Lecture 18: Networking

CS 105

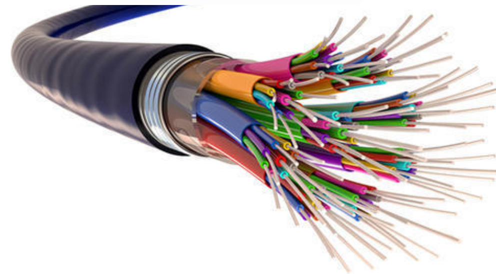
November 7, 2019

Physical Layer

- Twisted Pair

- Coaxial

- Fiber

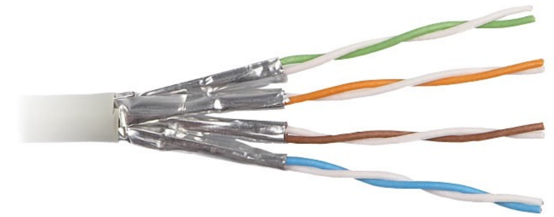


- Radio

Data Link Layer

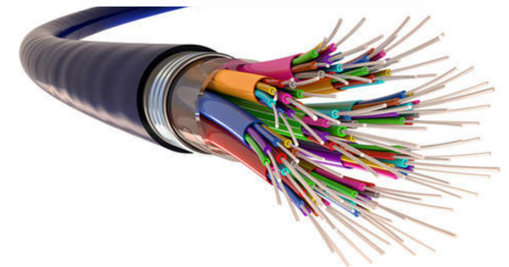
- DSL
- Ethernet
- WiFi (802.11)
- 3G
- LTE
- 5G

Twisted Pair

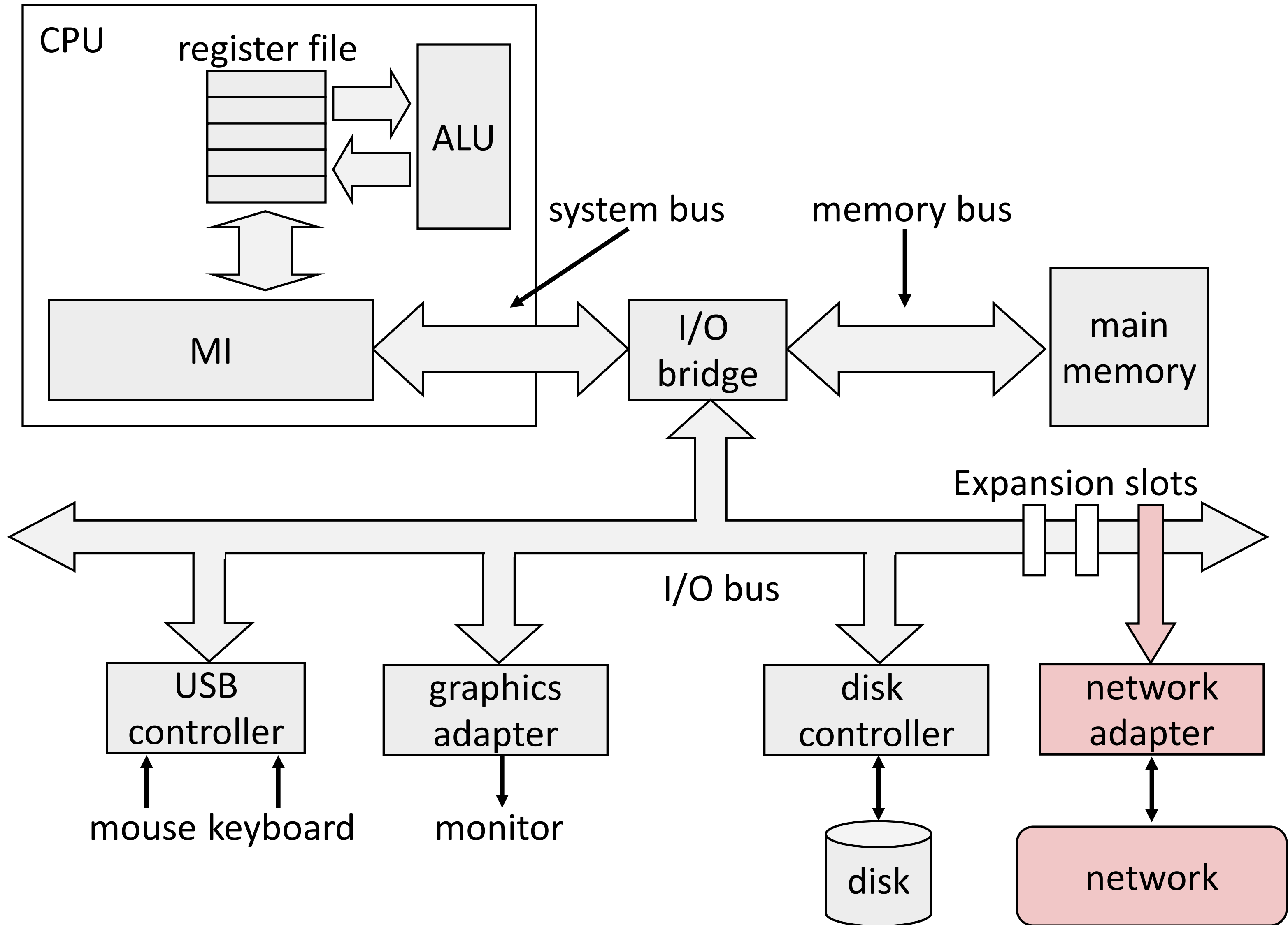


Coaxial

Fiber



Radio


CPU
register file
ALU
MI
system bus
memory bus
I/O
bridge
main
memory
Expansion slots
I/O bus
USB
controller
graphics
adapter
disk
controller
network
adapter
mouse keyboard
monitor
disk
network

Data Link Layer

- Each host has one or more network adapter (aka NIC)
  - handles particular physical layer and protocol
- Each network adapter has a media access control (MAC) address
  - unique to that networ instance
- Messages are organized as packets

Example: Ethernet

- Developed 1976 at Xerox
- Simple, scales pretty well
- Very successful, still in widespread use

- Example address: b8:e3:56:15:6a:72

- **Carrier sense:** listen before you speak
- **Multiple access:** multiple hosts on network
- **Collision detection:** detect and respond to cases where two messages collide

Example: Ethernet

- Carrier sense: broadcast if wire is available
- In case of collision: stop, sleep, retry
  - sleep time is determined by collision number
  - abort after 16 attempts

