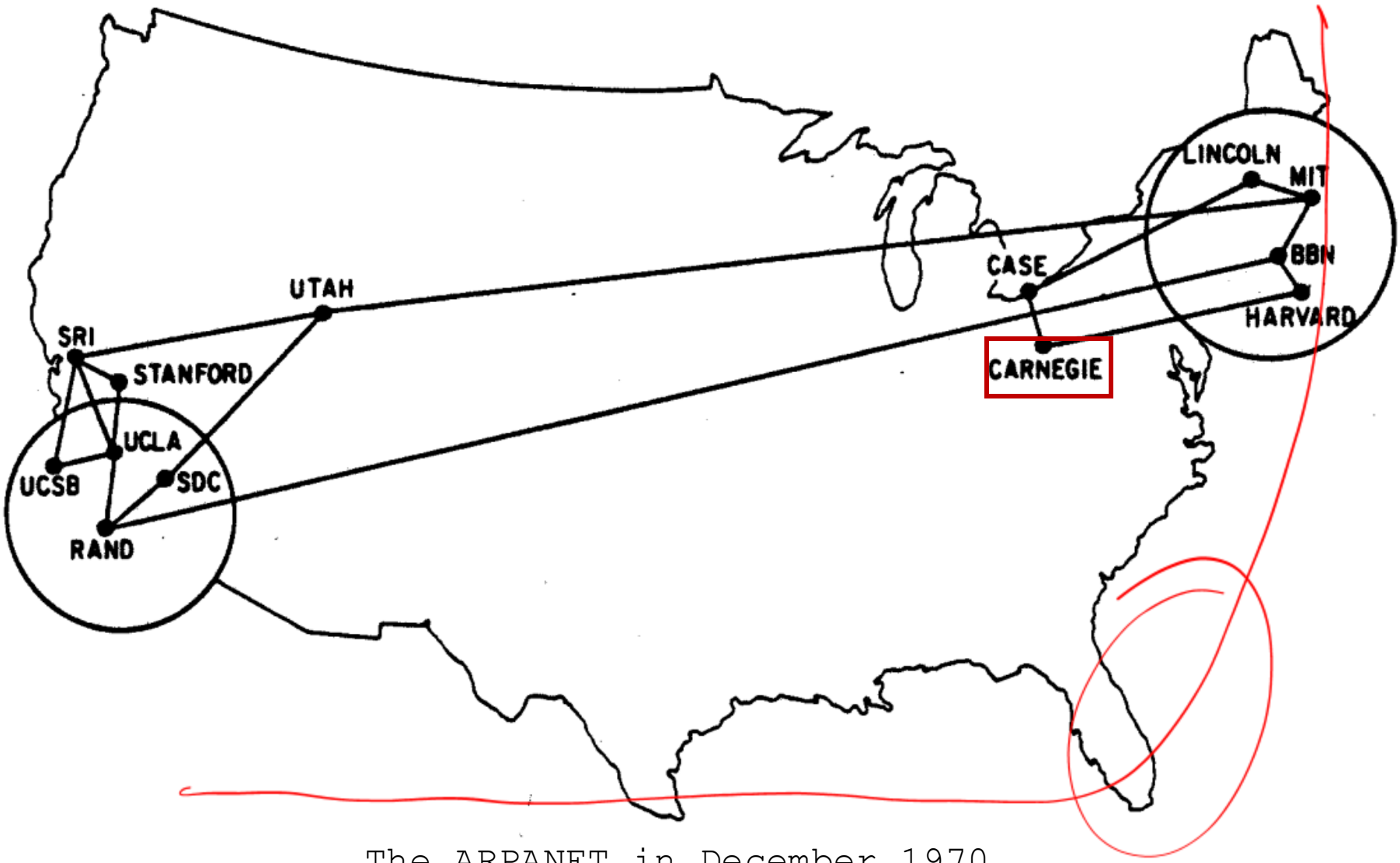

WELCOME
TO
15-213
14-513
18-213
18-613
15-513

Network Programming

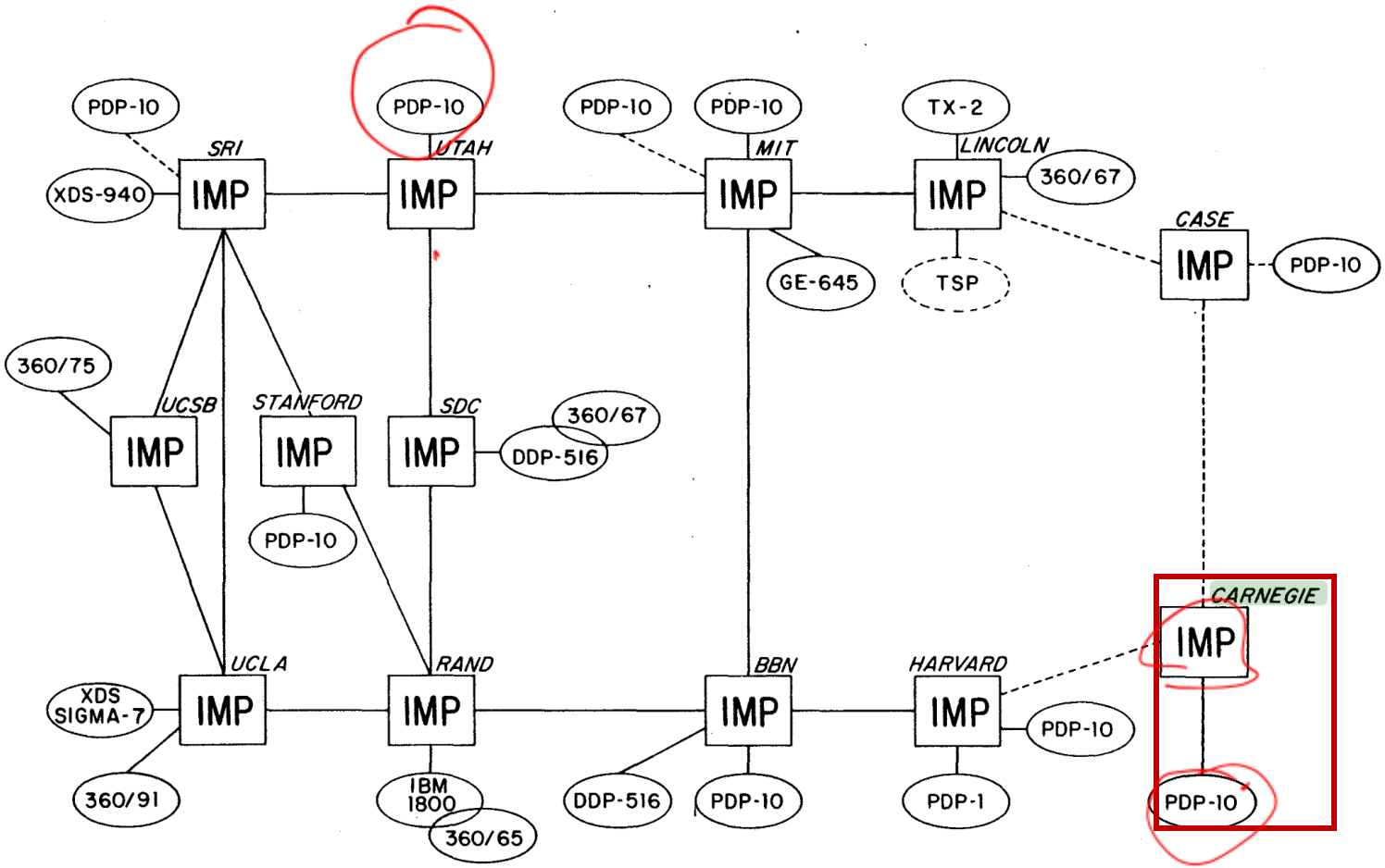
15-213/15-513/14-513: Introduction to Computer Systems
19st Lecture, November 10, 2026

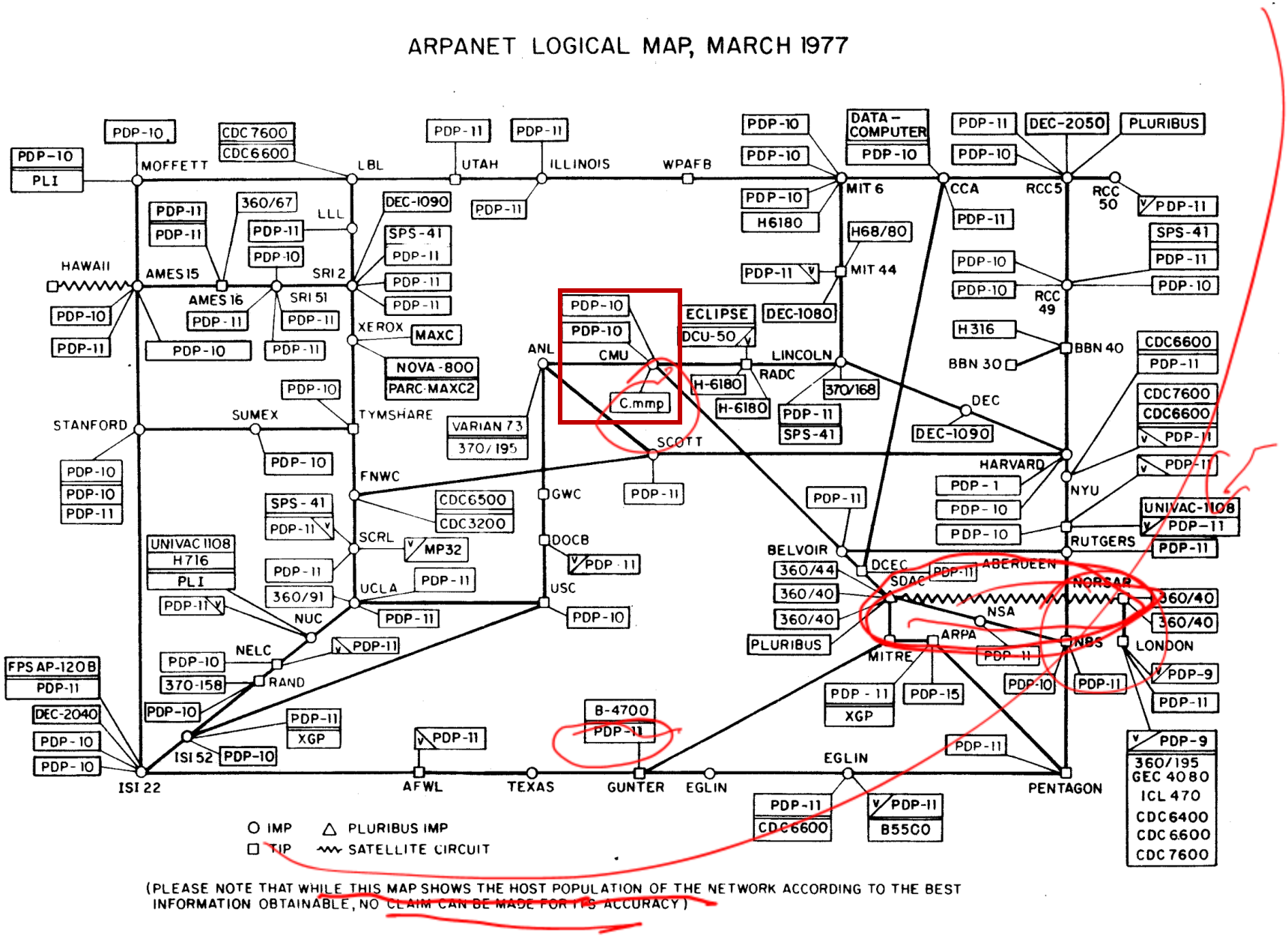


The ARPANET in December 1969

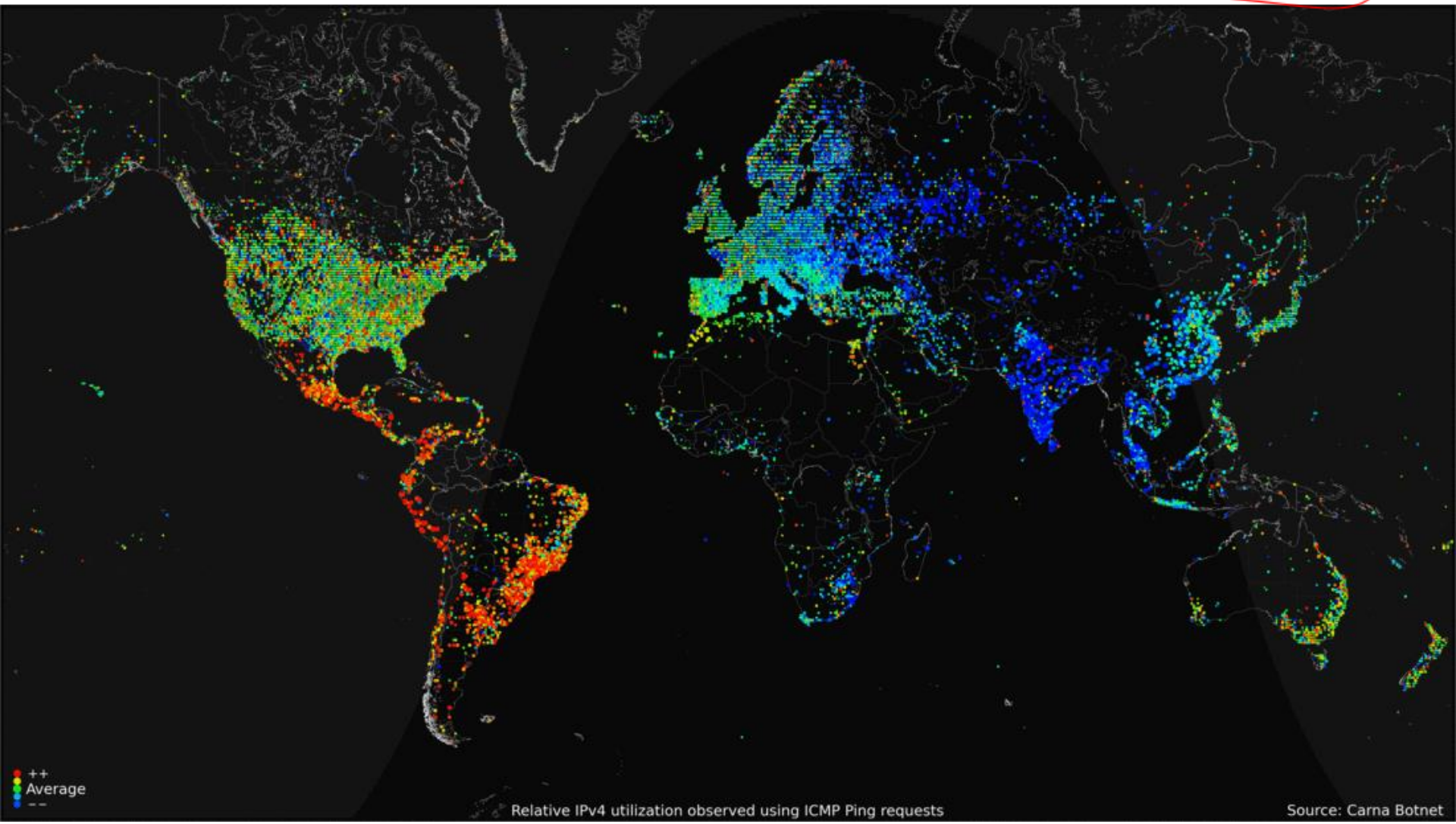


The ARPANET in December 1970


PDP-10
SRI
XDS-940
IMP
PDP-10
UTAH
IMP
PDP-10
PDP-10
MIT
IMP
GE-645
TX-2
LINCOLN
360/67
IMP
TSP
CASE
IMP
PDP-10
360/75
UCSB
IMP
STANFORD
IMP
PDP-10
SDC
IMP
360/67
DDP-516
CARNEGIE
IMP
PDP-10
UCLA
XDS SIGMA-7
IMP
360/91
RAND
IMP
IBM 1800
360/65
BBN
IMP
DDP-516
PDP-10
HARVARD
IMP
PDP-10
PDP-1
ARPA NET, DECEMBER 1970


ARPANET LOGICAL MAP, MARCH 1977
MOFFETT
LBL
UTAH
ILLINOIS
WPAFB
MIT 6
CCA
RCC 5
RCC 50
HAWAII
AMES 15
AMES 16
SRI 2
SRI 51
LLL
XEROX
MAXC
NOVA-800
PARC-MAXC2
ANL
CMU
C.mmp
ECLIPSE
DCU-50
LINCOLN
RADC
MIT 44
RCC 49
H 316
BBN 40
BBN 30
STANFORD
SUMEX
TYMSHARE
VARIAN 73
370/195
SCOTT
DEC
DEC-1090
HARVARD
NYU
RUTGERS
FNWC
GWC
SCRL
MP32
DOCB
UCLA
USC
NUC
NELC
RAND
BELVOIR
DCEC
SDAC
ABERDEEN
NORSAR
NSA
ARPA
MITRE
NBS
LONDON
PLURIBUS
ISI 22
ISI 52
AFWL
TEXAS
GUNTER
EGLIN
PENTAGON
IMP
TIP
PLURIBUS IMP
SATELLITE CIRCUIT
(PLEASE NOTE THAT WHILE THIS MAP SHOWS THE HOST POPULATION OF THE NETWORK ACCORDING TO THE BEST INFORMATION OBTAINABLE, NO CLAIM CAN BE MADE FOR ITS ACCURACY)

A Map of 460 Billion Device Connections to the Internet collected by the Carna Botnet



