HARVARD
School of Engineering
and Applied Sciences

Network Programming with Sockets

CS61, Lecture 23

Prof. Stephen Chong

November 22, 2011

Announcements

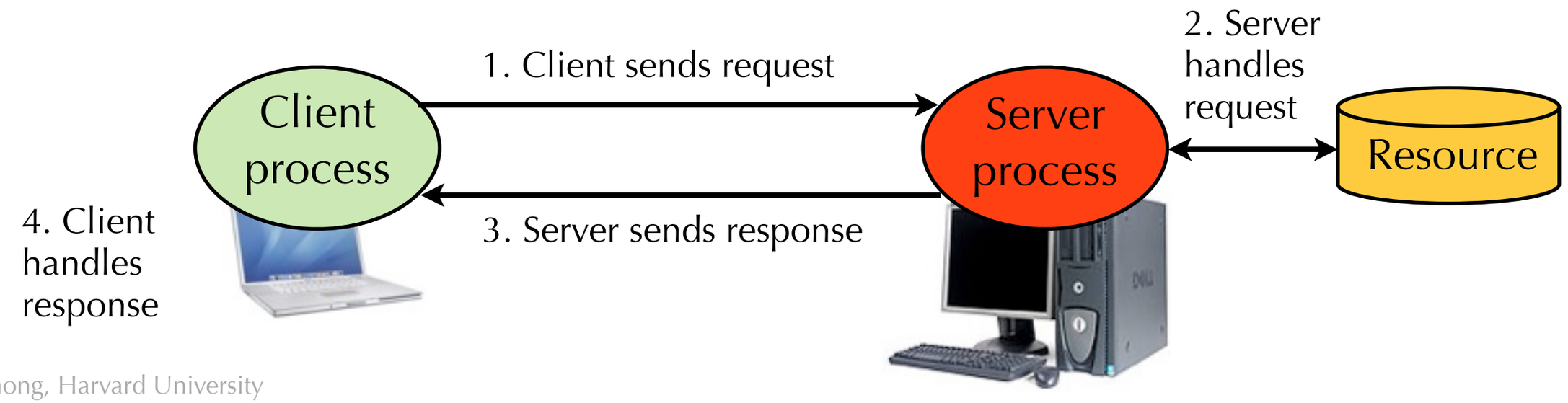
- No class Thursday; Happy Thanksgiving
- Final Thursday Dec 1
  - Practice exams posted on iSites
  - Section next week will cover the 2010 practice exam

Today

- Clients and servers
- Networks
  - Circuit vs packet
  - Ethernet
  - Internets
  - Protocol stack and TCP
- Network programming
  - Client's view
  - Server's view

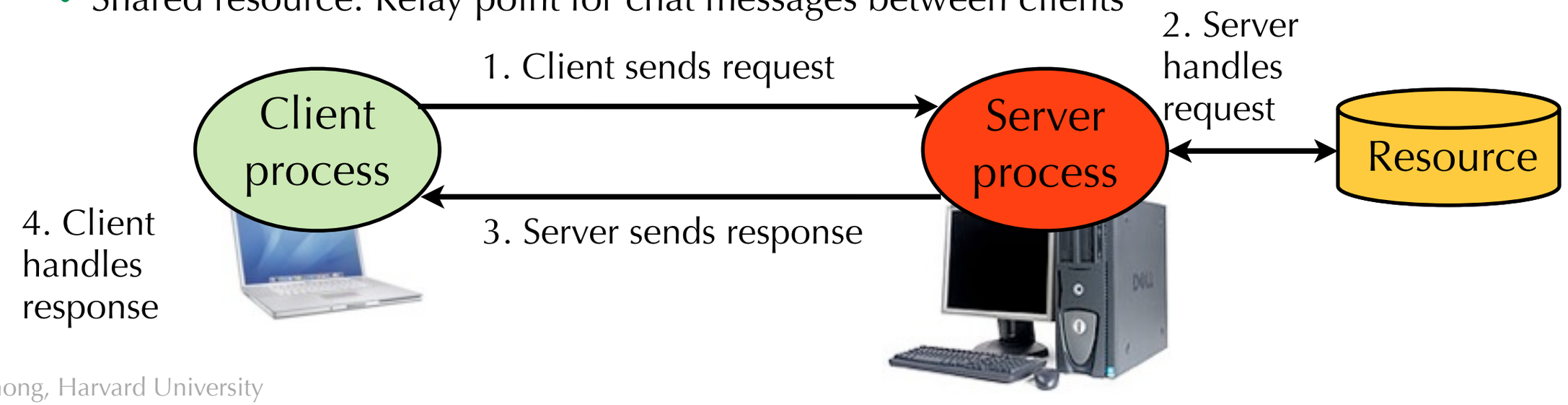
Clients and servers

- Most network applications are based on the client-server model:
  - A **server process** and one or more **client processes**
  - Server manages some **resource**
  - Server provides service by manipulating resource for clients
  - Server and client communicate by some **predefined protocol**
  - Server activated by request from client (vending machine analogy)



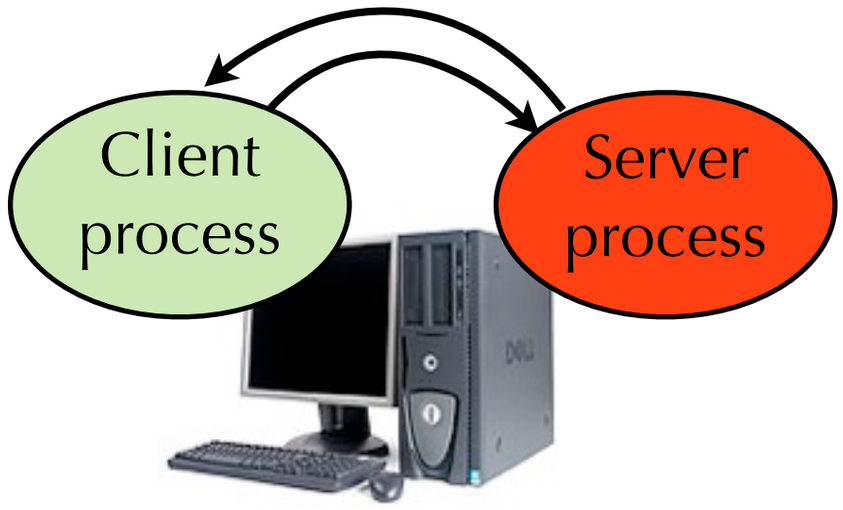
Clients and servers

- E.g.,
  - Web server
    - Protocol used: HTTP
    - Shared resource: Web pages, databases, Rick Astley video, etc.
  - Email server
    - Protocols used: SMTP, POP, IMAP
    - Shared resource: Mailboxes
  - Chat server
    - Protocols used: AIM, MSN, Jabber
    - Shared resource: Relay point for chat messages between clients

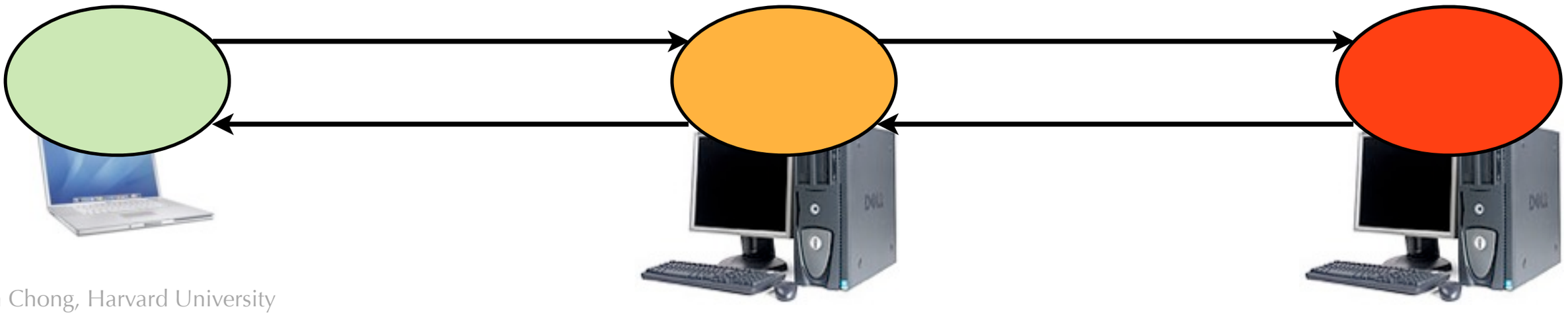


Clients and servers

- Client and server process could be on same machine



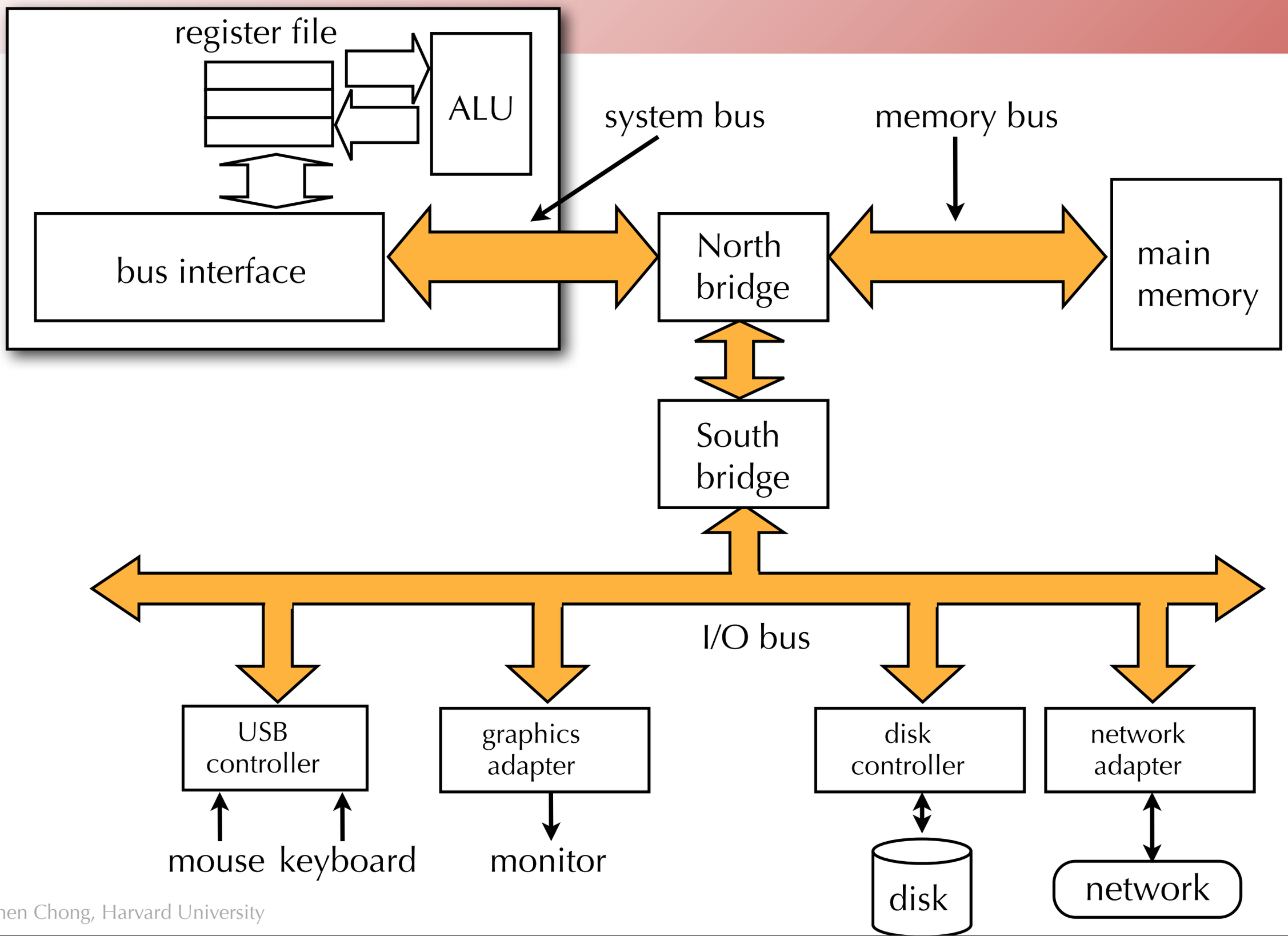
- Server may implement service by being a client of other services
  - E.g., multi-tier web application
    - Web server may be a client of a database service, credit card processing service, etc.