Example: Ethernet

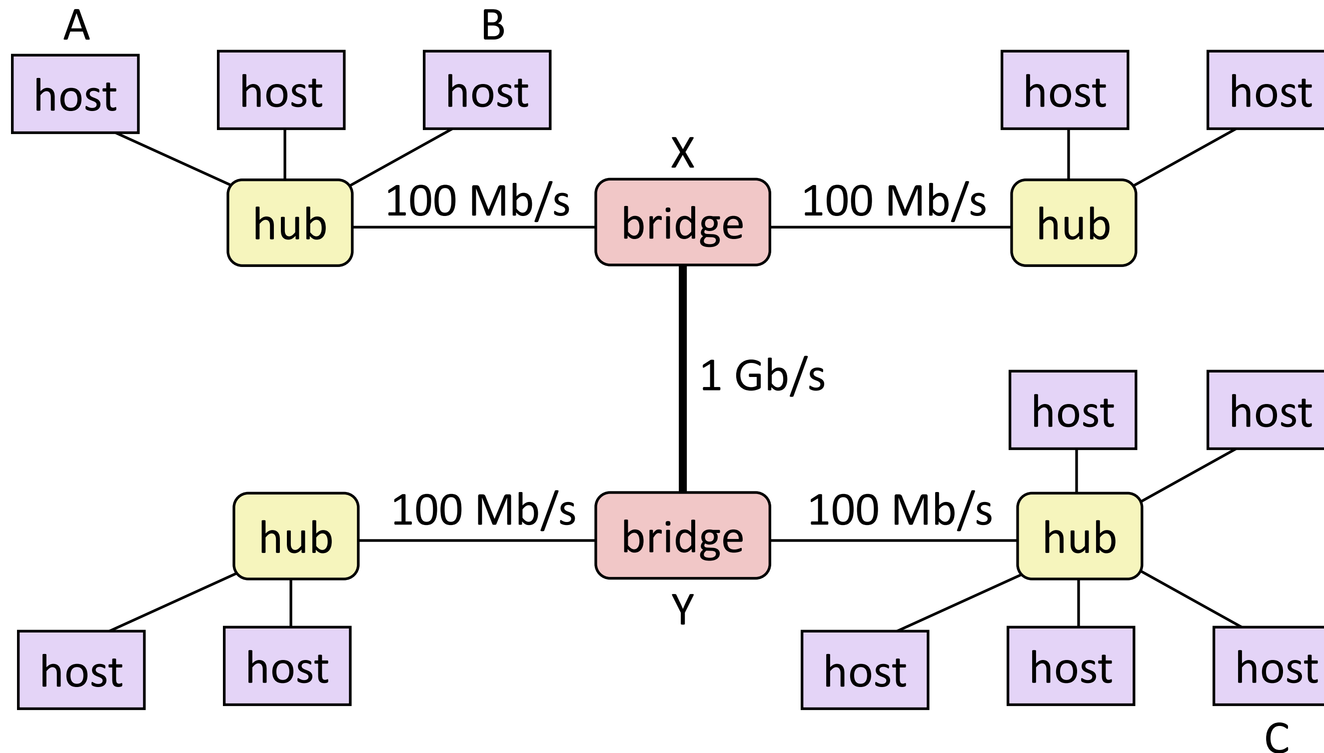
Advantages

- completely decentralized
- inexpensive
  - no state in the network
  - no arbiter
  - cheap physical links

Disadvantages

- endpoints must be trusted
- data is available for all to see
  - can place ethernet card in promiscuous mode and listen to all messages

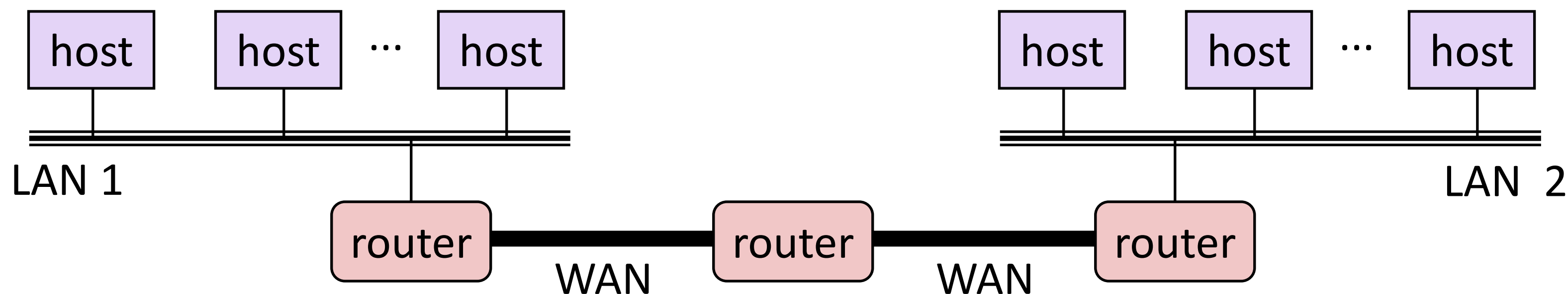
Bridged Ethernet



- Spans building or campus
- Bridges cleverly learn which hosts are reachable from which ports and then selectively copy frames from port to port

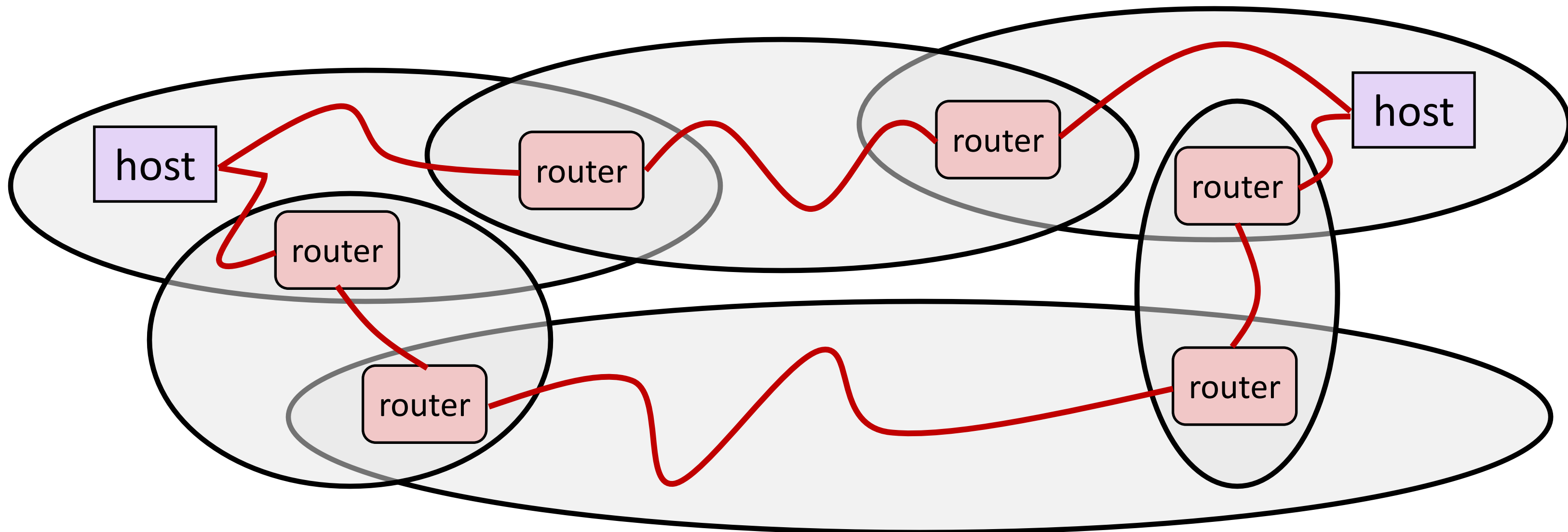
Network Layer

- There are lots of lots of local area networks (LANs)
  - each determines its own protocols, address format, packet format
- What if we wanted to connect them together?
  - physically connected by specialized computers called routers
  - routers with multiple network adapters can translate
  - standardize address and packet formats