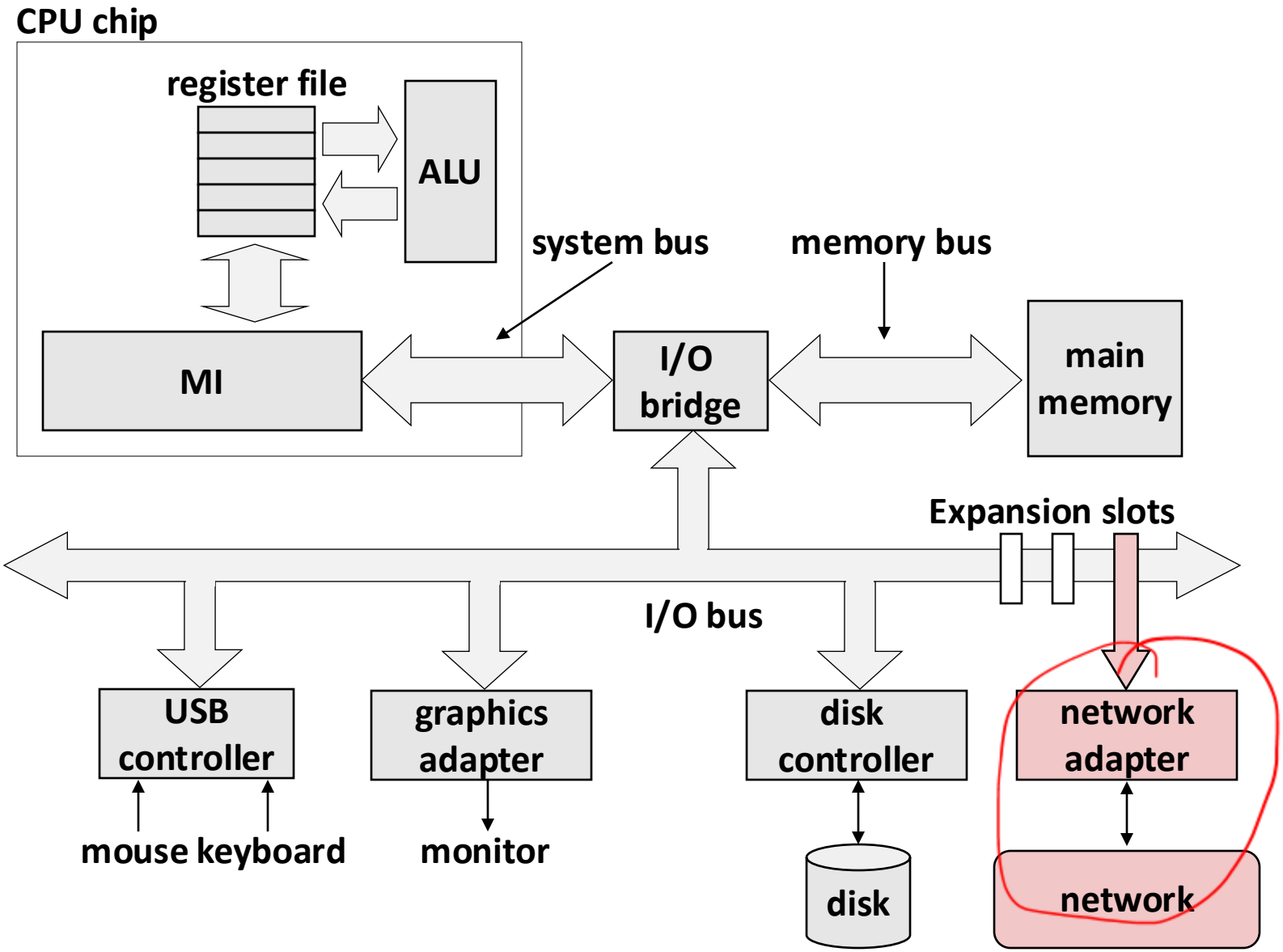
Outline

- Network types and structures
- Locating a host
- Setting up a connection
- HTTP Example

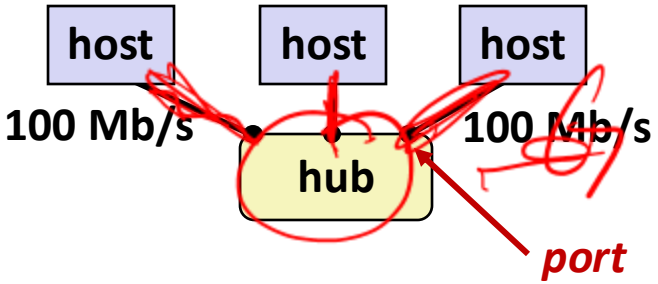
Computer Networks

- **A *network* is a hierarchical system of boxes and wires organized by geographical proximity**
  - SAN (System Area Network) spans a rack or a room
  - 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 (mostly optical) links
  - Also: SAN (Storage area network), MAN (Metropolitan), etc., etc.
- **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**

Hardware Organization of a Network Host

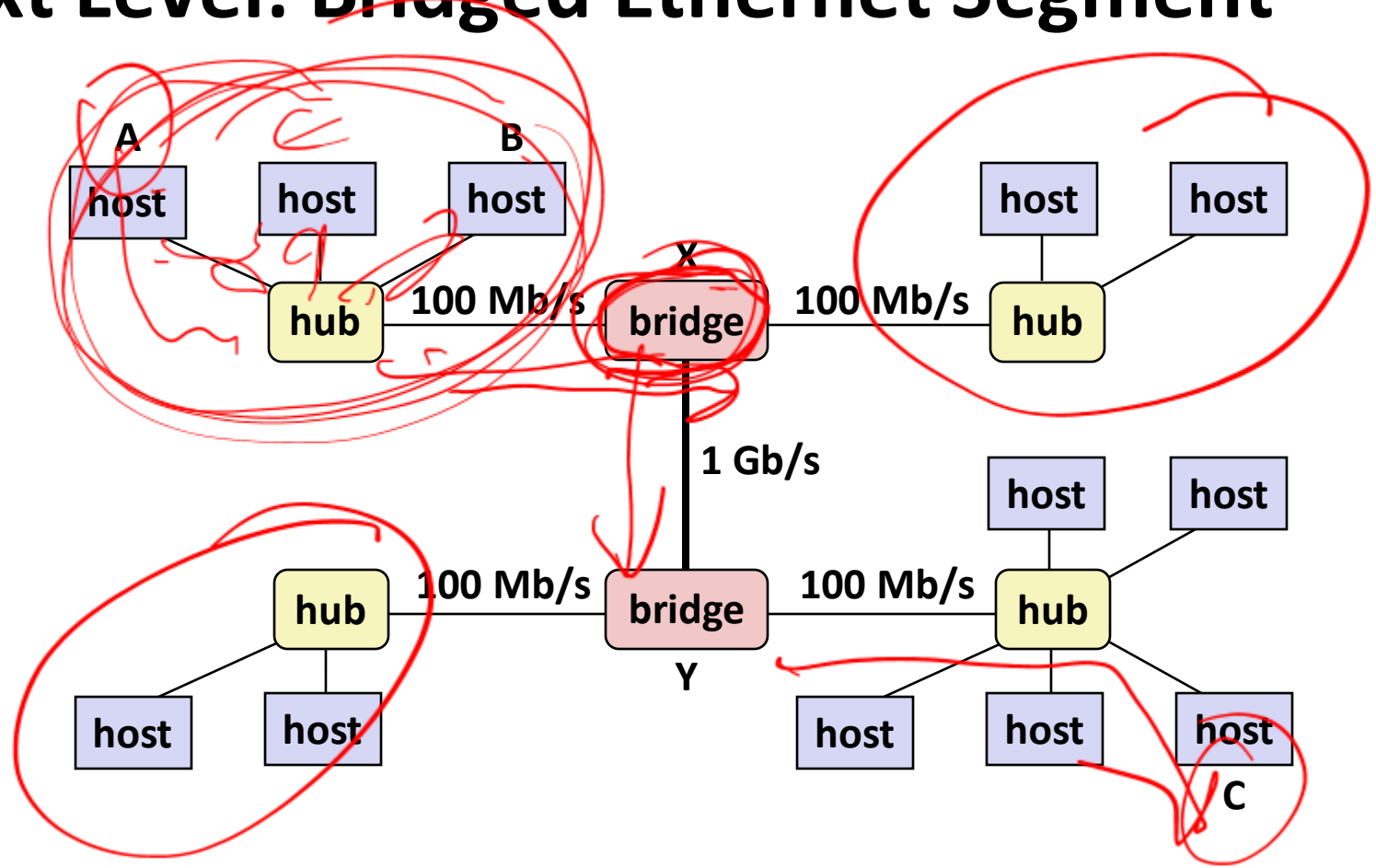


Example Lowest Level: Ethernet



- 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

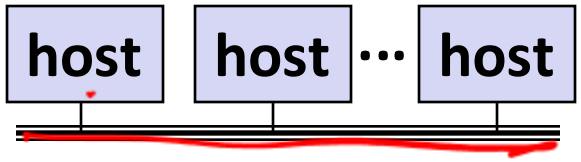
Next Level: Bridged Ethernet Segment



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

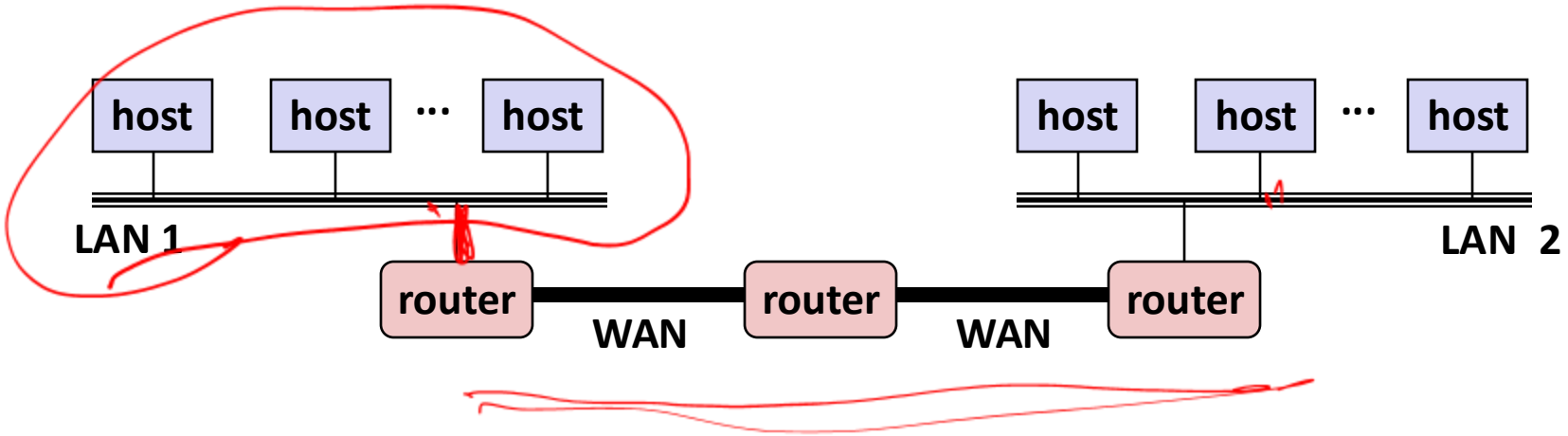
Conceptual View of LANs

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



Next Level: internets

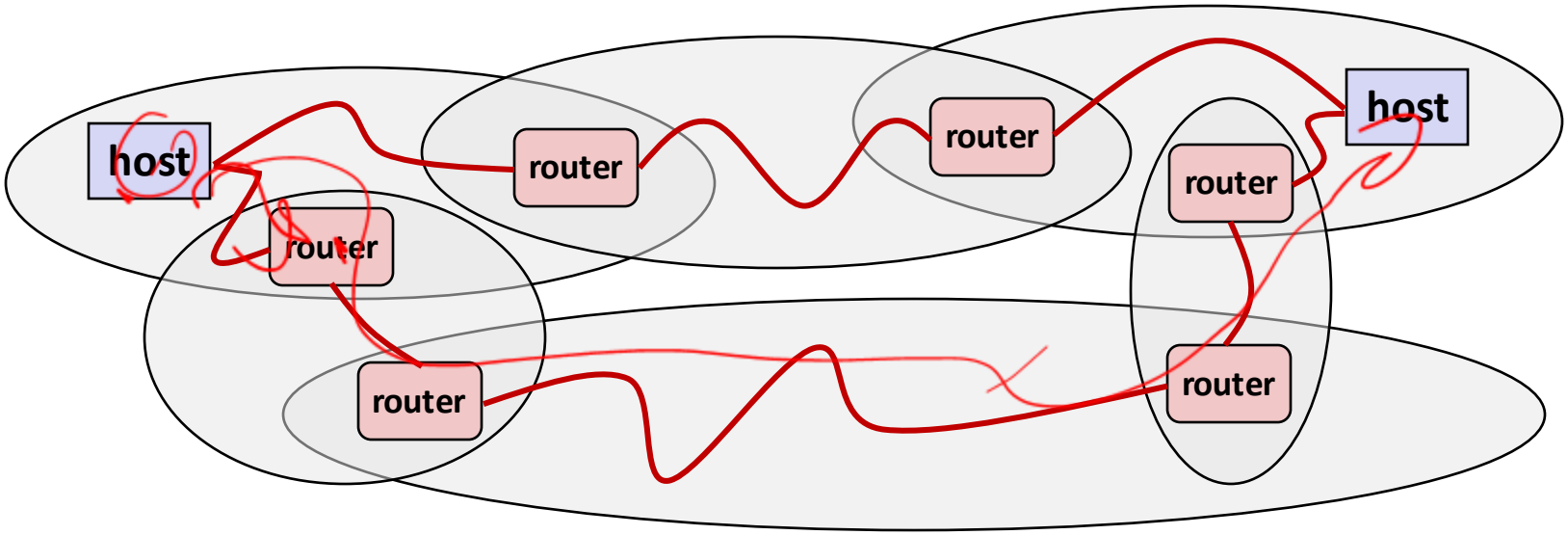
- Multiple incompatible LANs can be physically connected by specialized computers called *routers*
- The connected networks are called an *internet* (lower case)



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

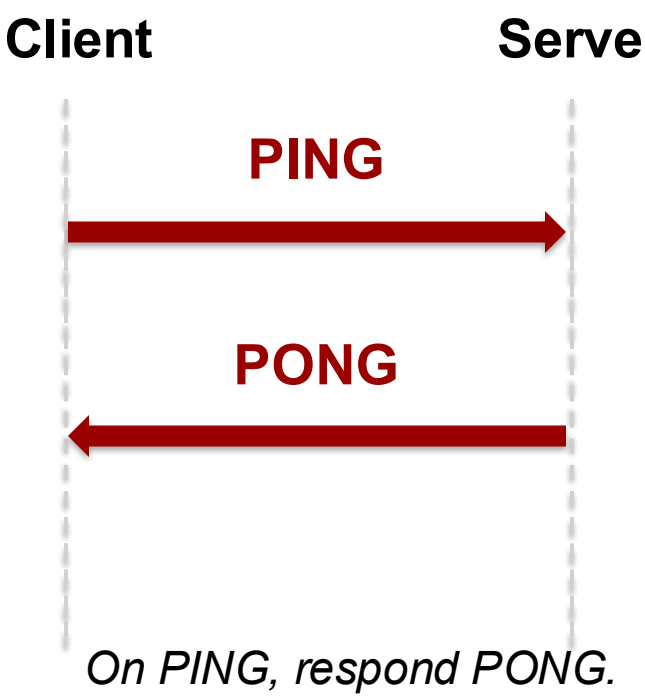
The Notion of an internet Protocol

- **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 Is a Protocol?

An agreement about how independent parties communicate.



- **Message format**
  Four ASCII bytes? A newline terminator?
- **Meaning**
  What exactly does PONG acknowledge?
- **Ordering**
  Can another PING be sent before PONG?
- **Failures**
  What if PONG never arrives?

Delivering bytes is not the same as communicating.

What Makes Something a Protocol?

- **Syntax**
  What messages look like.
- **Semantics**
  What the messages mean.
- **Sequencing**
  When each party sends and responds.
- **Error handling**
  What to do when something goes wrong.

What does IP deliberately leave out?

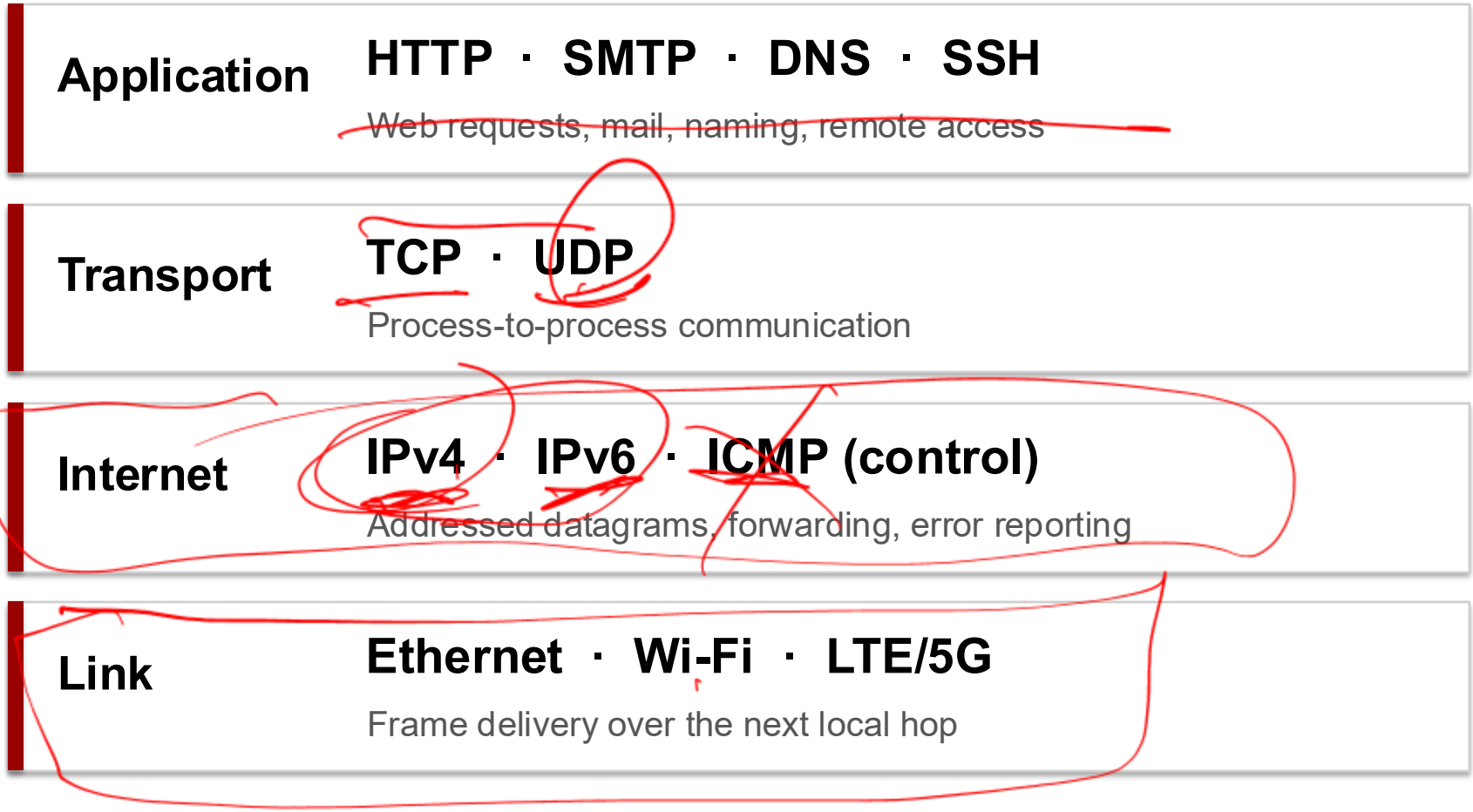
IP is BEST EFFORT

It does NOT provide:

- Guaranteed delivery
- In-order delivery
- Retransmission
- Connection setup

TCP adds reliability above IP when needed.

Protocols at Different Layers



A common IP layer can connect many applications and many networks.

Why Open, Minimal Protocols Matter

- **Open specifications**
  Anyone can read and implement the standards.
- **Interoperability**
  Independent vendors and systems work together.
- **Abstraction**
  Applications need not know the underlying links.
- **Independent evolution**
  Networks and software can change separately.

The power of doing less

IP only promises to try to forward each packet.