How do a client and server communicate?

- If the client and server are on different machines, they communicate over a **network**
- A **network** is a hierarchical system of boxes and wires organized by geographical proximity
  - SAN (System Area Network) spans cluster or machine room
    - Switched Ethernet, Quadrics QSW, …
  - LAN (Local Area Network) spans a building or campus
    - Ethernet is most prominent example
  - WAN (Wide Area Network) spans country or world
    - Typically high-speed point-to-point phone lines
- An internetwork (internet) is an interconnected set of networks
  - The Global IP Internet (uppercase "I") is the most famous example of an internet (lowercase "i")

Using the network

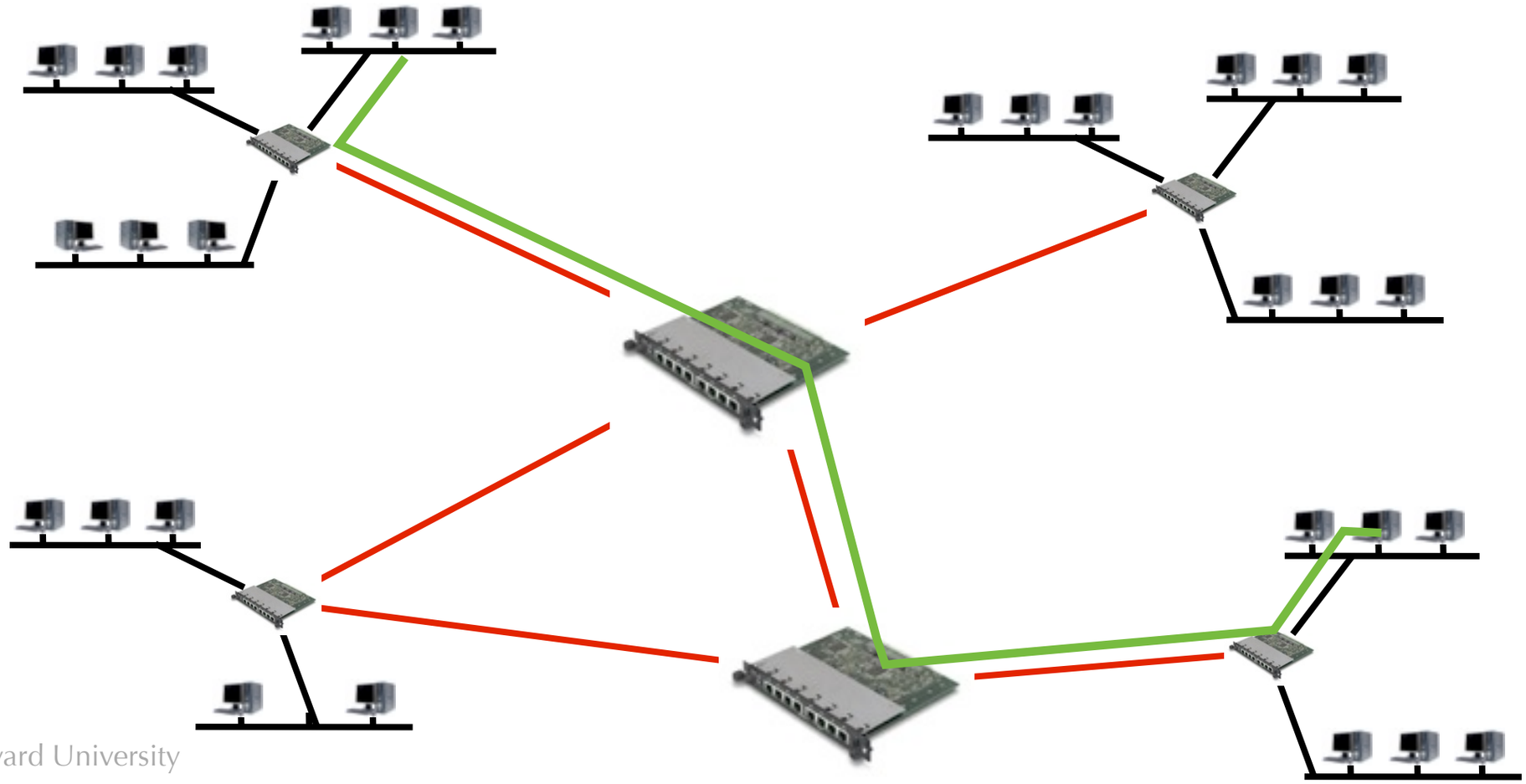


Today

- Clients and servers
- Networks
  - Circuit vs packet
  - Ethernet
  - Internets
  - Protocol stack and TCP
- Network programming
  - Client's view
  - Server's view

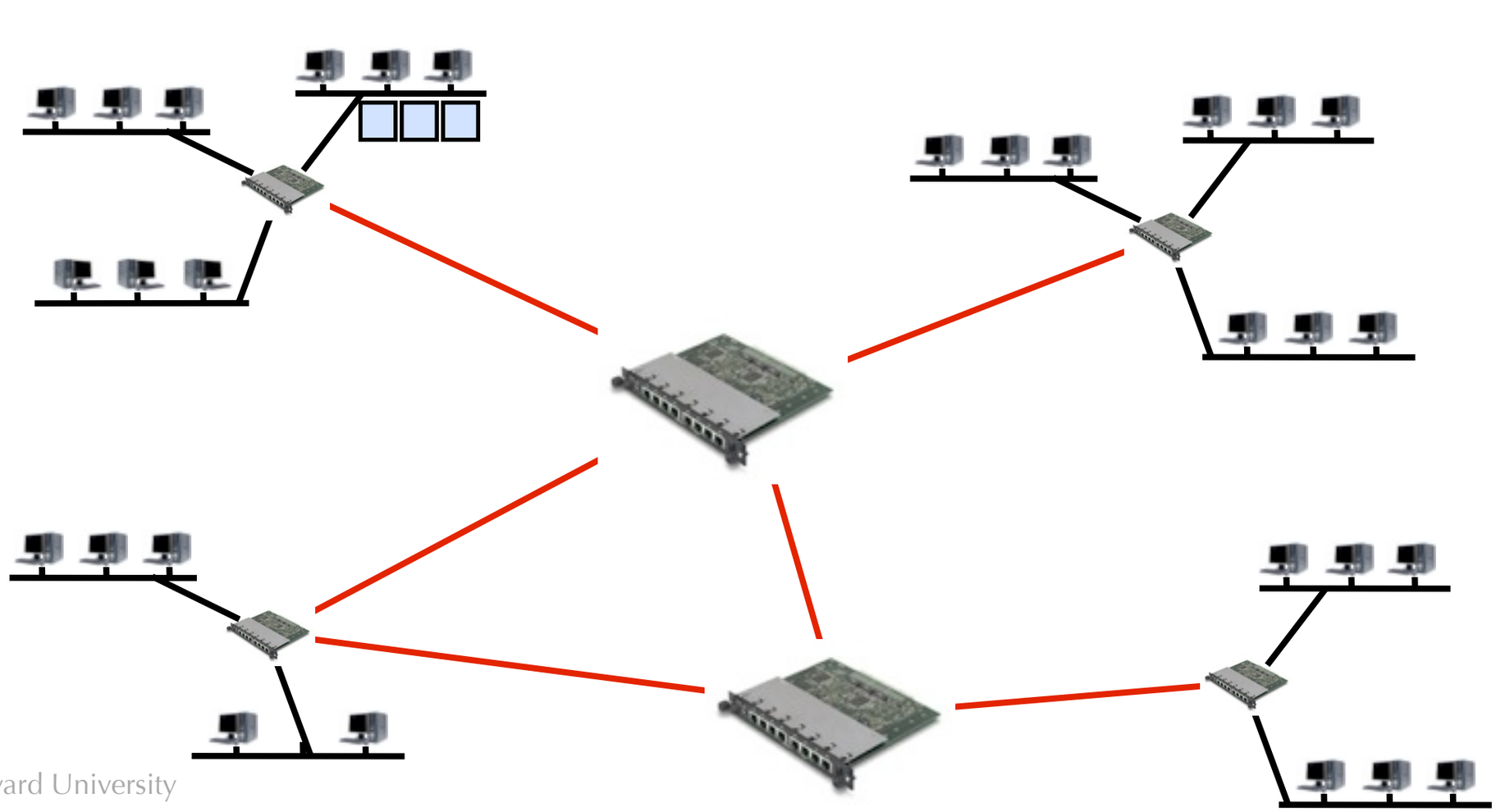
Circuit Switching Networks

- There are two basic types of networks:
- **Circuit switching**:
  - Network creates a dedicated **circuit** between two parties
  - Like the old phone system.



Packet Switching Networks

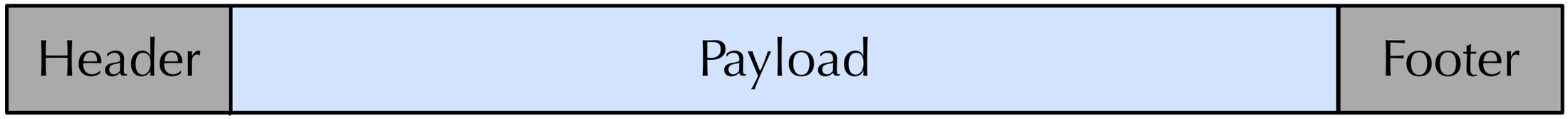
- **Packet switching**:
  - Each computer sends data in the form of (usually short) **packets**
  - Packets are **routed** through the network to their destination.
  - To send a long message, or a stream, use a bunch of packets



Pros and cons of packet switching

- No need to set up a dedicated circuit for each "call"
  - Avoids wasting capacity when no data being sent.
  - Can support much larger amount of traffic overall: **multiplexing**
- Allows network to make decisions on a per-packet basis
- Challenges:
  - Packets may experience variable delay en route
  - Routers may experience congestion, forcing them to drop packets
  - Packets may not be delivered in the order in which they were sent

What does a packet look like?

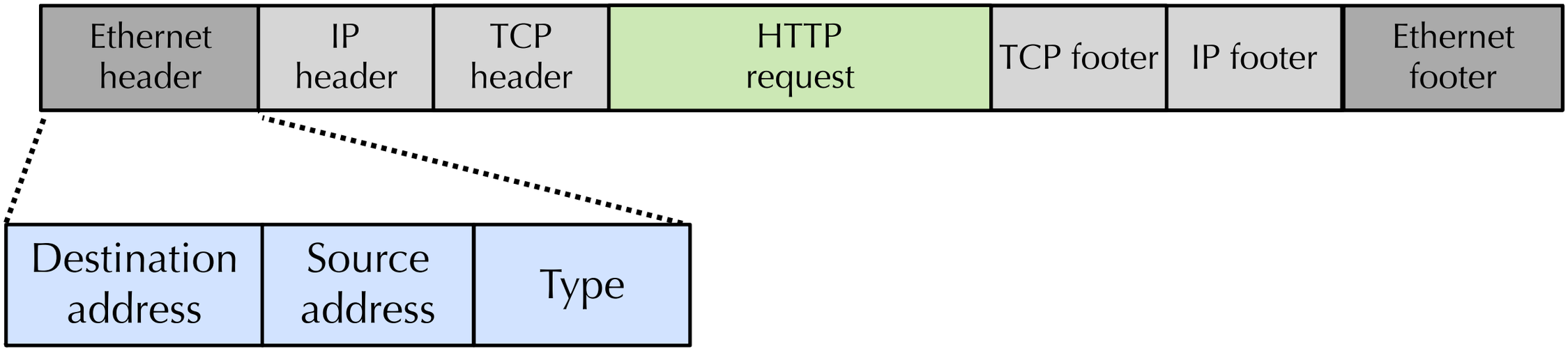


- High level view:
  - **Header** contains routing information (source, destination, etc.)
  - **Payload** contains application data
  - **Footer** contains additional information
    - e.g., CRC of the packet contents for error detection

