Network Programming: Part I

15-213: Introduction to Computer Systems
21st Lecture, July 19, 2017
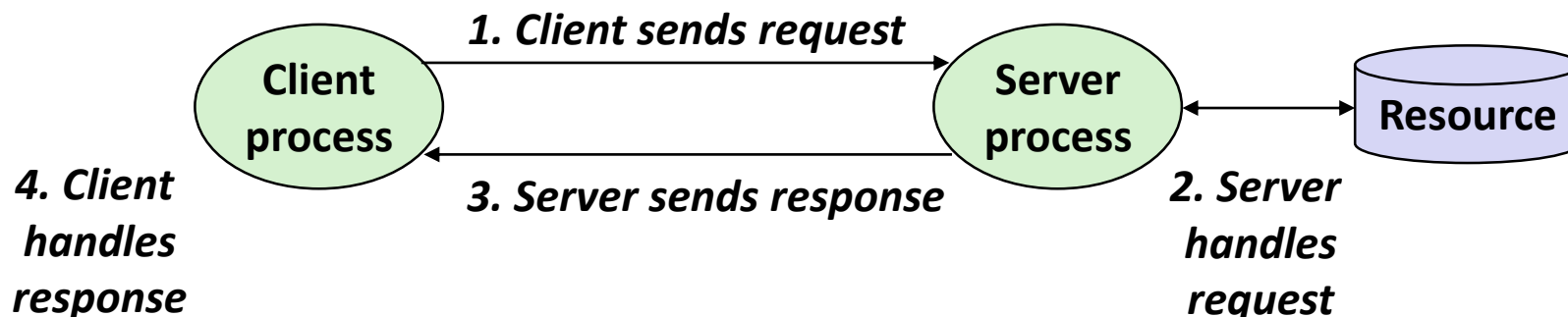
Instructor:

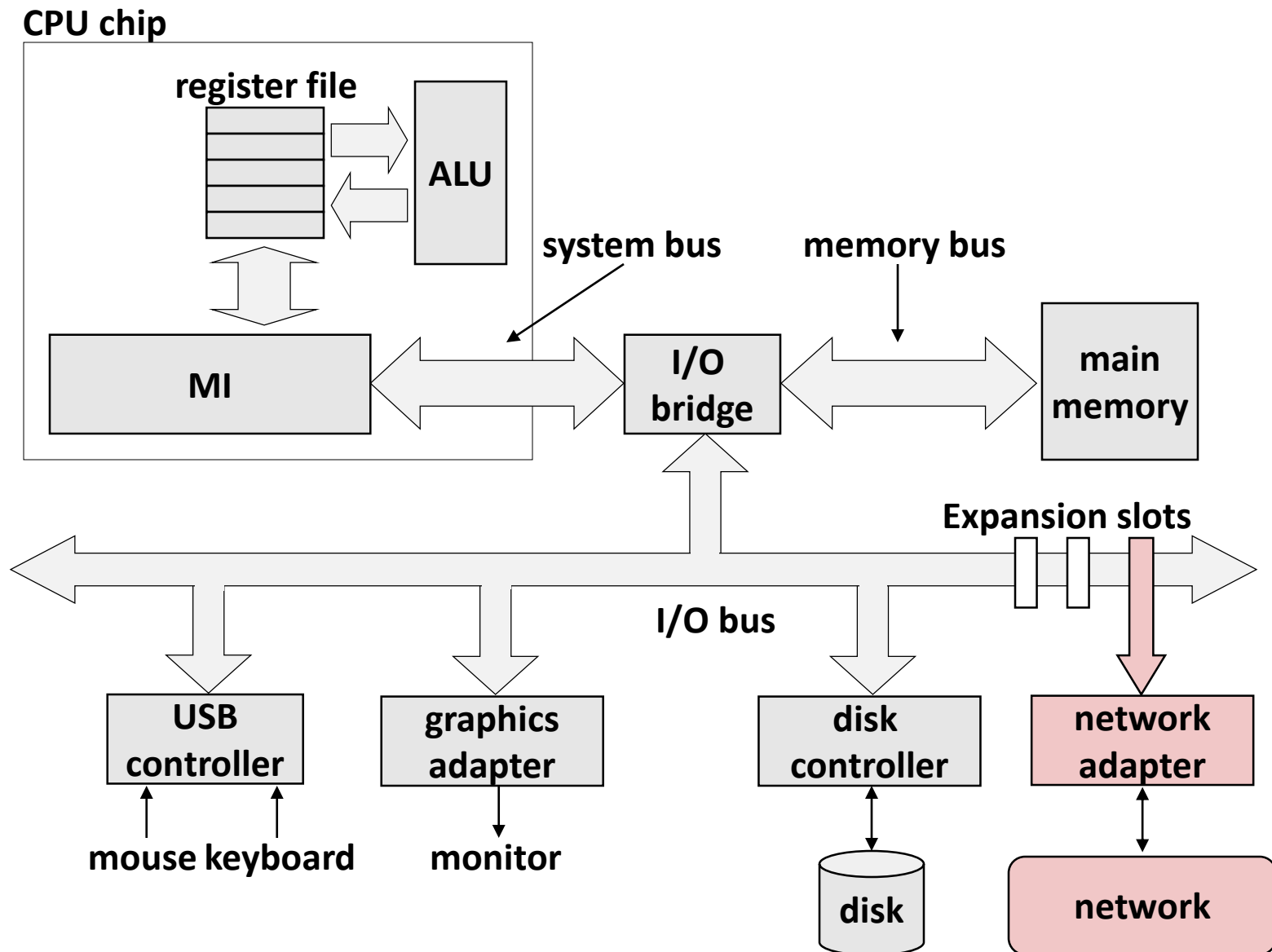~~Brian Railing~~

Dimitris Konomis

A Client-Server Transaction

- **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 activated by request from client (vending machine analogy)

1. Client sends request

Client process → **Server process** ↔ **Resource**

4. Client handles response

3. Server sends response

2. Server handles request

Note: clients and servers are processes running on hosts (can be the same or different hosts)

Hardware Organization of a Network Host

CPU chip

register file

ALU

system bus

memory bus

MI

I/O bridge

main memory

Expansion slots

I/O bus

USB controller

graphics adapter

disk controller

network adapter

mouse keyboard

monitor

disk

network

Computer Networks

- **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")

- **Let's see how an internet is built from the ground up**

Lowest Level: Ethernet Segment
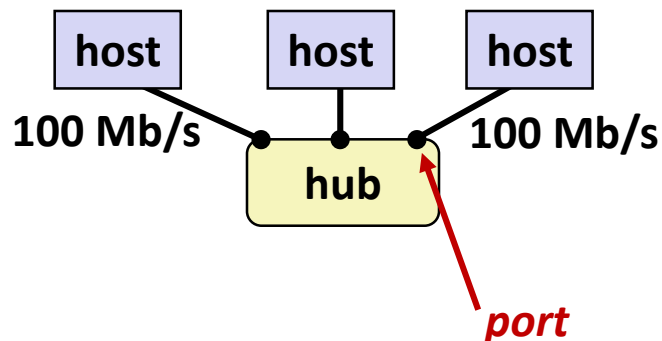
- **Ethernet segment consists of a collection of *hosts* connected by wires (twisted pairs) to a *hub***

- **Spans room or floor in a building**

- **Operation**
  - Each Ethernet adapter has a unique 48-bit address (MAC address)
    - E.g., 00:16:ea:e3:54:e6
  - Hosts send bits to any other host in chunks called *frames*
  - Hub slavishly copies each bit from each port to every other port
    - Every host sees every bit
   - Note: Hubs are obsolete. Bridges (switches, routers) became cheap enough to replace them