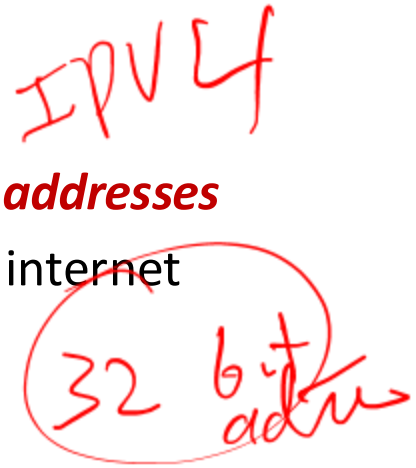
That simple contract works over almost any network.

TCP can add reliability above IP.

A small, open contract is easier to implement everywhere.

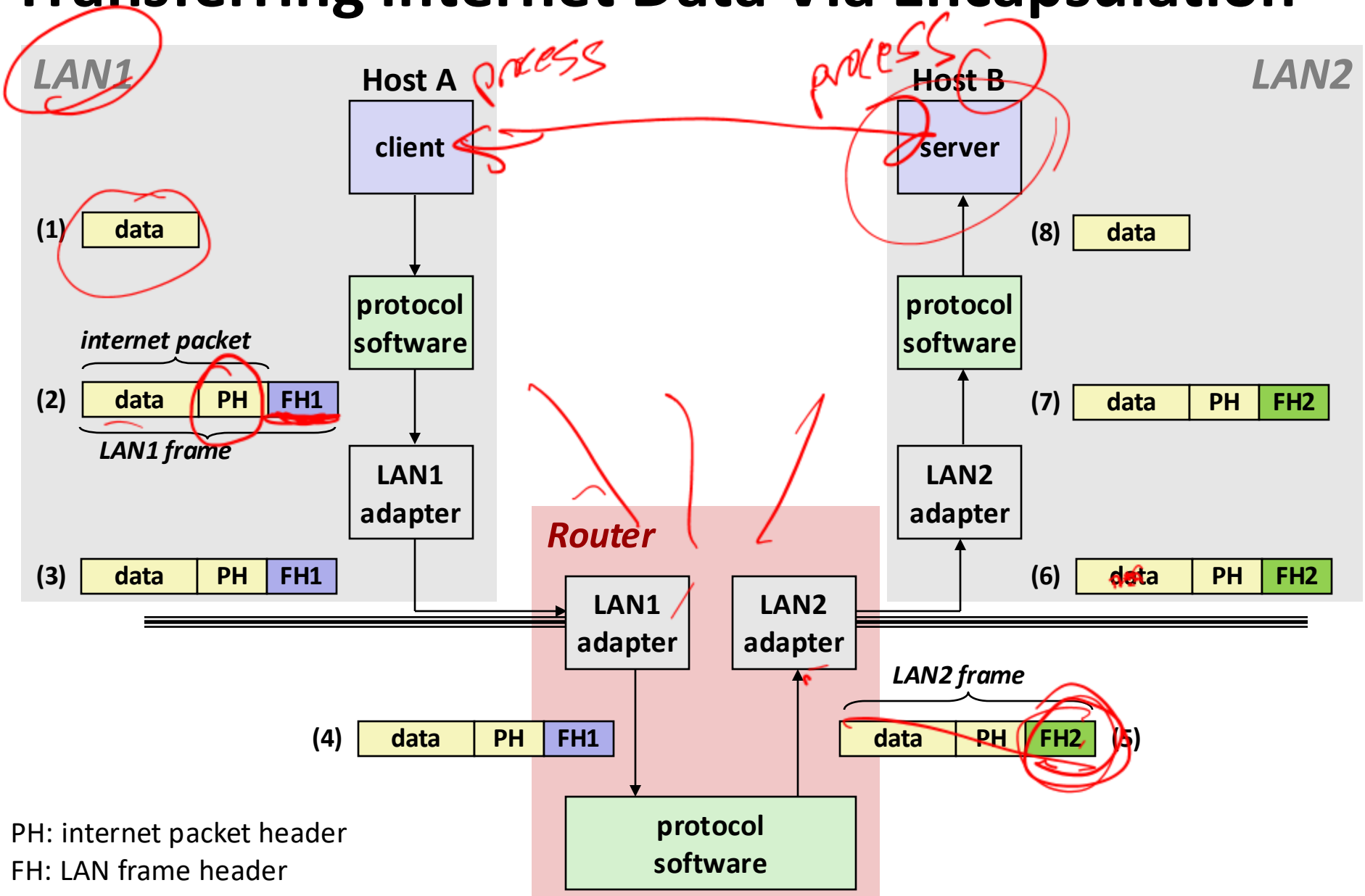
What Does an internet Protocol Do?

- **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

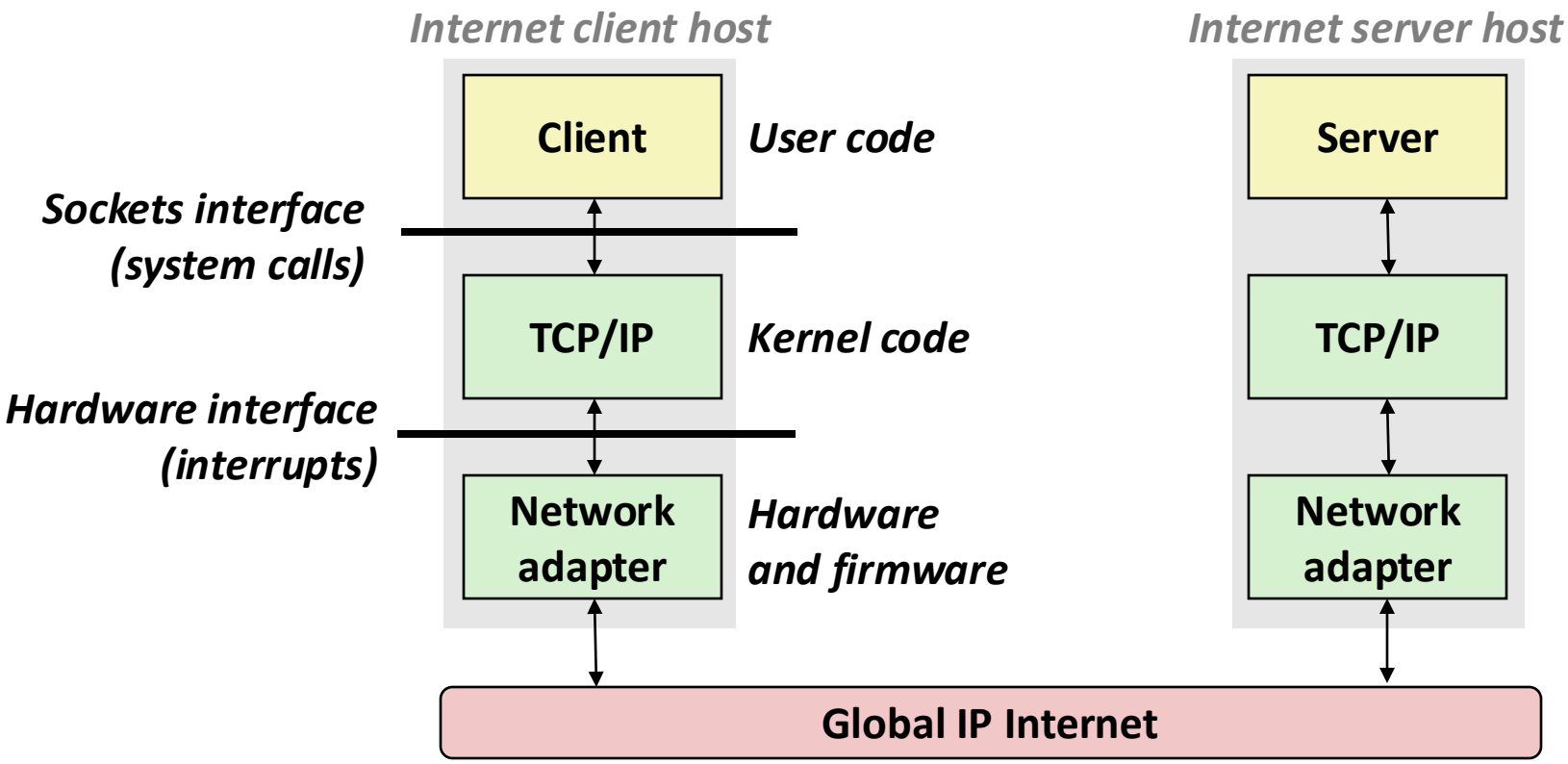
- **We are glossing over a number of important questions:**
  - What if different networks have different maximum frame sizes? (segmentation)
  - How do routers know where to forward frames?
  - How are routers informed when the network topology changes?
  - What if packets get lost?

- **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 (User 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



A Programmer's View of the Internet

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

- 128.2.203.179
- 127.0.0.1 (always *localhost*)

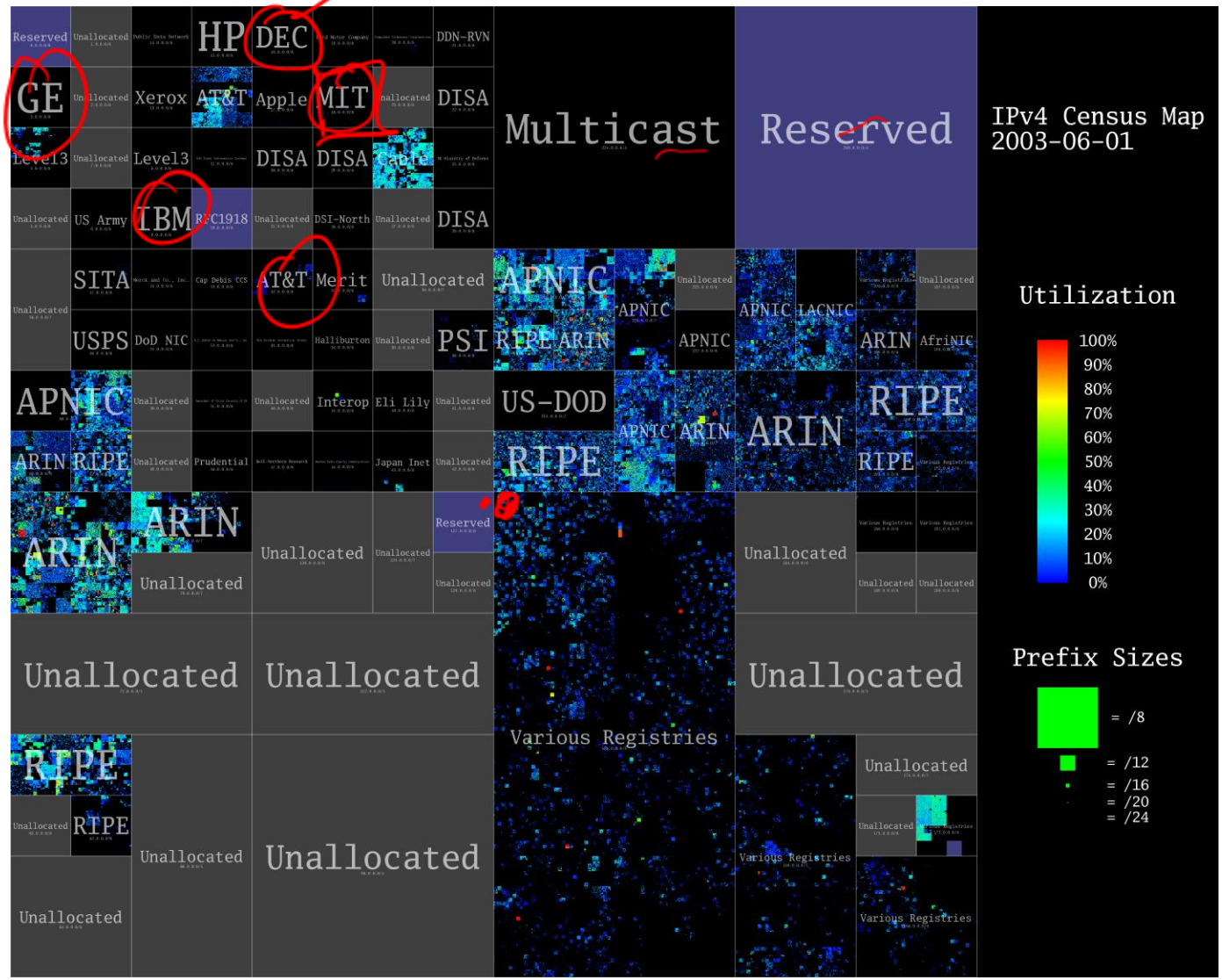
2. As a convenience for humans, the Domain Name System maps a set of identifiers called Internet *domain names* to IP addresses:

- www.cs.cmu.edu "resolves to" 128.2.42.95

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

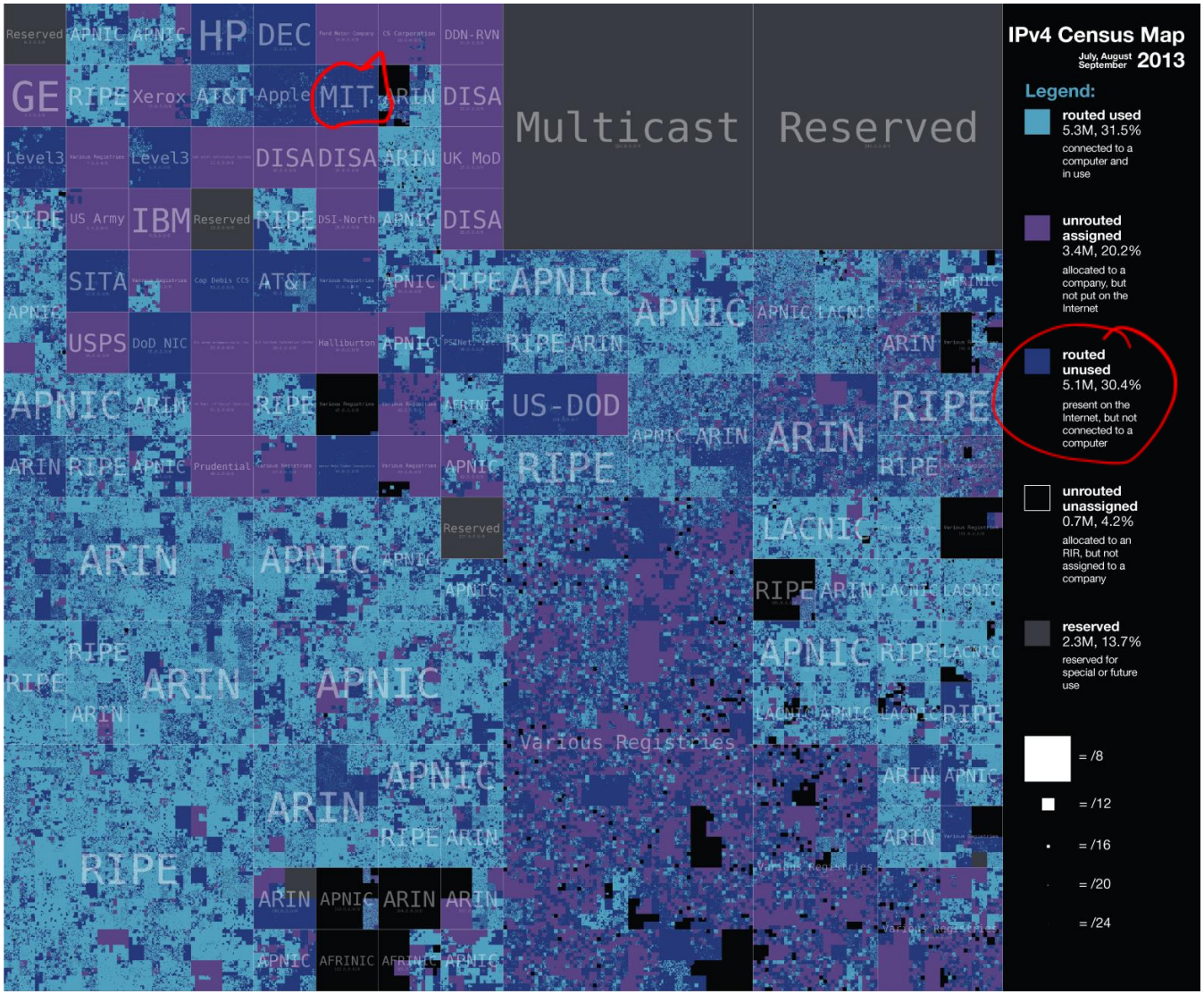
***: IPv4**

IPv4 map circa 2003



Caida.org

IPv4 map circa 2013



Caida.org

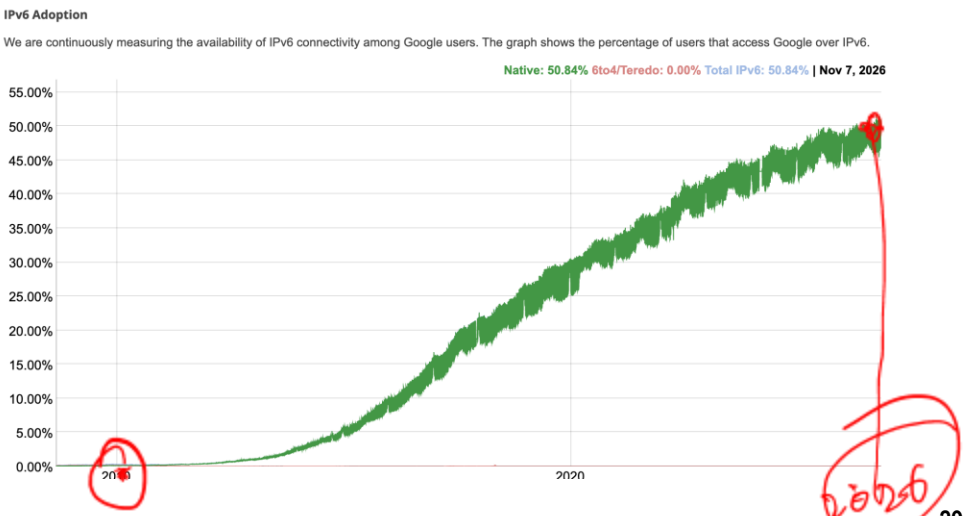
Aside: IPv4 and IPv6

- ***IPv4*** **(Internet Protocol version 4) specified 1981**
  - 32-bit host addresses (192.0.2.43)
  - Known to not be enough for everyone since ~1990
- **IPv6 (Internet Protocol version 6) specified 1996**
  - 128-bit addresses (2001:0db8:0:0:0:0:cafe:la7e)
  - Intended to replace IPv4
  - Very slow adoption due to need to replace routers (CMU's network doesn't support IPv6 at all!)