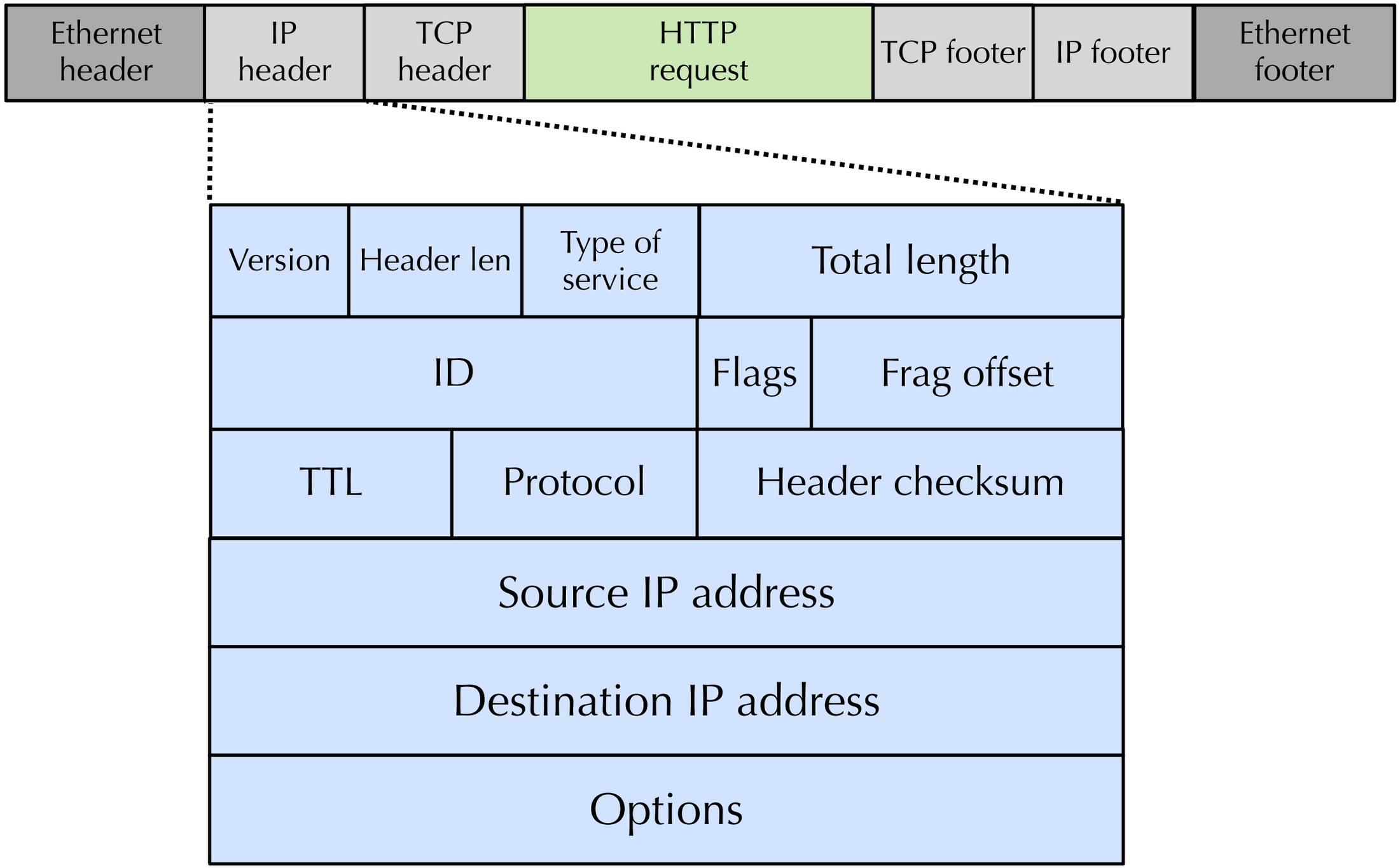
What does a packet look like?

- Protocol layering and encapsulation
  - Multiple protocol layers may operate on a packet.
  - Example: An HTTP request is carried inside a TCP packet, which is contained inside an IP packet, which is contained within an Ethernet packet.
  - This is called **packet encapsulation**.
- Note that not all protocols use footers.
  - TCP and IP don't use them.
  - Shown in the figure just for illustration.

What does a packet look like?

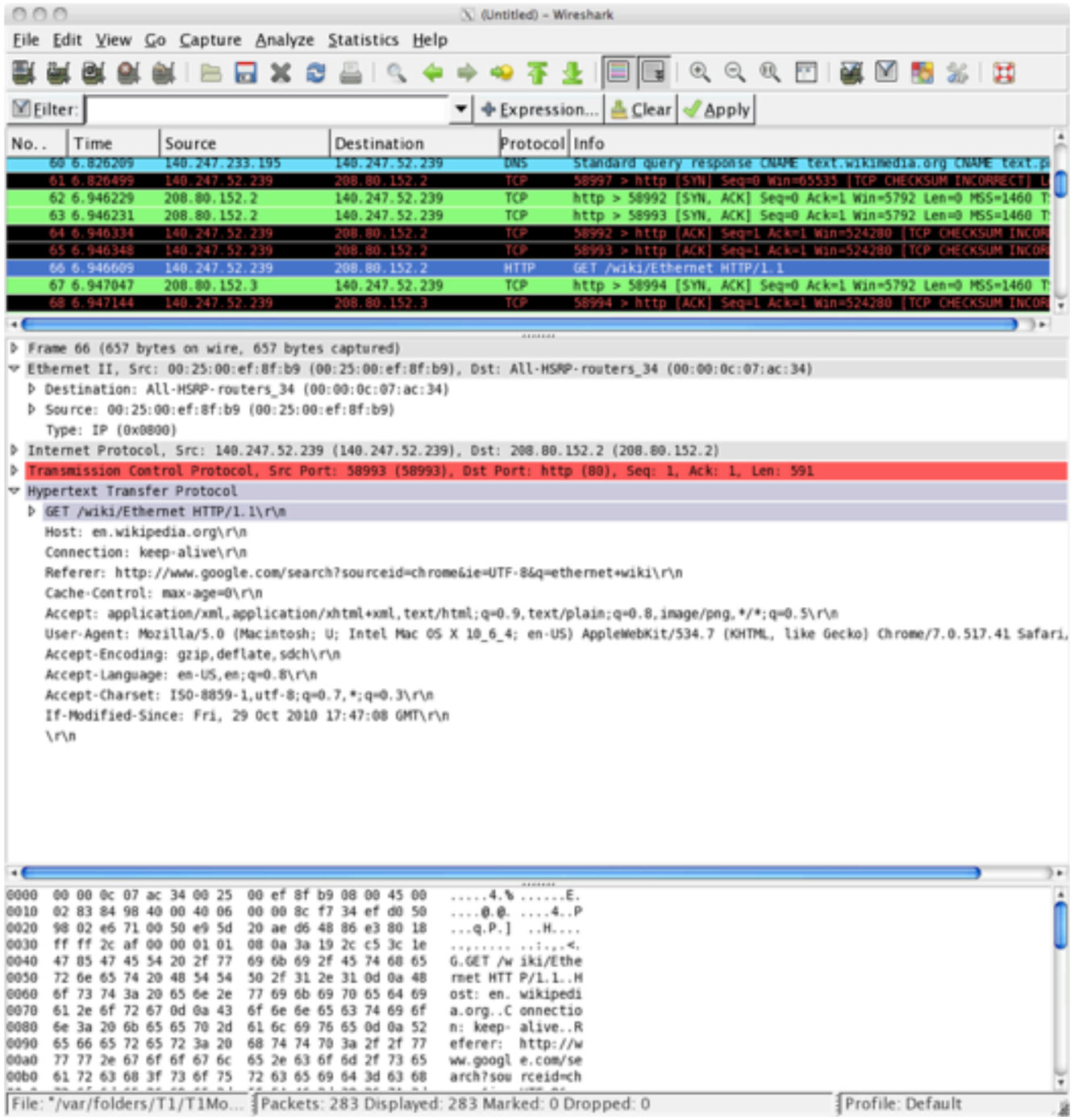


What does a packet look like?



Looking at packets

- Several tools allow you to inspect the contents of network packets.
  - Two most popular are tcpdump and Wireshark.



Bottom up: Ethernet

- Most commonly-used local area network (LAN) technology
  - Developed by Bob Metcalfe while at Xerox PARC in 1973, went on to found 3Com
- Was originally a shared bus design:

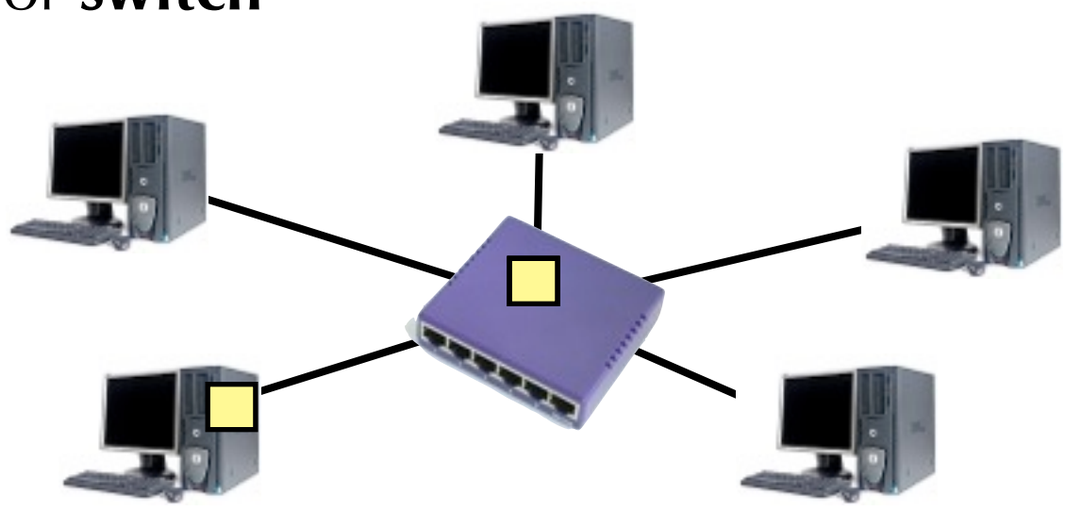
- Only one machine can transmit data at a time!
- So, how do you prevent two machines from interfering with each other?

CSMA/CD

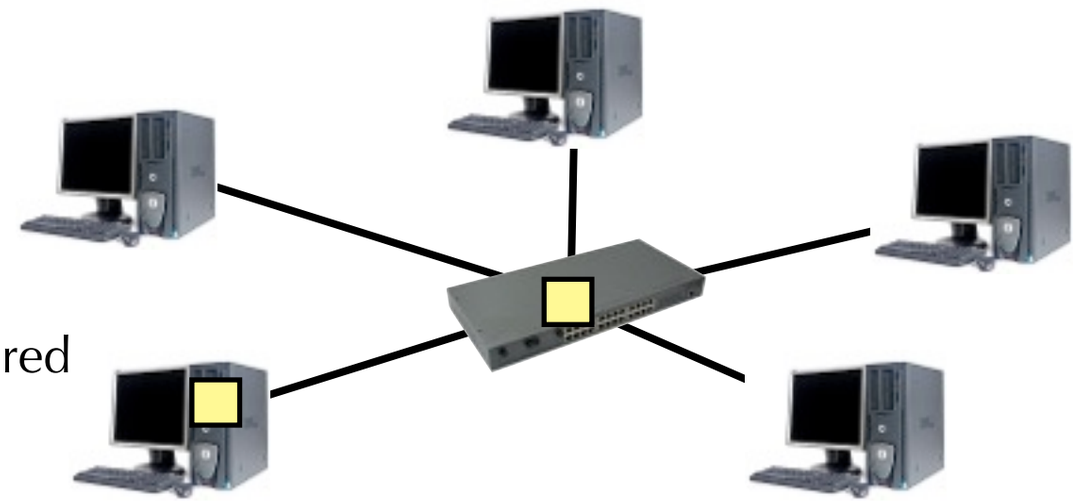
- CSMA/CD: Carrier sense multiple access with collision detection
  - **Carrier sense**: One node can "listen" to see if another node is speaking
  - **Collision detection**: Hardware can tell if two nodes stomp on each other
- What does a node do to transmit a message?
  - First, listen for another transmission
  - If channel is clear, go ahead and transmit
  - Transmitter can tell whether its message collided with another
- In the event of a collision:
  - On the first collision, wait for 0 or 1 "time slots" before retransmitting
    - Time slot is twice the maximum round-trip message delay (51.2 μsec on 10Mbps)
  - If there is another collision, wait for 0, 1, 2, or 3 time slots before retransmitting
  - After N collisions, wait for between 0 ... $2^N - 1$ time slots
    - This is called **exponential backoff**
  - Eventually give up and report error back to the network card driver!