Next Level: Bridged Ethernet Segment
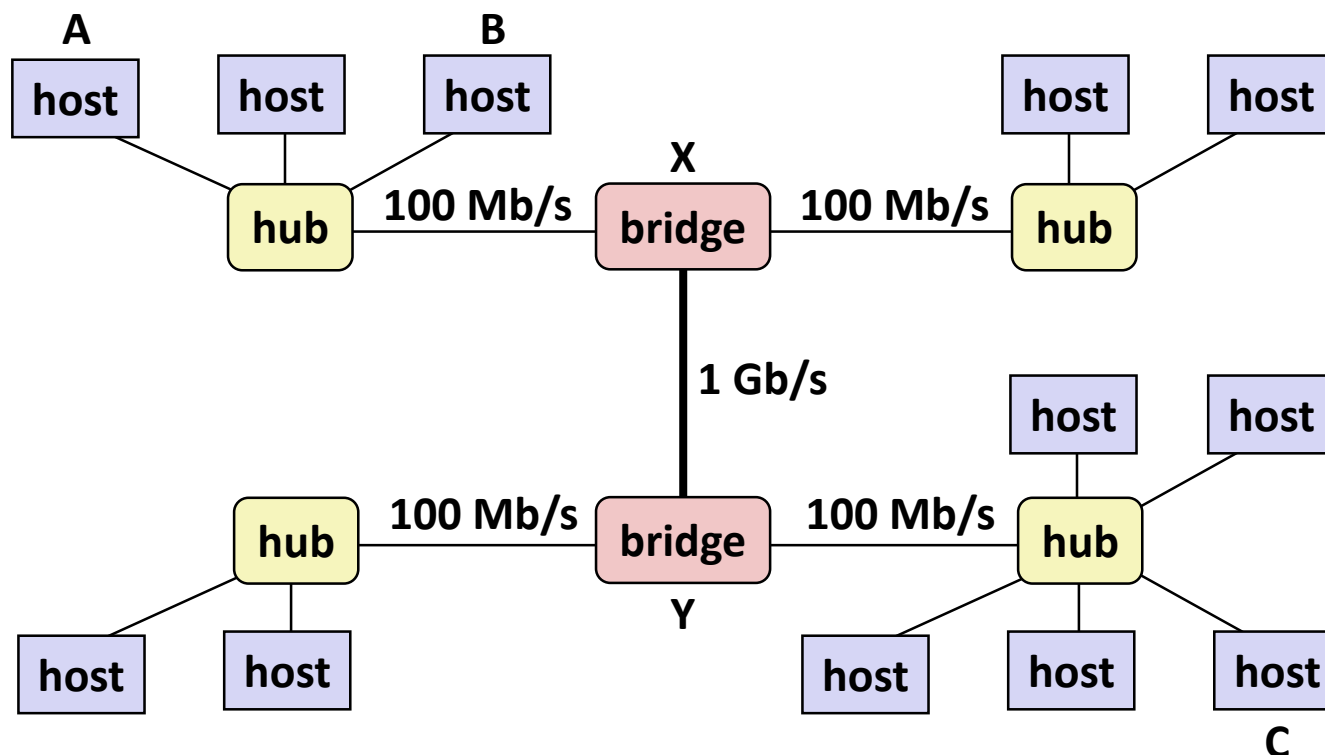
- **Spans building or campus**

- **Bridges cleverly learn which hosts are reachable from which ports and then selectively copy frames from port to port**

Conceptual View of LANs

- **For simplicity, hubs, bridges, and wires are often shown as a collection of hosts attached to a single wire:**
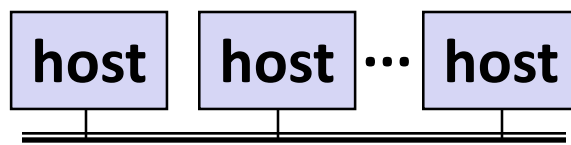
Next Level: internets

- **Multiple incompatible LANs can be physically connected by specialized computers called *routers***
- **The connected networks are called an *internet* (lower case)**
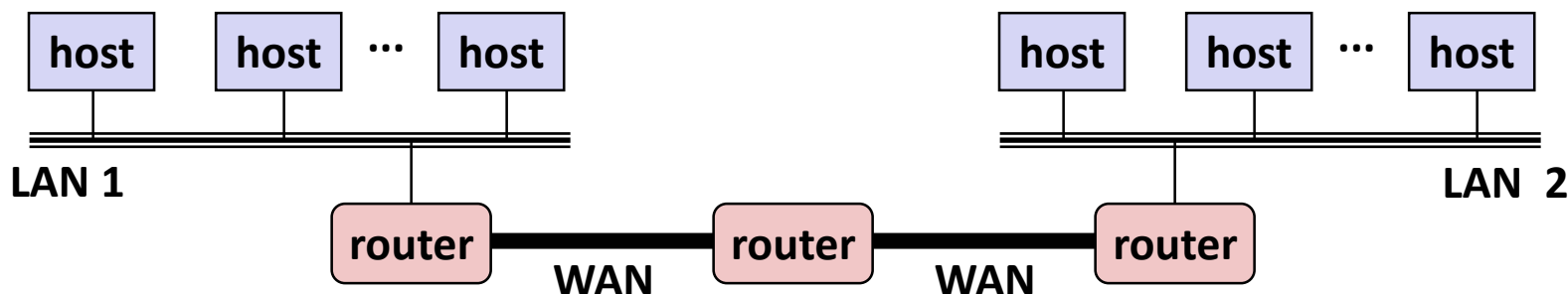
LAN 1 and LAN 2 might be completely different, totally incompatible

(e.g., Ethernet, Fibre Channel, 802.11*, T1-links, DSL, …)

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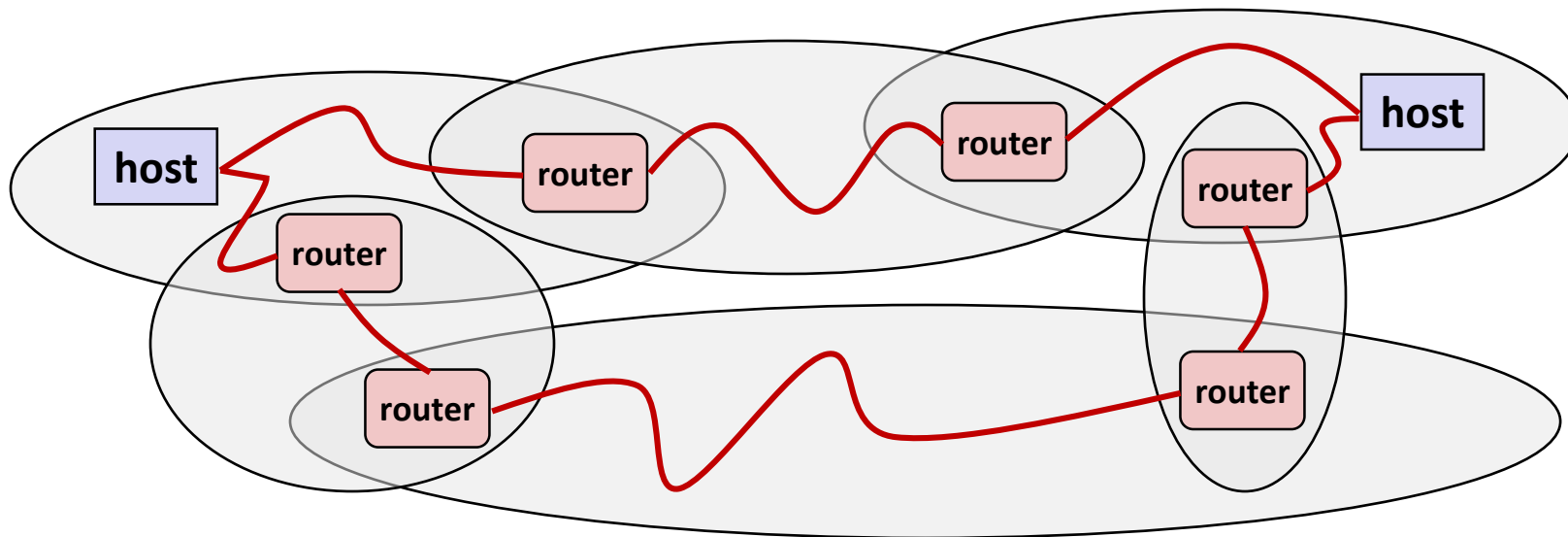
- **Ad hoc interconnection of networks**
  - No particular topology
  - Vastly different router & link capacities
- **Send packets from source to destination by hopping through networks**
  - Router forms bridge from one network to another
  - Different packets may take different routes

The Notion of an internet Protocol

- **How is it possible to send bits across incompatible LANs and WANs?**

- **Solution: *protocol* software running on each host and router**
  - Protocol is a set of rules that governs how hosts and routers should cooperate when they transfer data from network to network.
  - Smooths out the differences between the different networks

What Does an internet Protocol Do?

