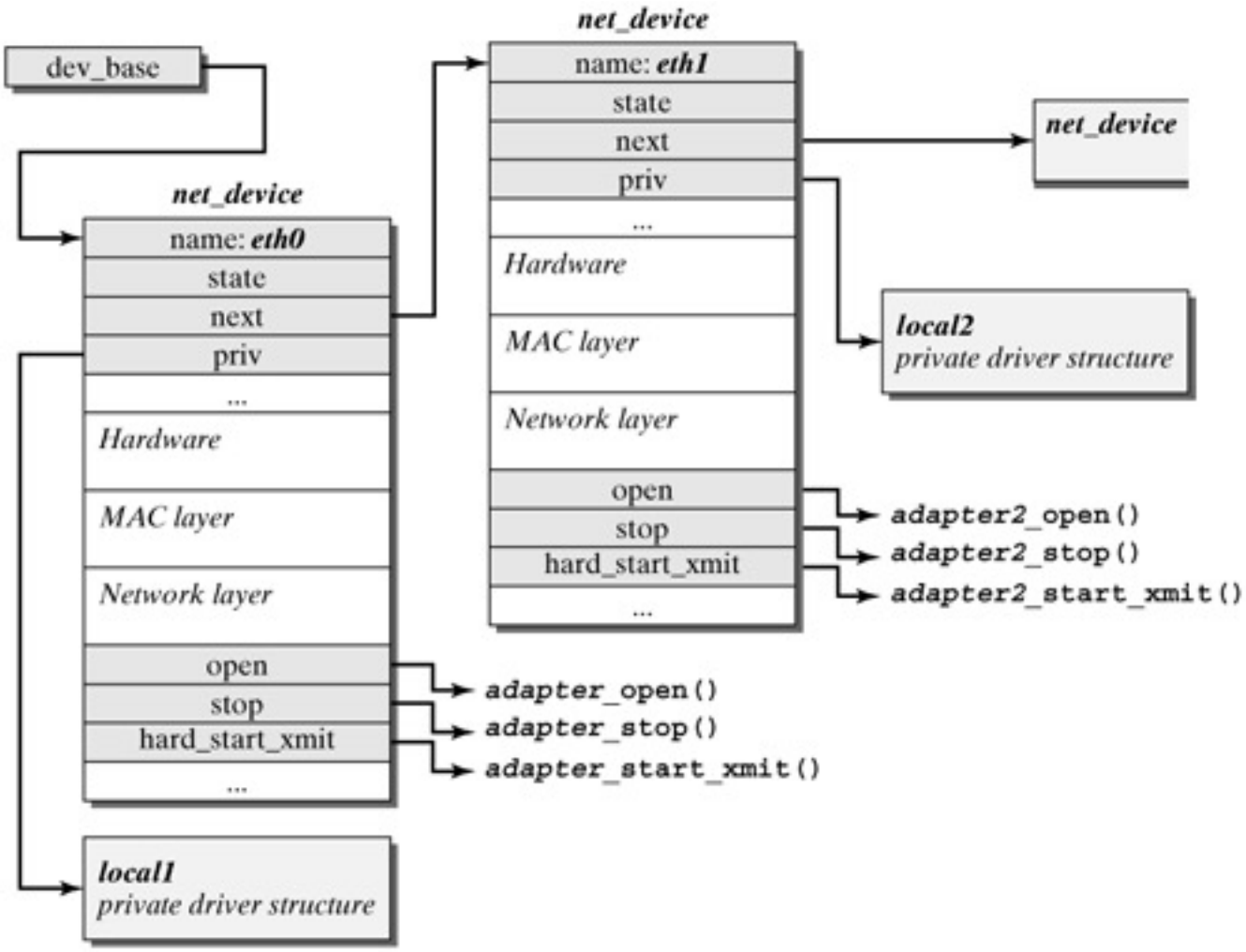
`mknod <file> c <major> <minor>`

- When open is called
  - major/minor #'s come in as **`inode->i_rdev`** **`(real device identifier)`**
  - can be split using **`MAJOR()`** and **`MINOR()`** macros
  - or **`imajor`**/**`iminor`** functions (argument is inode pointer)