Switched Ethernet

- These days, Ethernet cables are not shared across multiple machines
  - Collisions are a serious problem at high data rates
  - If shared Ethernet cable is broken or gets too long, serious problems arise
- Rather, each cable runs to an **Ethernet hub** or **switch**
- **Ethernet hub:**
  - Acts as a "virtual shared cable"
  - Multiple end hosts connect to the hub
  - Each message received by hub retransmitted to all hosts
  - Inexpensive simple circuitry, but collisions still a problem

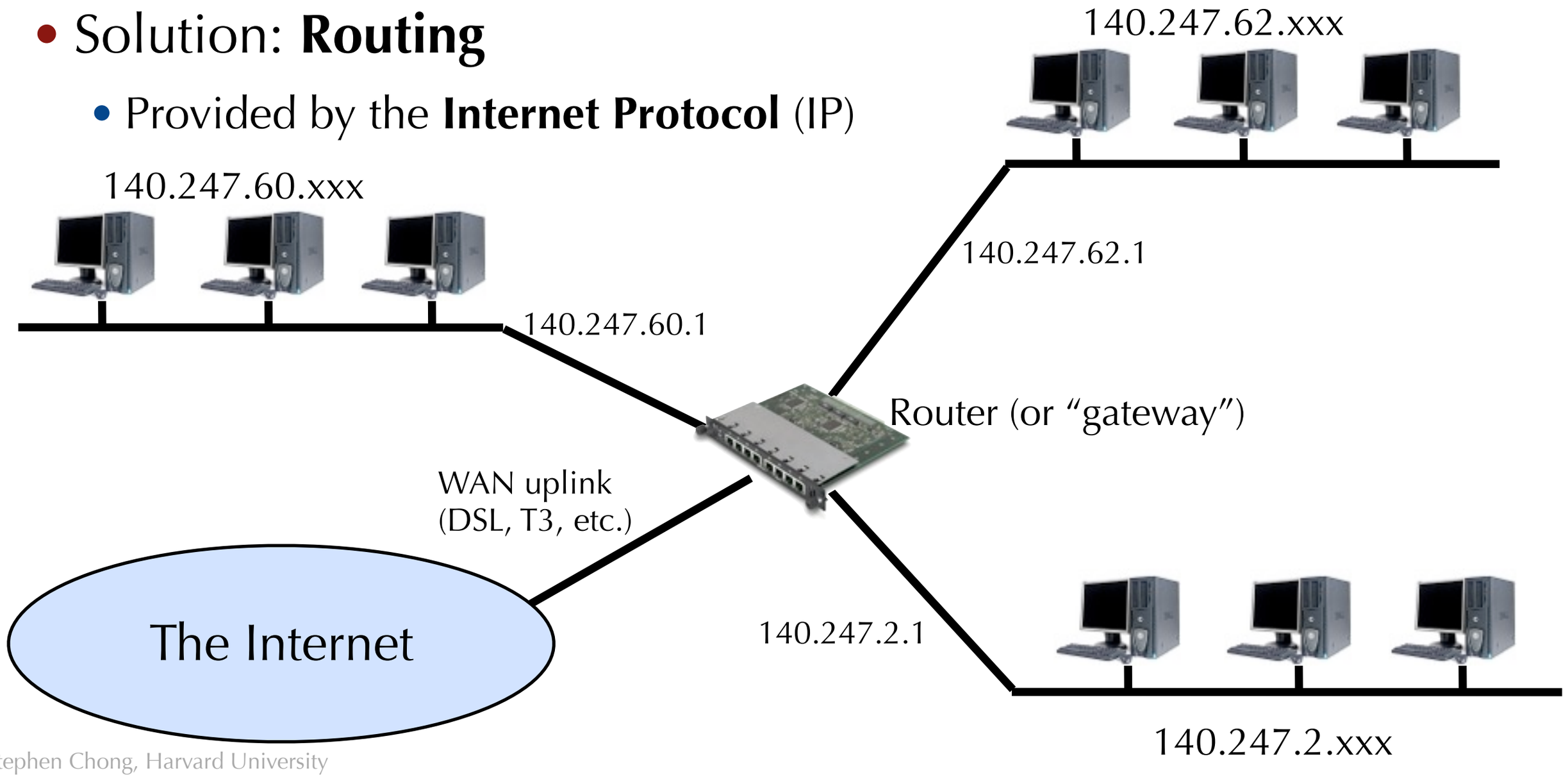


- **Ethernet switch:**
  - Only sends messages to the particular port that they are destined for
  - Needs to know which MAC (Media Access Control) address(es) are connected to each port (can be configured or learned)



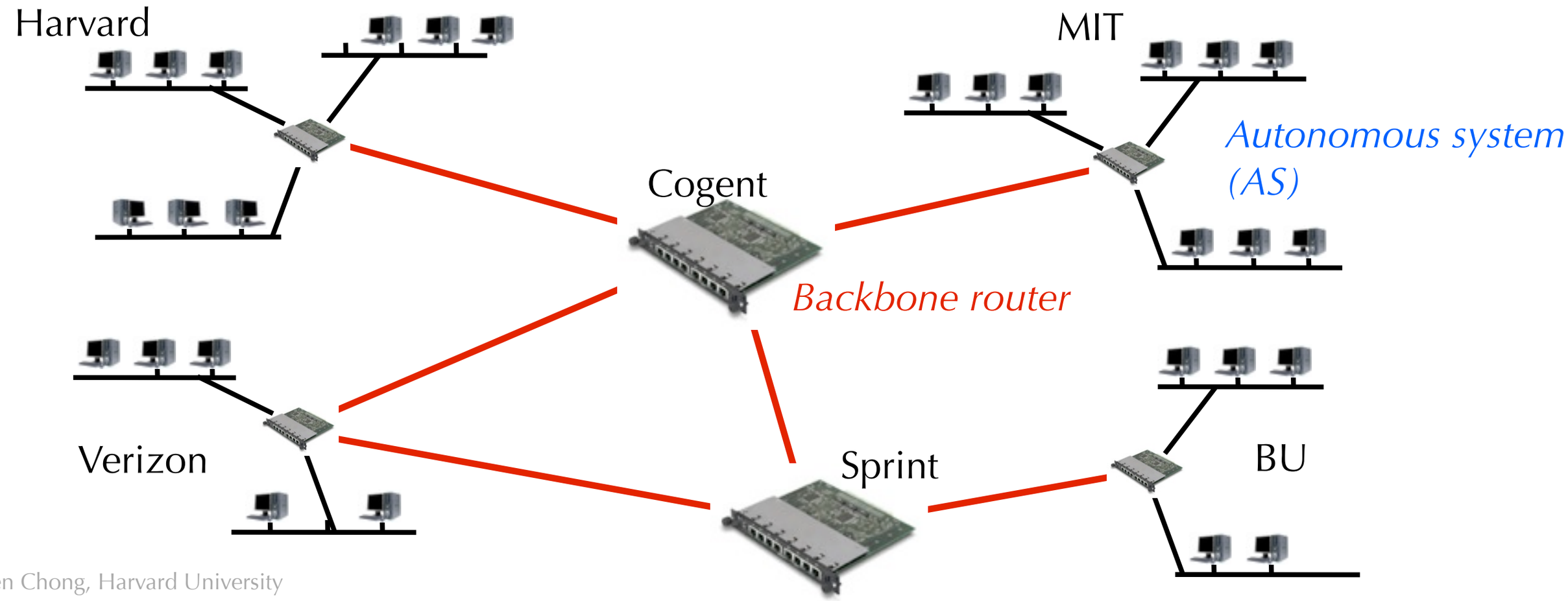
Routing

- Ethernet (and other LAN standards) designed only for local area
  - How do you get different physical networks to talk to each other?
- Solution: **Routing**
  - Provided by the **Internet Protocol** (IP)

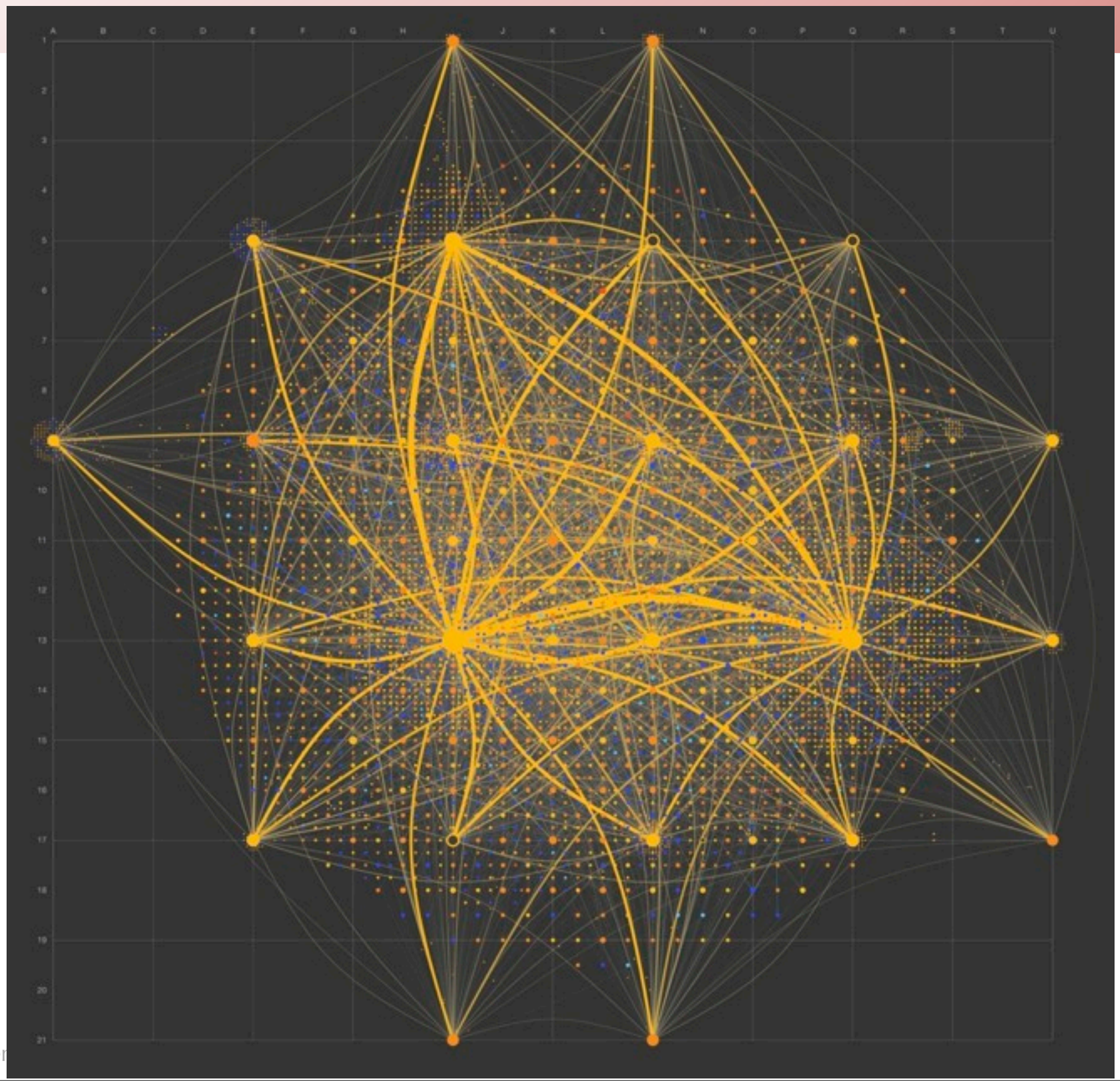


Internetworking

- But how do we communicate over long distances?
  - Group of networks owned by a single entity is an **autonomous system (AS)**
  - Long-distance (WAN) links between multiple sites
  - Routers exchange routing table state to figure out how to get packets around
    - Example: **Border Gateway Protocol** (BGP)



Map of the Internet



Topology of Autonomous Systems
January 2011
From http://www.peer1.com

19,869 autonomous system nodes, joined by 44,344 connections.