- **Provides a *naming scheme***
  - An internet protocol defines a uniform format for *host addresses*
  - Each host (and router) is assigned at least one of these internet addresses that uniquely identifies it

- **Provides a *delivery mechanism***
  - An internet protocol defines a standard transfer unit (*packet*)
  - Packet consists of *header* and *payload*
    - Header: contains info such as packet size, source and destination addresses
    - Payload: contains data bits sent from source host

Transferring internet Data Via Encapsulation

PH: Internet packet header

FH: LAN frame header
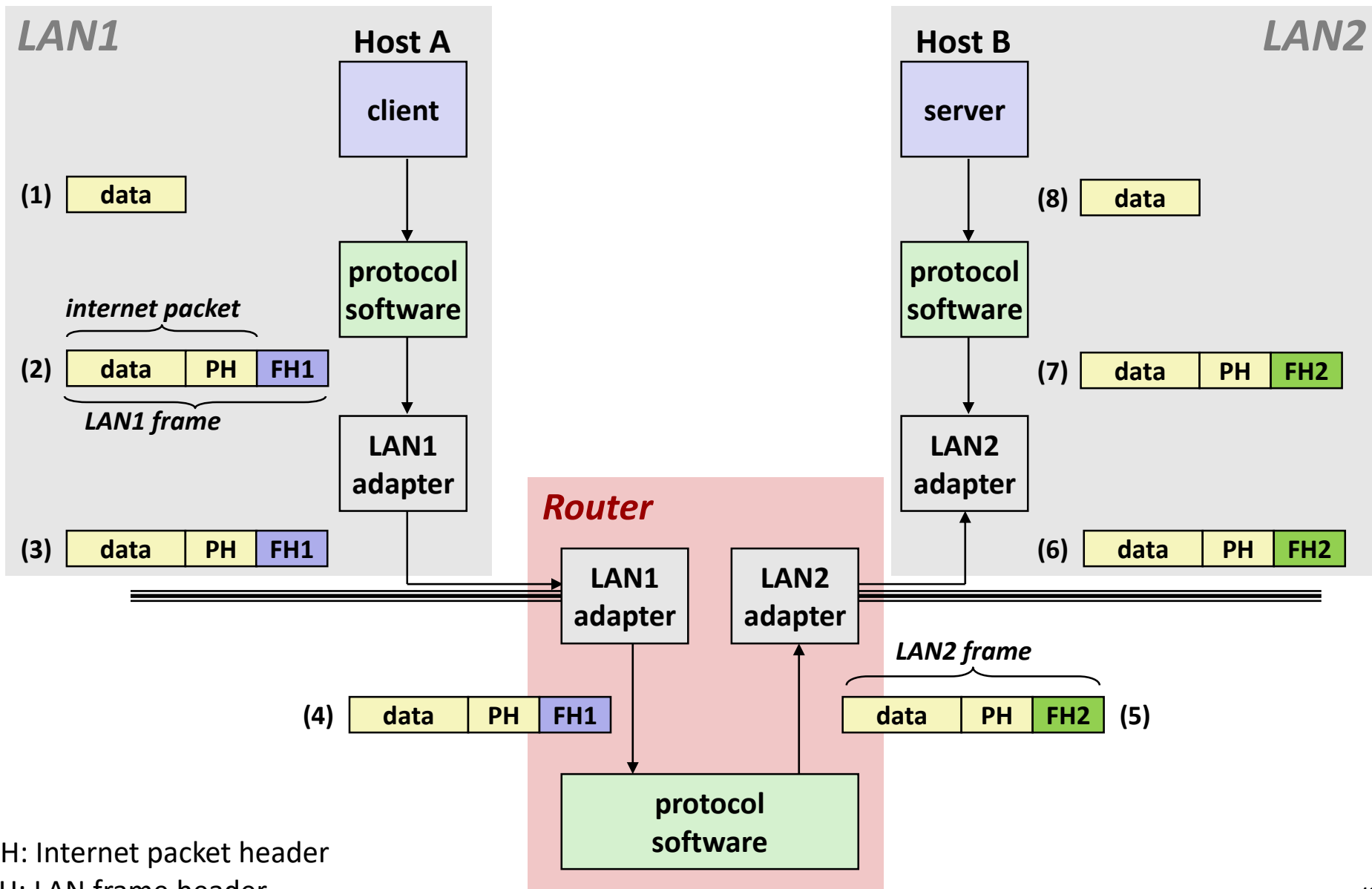
Other Issues

- **We are glossing over a number of important questions:**
    - What if different networks have different maximum frame sizes? (segmentation)
    - How do routers know where to forward frames?
    - How are routers informed when the network topology changes?
    - What if packets get lost?

- **These (and other) questions are addressed by the area of systems known as *computer networking***

Global IP Internet (upper case)

- **Most famous example of an internet**

- **Based on the TCP/IP protocol family**
  - IP (Internet Protocol) :
    - Provides *basic naming scheme* and unreliable *delivery capability* of packets (datagrams) from *host-to-host*
  - UDP (Unreliable Datagram Protocol)
    - Uses IP to provide *unreliable* datagram delivery from *process-to-process*
  - TCP (Transmission Control Protocol)
    - Uses IP to provide *reliable* byte streams from *process-to-process* over *connections*

- **Accessed via a mix of Unix file I/O and functions from the *sockets interface***

Hardware and Software Organization of an Internet Application

Internet client host

Internet server host

Client — *User code*

Sockets interface (system calls)

TCP/IP — *Kernel code*

Hardware interface (interrupts)

Network adapter — *Hardware and firmware*

Server

TCP/IP

Network adapter

Global IP Internet
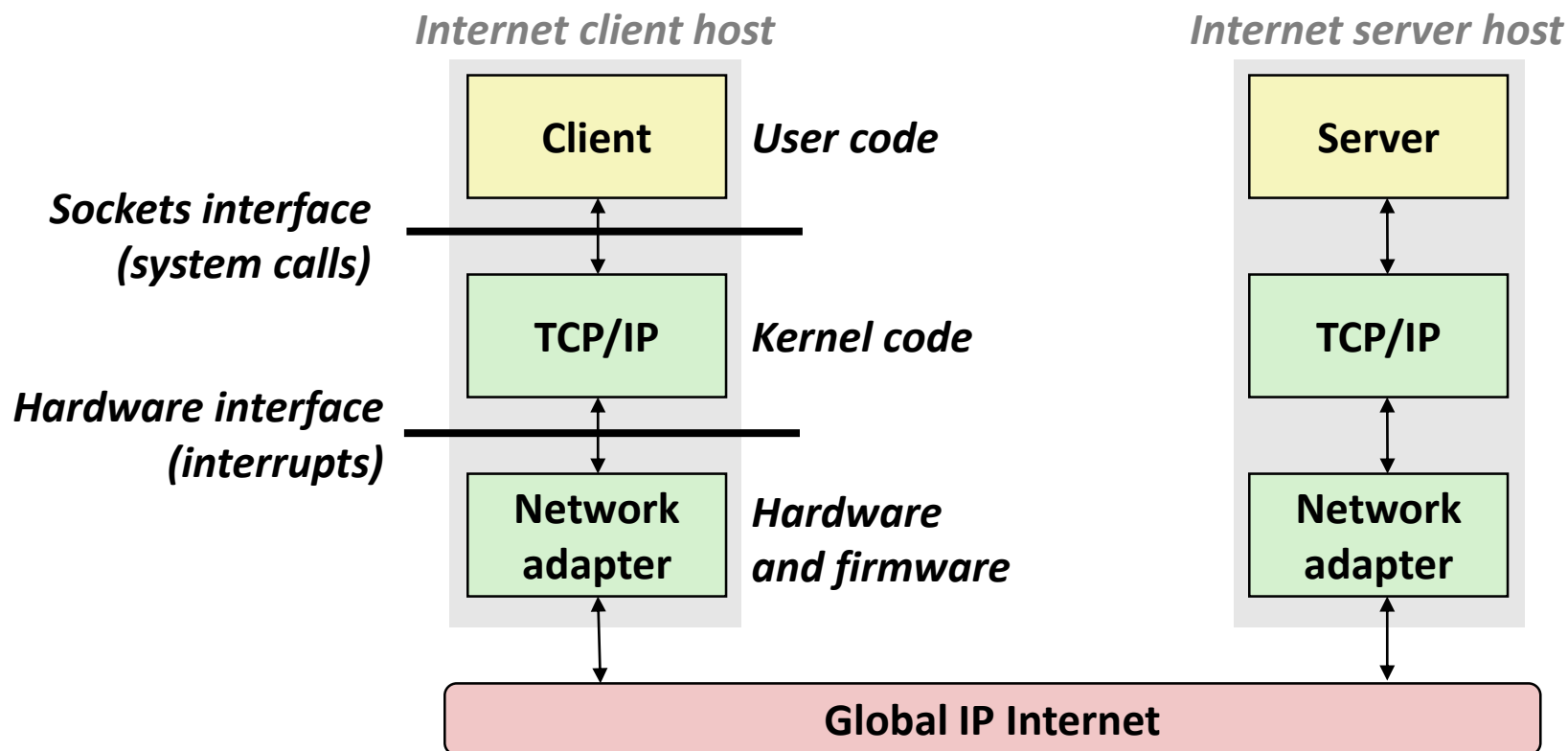
A Programmer's View of the Internet