- This is a internetwork
  - aka wide-area network (WAN)
  - aka internet

Logical Structure of an internet



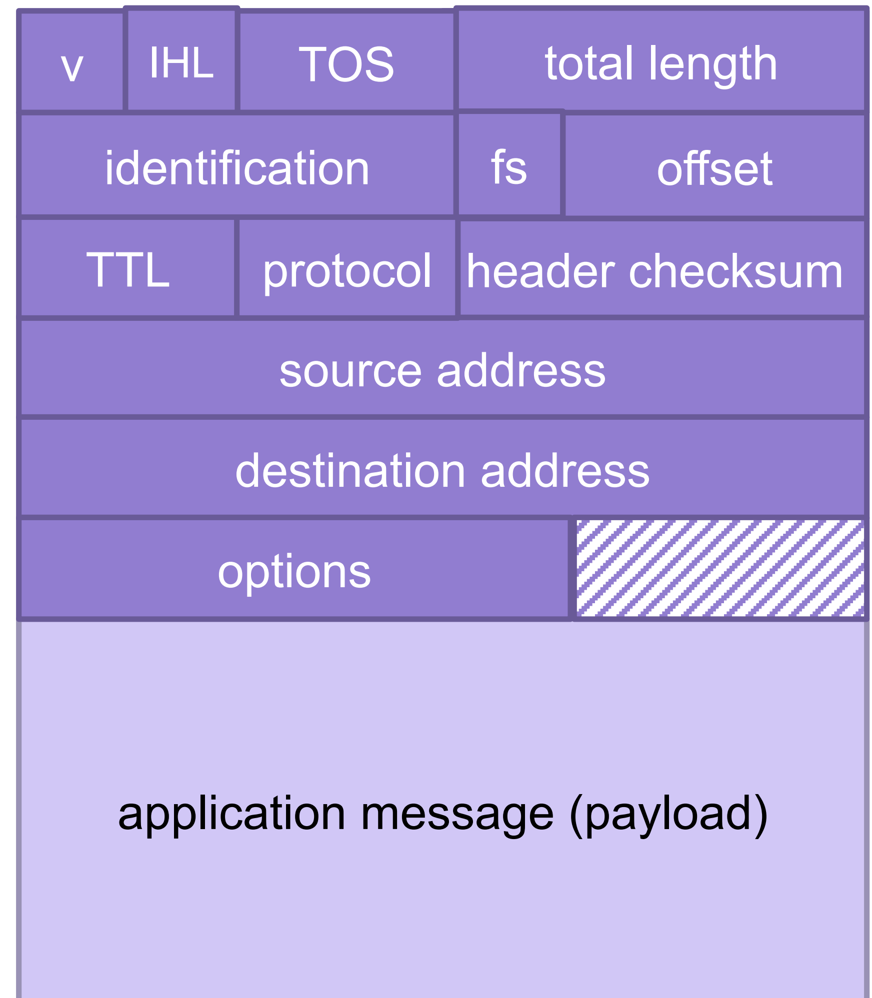
- Ad hoc interconnection of networks
  - No particular topology
  - Vastly different router & link capacities
- Send packets from source to destination by hopping through networks
  - Router forms bridge from one network to another
  - Different packets may take different routes

Internet Protocol (IP)

- Initiated by the DoD in 60s-70s
- Currently transitioning (very slowly) from IPv4 to IPv6

- Example address: 128.84.12.43

- interoperable
- network dynamically routes packets from source to destination



Aside: IPv4 and IPv6

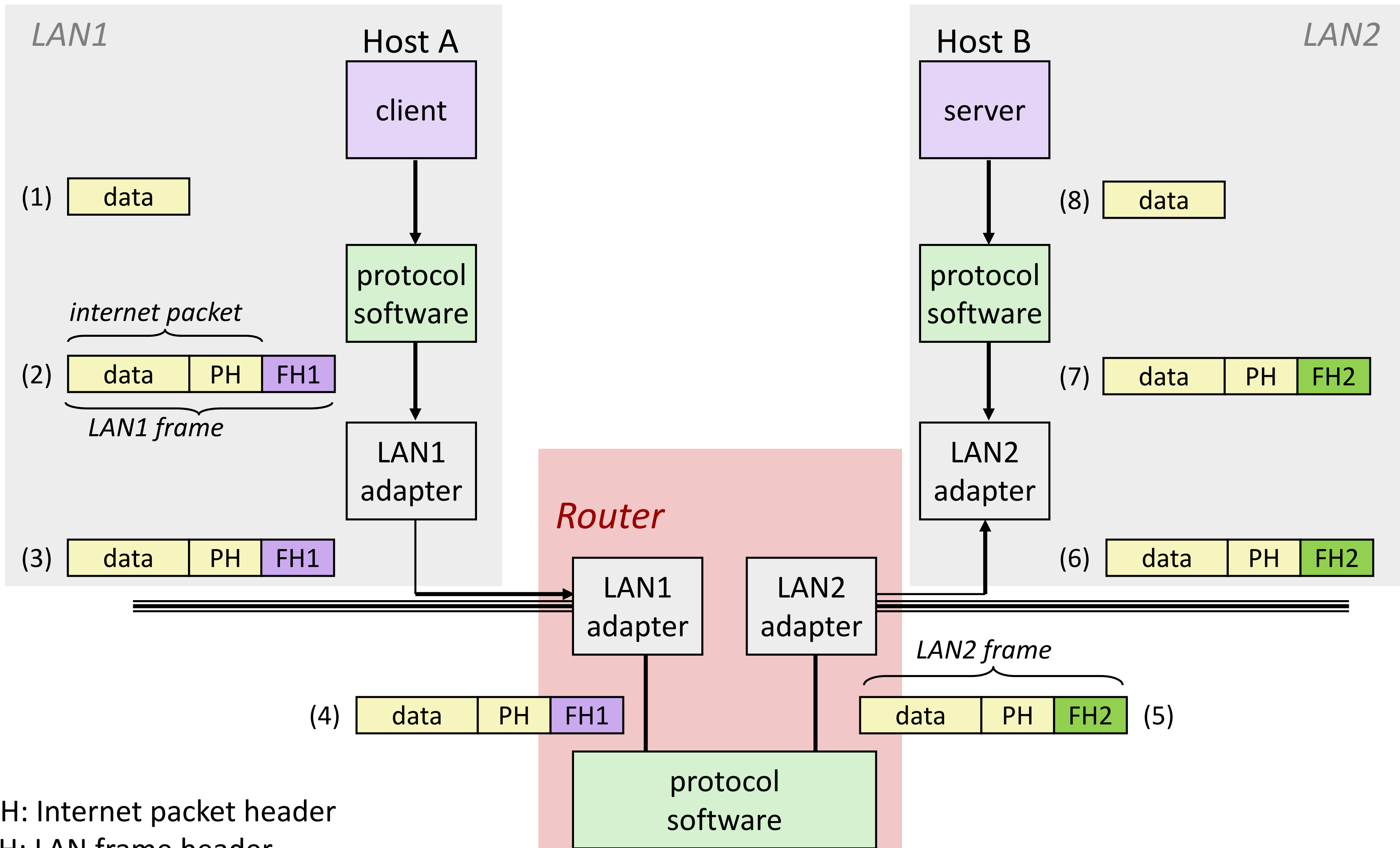
- The original Internet Protocol, with its 32-bit addresses, is known as *Internet Protocol Version 4* (IPv4)