Routing across the Internet

- IP routes packets across the Internet
- As packets flow across the Internet, they can be…
  - Fragmented (not all networks carry the same size data payloads)
  - Corrupted (transient bit errors along the physical lines)
  - Dropped (a router's memory may fill and it can't accept more packets)
  - Reordered (router may transmit packets in a different order)
  - Delayed (long buffering delays at a congested router)
- Makes it difficult to get reliable, in-order delivery!
- This is the goal of **Transmission Control Protocol** (**TCP**)
  - Part of the TCP/IP suite of Internet protocols

Protocol Stack and OSI Model

OSI Model	TCP/IP stack	
Application	HTTP, SMTP, Bittorrent, ...	Application data
Presentation	Socket interface	Transform app data into network payloads
Session / *Transport*	TCP, UDP	End-to-end delivery, maybe reliability and flow control
Network	IP, ICMP (ping etc.)	Routing
Data Link	Ethernet MAC (CSMA/CD)	Packet → bit encoding, medium access control
Physical	Ethernet, 802.11, etc.	Raw bits on line

The magic of TCP

- Transmission Control Protocol (TCP) invented in 1974
  - Revised and updated, IP added in 1977
  - Became standard on the emerging ARPAnet in 1983
- TCP provides **end-to-end, reliable, stream-based, connection-oriented** transport
  - **End-to-end**: Only end hosts are responsible for speaking TCP
    - Routers, gateways, etc. in the Internet don't have any part in
  - **Reliable**: Data sent is received exactly once, in the same order
  - **Stream-based**: don't need to think about packets, just send bytes
    - Of course, the IP and physical layers below will stuff TCP data into packets!!
  - **Connection-oriented**: Endpoints establish "connection" before communicating
    - As opposed to protocols where you just send stuff

TCP acknowledgements

- TCP requires **acknowledgements** for every packet sent
  - Each ACK received by sender means receiver got the data

  - If an ACK is not received, retransmit the packet
    - What if an ACK is dropped by the network on its way back to the sender?
  - To avoid sending lots of ACKs, one ACK covers a whole range of packets
- How do you know when to retransmit?
  - Wait for some amount of time after each transmission ...
  - But how long do you wait?
- Idea: **Round-trip-time** (RTT) estimation
  - Determine the delay from sender to receiver: measure time from send to ACK
  - Wait for (2 × RTT) seconds for an ACK before retransmitting each message

TCP congestion control

- TCP also provides **congestion control**
  - Try to avoid overwhelming the receiver with data that it can't handle
  - Try to avoid overwhelming the buffers of intermediate routers!!!
- Sender maintains a **congestion window**
  - Max amount of data it can send before receiving an ACK
  - Question: How do you know how much data you can stuff into the network?
- Idea: “search” for the ideal congestion window
  - Try to send *window_size* bytes (set initial *window_size* to something small)
  - If you don't get an ACK for all of it, assume it's because you sent too much
    - Reduce *window_size* by half
  - If you get an ACK, try to send a little more next time
    - Increase *window_size* by one
  - This is called **additive-increase-multiplicative-decrease window sizing**

Protocol Stack and OSI Model

OSI Model	TCP/IP stack	
Application	HTTP, SMTP, Bittorrent, ...	Application data
Presentation	Socket interface	Transform app data into network payloads
Session / *Transport*	TCP, UDP	End-to-end delivery, maybe reliability and flow control
Network	IP, ICMP (ping etc.)	Routing
Data Link	Ether /CD)	Packet -> bit encoding, medium access control
Physical	Ethe tc.	Raw bits on line

IP on Avian Carriers (RFC 1149)

Network Working Group
Request for Comments: 1149

D. Waitzman
BBN STC
1 April 1990

A Standard for the Transmission of IP Datagrams on Avian Carriers

Status of this Memo

This memo describes an experimental method for the encapsulation of IP datagrams in avian carriers. This specification is primarily useful in Metropolitan Area Networks. This is an experimental, not recommended standard. Distribution of this memo is unlimited.

Overview and Rational

Avian carriers can provide high delay, low throughput, and low altitude service. The connection topology is limited to a single point-to-point path for each carrier, used with standard carriers, but many carriers can be used without significant interference with each other, outside of early spring. This is because of the 3D ether . . .

IP on Avian Carriers (RFC 1149)

```
Script started on Sat Apr 28 11:24:09 2001
vegard@gyversalen:~$ /sbin/ifconfig tun0
tun0      Link encap:Point-to-Point Protocol
          inet addr:10.0.3.2  P-t-P:10.0.3.1  Mask:255.255.255.255
          UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:150  Metric:1
          RX packets:1 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2 errors:0 dr
          collisions:0
          RX bytes:88 (88.0 b)  TX

vegard@gyversalen:~$ ping -i 900 1
PING 10.0.3.1 (10.0.3.1): 56 data
64 bytes from 10.0.3.1: icmp_seq=0
64 bytes from 10.0.3.1: icmp_seq=4
64 bytes from 10.0.3.1: icmp_seq=2
64 bytes from 10.0.3.1: icmp_seq=1

--- 10.0.3.1 ping statistics ---
9 packets transmitted, 4 packets r
round-trip min/avg/max = 3211900.8
vegard@gyversalen:~$ exit
```

Today

- Clients and servers
- Networks
  - Circuit vs packet
  - Ethernet
  - Internets
  - Protocol stack and TCP
- Network programming
  - Client's view
  - Server's view

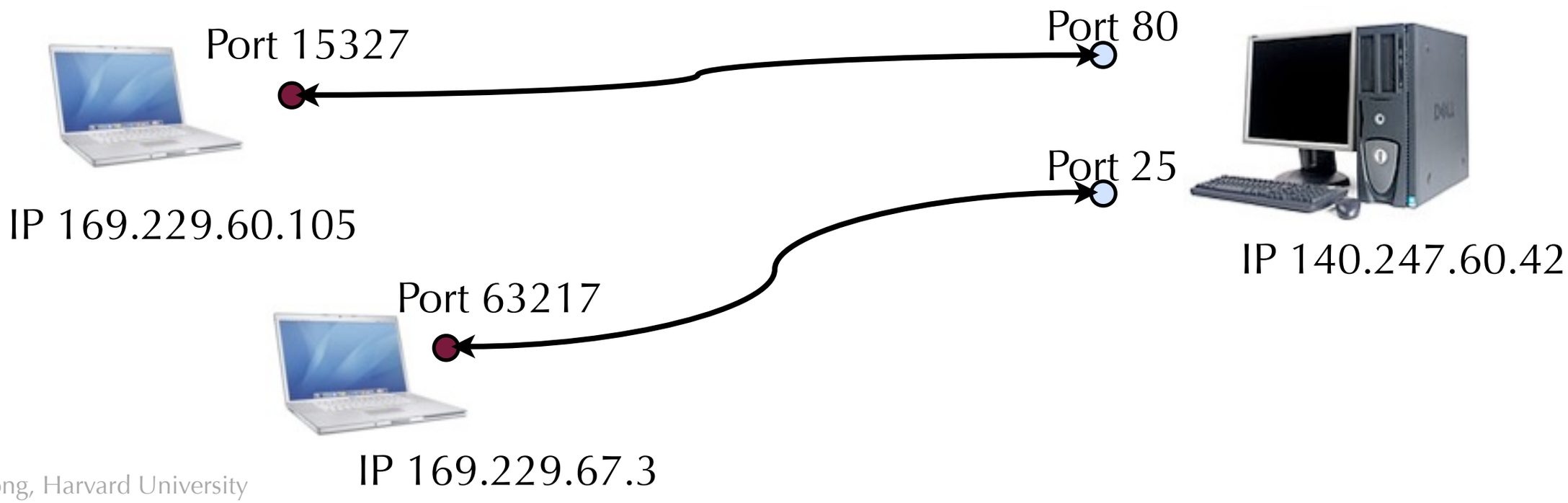
IP Addresses

- An **IP address** represents a machine on the Internet, called a host.
  - Note: IP addresses may not be unique!
  - For example, a private LAN might use the same IP addresses as another private LAN.
  - This is OK as long as they don't ever talk directly to each other.
- An IP address is a 32 bit unsigned integer
  - Typically shown in "**dotted quad**" notation
    - e.g., 140.247.232.88 is www.seas.harvard.edu
- The **Domain Name System** (DNS) converts hostnames to IP addresses.
- Use `gethostbyname()` to look up the IP address for a given hostname.

```
struct in_addr addr; /* An IP address */
struct hostent *host; /* Used for hostname lookup */

host = gethostbyname("www.digg.com");
if (host == NULL) /* error ... */

/* Hosts can have multiple IP addresses.
 * Usually we only care about the first one. */
addr.s_addr = *(in_addr_t*)host->h_addr_list[0];

printf("The IP addr is %s\n", inet_ntoa(addr));
```

Ports

- A single machine can support multiple TCP connections at one time.
  - Each connection uses a different **port number**.
- Servers listen for incoming connections on a (usually well-known) port number.
  - Examples: Port 80 is used for HTTP, port 25 for SMTP (email), port 22 for ssh.
- Clients usually pick an arbitrary **ephemeral** port number for their end
  - Port number is assigned automatically by the kernel when opening the connection.

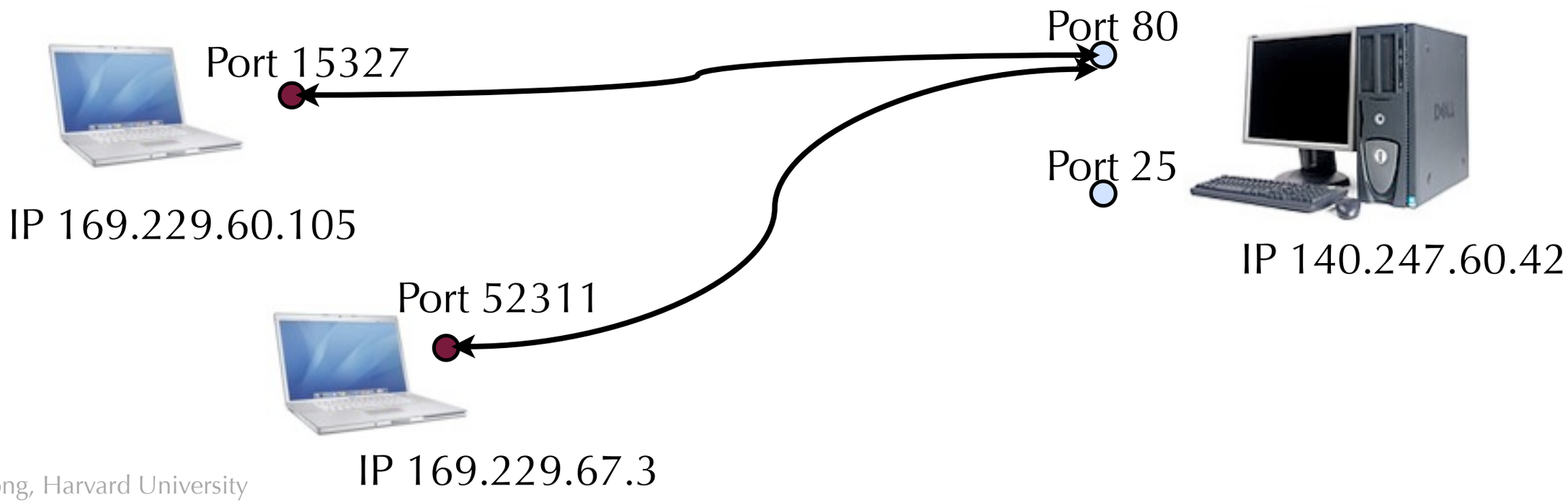


Ports

- Note that multiple connections can be active to the same port.
- How does the OS differentiate between packets coming into the same port number?
- Answer: The connection is uniquely identified by the tuple

  (source IP, source port, destination IP, destination port)

- Since the source IP/port differ, the server can keep connections separate.