1. Hosts are mapped to a set of 32-bit *IP addresses*

- 128.2.203.179

2. The set of IP addresses is mapped to a set of identifiers called Internet *domain names*

- 128.2.217.3 is mapped to www.cs.cmu.edu

3. A process on one Internet host can communicate with a process on another Internet host over a *connection*

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

- **Majority of Internet traffic still carried by IPv4**
  - But IPv6 is finally taking hold
  - % of users access Google services using IPv6:
    - Nov. 2014:   4%
    - Nov. 2015:   7%
    - Nov. 2016:   14%

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

(1) IP Addresses

- **32-bit IP addresses are stored in an *IP address struct***
  - IP addresses are always stored in memory in *network byte order* (big-endian byte order)
  - True in general for any integer transferred in a packet header from one machine to another.
    - E.g., the port number used to identify an Internet connection.

```
/* Internet address structure */
struct in_addr {
    uint32_t  s_addr; /* network byte order (big-endian) */
};
```

Dotted Decimal Notation

- **By convention, each byte in a 32-bit IP address is represented by its decimal value and separated by a period**
  - IP address: `0x8002C2F2 = 128.2.194.242`

- **Use `getaddrinfo` and `getnameinfo` functions (described later) to convert between IP addresses and dotted decimal format.**

(2) Internet Domain Names

unnamed root
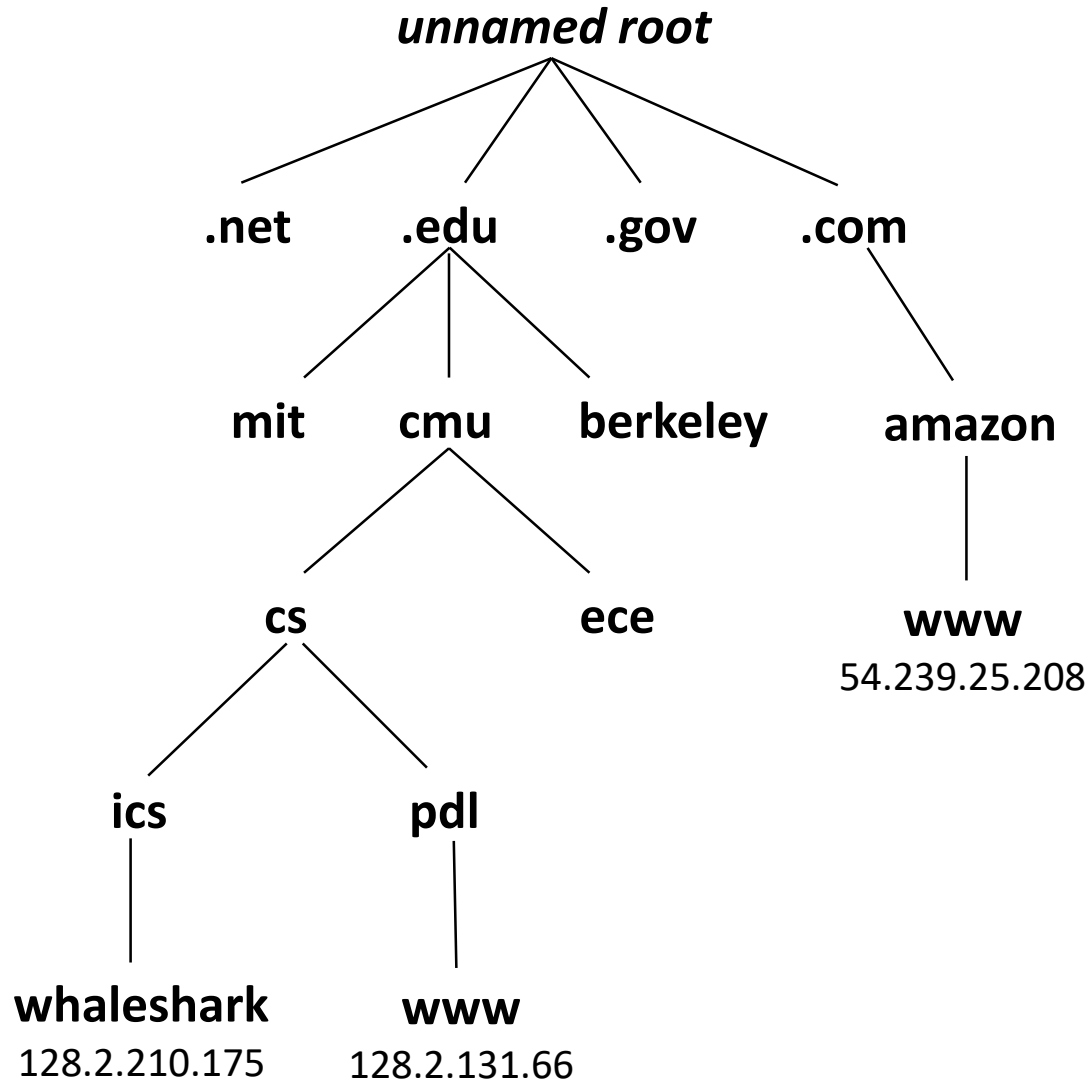
.net   .edu   .gov   .com   *First-level domain names*

mit   cmu   berkeley   amazon   *Second-level domain names*

cs   ece   www   *Third-level domain names*
54.239.25.208

ics   pdl

whaleshark   www
128.2.210.175   128.2.131.66

Domain Naming System (DNS)

- **The Internet maintains a mapping between IP addresses and domain names in a huge worldwide distributed database called *DNS***

- **Conceptually, programmers can view the DNS database as a collection of millions of *host entries.***
  - Each host entry defines the mapping between a set of domain names and IP addresses.
  - In a mathematical sense, a host entry is an equivalence class of domain names and IP addresses.

Properties of DNS Mappings

- **Can explore properties of DNS mappings using `nslookup`**

  - Output edited for brevity

- **Each host has a locally defined domain name `localhost` which always maps to the *loopback address* `127.0.0.1`**

  ```
  linux> nslookup localhost
  Address: 127.0.0.1
  ```

- **Use `hostname` to determine real domain name of local host:**

  ```
  linux> hostname
  whaleshark.ics.cs.cmu.edu
  ```

Properties of DNS Mappings (cont)

- **Simple case: one-to-one mapping between domain name and IP address:**

```
linux> nslookup whaleshark.ics.cs.cmu.edu
Address: 128.2.210.175
```

- **Multiple domain names mapped to the same IP address:**

```
linux> nslookup cs.mit.edu
Address: 18.62.1.6
linux> nslookup eecs.mit.edu
Address: 18.62.1.6
```

Properties of DNS Mappings (cont)

- **Multiple domain names mapped to multiple IP addresses:**

```
linux> nslookup www.twitter.com
Address: 104.244.42.65
Address: 104.244.42.129
Address: 104.244.42.193
Address: 104.244.42.1

linux> nslookup www.twitter.com
Address: 104.244.42.129
Address: 104.244.42.65
Address: 104.244.42.193
Address: 104.244.42.1
```

- **Some valid domain names don't map to any IP address:**

```
linux> nslookup ics.cs.cmu.edu
*** Can't find ics.cs.cmu.edu: No answer
```

(3) Internet Connections

- **Clients and servers communicate by sending streams of bytes over *connections*. Each connection is:**
  - *Point-to-point*: connects a pair of processes.
  - *Full-duplex*: data can flow in both directions at the same time,
  - *Reliable*: stream of bytes sent by the source is eventually received by the destination in the same order it was sent.