- 1996: Internet Engineering Task Force (IETF) introduced *Internet Protocol Version 6* (IPv6) with 128-bit addresses
  - Intended as the successor to IPv4

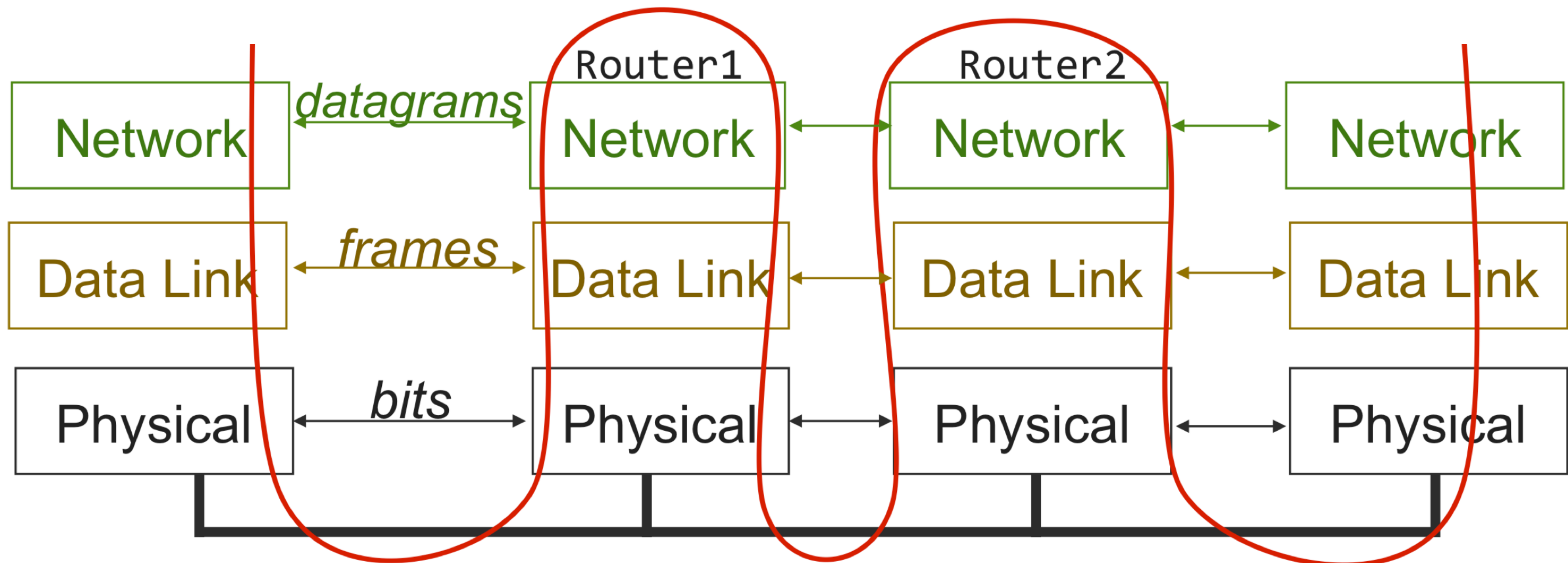
- As of November 2019, majority of Internet traffic still carried by IPv4
  - 24-29% of users access Google services using IPv6.

- We will focus on IPv4, but will show you how to write networking code that is protocol-independent.

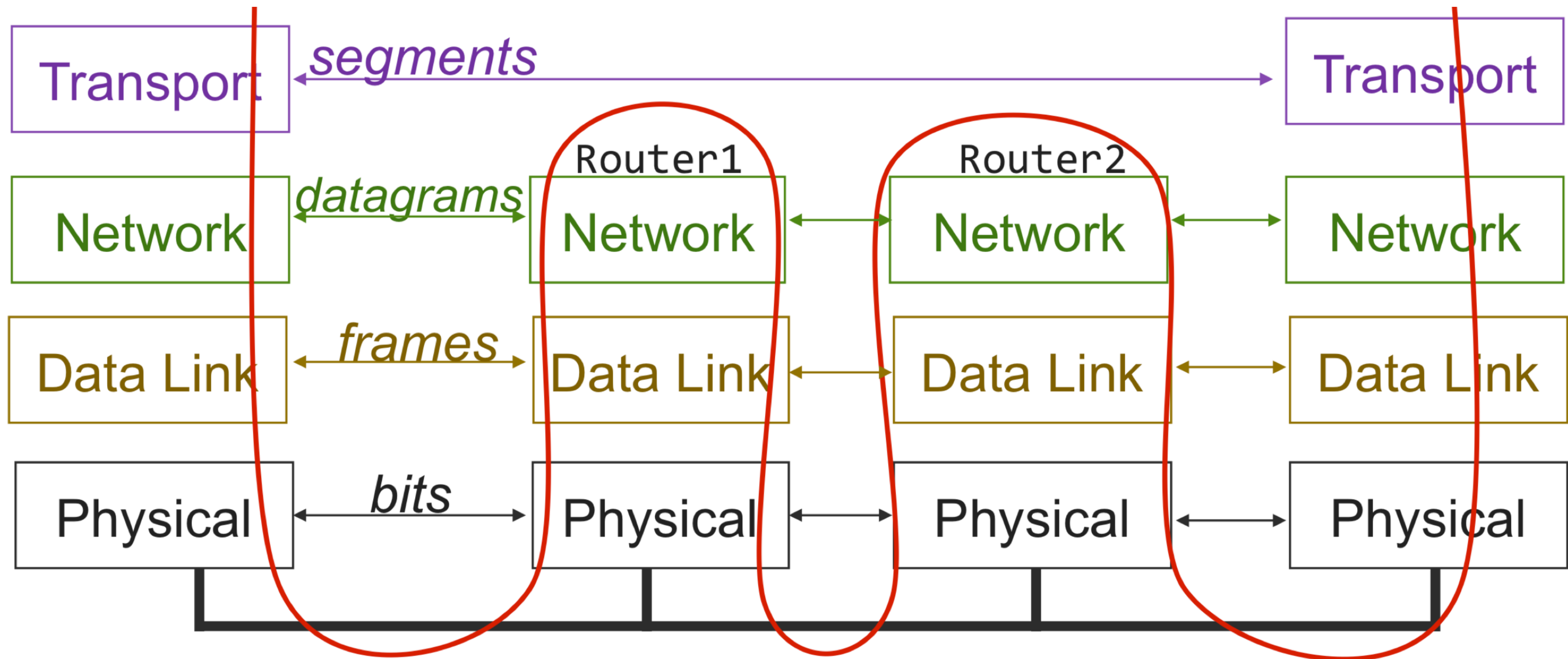
Transferring internet Data Via Encapsulation



Routing



Transport Layer



Transport Layer

- Clients and servers communicate by sending streams of bytes over a **connection**.