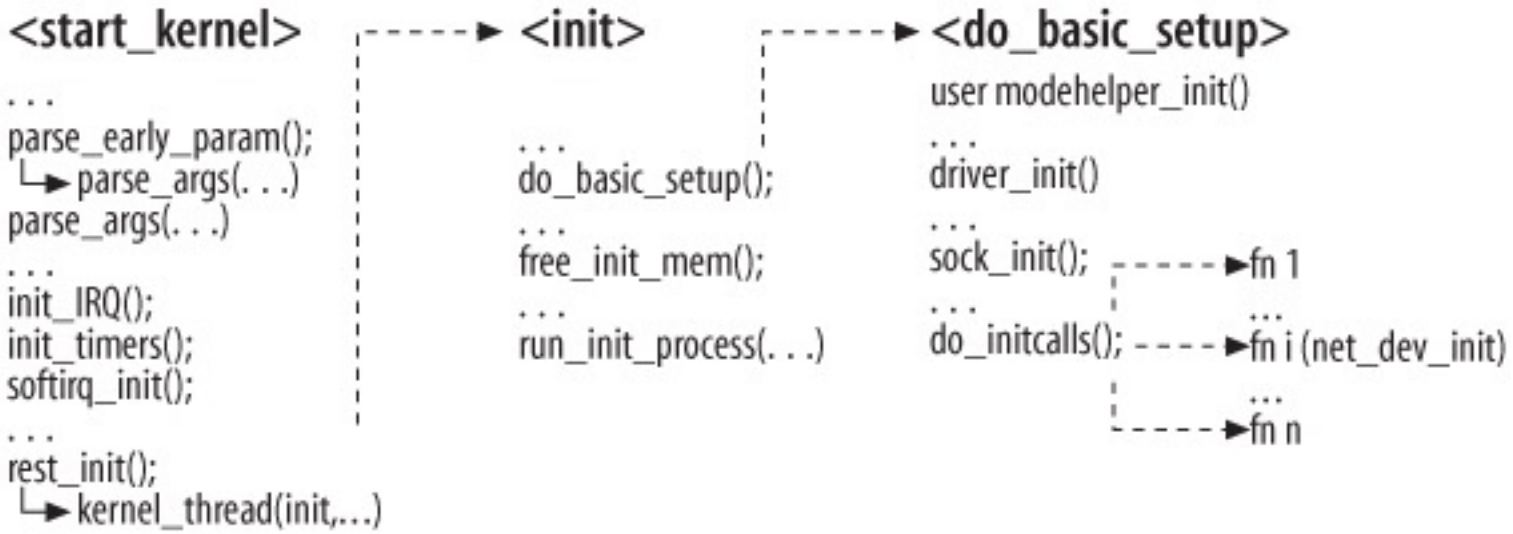
Recall data structures

- ***struct socket***
  - A network socket is a software structure within a network node of a computer network that serves as an endpoint for sending and receiving data across the network. The structure and properties of a socket are defined by an application programming interface (API) for the networking architecture. Sockets are created only during the lifetime of a process of an application running in the node.
- ***struct sk_buff***
  - This is where a packet is stored. The structure is used by all the network layers to store their headers, information about the user data (the payload), and other information needed internally for coordinating their work.
- ***struct net_device***
  - Each network device is represented in the Linux kernel by this data structure, which contains information about both its hardware and its software configuration.

Understanding Linux network internals. Benvenuti, Christian.


dev_base
net_device
name: eth0
state
next
priv
...
Hardware
MAC layer
Network layer
open
stop
hard_start_xmit
...
local1
private driver structure
adapter_open()
adapter_stop()
adapter_start_xmit()
net_device
name: eth1
state
next
priv
...
Hardware
MAC layer
Network layer
open
stop
hard_start_xmit
...
net_device
local2
private driver structure
adapter2_open()
adapter2_stop()
adapter2_start_xmit()