- **A *socket* is an endpoint of a connection**
  - *Socket address* is an `IPaddress:port` pair

- **A *port* is 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)`

Client socket address
128.2.194.242:51213

Server socket address
208.216.181.15:80

Client

Server (port 80)

Connection socket pair
(128.2.194.242:51213, 208.216.181.15:80)

Client host address
128.2.194.242

Server host address
208.216.181.15
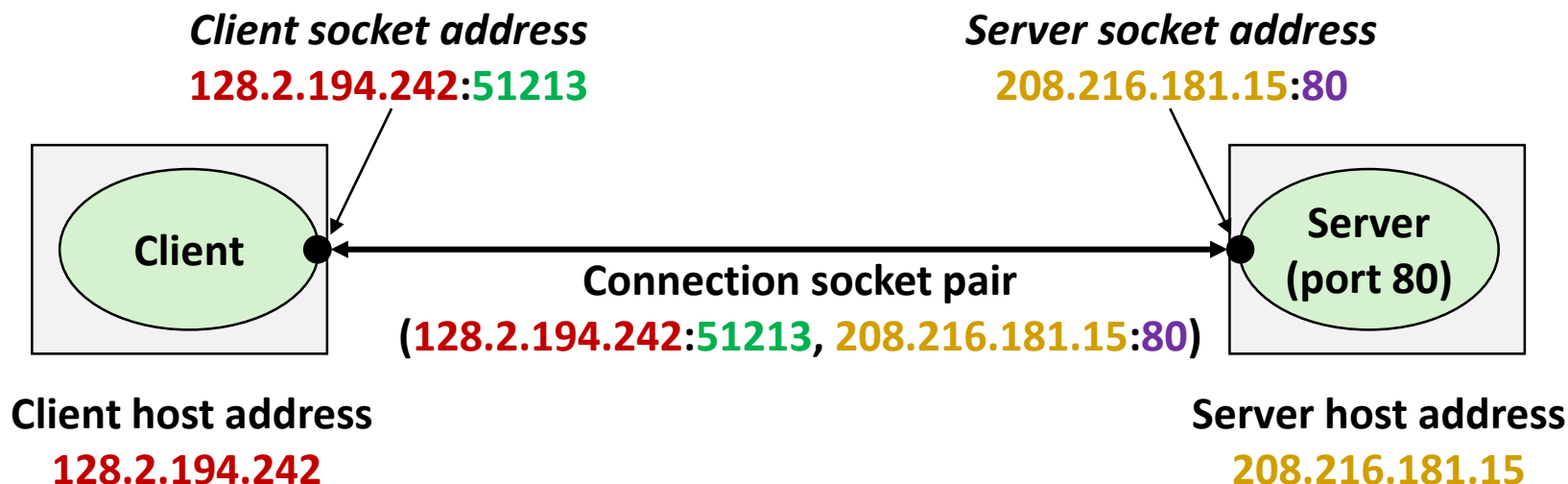
51213 is an ephemeral port allocated by the kernel

80 is a well-known port associated with Web servers

Using Ports to Identify Services

Server host 128.2.194.242

Client host

Service request for
128.2.194.242:80
(i.e., the Web server)

Client → Kernel → Web server (port 80)

Echo server (port 7)

Service request for
128.2.194.242:7
(i.e., the echo server)

Client → Kernel

Web server (port 80)

Kernel → Echo server (port 7)
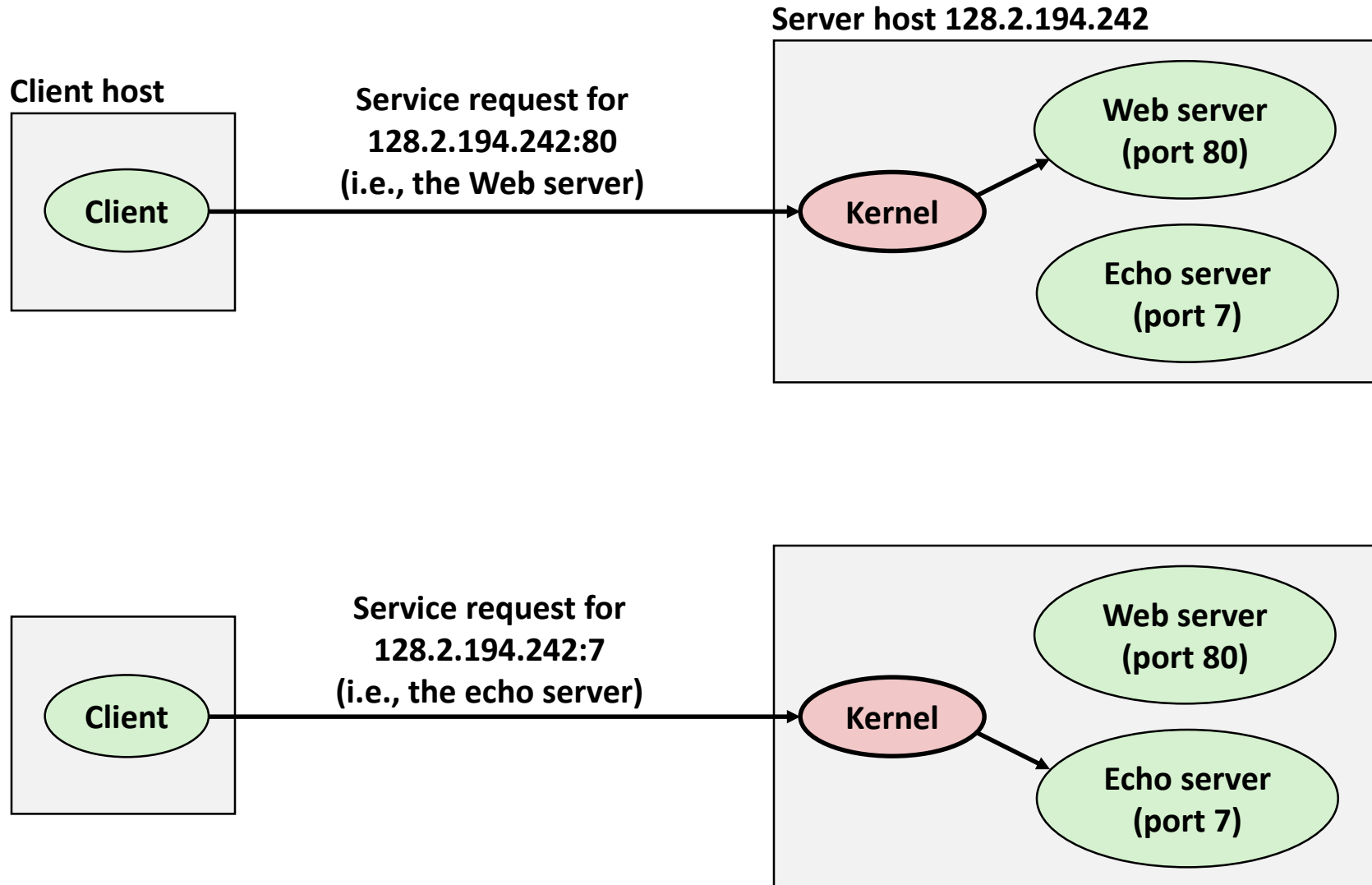
Sockets Interface

- **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

- **What is a socket?**
  - 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
    - *Remember:* 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**