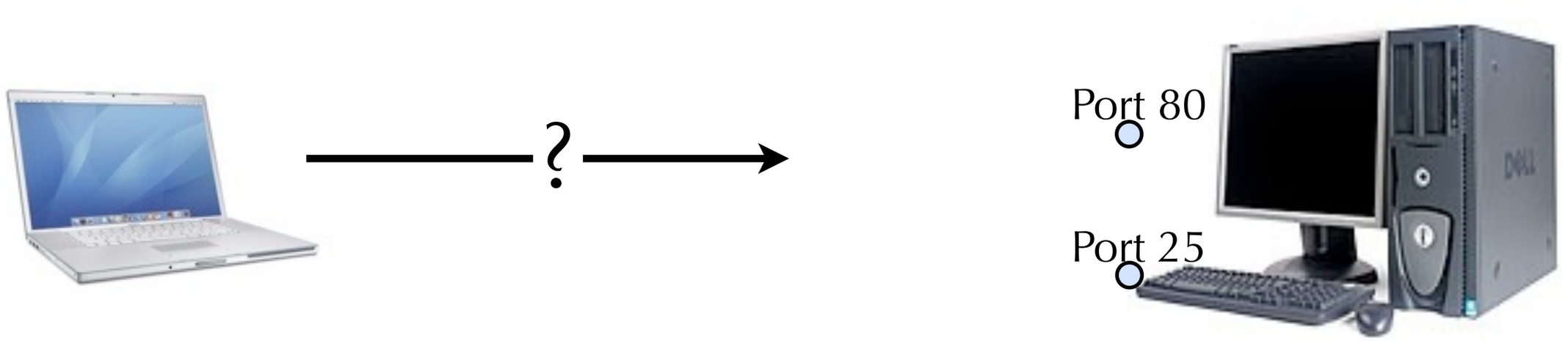


Writing network programs

- In UNIX, **sockets** are the standard interface for networking.
  - A socket represents one endpoint of a network connection: client or server.
  - Sockets can support both TCP (reliable, stream-based) or UDP (unreliable, datagram-based) communication.
  - Most services on the Internet use TCP, but not all. We will focus on TCP.
- Using TCP sockets
  - Create a socket using `socket()`
  - Connect to a remote server using `connect()`
  - Send data with `write()`, receive data with `read()`
  - `close()` the socket when you're done.
- Socket are a lot like files and pipes...
  - The main difference is how they are created and opened.

Client view

- How does a client initiate and use a connection to a server?



Opening a socket

```
struct sockaddr_in addr;   /* IP and port number for the socket. */
struct hostent host;       /* For looking up hostname. */

/* First look up host IP address */
host = gethostbyname(“www.youtube.com”);
memcpy(&(addr.sin_addr.s_addr), host->h_addr_list[0], host->h_length);

/* Set the port number that we want to connect to. */
addr.sin_port = htons(80);

/* Create the socket */
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
  perror("Cannot create socket");
  exit(1);
}

/* Connect to the remote server. */
if ((connect(sockfd, (struct sockaddr *)&addr, (socklen_t)sizeof(addr))) < 0) {
  perror("Cannot connect socket");
  exit(1);
}
```

Opening a socket

```
struct sockaddr_in addr;   /* IP and port number for the socket. */
struct hostent host;       /* For looking up hostname. */

/* First look up host IP address */
host = gethostbyname("www.digg.com");
memcpy(&(addr.sin_addr.s_addr), host->h_addr_list[0], host->h_length);

/* Set the port number that we want to connect to. */
addr.sin_port = htons(80);

...
```

- `struct sockaddr_in` represents the socket's IP and port number.
  - Fill in `addr.sin_addr.s_addr` with the desired IP address
  - Set `addr.sin_port` to the port number to connect to.

Opening a socket

```
struct sockaddr_in addr;   /* IP and port number for the socket. */
struct hostent host;       /* For looking up hostname. */

/* First look up host IP address */
host = gethostbyname("www.digg.com");
memcpy(&(addr.sin_addr.s_addr), host->h_addr_list[0], host->h_length);

/* Set the port number that we want to connect to. */
addr.sin_port = htons(80);

...
```

- Must use **network byte order** for IP and port numbers.
  - Not all systems on the Internet are little-endian!
  - Idea: Network byte order is a convention that all hosts on the Internet must follow when sending and receiving data. (Network byte order happens to be big-endian.)
  - `htons(val)` converts a 16-bit short value from host byte order to network byte order.
  - `ntohs(val)` does the opposite. Also see: `htonl(val)`, `ntohl(val)`, etc.

Opening a socket

```
struct sockaddr_in addr;   /* IP and port number for the socket. */
struct hostent host;       /* For looking up hostname. */
...
/* Create the socket */
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
  perror("Cannot create socket");
  exit(1);
}

/* Connect to the remote server. */
if ((connect(sockfd, (struct sockaddr *)&addr, (socklen_t)sizeof(addr))) < 0) {
  perror("Cannot connect socket");
  exit(1);
}
```

- Socket created using the `socket()` system call
  - Can support many types of protocols.
  - AF_INET: Use Internet protocols.
  - SOCK_STREAM: Create a streaming, reliable socket (that is, TCP).
  - SOCK_DGRAM would be used for UDP (datagram) sockets.

Opening a socket

```
struct sockaddr_in addr;   /* IP and port number for the socket. */
struct hostent host;       /* For looking up hostname. */
...
/* Create the socket */
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
  perror("Cannot create socket");
  exit(1);
}

/* Connect to the remote server. */
if ((connect(sockfd, (struct sockaddr *)&addr, (socklen_t)sizeof(addr))) < 0) {
  perror("Cannot connect socket");
  exit(1);
}
```

- Connect to the server using `connect()`
  - The struct `sockaddr_in` indicates which IP/port to connect to.
  - Blocks until the connection is established.
  - So, if server is down or not accepting connections, this will hang...
  - Can use `alarm()` before calling `connect()` to time out the connection request.

Sending and receiving data

- Can simply use `write()` to send data, and `read()` to read data.
- Problem: Socket may not give you all of the data you want on each call.
- This is where the RIO routines (from csapp.c) come in handy!

```
char buf[BUFSIZE];

/* Send an HTTP request for index.html */
sprintf(buf, "GET index.html HTTP/1.0\r\n\r\n");

Rio_writen(sockfd, buf, strlen(buf));

/* Read the response back from the server */
do {
  count = Rio_readn(sockfd, buf, BUFSIZE-1);
  /* Important: Set NULL byte at end of buffer! */
  buf[count] = 0;
  if (count > 0) {
    printf("%s\n", buf);
  }
} while (count > 0);

close(sockfd);
```

Sending and receiving data

- Send an HTTP request for "index.html" on the server.
  - HTTP is a simple, ASCII based protocol.
  - To request this URL from the server, we send the string:
    `GET index.html HTTP/1.0`
    followed by two end of line markers ("`\r\n`" in C).

```
char buf[BUFSIZE];

/* Send an HTTP request for index.html */
sprintf(buf, "GET index.html HTTP/1.0\r\n\r\n");

Rio_writen(sockfd, buf, strlen(buf));

/* Read the response back from the server */
do {
  count = Rio_readn(sockfd, buf, BUFSIZE-1);
  /* Important: Set NULL byte at end of buffer! */
  buf[count] = 0;
  if (count > 0) {
    printf("%s\n", buf);
  }
} while (count > 0);

close(sockfd);
```

Sending and receiving data

- We use `Rio_writen()` to send the data
  - Recall: This is just a wrapper to `write()`
  - Guarantees that all of the data will be written, unless there is an error.
  - Automatically exits if there is an error
    (use `rio_writen()` to do your own error handling).

```
char buf[BUFSIZE];

/* Send an HTTP request for index.html */
sprintf(buf, "GET index.html HTTP/1.0\r\n\r\n");

Rio_writen(sockfd, buf, strlen(buf));

/* Read the response back from the server */
do {
  count = Rio_readn(sockfd, buf, BUFSIZE-1);
  /* Important: Set NULL byte at end of buffer! */
  buf[count] = 0;
  if (count > 0) {
    printf("%s\n", buf);
  }
} while (count > 0);

close(sockfd);
```

Sending and receiving data

- Read the response from the server using `Rio_readn()`
  - Recall: This is just a wrapper to `read()`
  - Reads until the buffer is full, or EOF (socket closed).
  - Automatically exits if there is an error
    (use `rio_readn()` to do your own error handling).

```
char buf[BUFSIZE];

/* Send an HTTP request for index.html */
sprintf(buf, "GET index.html HTTP/1.0\r\n\r\n");

Rio_writen(sockfd, buf, strlen(buf));

/* Read the response back from the server */
do {
  count = Rio_readn(sockfd, buf, BUFSIZE-1);
  /* Important: Set NULL byte at end of buffer! */
  buf[count] = 0;
  if (count > 0) {
    printf("%s\n", buf);
  }
} while (count > 0);

close(sockfd);
```

Sending and receiving data

- Why do we call `Rio_readn()` in a loop?
  - The buffer may not be big enough to hold all of the data coming back from the server!
  - Need to read multiple times to get all of the data.

```
char buf[BUFSIZE];

/* Send an HTTP request for index.html */
sprintf(buf, "GET index.html HTTP/1.0\r\n\r\n");

Rio_writen(sockfd, buf, strlen(buf));

/* Read the response back from the server */
do {
  count = Rio_readn(sockfd, buf, BUFSIZE-1);
  /* Important: Set NULL byte at end of buffer! */
  buf[count] = 0;
  if (count > 0) {
    printf("%s\n", buf);
  }
} while (count > 0);

close(sockfd);
```

Sending and receiving data

- Why do we set `buf[count]` to 0?
  - In C, strings are terminated by a NULL byte (zero).
  - But the server does not send this NULL byte over the socket!
  - So it's our job to terminate the string before calling `printf()`

```
char buf[BUFSIZE];

/* Send an HTTP request for index.html */
sprintf(buf, "GET index.html HTTP/1.0\r\n\r\n");

Rio_writen(sockfd, buf, strlen(buf));

/* Read the response back from the server */
do {
  count = Rio_readn(sockfd, buf, BUFSIZE-1);
  /* Important: Set NULL byte at end of buffer! */
  buf[count] = 0;
  if (count > 0) {
    printf("%s\n", buf);
  }
} while (count > 0);

close(sockfd);
```

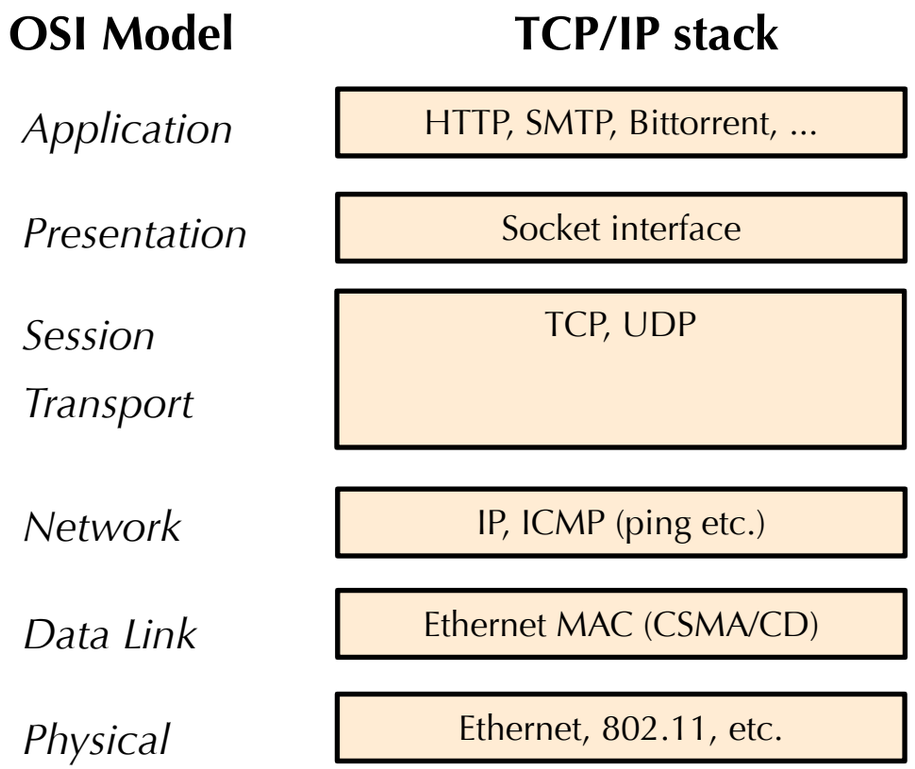
Sending and receiving data

- When done with connection, just `close()` the socket.
  - Note that the server might `close()` its end of the connection first!
  - This causes `read()` to return EOF (return value zero).
  - Likewise, trying to `write()` to a closed socket returns an error (ECONNRESET).

```
char buf[BUFSIZE];

/* Send an HTTP request for index.html */
sprintf(buf, "GET index.html HTTP/1.0\r\n\r\n");

Rio_writen(sockfd, buf, strlen(buf));

/* Read the response back from the server */
do {
  count = Rio_readn(sockfd, buf, BUFSIZE-1);
  /* Important: Set NULL byte at end of buffer! */
  buf[count] = 0;
  if (count > 0) {
    printf("%s\n", buf);
  }
} while (count > 0);

close(sockfd);
```