- **Application programmers mostly don't have to care**
  - Sockets API makes it easy to write code that seamlessly uses either, as necessary

IPv6 traffic to Google

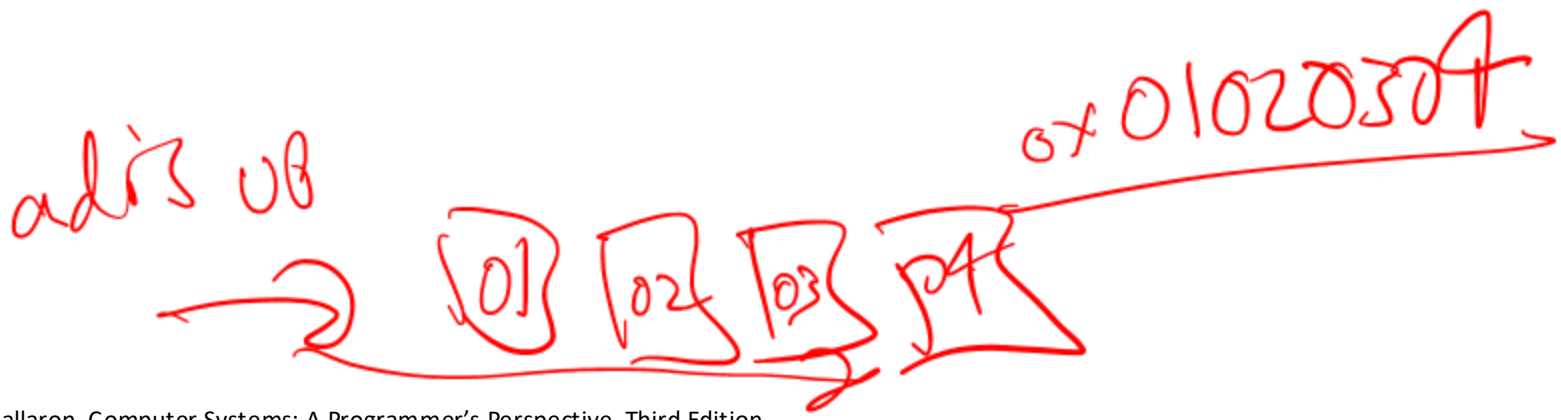
https://www.google.com/intl/en/ipv6/statistics.html



(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.

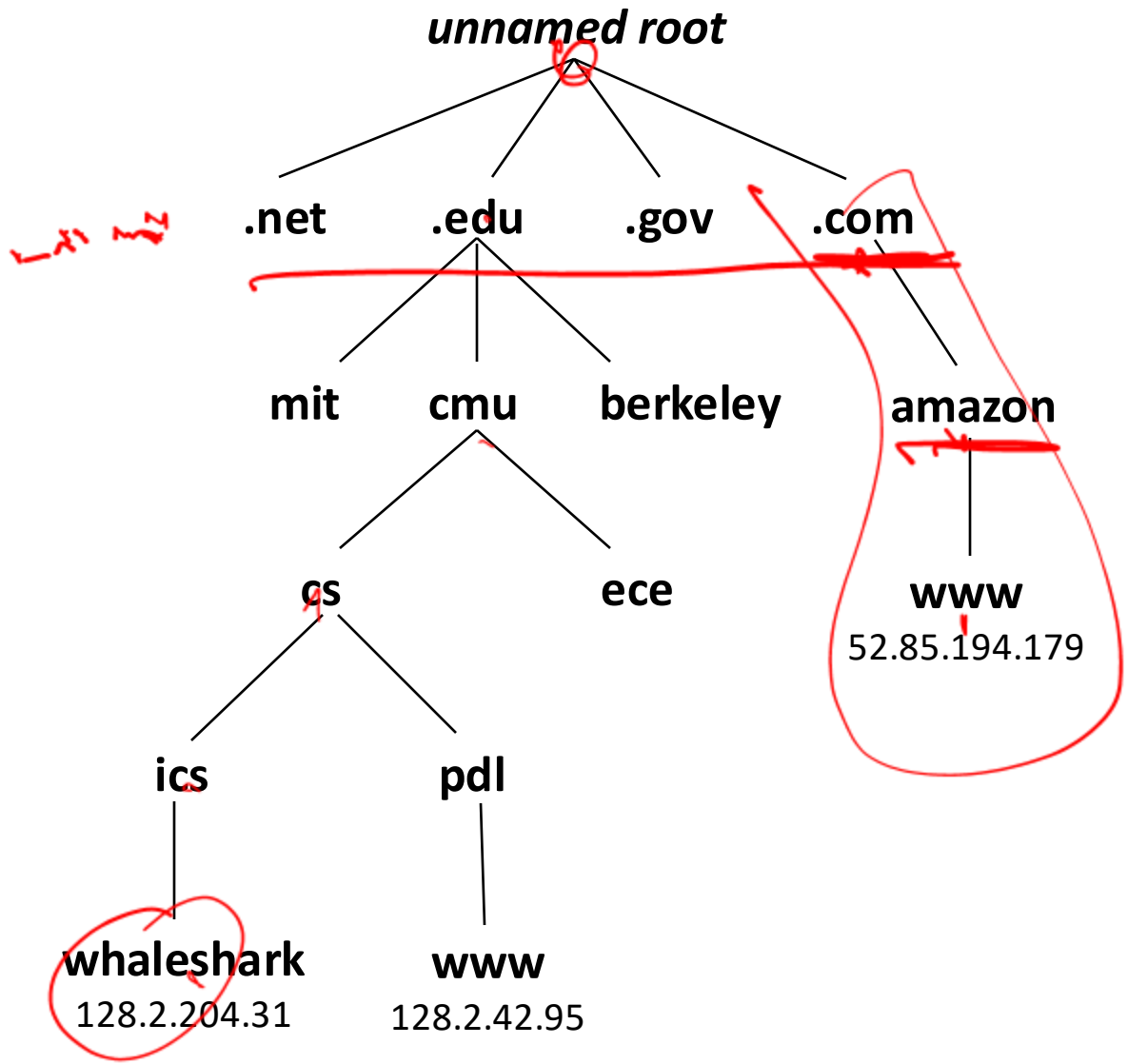
```
/* Internet address structure */
struct in_addr {
    uint32_t s_addr; /* network byte order (big-endian) */
};
```



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



First-level domain names

Second-level domain names

Third-level domain names

Domain Naming System (DNS)

- **The Internet maintains a mapping between IP addresses and domain names in a 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: 128.2.204.31
```

- Multiple domain names mapped to the same IP address:

```
linux> nslookup cs.cmu.edu
Address: 128.2.42.95
linux> nslookup scs.cmu.edu
Address: 128.2.42.95
```

- And backwards:

```
linux> nslookup 128.2.42.95
128.2.42.95.in-addr.arpa    name = SCS-WEB-LB.ANDREW.CMU.EDU.
```

Properties of DNS Mappings (cont)

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

```
linux> nslookup cnn.com
Address: 151.101.67.5
Address: 151.101.131.5
Address: 151.101.195.5
Address: 151.101.3.5

linux> nslookup cnn.com
Address: 151.101.3.5
Address: 151.101.67.5
Address: 151.101.131.5
Address: 151.101.195.5
```

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

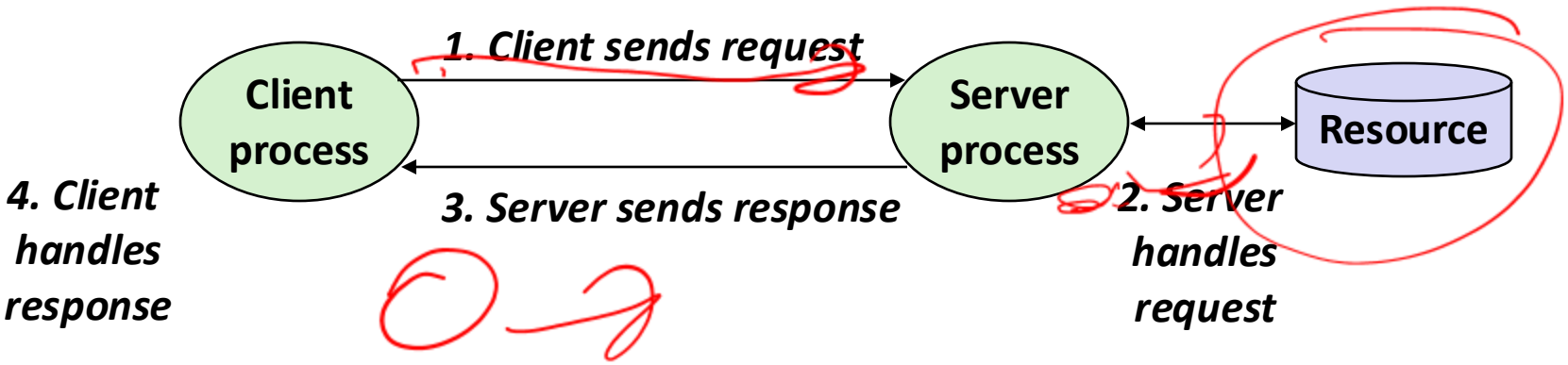
```
linux> nslookup ics.cs.cmu.edu
(No Address given)
```

Outline

- Network types and structures
- Locating a host
- **Setting up a connection**
- HTTP Example

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)



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

(3) Internet Connections

- **Clients and servers most often communicate by sending streams of bytes over TCP *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)**

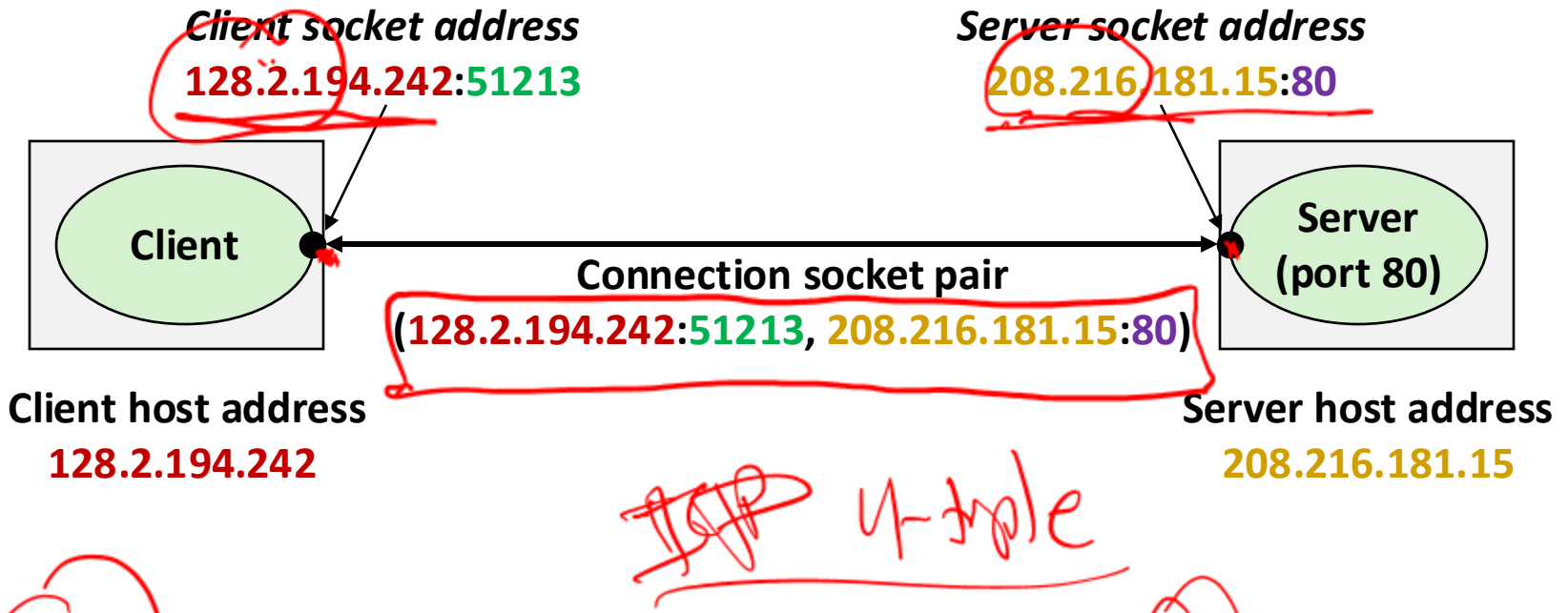
Well-known Service Names and Ports

- **Popular services have permanently assigned *well-known ports* *and* corresponding *well-known service names*:**
  - echo servers: echo 7
  - ftp servers: ftp 21
  - ssh servers: ssh 22
  - email servers: smtp 25
  - Unencrypted Web servers: http 80
  - SSL/TLS encrypted Web: https 443

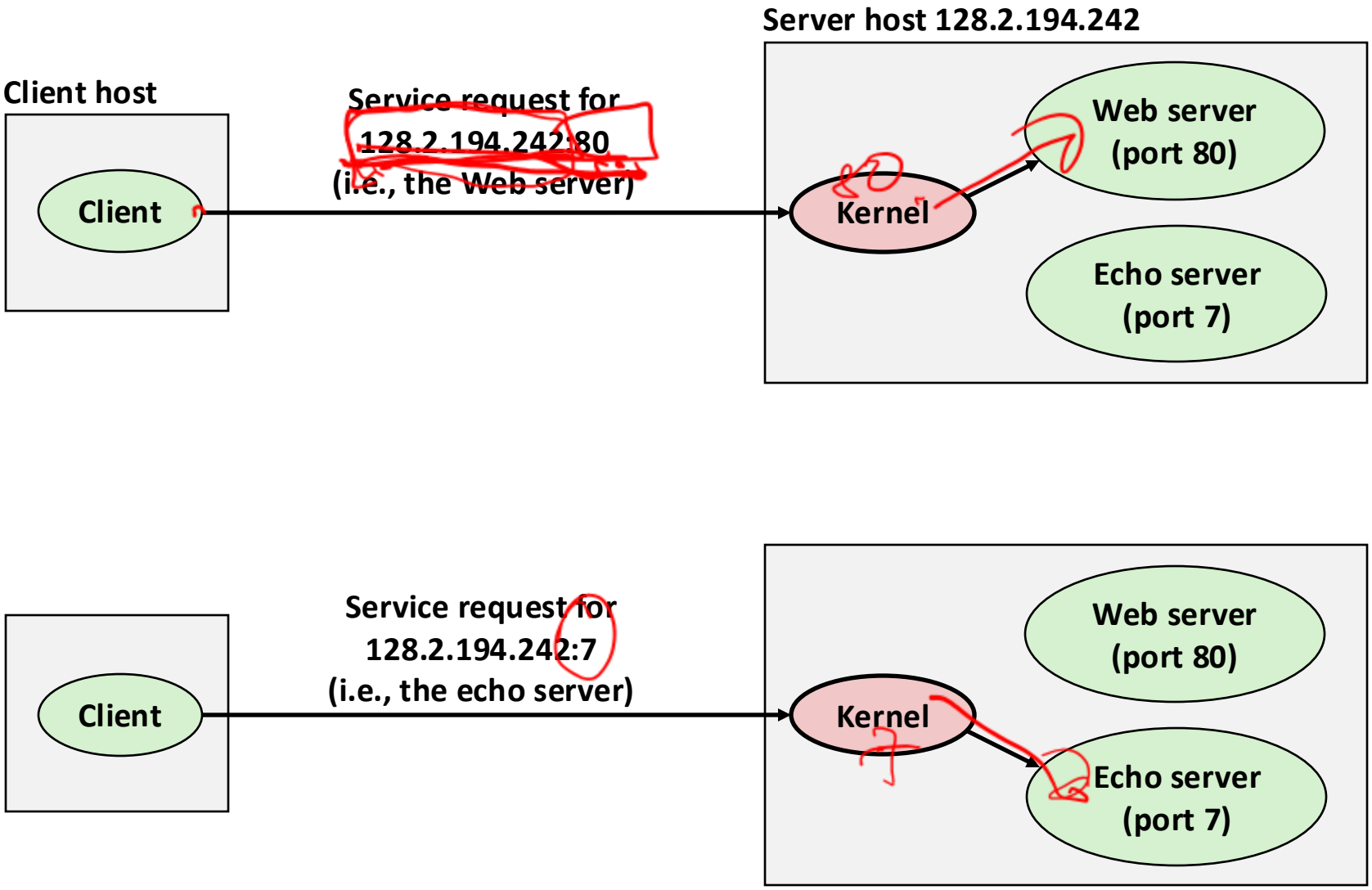
- **Mappings between well-known ports and service names is contained in the file `/etc/services` on each Linux machine.**

Anatomy of a Connection

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



Using Ports to Identify Services



Quiz

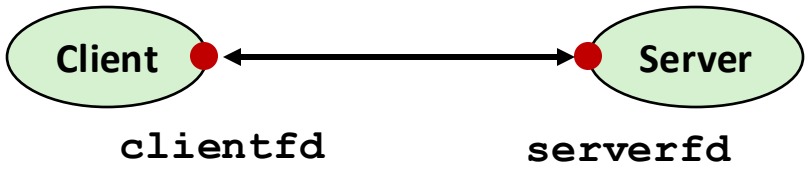
canvas.cmu.edu/courses/55964/quizzes/169264

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
  - Using the FD abstraction lets you reuse code & interfaces
- **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**

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

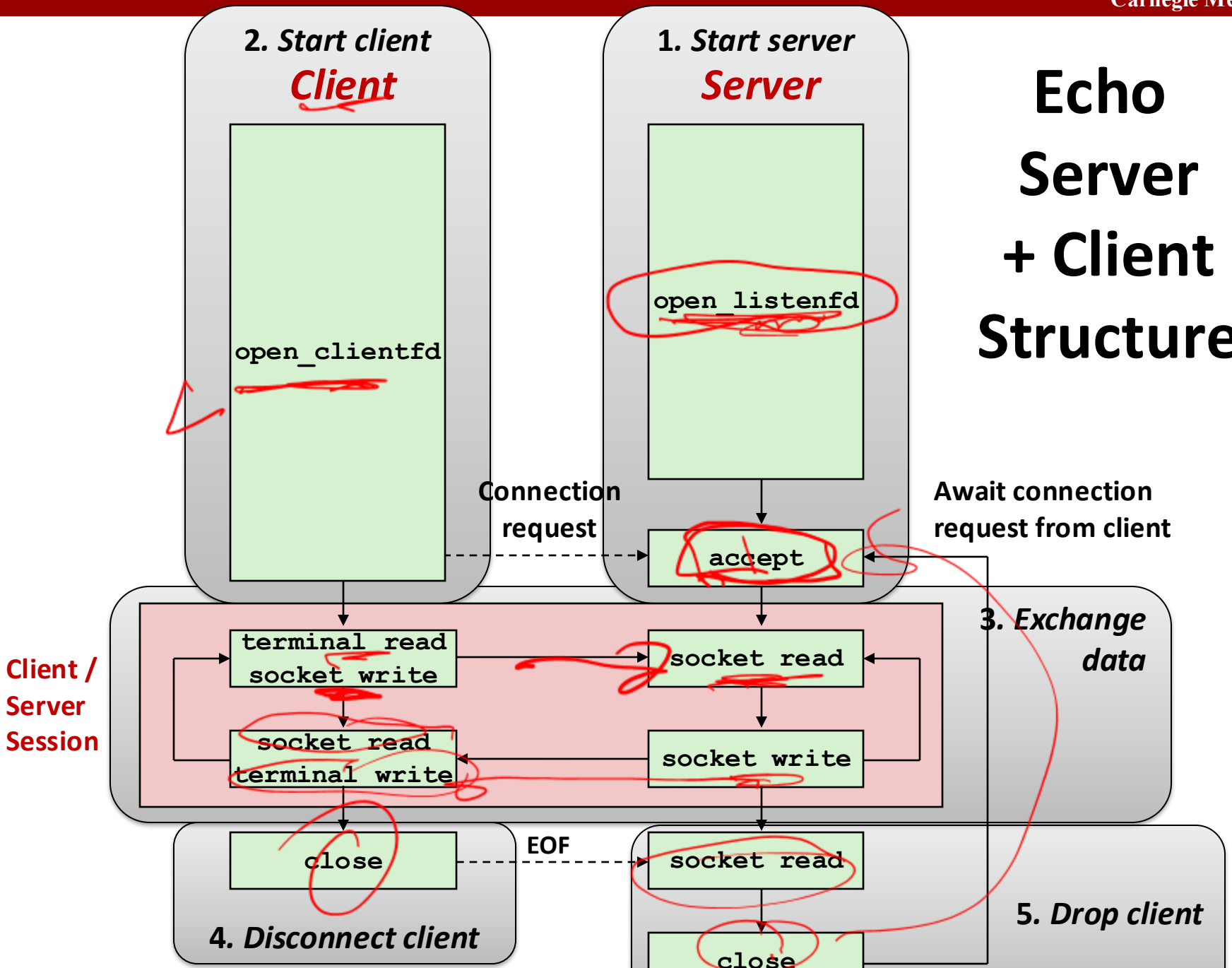
Client