Client ● ←———————→ ● Server

`clientfd`   `serverfd`

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

Socket Programming Example

- **Echo server and client**
- **Server**
  - Accepts connection request
  - Repeats back lines as they are typed
- **Client**
  - Requests connection to server
  - Repeatedly:
    - Read line from terminal
    - Send to server
    - Read reply from server
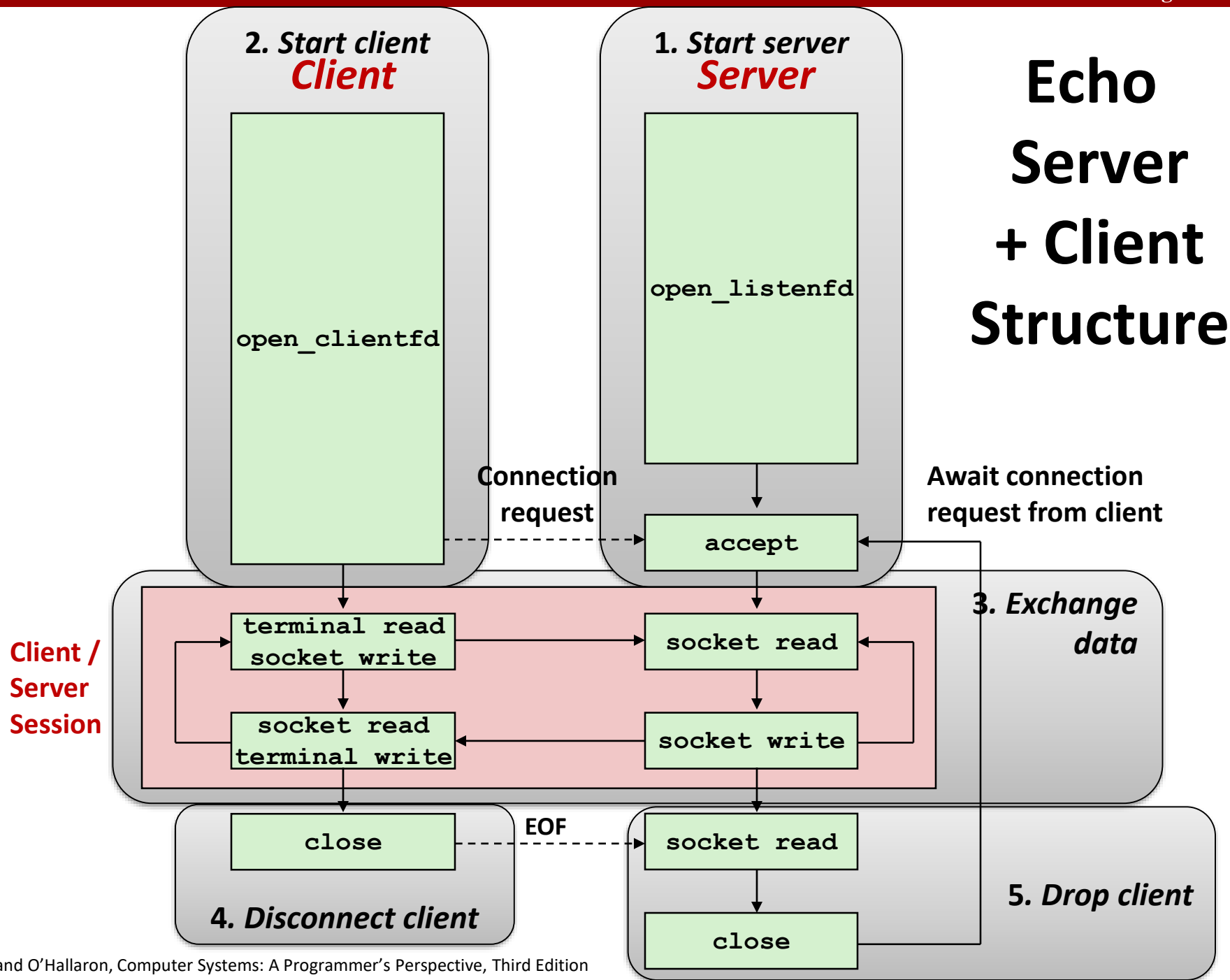   - Print line to terminal

Echo Server/Client Session Example

Client

```
bambooshark: ./echoclient whaleshark.ics.cs.cmu.edu 6616          (A)
This line is being echoed                                        (B)
This line is being echoed
This one is, too                                                 (C)
This one is, too
^D
bambooshark: ./echoclient whaleshark.ics.cs.cmu.edu 6616          (D)
This one is a new connection                                     (E)
This one is a new connection
^D
```

Server

```
whaleshark: ./echoserveri 6616
Connected to (BAMBOOSHARK.ICS.CS.CMU.EDU, 33707)         (A)
server received 26 bytes                                 (B)
server received 17 bytes                                 (C)
Connected to (BAMBOOSHARK.ICS.CS.CMU.EDU, 33708)         (D)
server received 29 bytes                                 (E)
```

Echo Server + Client Structure

2. *Start client*
Client

1. *Start server*
Server

`open_clientfd`

`open_listenfd`

Connection request

Await connection request from client

`accept`

3. *Exchange data*

Client / Server Session

| `terminal read` `socket write` | → | `socket read` |

| `socket read` `terminal write` | ← | `socket write` |

`close`

EOF

`socket read`

4. *Disconnect client*

5. *Drop client*

`close`

Echo Server + Client Structure

2. *Start client*
Client

open_clientfd

1. *Start server*
Server

open_listenfd

Connection request

Await connection request from client

accept

3. *Exchange data*

Client / Server Session

fgets
rio_writen

rio_readlineb

rio_readlineb
fputs

rio_writen

close

EOF

rio_readlineb

4. *Disconnect client*

5. *Drop client*

close

Echo Client: Main Routine

```c
#include "csapp.h"

int main(int argc, char **argv)
{
    int clientfd;
    char *host, *port, buf[MAXLINE];
    rio_t rio;

    host = argv[1];
    port = argv[2];

    clientfd = Open_clientfd(host, port);
    Rio_readinitb(&rio, clientfd);

    while (Fgets(buf, MAXLINE, stdin) != NULL) {
        Rio_writen(clientfd, buf, strlen(buf));
        Rio_readlineb(&rio, buf, MAXLINE);
        Fputs(buf, stdout);
    }
    Close(clientfd);
    exit(0);
}
```

echoclient.c

Iterative Echo Server: Main Routine

```c
#include "csapp.h"
void echo(int connfd);

int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr; /* Enough room for any addr */
    char client_hostname[MAXLINE], client_port[MAXLINE];

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage); /* Important! */
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        Getnameinfo((SA *) &clientaddr, clientlen,
                    client_hostname, MAXLINE, client_port, MAXLINE, 0);
        printf("Connected to (%s, %s)\n", client_hostname, client_port);
        echo(connfd);
        Close(connfd);
    }
    exit(0);
}
```

echoserveri.c

Echo Server: `echo` function

- **The server uses RIO to read and echo text lines until EOF (end-of-file) condition is encountered.**
  - EOF condition caused by client calling `close(clientfd)`

```c
void echo(int connfd)
{
    size_t n;
    char buf[MAXLINE];
    rio_t rio;

    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        printf("server received %d bytes\n", (int)n);
        Rio_writen(connfd, buf, n);
    }
}
```
echo.c

Socket Address Structures

- **Generic socket address:**
  - For address arguments to **connect**, **bind**, and **accept**
  - Necessary only because C did not have generic (**void ***) pointers when the sockets interface was designed
  - For casting convenience, we adopt the Stevens convention:

    typedef struct sockaddr SA;

```
struct sockaddr {
  uint16_t  sa_family;    /* Protocol family */
  char      sa_data[14];  /* Address data.  */
};
```

sa_family

Family Specific
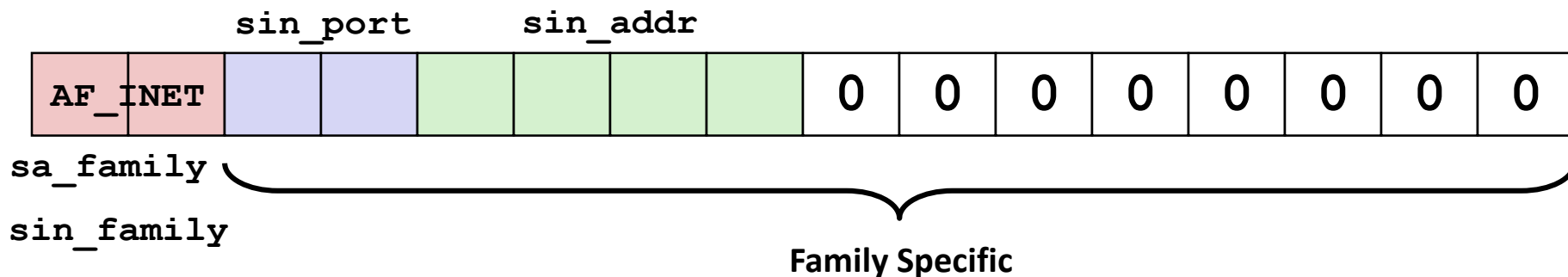
Socket Address Structures