- A transport layer endpoint is identified by an **IP address** and a **port**, a 16-bit integer that identifies a process
  - Ephemeral port: Assigned automatically by client kernel when client makes a connection request.
  - Well-known port: Associated with some service provided by a server (e.g., port 80 is associated with Web servers)

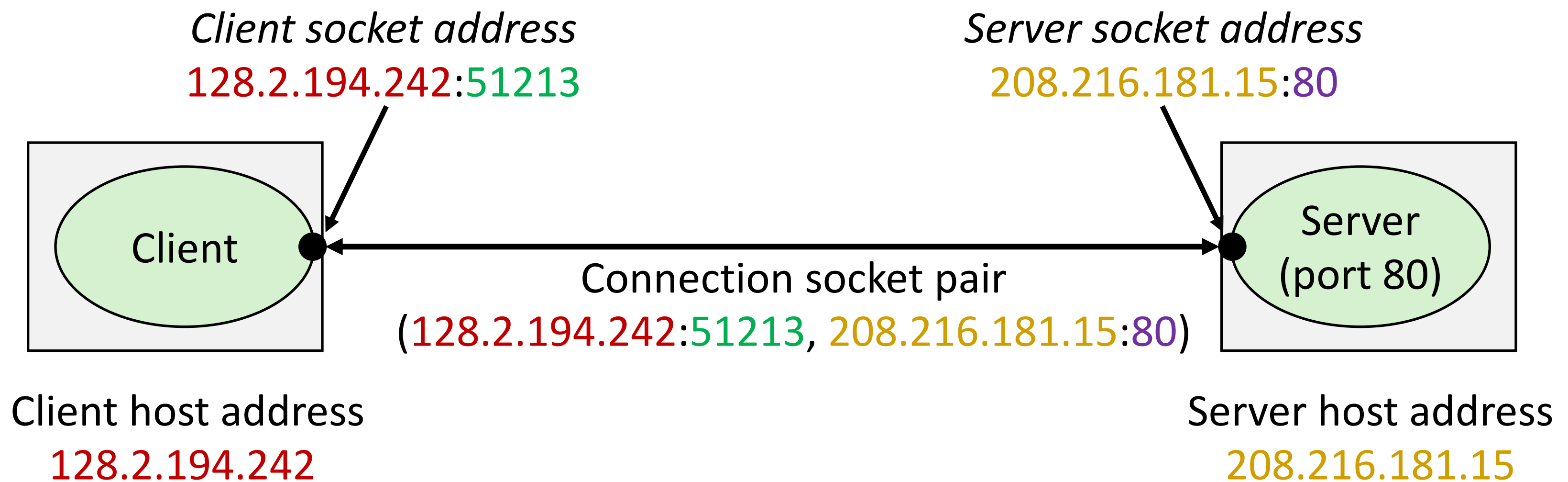
Well-known Ports and Service Names

- Popular services have permanently assigned **well-known ports** *and* corresponding ***well-known service names***:
  - echo server: 7/echo
  - ssh servers: 22/ssh
  - email server: 25/smtp
  - Web servers: 80/http

- Mappings between well-known ports and service names is contained in the file `/etc/services` on each Linux machine.

Anatomy of a Connection

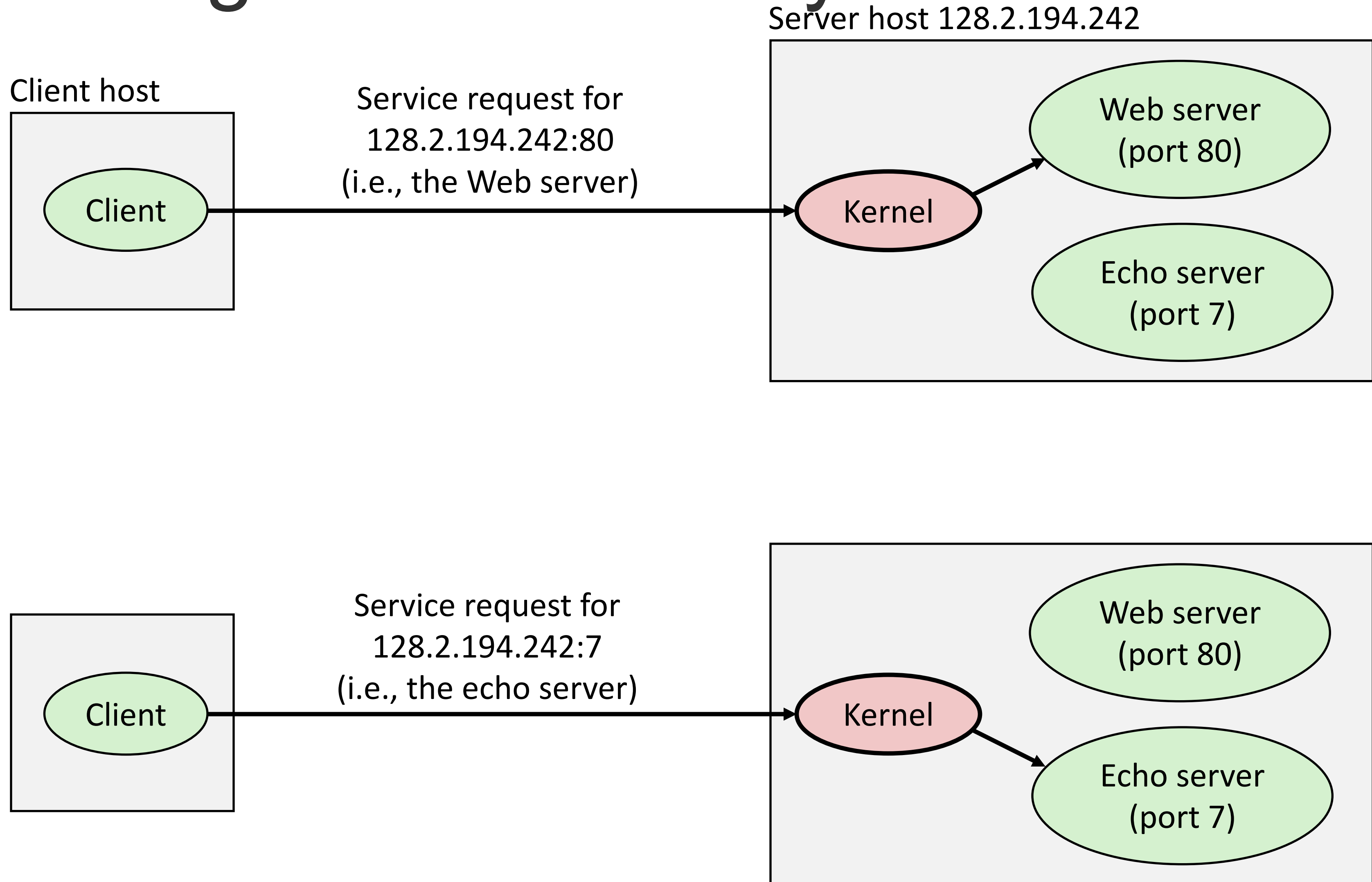
- A connection is uniquely identified by the socket addresses of its endpoints (*socket pair*)
  - `(cliaddr:cliport, servaddr:servport)`