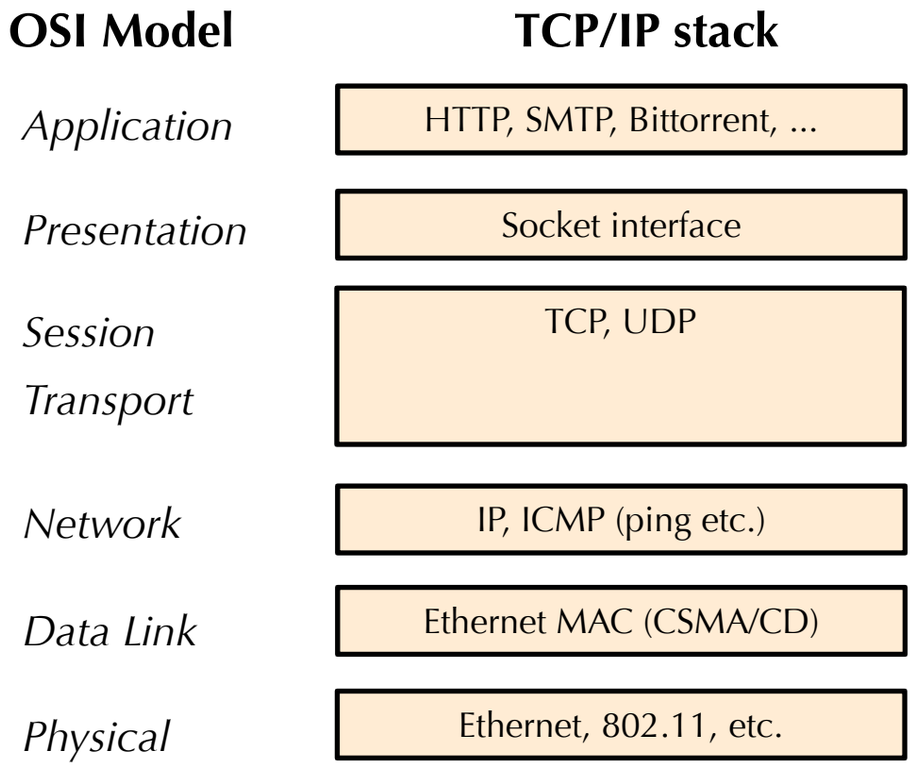
Protocols

- Every service on the Internet is defined by a protocol
  - The protocol defines the format of the messages exchanged by clients and servers, the ordering, and meaning of those messages.
- Example
  - HTTP – Hypertext Transfer Protocol. Used by Web servers and Web browsers
  - SMTP – Simple Mail Transfer Protocol. Used by email clients and servers



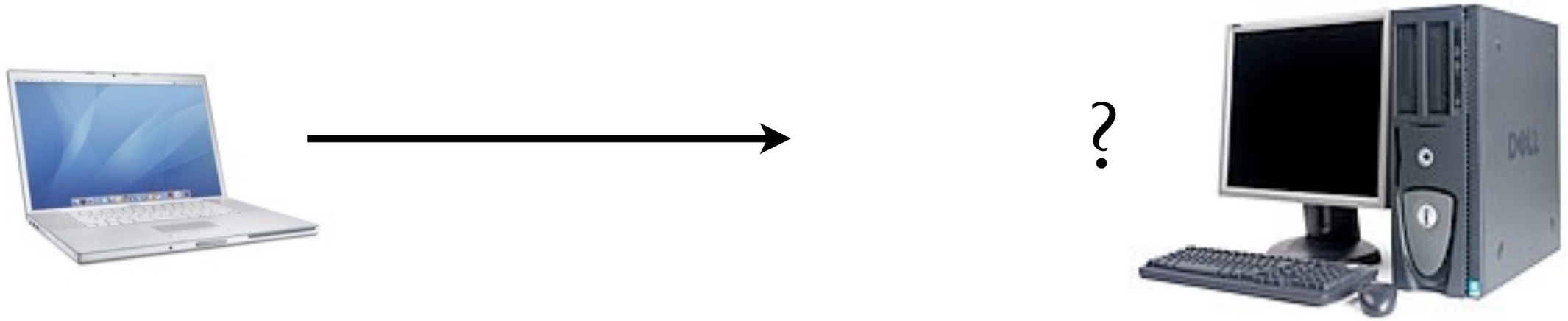
Protocols

- When writing a network client, you need to know how to speak the protocol to the server. How do you do this?
  - Option #1: Read the protocol definition and write the code yourself.
    Usually in a document called an RFC (Request For Comments) -- www.ietf.org
    - Example: HTTP v1.1 is defined in RFC 2616.
  - Option #2: Use a library that speaks the protocol for you
    - Many standard libraries that implement HTTP, SMTP, and other protocols.
    - Usually safer than implementing it from scratch on your own.

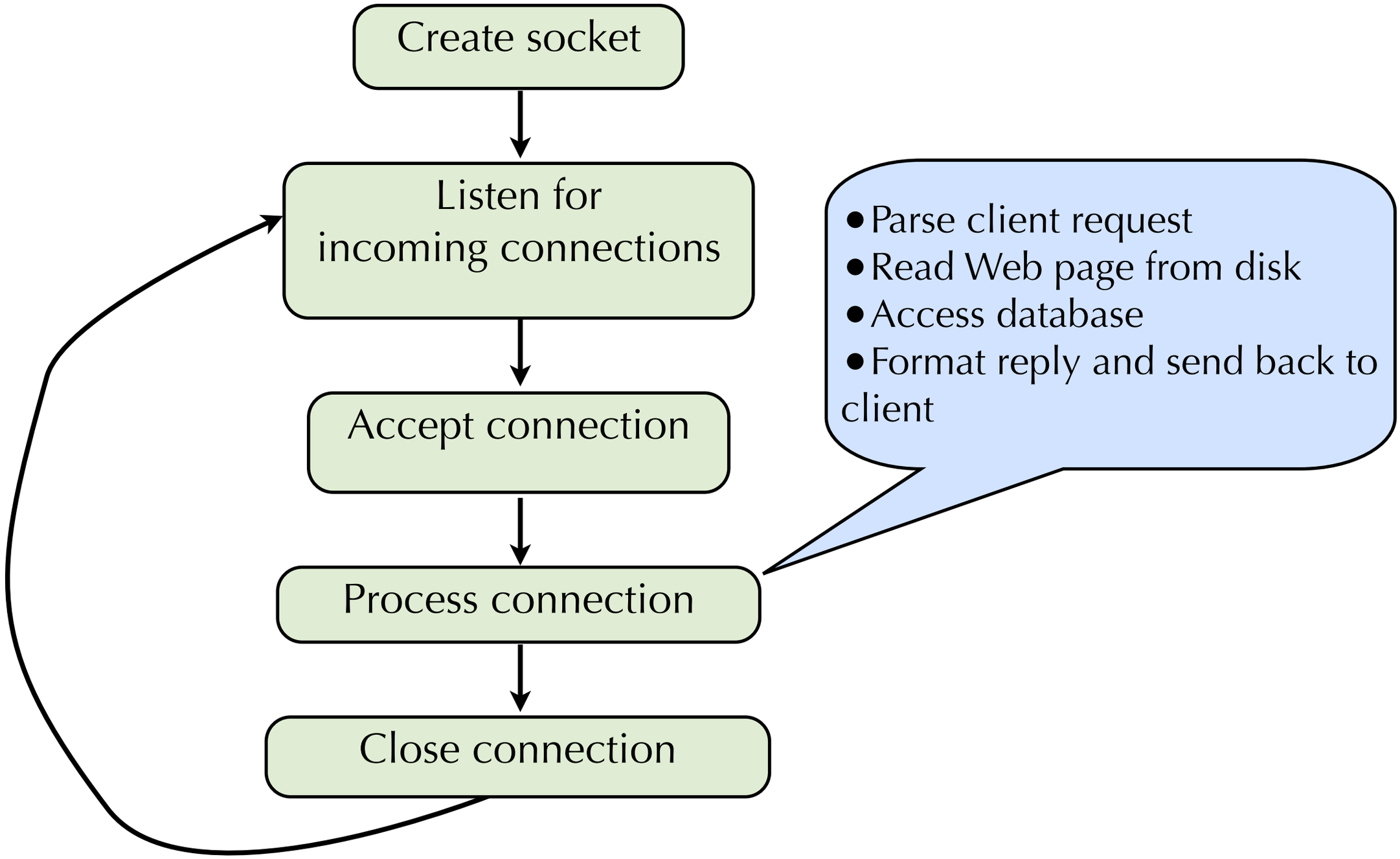


Server view

- How does a server listen for and accept connections from clients?



Lifecycle of a server



Step 1: Creating socket

```
int listenfd;   /* Socket to listen on */
int optval = 1; /* Used by setsockopt() below */

if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
  perror("Cannot create socket");
  exit(1);
}

/* Allow this socket to reuse a port number */
if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
    (const void*)&optval, sizeof(int)) < 0) {
  perror("Cannot set SO_REUSEADDR socket option");
  exit(1);
}
```

- Create a socket: `socket()` system call
  - A **socket** is an endpoint for communication
- Use `setsockopt()` to permit socket to reuse the server port number
  - Otherwise if you restart the server, the kernel thinks the port number is already in use by the (now dead) server process.
  - Eventually the OS would allow the port to be reused, but only after a long timeout.
  - This has to do with details of the TCP protocol.

Step 2: Binding and listening

```
struct sockaddr_in server_addr;

/* Zero out the server address */
bzero(&server_addr, sizeof(server_addr));
server_addr.sin_family = AF_INET;

/* Listen for connections from any client on the Internet. */
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);

/* Listen for connections on port 80 */
server_addr.sin_port = htons(80);
```

- First, prepare the server address
  - Generally set address to INADDR_ANY to indicate you will accept connections from any IP address on the Internet.

Step 2: Binding and listening

```
/* Bind socket to address */
if (bind(listenfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
  perror("Cannot bind socket");
  exit(1);
}

/* Listen for incoming connections. Use listen queue length of 10. */
if (listen(listenfd, 10) < 0) {
  perror("Cannot listen");
  exit(1);
}
```

- Then `bind()` the socket to the port you want to listen on.

- Then `listen()` for incoming connections.

Step 3: Accept connection

```
int clifd;
struct sockaddr_in client_addr;
int client_addr_len = sizeof(client_addr);

while (1) {
  clifd = accept(listenfd,
                 (struct sockaddr *)&client_addr,
                 (socklen_t*)&client_addr_len);
  /* Process connection here */
  do_something(clifd);

  close(clifd);
}
```

- The server spins in a loop doing the following:
  - Call `accept()` to accept an incoming connection from the Internet
    - This blocks until a connection is received!
  - Process the connection
  - `close()` the client socket
- `accept()` returns new file descriptor of socket for the new connection.