- **Internet (IPv4) 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) */
};
```


sin_port   sin_addr

| AF_INET | | | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

sa_family

sin_family

Family Specific

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.

Host and Service Conversion: `getaddrinfo`
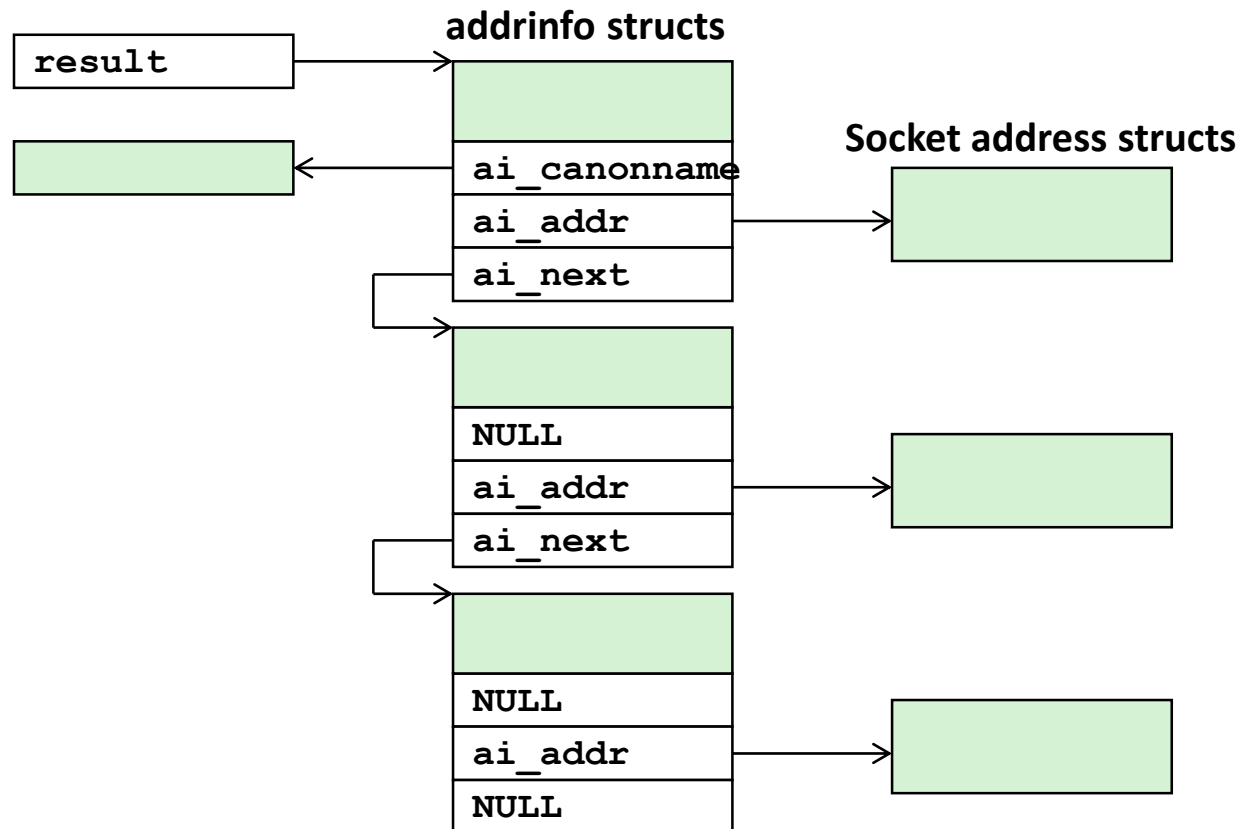
```
int getaddrinfo(const char *host,          /* Hostname or address */
                const char *service,       /* Port or service name */
                const struct addrinfo *hints,/* Input parameters */
                struct addrinfo **result);   /* Output linked list */

void freeaddrinfo(struct addrinfo *result);  /* Free linked list */

const char *gai_strerror(int errcode);       /* Return error msg */
```

- **Given `host` and `service`, `getaddrinfo` returns `result` that points to a linked list of `addrinfo` structs, each of which points to a corresponding socket address struct, and which contains arguments for the sockets interface functions.**

- **Helper functions:**
  - `freeadderinfo` frees the entire linked list.
  - `gai_strerror` converts error code to an error message.

Linked List Returned by `getaddrinfo`

addrinfo structs

result	

ai_canonname	
ai_addr	
ai_next	

Socket address structs

NULL	
ai_addr	
ai_next	

NULL	
ai_addr	
NULL	

- **Clients: walk this list, trying each socket address in turn, until the calls to `socket` and `connect` succeed.**

- **Servers: walk the list until calls to `socket` and `bind` succeed.**

`addrinfo` Struct

```
struct addrinfo {
    int                 ai_flags;      /* Hints argument flags */
    int                 ai_family;     /* First arg to socket function */
    int                 ai_socktype;   /* Second arg to socket function */
    int                 ai_protocol;   /* Third arg to socket function  */
    char               *ai_canonname;  /* Canonical host name */
    size_t              ai_addrlen;    /* Size of ai_addr struct */
    struct sockaddr    *ai_addr;       /* Ptr to socket address structure */
    struct addrinfo    *ai_next;       /* Ptr to next item in linked list */
};
```

- **Each addrinfo struct returned by getaddrinfo contains arguments that can be passed directly to `socket` function.**

- **Also points to a socket address struct that can be passed directly to `connect` and `bind` functions.**

Host and Service Conversion: `getnameinfo`

- **`getnameinfo` is the inverse of getaddrinfo, converting a socket address to the corresponding host and service.**
  - Replaces obsolete `gethostbyaddr` and `getservbyport` funcs.
  - Reentrant and protocol independent.

```
int getnameinfo(const SA *sa, socklen_t salen, /* In: socket addr */
                char *host, size_t hostlen,    /* Out: host */
                char *serv, size_t servlen,    /* Out: service */
                int flags);                     /* optional flags */
```

Conversion Example

```
#include "csapp.h"

int main(int argc, char **argv)
{
    struct addrinfo *p, *listp, hints;
    char buf[MAXLINE];
    int rc, flags;

    /* Get a list of addrinfo records */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;       /* IPv4 only */
    hints.ai_socktype = SOCK_STREAM; /* Connections only */
    if ((rc = getaddrinfo(argv[1], NULL, &hints, &listp)) != 0) {
        fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(rc));
        exit(1);
    }
```

hostinfo.c

Conversion Example (cont)

```
    /* Walk the list and display each IP address */
    flags = NI_NUMERICHOST; /* Display address instead of name */
    for (p = listp; p; p = p->ai_next) {
        Getnameinfo(p->ai_addr, p->ai_addrlen,
                    buf, MAXLINE, NULL, 0, flags);
        printf("%s\n", buf);
    }

    /* Clean up */
    Freeaddrinfo(listp);

    exit(0);
}
                                                          hostinfo.c
```

Running hostinfo