```
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
```

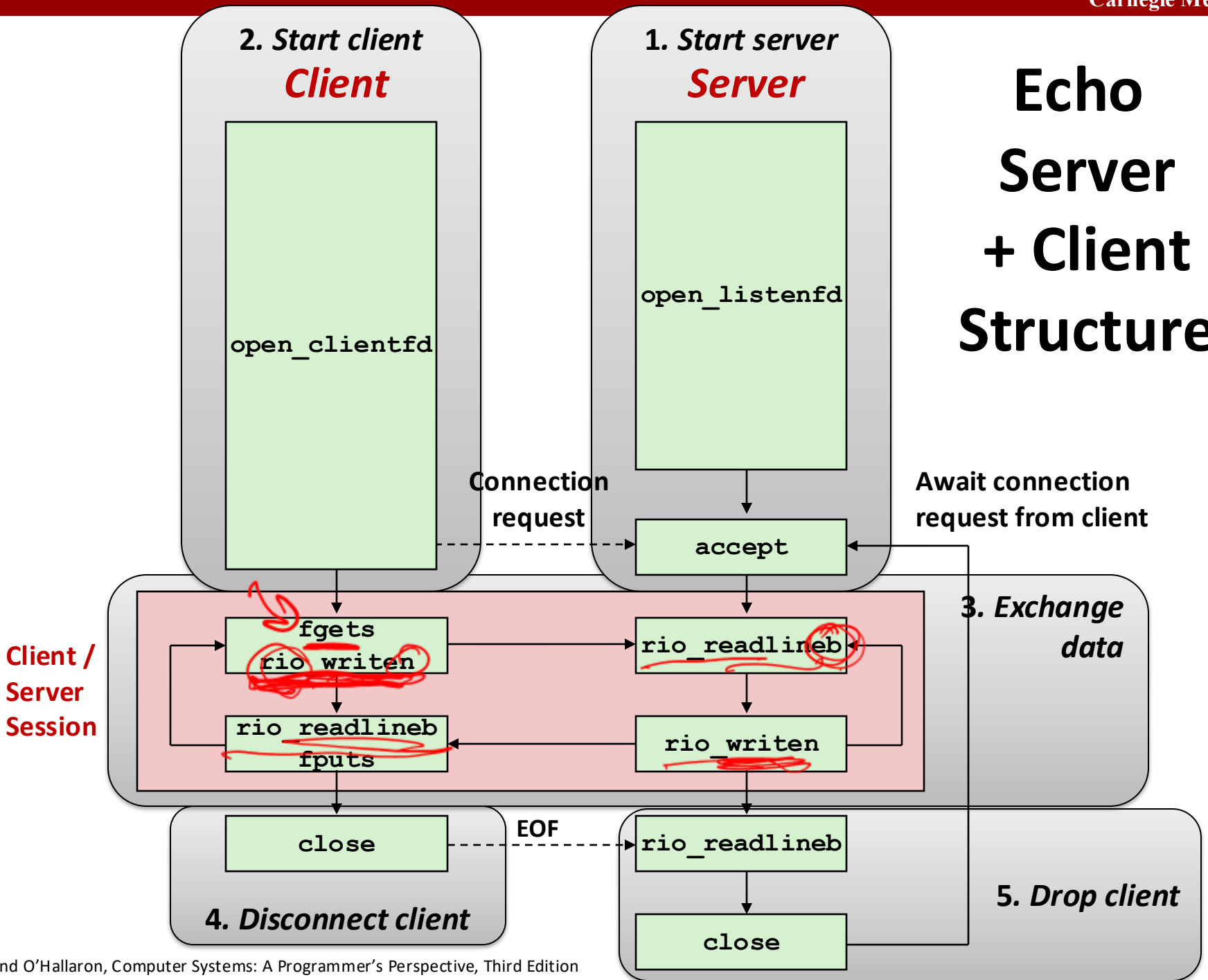
Server

```
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



Echo Server + Client Structure



Recall: Unbuffered RIO Input/Output

- Same interface as Unix `read` and `write`
- Especially useful for transferring data on network sockets

```
#include "csapp.h"

ssize_t rio_readn(int fd, void *usrbuf, size_t n);
ssize_t rio_writen(int fd, void *usrbuf, size_t n);
```

Return: num. bytes transferred if OK, 0 on EOF (`rio_readn` only), -1 on error

- **`rio_readn`** returns short count only if it encounters EOF
  - Only use it when you know how many bytes to read
- **`rio_writen`** never returns a short count
- Calls to **`rio_readn`** and **`rio_writen`** can be interleaved arbitrarily on the same descriptor

Recall: Buffered RIO Input Functions

- **Efficiently read text lines and binary data from a file partially cached in an internal memory buffer**

```
#include "csapp.h"

void rio_readinitb(rio_t *rp, int fd);

ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n);
```

Return: num. bytes read if OK, 0 on EOF, -1 on error

- `rio_readlineb` reads a ***text line*** of up to **`maxlen`** bytes from file **`fd`** and stores the line in **`usrbuf`**
  - Especially useful for reading text lines from network sockets
- Stopping conditions
  - **`maxlen`** bytes read
  - EOF encountered
  - Newline (‘**`\n`**’) encountered

Echo Client: Main Routine

```
#include "csapp.h"

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

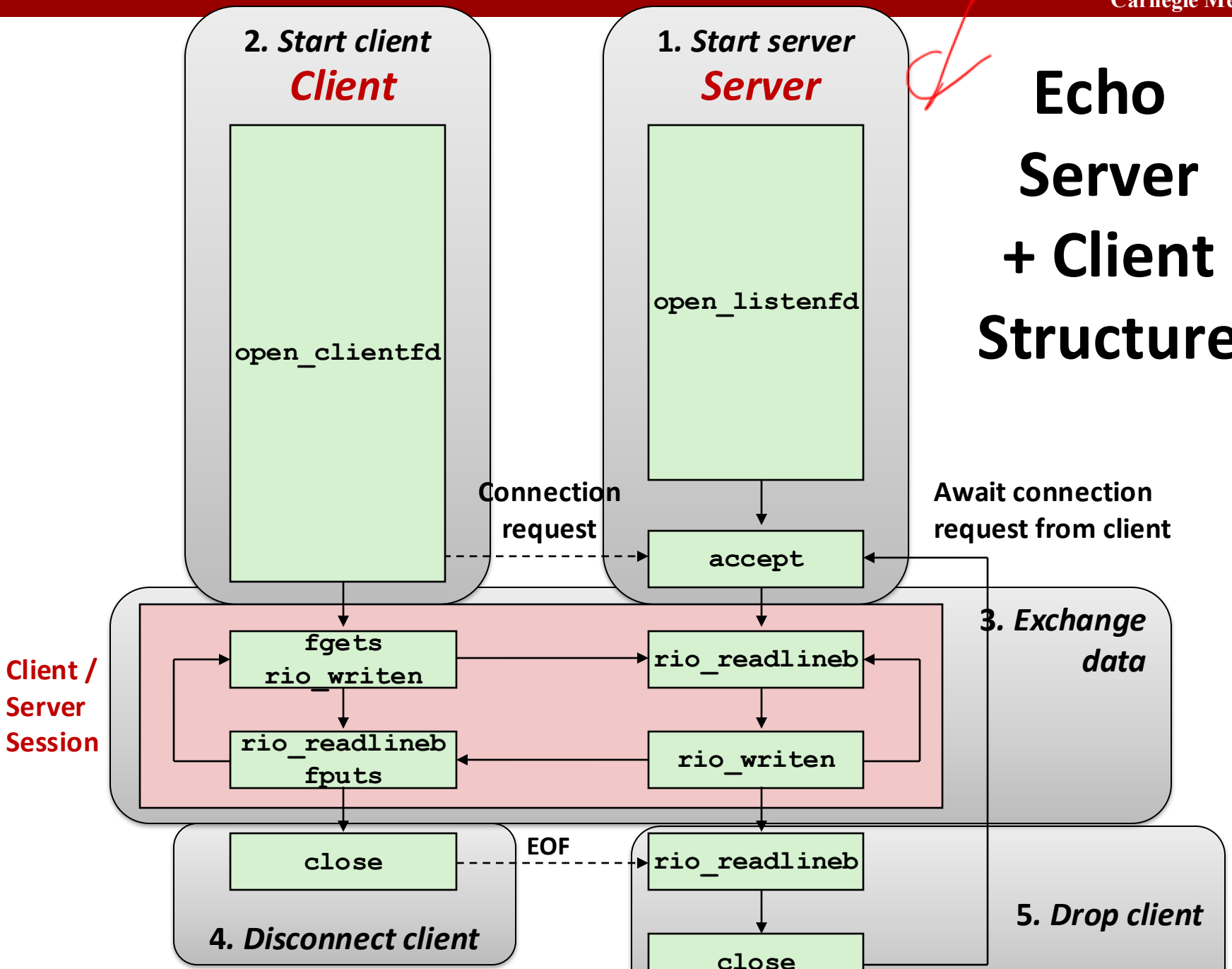
    host = argv[1];
    port = argv[2];

    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

Echo Server + Client Structure



Iterative Echo Server: Main Routine

```
#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];









}
```

echoserveri.c

Iterative Echo Server: Main Routine

```
#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);









}
```

echoserveri.c

Iterative Echo Server: Main Routine

```
#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);


}
```

echoserveri.c

Iterative Echo Server: Main Routine

```
#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

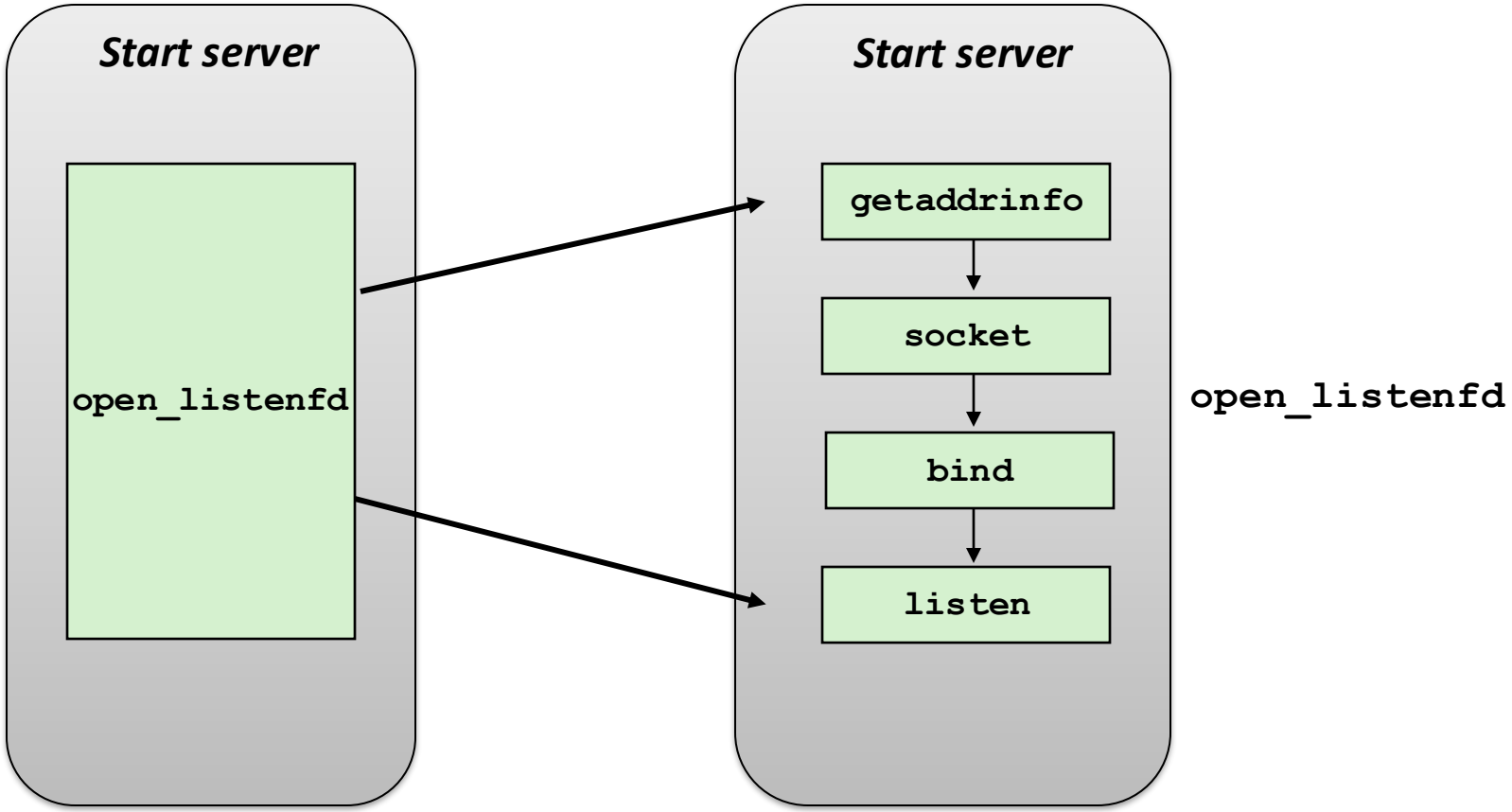
- **The server uses RIO to read and echo text lines until EOF (end-of-file) condition is encountered.**
  - EOF condition caused by client calling **`close(clientfd)`**

```
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

Digging Deeper

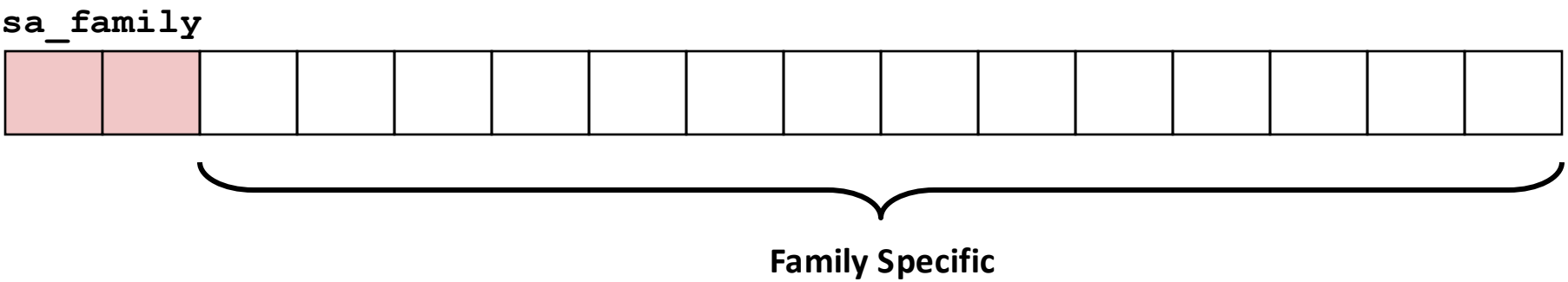


Socket Address Structures

- **Generic socket address:**
  - For address arguments to **`connect`**, **`bind`**, and **`accept`** *(next lecture)*
  - For casting convenience, we adopt the Stevens convention:

    `typedef struct sockaddr SA;`

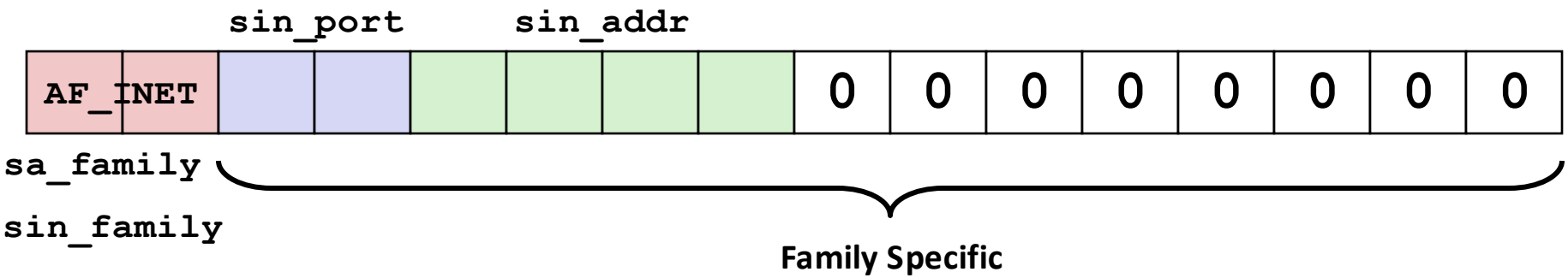
```
struct sockaddr {
  uint16_t  sa_family;    /* Protocol family */
  char      sa_data[14];  /* Address data  */
};
```



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) */
};
```