Kernel initialization

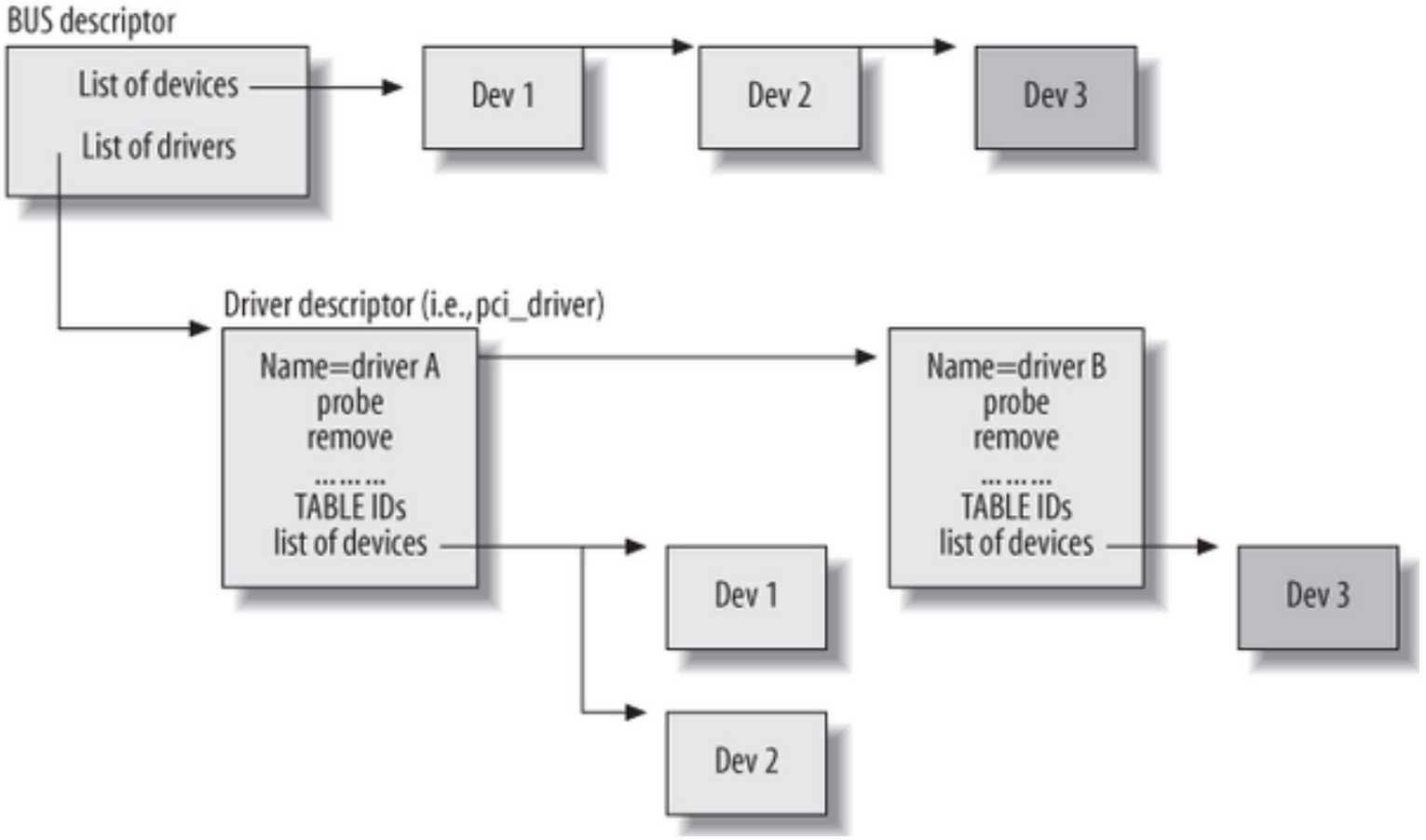


Boot-time options
Interrupts and timers
Initialization

PCI BUS

- When the system boots, it creates a sort of database that associates each bus to a list of detected devices that use the bus
  - For example, the descriptor for the PCI bus includes, among other parameters, a list of detected PCI devices
  - When device driver A is loaded, it registers with the PCI layer and providing its instance of pci_driver.
  - The pci_driver structure includes a vector with the IDs of those PCI devices it can drive.
  - The PCI layer then uses that table to see what devices match in its list of detected PCI devices

Binding between bus and drivers, and between driver and devices



Interaction Between Devices and Kernel

- Nearly all devices (including NICs) interact with the kernel in one of two ways:
  - Polling
    - Driven on the kernel side. The kernel checks the device status at regular intervals to see if it has anything to say.
  - Interrupt
    - Driven on the device side. The device sends a hardware signal (by generating an interrupt) to the kernel when it needs the kernel's attention.

NAPI

- The main idea behind NAPI is simple: instead of using a pure interrupt-driven model, it uses a mix of interrupts and polling.
- If new frames are received when the kernel has not finished handling the previous ones yet, there is no need for the driver to generate other interrupts.
- it is just easier to have the kernel keep processing whatever is in the device input queue (with interrupts disabled for the device), and re-enable interrupts once the queue is empty. This way, the driver reaps the advantages of both interrupts and polling:
  - Asynchronous events, such as the reception of one or more frames, are indicated by interrupts so that the kernel does not have to check continuously if the device's ingress queue is empty.
  - If the kernel knows there is something left in the device's ingress queue, there is no need to waste time handling interrupt notifications. A simple polling is enough.

Interrupt types

- With an interrupt, an NIC can tell its driver several different things. Among them are:
  - **Reception of a frame**
    - This is the most common and standard situation.
  - **Transmission failure**
    - This kind of notification is generated on Ethernet devices only after a feature called exponential binary backoff has failed (this feature is implemented at the hardware level by the NIC). Note that the driver will not relay this notification to higher network layers; they will come to know about the failure by other means (timer timeouts, negative ACKs, etc.).
  - **DMA transfer has completed successfully**
    - Given a frame to send, the buffer that holds it is released by the driver once the frame has been uploaded into the NIC's memory for transmission on the medium. With synchronous transmissions (no DMA), the driver knows right away when the frame has been uploaded on the NIC. But with DMA, which uses asynchronous transmissions, the device driver needs to wait for an explicit interrupt from the NIC.
  - **Device has enough memory to handle a new transmission**
    - It is common for an NIC device driver to disable transmissions by stopping the egress queue when that queue does not have sufficient free space to hold a frame of maximum size (e.g., 1,536 bytes for an Ethernet NIC). The queue is then re-enabled when memory becomes available.

Interrupt - reception of a frame

- The interrupt handler performs a few immediate tasks and schedules others in a bottom half to be executed later. Specifically, the interrupt handler:
  - Copies the frame into an sk_buff data structure.
  - Initializes some of the sk_buff parameters for use later by upper network layers
  - Updates some other parameters private to the device.
  - Signals the kernel about the new frame by scheduling the NET_RX_SOFTIRQ softirq for execution.