51213 is an ephemeral port allocated by the kernel

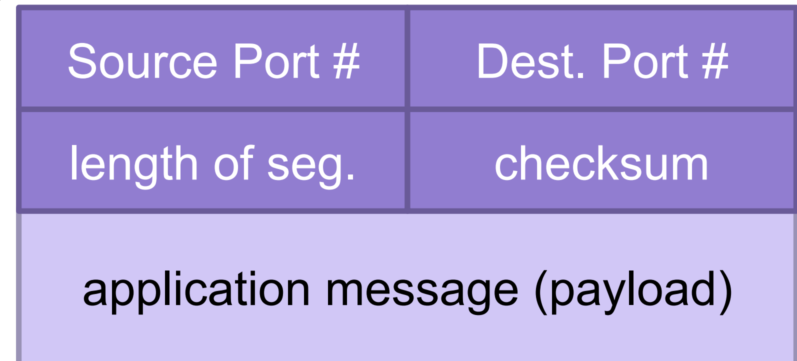
80 is a well-known port associated with Web servers

Using Ports to Identify Services

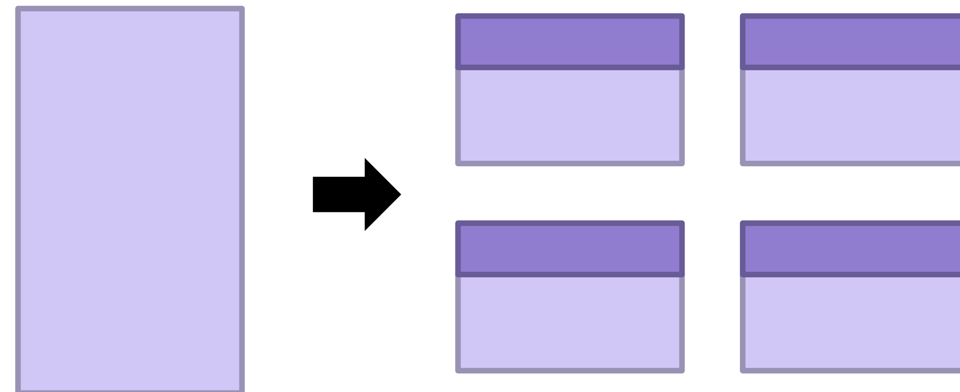


Transport Layer Segments

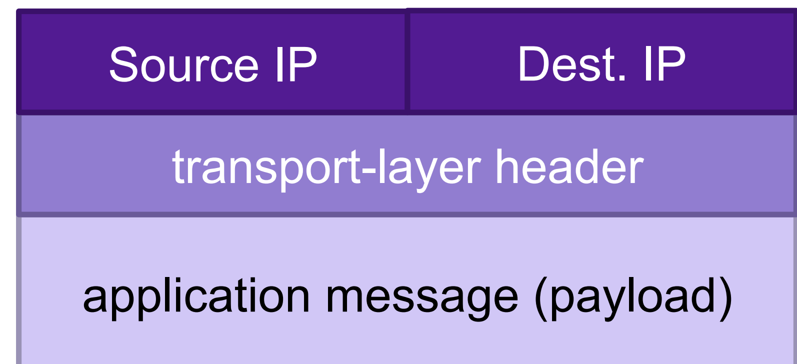
- Sending application:
  - specifies IP address and port
  - uses socket bound to source port



- Transport Layer:
  - breaks application message into smaller chunks
  - adds transport-layer header to each message to form a segment



- Network Layer (IP):
  - adds network-layer header to each datagram



Should the transport layer guarantee packet delivery?

Transport Layer Protocols

User Datagram Protocol (UDP)

- **unreliable, unordered delivery**
- connectionless
- best-effort, segments might be lost, delivered out-of-order, duplicated
- reliability (if required) is the responsibility of the app

Transmission Control Protocol (TCP)

- **reliable, inorder delivery**
- connection setup
- flow control
- congestion control