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`

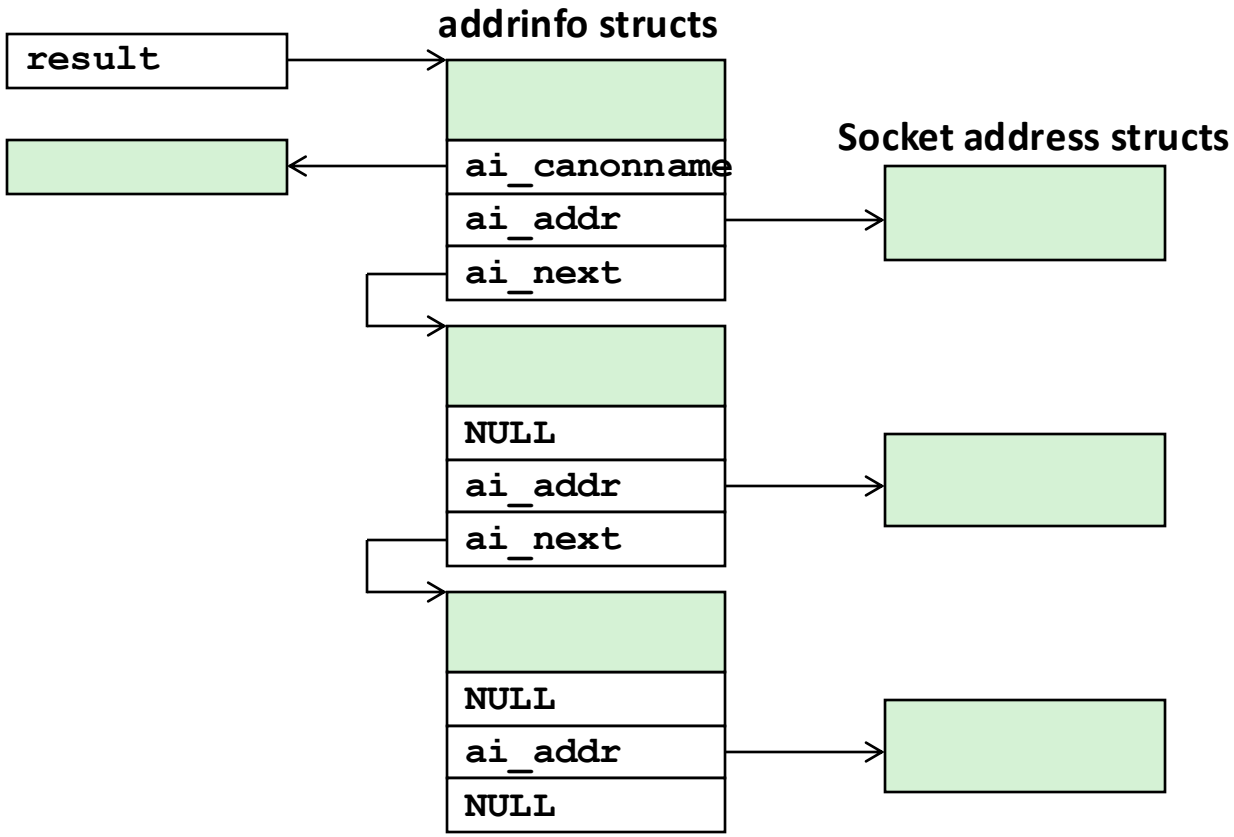
```
int getaddrinfo(const char *host,              /* Hostname or address */
                const char *service,           /* Port or service name */
                const struct addrinfo *hints,/* Input parameters */
                struct addrinfo **result);    /* Output linked list */

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 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 .

(`socket`, `connect`, `bind` to be discussed next lecture)

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 (localhost)

whaleshark> ./hostinfo whaleshark.ics.cs.cmu.edu
128.2.204.31 (whaleshark.ics.cs.cmu.edu)

whaleshark> ./hostinfo cnn.com
151.101.131.5 (151.101.131.5)
151.101.3.5 (151.101.3.5)
151.101.195.5 (151.101.195.5)
151.101.67.5 (151.101.67.5)
2a04:4e42::773 (2a04:4e42::773)
2a04:4e42:200::773 (2a04:4e42:200::773)
2a04:4e42:600::773 (2a04:4e42:600::773)
2a04:4e42:400::773 (2a04:4e42:400::773)
```

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);
```

Protocol specific!

Indicates that we are using 32-bit IPV4 addresses

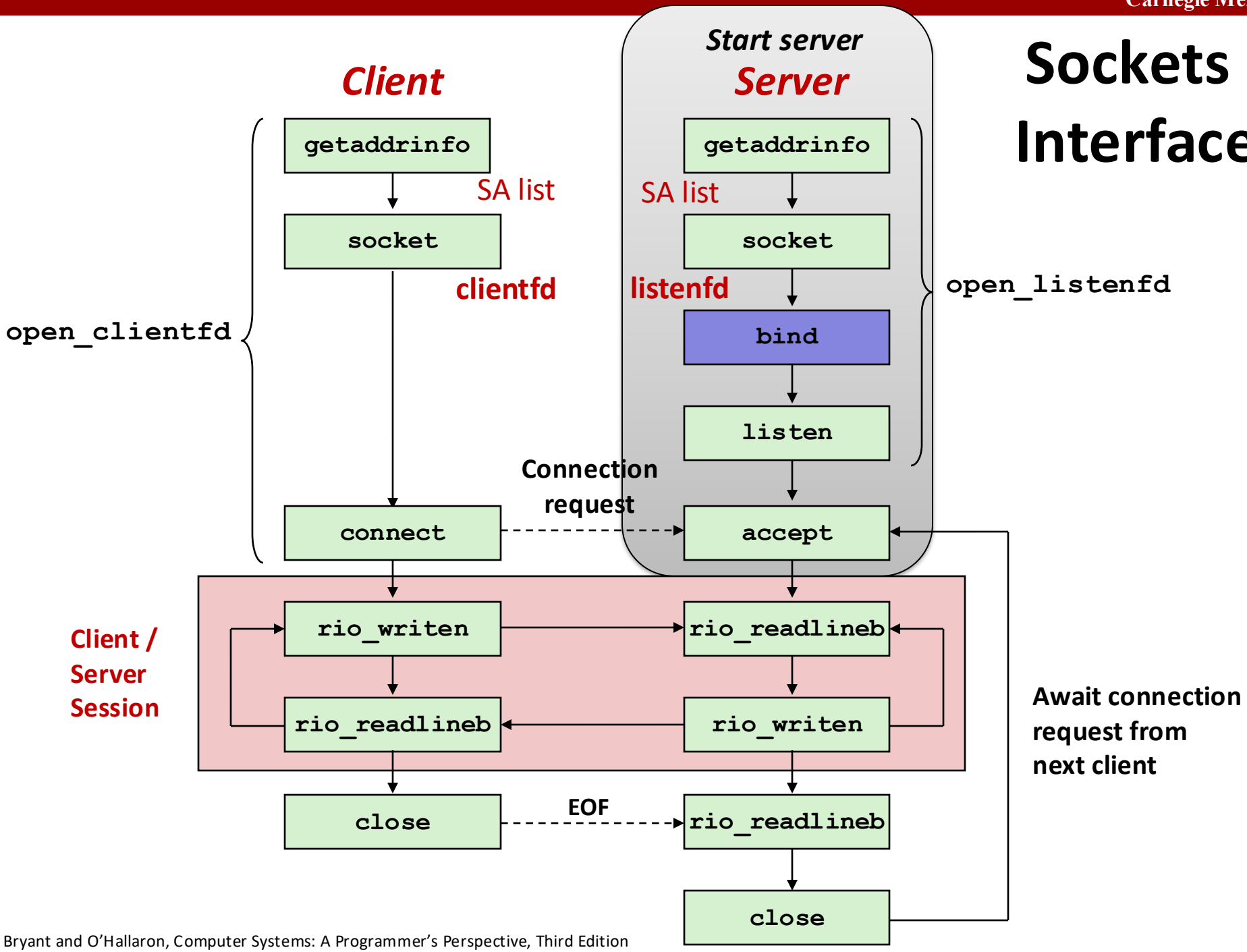
Indicates that the socket will be the end point of a reliable (TCP) connection

- Example:

```
int clientfd = socket(ai->ai_family, ai->ai_socktype,
                      ai->ai_protocol);
```

Use getaddrinfo and you don't have to know or care which protocol!

Sockets Interface



Sockets Interface: `bind`

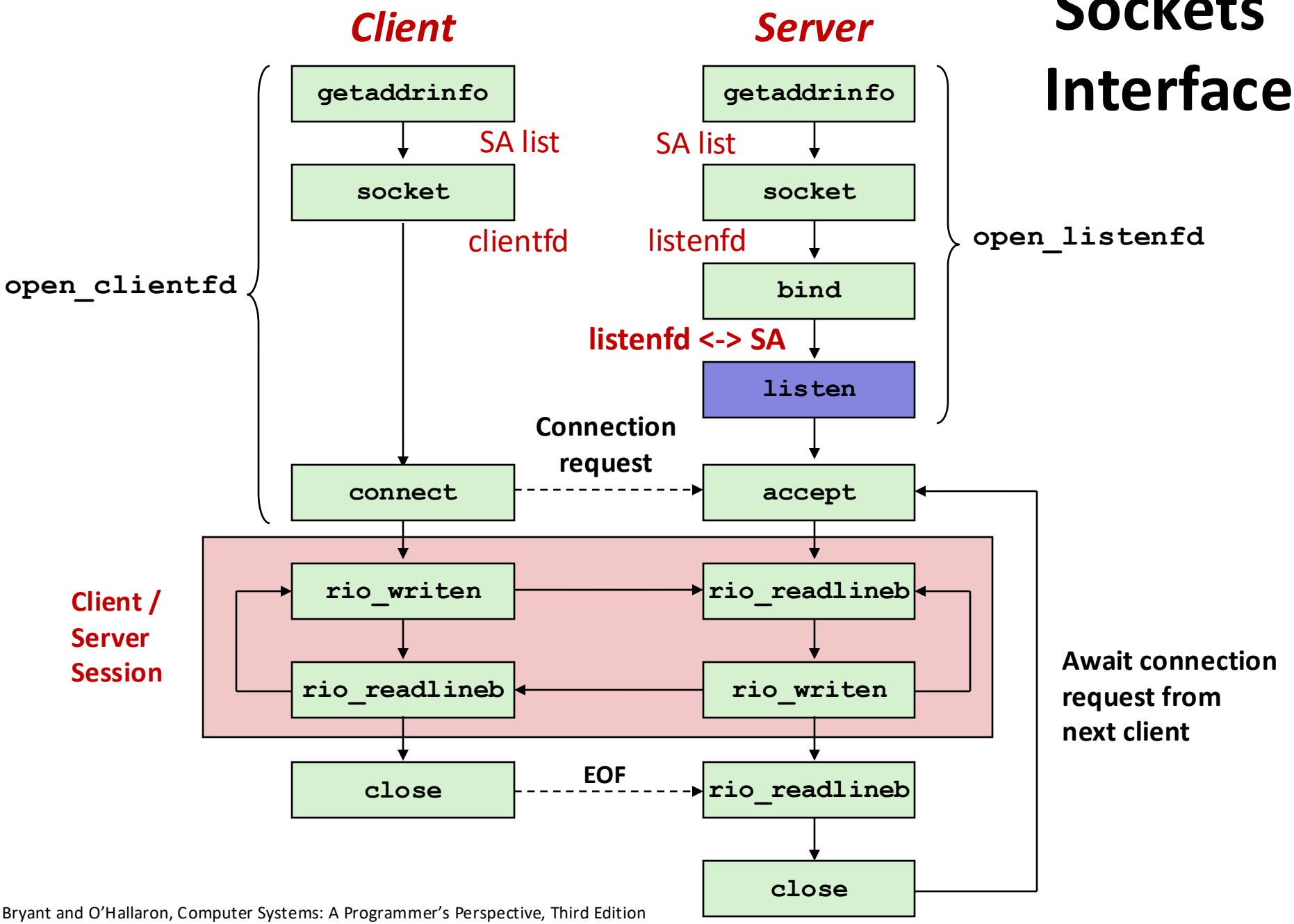
- A server uses `bind` to ask the kernel to associate the server's socket address with a socket descriptor:

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

Our convention: `typedef struct sockaddr SA;`

- 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`.

Sockets Interface



Sockets Interface: `listen`

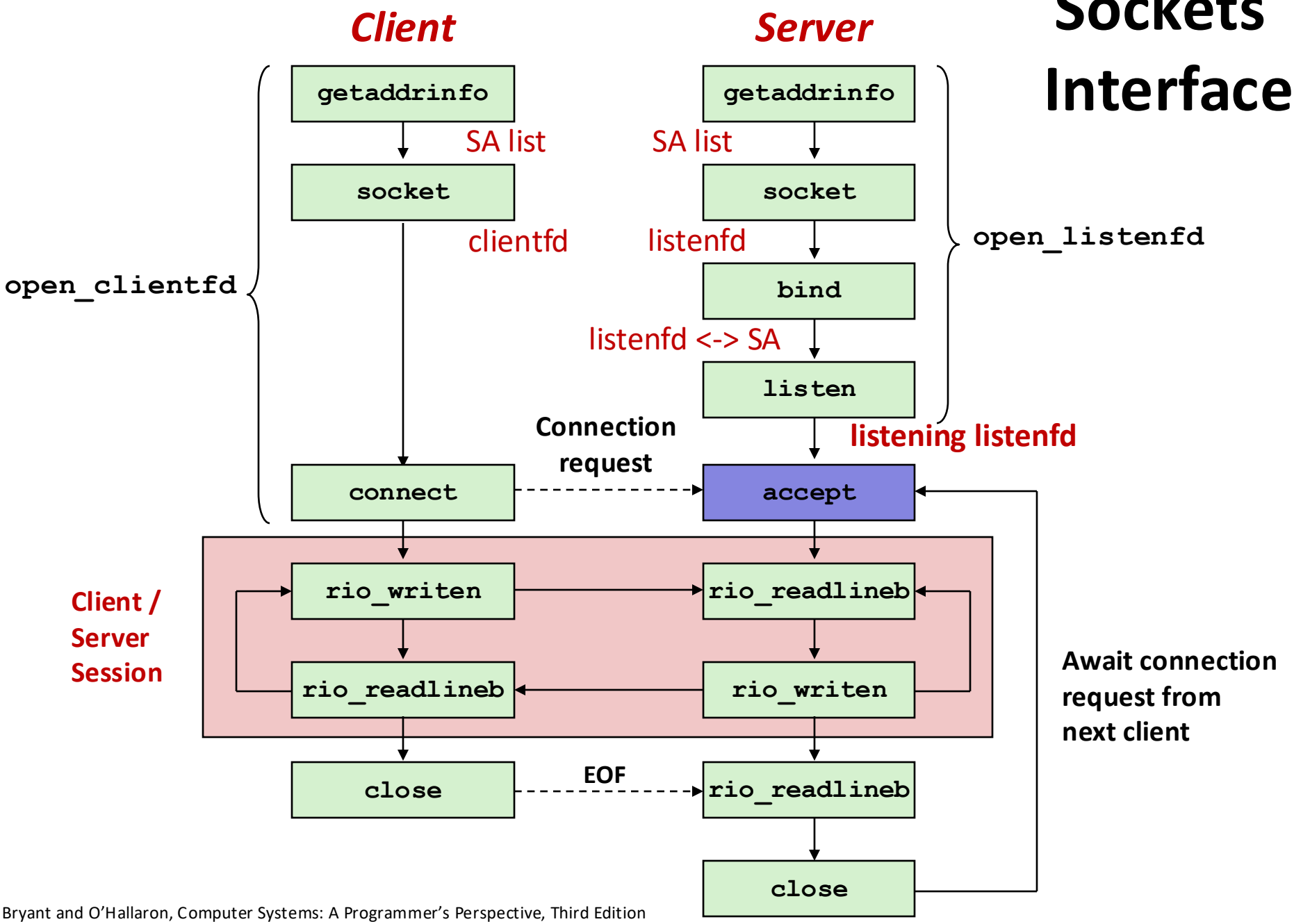
- Kernel assumes that descriptor from socket function is an *active socket* that will be on the client end
- A server calls the listen function to tell the kernel that a descriptor will be used by a server rather than a client:

```
int listen(int sockfd, int backlog);
```

- Converts `sockfd` from an active socket to a *listening socket* that can accept connection requests from clients.

- `backlog` is a hint about the number of outstanding connection requests that the kernel should queue up before starting to refuse requests (128-ish by default)

Sockets Interface



Sockets Interface: `accept`

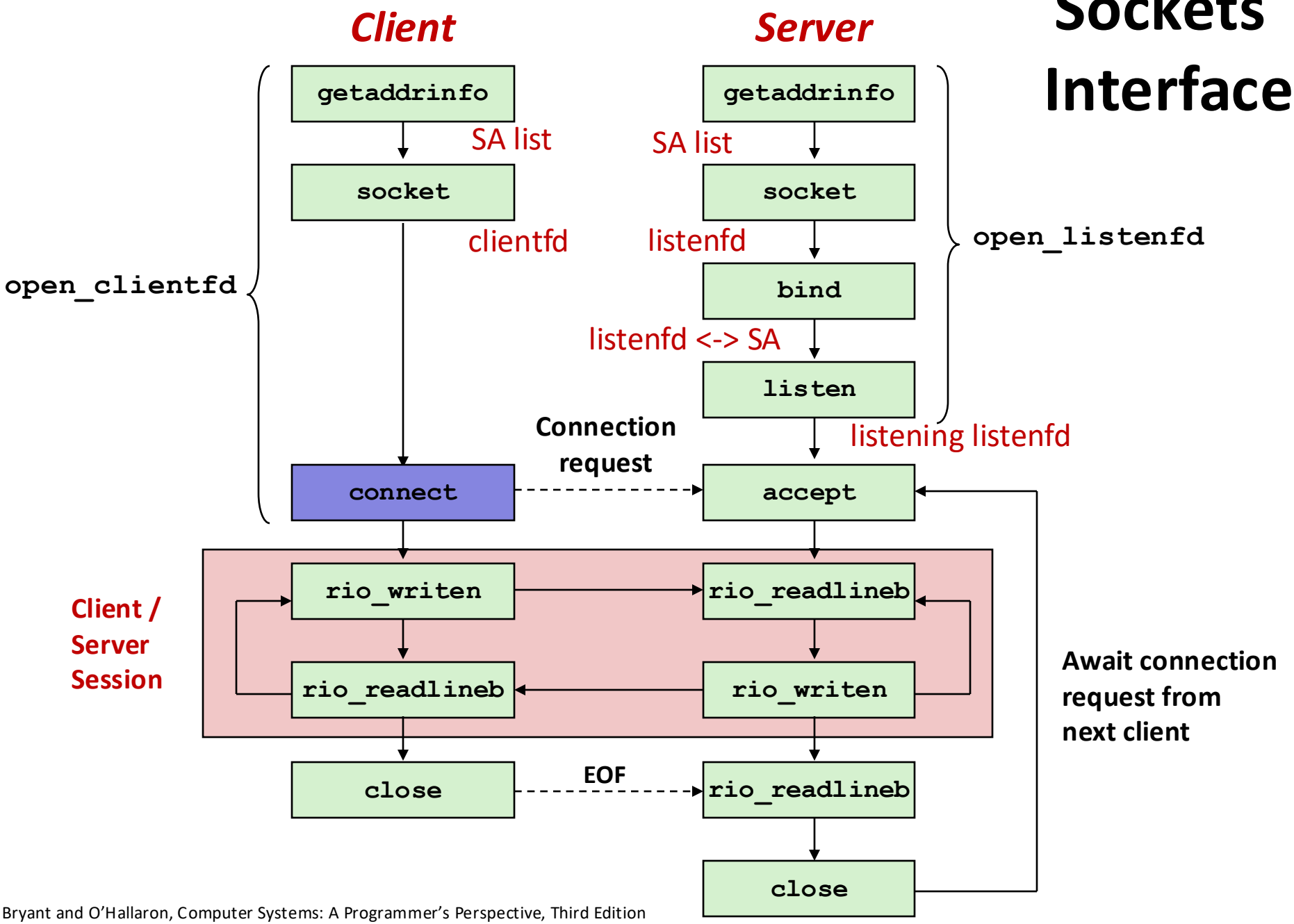
- Servers wait for connection requests from clients by calling `accept`:

```
int accept(int listenfd, SA *addr, int *addrlen);
```

- Waits for connection request to arrive on the connection bound to `listenfd`, then fills in client's socket address in `addr` and size of the socket address in `addrlen`.

- Returns a *connected descriptor* `connfd` that can be used to communicate with the client via Unix I/O routines.

Sockets Interface



Sockets Interface: `connect`

- **A client establishes a connection with a server by calling `connect`:**

```
int connect(int clientfd, SA *addr, socklen_t addrlen);
```

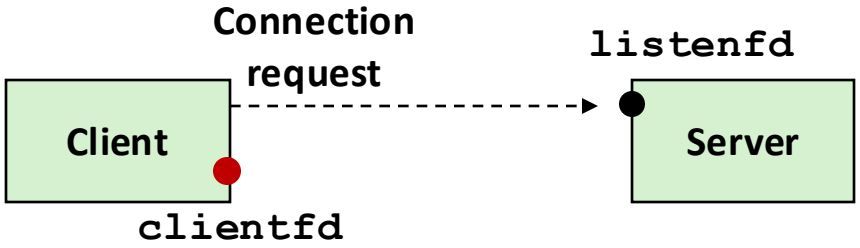
- **Attempts to establish a connection with server at socket address `addr`**
  - If successful, then **`clientfd`** is now ready for reading and writing.
  - Resulting connection is characterized by socket pair

    (**`x:y, addr.sin_addr:addr.sin_port`**)
    - **`x`** is client address
    - **`y`** is ephemeral port that uniquely identifies client process on client host
- **Best practice is to use `getaddrinfo` to supply the arguments `addr` and `addrlen`.**

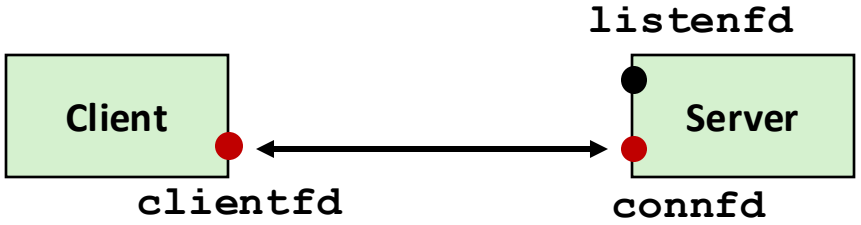
connect/accept Illustrated



1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`



2. Client makes connection request by calling and blocking in `connect`



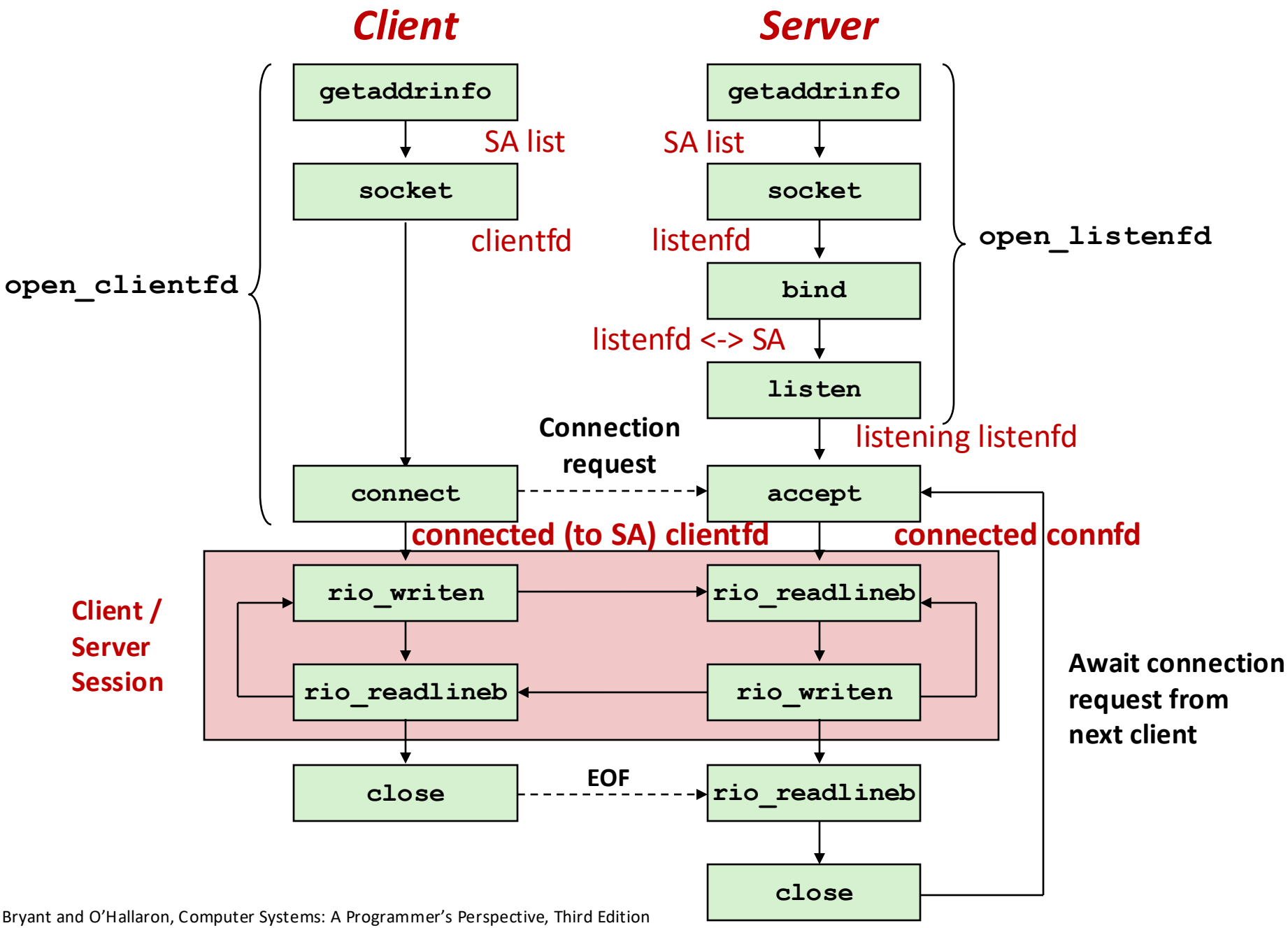
3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`

Connected vs. Listening Descriptors

- **Listening descriptor**
  - End point for client connection requests
  - Created once and exists for lifetime of the server

- **Connected descriptor**
  - End point of the connection between client and server
  - A new descriptor is created each time the server accepts a connection request from a client
  - Exists only as long as it takes to service client

- **Why the distinction?**
  - Allows for concurrent servers that can communicate over many client connections simultaneously
    - E.g., Each time we receive a new request, we fork a child to handle the request

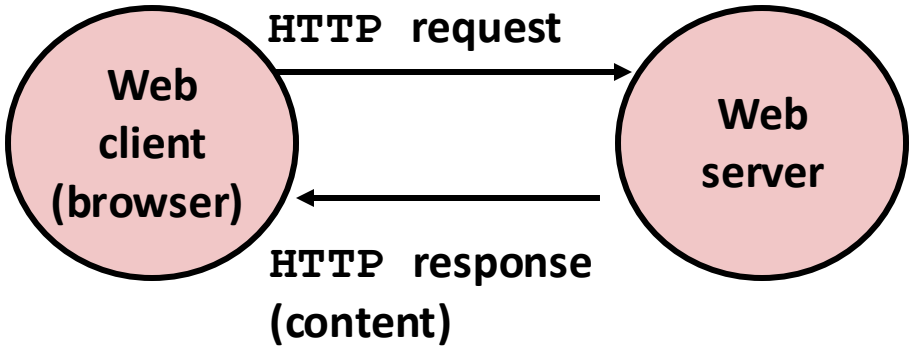

Client
Server
getaddrinfo
getaddrinfo
SA list
SA list
socket
socket
clientfd
listenfd
open_clientfd
open_listenfd
bind
listenfd <-> SA
listen
listening listenfd
Connection request
connect
accept
connected (to SA) clientfd
connected connfd
Client / Server Session
rio_writen
rio_readlineb
rio_readlineb
rio_writen
Await connection request from next client
close
EOF
rio_readlineb
close

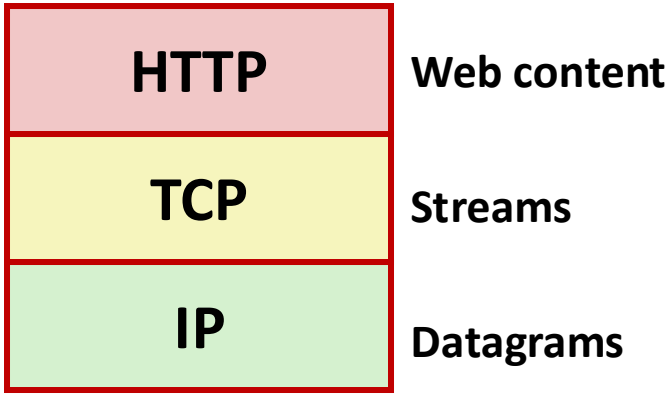
Outline

- Network types and structures
- Locating a host
- Setting up a connection
- HTTP Example

Web Server Basics

- **Clients and servers communicate using the HyperText Transfer Protocol (HTTP)**
  - Client and server establish TCP connection
  - Client requests content
  - Server responds with requested content
  - Client and server close connection (eventually)
- **Current version is HTTP/2.0 but HTTP/1.1 widely used still**
  - RFC 2616, June, 1999.





`http://www.w3.org/Protocols/rfc2616/rfc2616.html`

Web Content

- **Web servers return *content* to clients**
  - *content:* a sequence of bytes with an associated MIME (Multipurpose Internet Mail Extensions) type

- **Example MIME types**

`text/html`	HTML document
`text/plain`	Unformatted text
`image/gif`	Binary image encoded in GIF format
`image/png`	Binary image encoded in PNG format
`image/jpeg`	Binary image encoded in JPEG format

You can find the complete list of MIME types at:
`http://www.iana.org/assignments/media-types/media-types.xhtml`

Static and Dynamic Content

- **The content returned in HTTP responses can be either *static* or *dynamic***
  - *Static content*: content stored in files and retrieved in response to an HTTP request
    - Examples: HTML files, images, audio clips, Javascript programs
    - Request identifies which content file
  - *Dynamic content*: content produced on-the-fly in response to an HTTP request
    - Example: content produced by a program executed by the server on behalf of the client
    - Request identifies file containing executable code
- ***Web content associated with a file that is managed by the server***

URLs and how clients and servers use them

- **Unique name for a file: URL (Universal Resource Locator)**
- **Example URL: `http://www.cmu.edu:80/index.html`**
- **Clients use *prefix* (`http://www.cmu.edu:80`) to infer:**
  - What kind (protocol) of server to contact (HTTP)
  - Where the server is (**`www.cmu.edu`**)
  - What port it is listening on (80)
- **Servers use *suffix* (`/index.html`) to:**
  - Determine if request is for static or dynamic content.
    - No hard and fast rules for this
    - One convention: executables reside in **`cgi-bin`** directory
  - Find file on file system
    - Initial "**/**" in suffix denotes home directory for requested content.
    - Minimal suffix is "**/**", which server expands to configured default filename (usually, **`index.html`**)

HTTP Request Example

```
GET / HTTP/1.1                          Client: request line
Host: www.cmu.edu                       Client: required HTTP/1.1 header
                                        Client: blank line terminates headers
```

- HTTP standard requires that each text line end with "\r\n"
- Blank line ("\r\n") terminates request and response headers

HTTP Requests

- **HTTP request is a *request line*, followed by zero or more *request headers***

- **Request line: `<method> <uri> <version>`**
  - **`<method>`** is one of **`GET`**, **`POST`**, **`OPTIONS`**, **`HEAD`**, **`PUT`**, **`DELETE`**, or **`TRACE`**
  - **`<uri>`** is typically URL for proxies, URL suffix for servers
    - A URL is a type of URI (Uniform Resource Identifier)
    - See http://www.ietf.org/rfc/rfc2396.txt
  - **`<version>`** is HTTP version of request (**`HTTP/1.0`** or **`HTTP/1.1`**)

- **Request headers: `<header name>: <header data>`**
  - Provide additional information to the server

HTTP Responses

- **HTTP response is a *response line* followed by zero or more *response headers*, possibly followed by *content*, with blank line ("\r\n") separating headers from content.**

- **Response line:**

  <version> <status code> <status msg>

  - <version> is HTTP version of the response
  - <status code> is numeric status
  - <status msg> is corresponding English text
    - **200 OK** Request was handled without error
    - **301 Moved** Provide alternate URL
    - **404 Not found** Server couldn't find the file

- **Response headers: <header name>: <header data>**
  - Provide additional information about response
  - **Content-Type:** MIME type of content in response body
  - **Content-Length:** Length of content in response body

Example HTTP Transaction

```
whaleshark> telnet www.cmu.edu 80          Client: open connection to server
Trying 128.2.42.52...                      Telnet prints 3 lines to terminal
Connected to WWW-CMU-PROD-VIP.ANDREW.cmu.edu.
Escape character is '^]'.
GET / HTTP/1.1                             Client: request line
Host: www.cmu.edu                          Client: required HTTP/1.1 header
                                           Client: blank line terminates headers
HTTP/1.1 301 Moved Permanently             Server: response line
Date: Wed, 05 Nov 2014 17:05:11 GMT        Server: followed by 5 response headers
Server: Apache/1.3.42 (Unix)               Server: this is an Apache server
Location: http://www.cmu.edu/index.shtml   Server: page has moved here
Transfer-Encoding: chunked                 Server: response body will be chunked
Content-Type: text/html; charset=...       Server: expect HTML in response body
                                           Server: empty line terminates headers
15c                                        Server: first line in response body
<HTML><HEAD>                               Server: start of HTML content
...
</BODY></HTML>                             Server: end of HTML content
0                                          Server: last line in response body
Connection closed by foreign host.         Server: closes connection
```

- HTTP standard requires that each text line end with "\r\n"
- Blank line ("\r\n") terminates request and response headers

Example HTTP Transaction, Take 2

```
whaleshark> telnet www.cmu.edu 80          Client: open connection to server
Trying 128.2.42.52...                      Telnet prints 3 lines to terminal
Connected to WWW-CMU-PROD-VIP.ANDREW.cmu.edu.
Escape character is '^]'.
GET /index.shtml HTTP/1.1                  Client: request line
Host: www.cmu.edu                          Client: required HTTP/1.1 header
                                           Client: blank line terminates headers
HTTP/1.1 200 OK                            Server: response line
Date: Wed, 05 Nov 2014 17:37:26 GMT        Server: followed by 4 response headers
Server: Apache/1.3.42 (Unix)
Transfer-Encoding: chunked
Content-Type: text/html; charset=...
                                           Server: empty line terminates headers
1000                                       Server: begin response body
<html ..>                                  Server: first line of HTML content
…
</html>
0                                          Server: end response body
Connection closed by foreign host.         Server: close connection
```

Example HTTP(S) Transaction, Take 3

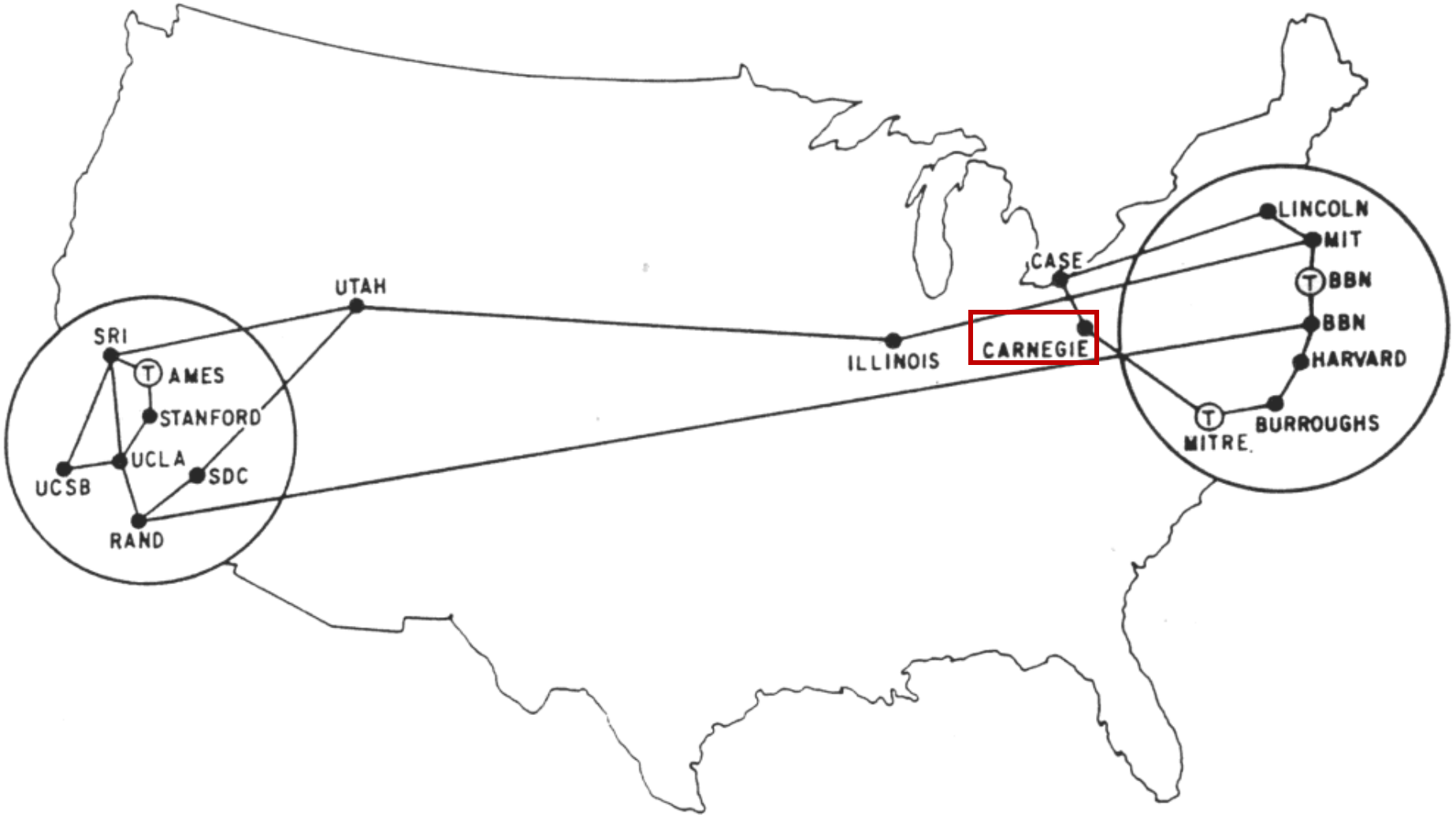
```
whaleshark> openssl s_client www.cs.cmu.edu:443
CONNECTED(00000005)
…
Certificate chain
…
-
Server certificate
-----BEGIN CERTIFICATE-----
MIIGDjCCBPagAwIBAgIRAMiF7LBPDoySilnNoU+mp+gwDQYJKoZIhvcNAQELBQAw
djELMAkGA1UEBhMCVVMxCzAJBgNVBAgTAk1JMRIwEAYDVQQHEwlBbm4gQXJib3Ix
EjAQBgNVBAoTCUludGVybmV0MjERMA8GA1UECxMISW5Db21tb24xHzAdBgNVBAMT
wkWkvDVBBCwKXrShVxQNsj6J
…
-----END CERTIFICATE-----
subject=/C=US/postalCode=15213/ST=PA/L=Pittsburgh/street=5000 Forbes
Ave/O=Carnegie Mellon University/OU=School of Computer
Science/CN=www.cs.cmu.edu          issuer=/C=US/ST=MI/L=Ann
Arbor/O=Internet2/OU=InCommon/CN=InCommon RSA Server CA
SSL handshake has read 6274 bytes and written 483 bytes
…
>GET / HTTP/1.0