```
whaleshark> ./hostinfo localhost
127.0.0.1

whaleshark> ./hostinfo whaleshark.ics.cs.cmu.edu
128.2.210.175

whaleshark> ./hostinfo twitter.com
199.16.156.230
199.16.156.38
199.16.156.102
199.16.156.198
```

Next time

- **Using `getaddrinfo` for host and service conversion**
- **Writing clients and servers**
- **Writing Web servers!**

Additional slides

Basic Internet Components

- **Internet backbone:**
  - collection of routers (nationwide or worldwide) connected by high-speed point-to-point networks

- **Internet Exchange Points (IXP):**
  - router that connects multiple backbones (often referred to as peers)
  - Also called Network Access Points (NAP)

- **Regional networks:**
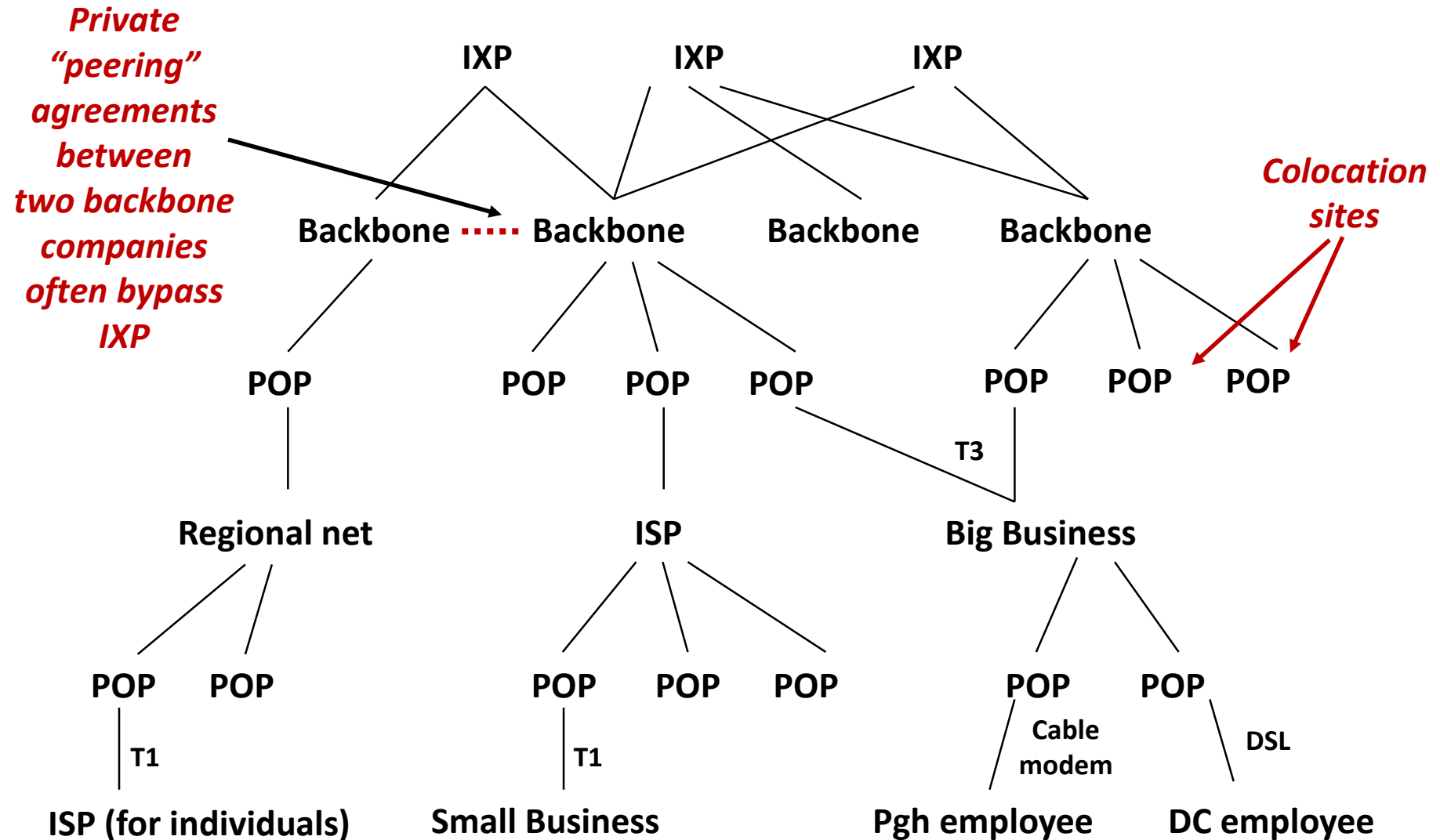 - smaller backbones that cover smaller geographical areas (e.g., cities or states)

- **Point of presence (POP):**
  - machine that is connected to the Internet

- **Internet Service Providers (ISPs):**
  - provide dial-up or direct access to POPs

Internet Connection Hierarchy

Private "peering" agreements between two backbone companies often bypass IXP

IXP   IXP   IXP

Colocation sites

Backbone ···· Backbone   Backbone   Backbone

POP   POP POP POP   POP POP POP

T3

Regional net   ISP   Big Business

POP POP   POP POP POP   POP POP

T1   T1   Cable modem   DSL

ISP (for individuals)   Small Business   Pgh employee   DC employee

IP Address Structure

- ## IP (V4) Address space divided into classes:

	0 1 2 3	8	16	24	31
Class A	**0**	Net ID		Host ID	
Class B	**1** **0**		Net ID	Host ID	
Class C	**1** **1** **0**			Net ID	Host ID
Class D	**1** **1** **1** **0**	Multicast address			
Class E	**1** **1** **1** **1**	Reserved for experiments			

- ## Network ID Written in form w.x.y.z/n

  - n = number of bits in host address
  - E.g., CMU written as 128.2.0.0/16
    - Class B address

- ## Unrouted (private) IP addresses:

  10.0.0.0/8   172.16.0.0/12   192.168.0.0/16

Evolution of Internet

- **Original Idea**
  - Every node on Internet would have unique IP address
    - Everyone would be able to talk directly to everyone
  - No secrecy or authentication
    - Messages visible to routers and hosts on same LAN
    - Possible to forge source field in packet header

- **Shortcomings**
  - There aren't enough IP addresses available
  - Don't want everyone to have access or knowledge of all other hosts
  - Security issues mandate secrecy & authentication

# Evolution of Internet: Naming

- **Dynamic address assignment**
  - Most hosts don't need to have known address
    - Only those functioning as servers
  - DHCP (Dynamic Host Configuration Protocol)
    - Local ISP assigns address for temporary use

- **Example:**
  - Laptop at CMU (wired connection)
    - IP address 128.2.213.29 (`bryant-tp4.cs.cmu.edu`)
    - Assigned statically
  - Laptop at home
    - IP address 192.168.1.5
    - Only valid within home network

Evolution of Internet: Firewalls

176.3.3.3

10.2.2.2

Firewall

1

4

2

3

Corporation X

216.99.99.99
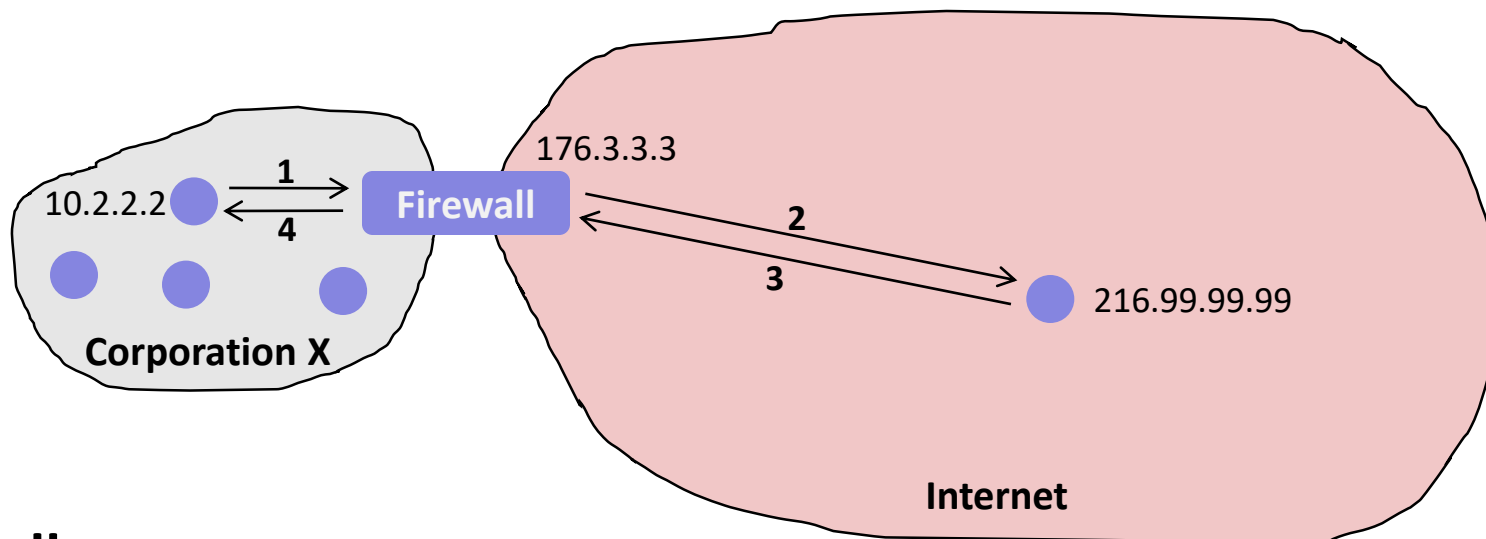
Internet

- **Firewalls**
  - Hides organizations nodes from rest of Internet
  - Use local IP addresses within organization
  - For external service, provides proxy service
    1. Client request: src=10.2.2.2, dest=216.99.99.99
    2. Firewall forwards: src=176.3.3.3, dest=216.99.99.99
    3. Server responds: src=216.99.99.99, dest=176.3.3.3
    4. Firewall forwards response: src=216.99.99.99, dest=10.2.2.2