Find out the client's hostname

- `accept()` returns the client's IP address and port number.
- Can use `gethostbyaddr()` to look up the hostname.
  - Note that not all IP addresses have a corresponding hostname.

```
struct hostent *hp;
char *hostname, *hostip;

while (1) {
  clifd = accept(listenfd,
                 (struct sockaddr *)&client_addr,
                 (socklen_t*)&client_addr_len);

  hp = gethostbyaddr((const char *)&client_addr.sin_addr.s_addr,
                     sizeof(client_addr.sin_addr.s_addr), AF_INET);
  if (hp != NULL) {
    hostname = hp->h_name;
  } else {
    hostname = "unknown hostname";
  }
  hostip = inet_ntoa(client_addr.sin_addr);
  fprintf(stderr, "Got connection from %s (%s)\n", hostname, hostip);

  /* Do stuff... */
```

Step 4: Process the connection

```
rio_t rio; /* Use the RIO libraries to do I/O */
Rio_readinitb(&rio, clifd);

/* Read lines from the client */
while ((n = Rio_readlineb(&rio, buf, BUFSIZE)) != 0) {
  fprintf(stderr, "Read line from client: %s\n", buf);
  if (strcmp(buf, "\r\n") == 0) {
    // Got a blank line, means request is done.
    break;
  }
}

/* Send the client some HTML */
sprintf(buf, "<html><body><b>Hello from my wicked awesome simple web server.</
b><p>You are coming from %s with IP address %s.</body></html>\n",
 hostname, hostip);

/* Send data back to client */
Rio_writen(clifd, buf, strlen(buf));
```

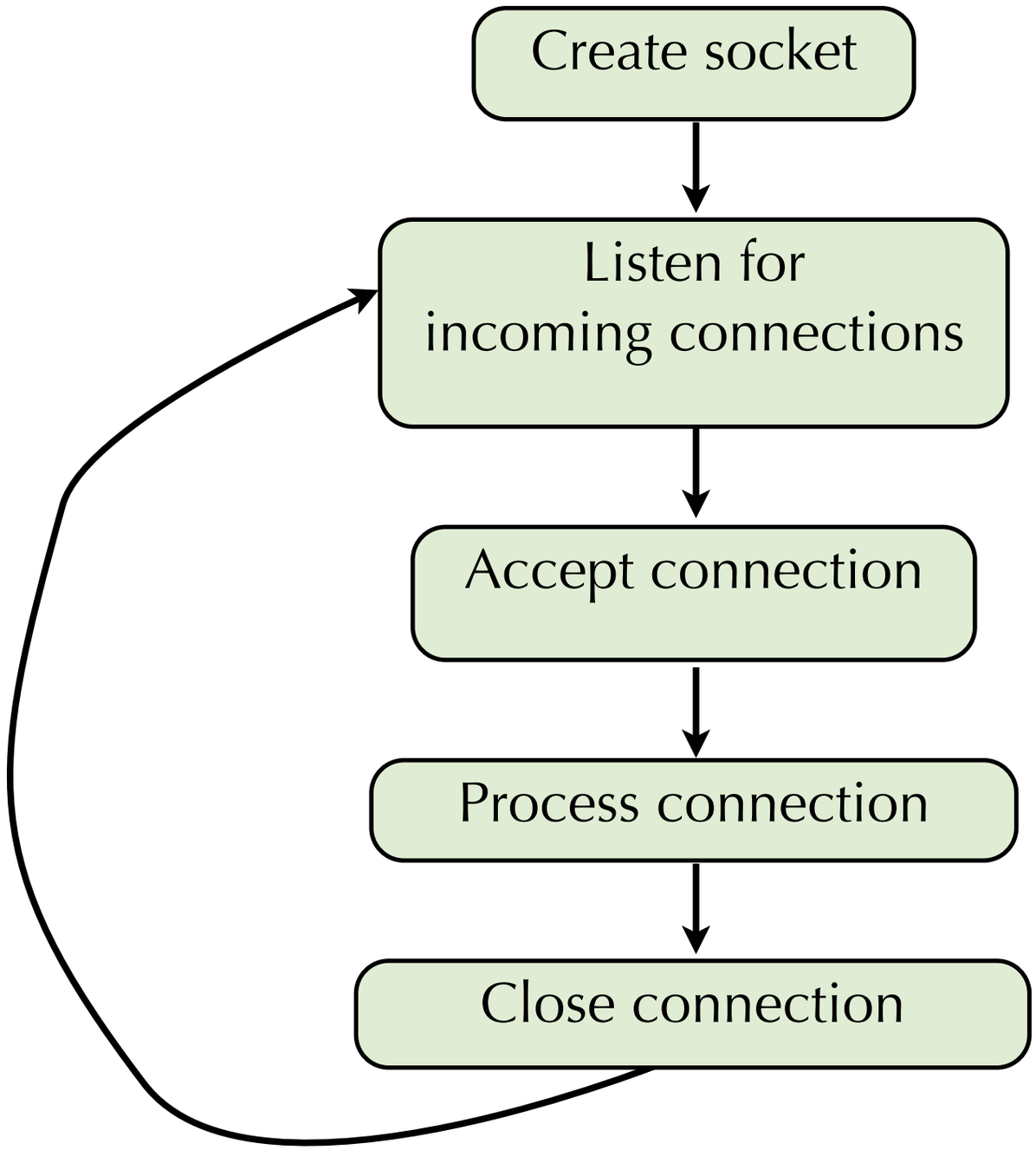
Sockets summary

- A **socket** is an endpoint for communication
- Client:
  - Create a socket
    - Get back a file descriptor identifying the socket
  - **Connect** socket to a server address
- Server
  - Create a socket
  - **Bind** the socket to a port, and set it to **listen**
  - **Accept** connections
    - Will create a new socket for each request

For More Information

- W. Richard Stevens, “Unix Network Programming: Networking APIs: Sockets and XTI”, Volume 1, Second Edition, Prentice Hall, 1998.
  - The network programming bible.
- Unix Man Pages
  - Good for detailed information about specific functions
- Textbook has more details as well, and sample code for a simple “echo” client and server.
  - Available from csapp.cs.cmu.edu
  - You should compile and run them for yourselves to see how they work.
  - Feel free to borrow any of this code.

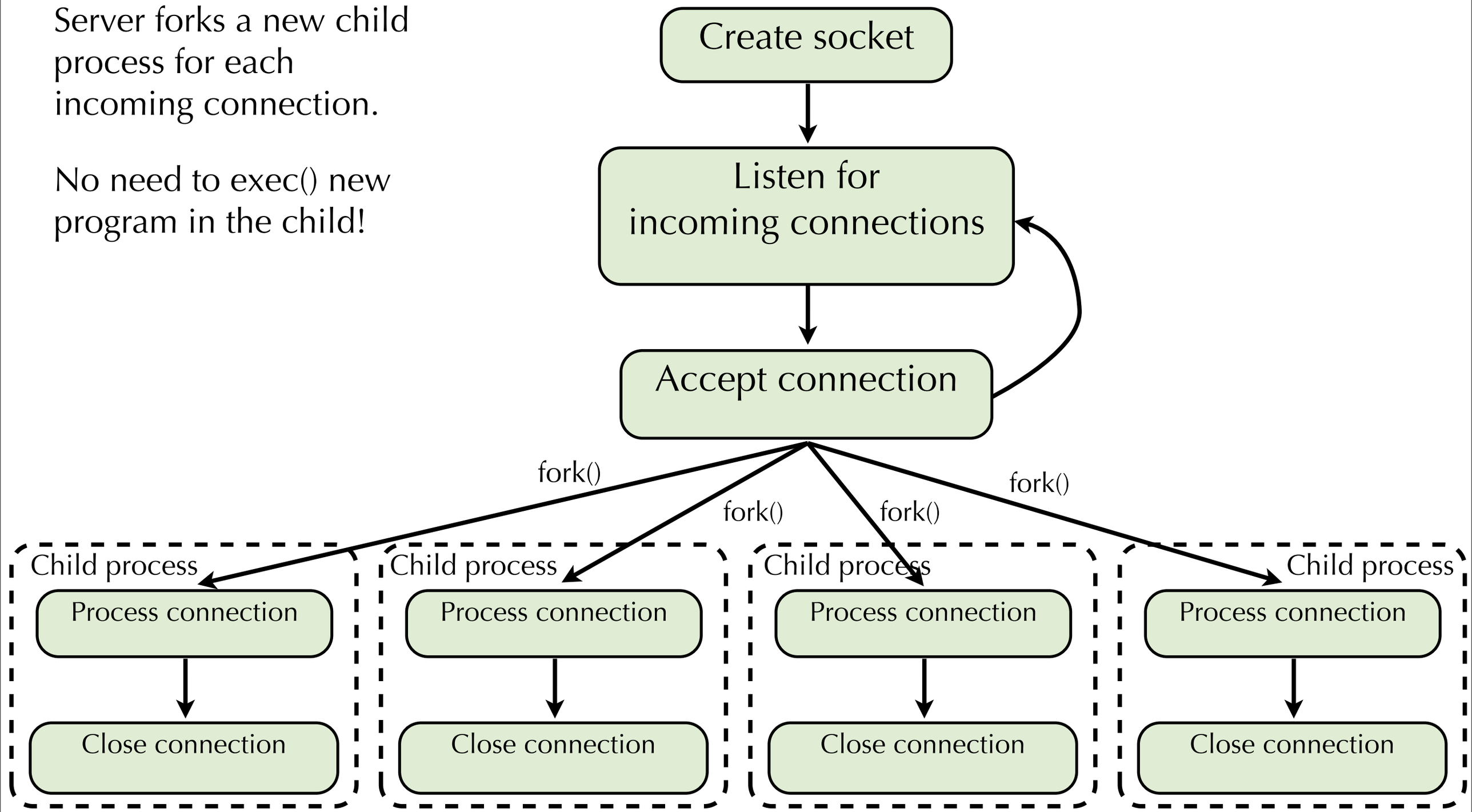
What's wrong with this design?



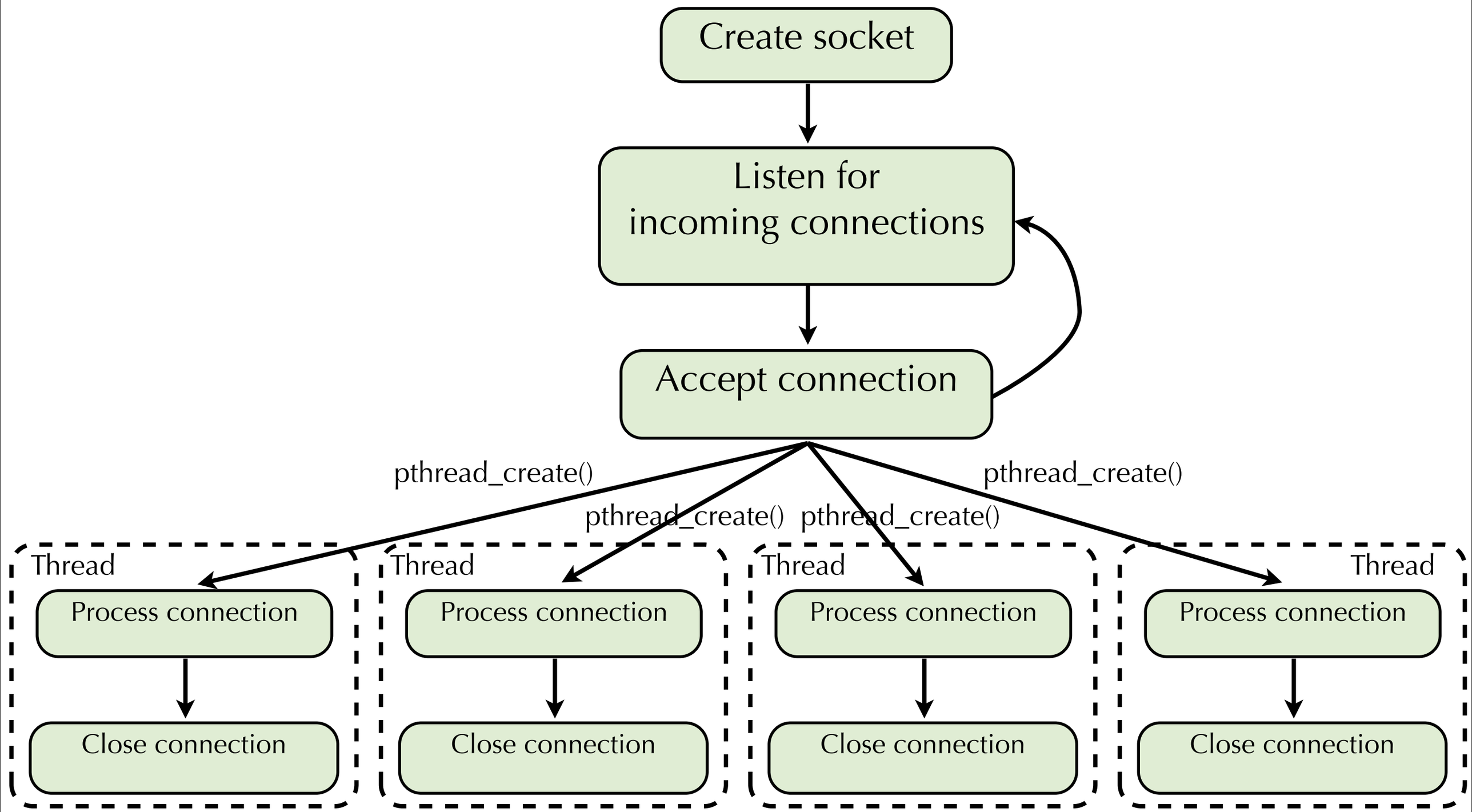
Multiprocess server

Server forks a new child process for each incoming connection.

No need to exec() new program in the child!



Multithreaded server



Threads vs. Processes

- Pro process
  - Processes isolated from each other
    - Harder to exploit bugs/security vulnerabilities
- Pro thread
  - Too many processes can slow down the system.
  - Child processes can't share memory with each other or the main server process.