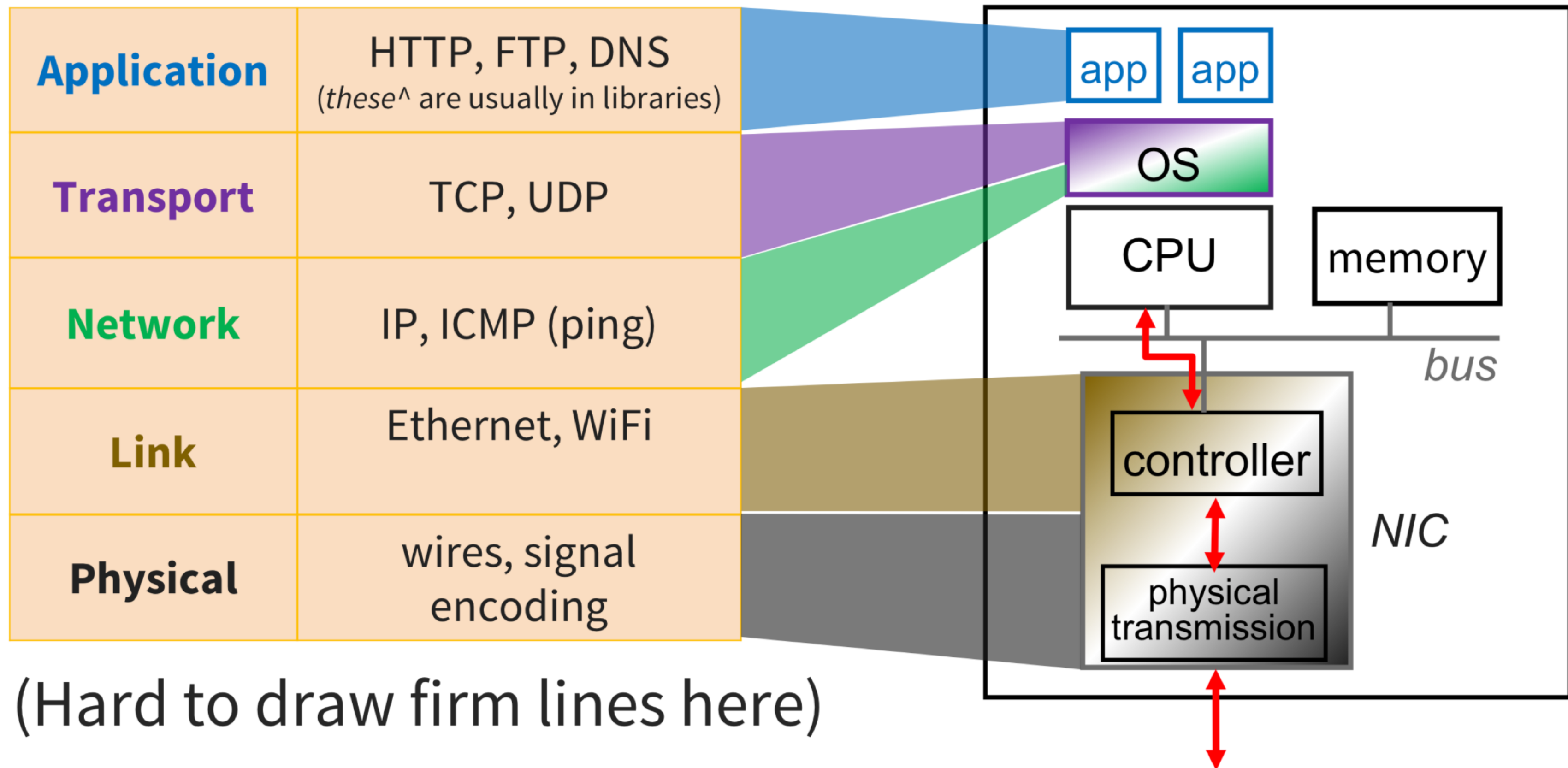
UDP: tradeoffs

- fast:
  - no connection setup
  - no rate-limiting
- simple:
  - no connection state
  - small header (8 bytes)
- (possibly) extra work for applications
  - reordering
  - duplicate suppression
  - handle missing packets

Transport Protocols by Application

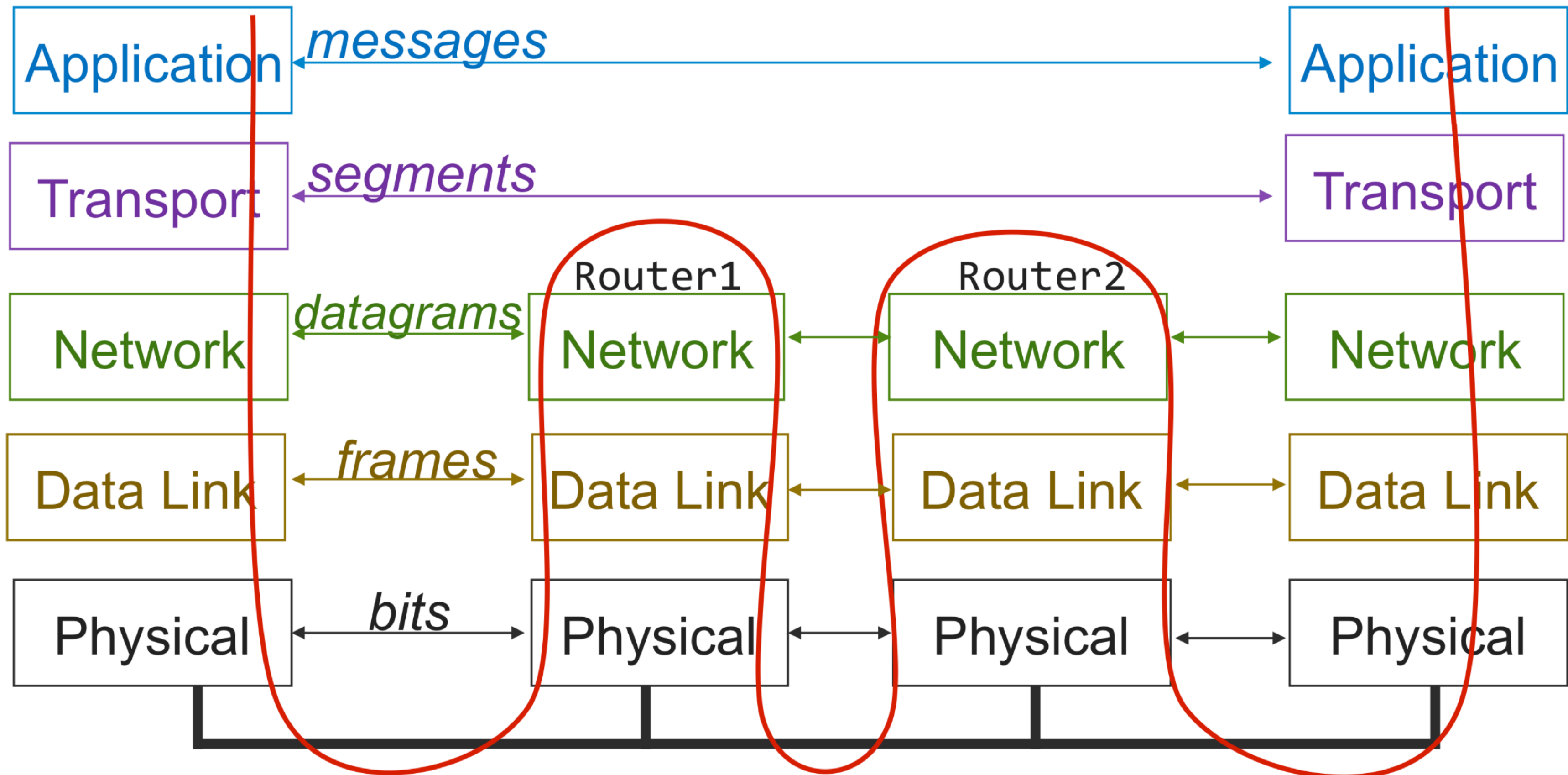
Application	Application-Level Protocol	Transport Protocol
Name Translation	DNS	Typically UDP
Routing Protocol	RIP	Typically UDP
Network Management	SNMP	Typically UDP
Remote File Server	NFS	Typically UDP
Streaming multimedia	(proprietary)	UDP or TCP
Internet telephony	(proprietary)	UDP or TCP
Remote terminal access	Telnet	TCP
File Transfer	(S)FTP	TCP
Email	SMTP	TCP
Web	HTTP(S)	TCP

Hardware and Software Interfaces



(Hard to draw firm lines here)

The Big Picture



Basic Network Abstraction

- A process can create "endpoints" to talk to other processes (possibly on other machines)
- Each endpoint has a unique address

- A message is a byte array
- Processes can:
  - send messages on endpoints
  - receive messages on endpoints

Sockets

- What is a socket?
  - IP address + port
  - To the kernel, a socket is an endpoint of communication
  - To an application, a socket is a file descriptor that lets the application read/write from/to the network
    - ***Note:*** All Unix I/O devices, including networks, are modeled as files
- Clients and servers communicate with each other by reading from and writing to socket descriptors

- The main distinction between regular file I/O and socket I/O is how the application “opens” the socket descriptors