HTTP/1.1 200 OK
Date: Tue, 12 Nov 2019 04:22:15 GMT
Server: Apache/2.4.10 (Ubuntu)
Set-Cookie: SHIBLOCATION=scsweb; path=/; domain=.cs.cmu.edu
... HTML Content Continues Below ...
```

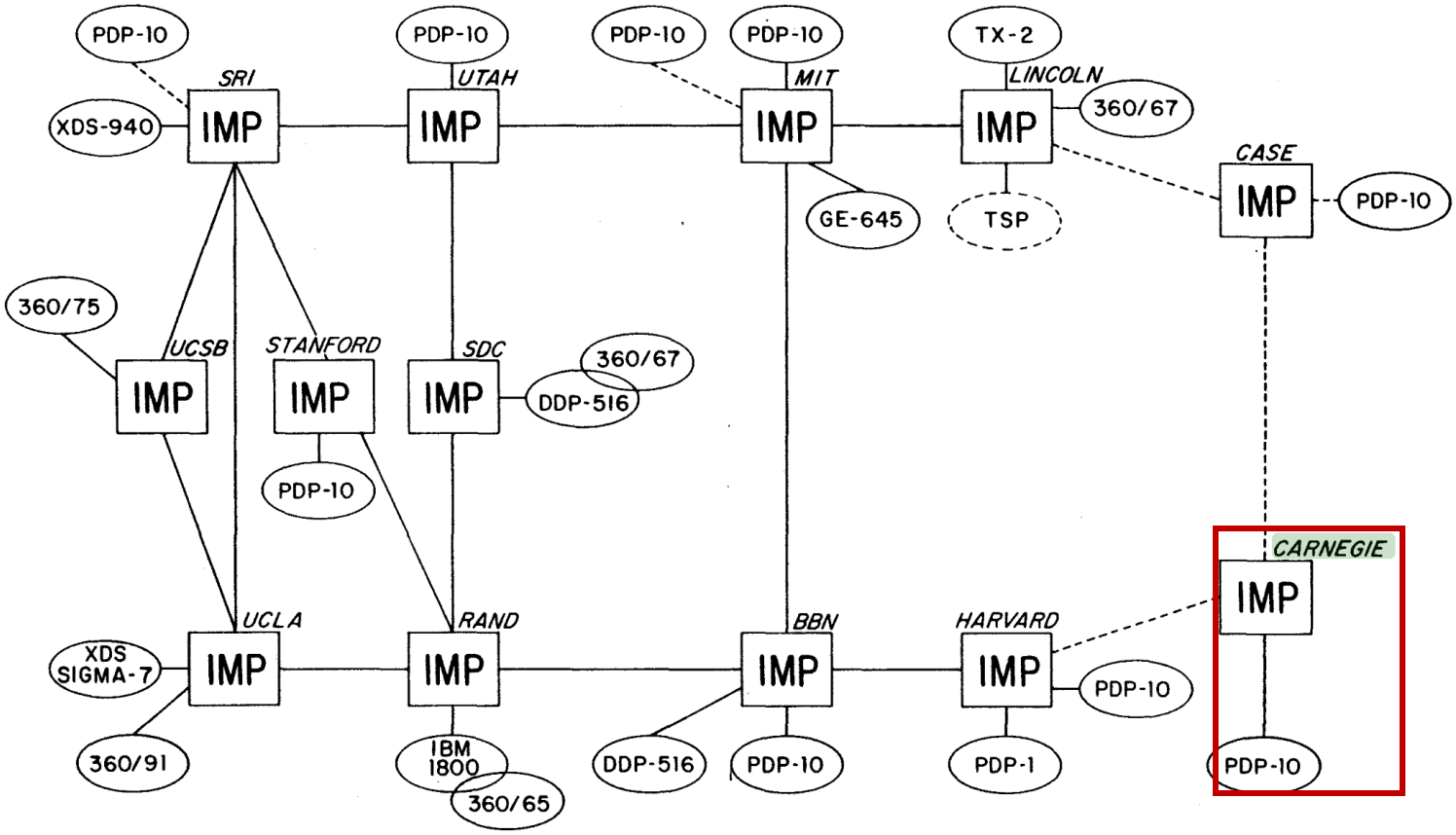
Appendix



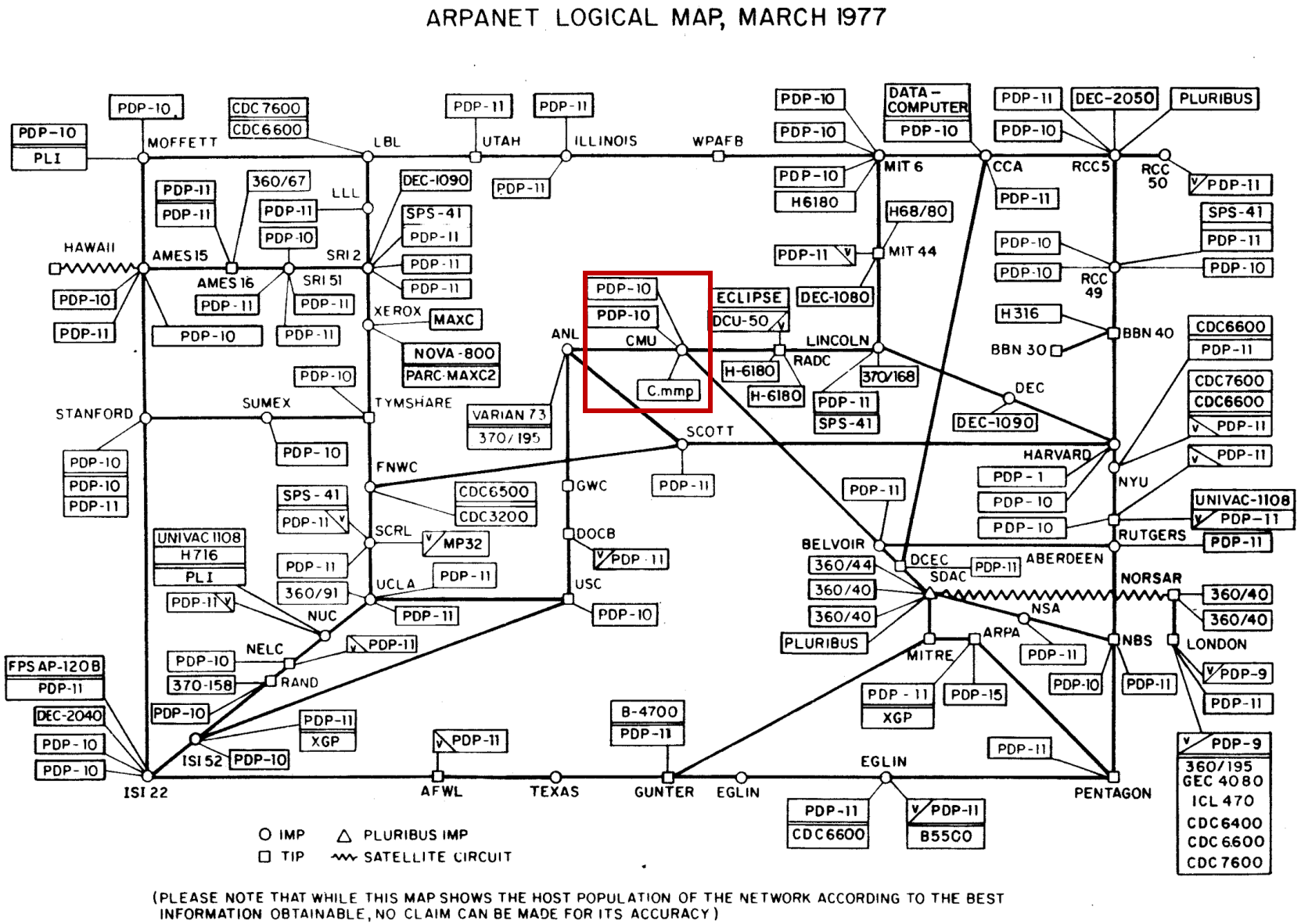
The ARPANET in December 1969

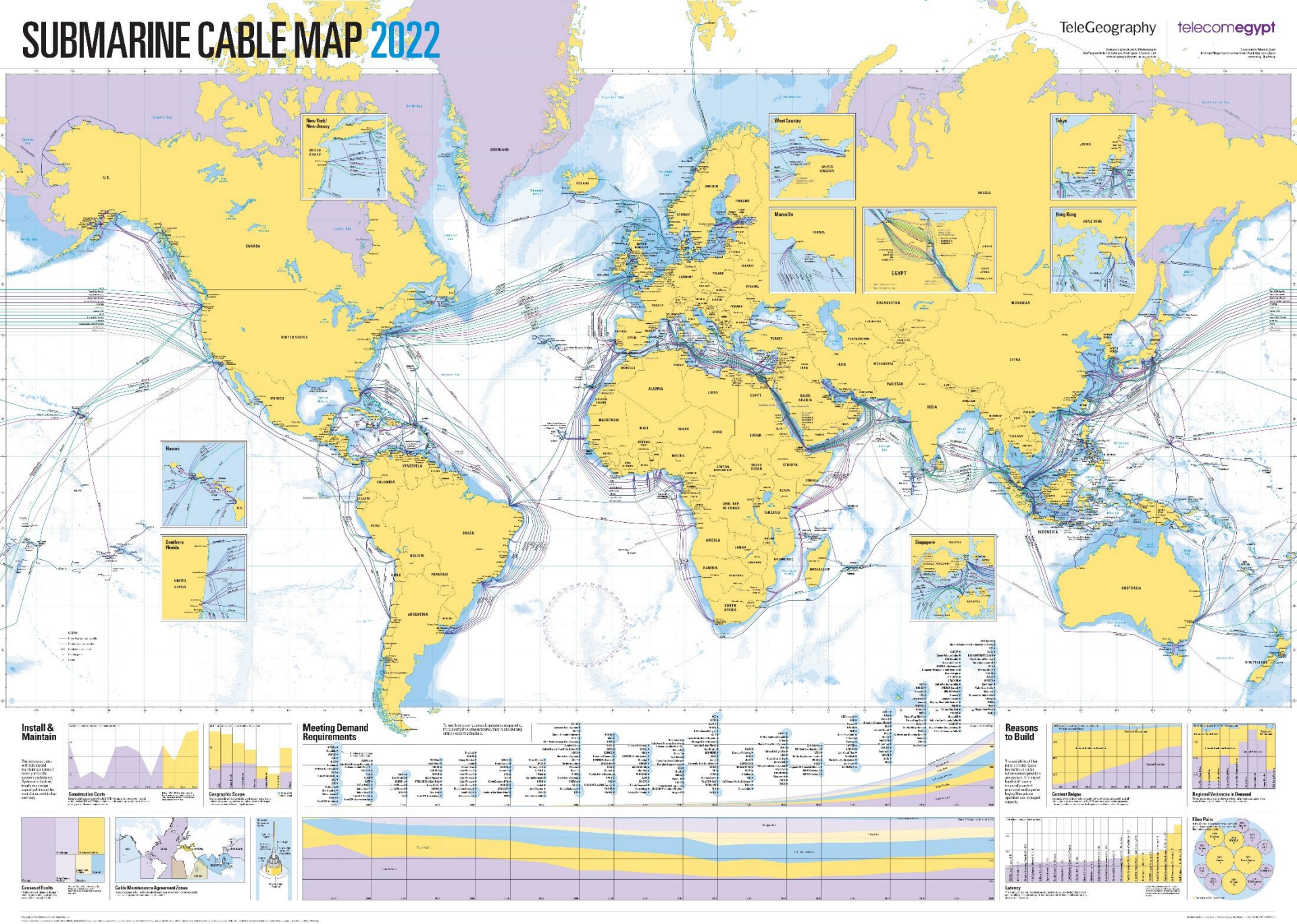


The ARPANET in December 1970

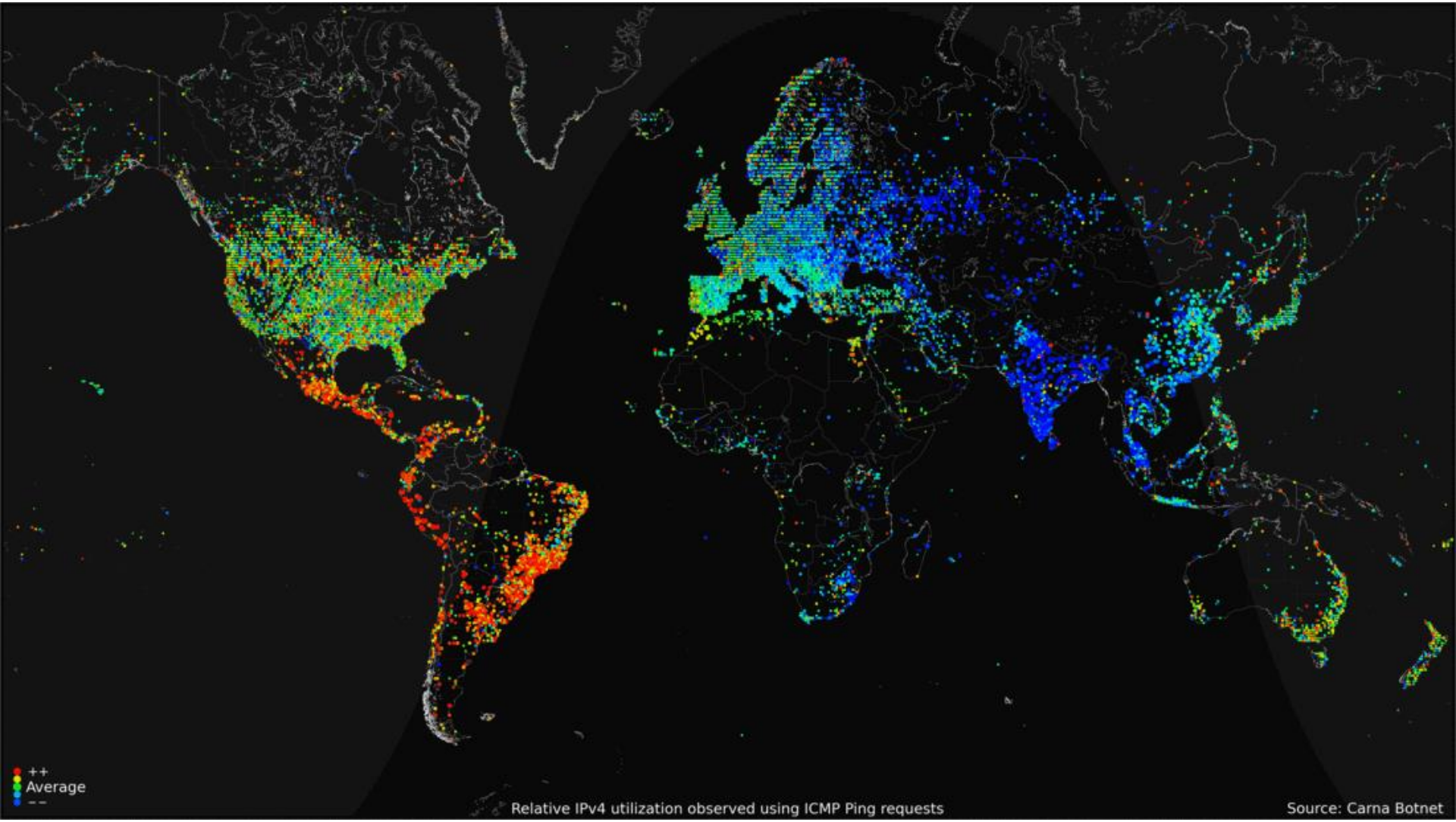


ARPA NET, DECEMBER 1970


ARPANET LOGICAL MAP, MARCH 1977
MOFFETT
LBL
UTAH
ILLINOIS
WPAFB
MIT 6
CCA
RCC5
RCC 50
HAWAII
AMES15
AMES 16
SRI 2
SRI 51
XEROX
MAXC
NOVA-800
PARC-MAXC2
LLL
360/67
DEC-1090
SPS-41
ANL
CMU
C.mmp
ECLIPSE
DCU-50
LINCOLN
RADC
H-6180
370/168
MIT 44
H68/80
H6180
DATA-COMPUTER
DEC-2050
PLURIBUS
RCC 49
H316
BBN 30
BBN 40
CDC6600
CDC7600
DEC
DEC-1090
STANFORD
SUMEX
TYMSHARE
VARIAN 73
370/195
SCOTT
FNWC
GWC
CDC6500
CDC3200
SCRL
MP32
DOCB
UNIVAC 1108
H716
PLI
360/91
UCLA
USC
NUC
NELC
RAND
FPSAP-120B
DEC-2040
ISI 22
ISI 52
AFWL
TEXAS
GUNTER
EGLIN
B-4700
XGP
BELVOIR
360/44
360/40
PLURIBUS
DCEC
SDAC
ABERDEEN
NSA
ARPA
MITRE
PDP-1
PDP-15
HARVARD
NYU
RUTGERS
UNIVAC-1108
NORSAR
NBS
LONDON
360/40
PDP-9
360/195
GEC 4080
ICL 470
CDC6400
CDC 6600
CDC 7600
PENTAGON
B5500
IMP
TIP
PLURIBUS IMP
SATELLITE CIRCUIT
(PLEASE NOTE THAT WHILE THIS MAP SHOWS THE HOST POPULATION OF THE NETWORK ACCORDING TO THE BEST INFORMATION OBTAINABLE, NO CLAIM CAN BE MADE FOR ITS ACCURACY)


SUBMARINE CABLE MAP 2022
TeleGeography
telecomegypt
New York/ New Jersey
West Country
Tokyo
Marseille
Hong Kong
EGYPT
Hawaii
Southern Florida
Singapore
Install & Maintain
Construction Costs
Geographic Scope
Meeting Demand Requirements
Reasons to Build
Content Relays
Regional Variances in Demand
Causes of Faults
Cable Maintenance Agreement Zones
Latency
Fiber Pairs

A Map of 460 Billion Device Connections to the Internet collected by the Carna Botnet



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