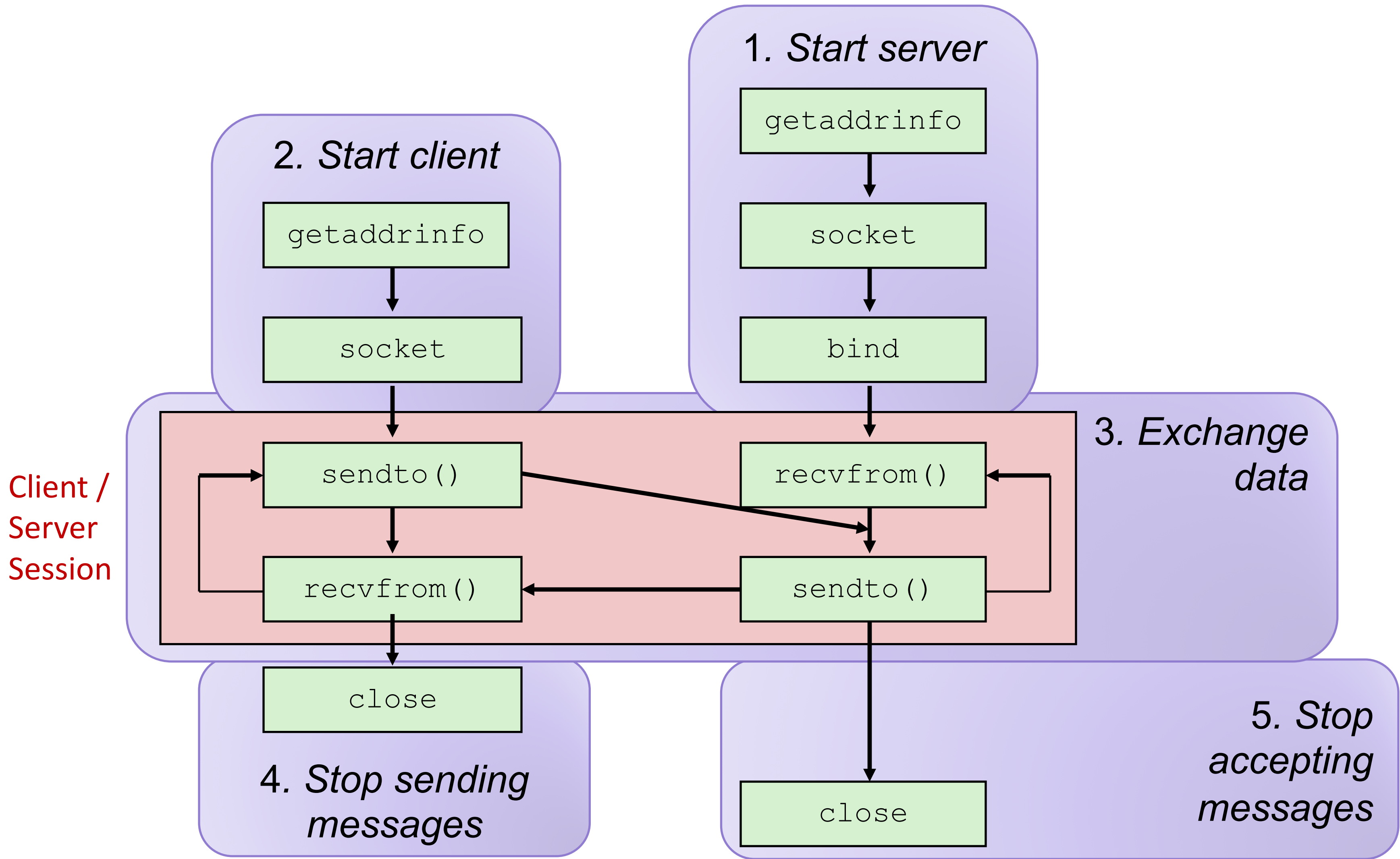
Sockets Interface

- Set of system-level functions used in conjunction with Unix I/O to build network applications.

- Created in the early 80’s as part of the original Berkeley distribution of Unix that contained an early version of the Internet protocols.

- Available on all modern systems
  - Unix variants, Windows, OS X, IOS, Android, ARM

Sockets Interface (UDP)



Host and Service Conversion: `getaddrinfo`

- `getaddrinfo` is the modern way to convert string representations of hostnames, host addresses, ports, and service names to socket address structures.
  - Replaces obsolete `gethostbyname` and `getservbyname` funcs.

- Advantages:
  - Reentrant (can be safely used by threaded programs).
  - Allows us to write portable protocol-independent code
    - Works with both IPv4 and IPv6

- Disadvantages
  - Somewhat complex
  - Fortunately, a small number of usage patterns suffice in most cases.

Sockets Interface: `socket`

- Clients and servers use the `socket` function to create a *socket descriptor*:

```
int socket(int domain, int type, int protocol)
```

- Example:

```
int clientfd = Socket(AF_INET, SOCK_STREAM, 0);
```

Indicates that we are using 32-bit IPV4 addresses

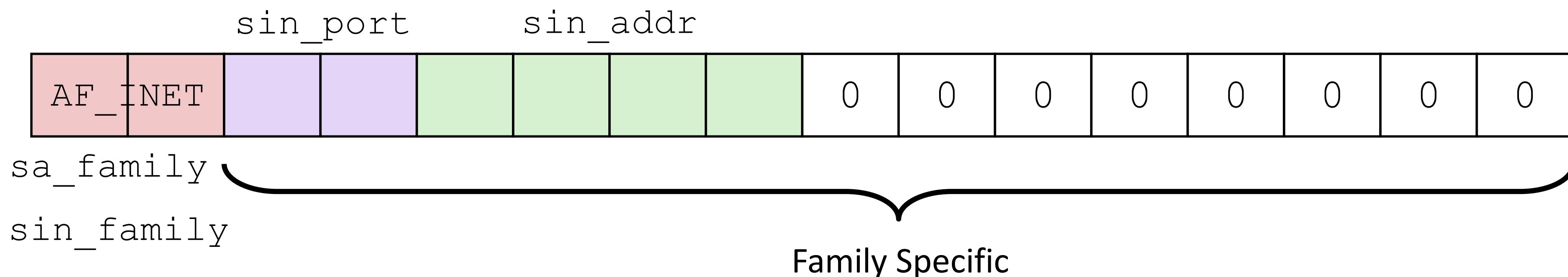
Indicates that the socket will be the end point of a connection

Protocol specific! Best practice is to use `getaddrinfo` to generate the parameters automatically, so that code is protocol independent.

Socket Address Structures

- Internet-specific socket address:
  - Must cast (`struct sockaddr_in *`) to (`struct sockaddr *`) for functions that take socket address arguments.

```
struct sockaddr_in  {
  uint16_t         sin_family;  /* Protocol family (always AF_INET) */
  uint16_t         sin_port;    /* Port num in network byte order */
  struct in_addr   sin_addr;    /* IP addr in network byte order */
  unsigned char    sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};
```



Sockets Interface: bind

- A server uses `bind` to ask the kernel to associate the server's socket address with a socket descriptor:

```
int bind(int sockfd, SA *addr, socklen_t addrlen);
```

- The process can read bytes that arrive on the connection whose endpoint is `addr` by reading from descriptor `sockfd`.
- Similarly, writes to `sockfd` are transferred along connection whose endpoint is `addr`.

Best practice is to use `getaddrinfo` to supply the arguments `addr` and `addrlen`.